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

HP Part Number 97089-90052

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.

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Printing History

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

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

September 1986...Edition 1

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

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

Table of Contents

Chapter 2: General-Purpose Routines (continued)

Chapter 3: Controlling the HP-IB Interface

Chapter 3: Controlling the HP-IB Interface (continued)

Chapter 4: Controlling the GPIO Interface

Interfacing Concepts 1

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

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

Variation Between Computer Systems

In general, DIL subroutines function identically on all HP-UX computers, whether Integral PC, Series 200/300, 500, or 800. However, because of certain inherent differences between processors and other hardware, some differences do exist. When such differences arise during an explanation, they are clearly identified by introductory headings such as:

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

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

Manual Organization

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

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

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

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

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

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

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

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

Appendix E: Character Codes

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

DIL Interfacing Subroutines

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

Linking DIL Routines

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

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

Similarly, for a Pascal program, use:

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

and for a FORTRAN program, use:

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

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

Calling DIL Routines from Pascal

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

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

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

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

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

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

Calling DIL Routines from FORTRAN

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

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

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

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

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

then write the onionskin routine:

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

General Interface Concepts

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

Definition

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

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

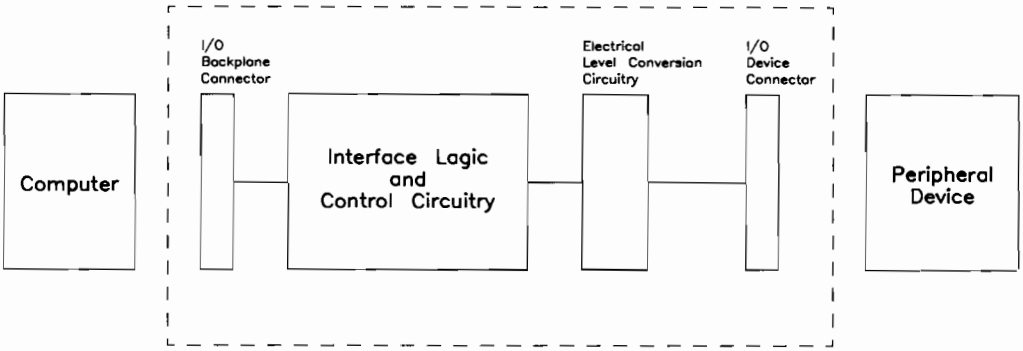


Figure 1–1. Interface Functional Diagram

Interface Functions

A usable interface must fill the following system requirements:

- **Electrical Compatibility:** The interface must convert electrical signal voltages, currents, frequencies, and timing from the computer to a form that is useful to the peripheral device, and vice-versa (unless no conversions are necessary). It must also provide any special protection that might be necessary to protect circuitry within the computer or peripheral from damage due to external effects related to the interface cable or power source.
- **Mechanical Compatibility:** The interface must be mechanically structured so that it is readily connected to both the computer and the peripheral device. This is usually accomplished by means of an interface cable that has appropriate connectors on each end.
- **Data Compatibility.** Just as two people must speak a common language before they can communicate well, the computer and peripheral must use compatible forms of communication. While in most cases, the computer operating system and the programmer are responsible for general data format, communication protocols such as those used in data communication networks and HP-IB interconnections are usually managed by the interface card, based upon various signals and commands from the computer and the peripheral device.

- **Timing Compatibility.** Peripheral devices within a given system rarely have identical data transfer rates and data transfer timing requirements. They also rarely match the timing and transfer rates in the computer or other devices in the system. For this reason, one of the most important functions of the interface is to manage and coordinate the interaction between the computer and the interface as well as timing between the interface and peripheral devices by using special timing signals that are inserted into the data being transferred (most common in data communication interfaces) or carried on separate control signal lines (typical for HP-IB and GPIO interfaces). These timing signals are used to coordinate when a transfer begins and at what rate the information is handled.
- **Processor Overhead Reduction:** Another important function of the interface card is to relieve the computer of low-level tasks, such as performing data transfer handshakes. This distribution of tasks eases some of the computer's burden and decreases the otherwise stringent response-time requirements of external devices. The actual tasks performed by each type of interface card vary widely. The remainder of this chapter concentrates on the functions of two particular interfaces: HP-IB and GPIO.

Handshake I/O

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

Handshake Output

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

Handshake Input

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

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

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

HP-IB Protocol

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

The HP-IB Interface

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

General Structure

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

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

System Controller	Master controller of the HP-IB. The computer interface is usually the bus controller when all peripheral devices on the bus are slaves to the system computer. However, any other device can become the active controller if it is equipped to act as a controller and control is passed to it by the System Controller. The System Controller is always the active bus controller at power-up.
Active Controller	Current controller of the HP-IB. At power-up or whenever IFC (InterFace Clear) is asserted by the System Controller, the System Controller is the active controller. Under certain conditions, the System controller may pass control to another device that is capable of managing the bus in which case that device becomes the new active controller. The active controller can then pass control to another controller or back to the System Controller. If the System Controller asserts IFC, the active controller immediately relinquishes control of the bus.
Talker	A device that has been authorized by the current active controller to place data on the bus. Only one talker can be authorized at a time.

Listener — Any device that has been programmed by the active controller to accept data from the bus. Any number of devices on the bus can be programmed by the active controller to listen simultaneously at any given time.

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

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

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

Handshake Lines

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

$\overline{\text{DAV}}$ — DAta Valid: Valid data has been placed on bus by talker.

$\overline{\text{NRFD}}$ — Not Ready For Data: One or more listeners not yet ready to accept data from the bus.

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

NOTE

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

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

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

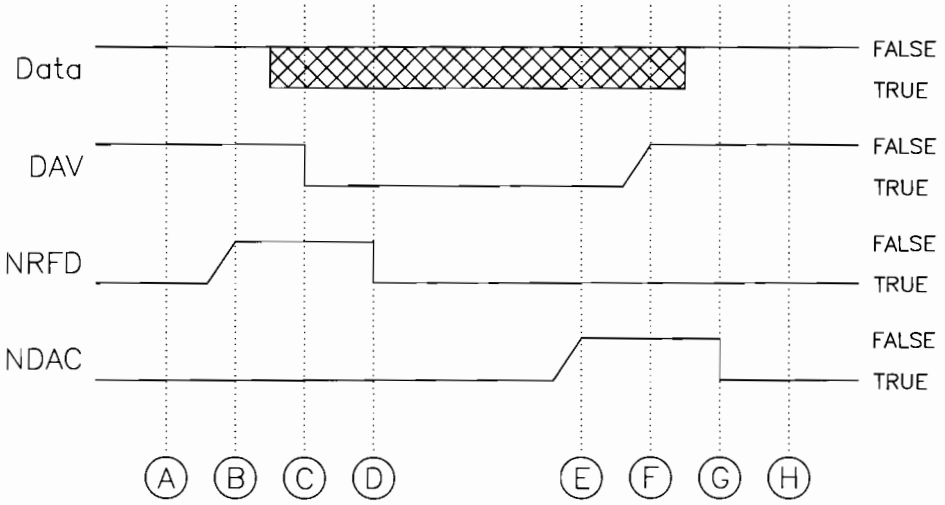


Figure 1−2. The HP-IB Handshake

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

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

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

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

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

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

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

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

Bus Management Control Lines

There are five bus management control lines:

$\overline{\text{ATN}}$	ATtentioN: Treat data on data lines as commands, not data.
$\overline{\text{IFC}}$	InterFace Clear: Unconditionally terminate all current bus activity.
$\overline{\text{REN}}$	Remote ENable: Place all current listeners in Remote operating mode.
$\overline{\text{EOI}}$	End Or Identify: End of data message. If $\overline{\text{ATN}}$ is true (low), Active Controller is conducting a parallel poll (Identify) of devices on the bus.
$\overline{\text{SRQ}}$	Service ReQuest: Bus device is requesting service from current Active Controller.

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

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

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

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

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

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

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

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

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

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

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

The GPIO Interface

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

While the GPIO interfaces used on various HP-UX computers are electrically very similar, they differ in certain important aspects. Refer to the appendices for Series 200/300, 500, 800, or the Integral PC for information pertaining to your specific application.

General-Purpose Routines 2

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

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

Background Basics

Interface Special Files

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

Entity Identifiers (eid)

Nearly all DIL routines require an **entity identifier** (*eid*) as a parameter. The entity identifier is an integer returned by the *open*(2) system call when opening the interface special file (*eid* is the file descriptor for the opened special file on Series 200/300 and 500). The *eid* supplied as a parameter to a DIL subroutine tells the subroutine which interface special file to use.

Programming Model

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

1. Use an *open* system call to obtain the interface entity identifier (*eid*) for the special file being used. Opening an interface special file is discussed later in this chapter.
2. Use the returned *eid* as a parameter in DIL subroutine calls to perform desired tasks through the corresponding interface. Suitable techniques are discussed throughout the remainder of this tutorial.
3. When the necessary DIL subroutine calls have been completed, close the interface special file that was opened in step 1 above as discussed later in this chapter.

General-Purpose Routines

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

Table 2-1. General-Purpose Routines.

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

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

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

Opening Interface Special Files

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

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

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

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

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

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

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

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

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

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

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

Closing Interface Special Files

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

NOTE

HP-UX limits the number of files a given process (program) can have open at one time to `NO_FILE` as defined in the */usr/include/param.h* header file. Series 300 systems limit the number of open DIL files in the entire system to the value of NDIL-BUFFERS (default is 30). On Series 200 systems, the maximum number of open DIL files is limited to 10.

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

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

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

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

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

Low-Level Read/Write Operations

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

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

Calls to *read* have the form:

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

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

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

Calls to *write* are very similar:

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

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

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

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

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

Example

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

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

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

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

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

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

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

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

Designing Error Checking Routines

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

The errno Variable

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

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

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

Using errno

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

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

The errno.h Header File

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

Displaying errno

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

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

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

Error Handlers

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

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

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

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

Resetting Interfaces

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

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

```
io_reset(eid);
```

Io_reset resets the interface whose entity identifier is *eid.* Refer to the appropriate hardware-specific appendix for more information about the exact effects of *io_reset* on HP-IB and GPIO interfaces when used with various computer models.

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

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

Locking an Interface

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

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

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

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

For good programming practice, any locks created by a process should be unlocked through a call to *io_unlock* before terminating. However, any locks held by a process are released when the process terminates, whether or not a call to *io_unlock* was executed. Refer to *io_lock*(2) in the *HP-UX Reference* for more information about locking and unlocking interfaces.

CAUTION

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

Controlling I/O Parameters

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

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

Note

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

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

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

Setting I/O Timeout

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

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

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

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

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

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

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

Note

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

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

IMPORTANT

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

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

Setting Data Path Width

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

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

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

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

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

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

    eid = open("/dev/raw_gpio", O_RDWR);
    io_width_ctl(eid, width); /*assign new width for GPIO bus*/

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

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

Setting Minimum Data Transfer Rate

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

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

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

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

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

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

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

Setting the Read Termination Pattern

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

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

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

Termination on Byte Count

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

Termination on Hardware Condition

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

Termination on Data Pattern

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

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

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

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

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

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

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

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

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

   eid = open("/dev/raw_hpib", O_RDWR);
   io_eol_ctl(eid, ENABLE, '\n');

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

}
```

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

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

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

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

Series 200, 300, and 500 Only: If a single program or process executes more than one *open* system call on the same interface file, each entity identifier returned by *open* can have its own associated read termination pattern. Using *io_eol_ctl* on a given *eid* does not effect the others. Thus, multiple entity identifiers can be set up for a single interface to facilitate recognition of various termination characters during program execution.

Disabling a Read Termination Pattern

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

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

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

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

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

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

Determining Why a Read Terminated

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

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

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

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

Note

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

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

Example

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

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

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

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

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

       bytes_read = read(eid, buffer, 50);
       termination_reason = io_get_term_reason(eid);
       switch (termination_reason) {
             case TR_ABNORMAL:       /* abnormal */
                   printf("Abnormal read termination, bytes_read = %d, errno
= %d\n", bytes_read, errno);
                   break;
             case TR_COUNT:          /* requested count was satisfied */
                   printf("Count satisfied.\n");
                   break;
             case TR_MATCH:          /* specified eol character was matched */
```

```
                printf("EOL character satisfied.\n");
                break;
            case TR_CNT_MCH:        /* TR_COUNT + TR_MATCH */
                printf("Count and EOL character satisfied.\n");
                break;
            case TR_END:            /* EOI was detected */
                printf("EOI detected.\n");
                break;
            case TR_CNT_END:        /* TR_COUNT + TR_END */
                printf("Count satisfied and EOI detected.\n");
                break;
            case TR_MCH_END:        /* TR_MATCH + TR_END */
                printf("EOL character satisfied and EOI detected.\n");
                break;
            case TR_CNT_MCH_END:    /* TR_COUNT + TR_MATCH + TR_END */
                printf("Count and EOL character satisfied and EOI
detected.\n");
                break;
            default:                /* io_get_term_reasoned failed */
                printf("io_get_term_reason failed, bytes_read = %d, errno
= %d\n", bytes_read, errno);
                break;
        }
    }
```

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

Interrupts

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

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

Integral PC Interrupt Support

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

Series 300 and 500 Interrupt Support

HP-IB Interrupts

Series 300 and 500 computers recognize the following HP-IB interrupt conditions:

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

Series 300 GPIO

Series 300 computers recognize the following GPIO interrupt conditions:

EIR	EIR line has been asserted.

Series 500 GPIO

Series 500 computers recognize the following GPIO interrupt conditions:

SIE0	Status line 0 has been asserted.
SIE1	Status line 1 has been asserted.

io_on_interrupt

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

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

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

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

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

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

mask is an integer value that is used to define which parallel-poll response lines are to be recognized in an HP-IB parallel poll interrupt. The value for **mask** is formed from an 8-bit binary number, each bit of which corresponds to one of the eight parallel-poll response lines. For example, to invoke an interrupt handler for a response on line 2 or 6, the correct binary number is 01000100 which converts to a decimal equivalent of 68, the correct value for **mask**.

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

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

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

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

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

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

io_interrupt_ctl

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

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

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

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

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

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

   io_interrupt_ctl(eid,1);

 }
```

Controlling the HP-IB Interface 3

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

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

Overview of HP-IB Commands

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

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

HP-IB Data Bus Commands fall into four categories:

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

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

Table 3.1 HP-IB Bus Commands

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

UNLISTEN

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

UNTALK

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

DEVICE CLEAR

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

LOCAL LOCKOUT

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

SERIAL POLL ENABLE

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

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

SERIAL POLL DISABLE

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

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

TRIGGER (Group Execute Trigger)

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

SELECTED DEVICE CLEAR

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

GO TO LOCAL

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

PARALLEL POLL CONFIGURE

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

PARALLEL POLL ENABLE

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

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

PARALLEL POLL DISABLE

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

Overview of HP-IB DIL Routines

Standard DIL Routines

These 14 subroutines, in addition to the general-purpose subroutines discussed in Chapter 2, provide full capabilities for controlling and using the HP-IB interface.

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

Additional Series 200/300 and Integral PC Routines

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

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

HP-IB: The Computer's Role

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

Ground Rules

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

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

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

Available Subroutines versus Controller Role

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

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

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

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

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

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

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

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

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

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

io_lock and io_unlock

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

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

io_burst

Series 200/300 systems support burst I/O (also called fast handshake) which bypasses the kernel by performing a high-speed non-interrupt transfer. This method can produce considerable performance improvement when handling short transfers to or from high-speed HP-IB devices. See the Series 200/300 Appendix for more information about burst I/O.

hpib_io

The DIL subroutine *hpib_io* is used to perform bus configuration, data transfer, and bus clean-up as a single operation through a locked interface file. When using *hpib_io*, control commands (the preamble), data to be written or a buffer for incoming data (the data message), and clean-up commands (postamble) are placed in a data structure prior to calling *hpib_io*. *hpib_io* then locks the interface, handles the transfer as defined in the data structure (which configures the HP-IB and handles the transfer and clean-up) unlocks the interface, then returns with the result (transfer complete or transfer failed). While *hpib_io* often makes programs shorter and simpler, the added overhead associated with *hpib_io* is less efficient than when using individual DIL subroutine calls.

Opening the HP-IB Interface File

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

Sending HP-IB Commands

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Errors While Sending Commands

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

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

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

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

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

Active Controller Role

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

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

Determining Active Controller

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

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

where *eid* is the entity identifier for the opened HP-IB interface device file and *4* tells the subroutine to examine interface status and determine whether or not the card is the Active Controller. The value returned by the subroutine can be tested as indicated in the example source code which follows.

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

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

   if ((status = hpib_bus_status(eid,4)) == -1)
     :                  /*an error occurred; error-handling code*/
     :                  /*goes here.                              */
   else if (status == 0)
     :                  /*not Active Controller; code to request  */
     :                  /*Active Controller status goes here */
   else
     :                  /*Active Controller; bus-management code */
     :                  /*goes here    */
}
```

Setting Up Talkers and Listeners

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

Series 200/300 and 500 computers provide two methods for addressing listeners and talkers on HP-IB: auto-addressing and command addressing.

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

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

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

Auto-Addressing on Series 200/300 and 500

Much of the tedium of addressing devices to talk or listen can be avoided by using auto-addressed device special files to take advantage of HP-UX auto-addressing capabilities for many peripherals. Auto-addressing is performed only on auto-addressed HP-IB device files. Some DIL subroutines require a **raw** HP-IB device file, and will fail if you attempt to use them on an auto-addressed device file. DIL subroutines that can be used with non-raw device files include *hpib_eoi_ctl*, *hpib_eol_ctl*, *io_burst*, *io_get_term_reason*, *io_lock*, *io_unlock*, *io_speed_ctl*, and *io_timeout_ctl*,

Series 200, 300, and 500 systems determine whether a device file is raw or auto-addressed by the address parameter used when the file is created. Address 31 (hexadecimal 1f) is reserved for raw files. Any address in the range 0 through 30 is auto-addressed. Refer to the appropriate appendix for procedures used to create device and interface special files.

For example, suppose you are using a Series 500 computer with an HP 27110A/B HP-IB card on select code 01 to access a peripheral device located at bus address 03. Use *mknod* to create a new device file named *device* for the peripheral device and place the file in directory *dev* underneath the root directory as explained in Appendix A (a similar procedure described in Appendix B is used for Series 200/300):

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

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

```
crw-rw-rw-   1 root    other      12 0x010300 Nov  22 1986 /dev/device
```

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

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

Using hpib_send_cmnd

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

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

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

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

Calculating Talk and Listen Addresses

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

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

where *eid* is the entity identifier for the interface file and *7* indicates a request for the interface HP-IB bus address.

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

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

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

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

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

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

```
MTA = hpib_bus_status(eid, 7) + 64;
MLA = hpib_bus_status(eid, 7) + 32;
```

An Example Configuration

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

```
#include <fcntl.h>
main()
{
     int  eid, MTA;
     char command[5];
     eid = open("/dev/raw_hpib", O_RDWR);
     MTA = hpib_bus_status(eid, 7) + 64; /*calculate My Talk Address*/
     command[0] = 95;       /* UNTALK command*/
     command[1] = 63;       /* UNLISTEN command*/
     command[2] = MTA;      /* interface talk address*/
     command[3] = 32 + 4;   /* listen address for device at bus address 4*/
     command[4] = 32 + 8;   /* listen address for device at bus address 8*/
     hpib_send_cmnd(eid, command, 5);
 }
```

Remote Control of Devices

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

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

Locking Out Local Control

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

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

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

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

Enabling Local Control

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

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

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

Triggering Devices

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

The following code triggers the device at bus address 5:

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

Transferring Data

Data Output

To output data from an Active Controller the controller must:

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

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

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

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

   eid = open("/dev/raw_hpib", O_RDWR);
   MTA = hpib_bus_status(eid, 7) + 64;  /*calculate MTA*/
   command[0] = 95;                      /*UNTALK command*/
   command[1] = 63;                      /*UNLISTEN command*/
   command[2] = MTA;                     /*address interface to talk*/
   command[3] = 32 + 5;                  /*listen address of device at*/
                                         /*address 5                  */
   hpib_send_cmnd(eid, command, 4);
   write(eid, "data message", 12);   /*send the data*/
   hpib_send_cmnd(eid, command, 2);      /*clear talkers and listeners*/
}
```

Data Input

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

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

   eid = open("/dev/raw_hpib", O_RDWR);
   MLA = hpib_bus_status(eid, 7) + 32;  /*calculate MLA*/
   command[0] = 95;                     /*UNTALK command*/
   command[1] = 63;                     /*UNLISTEN command*/
   command[2] = 64 + 5;                 /*address device at address 5*/
                                        /*to talk                    */
   command[3] = MLA;                    /*address interface to listen*/
   hpib_send_cmnd(eid, command, 4);
   len = read(eid, buffer, 50);         /*store the data in "buffer"*/
   buffer[ len ] = '\0';                /*terminate with NULL for printf*/
   hpib_send_cmnd(eid, command, 2);
   printf("Data read is: %s", buffer);  /*print message*/
}
```

Clearing HP-IB Devices

Two HP-IB commands are used to reset devices to pre-defined, device-dependent states. The first, DEVICE CLEAR, causes all devices that recognize the command to be reset, whether addressed or not.

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

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

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

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

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

Responding to Service Requests

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

Monitoring the SRQ Line

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

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

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

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

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

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

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

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

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

Processing the Service Request

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

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

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

- Serial poll each individual device in sequence until the one that is requesting service is found. This approach is reasonable if there are only a few devices on the bus.

- Conduct a parallel poll to locate the device requesting service. Normally each device (when capable) is programmed to respond on a given data line. However, up to 15 devices can reside on the bus which has only 8 data lines. Therefore it is sometimes necessary for more than one device to respond on a given line.

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

Thus, the Active Controller responds to SRQ by:

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

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

Parallel Polling

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

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

Configuring Parallel Poll Responses

HP-IB devices fall into three general categories:

1. Those devices that can be remotely programmed by the Active Controller to respond to a parallel poll in a certain way, The next several pages explain how to program these devices.

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

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

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

- No two devices can respond on the same data line. This means that only eight or fewer devices in this category can reside simultaneously on a given bus. If fewer than eight are present, data lines not used by these devices for parallel poll response can be shared among remaining devices on the bus if any are present.

- Each device in this category responds to a parallel poll on an assigned data line determined by the device's HP-IB address. Devices residing at HP-IB addresses 0 through 7 respond on data lines DI7 through DI0, respectively (note the reversed numbering sequencing).

- Devices in this category respond to parallel polls when they need service by driving the specified data line LOW to its ground-true logic state (the sense cannot be reversed to high-true).

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

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

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

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

where:

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

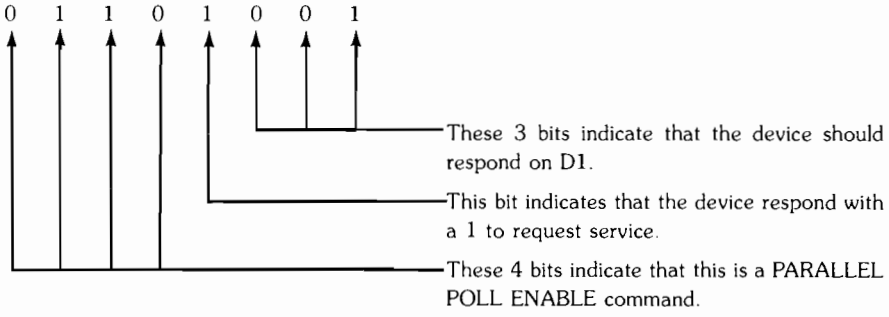
For example, to program a given device to respond to a parallel poll by placing a logic 1 on data line D0 if it needs service, use a PARALLEL POLL ENABLE command with a decimal value of 104 (binary 01101000).

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

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

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

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



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

Disabling Parallel Poll Responses

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

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

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

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

Conducting a Parallel Poll

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

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

hpib_ppoll is invoked as follows:

```
hpib_ppoll(eid);
```

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

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

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

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

Errors During Parallel Polls

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

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

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

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

Waiting For a Parallel Poll Response

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

To call *hpib_wait_on_ppoll*, use the form:

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

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

Calculating the mask

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

The value for *mask* is determined as follows:

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

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

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

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

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

Calculating the sense

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

The value for *sense* is determined as follows:

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

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

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

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

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

Example

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

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

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

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

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

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

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

```
#include <fcntl.h>
main()
{
   int eid;
   eid = open("/dev/raw_hpib", O_RDWR);
   io_timeout_ctl(eid,10000000);   /*Set a 10-second timeout*/

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

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

Serial Polling

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

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

The Active Controller cannot poll itself.

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

Conducting a Serial Poll

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

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

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

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

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

```
#include <fcntl.h>
main()
{
   int eid, status;
   eid = open("/dev/raw_hpib", O_RDWR);
   io_timeout_ctl(eid,100000);   /*Set a 0.1-second timeout*/

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

Errors During Serial Poll

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

- Addressed device did not respond to serial poll before the timeout limit defined by *io_timeout_ctl* was exceeded.
- Computer interface is not current Active Controller.
- Entity identifier *eid* does not refer to an HP-IB raw interface file.
- Entity identifier *eid* does not refer to an open file.
- Address is outside the range [0,30].

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

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

Passing Control

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

hpib_pass_ctl is called as follows:

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

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

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

What If Control Is Not Accepted?

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

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

Errors While Passing Control

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

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

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

***errno* Value**	**Error Condition**
EBADF	*eid* does not refer to an open file.
ENOTTY	*eid* does not refer to a raw interface file.
EIO	Interface is not Active Controller.
EINVAL	Invalid bus address.

System Controller Role

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

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

Determining System Controller

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

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

where *eid* is the entity identifier for an open interface file and *3* indicates that you want to determine whether it is the System Controller. The subroutine returns 1 if it is the System Controller, 0 if not, and −1 if an error occurs.

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

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

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

System Controller's Duties

The HP-IB System Controller has three major functions:

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

hpib_abort

A call to *hpib_abort* performs the following actions:

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

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

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

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

hpib_abort is called as follows:

```
hpib_abort(eid);
```

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

hpib_ren_ctl

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

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

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

hpib_ren_ctl is called as follows:

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

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

Errors During hpib_abort and hpib_ren_ctl

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

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

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

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

The Computer As a Non-Active Controller

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

Checking Controller Status

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

The call to *hpib_bus_status* has the form:

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

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

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

If the value of *status_question* is in the range 0-6, *hpib_bus_status* returns **1** if the answer to the question is yes, or **0** if the answer is no. If the value of *status_question* is 7, *hpib_bus_status* returns the bus address of the computer's HP-IB interface. If the value of *status_question* is outside the allowable range of 0 through 7, **−1** is returned, indicating an error.

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

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

Requesting Service

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

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

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

To request service, invoke *hpib_rqst_srvce* as follows:

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

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

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

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

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

A couple of notes of caution are in order here:

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

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

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

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

Errors While Requesting Service

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

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

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

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

Responding to Parallel Polls

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

To configure a parallel-poll response:

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

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

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

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

Calculating the Response

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

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

where:

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

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

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

Limitations of hpib_card_ppoll_resp

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

Error Conditions

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

- Entity identifier *eid* does not refer to an HP-IB raw interface file.
- Entity identifier *eid* does not refer to an open file.
- **Series 500 Only:** Interface parallel poll response cannot be altered under local program control.

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

errno Value	Error Condition
EBADF	*eid* does not refer to an open file.
ENOTTY	*eid* does not refer to a raw interface file.
EINVAL	(**Series 500 Only:**) Interface cannot respond on the line indicated by *response*

hpib_ppoll_resp_ctl

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

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

Parallel poll response is set as follows:

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

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

Disabling Parallel-Poll Response

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

For example, the subroutine call:

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

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

Accepting Active Control

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

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

Call *hpib_status_wait* as follows:

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

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

Value	Wait Condition
1	Wait until the SRQ line is asserted
4	Wait until this computer is the Active Controller
5	Wait until this computer is addressed as a talker
6	Wait until this computer is addressed as a listener

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

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

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

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

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

Errors While Waiting on Status

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

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

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

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

Determining When You Are Addressed

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

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

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

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

In the above call to *hpib_bus_status*, *eid* is the entity identifier for the interface device file and *5* indicates that you want to know if it is addressed to talk. The routine returns the value **1** if the answer is yes; **0** if not.

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

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

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

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

As with *hpib_bus_status*, a condition value of 5 causes the program to wait until the interface is addressed as a talker. With a condition value of 6 the routine waits until it is addressed to listen. How maximum time that the routine can wait for the specified condition is controlled by the timeout value that was previously set for the entity identifier using subroutine *io_timeout_ctl* (discussed in Chapter 2). *hpib_status_wait* returns **0** if the wait condition terminated the wait or **−1** if a timeout or other error occuered before the wait condition was fulfilled.

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

```
#include <fcntl.h>
main()
{
   int eid, len;
   char buffer[51];                    /*storage for message*/
   eid = open("/dev/raw_hpib", O_RDWR);
   io_timeout_ctl(eid, 5000000);       /*5-second timeout*/

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

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

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

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

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

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

Combining I/O Operations into a Single Subroutine Call

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

A call to *hpib_io* has the form:

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

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

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

Iodetail: The I/O Operation Template

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

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

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

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

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

```
iovec->component
```

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

The Mode Component

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

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

Note

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

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

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

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

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

The Terminator Component

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

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

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

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

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

The Count Component

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

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

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

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

The Buf Component

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

Note

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

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

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

Allocating Space

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

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

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

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

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

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

Example

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

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

The following code segment illustrates how the 4 structures can be built and implemented.

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

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


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


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

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

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


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


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


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

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


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

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

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

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

Locating Errors in Buffered I/O Operations

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

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

For example, suppose an array of 10 *iodetail* structures has been built to execute a sequence of I/O operations. The following code segment executes the operations then checks for errors. If an error occurs, the number of the structure that caused it (the first structure in the array is number 1) is printed.

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

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

Controlling the GPIO Interface 4

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

Configuring the GPIO Interface

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

Configuring the Integral PC GPIO

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

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

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

Setting Interface Switches

Series 200/300 and 500 computer GPIO interface cards have several configuration switches that are used to set up the interface. The interface installation manual explains how each switch is used and how it should be configured. Configurable functions associated with these switches include:

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

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

Note

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

Creating the GPIO Interface File

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

Interface Control Limitations

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

- There is no direct access to interface handshake lines: Peripheral Control (PCTL) line, Peripheral Flag (PFLG) line, and Input/Output (I/$\overline{O}$) line.
- You cannot read the value of the Peripheral Status line (PSTS) directly.
- **Series 500 Only**: You cannot recognize interrupts sent by the peripheral over the External Interrupt Request line (EIR).

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

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

Using DIL Subroutines

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

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

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

Two DIL routines used exclusively with GPIO interfaces:

- *gpio_get_status*
- *gpio_set_ctl.*

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

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

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

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

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

Resetting the Interface

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

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

This has the following effect:

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

Lines that are left unchanged are:

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

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

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

Performing Data Transfers

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

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

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

Using Status and Control Lines

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

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

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

The call to *gpio_set_ctl* has the following form:

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

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

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

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

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

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

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

Reading STI0 and STI1

The call to *gpio_get_status* has the following form:

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

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

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

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

To illustrate:

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

Controlling Data Path Width

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

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

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

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

```
int eid;

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

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

Controlling Transfer Speed

You can request a minimum speed for the data transfer across a GPIO interface by issuing a call to *io_speed_ctl*. Your system rounds the specified speed up to the nearest defined speed. If you specify a speed that is faster than your system allows, the highest available speed is used instead. Refer to Chapter 2 for more information about *io_speed_ctl*. Series 500 systems always use DMA, so use of this subroutine on Series 500 is meaningless, although it is supported for software compatibility reasons.

GPIO Timeouts

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

Burst Transfers

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

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

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

Read Terminations

Determining Why a Read Operation Terminated

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

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

Specifying a Read Termination Pattern

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

Interrupts

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

Interrupt-Driven Transfer Mode

Implemented on Integral PC Only:

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

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

Series 500 Dependencies A

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

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

Device I/O Library Location

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

The GPIO Interface

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

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

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

Data Lines

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

Handshake Lines

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

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

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

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

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

Special-Purpose (Control and Status) Lines

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

Data Handshake Methods

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

Latching Data Transfers

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

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

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

Creating the Interface Special File

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

Creating an Interface File

Special files are created using the *mknod(*1M) command which requires super-user access. When creating an interface special file, *mknod* has the following syntax:

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

The `c` parameter to *mknod* tells the system to create the file as a character special file. The remaining parameters in the *mknod* command are as follows:

pathname

The *pathname* parameter specifies the name being given to the new interface special file. **pathname** identifies the interface file itself, not a peripheral connected to the interface. Special files are usually kept in the directory */dev*. This HP-UX convention is used because some commands expect to find device special files in the */dev* directory and fail if the file is not there.

major_number

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

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

minor_number

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

```
0xScAdUV
```

where:

`0x` Identifies the remainder of the expression as a hexadecimal number. The two characters (zero followed by x) are entered exactly as shown.

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

`Ad` A two-digit hexadecimal value specifying the device bus address. To use DIL subroutines with the interface, the special file should be created as a **raw** special file: the `Ad` component of the minor number should be 31 (1f in hexadecimal). If `Ad` is less than 31, the file is *not* created as a raw file but rather as an auto-addressable file (in which case, `Ad` specifies the bus address of the device for which the special file is created). If only one device can be connected to the interface (as when using the GPIO interface), this component of the minor number is ignored (use `00` instead of a device bus address).

`U` A single-digit hexadecimal value specifying a secondary address such as a device unit number. This component of the minor number is not used when creating interface special files; set it to 0.

`V` A single-digit hexadecimal value specifying a secondary address such as a device volume number. This component of the minor number is not used when creating interface special files; set it to 0.

Creating an HP-IB Interface File

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

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

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

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

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

```
mknod /dev/raw_hpib1 c 12 0x021f00
mknod /dev/raw_hpib2 c 12 0x031f00
```

Creating a GPIO Interface File

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

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

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

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

Determining Interface Card Bus Address

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

Effects of Resetting (via io_reset)

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

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

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

Entity Identifiers

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

DIL Subroutine Use Restrictions

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

hpib_bus_status

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

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

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

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

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

hpib_card_ppoll_resp

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

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

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

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

hpib_rqst_srvce

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

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

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

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

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

clears it.

hpib_send_cmnd

HP 27110A/B HP-IB and Model 550 Internal HP-IB interfaces send all *hpib_send_cmnd* commands with odd parity by overwriting the most significant bit of each command byte with a parity bit. This should not cause a problem since all HP-IB commands use only 7 bits, and the eighth is free for use as parity.

hpib_status_wait

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

hpib_wait_on_ppoll

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

io_get_term_reason

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

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

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

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

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

io_on_interrupt

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

io_timeout_ctl

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

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

io_speed_ctl

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

io_width_ctl

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

Performance Tips

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

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

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

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

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

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

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

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

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

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

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

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

- Use auto-addressing for all read and write operations (see the Chapter 3 section "Setting up Talkers and Listeners" for details).
- *rtprio*(2) can be used to increase the system priority of an I/O process. *rtprio* requires that the process be running with an effective user *ID* of super-user. The real-time priorities available with *rtprio* are non-degrading priorities. However, caution must be observed when using real time priorities because one can increase their priority above system processes resulting in possibly undesirable behavior.

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

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

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

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

Series 200/300 Dependencies B

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

- Location of the DIL subroutines,
- Information about creating interface special files used by DIL subroutines,
- Relationship between entity identifiers and file descriptors,
- Restrictions imposed by the hardware on using the DIL subroutines,
- Techniques for improving data transfer performance when using DIL subroutines.
- Information on how to simulate I/O interrupt programming on Series 200/300 computers.

Location of the DIL Subroutines

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

Linking DIL Subroutines

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

The GPIO Interface

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

The GPIO interface is comprised of the following lines:

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

Data Lines

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

Handshake Lines

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

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

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

Special-Purpose Lines

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

Data Handshake Methods

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

Data-In Clock Source

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

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

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

Creating the Interface Special File

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

Creating the Special File

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

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

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

pathname

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

major_number

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

Major Number	Interface
21	HP-IB Interface
22	GPIO Interface

minor_number

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

```
0xScAdUV
```

where:

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

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

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

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

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

Creating an HP-IB Interface File

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

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

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

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

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

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

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

Creating a GPIO Interface File

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

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

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

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

Effects of Resetting (via io_reset)

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

Entity Identifiers

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

Restrictions Using the DIL Subroutines

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

hpib_io

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

hpib_send_cmnd

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

hpib_status

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

io_interrupt_ctl

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

io_on_interrupt

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

io_reset

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

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

io_speed_ctl

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

IMPORTANT

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

---

io_timeout_ctl

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

Performance Tips

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

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

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

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

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

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

Simulating Interrupts for the HP-IB Interface

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

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

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

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

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

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

Some additional points about simulating interrupts in this way are:

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

Simulating Interrupts on the GPIO Interface

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

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

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

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

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

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

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

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

   }
}
```

Integral PC Dependencies C

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

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

Location of the DIL Routines

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

The GPIO Interface

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

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

The HP 82923A GPIO interface has six handshake types. The handshake type is selected using the *gpio_handshake_ctl* routine.

Creating an Interface Special File

Two utility programs, *load_hpib* and *load_gpio*, must be used to create the appropriate special files for your HP-IB and GPIO interfaces, respectively. These routines create a special (device) file for each HP-IB or GPIO interface found, and load the appropriate DIL driver. The data files containing the DIL drivers, *dhpib.data* and *dgpio.data*, must be in the search path defined by your PATH variable when the load utility is invoked. For more information on *load_hpib* and *load_gpio* refer to the *load_hpib.1* and *load_gpio.1* files provided in the *doc* folder on the DIL disc.

GPIO Interface Files

The special files for GPIO interfaces have the following form:

`/dev/gpio`*GPIO_port.IO_port*

where *GPIO_port* is the letter designation for GPIO ports `a`, `b`, `c`, or `d`; and *IO_port* is a one- or two-character designation (`a`, `b`, `a1`, `a2`,...) for the Integral PC I/O port. Note that the top port on the Integral PC is port `a`, the bottom port is port `b`, while the bus expander ports have a combination letter and number designation as shown below.

HP-IB Interface Files

The special (device) files for HP-IB interfaces have two forms:

`/dev/dhpib.i`	for the built-in HP-IB interface
`/dev/dhpib.`*IO_port*	for the plug-in HP-IB interface, where *IO_port* is the Integral PC I/O port designator (`a`, `b`, `a1`, `a2`,...) described above.

Unloading the DIL Drivers

Two additional utilities, *unload_hpib* and *unload_gpio*, are provided on the DIL disc. These utilities are used to remove both the DIL drivers and the special files created by *load_hpib* and *load_gpio*. For more information on using these utility programs, refer to *load_hpib.1* and *load_gpio.1* in the *doc* folder on the DIL disc.

Interrupts

Unlike the Series 500, the Integral PC supports only one interrupt condition, PIR, meaning that the Peripheral Interface Request line has been asserted. For hardware restrictions on using the HP-IB interrupts on the Integral PC, refer to the *io_on_interrupt.3d* file in the *doc* folder on the DIL disc.

Controlling the HP-IB Interface

Limitations on the HP-IB Interface

The use of DIL routines with the built-in HP-IB interface has the following limitations:

- The user must not pass control when using the DIL routines with the built-in HP-IB interface. The built-in HP-IB interface *must* always be the System Controller/Active Controller.
- Loading the DIL drivers and then opening the *built-in* HP-IB interface special file prevents the operating system from accessing printers, plotters, and mass-storage drives on the built-in HP-IB interface until the built-in HP-IB interface special file is closed. This means that any operation using a printer, plotter, or mass-storage device on the built-in HP-IB interface will be suspended until the built-in HP-IB device file is closed. This limitation can result in a deadlock situation if your program both uses the DIL routines with the built-in HP-IB interface and attempts to use a printer, plotter, or mass-storage drive on the built-in HP-IB interface.

To avoid these limitations, we recommend that you **use the HP-IB DIL routines only with the HP 82998A HP-IB interface**.

The Computer as a Non-Active Controller

The built-in HP-IB interface must be in the system controller, active controller state to use the DIL routines on the Integral PC.

Non-Standard DIL Routines

The Integral PC DIL library supports several routines that are not part of the DIL standard. This section describes these routines.

General-Purpose Routines

In addition to the standard DIL routines, the Integral PC DIL library supports the following two routines:

io_lock	Locks the interface port to the calling process until the *io_unlock* routine is called.
io_unlock	Used by the calling process to remove the lock created by *io_lock*.

For details on using these routines, refer to the *io_lock.3d* file located in the *doc* folder on the DIL disc supplied with your Integral PC.

Non-Standard HP-IB Routines

In addition to the standard DIL routines for controlling the HP-IB interface, the Integral PC supports the following non-standard DIL routine:

io_burst(eid, flag)	Used to control the high-speed HP-IB mode. If *flag* = 0, high-speed mode is turned off; otherwise it is turned on.

For information on the *io_burst* routine, refer to the *io_burst.3d* file in the *doc* folder on the DIL disc.

Non-Standard GPIO Routines

The following non-standard DIL routines have been added to control the HP 82923A GPIO interface:

- *gpio_handshake_ctl*
- *gpio_normalize_ctl*
- *gpio_delay_time_ctl*

A description of these routines is provided in the *doc* folder on the DIL disc.

Restrictions Using the DIL Routines

This section presents some restrictions on using DIL routines with the Integral PC computer. Restrictions on using system routines, such as *open(2)*, are also discussed here. These restrictions are organized under the routine to which they apply; the routines are presented in alphabetical order.

hpib_bus_status

On the Integral PC, it is not possible to determine the status of the NDAC and SRQ lines under certain conditions. This can result in incorrect results when using the *hpib_bus_status* routine to determine the status of these two lines. If the HP-IB interface is talk-addressed, the SRQ status is incorrect; if it is listen-addressed, the NDAC status is incorrect.

hpib_card_ppoll_resp

The parallel poll response of the HP 82998A HP-IB interface can *not* be remotely programmed. Instead, use the *hpib_card_ppoll_resp* routine.

hpib_ppoll_resp_ctl

The "sense" bit of the flag value for the *hpib_ppoll_resp_ctl* routine determines whether a zero or non-zero "response value" means that the computer requires service. If the "s" bit is a 0, then a zero response value means service is needed.

io_eol_ctl

On the Integral PC, a read operation from a GPIO interface will terminate only when a specified number of read operations have been performed, or when the read termination pattern has been found.

The Integral PC does not support different read termination patterns on multiple opens to the same *eid*.

io_reset

When used to reset a GPIO interface, the *io_reset* routine will pulse the $\overline{\text{RES}}$ (reset) line only on the GPIO controller port specified by the *eid*.

io_speed_ctl

GPIO

Setting the speed on a GPIO interface determines the transfer mode used by the driver: either interrupt-driven, flag-driven handshake, or "fast handshake" mode. (Note that the driver's fast handshake mode is not the same as the fast handshake mode described in the *HP 82923A GPIO Owner's Manual*; it refers to a flag-driven mode where the EOL and timeout settings are ignored to achieve a faster transfer rate.)

DMA transfers are not available on the Integral PC.

Interrupt-Driven Transfer Mode

Two transfer modes exist between the Integral PC and the HP 82923A GPIO interface: flag-driven mode and interrupt-driven mode. To select the interrupt-driven mode, use *io_speed_ctl* to set the speed to 0.

While in the interrupt-driven mode, *read* and *write* calls to the GPIO interface will cause the user's process to go to sleep until an interrupt occurs at the completion of the *read* or *write*.

HP-IB

The DIL routines on the Integral PC support two HP-IB transfer modes: flag-driven mode and high-speed transfer mode. The default mode is the flag-driven mode until it is set to the high-speed transfer mode using the *io_burst* routine.

In the high-speed transfer mode, the driver talks directly to the interface without going through the operating system. For more information on *io_burst*, refer to the documentation provided in the *io_burst.3d* file in the *doc* folder on the DIL disc.

io_timeout_ctl

This routine allows you to set a time limit for operations carried out by DIL routines on a specified entity identifier. The timeout value you specify is a 32-bit long integer that indicates the length of the timeout in microseconds (μ-secs). However, the resolution of the effective timeout is system-dependent. On the Integral PC, the timeout resolution on both the HP 82923A GPIO interface and the HP 82998A HP-IB interface is 1 millisecond (msec).

For example, suppose you specify a timeout of 99 999 microseconds (99.999 milliseconds). Then the effective timeout is rounded up to 100 milliseconds.

io_width_ctl

The data path width for the HP-IB interface is always 8 bits on the Integral PC. However, the four 8-bit ports on the HP 82923A GPIO interface can be combined to form 8-, 16-, or 32-bit data paths.

For 16- or 32-bit ports, only one port acts as a controller; that port's *eid* is used in the *io_width_ctl* routine. The allowable data path widths for each port are shown in the following table.

GPIO Data Path Widths

Data Path Width	Controller Port	Data Ports*
8-bit	a	a
	b	b
	c	c
	d	d
16-bit	b	b a
	d	d c
32-bit	b	b a d c

* Data ports are listed in order, left to right, from most-significant byte to least-significant byte.

Combinations of 8- and 16-bit or two 16-bit ports are also allowed on the same GPIO interface. 24-bit ports are not allowed.

open(2)

When opening the special file for an interface, you must use the special file name for the specific GPIO or HP-IB interface created by *load_hpib* or *load_gpio*. Note that each GPIO port has a separate special file name. For details on interface special file names, see the previous section "Creating an Interface Special File."

read(2) and write(2)

During a read or write operation to a 16- or 32-bit **GPIO** port, the data must start on a word boundary. This restriction applies only to the GPIO interface.

Series 800 Dependencies D

The following information, specific to the Device I/O Library (DIL) on Series 800 computers, is discussed in this appendix:

- compiling programs that use DIL routines
- accessing the special files for the interfaces that you plan to use with DIL
- creating special files for the interfaces that you plan to use with DIL
- DIL routines affected by the Series 800 hardware
- DIL support of HP-IB auto-addressed files
- improving performance of DIL programs

Compiling Programs That Use DIL

The DIL routines are located in the library */usr/lib/libdvio.a*. Thus, programs can be linked as:

```
cc test.c -ldvio
```

Accessing the Interface Special Files

The Series 800 kernel is shipped with a default I/O configuration. This means a default set of special files is made for you. For example, the */dev/hpib* directory contains special files created for use with HP-IB instruments connected to the HP 27110B HP-IB interface. The special file */dev/gpio0* is created for use with instruments or peripherals connected to the HP27114A Asynchronous FIFO interface (AFI). The *insf* command is used to install these special files all at one time. *Mknod* could also be used to create them one at a time. For more information on *insf* and *mknod* refer to the *HP-UX Reference.*

Major Numbers

Major numbers map the hardware I/O cards to the software I/O driver for the type of I/O application the card will be doing. The driver used to talk to the HP-IB card for instrument I/O is called *instr0*, and corresponds to major number 21. The HP-IB card talks to different drivers (which use different major numbers) to do I/O to other kinds of devices, such as disc drives or printers. All default special files in the */dev/hpib* directory use major number 21. The driver that talks to the AFI card is called *gpio0*, and corresponds to major number 22. The */dev/gpio0* special file uses major number 22.

Minor Numbers and Logical Unit Numbers

Drivers use minor numbers to map the hardware I/O cards to their locations in the Model 840 I/O backplane. The default I/O configuration shipped with your Model 840 creates special files accessing a subset of the available backplane slots. For the HP-IB card, two slots are available for instrument I/O, and one slot is available for the AFI card. Slot information is accessed through the device's *logical unit* number. The logical unit number is mapped into the special file's minor number. For HP-IB special files, the HP-IB bus address is also mapped into the minor number.

The minor number syntax for an HP-IB special file is:

```
0x00LuBa
```

where **Lu** is the device's logical unit number, and **Ba** is the bus address of the HP-IB device. Both numbers are in hexadecimal.

The minor number syntax for an AFI special file is:

```
0x00Lu00
```

where **Lu** is the device's logical unit number in hexadecimal.

For example, a long listing of the special file */dev/hpib/0a16* shows

```
$ ll /dev/hpib/0a16
crw-rw-rw-   1 root     root     21 0x000010 Mar 11 15:19 0a16
```

The logical unit number is 0, and bus address 16 is 10 in hexadecimal.

Listing Special Files

The Series 800 I/O architecture is based on a hierarchical design. The use of logical numbers in conjunction with the major and minor number allows the system to keep track of all the information about the I/O structure. The *lssf* command, list special file, is a tool that makes it easy to read information about a special file without decoding it by hand.

The syntax of *lssf* is:

```
lssf [-f dev_file] path
```

where **path** is the pathname of the special file. *Lssf* uses the major number from the special file to find the name of the device driver in a file called */etc/devices*. If you use the **-f** option, *lssf* looks in **dev_file** instead of */etc/devices*. It then decodes the minor number, outputs the logical unit number, the device bus address (if there is one), and the corresponding CIO slot address for the actual card in the I/O backplane.

Using the default special file */dev/hpib/0a16* as an example, the following output is produced:

```
$ lssf /dev/hpib/0a16
instr0 lu 0 bus address 16 address 8.2.16 /dev/hpib/0a16
```

where *instr0* is the name of the instrument HP-IB driver, the logical unit number is *0*, the HP-IB bus address is *16*, and the backplane address of the HP-IB card is *8.2.16*. This says that the CIO channel card is in mid-bus address *8*, and the HP-IB card should be in slot *2* of that CIO channel. There are 12 CIO slots available, numbered 0-11. The last digit, in this case *16*, is the HP-IB bus address of the device *0a16*.

The default HP-IB special files are set up for cards in slot 2 or slot 7 of the CIO channel at mid-bus address 8. A special file for each possible bus address (0-31) is made for each card. The special files for the card at slot 2 all have a logical unit number of 0, and the special files for the card in slot 7 all have a logical unit number of 1.

The default GPIO special file is set up for an AFI card in slot 5 of the CIO channel at mid-bus address 8, and uses a logical unit number of 0.

For more information on *lssf* refer to the *HP-UX Reference*.

Naming Conventions for Interface Special Files

If your Series 800 computer was configured correctly, the special files discussed above will already have been created.

By convention, HP-IB special files reside in the */dev/hpib* directory. Also by convention, the default special files for the HP-IB *raw bus* (a HP-IB card itself) are named */dev/hpib/***X**, where **X** is the bus's logical unit. *Auto-addressed* files are named */dev/hpib/***X***a***Y**, where **X** is the logical unit, *a* stands for an auto-addressed file, and **Y** is the file's associated HP-IB bus address (see the "DIL Support of HP-IB Auto-Addressed Files" section of this appendix).

The naming convention for the GPIO default special files is */dev/gpio***X**, where **X** is the device's logical unit.

If you cannot locate the default special files on your system, refer to the next section for how to create them.

Creating Interface Special Files

If the special files you need for HP-IB or GPIO are not available on your system, you can use the *mksf* command to create them. *Mksf* is a high-level command implemented for the Series 800, that can be used instead of *mknod*. Like *lssf*, *mksf* frees you from having to know the major number and minor number format. *Mksf* makes the special file creation process consistent for all classes of devices. The syntax of *mksf* is:

```
mksf -d driver -l lu other_flags... sfname
```

where **driver** is the name of the driver associated with the special file, **lu** is the file's logical unit, and **sfname** is the name of the special file you wish to create.

Each class of device can have additional class-dependent attributes (such as the bus address for an HP-IB auto-addressed file).

For HP-IB devices, the driver is *instr0*. Thus, to create a special file named */dev/bus* for HP-IB lu 1, you use the command:

```
mksf -d instr0 -l 1 /dev/bus
```

When creating auto-addressed HP-IB special files, you add another option **-a** to associate the address with the device. For example, to create an auto-addressed special file called */dev/plotter*, at bus address 7 on HP-IB lu 2, you could type:

```
mksf -d instr0 -l 2 -a 7 /dev/plotter
```

For the AFI card, the driver is *gpio0*. Thus, to create a special file named */dev/afi* for GPIO lu 0, you could use the command:

```
mksf -d gpio0 -l 0 /dev/afi
```

For more information on *mksf* or *mknod*, refer to the *HP-UX Reference*.

Hardware Effects on DIL Routines

The HP-IB card supported on the Series 800 is the HP 27110B HP-IB interface; the GPIO card is the HP 27114A Asynchronous FIFO Interface (*AFI*).

This section presents some restrictions on using the DIL routines on Series 800 computers. These restrictions are organized under the DIL routine to which they apply. The routines are presented in alphabetical order. A list of *errno* error names can be found in section two of the *HP-UX Reference*. *Errno* numeric values are defined in the file */usr/include/sys/errno.h*.

hpib_rqst_srvce

The *hpib_rqst_srvce* routine only permits bit 6 of the serial poll *response* to be set. If *hpib_rqst_srvce* is called with a *response* having bit 6 set, the interface sends <01000000> (64 decimal) in response to serial poll; if bit 6 is not set in *response*, the interface sends <10000000> (128 decimal). See "The Computer as a Non-Active Controller" in Chapter 3.

io_eol_ctl

The AFI driver does not support pattern matching on reads; all *io_eol_ctl* calls return -1 and set *errno* to **EINVAL**.

io_reset

When an HP-IB interface is reset via *io_reset*, the card's parallel poll response is set to 0; its serial poll response is set to 128; its HP-IB address is read off the hardware switches; and the card is put on-line. Any enabled interrupts are preserved. If the card is configured as system controller, then Interface Clear (IFC) is pulsed and Remote Enable (REN) is asserted.

When an AFI interface is reset via *io_reset*, each of the three control output lines is reset to zero, the incoming Attention Request (ARQ) is disabled, the ARQ flip flop is cleared, the ARQ enable flip flop and the handshake to the peripheral are disabled, and the FIFO buffer is flushed out.

io_speed_ctl

The *io_speed_ctl* routine is not supported on Series 800 computers; transfer is always done via DMA.

io_timeout_ctl

On Series 800 computers, the timeout you specify via *io_timeout_ctl* is rounded up to the nearest 10-millisecond boundary. For example, if you specify a timeout of 125000 microseconds (125 milliseconds), the effective timeout is rounded up to 130 milliseconds.

DIL functions, *read*, or *write* requests that time out, return a value of -1 and set *errno* to either **ETIMEDOUT** or **EINTR**. If the request can be aborted normally, then *errno* is set to **ETIMEDOUT** . Otherwise, the HP-IB card is reset and **EINTR** is returned.

io_width_ctl

The only allowable data path width for HP-IB devices is 8. AFI devices support 8-bit and 16-bit data paths. If you specify any other width, *io_width_ctl* returns an error indication.

Return Values for Special Error Conditions

On specific error conditions, the Series 800 sets *errno* values which are different from what is expected from the DIL as documented in the HP-UX Standard. For example, when any request times out, *errno* is set to **ETIMEDOUT** ("connection timed out") or instead of setting it to **EOI**. Also, upon HP-IB requests that require the interface to be the active controller or the system controller, set *errno* to **EACCES** ("permission denied"). Requests that are aborted due to system power failure set *errno* to **EINTR** ("interrupted system call"); in addition, your process receives the signal **SIGPWR**, which indicates recovery of system power.

DIL Support of HP-IB Auto-Addressed Files

As noted in Chapter 3 in the section called "Setting Up Talkers and Listeners," one class of HP-IB special files, known as *auto-addressed* files, are associated with a given address on the bus. For *read* and *write* requests to these files, addressing is done automatically; that is, the sequence of talk and listen bus commands is generated for you.

In general, the DIL functions are not defined for auto-addressed files. On the Series 800, however, many of them are implemented, but with more device-oriented actions.

Important

The DIL Standard does not currently specify a functional definition for the support of auto-addressed files. When support for auto-addressed files becomes part of the DIL Standard, the specific functionality implemented may differ from the implementation described here for the Series 800. Please keep this in mind when developing programs which take advantage of this new functionality.

The following table shows which DIL functions are supported on auto-addressed files. Entries in the first column work the same on both auto-addressed and non-auto-addressed (also called *raw bus*) files. Entries in the second column are somewhat different for auto-addressed files; entries in the third column are not supported on HP-IB auto-addressed files and will return an error indication if used.

Routine	Same Effect	Different Effect	Not Allowed
hpib_abort	X		
hpib_bus_status	X		
hpib_card_ppoll_resp		X	
hpib_eoi_ctl	X		
hpib_io		X	
hpib_pass_ctl			X
hpib_ppoll	X		
hpib_ppoll_resp_ctl			X
hpib_ren_ctl		X	
hpib_rqst_srvce			X
hpib_send_cmd		X	
hpib_spoll		X	
hpib_status_wait			X
hpib_wait_on_ppoll		X	
io_eol_ctl	X		
io_get_term_reason	X		
io_interrupt_ctl	X		
io_on_interrupt		X	
io_reset			X
io_speed_ctl	X		
io_timeout_ctl	X		
io_width_ctl	X		

Those functions in the second column, which operate differently on raw bus and auto-addressed special files, are discussed below.

hpib_card_ppoll_resp

Calling *hpib_card_ppoll_resp* on an auto-addressed file does not configure the HP-IB interface card; rather, it configures the device associated with the file with the appropriate addressing and Parallel Poll configuration commands.

hpib_io

For those *iodetail* structures that send commands (by setting the *mode* flag to HPIBWRITE or HPIBATN), *hpib_io* prefixes the command buffer *buf* with the appropriate device addressing (see *hpib_send_cmd*, below). For data transfers (with *mode* set to HPIBREAD or HPIBWRITE) using auto-addressed files, the addressing is also done for you.

hpib_ren_ctl

Setting REN (by setting the *flag* parameter to a non-zero value) on an auto-addressed file addresses the associated device *before* asserting REN. Clearing REN (by setting *flag* to a zero) addresses the device and sends it a Go To Local command, in lieu of clearing REN.

hpib_send_cmd

Sending HP-IB commands to an auto-addressed file via *hpib_send_cmd* does the appropriate device addressing for you. The *command* buffer you pass down to the device is preceded by the commands necessary to remove any previous listeners on the bus, address the Active Controller to talk, and configure the file's associated device to listen.

hpib_spoll

Performing a serial poll on an auto-addressed file polls the associated device; any bus address passed via the *ba* argument is ignored.

hpib_wait_on_ppoll

For auto-addressed files, the *mask* argument is ignored; only the address associated with the device is polled. In addition, the *sense* argument only specifies the sense of the particular device's assertion. Successful completion of the *hpib_wait_on_ppoll* request implies that the device responded to parallel poll.

io_on_interrupt

The only allowable interrupt for auto-addressed files is SRQ.

Performance Tips

DIL performance improvements for the Series 800 fall into two categories: those that keep your process from waiting for resources, and those that actually improve your I/O performance. The first three of the tips described below fall into the first category; the last two are in the second category.

Process Locking

Normally, the operating system swaps processes in and out of memory; you can circumvent this swapping by using the *plock* system call.

If you are running as the super-user (or have the **PRIV_MLOCK** capability), you can use *plock* to lock your process in memory; *plock* prevents the system from swapping out the process's code, data, or both.

The following example illustrates its use:

```
#include <sys/lock.h>
int plock();

main() {

      plock(PROCLOCK);        /* lock text and data segments into memory */
      :
      plock(UNLOCK);                /* unlock the process */
}
```

Refer to *plock(2)* and *getprivgrp(2)* in the *HP-UX Reference* for more information.

Setting Real-Time Priority

The operating system schedules processes based on their priority. Under normal circumstances, the priority of a process drops over time, allowing newer processes a greater share of CPU time. You can assign a higher priority to your process and keep its priority from dropping by using the *rtprio* system call.

If you are running as the super-user (or have the **PRIV_RTPRIO** capability), you can use *rtprio* to give your process a real-time priority. Real-time processes run at a higher priority than normal user processes; they get preempted only by voluntarily giving up the CPU or by being interrupted by a higher priority process or interrupt.

You must be careful when using real-time priorities because you can increase your priority above those of important system processes. The following example places the calling process at the lowest (least important) real-time priority:

```
#include <sys/rtprio.h>
#define  ME   0       /* a zero process ID means this process */
int rtprio();

main() {
    rtprio(ME, 127);                  /* Turn on real-time priority for ME  */
    :
    rtprio(ME, RTPRIO_RTOFF); /* Turn off real-time priority for ME */
}
```

Refer to *rtprio(2)* and *getprivgrp(2)* in the *HP-UX Reference* for more information.

Preallocating Disc Space

If your process is reading large amounts of data and writing it to a file, you can block while the operating system allocates disc space. However, you can allocate disc space in advance by using the *prealloc* system call. The following example opens a file and preallocates 65536 bytes of space for that file:

```
#include <fcntl.h>
#define MAX_SIZE 65536
int prealloc();

main() {
    int eid;

    eid = open("data_file", O_WRONLY);
    prealloc(eid, MAX_SIZE);     /* preallocate space to write into */
    :
}
```

Refer to *prealloc(2)* in the *HP-UX Reference* for more information.

Reducing System Call Overhead

Most DIL function calls you make on the Series 800 map into system calls. Therefore, you can cut down on operating system overhead by using fewer library calls. In particular, use auto-addressed files for all read and write operations, rather than using an extra call to *hpib_send_cmd* to do addressing.

Setting Up Faster Data Transfers

Because of the I/O architecture of the Series 800, data transfers run more efficiently if your data buffers are aligned on a page boundary. The number of bytes per page is defined as **NBPG** and can be referenced by including *<sys/param.h>*. The following example shows how to allocate and page-align a data buffer:

```
#include <sys/param.h>        /* defines NBPG and roundup(x, y)         */
#define REAL_SIZE 1024        /* amount of memory we want to page-align */
char *malloc();

main() {
    char *malloc_ptr, *align_ptr;

    :
    malloc_ptr = malloc(NBPG + REAL_SIZE);   /* allocate memory       */
    align_ptr  = roundup(malloc_ptr, NBPG);  /* and round up the ptr  */
                    /* in future data transfers, use align_ptr      */
    :
    free(malloc_ptr);         /* when we're done with the data */
}
```

In addition, even count transfers run more quickly than odd count transfers.

ASCII Character Codes

E

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
NUL	0	00000000	000	00	
SOH	1	00000001	001	01	GTL
STX	2	00000010	002	02	
ETX	3	00000011	003	03	
EOT	4	00000100	004	04	SDC
ENQ	5	00000101	005	05	PPC
ACK	6	00000110	006	06	
BEL	7	00000111	007	07	
BS	8	00001000	010	08	GET
HT	9	00001001	011	09	TCT
LF	10	00001010	012	0A	
VT	11	00001011	013	0B	
FF	12	00001100	014	0C	
CR	13	00001101	015	0D	
SO	14	00001110	016	0E	
SI	15	00001111	017	0F	
DLE	16	00010000	020	10	
DC1	17	00010001	021	11	LLO
DC2	18	00010010	022	12	
DC3	19	00010011	023	13	
DC4	20	00010100	024	14	DCL
NAK	21	00010101	025	15	PPU
SYNC	22	00010110	026	16	
ETB	23	00010111	027	17	
CAN	24	00011000	030	18	SPE
EM	25	00011001	031	19	SPD
SUB	26	00011010	032	1A	
ESC	27	00011011	033	1B	
FS	28	00011100	034	1C	
GS	29	00011101	035	1D	
RS	30	00011110	036	1E	
US	31	00011111	037	1F	

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
space	32	00100000	040	20	LA0
!	33	00100001	041	21	LA1
"	34	00100010	042	22	LA2
#	35	00100011	043	23	LA3
$	36	00100100	044	24	LA4
%	37	00100101	045	25	LA5
&	38	00100110	046	26	LA6
'	39	00100111	047	27	LA7
(	40	00101000	050	28	LA8
)	41	00101001	051	29	LA9
*	42	00101010	052	2A	LA10
+	43	00101011	053	2B	LA11
,	44	00101100	054	2C	LA12
-	45	00101101	055	2D	LA13
.	46	00101110	056	2E	LA14
/	47	00101111	057	2F	LA15
0	48	00110000	060	30	LA16
1	49	00110001	061	31	LA17
2	50	00110010	062	32	LA18
3	51	00110011	063	33	LA19
4	52	00110100	064	34	LA20
5	53	00110101	065	35	LA21
6	54	00110110	066	36	LA22
7	55	00110111	067	37	LA23
8	56	00111000	070	38	LA24
9	57	00111001	071	39	LA25
:	58	00111010	072	3A	LA26
;	59	00111011	073	3B	LA27
<	60	00111100	074	3C	LA28
=	61	00111101	075	3D	LA29
>	62	00111110	076	3E	LA30
?	63	00111111	077	3F	UNL

Character Codes (cont.)

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
@	64	01000000	100	40	TA0
A	65	01000001	101	41	TA1
B	66	01000010	102	42	TA2
C	67	01000011	103	43	TA3
D	68	01000100	104	44	TA4
E	69	01000101	105	45	TA5
F	70	01000110	106	46	TA6
G	71	01000111	107	47	TA7
H	72	01001000	110	48	TA8
I	73	01001001	111	49	TA9
J	74	01001010	112	4A	TA10
K	75	01001011	113	4B	TA11
L	76	01001100	114	4C	TA12
M	77	01001101	115	4D	TA13
N	78	01001110	116	4E	TA14
O	79	01001111	117	4F	TA15
P	80	01010000	120	50	TA16
Q	81	01010001	121	51	TA17
R	82	01010010	122	52	TA18
S	83	01010011	123	53	TA19
T	84	01010100	124	54	TA20
U	85	01010101	125	55	TA21
V	86	01010110	126	56	TA22
W	87	01010111	127	57	TA23
X	88	01011000	130	58	TA24
Y	89	01011001	131	59	TA25
Z	90	01011010	132	5A	TA26
[	91	01011011	133	5B	TA27
\	92	01011100	134	5C	TA28
]	93	01011101	135	5D	TA29
^	94	01011110	136	5E	TA30
_	95	01011111	137	5F	UNT

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
`	96	01100000	140	60	SC0
a	97	01100001	141	61	SC1
b	98	01100010	142	62	SC2
c	99	01100011	143	63	SC3
d	100	01100100	144	64	SC4
e	101	01100101	145	65	SC5
f	102	01100110	146	66	SC6
g	103	01100111	147	67	SC7
h	104	01101000	150	68	SC8
i	105	01101001	151	69	SC9
j	106	01101010	152	6A	SC10
k	107	01101011	153	6B	SC11
l	108	01101100	154	6C	SC12
m	109	01101101	155	6D	SC13
n	110	01101110	156	6E	SC14
o	111	01101111	157	6F	SC15
p	112	01110000	160	70	SC16
q	113	01110001	161	71	SC17
r	114	01110010	162	72	SC18
s	115	01110011	163	73	SC19
t	116	01110100	164	74	SC20
u	117	01110101	165	75	SC21
v	118	01110110	166	76	SC22
w	119	01110111	167	77	SC23
x	120	01111000	170	78	SC24
y	121	01111001	171	79	SC25
z	122	01111010	172	7A	SC26
{	123	01111011	173	7B	SC27
|	124	01111100	174	7C	SC28
}	125	01111101	175	7D	SC29
~	126	01111110	176	7E	SC30
DEL	127	01111111	177	7F	SC31

DIL Programming Example

F

This appendix contains a program listing for an HP-IB driver that uses Device I/O Library subroutines to drive various models of Hewlett-Packard Amigo protocol HP-IB printers. It is provided solely for illustrative use, and is not to be construed as optimum programming technique nor necessarily totally bug-free although the program has been extensively tested.

It contains not only examples of DIL subroutine usage, but also other useful programming techniques and structures that can make the task of writing specialized I/O programs much easier.

```
/************************************************************************/
/* This example Amigo printer driver uses a byte stream as standard     */
/* input and Amigo protocol as output to HP-IB driver (21).  Any special */
/* character handling should be done by a filter that feeds this driver. */
/*                                                                      */
/* This example program is provided for solely illustrative purposes to  */
/* demonstrate typical use of Device I/O Library (DIL) subroutines.  No  */
/* representations are made as to its suitability for any given         */
/* application.                                                         */
/*                                                                      */
/* While the program is intended to show good programming practice, it   */
/* does not necessarily represent optimum programming efficiency.       */
/************************************************************************/

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/sysmacros.h>

/* HP-IB addressing group bases */
#define    LAG_BASE        0x20     /*  listener address base */
#define    TAG_BASE        0x40     /*  talker address base */
#define    SCG_BASE        0x60     /*  secondary address base */

/* HP-IB command equates in odd parity */
#define    GTL         0x01        /*  go to local */
#define    SDC         0x04        /*  selective device clear */
#define    DCL         0x94        /*  device clear */
#define    UNL         0xbf        /*  unlisten */
#define    UNT         0xdf        /*  untalk */

```

```
34  /* HP-IB secondary commands */
35  #define             PR_SEC_DSJ      SCG_BASE+16
36  #define             PR_SEC_DATA     SCG_BASE+0
37  #define             PR_SEC_RSTA     SCG_BASE+14
38  #define             PR_SEC_MASK     SCG_BASE+01
39  #define             PR_SEC_STRD     SCG_BASE+10     /* 2608A */
40
41  /* output of DSJ operation 2608A */
42  #define             PR_ATTEN        0x0001
43  #define             PR_RIBBON       0x0002
44  #define             PR_ATT_PAR      0x0003
45  #define             PR_PAPERF       0x0010
46  #define             PR_SELF         0x0020
47  #define             PR_PRINT        0x0040
48
49  /* output of DSJ operation the rest of the printers */
50  #define             PR_RFDATA       0x0000
51  #define             PR_SDS          0x0001
52  #define             PR_RIOSTAT      0x0002
53
54  /* ppoll mask bits */
55  #define             PR_M_RFD        0x0010
56  #define             PR_M_STATUS     0x0020
57  #define             PR_M_POWER      0x0040
58  #define             PR_M_PAPER      0x0080
59
60  /* default parallel poll mask */
61  unsigned char pmask[1] = {PR_M_PAPER+PR_M_POWER+PR_M_STATUS+PR_M_RFD};
62
63  /* masks for io status byte in case of 2608A */
64  #define    PR_I_POW         0x0001
65  #define             PR_I_OPSTAT     0x0040
66  #define             PR_I_LINE       0x0080
67
68  /* masks for io status byte the rest of the printers */
69  #define    PR_I_POWER       0x0001
70  #define             PR_I_PAPER      0x0002
71  #define             PR_I_PARITY     0x0008
72  #define             PR_I_RFD        0x0040
73  #define             PR_I_ONLINE     0x0080
74
75  /* define printer types */
76  #define T2608A              1
77  #define T2631A              2
78  #define T2631B              3
79  #define T2673A              4
80  #define QjetPlus    5
81  #define T2632A              6
82  #define T2634A              7
83
```

```
84  int ptr_type;       /* type of printer */
85
86  /* setup defines for fatal returns */
87  #define F_RTRN      1
88  #define F_EXIT      0
89
90  /* setup defines for HP-IB_msg */
91  #define H_READ              1
92  #define H_WRITE     2
93  #define H_CMND              4
94
95  /* default timeout value (in seconds) to infinity */
96  int timeout =       0;
97
98  /* default size of output buffer to printer */
99  int bufsz =         32;
100
101 /* device file suffix for raw hpib dev */
102 char ptr_raw[] = "_00";
103
104 /* default output dev to printer */
105 char ptr_dev[100] = "/dev/lp";
106
107 extern char *optarg;
108 extern int optind;
109 extern int errno;
110
111 /* file id for raw HP-IB dev */
112 int eid;
113
114 /* configured listen and talk commands */
115 int MTA;    /* my talk address */
116 int MLA;    /* my listen address */
117 int DTA;    /* device (printer) talk address */
118 int DLA;    /* device (printer) listen address */
119
120 /* device bus address & my bus address */
121 int devba, myba;
122
123 /* my name */
124 char *procnam;
125
126 int Debug = 0;
127
128 main(argc, argv)
129 int argc;
130 char *argv[];
131 {
132
133    register i, c;
```

```
134     register unsigned char *outbuf; /* output buffer pointer */
135     int status;
136     int selcode;                    /* select code of printer */
137     struct stat statbuf;
138     int errflg = 0;
139
140     procnam = argv[0];              /* save pointer to my name */
141
142     /* GET USER SUPPLIED OPTIONS AND PRINTER FILE NAME */
143     while ((i = getopt(argc, argv, "b:t:p:D")) != EOF) {
144             switch (i) {
145             /* set the buffer size to output to printer */
146             case 'b': if ((bufsz = atoi(optarg)) <= 0) errflg++;
147                     break;
148
149             /* get the new timeout value in seconds */
150             case 't': if ((timeout = atoi(optarg)) < 0) errflg++;
151                     break;
152
153             /* Set the parallel poll pmask (mostly for debugging) */
154             case 'p': if ((pmask[0] = atoi(optarg)) < 0) errflg++;
155                     break;
156
157             case 'D': Debug++; break;
158
159             case '?': errflg++;break;
160             }
161     }
162     /* get printer dev if supplied */
163     if (optind < argc)
164             strcpy(ptr_dev, argv[optind]);
165
166     if (errflg) {
167   fprintf(stderr, "usage: %s [-bbufsz -ttmout] [printer_dev]\n", procnam);
168   fprintf(stderr, "-b bufsz > Output buf size to printer (%d)\n", bufsz);
169   fprintf(stderr, "-t tmout > Max seconds to output buffer (%d)\n", timeout);
170   fprintf(stderr, "printer_dev > Printer device file     (%s)\n", ptr_dev);
171   fprintf(stderr, "-p ppoll_mask > Parallel poll mask  (0x%02x)\n",pmask[0]);
172             exit(2);
173     }
174     /* get memory for the output buffer */
175     outbuf = (unsigned char *)malloc (bufsz + 4);
176   /*
177     NOTE: Printer device file (/dev/lp) is used only to get printer select
178     code and HP-IB bus address.  This is because attention-true (ATN)
179     requests can only be sent to an "HP-IB raw bus device file".  Therefore
180     after getting the SC and BA we will use a "HP-IB raw bus device file" to
181     do all the work, but it must exist with a name similar to the printer
182     device; i.e. "/dev/lp" is changed to "/dev/lp_07", where the "07" is the
183     select code.
```

```
184 */
185     /* check if printer device exists */
186     if (stat(ptr_dev, &statbuf) < 0)
187             fatal_err("stat", ptr_dev, F_EXIT);
188
189     /* check if it is a character device file */
190     if ((statbuf.st_mode & S_IFMT) != S_IFCHR)
191             fatal_err("Must be a char_special file", ptr_dev, F_EXIT);
192
193     /* extract selectcode from the printer device */
194     selcode =       m_selcode(statbuf.st_rdev);
195
196     /* make the HP-IB raw bus device file name from selectcode */
197     ptr_raw[1] += selcode / 16;
198     ptr_raw[2] += selcode % 16;
199     if ((selcode % 16) >= 10) ptr_raw[2] += ('a' - '0' -10);
200     strcat(ptr_dev, ptr_raw);
201
202     /* get device BA from the printer device and config control bytes */
203     devba = m_busaddr(statbuf.st_rdev);
204     DLA = LAG_BASE + devba; /* device listen address */
205     DTA = TAG_BASE + devba; /* device talk address */
206
207     /* open the HP-IB raw bus device */
208     if ((eid = open(ptr_dev, O_RDWR)) <0) {
209             fatal_err("Raw HP-IB open", ptr_dev, F_RTRN);
210 fprintf(stderr,
211 " The following commands executed as a super user may be necessary\n\n");
212 fprintf(stderr, "     # mknod %s c 21 0x%s1f00\n", ptr_dev, &ptr_raw[1]);
213 fprintf(stderr, "     # chmod 555 %s\n", ptr_dev);
214 fprintf(stderr, "     # chown lp  %s\n", ptr_dev);
215             exit(2);
216     }
217     /* get (my) BA of the controller and configure control bytes */
218     if ((myba = hpib_bus_status(eid, 7)) < 0)
219             fatal_err("Must be raw hpib driver (21)", ptr_dev,F_EXIT);
220     MLA = LAG_BASE + myba; /* controller (my) listen address */
221     MTA = TAG_BASE + myba; /* controller (my) talk address */
222
223     /* go do the Amigo identify */
224     ptr_type = amigo_identify();
225
226     if (Debug) {
227             printf("%s Identified ", ptr_dev);
228             switch(ptr_type) {
229             case T2608A:     printf("2608A");        break;
230             case T2631A:     printf("2631A");        break;
231             case T2631B:     printf("2631B");        break;
232             case T2673A:     printf("2673A");        break;
233             case QjetPlus:   printf("QuietJet Plus");break;
```

```
234             case T2632A:     printf("2632A");        break;
235             case T2634A:     printf("2634A");        break;
236             default: printf("You forgot one dummy"); break;
237             }
238             printf(" printer\n");
239     }
240     /* set the timeout to user requested value */
241     if (io_timeout_ctl(eid, timeout * 1000000) < 0)
242             fatal_err("io_timeout_ctl", ptr_dev, F_EXIT);
243
244     /* always tag last output data byte with EOI */
245     if (hpib_eoi_ctl(eid, 1) < 0)
246             fatal_err("hpib_eoi_ctl", ptr_dev, F_EXIT);
247
248     /* clear out the status bits */
249     amigo_clear();
250
251     /* check the status bits */
252     status = amigo_status();
253     if (Debug) printf("%s Printer status = 0x%x\n", ptr_dev, status);
254
255     /* set the ppoll mask required by some printers */
256     amigo_set_pmask();
257
258     /* MAIN OUTPUT LOOP */
259     i = 0;
260     while ((c = getchar()) != EOF) {
261             if (i == bufsz) {
262                     amigo_write(outbuf, i);
263                     i = 0;
264             }
265             outbuf[i++] = c;
266     }
267     /* post remaining buffer */
268     if (i) amigo_write(outbuf, i);
269     exit(0);
270 }
271
272 /* ROUTINE TO DO THE MAIN I/O TO THE BUSS */
273 /* lock bus, do preamble, read/write, do postamble and unlock bus */
274 /* preamble must be 3 or 4 bytes, postamble must be 1 or 2 bytes */
275 int
276 HPIB_msg(rw_flag, pcm1, pcm2, pcm3, buffer, length, ocm0, ocm1)
277 int         rw_flag;
278 int         pcm1;
279 int         pcm2;
280 int         pcm3;
281 char        *buffer;
282 int         length;
283 int         ocm0;
```

```
int        ocm1;
{
   unsigned char pre_cmd[4];
   unsigned char post_cmd[2];
   int tlog = -1;

   pre_cmd[0] = UNL;       /* always issue unlisten command first */
   pre_cmd[1] = pcm1;
   pre_cmd[2] = pcm2;
   pre_cmd[3] = pcm3;

   post_cmd[0] = ocm0;
   post_cmd[1] = ocm1;

   /* first get exclusive use of the bus */
   if (io_lock(eid) < 0)
          fatal_err("io_lock", ptr_dev, F_EXIT);

   /* send the preamble 3 or 4 bytes with attention true */
   if (hpib_send_cmnd(eid, pre_cmd, (pcm3 ? 4 : 3)) < 0)
          fatal_err("hpib_send_cmnd preamble", ptr_dev, F_EXIT);

   switch (rw_flag) {
   case H_READ:
          if ((tlog = read(eid, buffer, length)) < 0)
                 fatal_err("read", ptr_dev, F_EXIT);
          break;

   case H_WRITE:
          if ((tlog = write(eid, buffer, length)) < 0)
                 fatal_err("write", ptr_dev, F_EXIT);
          break;

   case H_CMND:
          return(0);
   default:
          return(-1);
   }
   /* send the postamble 1 or 2 bytes with attention true */
   if (hpib_send_cmnd(eid, post_cmd, (ocm1 ? 2 : 1)) < 0)
          fatal_err("hpib_send_cmnd postamble", ptr_dev, F_EXIT);

   /* at last unlock the bus so other bus users can access it */
   if (io_unlock(eid) < 0)
          fatal_err("io_unlock", ptr_dev, F_EXIT);

   return(tlog);
}

int
```

```
amigo_identify()
{
   unsigned char identify[2];

   /* TLK31 (UNT) is special for amigo identify */
   /* finish with a MTA (UNT is not save for non-amigo devices) */
   HPIB_msg(H_READ, MLA, UNT, SCG_BASE + devba, identify, 2, MTA, 0);

   switch(identify[0]) {
   case 32:
           /* Amigo identify */
           switch(identify[1]) {
           case 1:  return(T2608A);
           case 2:  return(T2631A);
           case 9:  return(T2631B);
           case 11: return(T2673A);
           case 13: return(QjetPlus);
           case 16: return(T2632A);
           case 17: return(T2634A);
           default:
                   printf("Unrecognized Amigo printer, ID2 = %d\n",
                           identify[1]); break;
           }
           break;
   case 33:
           if (identify[1] == 1)
                   printf("Ciper printer not supported yet!\n");
           break;
   default:
printf("Unrecognized Amigo Printer identify, ID1 = %d, ID2 = %d\n",
                   identify[0], identify[1]);
           break;
   }
   exit(2);
}

/* set the parallel poll mask value */
amigo_set_pmask()
{
   HPIB_msg(H_WRITE, MTA, DLA, PR_SEC_MASK, pmask, 1, UNL, 0);
}

/* do the amigo clear followed by selective device clear */
amigo_clear()
{
   HPIB_msg(H_WRITE, MTA, DLA, SCG_BASE + 16, "\0", 1, SDC, UNL);
}

/* get the dsj byte */
int
```

```
384 amigo_dsj()
385 {
386    unsigned char dsj_byte[1];
387
388    HPIB_msg(H_READ, MLA, DTA, PR_SEC_DSJ, dsj_byte, 1, UNT, 0);
389    return(dsj_byte[0]);
390 }
391
392 /* return the amigo status byte */
393 int
394 amigo_status()
395 {
396    unsigned char status_byte[1];
397
398    HPIB_msg(H_READ, MLA, DTA, PR_SEC_RSTA, status_byte, 1, UNT, 0);
399    return(status_byte[0]);
400 }
401
402 /* output a buffer to printer */
403 amigo_write(buffer, length)
404 char *buffer;
405 int length;
406 {
407    int status, dsj = 0;
408
409    /* write the buffer */
410    HPIB_msg(H_WRITE, MTA, DLA, PR_SEC_DATA, buffer, length, UNL, 0);
411 again:
412    /* now wait for parallel poll response */
413    if (Debug)  printf("%s Ppoll wait\n", ptr_dev);
414    if (hpib_wait_on_ppoll(eid, 0x80>>devba, 0) < 0)
415            fatal_err("hpib_wait_on_ppoll", ptr_dev, F_EXIT);
416
417    /* a DSJ is required to remove the ppoll response from device */
418    if (dsj = amigo_dsj()) {
419            if (Debug) printf("%s DSJ = 0x%x\n", ptr_dev, dsj);
420
421            status = amigo_status();
422            if (Debug) printf("%s STATUS = 0x%x\n", ptr_dev, status);
423            goto again;
424    }
425 }
426
427 /* output error message and conditionally abort */
428 fatal_err(message, fname, flag)
429 char *message;
430 char *fname;
431 {
432    fprintf(stderr, "%s: Error - %s of %s ", procnam, message, fname);
433    if   (errno) perror("");
```

```
434     else fprintf(stderr, "\n");
435
436     if (flag == F_RTRN) return;
437     if (flag == F_EXIT) exit(2);
438     exit(3);
439 }
```

Index

a

b

c

d

e

f

g

h

i

l

n

o

p

r

s

t

u

w

Table of Contents

The X.25 Network

Log, Status and Cleanup

Solving Problems

Introducing uucp 1

This article describes how to use ***uucp***, an HP-UX utility that lets you transfer files among HP-UX and UNIX™ systems linked by modem or direct connections.

To use the article, you should be the **system administrator** for some group (possibly only yourself). After a system has been configured to use *uucp*, you might identify some tasks that users in a group could perform.

As a system administrator, you should understand the HP-UX operating system, other UNIX™ systems, and the tasks performed by system administrators. For the sake of brevity, this article assumes you know the routine aspects of using HP-UX (i.e. using *vi*, using a Bourne or C shell, becoming the root user, manipulating files, and writing scripts).

Using *uucp* requires coordination of combinations of hardware and software; and getting a combination that works can take time and effort. Without looking at details, you need to coordinate or account for the following things:

Your version of HP-UX:	Some minor differences exist; for example, device file conventions.
Local and Remote Systems:	*Uucp* can work with Series 200, 300, and 500 computers in various Models (236, 320, 550, and so on).
Type of RS-232C Port:	The RS-232C port can be built-in, be incorporated into a multiplexer, or be on an installed interface card.
Type of Connection:	You can have modem or direct connections.
Type of System Interaction:	A given system can be active (send a file), passive (receive a file), or both. In addition, a given system can receive files from one system and forward them to another.

* UNIX is a trademark of AT&T Bell Laboratories.

Overview of General Tasks

To coordinate these things, you will need to complete the following general tasks:

- Examine possiblities for using *uucp* (e.g. the method for transferring files) in relation to your needs;
- Make decisions about which system configuration will meet those needs;
- Work through procedures for installing and configuring your system;
- Coordinate with other system administrators about such things as: hostnames, baud rates, phone numbers, and so on;
- Test your installation and configuration to see that it works;
- Use your system for transferring files among systems; and
- Maintain your system.

A Recommendation: Skim and Plan, then Configure

Because you have many options for doing these things, it is strongly recommended that you skim through this entire article before you do anything. Determine the locations of information. Think about possibilities and alert yourself to the tasks you need to perform. After you conceptualize the overall scheme of using *uucp*, plan your system and work systematically through the article. You should find the information you need; but at the same time, you should adopt a problem-solving approach to completing tasks.

Article Organization

Beyond this introductory chapter, the article has the following content:

Chapter 2	describes how to install the necessary, additional hardware for *uucp*. The chapter assumes you already have a working HP-UX system on an HP 9000 Series 200, 300, or 500 computer.
Chapter 3	steps you through the entire procedure for configuring the *uucp* utility. When you finish working through the steps, *uucp* should be working on your system.
Chapter 4	explains how to use *uucp*. The chapter begins with showing you how to transfer a file to a remote system and continues by showing more complex operations.

The following chapters provide optional or supplementary information. Some of the information duplicates or generalizes previous information, often going into more depth or detail.

Chapter 5	describes the file structure for *uucp*. You get examples of file transfer followed by discussions of the files involved in the examples.
Chapter 6	describes the daemons used by *uucp*.
Chapter 7	provides supplemental information. For example, you see the general syntax for *uucp* and get a discussion of its options. The chapter discusses related commands such as *uuclean*, *uulog*, and *uuname*.
Chapter 8	explains how to install, configure, and use X.25, which is another network that can work with *uucp*.
Chapter 9	described the information you can get from the LOGFILE, DIALLOG, and STST files and the Cleanup script.
Chapter 10	describes some procedures for solving problems. In addition, it explains many of the error messages you can see on the display when you encounter problems.

Refer to the Table of Contents to find general topics, and examine the index to find specific items.

A Few Reminders and Suggestions

Getting *uucp* to work properly can take some time. Before you begin to work, please note the following mild reminders and suggestions:

- Within HP-UX, *uucp* provides one of several ways to perform data communication operations. In this case, you will be communicating among systems. Besides using *uucp*, you might set up other data communications operations. This article discusses *uucp* in depth and mentions some other possibilities such as LAN and X.25.
- System-to-system communications can become highly technical. Do not work in a haphazard manner because you need to accommodate many details in the overall scheme of using *uucp*.
- Work patiently and systematically. Take time to read all information related to an operation. Think through what you want an operation to do. Then, perform necessary tasks. In some cases, you need to be aware of the subtle aspects of how something works.
- Expect to encounter some problems.
    - You can use *uucp* with myriad combinations of computers, interface cards, modems, cables and connections, and software configurations.
    - You can use *uucp* in several ways; and depending on what you do, you can use several commands that relate to *uucp*.

  It is possible to cycle through several installations and configurations before **your system** works properly.

By accounting for these things, you should have *uucp* functioning to your satisfaction. Most people take parts of one to three weeks to get everything working.

Installing Hardware for uucp 2

This chapter describes how to install the hardware for *uucp* on HP 9000 Series 200, 300, and 500 computers running HP-UX. The procedure is intended for system administrators.

During some steps, you need to differentiate among:

- Series 200, 300, or 500 computers.
- Modem or Direct connections.
- Built-in RS-232C port, interface cards, cables, connectors, and jumping of lines.
- Active, passive, and both active and passive options for transferring files. Active refers to a system that sends a file (i.e. does the calling); passive refers to a system that receives a file (i.e. gets the call).
- Versions of HP-UX. Differences among versions show up in such things as conventions for naming device files, and so on.
- Miscellaneous items such as editing files, typing commands, using special characters, and naming systems.

When you need to account for these things, find the documentation in this manual that applies to your situation, complete required tasks, and skip ahead to the next appropriate section. When there might be some confusion in a particular section, that section describes where to go to continue or get more information. Beyond this, the manual attempts to indicate when you need to consult another manual.

Step 1: Contact Remote System SAs

Because *uucp* transfers data among HP-UX and UNIX systems, you must first contact the system administrators for **all** systems with which you want to communicate. You will be gathering much information; therefore, use some systematic way to record the data.

Possible Problems in Collecting Data

This task can pose a before/after problem. On one hand, you need to get information about remote systems before you decide how to install your system. On the other hand, you might not yet know what information to collect or how to collect it and write it down.

To get help with this, you can visit with experienced SAs or take time to read ahead to see how to complete tasks. When you have enough background, come back here and visit with other SAs. In any case, you will need information about your system and all remote systems that will receive files from you (your system is active and those systems are passive) and that will send files to you (your system is passive and those systems are active).

You will add the information about your system and remote systems to the files used by *uucp*, among other things. In meeting with SAs for other systems, agree on and get the following information (you will see some examples that make sense later):

- The **node name** of each remote systems. Typical names include: `hpfcma`, `hpopus`, `acrdse`, and `norad`.
- The **type of connection**. This can be a direct connection (hardwired) or modem connection (telephone). Also, determine if the connections will be active, passive, or both.
- The **calendar/clock** times during which a remote system will accept communications from other systems. Gather this information according to the times for combinations of systems, especially your system and each other system. Also, determine when your system will accept files if you intend to work in an active and passive manner.

- The **telephone number** of each remote system. Each remote system that receives files from your system will have a phone number. For a modem connection, you need an actual phone number such as `303-229-3114`. For a direct connection, the modem device name (the **cul** line) might be something like `cul01`.
- The **baud rate** (rate of transfer of data) between your system and each remote system. Modem connections typically use 1200 or 2400 baud; direct connections typically use 9600 or 19 200 baud. In every case, your system and the remote system with which you communicate must use the same baud rate.
- Your **login name and password** (if any) on each remote system. Use typical names and passwords.
- The **commands** that must be included in the file named `L.cmds`. In addition, you need to know what information should go in several files (e.g. `L-devices`, `USERFILE`, and so on).

When you have this information, go to Step 2.

Step 2: Select Modem or Direct Connections

Uucp always uses standard RS-232C modem signals. Using these signals, you have two alternatives for connecting two systems:

Direct Connection	This is two "hard-wired" systems. That is, your system sends files directly to another system; or your system receives files directly from another system.
Modem Connection	Your system sends files to a modem that sends the files to a remote modem, and that modem sends the files to its remote system; or a remote system sends files to your system, using the intervening modems. With modem connections, you need to also work with phone lines and numbers.

The next several sections discuss modem and direct connections. Use the information to help you decide which connection better suits your needs.

Choosing a Modem or Direct Connection

Do not use a direct connection if the distance between two systems exceeds 15 meters or if other devices subject the systems to excessive noise. Use a modem connection instead.

Modem Connection Features

Using a modem provides a slow means of data communication (typically 1200 baud) over great distances (thousands of miles). The modems that connect two systems accommodate the Data Terminal Equipment (DTE) at the ends of the modems **and** the Data Communications Equipment (DCE) between the two modems. You can use any combination of systems that can be made to talk to each other. Figure 2-1 (shown later) illustrates a typical modem connection.

Direct Connection Features

Using a direct connection provides a fast means of communication (19 200 baud) over limited distances (up to 15 meters). You can make direct connections between:

- Two RS-232C interface cards.
- Two multiplexer cards, which consist of an interface card and an RS-232C panel that provides several connectors.
- A multiplexer on one system and an interface card on a different system.

That is, you can directly connect any combination of Series 200/300/500 computers via serial interfaces or multiplexers. Figures 2-2 through 2-15 (shown over several later pages) illustrate typical combinations of direct connections. Study all the illustrations to get an overall picture of using a direct connection.

A Typical Modem Connection

The following illustration (Figure 2-1) shows a typical modem connection. Note that RS-232C interface cards link the systems to their modems and the modems provide for transfer of data over telephone lines.

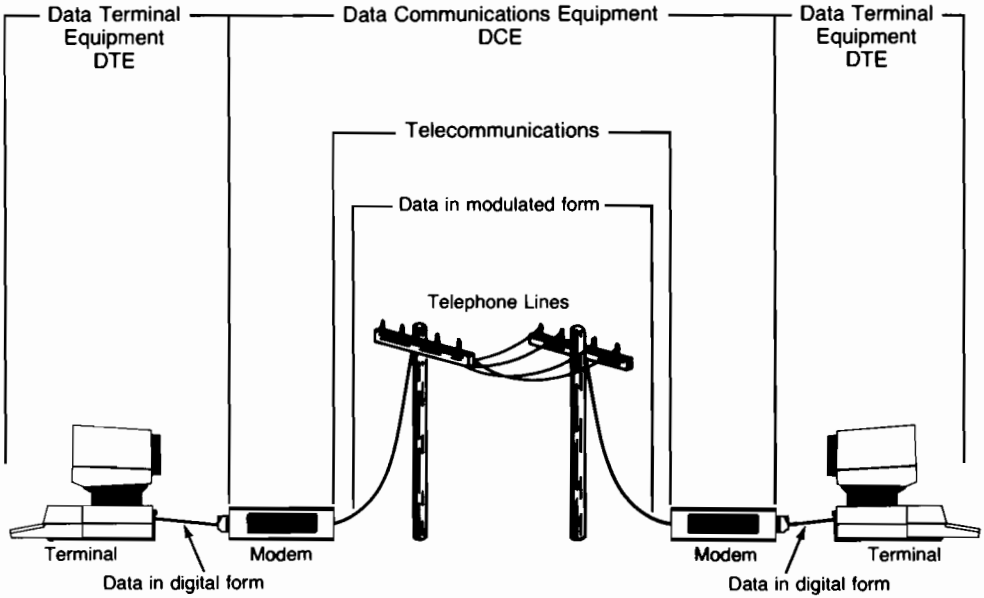


Figure 2-1. A Typical Modem Connection

Typical Combinations of Direct Connections

Before you install any interface cards, and before you decide on a direct or modem connection, it can be helpful to study Figures 2-1 (shown in the preceeding section) and Figures 2-2 through 2-15 (shown later) in relation to your situation. One of them probably approximates what you want to do. While the figures can suggest a difficult process, be aware that you need only to deal with **your** situation.

When you finish studying Figures 2-1 through 2-15:

- **If you decide to use a modem connection**, go to the later section in this chapter called "Installing a Phone Line".
- **If you decide to use a direct connection**, go to the later section in this chapter called "Step 3: Installing An Interface Card".

The next several pages show direct connections.

RS-232C to Multiplexer

In Figure 2-2, a Series 200/300 computer with an HP 98628A interface card is connected to a Series 500 computer with an HP 27140A (6-channel) multiplexer.

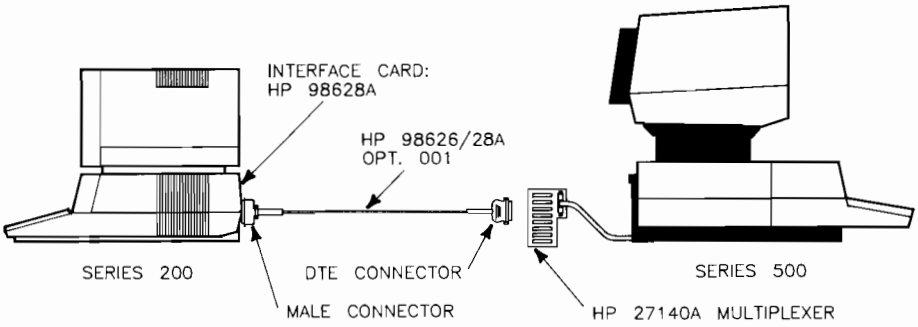


Figure 2-2. RS-232C to Multiplexer Connection Between Different Series

Getty On One End Only

In Figure 2-3, a Series 200/300 originating computer with an HP 98628A interface card is connected to a Series 500 receiving computer with an HP 27128A interface card. Note the callout which shows the getty on one end only situation.

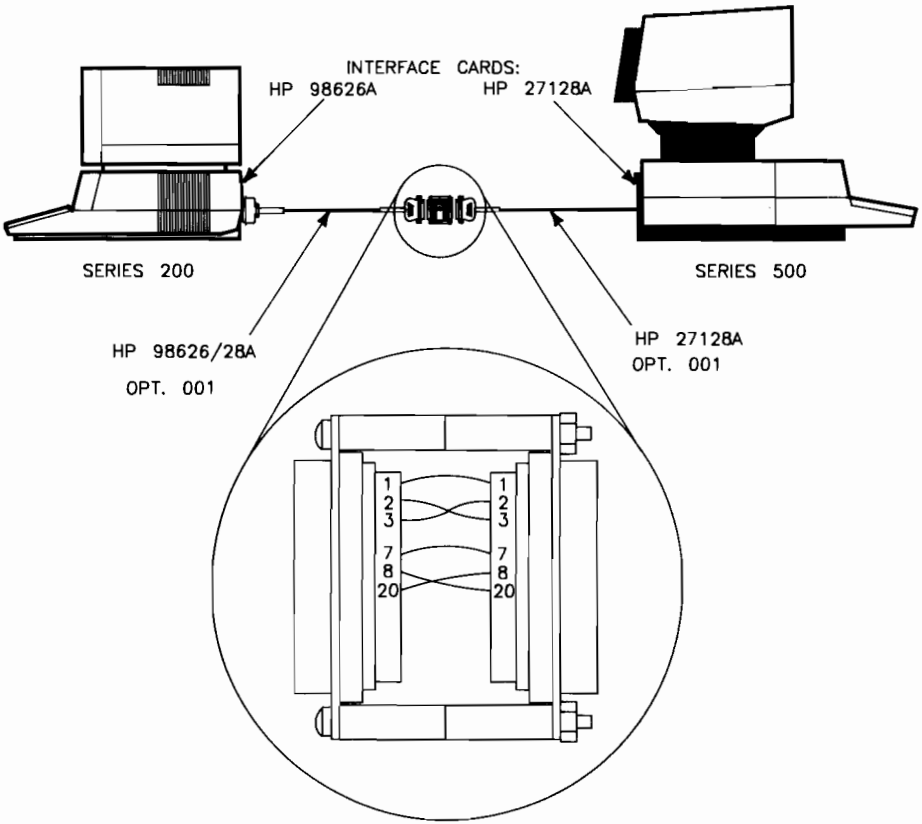


Figure 2-3. Two Different Series with Getty in One Direction

Getty On Both Ends

In Figure 2-4, a Series 200/300 computer with an HP 98628A interface card is connected to a Series 500 computer with an HP 27128A interface card. Again, note the callout which this time shows the getty on both ends at the same time.

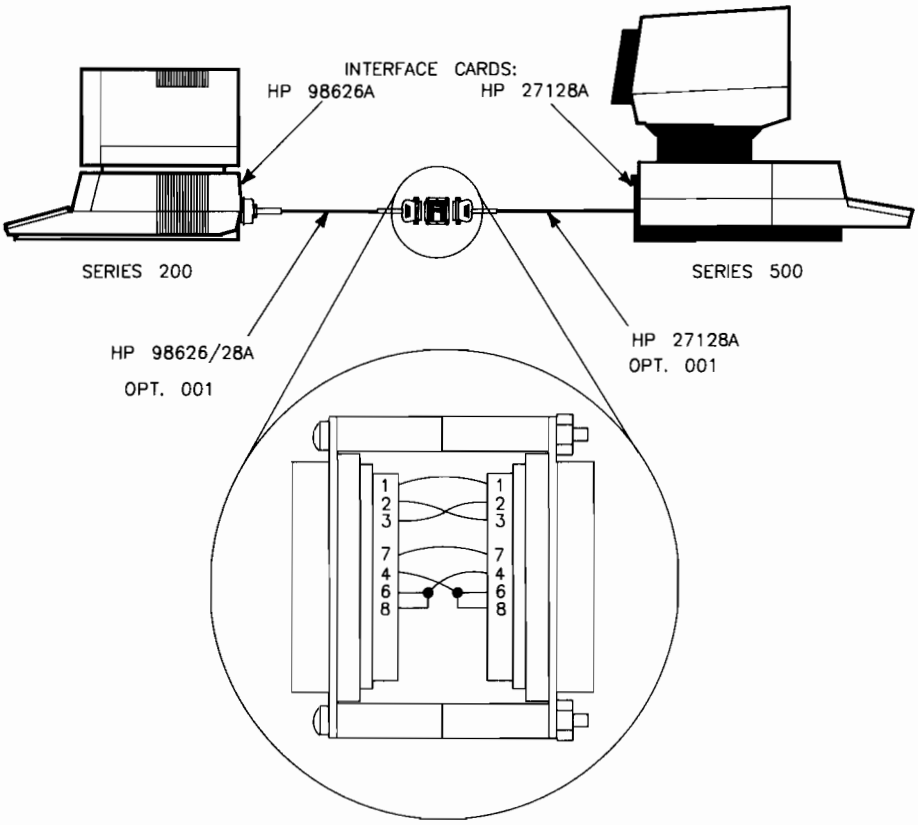


Figure 2-4. Two Different Series with Getty on Both Ends

Two Series 200/300 Serially

In Figure 2-5, HP 98628A interface cards connect two Series 200/300 computers. In this illustration, give close attention to the combination of cables and connectors and note the DTE/DCE relationships.

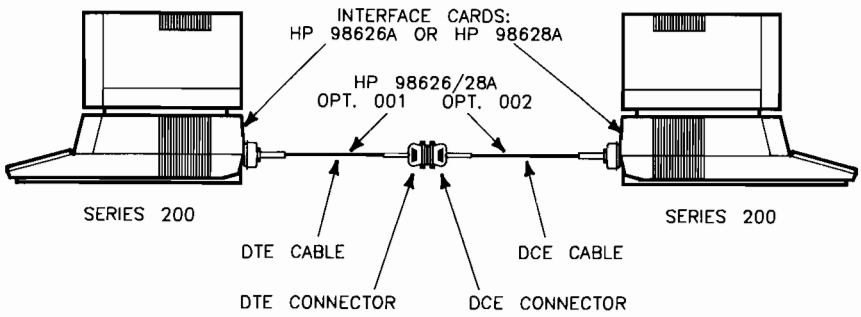


Figure 2-5. Two Similar Series Connected Serially

Serial to Multiplexer

In Figure 2-6, an HP 27128A serial interface card and an HP 27140A (6-channel) multiplexer card connect two Series 500 computers. Again, give close attention to the cables and connectors, noting the DTE/DCE relationships.

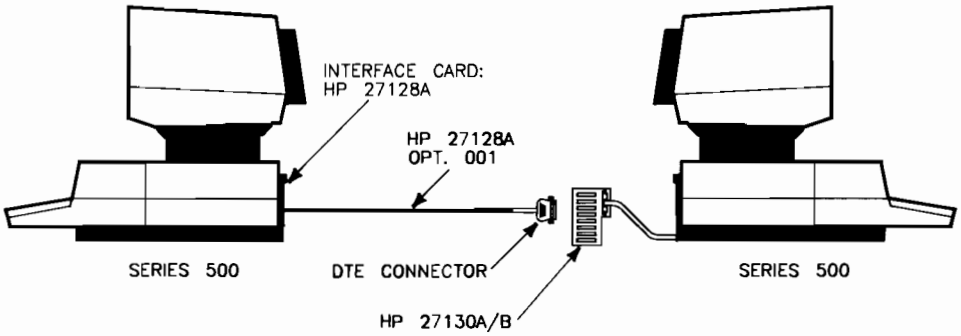


Figure 2-6. Series 500 Serial Interface to a Series 500 Multiplexer

Multiplexers with Jumping

In Figure 2-7, note the use of connectors between cables to link two HP 27140A multiplexers on two Series 500 computers. The callout shows the jumping of lines on the cable that connects the multiplexers.

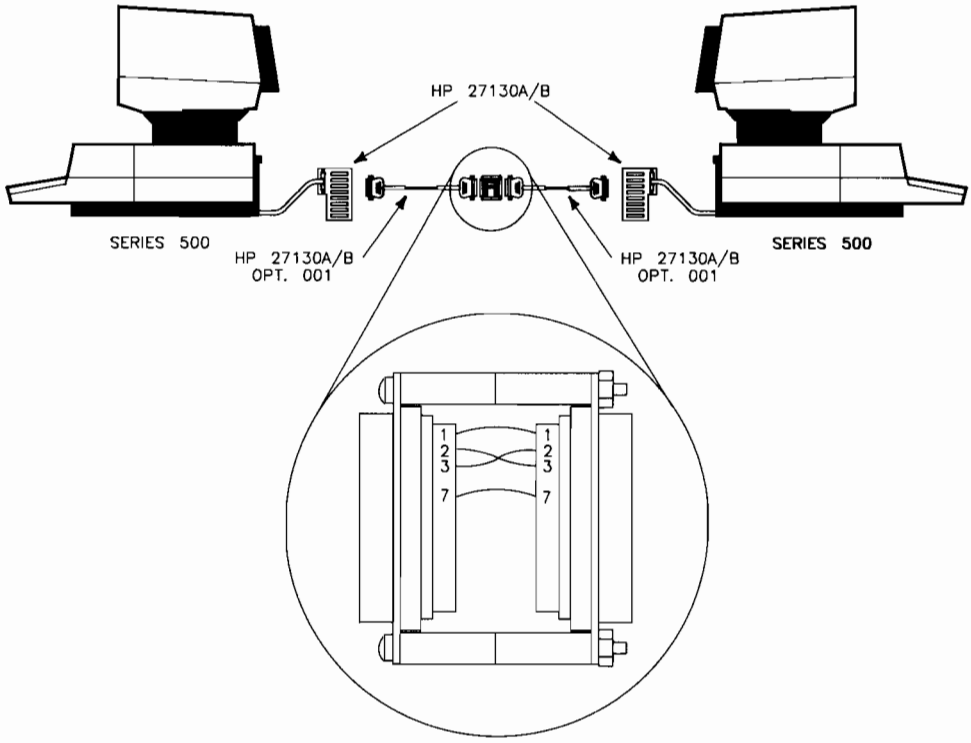


Figure 2-7. Jumping Lines to Connect Two Multiplexers

One-directional Serial with Jumping

In Figure 2-8, two HP 27182A interface cards connect two Series 500 computers. The call-out shows the jumping of lines between the two connectors that permits one-directional communication. The next figure shows a two-directional connection.

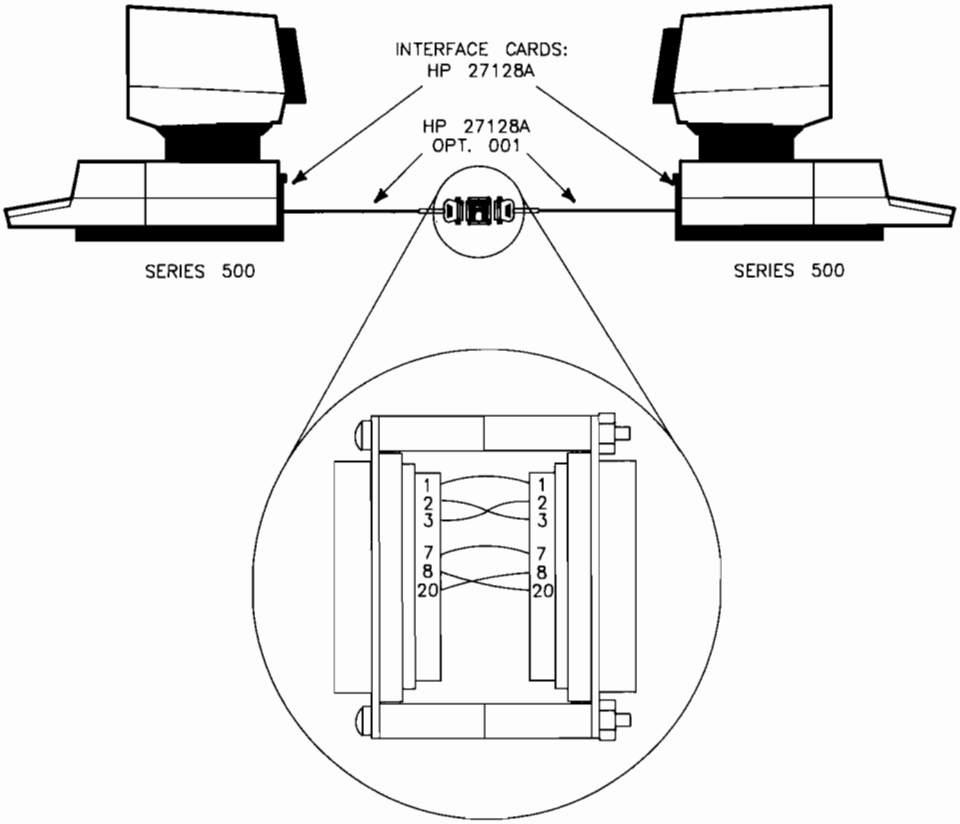


Figure 2-8. Jumping Serial Connections for One-directional Communication

Serial Connections with Jumping

In Figure 2-9, two HP 27182A interface cards serially connect two Series 500 computers. The callout shows the jumping of lines between the two connectors that permit two-directional communications.

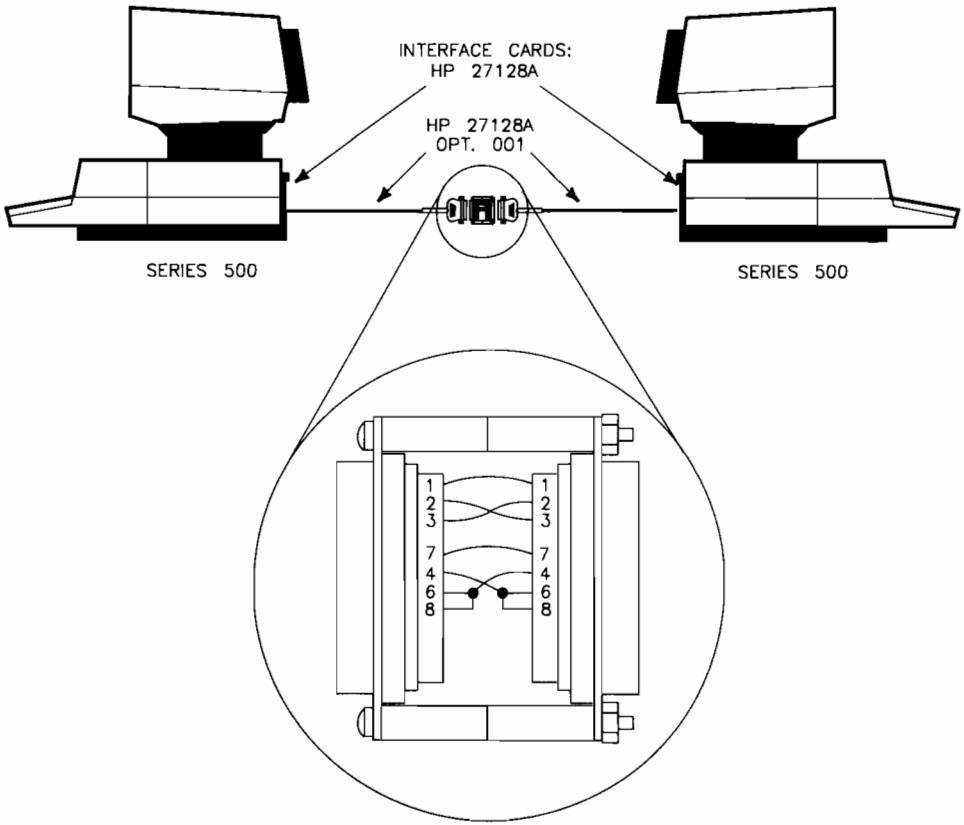


Figure 2-9. Jumping Serial Connections for Two-directional Communication

Special Serial to Multiplexer

In Figure 2-10, a special user-made cable that has the connections shown in the callout provides a one-directional connection between an RS-232C serial card and a 6-channel multiplexer card.

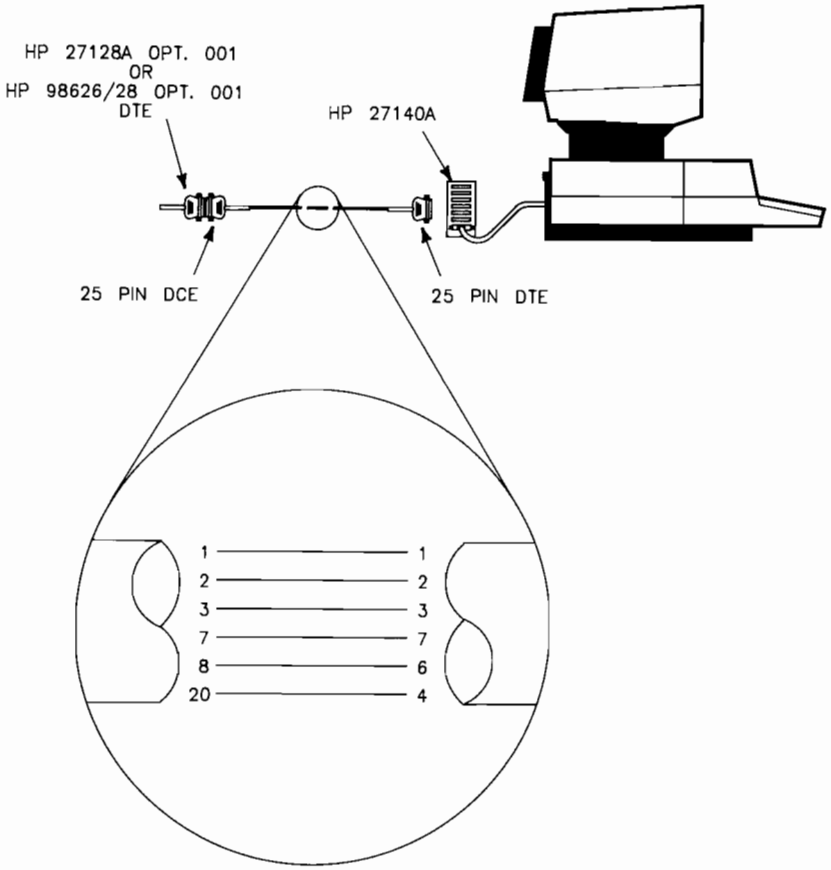


Figure 2-10. Serial-to-Multiplexer, One-directional Cabling

Special Multiplexer to Multiplexer

In Figure 2-11, a cable that has the connections shown in the callout provides a one-directional connection between an 8-channel multiplexer and a 6-channel multiplexer.

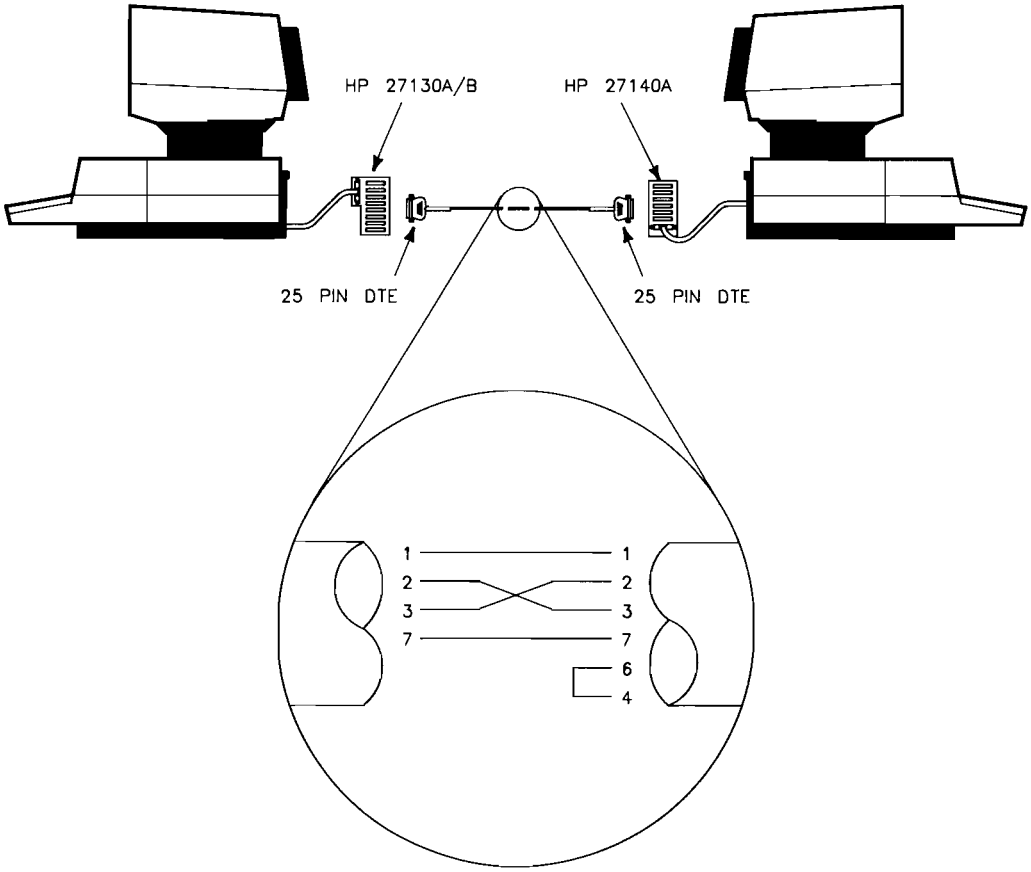


Figure 2-11. Dissimilar Multiplexer-to-Multiplexer, One-directional Cabling

Special Multiplexer to Multiplexer

In Figure 2-12, a special user-made cable that has the connections shown in the callout provides a one-directional connection between a 6-channel multiplexer and a 6-channel multiplexer.

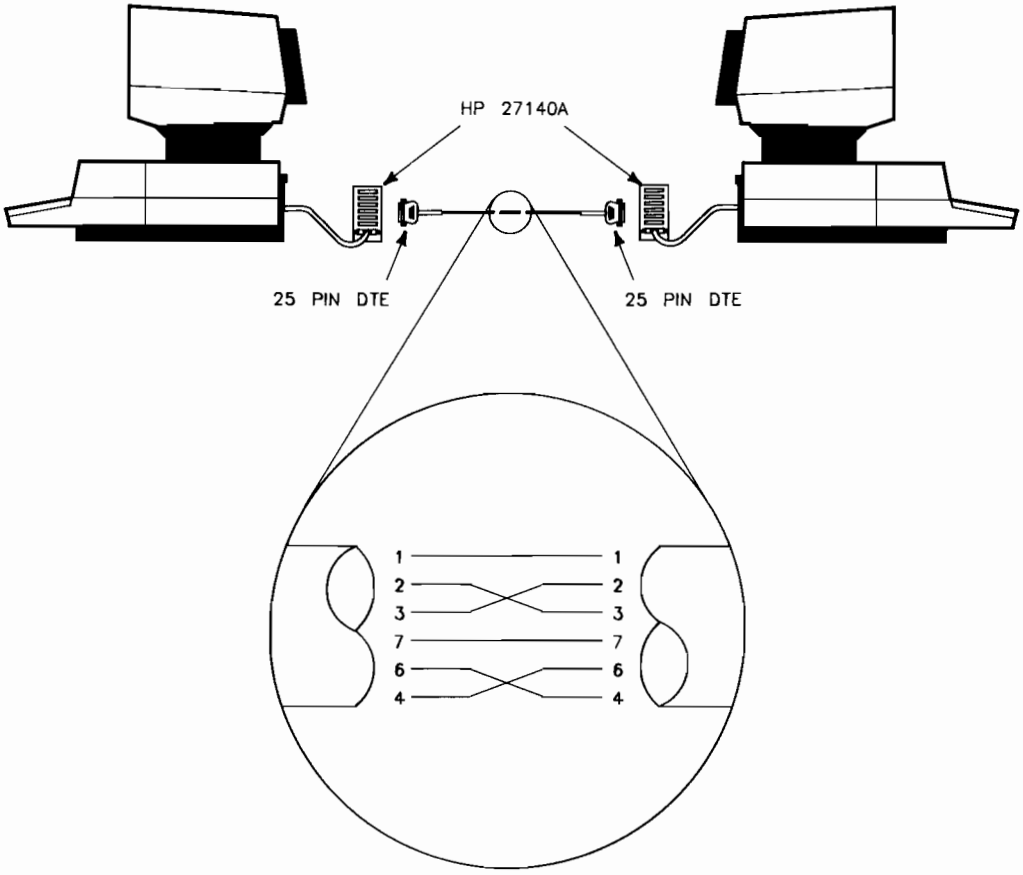


Figure 2-12. Same Channel, Multiplexer-to-Multiplexer, One-directional Cabling

Special Serial to Multiplexer

In Figure 2-13, a special user-made cable having the connections shown in the callout provides a two-directional connection between a serial card and a multiplexer card.

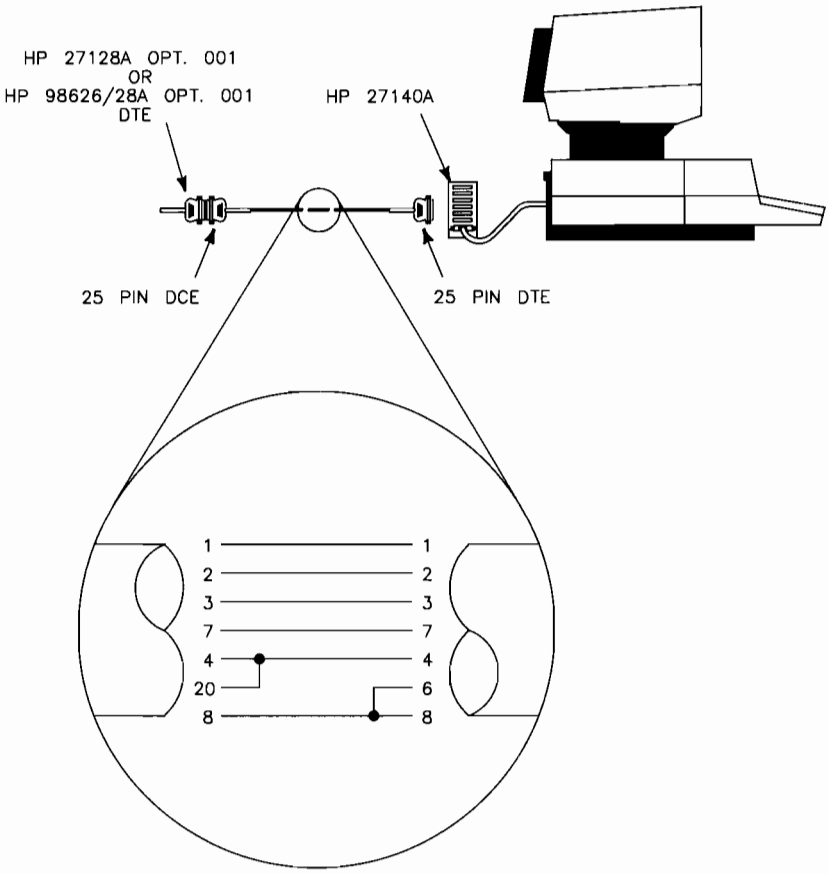


Figure 2-13. Serial-to-Multiplexer, Bidirectional Cabling

Special Multiplexer to Multiplexer

In Figure 2-14, a special user-made cable having the connections shown in the callout provides a two-directional connection between two similar multiplexers.

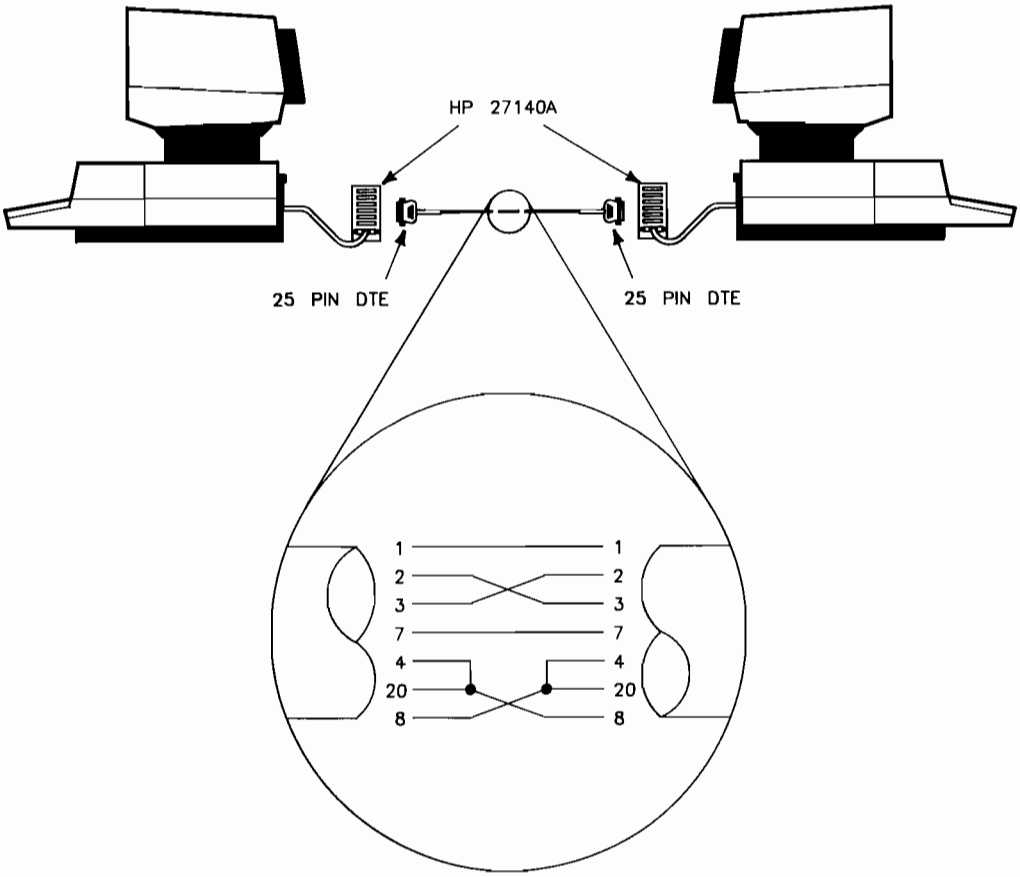


Figure 2-14. Same-Channel, Multiplexer-to-Multiplexer, Two-directional Cabling

Special Connectors

You have just examined several possibilities for direct connections. Note that some of them require a special connector. You cannot purchase a special connector, you need to make one from the following parts:

- Two DCE connectors (25-pin RS-232C female, HP part number 1251-0063 or equivalent).
- Two 1½-inch long machine screws (HP part number 2200–125) with nuts and lock washers. The screws should fit through the holes on both sides of the connectors, run through spacers, and permit plugging in the connector.
- Four .625-inch metal spacers (HP part number 0380-0010).
- Eight 1-inch long pieces of 24-guage electrical wire with ¼-inch of the insulation stripped from each end.

Setting aside the exact connections among wires, Figure 2-15 shows the appearance of a special conductor.

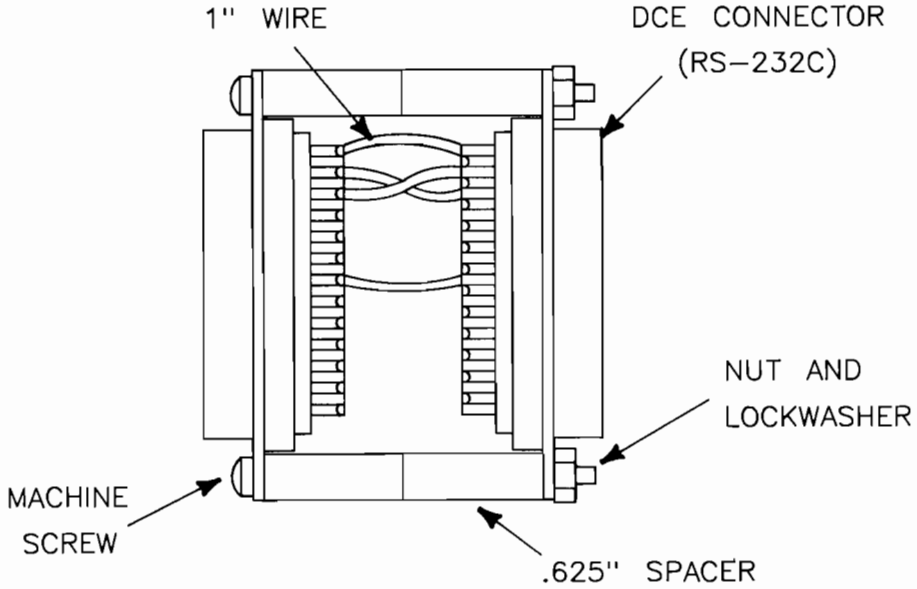


Figure 2-15. Illustration of a Special Connector

Should you need to make a special connector, use the appropriate illustration from previous figures to determine how to solder the wires to pins. If you have a Series 500 computer, the *HP 9000 Series 500 Configuration Information and Order Guide* has additional information about making direct connections.

Making a Decision and Continuing

At this point, you should decide to use a modem or direct connection.

- If you decide to use a **modem connection**, go to the next section.
- If you decide to use a **direct connection**, go to "Step 3: Installing An Interface Card".

Installing a Phone Line

Obtain an "outside" phone supplied by a local telephone company. The RS-232C interface card in the computer connects by cable to a modem, and a cable on the modem having a phone connector (jack) plugs into the phone outlet. Note these constraints:

- Do not get a line with "call waiting" capability because the tone generated by a phone company or PBX (Private Branch Exchange) causes a loss of modem carrier which terminates a *uucp* call.
- Having an extension connected through a switchboard or PBX can cause the following problems:
    - Remote systems cannot autodial your system because they would need to speak to a human operator.
    - Some digital PBXs cannot handle modem signals sent at 1200 baud. You would have excessive retries or be limited to 300 baud.
    - Outgoing calls from your system have a high probability of failing because the outside line of the PBX might be busy.

After you account for the phone line, move ahead to "Step 3: Installing An Interface Card".

Step 3: Installing An Interface Card

In any case (direct or modem connection), you can:

- Use a built-in RS-232C port on the backplane of your computer;
- Install an RS-232C interface card; or
- Install a multiplexer that has RS-232C connectors.

While you have these options, some might be better than others.

The next several sections describe the procedure. Read the section on shutting down your system and then find the later section that applies to your situation (you might need to work from a close approximation).

Possible Problems With a Built-in Port

Some computers, notably Series 300 computers, have a built-in HP 98644-like RS-232C port. While you can make a built-in port work, it can have characteristics (such as a 1-byte receiver buffer which is easily overriden at higher baud rates) that limit your options for transferring files. It is strongly recommended that you use a Data Communications interface card that provides for the full range of options for using RS-232C.

Shutting Down Before You Install Cards

Before you install any card in any computer:

1. **Shutdown** and **halt** your HP-UX system.
2. **Turn off the power** to your computer and all devices.

Then, skim through the following subsections to find your situation and install the appropriate card(s). As you install the cards, be aware of required cables, connectors, and possible jumping of lines.

Installing Series 500 Interface Cards

Install one of the following cards according to its documentation.

If You Have Problems

If you have any problems, determine that you installed the interface card according to its documentation. If this does not solve the problem, consult your HP Service Engineer or Sales Representative.

HP 27128A
A single channel asynchronous link to an RS-232C device.
Set switches 2 and 8 of the node address switches to ON (down).
Use a male cable (HP 27128A Opt. 001, Part No. 27122-63001).

HP 27130A/B MUX
Supports up to eight RS-232C compatible devices
Do not set any switches on the card.
There is no Option Number; use Cable Part No. 92219Q
Cannot be used for modem connections.

HP 27140A MUX
Supports up to six RS-232C/CCITT V.22 compatible devices
Do not set any switches on the card.
There is no Option Number; use Cable Part No. 92219Q

When you finish installing the card, do not turn on your system. Instead:

- If you will also install a modem, go to the section in this chapter called "Step 4: Configuring a Modem".
- If you will use a direct connection, go to Chapter 3, "Installing uucp".

Installing Series 200/300 Interface Cards

If you have a built-in RS-232C port, you can bypass this section unless you want to install an interface card specifically for *uucp*. It is recommended that you **not** use a built-in port. Instead, install one of the following RS-232C interface cards according to its documentation. The HP 98628A interface card (Data Communications card) is the recommended card. The built-in port, multiplexer card, and other cards have limitations.

If You Have Problems

If you have any problems, determine that you installed the interface card according to its documentation. If this does not solve the problem, consult your HP Service Engineer or Sales Representative.

HP 98628A

The recommended single channel asynchronous interface card
Use Cable Option 001
Use Cable Part No. 5061-4215
Connect all U3 Modem lines
Set switch 4 to 1
Set switch 5 to 0
Set Interrupt Level 3
Set Remote Keyboard to 0
Set Hardware Handshake to OFF
Set Modem Handshake to recommendation given for your modem
Select Async Protocol, Full duplex, and Modem

HP 98626A

Not the recommended interface card
Set all U3 switches to 1
Set Control Lines enabled
Set U20 Interrupt Level to 5
Have Remote Keyboard Jumper In
Connect all U3 Modem Lines

HP 98644A

This interface card is not recommended
Has no Option
Use Cable Part No. 13242N
Set Interrupt Level to 5
Have Remote Keyboard Jumper In
Enable Modem Line

HP 98642A MUX

Supports up to four RS-232C compatible devices
Has no Option
Use Cable Part No. 92219S
Set Interrupt Level to 3
Only Port #0 (one port) can be used for a Modem Connection
Set Remote Keyboard to 0 unless you have the Console on
Port #1, then set it to 1

When you finish installing the card, do not turn on your system. Instead:

- If you will also install a modem, go to the section in this chapter called "Step 4: Installing a Modem".
- If you will use a direct connection, go to Chapter 3, "Configuring uucp".

Step 4: Installing a Modem

If you install a modem, note the following items:

- The cable for the modem should have:
  - A 25-pin female connector on the end that connects to the modem.
  - A 25-pin male connector on the end that connects to the interface card.
- HP computers can use some modems not made by Hewlett-Packard Company. If you have problems installing a non-HP modem, ensure that you installed the modem according to its documentation. If this does not solve the problem, consult the Sales Representative or appropriate official at the company from which you obtained the modem.
- Find the section for the modem you have and install the modem according to its documentation.
- After you install the modem, go to Chapter 3, "Configuring uucp".

HAYES 1200 Baud Smartmodem

Use these steps:

1. Shutdown and powerdown your system.
2. Remove the front panel by prying out on the side tabs with a screwdriver. Then, pull the panel away from the modem.
3. Looking at the switches and working from left to right, set the switches to the left of the LEDs to: **up down up down up up up down**
4. Replace the front panel.
5. Connect the male RS-232C cable connector from the interface card or multiplexer to the female modem connector.
6. Connect the jack from the modem to the phone jack you had the phone company install.
7. Startup your system.

HP 37212A (Queensferry Modem)

Use these steps:

1. Shutdown and powerdown your system.
2. Pull the rectangular plastic harness forward from the modem case. You might need to pry the side tabs open with a screwdriver.
   Then, pull the similar back piece off the back of the modem.
3. Remove the six screws from the top of the case, and remove the two screws on either side.
   Then, remove the top cover.
4. Arrange the modem so the front is to your left. Locate the eight dip switches just behind the front panel and set them to: **up up up up up up up down**
5. Replace the top cover, screws, and harness.
6. Connect the male RS-232C cable connector from the interface card or multiplexer to the female modem connector.
7. Connect the jack from the modem to the phone jack you had the phone company install.
8. The front panel has four switches. Set the HS (High Speed Data) switch to IN. Set the other switches (ALB, RDL, and DATA) to OUT.
9. Startup your system.

Unsupported Modems

You can often use an unsupported modem with *uucp*. In some cases, you can receive incoming calls on an unsupported modem that does not provide auto-dial functions. Just remember that the term, **unsupported**, means you work entirely on your own and get no help from Hewlett-Packard.

If you do use an unsupported auto-answer modem, set the standard switches as follows:

Switch Function	Suggested Setting
DTR	Set this switch to monitor the line without forcing the line high. HP-UX asserts Data Terminal Ready when it is ready to accept a login.
Commands	If you do not have an autodial routine for your modem, which is explained shortly, disable the commands option on the modem.
Result Codes	If you do not have an autodial routine for your modem, disable this option on the modem.
FDX/HDX	Set these switches for full duplex.
Auto/Answer	Set the switch to allow the modem to **answer** an incoming call and transmit its **answer carrier**.
DCD Connect	Set this switch to let the modem assert DCD (Data Carrier Detect) when originating a call and detecting the answer carrier on the remote modem.

Beyond this, if you want an unsupported modem to dial out of your system, you will have to modify the routine that does the auto-dialing. The source code for this routine is in:

```
/usr/lib/dialit.c
```

Before you modify this code, copy it to:

```
/usr/lib/dialit.old
```

Then, edit `dialit.c` as required (the section called "Dialit.c" in Chapter 5 describes this file). Finally, compile the file to create a new version of:

```
/usr/lib/dialit
```

You have to know what needs to be edited. HP cannot help you with this. In any case, *uucp* uses this routine to do auto-dialing.

Configuring uucp 3

After you install your hardware, configure *uucp* by working through the steps described in the following sections.

LAN Is Optional

The Optional Step reconfigures your kernel so you can use LAN (Local Area Network). While LAN is not required to use *uucp*, users often employ LAN to transfer files among systems linked to a hub (centralized) system and use *uucp* to transfer files among remote hubs. Therefore, the procedure for reconfiguring your kernel so it uses LAN is included here for convenience. Before working through the Optional Step, you need to install the hardware for LAN in accordance with the documentation for LAN devices and interface cards.

If you have serious problems that you cannot solve during this step, read the *LAN Node Manager's Guide* and the *LAN User's Guide.* Then, when you get ready, go on to "Step 1: Setting the System Node Name".

Optional Step: Reconfiguring the Kernel for LAN

To use *uucp* **and** LAN, you need to reconfigure your kernel for LAN after you install appropriate hardware.

As a system administrator, you know that reconfiguring a kernel can cause problems unless you account for the existing parts of your system. **Therefore, before you complete the next task**:

- **Make sure** you understand your existing system (e.g. know what is in `/etc/conf/dfile`, `/etc/rc`, and so on).
- **Accommodate** existing customizations in your `dfile` (e.g. dividing swap space among discs) when you do the reconfiguration.

Editing Your dfile

Your `dfile` needs to have the appropriate device drivers. The following two columns show the dfiles for a full system and a full system with LAN. You probably used one of these dfiles when you originally configured your HP-UX system, and you might have added some customizations.

Full System dfile	**Full System `dfile` with LAN**
cs80	cs80
amigo	amigo
tape	tape
printer	printer
stape	stape
srm	srm
ptymas	ptymas
ptyslv	ptyslv
hpib	ieee802
gpio	ethernet
ciper	hpib
rje	gpio
* cards	ciper
98624	rje
98625	* cards
98626	98624
98628	98625
98642	98626
	98628
	98642

Having LAN adds the drivers named **ieee802** and **ethernet**. As an example of a customization, your **dfile** might include the following two lines, which cause swap space to be divided among HP 7946 and HP 7945 disc drives.

```
swap cs80 e0000 -1
swap cs80 e0100 -1
```

You need to become the root user and edit your dfile, making sure it has:

- The drivers for the full system.
- Your customizations.
- The two drivers shown for the full system with LAN.

When you finish editing the **dfile** and save it, reconfigure your kernel.

Reconfiguring Your Kernel

Having edited your dfile, remain the root user and reconfigure your kernel by executing the commands shown in the following list (execute each line in order— the entire process takes a few minutes— do not type the comment at the end of each line):

```
cd /tmp                    (* You need to be in a dir. other than root *)
config dfile               (* Creates a make script *)
make -f config.mk          (* Makes a new kernel, hp-ux, under /tmp *)
mv /hp-ux /SYSBCKUP        (* Saves the old kernel *)
mv hp-ux /hp-ux            (* Moves the new kernel into the root *)
```

When you complete these minor steps, type:

```
reboot Return
```

to reboot the reconfigured HP-UX system. Continue by making a device file for LAN.

Making a Device File for LAN

If a device file for LAN does not exist, become the root user and make one. Execute both of the following commands (they assume a factory default Select Code of 21, which is 15 hex). Only one line is necessary, but having both of them covers more possibilities and does not hurt.

```
mknod /dev/ieee c 18 0x150000
mknod /dev/ethernet c 19 0x150000
```

The Number Is Not Important

The driver number does not matter, except when you use LLA. Although 18 is ieee and 19 is ethernet, there is really only one driver.

Continue by starting the LAN daemons.

Start Up Daemons

Still as the root user, edit `/etc/rc` so it sets up LAN and a host name when you powerup or reboot the system.

Assuming you have a host name such as **hpfcba** and an internet address such as **i0xc0069901** study `/etc/rc` and edit (possibly insert) the following lines:

```
.
.
hostname hpfcba
.
.
PATH=/bin:/usr/bin:/etc
npowerup -m250000 -nhpfcba.fc.hp -i0xc0069901 /dev/ieee
rfadaemon
nftdaemon
vtdaemon
ptydaemon
.
.
```

When you first look at `/etc/rc`, you might see `hostname unknown` . You must replace `unknown` with the system name (node name or host name) that was set up for your system when you met with the SAs.

In the `npowerup` line, note the following:

- The `m250000` specifies the memory requirement. The example has enough to accomplish all Network Services/9000 tasks. You can specify more memory if you have it. To get more information, see section called "Network Memory Utilization" in the *Network Services/9000: LAN Node Manager's Guide.*
- The `nhpfcba.fc.hp` is the complete node name for your system. This is an item you agreed upon when you met earlier with other system administrators.
- The `i0xc0069901` is the internet address. This is an item you agreed upon when you met earlier with other system administrators.

When you finish editing `/etc/rc`, reboot your system to get the daemons running.

Set up and Test Remote File Access

The following commands have some assumed names. Substituting the names you determined earlier with other system administrators, execute the following commands:

`mkdir /net`	Makes a directory for networking.
`cd /net`	Changes to the new directory.
`mknod remsys2 n remsys2.tj.hp`	Makes a node for a remote system called `remsys2` with its hypothetical complete node name (actually, you check your notes and use names according to your agreements with the remote system administrators).

Then, if your user name is `sueb` and your password is `piper1`, you can test connecting with the remote system by executing:

`netunam /net/remsys2 sueb:piper1`	If you stop at typing the colon, the system prompts for a password.

To differentiate between your local system and the remote system:

`ls /net/remsys2/users`	lists the contents of `/users` on the remote system.

whereas:

`ls /users`	lists the contents of `/users` on your local system.

Switching to a more general context, the following lines let you test NFT (Network File Transfer) for a file named `file1`.

```
dscopy localnodename#localusername:localpassword#/users/demos/file1
remotenodename#remoteusername:remotepassword#/users/demos/file1
```

Step 1: Setting the System Node Name

During the time you met with other system administrators, you determined the system node names. The principle was to use unique, suggestive node names (e.g. if your name is Sue Ellen and you work in the Research and Development lab at Acme Corporation, your system name (hostname) might be **acrdse** for "Acme Corporation Research and Development Sue Ellen).

If you have not yet done so, and assuming your hostname is **acrdse**, edit **/etc/rc** so it contains the line:

```
.
.
hostname acrdse
.
.
```

This line may have been **hostname unknown** (i.e. the default value).

Step 2: Gathering Information About Systems

At this point, gather the information you originally obtained from the remote system administrators (i.e. the information about the remote systems you want to call). The following table shows some examples each of which shows the node name; phone number; login name; password; baud rate; and time when you can access a remote system.

System	Phone Number	uucp Login,Password	Baud Rate	Accessibility
acrdjr	303-223-2067	ladonna,Sundown1	1200	Any time
norad	303-555-1212	casper,MiSSle9	1200	Mon-Fri
hpopus	503-593-2244	mikek,123gonow	300	Weekends

Using this example to suggest some other information, you might need:

1. The names, and so on, of systems you allow to call into your system.
2. The names, and so on, of systems that can use your system for *uucp* forwarding to other remote systems.

When you have this information, go to "Step 3: Creating Device Files".

Step 3: Creating Device Files

In this step, you create device files using the typical techniques. The procedures assume you know how to do this, and therefore, you see examples without a discussion of general situations.

Two outgoing device files must exist in `/dev` before you can make outgoing phone calls from your system. The file names begin with `cul` for modem and `cua` for autodialer situations and include two digits. While their names differ, the two device files point to the same physical device (i.e. an interface card or built-in port).

An incoming device file must exist before you can receive incoming *uucp* or user phone calls. The file name begins with `tty` , and a device file must exist for each incoming port (remote system).

Shortly, you will see file names such as `cul01` , `tty00` , and so on. Your situation determines the two digits:

00 Use for incoming device files and the HP 27130A MUX card (it has no modem capabilities).

01 Use for outgoing device files.

02 Use for an incoming device file if your modem obeys CCITT protocols.

03 Use for an outgoing device file if your modem obeys CCITT protocols.

In cases where you need to specify the driver number when you make device files for `cul` , `cua` , and `tty` device drivers, use the following driver numbers:

31 Use for the HP 27128A serial interface card and the HP 27130A MUX card on Series 500 computers.

29 Use for the HP 27140A MUX card on Series 500 computers.

1 Use for serial interface cards on Series 200 and 300 computers.

While you use *mknod* to create device files, the file can differ among Series 200, 300, and 500 computers. Therefore, find your situation among the next three subsections (noting the 200, 500, 300 designations) and make the necessary files.

Series 200 Computers

On Series 200 computers up to the 2.3 release of HP-UX, the device files map to their Select Codes in increasing order. Therefore, if you had a system with two datacomm cards and neither of the cards is used as the system console, then the names of the device files would be:

cua00, cul00, and tty00	For the HP 98628A card having the lower Select Code.
cua01, cul01, and tty01	For the HP 98628A card having the higher Select Code.

To make incoming and outgoing device files, become the root user, change the directory to / , and execute each of the following commands in order (use **/etc/mknod** if **/etc** is not in **PATH**):

```
mknod /dev/cua00 1 128
mknod /dev/cul00 1 128
mknod /dev/tty00 1 0
mknod /dev/cua01 1 129
mknod /dev/cul01 1 129
mknod /dev/tty01 1 1
```

Use ls -l /dev if you want to verify that the files exist.

Series 500 Computers

On releases prior to 4.0, incoming and outgong lines must be on separate phone lines, modems, and datacomm cards. Otherwise, incoming and outgoing calls can occur over the same hardware.

Create outgoing **cul** and **cua** device files if you want your system to originate calls. Create a **tty** device file if you want your system to receive calls. Make outgoing and incoming device files character-special files. Use driver number 31 for the HP 27128A interface card; and use number 29 for the HP 27140A MUX card.

If your system will have incoming and outgoing calls, set the least-significant bit of the outgoing device files. If your system makes only outgoing calls, do not set the bit.

The following example sets up device files for an HP 27128A card at Select Code 4.

```
/etc/mknod /dev/cul00 c 31 0x040001
/etc/mknod /dev/cua00 c 31 0x040001
/etc/mknod /dev/tty00 c 31 0x040000
```

The following example sets up device files on port 2 of an HP 27140A MUX card at Select Code 3.

```
/etc/mknod /dev/cul02 c 29 0x030201
/etc/mknod /dev/cua02 c 29 0x030201
/etc/mknod /dev/tty02 c 29 0x030200
```

You also need to load any necessary optional drivers not installed when HP-UX was installed. Assuming an HP 27140A card (requires the HP27140.opt driver), use the following procedure:

1. Become the super-user and type:

   osck -v /dev/sys_disc Return

   to see if the driver exists. If you do not see the driver, continue with this procedure.

2. Determine your Series 500 computer (e.g. 97078C):

   a. Model 520 single-user system is 97070C.

   b. Model 520 multi-user system for 16 users is 97080C.

   c. Model 520 multi-user system for 32 users is 97078C.

   d. Model 530/540/550 single-user system is 97079C.

   e. Model 530/540/550 multi-user system for 16 users is 97089C.

   f. Model 530/540/550 multi-user system for 32 users is 97088C.

3. Then, type:

   oscp -a /system/970*xx*A/HP27140.opt /dev/sys_disc Return

   where **-a** appends to an existing operating system from a list of files and puts the resulting system in the boot area. In **/system/970*xx*A/HP27140.opt** , the **HP27140.opt** driver is copied to the boot area of the operating system and the *xx* is the last two digits taken from the number above for your Model number (use the 89 taken from 97089C for a Model 530/540/550 multi-user system for 16 users).

4. Use the command you executed in Step 1 to verify that the driver exists. Then, reboot HP-UX to activate the **HP27140.opt** driver.

For a different interface card, use the same procedure, substituting the system and driver names.

Series 300 Computers

Series 300 device files are like those used for Series 500 computers, but driver number 1 is the appropriate driver for the Series 300.

For example, to create incoming device files for a direct connection using an HP 98628A datacomm card that you set to Select Code 23, which is not the default Select Code, execute the following commands:

```
/etc/mknod /dev/cul00 c 1 0x170001
/etc/mknod /dev/cua00 c 1 0x170001
/etc/mknod /dev/tty00 c 1 0x170000
```

In the minor number, remember that 23 is 17 hex.

For another example, to create outgoing device files for a modem on port 0 of an HP 98642A MUX card set to Select Code 20, execute the following commands:

```
/etc/mknod /dev/cul01 c 1 0x140001
/etc/mknod /dev/cua01 c 1 0x140001
/etc/mknod /dev/tty01 c 1 0x140000
```

Note that in making device files, you specify the Select Code in the minor number. In some cases, you can determine the Select Code by setting switches; but when you use a built-in RS-232C on a Series 300 computer, the Select Code is 9.

Step 4: Starting a Getty for an Incoming Device File

If your system is to be passive (i.e. receive calls initiated by a remote system), start a getty for the incoming device file (e.g. `tty04`). To do this, add the following line to `/etc/inittab`:

```
04:12:respawn:/etc/getty -t 240 tty04 3
```

The entries have the following meanings:

04 — This is the last two characters of the device file on which you start the getty.

12 — The `12` goes with the `:respawn:/etc/getty` and represents the init states for which the process is allowed to run (i.e. run-levels 1 or 2). When you boot a system, it enters the init state given in the first entry in the `/etc/inittab` file. The superuser can specify other states by executing the command:

`/etc/init` *init_state*

where *init_state* is any of **0-6**, **S**, or **s**. If the current init-state matches any of the states given in the second field of an entry, that process starts if it is not already running.

-t 240 — The `240` is the recommended timeout value (the `-t` option). The value specifies that when a login is attempted and is not successfully completed within 4 minutes, the connection should be dropped.

tty04 — This is the name of the incoming device file for the getty.

3 — The final `3` specifies the getty type. In this case, the getty will start at 1200 baud. If this does not work for an attempt to login, the getty will try 300, 150, and 110 baud. If none of these rates work, the getty will try 1200 baud again. The getty process cycles baud rates whenever it receives a BREAK character.

On the other hand, when you send a file (your system is active and the remote system is passive), the remote system must have previously started a getty for its incoming device file. You need to coordinate this with the SAs for remote, receiving systems.

Step 5: Setting Up a uucp Login

To set up a login for *uucp*, edit **/etc/passwd** so it contains lines like those that follow, allowing that you use names and numbers related to your system:

```
root:McSTk.7AUh.fo:0:1: System Admin :/users/root:/bin/sh
who::6:7::/:/bin/who
date::7:7::/:/bin/date
sync::20:7::/:/bin/sync
uucp::5:5: UUCP LOGIN :/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

The **/usr/spool/uucppublic** is the usual login directory for *uucp*, and this login runs **/usr/lib/uucp/uucico** instead on running a login shell.

Assign a password for *uucp* by executing:


```
passwd uucp [Return]
```

For increased security, you can restrict *uucp* logins to named systems as follows:

1. In the *uucp* line in **/etc/passwd** , set the password field to * or **NO LOGIN** .
2. For each calling system, copy the *uucp* line into **/etc/passwd** . Change the first field to the node name of the remote system (or to some other unique user name). Keep the same group ID, but assign a unique user ID, and assign a password for each *uucp* login.

Verifying Your Steps

You can verify being able to answer *uucp* calls by logging out and logging in as **uucp** with the correct password. The system should answer with **Shere** , which means "Slave Here" and indicates that *uucp* is ready to swap files with you. If you do nothing for several minutes, *uucp* gives up and you get a new login prompt.

Step 6: Checking the Modem Type

Ignore this step if you use a direct connection. Otherwise, for a modem to work, check your type of modem and map this modem to a string in **/usr/lib/uucp/dialit.c** that *uucp* can understand. The currently supported types of modems and strings include:

Modem	Character String
Hayes_Smartmodem	ACUHAYESSMART or ACUHP92205A
HP 35141	ACUHP35141A
VADIC	ACUVADIC3450
VENTEL212	ACUVNETEL212
bridge_cs100	ACUBRIDGECS100

Using this information, edit **dialit.c** so it contains the appropriate string(s).

Step 7: Setting Up L-devices

Beginning with this step, you will modify several files in `/usr/lib/uucp` , so change to this directory by executing:

```
cd /usr/lib/uucp Return
```

Get the Information You Collected About Systems

At this point, get the information you agreed upon with other SAs and use it as you edit the files described in this and the next several sections. Look at the examples to see what to do and then use your information when you edit a file.

In the file named `L-devices` , you specify the:

- modem type according to `dialit.c`;
- modem line (cul); autodial line (cua); and
- data transfer rate.

For a Hayes Smartmodem connected to the device files named `cul02` and `cua02` , you would add:

```
ACUHAYESSMART cul02 cua02 1200
```

or

```
ACUHP92205A cul02 cua02 1200
```

to `L-devices`. You can adjust this line (or lines if you have more than one modem) to suit your type of modem, device files, and baud rates.

Step 8: Dialing Out With cu

At this point, with `L-devices` set up, you should be able to use *cu* to call a remote system. For example,

```
cu -s1200 3039991212
```

should call the remote system at the specified phone number (303 999 1212) at the specified baud rate (1200).

Cu checks in `L-devices` to determine the type of modem for the connection. If someone is already using this modem, a LOCK FILE keeps you from accessing the same modem. To see if the lock file exists, execute:

```
ls /usr/spool/uucp
```

If such a file exists for the preceding example, you will see a file named `LCK.cul02`.

What If You Cannot Dial Out With Cu?

This is your first critical point in setting up *uucp*. Your system is not correctly installed and configured until you can use *cu*. Therefore, if *cu* did not work in the preceding example (modified for your situation), work through the immediately following steps. If the steps do not help you and you cannot get it to work, see other SAs using *uucp* or call your HP Service Engineer.

In addition, you can try these procedures:

1. Use `ps -ef` to see if another person is running *cu*. If they are, wait awhile and try again.
2. Make sure the switches on the modem have the correct settings.
3. If there is a device lock-file in `/usr/spool/uucp` and no one is running *cu* or *uucp*, remove the lock file.
4. Check the entries in `/usr/lib/uucp/L-devices` for typing errors or having parameters in the wrong order.

5. If a getty is running on a Series 500 (version 4.0 or newer), ensure that the least-significant bit is set for the `cul` and `cua` device files. Make sure the least-significant bit of the `tty` device is not set. For `ls -l /dev/*02` , a proper listing looks something like:

```
crw-rw-rw- 1 root other 31 0x020001 Oct 14 12:14 /dev/cul02
crw-rw-rw- 1 root other 31 0x020001 Oct 14 12:14 /dev/cua02
crw-rw-rw- 1 root other 31 0x020000 Oct 14 12:15 /dev/tty02
```

6. Execute `ps -ef` and look for question marks. If a getty is running for your modem and no one has dialed in, a `?` should appear in the `tty` entry for that getty. If a terminal number is associated with this port, the modem is probably configured to assert DCD (Data Carrier Detect) even when no carrier is present. Change the modem configuration to assert DCD only when the other end of the connection has a carrier.

7. For Series 500, you can try to send data directly to the modem with *aterm*. The *HP-UX Concepts and Tutorials Vol. 5* manual documents *aterm*. You must have a serial driver on the system, not a serial terminal driver. The *aterm* device file must use driver number 19.

Taking Time to Use Cu

If *cu* works, you should continue with Step 9. Later, you might want to make additional use of *cu*. If you do want to use *cu* again as a utility in its own right, this section provides additional information.

The *cu* ("call UNIX") utility 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 odd or even 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 appear on your CRT as the connection is made:

```
autodialing - please wait
Connected
login:  <this comes from the remote system>
```

If the autodial failed, the message:

```
Connect failed: autodial failed
```

indicates why the connection failed.

You need to know how to login on the remote system 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 odd or even 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.

Step 9: Setting Up L.Sys and L-dialcodes

At this point, edit `/usr/lib/uucp/L.sys` to determine how *uucp* can automatically reach remote systems and whether remote systems can log into your system. Then, edit `/usr/lib/uucp/L-dialcodes` and add information that correlates with the information you put in `L.sys`.

Editing L.sys

The general syntax for a line in `L.sys` is:

```
sysname time[,retry] device baud-rate phone login-information
```

Some examples of lines, which you adjust according to your agreements with other SAs, look like:

```
acrdse Any,5 ACUHAYESSMART 1200 503-999-1212 login:-^M-login: norad sword: yZzYx
hpfcarf Any,5 cul01 9600 cul01 login:-EOT-login:-BREAK-login: uucp sword: hiccup
rosebud Never
```

Note that each line has up to 9 fields (the following subsections describe them).

Remote Host Name (acrdse)

The field specifies the name that the remote system has assigned to itself via `hostname` in `/ect/rc`. You must have an entry such as `acrdse` or `hpfccarf` for every remote system.

Communication Times (Any)

The field specifies the days of the week and the times of the day that you can call out to a remote system. If you do not want to call a system, but want the system to call your system, set the field to `Never` and leave all remaining fields blank. Using `Any` permits calling at any time. Otherwise, specify a time such as `Fri`.

Retry Times (5)

The field specifies the number of minutes that pass before your system can retry to make a connection. When an attempt to call a remote system fails, a STATUS FILE is created; and this lock file keeps users on your system from being able to retry making a connection until the specified minutes pass. Note that *uucp* has no auto-retry mechanism. Therefore, the retry time specifies the earliest time for attempting a connection, not necessarily the earliest time another try will be made.

Modem Type (ACUHAYESSMART)

The field specifies the string that maps to the type of modem you use to make a call. When you use a direct connection, you specify the modem device line (the `cul` line).

Data Transfer Rate (1200)

The field specifies the baud rate for communication, typically 1200 or 300 baud.

Phone Number (503-999-1212)

The field specifies the phone number of the remote system you try to call. Besides a phone number such as `303-223-2053`, you can append an [v] to the number (wait for a secondary dialtone) and - (5-second pause). The routine named `dialit` maps the correct characters for your modem. If you use a direct connection, repeat the modem device (cul line) in this field.

The phone number can also have a prefix, `f` or `g` , to indicate which protocol to use. The default protocol is `g` , which is used with dial-up phone lines and other less reliable connections. Using `g` sends data in 64-byte packets with acknowledgements after each packet. Use `f` only with very reliable connections such as hardwired direct connections or X.25. This protocol sends an entire file followed by a checksum. On an error, the entire file is retransmitted! Two examples of specifying protocol are:

```
g/9991234
fg/cul02
```

Preliminary Prompt/Response (Login:-^M-)

The field specifies the login prompt you expect and your response if it is received. You should have a fresh login to begin *uucp*, so you need to "kick" the remote system with several characters. In the above example, a carriage return is used to force the remote system to respond with a fresh login prompt. The carriage return, which is entered when you call a system by typing [CTRL]-[M] [Return], is delimited in the line by a dash on either side. When you actually call a remote system, do not type a space after the initial prompt, `Login:`.

Login Prompt/Response (Login: norad sword: yZzYx)

The field specifies the true login prompt your system looks for after the preliminary prompts have been received and the responses have been sent. The field should contain `login:` and you should note that there is a space following the colon. Your system must then supply the *uucp* user name (hostname) to log into the remote system.

Editing L-dialcodes

This file specifies telephone prefix translations. When you use a modem connection to call a remote system, your system looks in `L.sys` for the remote system phone number. If you used a string such as `Boston` in the phone field for the remote system, your system looks in `L-dialcodes` to see the actual number for Boston. Therefore, entries in `L-dialcodes` might look like this:

```
Boston 999-1234
Denver 223-2063
```

Step 10: Setting Up USERFILE

At this point, edit **/usr/lib/uucp/USERFILE** so it performs three functions:

1. Determines whether your system should call a remote system back after it has called your system. This provides a security measure to make sure a system is the system it claims to be.
2. Determines the file access that should be granted to a remote system.
3. Controls which files can be copied from the local system to remote systems.

You need to think about the entries because they must be coordinated with the hostnames of the systems you want to call and where you want to put files in those systems.

An example might look like this (you must make the adjustments for your system):

```
,               /usr/spool/uucppublic /dev/null
uucp,dasher     /
uucp,norad      /usr/spool/uucppublic /public /dev/null
,               c       /
```

The next few subsections describe the 4 fields for lines in the USERFILE.

Userfile Search 1: Check for Callback

The first field (e.g. , and **uucp**) specifies the callback field. When a remote system calls your system, the two systems exchange node names. *Uucp* opens USERFILE and sequentially looks through the lines. If the second field (a system name such as **dasher**) matches the name supplied by the other system, the third field is checked. If this field specifies **c** , your system must call back the remote system. *Uucp* notifies the remote system that the connection will be taken down, and then both systems hang up.

Be sure the callback option is not specified in both USERFILEs of the two systems that communicate. If this happens, both systems call each other indefinitely.

Your system then looks for the remote system name in **/usr/lib/uucp/L.sys** and tries to call the system back.

If the name provided by the remote system does not match any of the entries, the line with the first blank node name field is used. Then, the callback field of this line is checked.

For example, if a remote system named **comet** logs into your system with the username **uucp** and provides the correct password, *uucp* checks USERFILE for an entry with **comet** in the second field. The search would end when *uucp* reaches the 4th entry. If there is no **c** in the 3rd field, there is no need to hang up and call **comet** back— so the search is completed.

On the other hand, if a system named **norad** logs into your system with the username **uucp** and the same password, *uucp* checks USERFILE for an entry with **norad** in the second field. Since there is such an entry, and if the callback field contains a **c** , your system notifies the remote system that it will hang up and call **norad** back. On not finding **norad** in **L.sys** , there will be no attempt to call back.

Userfile Search 2: Check Remote Permission

When a remote system tries to send a file to or from your system, *uucp* checks USERFILE to see if the system has the necessary file permission by opening USERFILE and checking the username employed by the remote system to log in against the first field of each line. Note in the above USERFILE that access permission can be given to several directories (e.g. **/usr/spool/uucppublic** and **/dev/null**).

Be aware also that *uucp* file access requires read and write permission for everyone.

Userfile Search 3: Check Local Permission

To copy a file from your system to a remote system, *uucp* must grant access to the file on the local system. To promote this and to prevent problems, your USERFILE should contain the last line in the above example:

```
,      c      /
```

The line allows users to copy files to another system that have read-write permission by everyone. The **c** field is used to make sure that unknown system cannot use this line to access unrestricted directories.

Step 11: Setting Up FWDFILE and ORIGFILE

To use *uucp* forwarding, you need to create FWDFILE and ORIGFILE. FWDFILE contains the nodemanes of all systems that your system can forward to or through. ORIGFILE contains the nodenames of all systems that can forward through your system. Both files have the same format— a list of nodenames. For example, FWDFILE could look like this:

```
norad
hpopus
```

Step 12: Try A uucp Call

After completing all the preceding steps, it might be nice to transfer a file to a remote system. Assuming a file named `my-junk` in your current directory and a remote system named `norad`, execute:

```
uucp my-junk norad!~uucp/my-junk
```

If the transfer does not work, you have some options:

- Visit with the SA for the remote system and see if that person can help.
- Revisit the steps you completed so far to ensure that you accounted for your system and the specific remote system you called.
- Check with your HP Sales Engineer.

At this point, you could also plow through this manual, hoping to find the information you need. In reality, it is probably better at this point to seek expert help. Then, read the material in later chapters about using *uucp* after you are able to make a simple transfer to a remote system.

If you want to examine some information before you seek help, the next step explains some debugging procedures.

Step 13: Debugging uucp

You can encounter problems and need to do some debugging. Try the following steps:

1. Examine the last few lines of `/usr/spool/uucp/LOGFILE`. This file keeps track of what happens. (An example is shown soon.)
2. Examine the contents of `/usr/spool/uucp`.
3. Examine `/usr/spool/uucp/ERRLOG`.
4. Examine `/usr/spool/uucp/DIALLOG`.
5. Execute *uucico* with debugging output.

Collectively, these files provide information that can point to problems. By studying the problems, you can retrace what you did in earlier steps and make necessary corrections.

The LOGFILE

The LOGFILE keeps a record of locally and remotely originated *uucp* transactions. For example, to see the last 20 lines of LOGFILE, you could execute:

```
tail -20 /usr/spool/uucp/LOGFILE
```

These lines (with fields F1 through F4) might look like this:

```
F1   F2     F3                F4
--------------------------------------------------------------

tomh hpfcla (1/1-8:42-28677) NO CALL (RETRY TIME NOT REACHED)

root hpopus (1/2-5:28-11853) CAN NOT CALL (SYSTEM STATUS)

pmp hpfcla (1/2-11:43-26621) FAILED (AUTODIAL)
pmp hpfcla (1/2-11:43-26621) FAILED (call to hpfcla)

root hpabcl (1/3-16:23-27259) WRONG TIME TO CALL (hpabcl)
root hpabcl (1/3-16:23-27259) FAILED (call to hpfcla)

root hpcnoa (1/3-16:23-27219) SUCCESSFUL (AUTODIAL)
root hpcnoa (1/3-16:23-27219) SUCCEEDED (call to hpcnoa)
root hpcnoa (1/3-16:23-27219) OK (startup)
root hpcnoa (1/3-16:23-27219) OK (conversation complete)

uucp hpfcmc (1/14-13:31-1382) OK (startup)
uucp hpfcmc (1/14-13:31-1382) REQUESTED (S D.hpfcmsB8451 news)
news hpfcmc (1/14-13:31-1382) COPY (SUCCEEDED)
news hpfcmc (1/14-13:31-1382) REQUESTED (S D.hpfcmcX8449 X.hpfcmcX8449  news)
news hpfcmc (1/14-13:31-1382) COPY (SUCCEEDED)
news hpfcmc (1/14-13:31-1382) OK (conversation complete)

tomh hpopus (1/15-16:57-1905) SUCCESSFUL (AUTODIAL)
tomh hpopus (1/15-16:57-1905) SUCCEEDED (call to hpopus)
tomh hpopus (1/15-16:57-1905) OK (startup)
tomh hpopus (1/15-16:57-1905) REQUEST (R ~uucp/COOKBOOK /tmp/cookbook/tomh)
```

The four fields in LOGFILE provide the following information:

1. The username of the local system that originated the *uucp* call (e.g. `tomh` or `pmp`).
2. The username of the remote system that received the *uucp* call (e.g. `hpfcla` or `hpopus`).

3. The time when the action reported in LOGFILE occurred (e.g. `1/15-16:57-1905`).

4. The status report on the action with a report of the success or failure of the action (e.g. `SUCCESSFUL (AUTODIAL)`).

Of the four fields, the last field has the information that alerts you to problems. The next few subsections indicate what the messages tell you.

Retry Time Not Reached

In the line for the first call:

```
tomh hpfcla (1/1-8:42-28677) NO CALL (RETRY TIME NOT REACHED)
```

the status report means that, while attempting to communicate with a remote system, a STATUS FILE named `STST.hpopus` was created in `/usr/spool/uucp`. The problem is that this file must "age" to a point specified in RETRY TIME (the 3rd entry in `L.sys`) before the local system can attempt another call to the remote system. Removing `STST.hpopus` will eliminate the problem.

System Status

The line for the second call:

```
root hpopus (1/2-5:28-11853) CAN NOT CALL (SYSTEM STATUS)
```

indicates that another call to the remote system has already started. You can check this with `ps -ef` , looking to see if *uucico* is running.

Any scheduled *uucp* or *mail* transfers will automatically go through the connection that is already in use.

If no call is already in progress, check for a SYSTEM LOCK FILE named `LCK.hpopus` (in this case) in `/usr/spool/uucp` and remove any such files not being used by *uucico*.

Failed Autodial

The third attempted conversation:

```
pmp hpfcla (1/2-11:43-26621) FAILED (AUTODIAL)
pmp hpfcla (1/2-11:43-26621) FAILED (call to hpfcla)
```

failed because your system was not able to dial out on your modem. Possible reasons include:

- The modem is not correctly configured; check its switch settings.
- You specified a type of modem in `/usr/lib/uucp/L.sys` that does not match your modem. Check in `dialit.c` to see that you specified a valid type of modem.
- The modem dialed incorrectly. The VENTEL modems, for example, sometimes dial random numbers regardless of the number given to them. When an autodial fails, unplug the modem for a few minutes, plug it in again, and try dialing via *cu*.

Wrong Time to Call

The fourth attempted conversation:

```
root hpabcl (1/3-16:23-27259) WRONG TIME TO CALL (hpabcl)
root hpabcl (1/3-16:23-27259) FAILED (call to hpfcla)
```

means that you attempted to call at a time not allowed in `/usr/lib/uucp/L.sys` by the remote system in the second field. If you start a successful conversation later, the system will also process this call.

Successful Dial - No Data Exchanged

On the fifth attempted conversation:

```
root hpcnoa (1/3-16:23-27219) SUCCESSFUL (AUTODIAL)
root hpcnoa (1/3-16:23-27219) SUCCEEDED (call to hpcnoa)
root hpcnoa (1/3-16:23-27219) OK (startup)
root hpcnoa (1/3-16:23-27219) OK (conversation complete)
```

the local system could not reach the remote system, log in, and start a *uucp* conversation. The messages mean that, after the two systems negotiated a protocol, they indicated to each other that there were no files to transfer and terminated the connection.

Successful Dial - Data Sent

On the sixth attempted conversation:

```
uucp hpfcmc (1/14-13:31-1382) OK (startup)
uucp hpfcmc (1/14-13:31-1382) REQUESTED (S D.hpfcmsB8451 news)
news hpfcmc (1/14-13:31-1382) COPY (SUCCEEDED)
news hpfcmc (1/14-13:31-1382) REQUESTED (S D.hpfcmcX8449 X.hpfcmcX8449  news)
news hpfcmc (1/14-13:31-1382) COPY (SUCCEEDED)
news hpfcmc (1/14-13:31-1382) OK (conversation complete)
```

the local system successfully logged into the remote system and transferred two files from the local system to the remote system. The `S` in parentheses indicates that the temporary files used by *uucp* were sent from the local system to the remote system.

Successful Dial - Data Received

The seventh attempted conversation:

```
tomh hpopus (1/15-16:57-1905) SUCCESSFUL (AUTODIAL)
tomh hpopus (1/15-16:57-1905) SUCCEEDED (call to hpopus)
tomh hpopus (1/15-16:57-1905) OK (startup)
tomh hpopus (1/15-16:57-1905) REQUEST (R ~uucp/COOKBOOK /tmp/cookbook/tomh)
```

the local system successfully logged into the remote system. The remote system then transferred the file, `COOKBOOK`, from the default *uucp* login directory to the user's current directory. The `R` in the file transfer indicates that the file was received from the remote system.

Debugging With ERRLOG

The file named `/usr/spool/uucp/ERRLOG` can provide *uucp* debugging information. A message logs into this file when a read or write error occurs while executing *uucp* or when a file cannot be opened.

When you cannot get through with *uucp*, note the time the failure occurred. Then, examine the entry that corresponds with this time.

When a device file cannot be opened, check the permission mode of the file. Device errors also occur when the `x` (search) bit is not set on the `/dev` directory. This is probably the problem when you cannot dial out to another system with *cu* and cannot dial out with *uucp*.

Running uucico With Debugging Output

Uucp uses *uucico*; and this fact lets you use *uucico* for debugging.

For example, suppose you expect a remote system named **hpopus** to be able to answer a call. You could execute:

```
/usr/lib/uucp/uucico -rl -shpopus -x5
```

where:

- **-rl** means you want *uucico* to run in master (originate) mode at the start of a conversation.
- **-shpopus** specifies the system from **L.sys** that you want to contact. The parameter is **-s** followed by the system name.
- **-x5** indicates the level of debugging information to be given to you (the range is from 1 (least) to 9 (most)).

If the process works, you should get output something like this:

```
finds called
getty called
call: no. 999-1234 for sys hpopus DEVICE found > cul14
Status 0 :device /dev/cul14 :descriptor 5
Call device cua14 :descriptor 6
Setting up call device line
autodial return status > 0
ret > 22034 :pid >22034
login called
wanted login: ___login:got that
wanted sword: _bilbo__Password:got that
msg-ROK
 Rmtname hpopus, Role MASTER, Ifn - 6, Loginuser - tomh
rmesg - 'P' got Pg
wmesg 'U'g
Proto started g
protocol g
*** TOP *** - role=1, setline - X
wmesg 'H'
rmesg -'H' got HY
 PROCESS: msg - HY
HUP:
wmesg 'H'Y
cntrl - 0
send 00 0,exit code 0
```

where the parts have the following meanings:

- The lines:

```
finds called
getto called
call: no. 999-1234 for sys hpopus DEVICE found > cul14
```

  get the phone number or direct line to the remote system. Select a device file from `L-devices` that corresponds to the modem or direct connect and the `hpopus` taken from `L.sys`.

- The lines:

```
Status 0 :device /dev/cul14 :descriptor 5
Call device cua14 :descriptor 6
Setting up call device line
autodial return status > 0
```

  indicate that the modem autodial on the line `cul14` was successful.

- The lines:

```
ret > 22034 :pid >22034
login called
wanted login: ___login:got that
```

  indicate that the remote system offered the string `Login:`. This matches the first expected prompt in `L.sys`.

- The line:

```
wanted sword: _bilbo__Password:got that
```

  indicates that the remote system offered the string `Password:`. This matches the string `sword:` (the second expected prompt), in `L.sys`.

- The lines:

```
msg-ROK
 Rmtname hpopus, Role MASTER, Ifn - 6, Loginuser - tomh
```

  indicate who initiated the connection.

- The lines:

```
rmesg - 'P' got Pg
wmesg 'U'g
Proto started g
protocol g
```

  indicate that the answering (slave) system offers "g protocol" as the available communications protocol. Your system (the master system) accepts this protocol.

- The line:

```
*** TOP *** - role=1, setline - X
```

  indicates that your system is in role 1 (master) and is ready to exchange data.

- The lines:

```
wmesg 'H'
rmesg -'H' got HY
 PROCESS: msg - HY
HUP:
wmesg 'H'Y
```

  indicate that there is no data to exchange, negotiate the hangup, and terminate the connection.

- The lines:

```
cntrl - 0
send 00 0,exit code 0
```

  indicate that *uucico* ends with a successful termination code.

At this point, if you pause a moment, and look at the cause/effect relationships in the previous examples, you see that solving most problems requires removing inconsistencies among the hardware and files used by *uucp*, *uucico*, *cu*, *uuxqt*, and so on. In most cases, you need to let any feedback you receive direct you to re-examine the tasks you performed to get this far. By working out such problems, you can get *uucp* running to your satisfaction.

Using uucp 4

After you ensure that *cu* functions properly (i.e. you know the hardware and configuration works) and you can successfully use *uucp* to send or receive a file, you can work through this chapter to learn how to use *uucp*.

How uucp Works

The *uucp* utility actually does a small portion of the work required to transfer files among systems. In general, the following things happen:

1. *Uucp* creates spool files according to your specifications.
2. *Uucp* executes *uucico*, which continues the transferring process, and *uucp* terminates.
3. *Uucico* checks the time (according to day and time) to reach the remote system; and on having an appropriate time, *uucico* attempts to log into the remote system.
4. *Uucico* starts up on the remote system.
5. The local and remote *uucico* programs swap specified files.
6. *Uuxqt* executes on both systems and attempts to execute any commands that have been delivered from the corresponding remote systems.

Some Examples

The next several subsections use some examples to examine these processes. (By the way, Chapter 5, "Uucp File Structure", has additional examples.)

The uucp Spool Files

When you use *uucp* to copy a file from your system to a remote system, two types of files can be created and put into `/usr/spool/uucp`:

Command file: tells *uucico* the name of the remote system receiving the file and where the file must go in the remote file system. Command files begin with `C.`

Data file: an intermediate copy of the file being transferred (by default, *uucp* uses the original file).

For example, executing:

```
uucp update hpopus!~uucp/update
```

sends a data file named `update`.

Uucp tries to copy your original file unless you specify the `-C` option, which might create `C.hpopusn4569`. The last four digits of the file come from the contents of `/usr/spool/uucp/SEQF`, which is a sequence file that gets updated after the number has been read. The initial `C` means the file is a command file; the `hpopus` is the remote system; the `n` is the priority of the file (`n` is the default in a range from a grade of **A** to a low grade of `z`.

The contents of such a command file might be:

```
S /users/tomh/update ~uucp tomh -dc D.0 666
```

where:

- The `S` makes the local system the SOURCE system in the file exchange. To make the remote system copy to the local system, use `R` , which means Receive.
- The `/users/tomh/update` is the path for the source file.
- The `~uucp/update` is the path for the destination system (note the shorthand form that will be expanded by the destination system).
- The `-dc` specifies options used by *uucp*.
  - **-d** makes necessary intermediate directories on the remote system.
  - **-c** forces use of the original source file as the file to copy to the remote system rather than using an intermediate copy (specified by the **-C** option).
- The `D.0` specifies copying a data file.
- The `666` indicates the file-permission modes of the file being copied (the destination file will have the same modes).

Mail Spool Files

Sending mail to another system creates command and data files. For example, assume you execute:

```
mailx hpopus!spock
```

The created spool files might be:

```
C.hpopusA4573
D.hpopusB4572
D.hpfcmqX4570
```

The command file, `C.hpopusA4573` , might give directions for delivering the intermediate files by containing these lines:

```
S D.hpopusB4572 D.hpopusB4572 tomh - D.hpopusB4572 0666
S D.hpfcmqX4570 D.hpfcmqX4570 tomh - D.hpfcmqX4570 0666
```

The first line contains the intermediate data file, `D.hpopusB4572` , which is a copy of the mail file you originated. Note that the command file is grade `A` (the highest priority) and the intermediate file has grade `B` (the next highest priority).

The file named `D.hpfcmqX4570` is created to deliver the mail. This file includes the local system name, not the remote system name, to indicate the remote system that initiated the mail transfer. The contents might be:

```
U tomh hpfcmq
F D.hpopusB4572
I D.hpopusB4572
C rmail spock
```

where:

- The first line names the user who originated the file and the name of his system.
- The second line specifies the intermicate data file to be transferred.
- The third line contains the name of the file to be used for input— almost always the same file as the data to be copied.
- The final line contains the name of the command to be executed on the remote system— note the use of *rmail* to send mail to user on a remote system. Note also the grade, `X` , which specifies an executable file.

Transferring Spool Files

After *uucico* establishes a connection between the local and remote systems, the command and data files are transferred; the `X` file `D.hpfcmqX4570` gets a new file name beginning with `X` in the remote spool directory.

After the Spool Files Have Been Transferred

When the local and remote *uucico* programs finish exchanging files, and the connection has been terminated, `/usr/lib/uucp/uuxqt` executes. This program tries to execute the commands held in the files beginning with `X` in the spool directory. When a command requested by another system cannot be executed, mail is sent back to the user on the requesting system.

Remote Execution Permission

Remote execution permission is determined by `/usr/lib/uucp/L.cmds` . This file usually grants the capability to deliver mail only on remote systems. If yuou receive mail back from the other system, indicating that you were not allowed to execute `rmail` , then you should have the system administrator on the other system add `rmail` to this file.

Continuation

Beyond this brief introduction, using *uucp* is mostly a matter of determining how you want to transfer files among remote systems. You will need to coordinate your actions with other system administrators and the needs of the people in your group.

Uucp File Structure 5

By now, you have probably noticed that the *uucp* utility uses files from the directory structure shown below in Figure 5-1. This chapter discusses the *uucp* file structure for the purpose of showing the location of files and helping you see what they do.

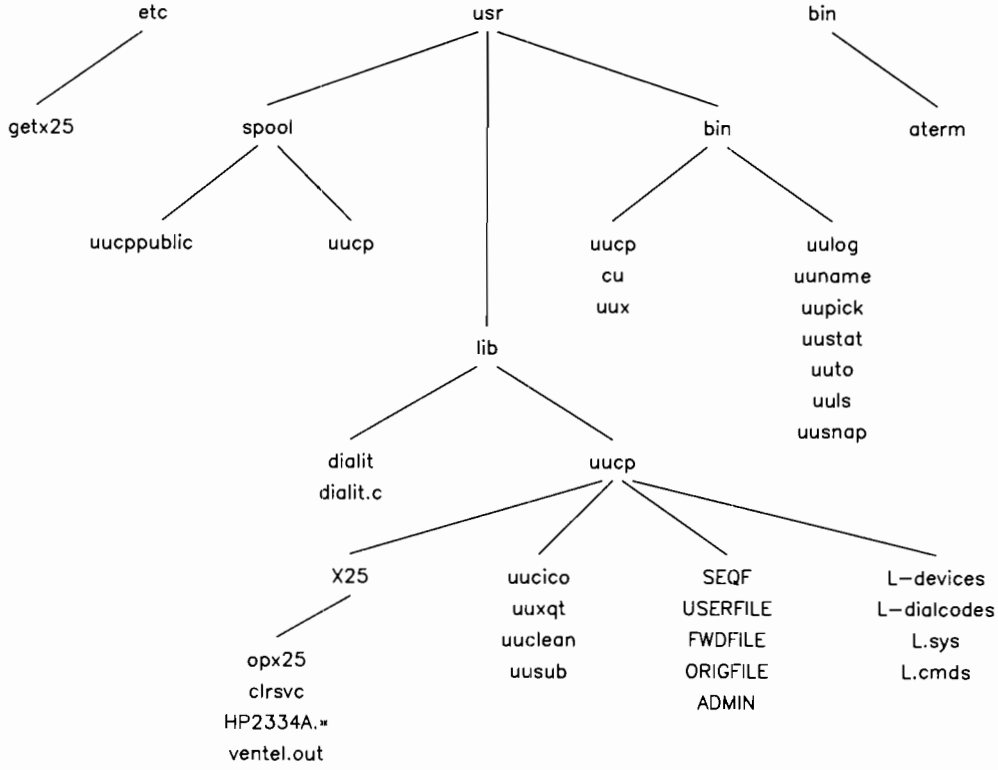


Figure 5-1. File Structure for Uucp

Examples of uucp Data Transfer

The next three subsections contain examples that illustrate the dynamics of the *uucp* data transfer. Subsequent descriptions of the *uucp* file structure use these examples.

Transferring a File Between Systems

To transfer a file from one system to another, use one of the following commands:

- Local to Remote. Execute:

  `uucp` *local_source_file remote_dest_file*

- Remote to Local. Execute:

  `uucp` *remote_source_file local_dest_file*

This following example transfers one file from a local system to a remote system— like you executed:

```
uucp junk norad!/usr/spool/uucppublic/junk
```

By the way, if you used the C shell, you would need to preceed the `!` with a backslash, `\!`.

Having determined that `junk` has read-permission by everyone and `/usr/spool/uucppublic` has write-permission by everyone, the following things happen:

1. The local system checks its `USERFILE` to verify the user access to the source file and to see that the file has read-permission by everyone. If the user is not found, the default is everyone (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` for information about connecting the device to the remote system.

5. *Uucp* checks `L-devices` 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 you have a modem, `dialit` 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).
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 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 (when the destination file already exists, it must have write permission).
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

To transfer multiple files between a local and remote system, execute the following command:

`uucp -C` *local_source_file remote_dest_file*

The following 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 it discusses the callback option as-well-as multiple work orders on both systems.

As before, make the source_file has read-permission by everyone.

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. 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.
4. *Uucp* checks the local `L.sys` for information on how to connect the device to the remote system.
5. *Uucp* checks the `L.devices` 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 the transfer involves a modem, `dialit` executes a dialing routine.
8. *Uucp* logs into the remote system according to the information in `L.sys` (device filename) and `L-devices` (type of connection). The remote system is the slave, so the master waits for the return message "`Shere`", which indicates 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 (the `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. when the destination file already exists, it must have write-permission. 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.

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 the process goes back to Step 10.

15. The slave sends a hangup message, "`HY`", to the master and they both disconnect.

Uux Command Sequences

Uux has the following syntax:

```
uux command_string
```

The following example illustrates the *uux* command sequence of actions:

1. The command sets up a workfile (`C.*`) and two data (`D.*`) files 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 contains any files the command requires.

2. Steps 2 through 13 are the same as in the previous example.

   When a request for work is sent to remote system, the file named `D.aaXbb` becomes an execution file (`X.*`) on the remote spool directory. The command transfers any other data files. At this point both systems can disconnect; and then the execution daemon named *uuxqt* starts.

14. The remote system checks its `L.cmds` to verify that your local system can execute the specified *command_string* on the remote system.

15. The remote system executes the command.

16. The remote system notifies the local system by mail of the execution status.

Spool Directory

The following subsections examine the `/usr/spool` directory.

The Public Area

The public area, `/usr/spool/uucppublic`, has 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. Using *uucp* automatically creates these files.

Workfiles

Workfiles have a `C.` prefix. They contain work orders to copy data files. Using *uucp*, *uux*, or the remote *mail* command, automatically creates these workfiles. A child process of the parent *uucp* scans `/usr/spool`, 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. Figure 5-2 shows the general form of a `C.workfile`.

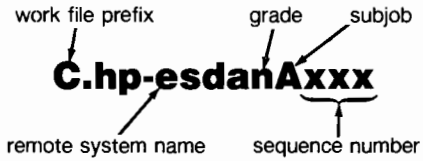


Figure 5-2. Workfile General Form

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 get 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— the 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, shown in Figure 5-3, contains one or more lines having the following form (you see only seven fields):

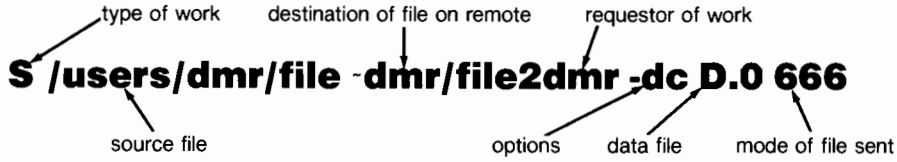


Figure 5-3. Workfile Contents

In looking at the lines, note the following items:

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 (-). In the **-dc** options (the defaults), the `d` directs the system to make needed directories on the destination; the `c` directs the system to copy from the source file(s). The `C` option indicates that a copy of the source file exists as a `D.` file in `/usr/spool/uucp`.

data file is the data file to be copied. The `-c` option means the data file should be `D.0` while the `C` option means the file is `D.*`. A `D.0` data file uses the source file that is current at the time of transfer (background processing can cause a delay, which makes it possible for the file to be altered by another process before the transfer occurs). A `D.*` data file uses the source file in its current state as of the time of the request.

mode is the mode of the source file sent in an `S` type of work. This can have read, write and/or execute permissions.

The eighth field exists only 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 of these letters).

Data Files

Two types of data files (image and execution) begin with the prefix "`D.`".

Image Data Files

Figure 5-4 shows image data file names.

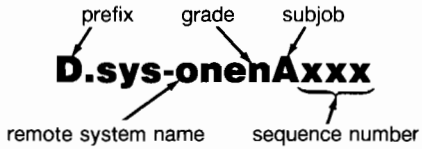


Figure 5-4. Image Data File Name

Note the following:

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 the work sequencing number. See grade under workfiles. The grade of these files is usually `n` when it is generated by a *uucp* command and it is `B` when it is 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— the sub-jobs of the request.

The image data file, `D.0`, generated with the `-c` option (the default) 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. Figure 5-5 shows data execution filename fields.

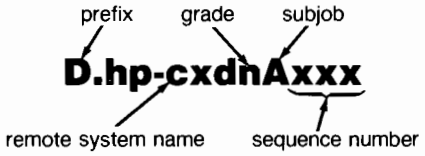


Figure 5-5. Data Execution Filename

Note the following items:

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 the job number associated with the data file.
sub-job	is the character used to differentiate among files having the same sequence number— the sub-jobs of the request.

Data execution files become execution files, `X.*`, on being transferred to the remote system. Like with all data files, their contents remain unchanged; only their names are 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. *Uucp* creates these files when the request for work is transferred.

Figure 5-6 illustrates their fields.

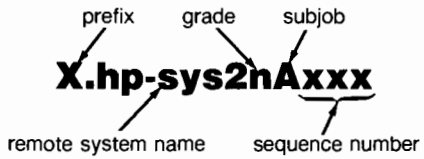


Figure 5-6. Execution File Name

Note the following items:

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.

A typical execution file contains these five lines:

- a user line (has a `U` prefix)
- a required file line (has an `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).

The file must contain F, I and O lines **only** if executing the command requires their respective files.

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.

Possible Use of Intermediate Nodes

You can route an execution file through intermediate nodes, but the information concerning its origin is lost. The user line in this file shows the last intermediate system routed through and the login used for *uucp*.

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

The standard input information line has the following syntax:

```
I D.file_for_standard_input
```

Here is an 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 the line shown in the example.

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 get created to provide exclusive access to a system while you use 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.

The following examples show 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 aborts abnormally, one or more lockfiles can still exist, and this prevents future communication through use of *uucp*. They can be deleted by using the */usr/lib/uucp/uuclean* program.

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 include:

`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) include:

`/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 abbreviations as strings 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 lines in `L.cmds` determine the commands that can be executed from remote systems, using *uux* on your local system. Edit this file to add new commands or delete old commands.

In the following example of `L.cmds`:

```
rmail,node1,node2,node3
pr,node1
col
lp,node12
```

limits remote execution to four commands. Some commands also include system nodes. 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

Uucp programs use *SEQF* to record the general sequence numbers used in work and data filenames. No external management is necessary.

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>
```

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.

There must be a blank or tab between the `user,system_name` field, the `c` option (if used), and each `pathname`.

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`.

In setting this up, check the following items:

1. Check the user field. 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.)

3. Check the path/file access for local user permission (if no user match is found, use the first blank or null user field).

4. 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.

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 131-1490
```

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, `131-1490`.

These abbreviations for telephone number prefixes can reduce 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.

The `dialit.c` file has four examples dialing routines. You might need to edit these routines or modify the USERSUPPLIED ROUTINES section for your specific needs. The comment lines in `dialit.c` and the following chart will help you modify these routines. The procedural headers in `dialit.c` contain information about configuring modems.

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 in this module exemplify the autodialing routines for selected modems. Hewlett-Packard makes no claim for the validity, reliability, or compatibility of this unsupported code.

In places where you deal with tone dialing, the rate of dialing the numbers can exceed the ability of the PBX to respond. Where this happens, you should insert a - or any defined pause signal between the dialed numbers.

The following code shows 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,
};
```

The lines at the beginning of `dialit.c` 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.

The Dialit File

The `dialit` file contains the object code that implements autodial sequences for the specific modem you use. When *uucp* checks `L-devices` and finds the device for the remote system that uses a modem connection, a call to `dialit` performs the autodialing and makes an entry in the file named `DIALLOG`.

Refer to the preceding section about `dialit.c` to get information about modifying the contents of `dialit`.

The L.sys File

The *uucp* utility uses `L.sys` to indicate:

- 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 for the data communications link; and
- How to log in on the remote system.

The following example shows 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
```

The next several subsections explain each entry.

Sysname

Thentry is 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

This entry 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

This entry 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.

This retry time specifies the **wait** time only. It does not specify that a retry dial operation will be attempted

Dev

This entry 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

This entry 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

This entry 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.

You Might Need To Add @

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
```

The **@** is the line-kill character before logging in. The **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 might 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 keeps 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 6

The *uucp* daemons perform the operations that let *uucp* transfer files. This chapter discusses these daemons.

Running The Uucp Utility

When you execute a *uucp* command two things happen:

- *Uucp* sets up workfiles in the */usr/spool/uucp* directory.
- A child process is spawned that invokes the *uucp* communications daemon:

```
/usr/lib/uucp/uucico -r1
```

Uucico means 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. On specifying a remote system, *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 that 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 on the system that originated the call, *uucico* 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

The scripts called *uudemons*:

- 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 `-s`*system* option, then *uudemon*s 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.

More Details About Uucp 7

This chapter provides information you can use after your system is running to make *uucp* work according to your needs.

The uucp Command

The next few sections discuss the *uucp* utility.

General uucp Snytax

The general syntax for the *uucp* command is:

`uucp [options]` *source_file(s) destination_file*

You can use the following options:

-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)

-e*sys* 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
-g*grade*	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*. In this example, ~ represents both the local login directory and the remote login directory. In this case, you have **~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]. You cannot use * 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:

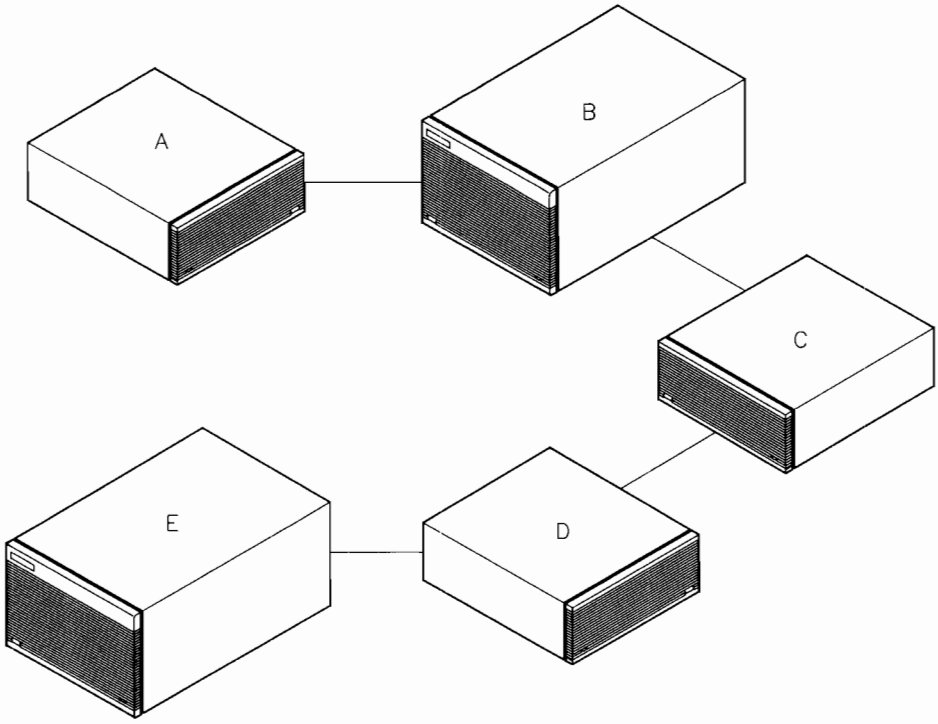


Figure 7-1. Network of Systems Using Uucp

The letter characters used in the following table represent system node names.

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 wanting 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 being sent to a remote system when it is placed in the destination directory **/usr/spool/uucppublic**.

Uucp Command Errors

An error in a *uucp* command transfer identifies the problem on the standard output device.

The only generated error, **1**, 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.

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` Requests notification of the remote system 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` Requests no mail notification for the remote system.

The *command_string* must be enclosed in double quotes, "*command_string*", when you specify an input or output diversion for the *command_string*. Without the quotes, the shell tries to redirect the input/output of the *uux* command. For example:

```
uux "sys2!pr !ems/cmd/file1 > sys2!myoutput"
```

requests execution of *pr* on the 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**).

The previous example showed two things:

- 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 0
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 **hpsys1** to 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.

The following example shows the use of *uuclean*:

```
/usr/lib/uuclean -pLOG -pLCK -n24
```

The command 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 chapter named "Log, Status and Cleanup" 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 named `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`.

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.

The list contains the names of systems with which you can communicate.

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 [-s`*sys_name*`]`

where:

`-s`*sys_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:

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)
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

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:

-c*hour*	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).
-j*all*	Report the status of all the *uucp* requests.
-k*job_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.).
-m*machine*	Report the status of accessibility of **machine**. If **machine** is **all**, then the status of all machines known to the local *uucp* are displayed.
-o*hour*	Report the status of all *uucp* requests that are older than **hour** hours old.
-y*hour*	Report the status of all *uucp* requests that are younger than **hour** hours old.
-s*sys*	Report the status of all *uucp* requests involving system **sys**.
-u*user*	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
```

you could get the following 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:

`-a`*sys* add `sys` to the subnetwork (only one system can be added at a time).

`-c`*sys* exercise the connection to system `sys` by making a call to that system (`sys` may be **all** for all systems in the subnetwork).

`-d`*sys* delete `sys` from the subnetwork.

`-f` flush (erase) the connection statistics.

`-l` report the statistics on connections.

`-r` report the statistics on traffic amount.

`-u`*hour* 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 sub-network 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.
Bring a sack lunch or go through the line.
CTRL-D
```

The command typed on the first line mails lines two and three 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
```

The X.25 Network 8

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. An X.25 packet is a serially transmittable string of information containing the following fields:

ADDRESS	identifies the packet destination.
DATA	the data to be transmitted, usually not more than 256 bytes.
CHECKSUM	the error detection information.
SEQUENCE	ensures that data packets are handled in correct sequence.
OTHER	any other fields not discussed in this manual.

A Packet Switching Network consists of nodes (stations) where each node can interpret routing information contained in a packet and forward the packet to the appropriate node in the network. The computer originating the packet combines routing and error checking information with the data being transmitted and sends the packet to the nearest switching node. That node forwards the packet to its destination.

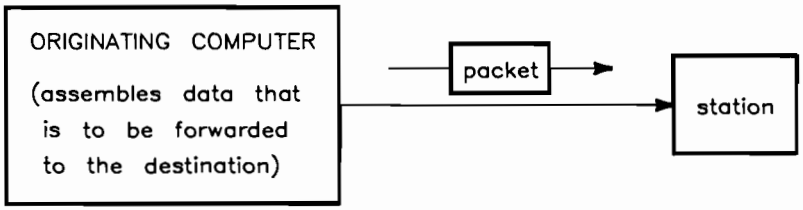


Figure 8-1.

Each switching node, in turn, examines address information in a packet as it is received, and automatically determines where to send it. The process repeats until each packet reaches its destination.

The following hypothetical stations and interconnections illustrate the idea:

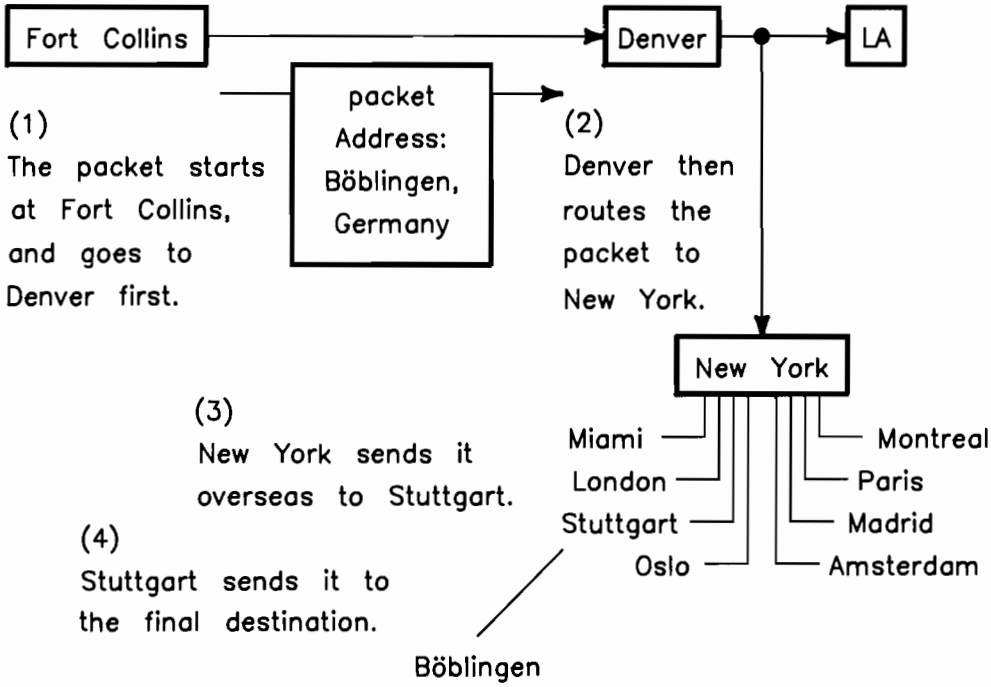


Figure 8-2.

The packet goes from Fort Collins, Colorado to Denver. The Denver node forwards the packet to New York for transatlantic transmission. New York forwards the packet to Stuttgart, Germany. Stuttgart forwards the packet to Boeblingen, the destination.

When the receiving system 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,

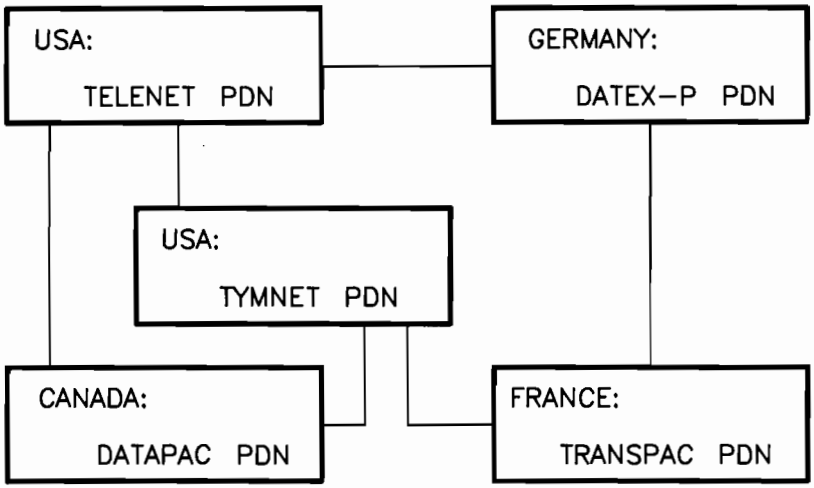


Figure 8-3. Public Data Network

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 performed 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 tests follow:

- 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* called "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.

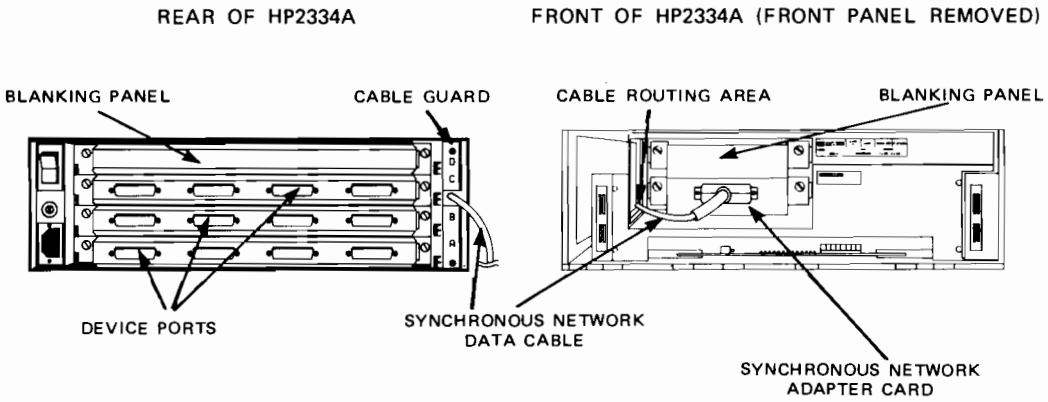


Figure 8-4. HP 2334A to Device and Synchronous Network Connection

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.

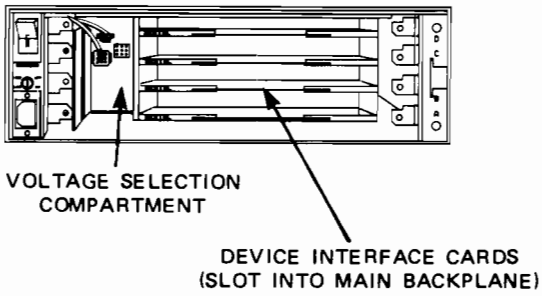


Figure 8-5. Adapter Card Slots

For local off-line configuration, install an HP 40261A card 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 8-4) remove the cable guard.

Do Not Remove Cards

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 8-4) 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 can 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 8-6 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.

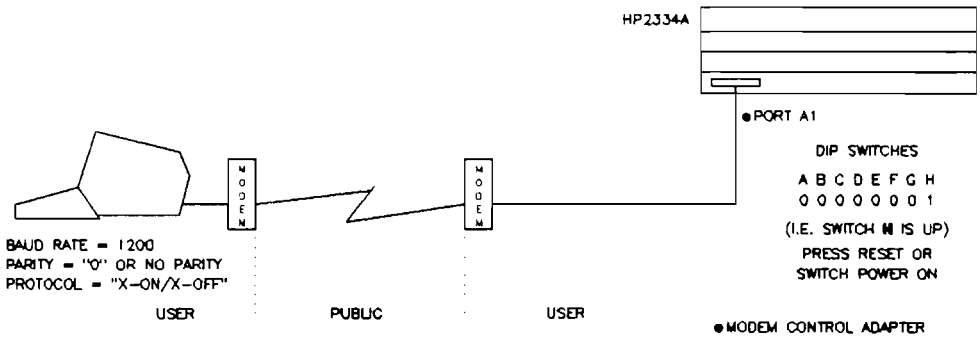


Figure 8-6. Remote Off-line Configuration

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).

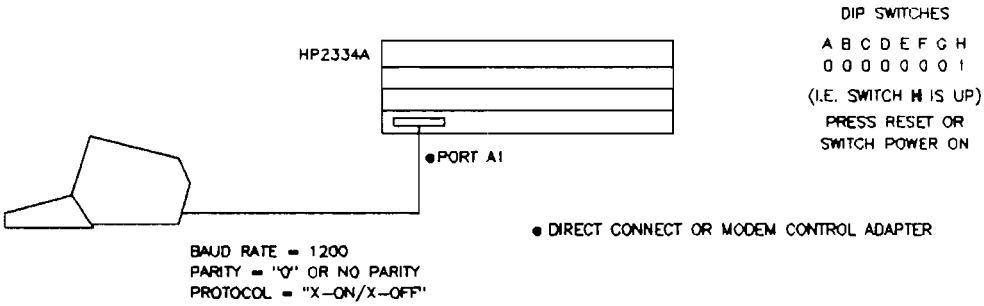


Figure 8-7. Local Off-line Configuration

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.
```

Pre-configured Mode

When the HP 2334A is in CONFIGURE MODE **only** port A1 is enabled and it is pre-configured at 1200 baud.

Configuration Procedure

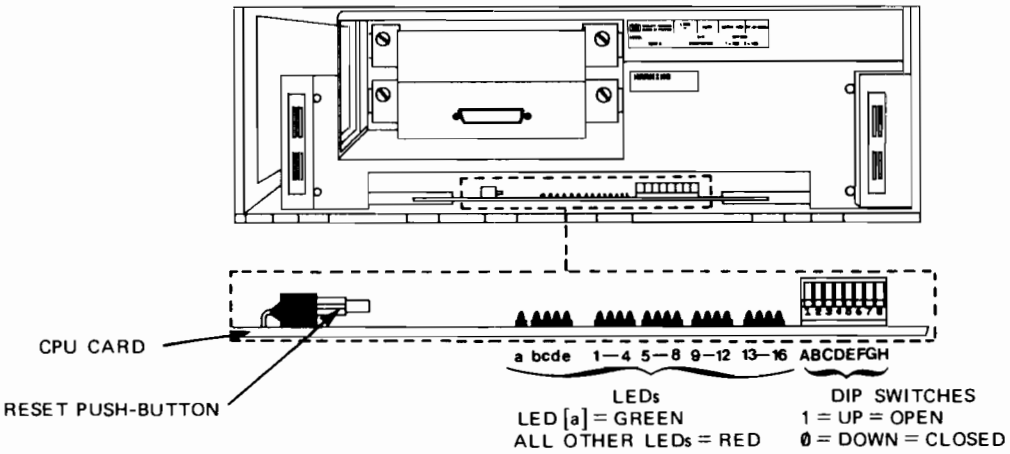
This section covers the following topics:

- 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 8-8. 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 lasts 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 it succeeds, the Internal Bus and Device Interface Cards get 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 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 items explain the 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.

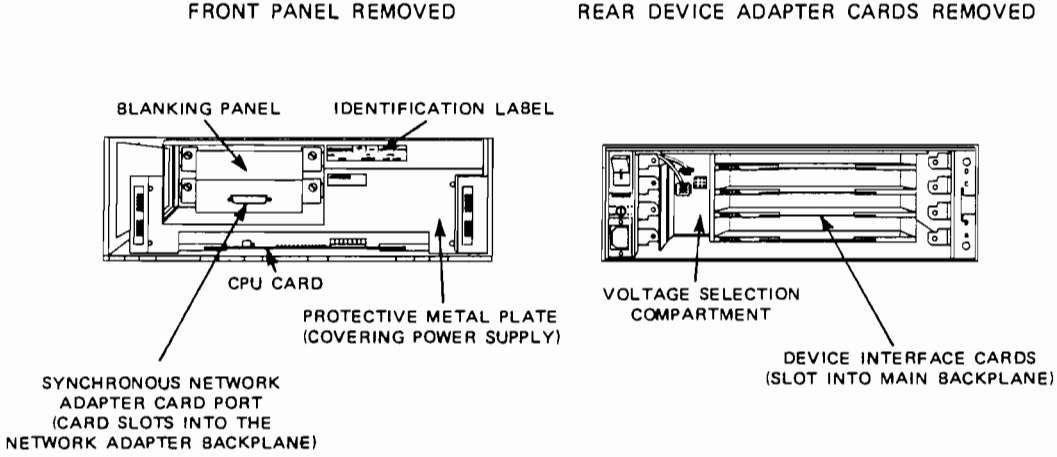


Figure 8-9. Front and Rear View of the HP 2334A

- 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
   ```

   You see the following message:

   ```
   Line_speed?
   ```

   Respond by entering:

   ```
   9600
   ```

2. Enter the fields for Level 2 of the configuration, entering:

   ```
   LEVEL2
   ```

   You see the following message:

   ```
   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 MULTIMUX Reference and Service Manual* for the proper response.

   ```
   TEL,12 [Return]
   ```

   You see the following message:

   ```
   Frame_window?
   ```

   Respond by typing:

   ```
   7 [Return]
   ```

   You see:

   ```
   Timer_T1?
   ```

   Respond by typing:

   ```
   3000 [Return]
   ```

3. Enter the values for Level 3 of the configuration, enter:

   ```
   LEVEL3
   ```

   You see:

   ```
   Lcl Addr.?
   ```

   Respond 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 appears in a tabular form. To use the table, start with the message and response at the top of the table and work your way to the bottom. Read the message in the left column and respond with the prompt in the right column. Type [Return] after typing your response. In some cases, you type [Return] without typing 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. You see:

```
Prof number?
```

Respond by typing:

```
2
```

You see:

```
    -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?
```

You see:

```
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].

You then see:

```
Prof number?
```

Respond by typing [Return].

The * should appear in the display. Proceed to the next section, which explains the ASG mode.

Entering the ASG Command Mode

The ASG primary user command assigns PAD or CAS/PAD profiles to the HP 2334A asynchronous ports. 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
```

You see a listing of the profile assignments of the ports:

```
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 * appears 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 a 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/**. You you should see the following message:

```
Connected
ERR
@
```

7. You can now test the HP 2334A. To do this, type:

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 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 the items have the following meanings:

[**-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
```

Log, Status and Cleanup 9

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 DIALLOG, LOGFILE and SYSLOG.

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)
```

The LOGFILE Shows Local Time Zone

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 *uudemon* 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, called "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

Here is the 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

Here is the 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.

Solving Problems 10

This chapter discusses problems that are most commonly encountered when using *uucp* facilities.

The Major Sources of Problems

While you can encounter several problems, the items described in the next few subsections cause most of the problems.

Bad Connections

A bad connection is the most common 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".

Log Entry Messages

When you cannot determine the cause of a problem by looking at the items just discussed, you can examine the messages found in DIALLOG, LOGFILE and SYSLOG files. The following subsections describe them.

/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 time zone 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.***.

Possible Loss of Information

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 time zone 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:

Reason not given by remote system **<no message appears>**.
Cannot copy to directory/file.
File left in **uucppublic**.
Local access to path denied.
Remote access to path/file denied.
Remote system cannot 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 time zone given by the originator (the time zone 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 time zone 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.

Index

a

b

c

d

e

f

g

h

i

k

l

m

n

o

p

r

s

t

u

W

X

Table of Contents

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)
setscreg(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()*.

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();
}
```

Index

a

b

c

d

e

f

g

h

i

k

l

m

n

o

p

r

s

t

u

v

w

Table of Contents

Developing International Software

Native Language Support Library and Commands

Task Reference of NLS Routines

Overview

This tutorial describes Native Language Support (NLS) and how to use the NLS tools on your Hewlett-Packard computer.

Please use one of the reply cards at the back of this tutorial 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?

This tutorial addresses end-users and system administrators as well as OEMs (Original Equipment Manufacturers), ISVs (Independent Software Vendors), applications programmers, and Hewlett-Packard Localization Centers who will be the primary users of Native Language Support (NLS). These are the people accessing, writing or translating programs in a multi-national environment. This tutorial 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: Introduction to NLS
Presents the basic description and scope of Native Language Support, localization, and internationalization including the aspects of NLS (character set handling, local conventions, and messages), and prelocalization.

Chapter 2: Using International Applications
Teaches the end-user how to execute a localized application including terminal configuration, environment setup, and selection of the language to use.

Chapter 3: Administering Prelocalized Software
This chapter is aimed at the system administrator. It identifies the HP-UX directories and files in which the NLS tools reside, explains setting the user environment, and describes the localization process.

Chapter 4: Developing International Applications
This chapter is aimed at the programmer. It explains the overall programming concepts of NLS, the language processing tools, the format(s) in which character data are presented, how to develop international software, how to handle characters with special tools, local conventions used while programming, and how to develop the user interface.

Appendix A: Native Language Support Library and Commands
Overview of NLS library routines, standard routines, and commands.

Appendix B: Task Reference of NLS Routines
An alphabetic reference of all NLS routines by task rather than command name.

Appendix C: Character Representation and Sets
A detailed explanation of character set representation and tables of character sets.

Appendix D: Peripherals Supporting NLS
Table summaries of HP 9000 peripherals that support alternate character sets.

Glossary:
Definitions of major words and concepts used in this tutorial.

Conventions Used In This Manual

The following conventions are used throughout this tutorial.

- *Italics* indicate manual names and references to manual pages in the *HP-UX Reference*. For example, see *date(1)* in the *HP-UX Reference*. Italics are also used for symbolic items either typed by the user or displayed by the system as discussed below under computer font.
- **Boldface** is used when a word is first defined and for **general emphasis**.
- `Computer font` indicates a literal typed to the system 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*. Computer font also indicates files, HP-UX commands, system calls, subroutines, path names, etc.
- Environment variables such as LANG or PATH are represented in uppercase characters, as required by HP-UX conventions.
- Unless otherwise stated, all references such as "see the *langinfo(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 *toupper(3C)* entry is under the *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 tutorial 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 NLS languages, if they are optional for your system. It also explains certain concepts used and implemented in HP-UX, describes system boot and login, and contains the guide for implementing administrative tasks.
- The *Native Language I/O User's Guide* describes how to use the NLIO system, which is the set of servers and filters, to input and output 16-bit characters efficiently on 16-bit hardware.
- The *NL I/O Japanese Supplement* describes different input methods available in the Japanese NLIO Subsystem and techniques for conversion of Katakana to Kanji candidates.
- The *HP-UX Documentation Roadmap* provides a short description and part numbers for all Series 200, Series 300, and Series 500 HP-UX manuals.
- The *Documentation Guide* provides part numbers for the Series 800 manuals.

Introduction to NLS 1

Developing international software requires an awareness of languages and custom differences around the world. Users want to be able to interact with Hewlett-Packard products in their native language. Interacting in their native language implies preserving the integrity of the data, properly handling their characters, providing interfaces in the local language and properly representing their local customs. **Native language** refers to the user's first language (learned as a child), such as English, Finnish, Portuguese, or Japanese. **Local customs** refer to local conventions such as date, time, currency formats, and names of days or months.

Preserving data integrity requires the application programmer's awareness of the different character sets and how to process characters from those sets.

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 or locale to locale.

Most existing software does not take this into account, making it appropriate for use only in the country or locale for which it was originally written. Formerly, this meant the programmer redesigned and recompiled the software for every local language and local environment. Now, the features of Hewlett-Packard **Native Language Support (NLS)** enable the application designer or programmer to create applications for an end-user's needs regardless of the local language. NLS addresses the internal functions of an application (such as sorting) and character handling as well as the user interface (which includes displayed messages, user inputs, and currency formats.) Incorporating NLS in HP products allows the development of a single product that operates in many different languages, each of which can be specified at run time. Then the product can be translated to supported languages without modifying and possibly corrupting the original code.

What is NLS?

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.

Prelocalization is application design or modification by the programmer where original HP-UX commands and routines are replaced with commands that incorporate NLS. For example, the routine `cxtime` would be replaced with the NLS enhanced version `nl_cxtime`. Also, prelocalization provides tools for placing all hard-coded information that is language or locale dependent in **message catalogues**. Then the message catalogues and language tables can be specified at run time, rather than having the information compiled into the programs. For a given piece of hardware or software, this process only needs to be done once.

Localization refers to the process of adapting a prelocalized software application or system for use in different local environments or locales. This process is completed by either a programmer or the system administrator. This consists of translating the text in the message catalogues to a particular language (such as French). This process must be done for each language or locale to be supported.

Once the prelocalization and localization processes are complete, NLS provides end-users with the following features:

- NLS permits users to specify the desired language at run time.
- NLS allows different users to use different languages on the same system.
- NLS saves disc space by separating the data tables required for a specific language product from the executable code.

If you are an end-user, your interest is in running international applications and, although informative, it is not necessary to read the rest of this chapter. Instead, read "Using International Applications".

If you are a system administrator, you should read the rest of this chapter and then read "Administering Prelocalized Software".

If you are programmer, read the rest of this chapter and then read "Developing International Applications".

Scope of Native Language Support

NLS facilities allow applications to be designed and written such that the language interface for the end-user and language for internal processing can be specified at application run time. The end-users then benefit from application programs that interact in their native language and conform to their local conventions.

For the ASCII-only user, the system appears unchanged. For the programmer and the system administrator, the interface does not change. Many HP-UX interfacing, subsystems, programmer productivity tools, and compilers have not been prelocalized. Applications programmers must still use American English to interact with HP-UX and its subsystems, although the compilers have been modified to use message catalogues. For example, a complete local language application program can be written using C, but the C compiler retains its English-like characteristics. For example, C keywords such as `main`, `if`, and `while`, and library calls, such as `fopen`, are still in English. However, C will accept comments and string literals in native languages and uses message catalogues for error messages.

Aspects of NLS Support

There are three aspects, or levels, of Native Language Support included in HP-UX software. These three aspects, **character handling**, **local customs**, and **messages**, describe the extent of NLS's ability to internationalize an application. If you are an applications programmer, you should consider each aspect carefully when creating software that is language independent.

Character Handling

NLS provides the ability to identify and manipulate the characters in a local languages character set. The default set is 7-bit ASCII (or USASCII); all programs not localized use this character set. 7-bit ASCII is not sufficient to span the Latin based alphabet used in many European Languages. In international software, the 8th bit of a character byte is never stripped or modified. Using the extra bit allows expansion to support languages that have additional characters, accented vowels, consonants with special forms, and special symbols. When large character sets are needed, characters span 2 bytes (16 bits). Hewlett-Packard has defined character sets with character codes in the range 0 thru 255 and 0 thru 65535 (although not all 65535 character codes are legal) for certain local languages instead of ASCII's 0 to 127. Every HP character set is a superset of ASCII. The HP supported 8-bit character set for European Languages is **ROMAN8**.

For languages with more than 256 characters, such as **Kanji** (the Japanese ideographic character set based on Chinese), multi-byte (16-bit) character codes are required. For more detail on the different character sets, refer to the appendix "Character Representation and Sets".

All sorting, case 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 Japanese); characters are not shifted at all.

Each language uses its own distinct **collating sequences** (the sequence in which characters are ordered by the computer). The ASCII collation order is inadequate even for American dictionary usage. Each language orders the characters in its character set differently. Certain **ideographic** character sets, which represent ideas by graphic symbols, can have multiple orderings. For instance, Japanese ideograms 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.

The assumption that text is displayed from left to right is not true for all languages. Latin based names interspersed with right to left text are still displayed left to right. Some Far Eastern languages are read from top to bottom, starting from the rightmost column.

Local Customs

Some aspects of NLS relate more to the local customs or conventions 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 local conventions. For instance, although Great Britain, the United States, Canada, Australia, and New Zealand share the English language, aspects of data representation differ according to local custom.

The representation of numbers, variations in the symbol indicating the radix character (the symbol that separates whole numbers from their fractional portion, a 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 local customs. For example, the United States and France both represent currency using periods and commas, but the symbols are transposed ($2,345.77 versus 2.345,77 FF).

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 calendar 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 vary with language. Rules for abbreviations also differ. The order of the year, month, and day, as well as the separating delimiters, are 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 and *7/10/86* in the U.K.

The chapter "Developing International Applications" and appendix "Task Reference of NLS Routines" describe the library routines used to programmatically handle these local customs.

Messages

The need to customize messages for different countries is a major reason for using NLS. 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 or translation of the meaning if messages are built in segments (using `printf`, refer to *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."* Noun endings, verb endings and article spelling can also change as segments are used differently or in different locations within a sentence. For example, the German translation of the word *"the"* may be *"der"*, *"die"*, *"das"*, *"den"*, or *"dem"* depending on gender and number of the noun and its position in the sentence.

To do this, messages must be put in a **message catalogue** from which they are retrieved by special library calls. See the chapter "Developing International Applications" for details.

Supported Aspects

For each of the aspects described above, certain **unsupported** examples were used for explanation and clarification. However, the programmer should take them into account to provide programming flexibility. These unsupported examples are:

- the number of digits in a group
- alternate methods of representing small monetary values
- the non-Gregorian calendar

On the other hand, a fully localized command, such as `pr` (for print), would incorporate all the **supported** aspects of NLS and therefore:

- never strip the 8th bit of a character code
- properly handle 16 bit characters
- properly format the date in each page header according to local conventions
- use the message catalogue system to select user error messages
- correctly deal with margins and indents for Middle East and African languages

Using International Applications 2

As an end-user, the details of NLS are not as important to you as for the original programmer or system administrator. Therefore, this chapter covers the tasks you perform to run software already localized to your native language and environment. Many HP-UX commands have been internationalized to different **levels of support**. Refer to the "Commands" section of the appendix "Native Language Support Library and Commands" for details as to these levels of support.

Checklist

The following is a checklist of the major steps you need to perform to use prelocalized software. Details for each step are described later in this chapter.

_____ Configure your terminal for use with prelocalized software.

_____ Check with the system administrator on correct "tty" settings.

_____ Set your LANG environment variable.

_____ Set your NLSPATH environment variable.

_____ Check with the system administrator to be sure the software has been localized.

_____ Run the software.

If you are interested in the processes of prelocalization and localization, see the chapter "Developing International Applications".

Terminal Configuration

Your 8-bit terminal configuration changes slightly for use with NLS and is described below. For 16-bit terminals and printers, there are special considerations. See the *Native Language I/O User's Guide* for details on how to set your peripherals.

If accessing prelocalized software from a 8-bit terminal, you must change your terminal configuration for 8 bits per character, no parity, and data bits to 8. If your terminal allows it, you may also need to select a base language set. Base sets are: KANA8, ARABIC8, TURKISH8, GREEK8, and ROMAN8. See your terminal's operation manual for details on how to access and set each of these values.

If your terminal configuration does not support the character set for your language, contact the local software sales office. The table "Supported Native Languages and Character Sets" will tell you which character set you need to support your langauge.

Setting the Terminal Line

You must ensure your **tty** line is capable of handling all the characters your terminal will generate. First, determine your current "tty" communications capabilities; type:

```
stty -a
```

That command lists terminal details similar to this:

```
speed 9600 baud; line = 0; susp = ^Z; dsusp <undef>
parenb -parodd cs7 -cstopb hupcl cread -clocal -crts
-ignbrk -brkint -ignpar -parmrk -inpck strip -inlcr -igncr icrnl -iuclc
ixon ixany -ixoff
```

To use international software, the parameters `-istrip` and `-parity` should be set. To change the "tty" line and ensure correct data transmission, type:

```
stty -istrip -parity
```

Alternately, you can add this command to your `.login` or `.profile`, respectively, to be invoked at login.

These changes should be made at a global level as well. See your system administrator to make sure that the `/etc/gettydefs` file is set properly for your "tty line". The system administration information for this process is in the chapter "Administering Prelocalized Software".

Setting Your Environment

The last steps you need to perform are selecting the language and setting the path names in local environment variables. The environment variable LANG (LANGuage) specifies the language to use and needs to be set to the language of your choice. The environment variable NLSPATH must be set to the appropriate path names.

Setting LANG

LANG is an environment variable that must be set to the native language you desire. LANG must be set to one of the languages listed in **/usr/lib/nls/config** and in the following "Supported Native Languages and Character Sets" table. When a prelocalized program is executed, LANG tells the application which language rules to use. It specifies the rules for lexical order, upshift and downshift tables, and other conventions that vary with language and locality.

Set LANG interactively from the shell or by editing either your **.profile** or **.login**.

For example, in Bourne shell, the syntax for setting LANG to **american**, either interactively or in **.profile**, is:

```
LANG=american
export LANG
```

For the C shell, edit **.login** to add the following or interactively execute the following:

```
setenv LANG american
```

If LANG is not set, or is set to an invalid **language name**, all prelocalized programs default to the native computer (**n-computer**) language. Programs run in this environment do so as if NLS does not exist.

The native languages to which LANG can be set are shown in the "Language Name" column of this table:

Table 2-1. Supported Native Languages and Character Sets

Language Name	Character Set
n-computer (native computer)	ASCII
american	ROMAN8
c-french (canadian french)	ROMAN8
danish	ROMAN8
dutch	ROMAN8
english	ROMAN8
finnish	ROMAN8
french	ROMAN8
german	ROMAN8
italian	ROMAN8
norwegian	ROMAN8
portuguese	ROMAN8
spanish	ROMAN8
swedish	ROMAN8
katakana	KANA8
arabic	ARABIC8
arabic-w	ARABIC8
greek	GREEK8
turkish	TURKISH8
japanese	JAPAN15

If after setting LANG, you do not get the expected language, check with the system administrator to verify that the required language-specific files are properly installed on the system.

Setting NLSPATH

NLSPATH is an environment variable that tells NLS where to search for a message catalogue. A **message catalogue** is a file containing prompts, responses to prompts, error messages, and mnemonic command names in your language.

NLSPATH consists of a series of colon (":") separated path names, which may contain the wildcards shown in the following table:

Table 2-2. NLSPATH Wildcards

Wildcard	Expansion by NLS
%L	%L is replaced by the current value of the LANG environment variable.
%N	%N is replaced by a *name* which is typically the name of the program.

Consider the following example where NLSPATH and LANG are set to values indicated below:

```
LANG=german
NLSPATH=/users/project/localized/%L/%N.cat:/users/project/test/%N.cat
```

When the prelocalized application, called **exfile**, executes. it first checks to see if NLSPATH is set. Next, after replacing %N with **exfile** and %L with "german" in the first path name specified by NLSPATH, the program searches for a file whose path name is **/users/project/localized/german/exfile.cat**. If this file exists, the program attempts to use it as the message catalogue. If it doesn't exist, it checks to see if another path is specified by NLSPATH. Then, replacing %N with **exfile** in the path name, the program searches for a file whose path name is **/users/project/test/exfile.cat**.

If the application can't find a message catalogue with the path names specified in NLSPATH, it searches the default location:

```
/usr/lib/nls/%L/%N.cat
```

This is the location of the message catalogues of all HP-UX commands and HP supplied applications.

As with LANG, NLSPATH is set in your `.profile`:

```
NLSPATH=/users/localized/%L/%N.cat:/users/test/%N.cat
export NLSPATH
```

or `.login`:

```
setenv NLSPATH=/users/localized/%L/%N.cat:/users/test/%N.cat
```

For a more detailed description of how NLSPATH and LANG are used by a program, refer to the chapter "Developing International Applications".

At this time you are ready to run any international application that has been localized. If the software has not been localized, see your system administrator.

Administering Prelocalized Software 3

After programmers prelocalize the applications, they need to be localized to the area. The administrator must understand the basic constructs of NLS to perform the localization as well as maintain NLS. This chapter describes the tasks you, the system administrator, should be concerned with in managing a system with NLS and localizing applications. These tasks should be performed by the administrator and transparent to other users.

Checklist

The following is a checklist of the major tasks you must perform to support NLS and your users. Each task is described in detail later in this chapter.

Here is a checklist of the major administration tasks:

_____ Install or update the software, if necessary.

_____ Edit `/etc/profile` and `/etc/csh.login` for LANG, NLSPATH, and TZ.

_____ Edit `/etc/gettydefs`, if necessary.

_____ Localize prelocalized applications.

File Hierarchy

You need to understand the NLS file system hierarchy. Prelocalized HP-UX commands and C library routines for NLS are in standard directories (`/lib`, `/bin`, `/usr/bin`, `/usr/include`, and `/usr/lib`), but language-dependent features reside in specific NLS directories and files: `/usr/lib/nls/config` and `/usr/lib/nls/`*language_name*.

The language configuration file, `/usr/lib/nls/config`, lists all the native languages available on the system or from the sales office as separate products. The `config` file is readable ASCII and contains the **language-ID** and **language name**. Refer to the "Languages" table below for a listing.

Table 3-1. Languages

Language-ID[1]	Language Name
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
51	arabic
52	arabic-w
61	greek
81	turkish
221	japanese

[1] Language-ID is listed for historical reasons only and should not be used.

The **config** file must be present before prelocalized commands can work correctly. If a language is not specified or is set to an invalid language string, all prelocalized applications default to native computer, **n-computer**. Native computer is the artificial language computers used to deal with languages before the introduction of NLS.

For each language available on the system, a corresponding **/usr/lib/nls/***language_name* directory must be present. This directory will contain language-dependent processing information required by *language_name*. Specifically, the directory contains the tables necessary for collating, upshifting, downshifting, character type, and language information. The files **ctype**, **collate8**, **shift**, and **info.cat** contain the same information and are necessary for backward compatibility only. Additionally, the directory typically contains the translated message catalogue files for *language_name*.

Installing Optional Languages

HP-UX is always shipped with the default language (**n-computer**). For some operating systems, the other languages (such as German) are shipped as well. However, for other products they must be ordered as an option from your Hewlett-Packard sales office. If uncertain about the languages supplied with your product, contact your local sales office.

A **language** includes a table for collating, upshifting, downshifting, and character type and language information. Not all character sets are supported on all peripherals, so peripherals which support the desired character set must also be obtained.

To install a language, use the **update** command, as explained in the update section of the *HP-UX System Administrator Manual* for your computer. The **/etc/update** script automatically installs the language support files in the correct directories 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.

Setting the User Environment

To support NLS, changes to the user environment within HP-UX are needed. The environment variable **LANG** (LANGuage), which specifies the language to use, needs to be set to the users choice. The environment variable **NLSPATH** must be set to the appropriate path names. The variable **TZ** (Time Zone), which allows selection of different zones, needs to be set if it has not been already.

Setting LANG

LANG is an environment variable you must set to the language you want to use. LANG must be set to one of the language names listed in `/usr/lib/nls/config` and the preceding "Languages" table. When a prelocalized program is executed, LANG tells the application which language rules to use. It specifies the rules for lexical order, upshift and downshift tables, and other conventions that vary with language and locality. By setting LANG in `/etc/profile` or `/etc/csh.login`, you provide a default native language for all users. Additionally, each user can specify a language by setting LANG interactively, or in their `.profile` (for Bourne Shell) or `.login` (for C shell). This information has been provided to the end user in the chapter "Using International Applications" so it is very possible the end user has already set LANG.

For example, in `/etc/profile`, the syntax for setting LANG to `american` is:

```
LANG=american
export LANG
```

This is the same syntax used by the Bourne Shell interactively or by the end users in `.profile`.

For the C shell, edit `/etc/csh.login` globally, edit the end users `.login` or interactively use:

```
setenv LANG american
```

If LANG is not set, or is set to an invalid language name, all prelocalized programs default to the `n-computer` language.

Setting NLSPATH

NLSPATH is an environment variable that tells NLS where to search for a message catalogue. It consists of a series of colon (":") separated path names, which may contain the following wildcards:

Table 3-2. NLSPATH Wildcards

Wildcard	Expansion by NLS
%L	%L is replaced by the current value of the LANG environment variable.
%N	%N is replaced by a *name* which is typically the name of the application.

Consider the following example where NLSPATH and LANG are set to values indicated below:

```
LANG=german
NLSPATH=/users/project/localized/%L/%N.cat:/users/project/test/%N.cat
```

When the prelocalized application, **exfile**, executes it first checks to see if NLSPATH is set. Next, replacing %N with **exfile** and %L with "german" in the first path name specified by NLSPATH, the program searches for a file whose path name is **/users/project/localized/german/exfile.cat**. If this file exists, the program attempts to open it as the message catalogue. If it doesn't exist, it checks to see if another path is specified by NLSPATH. Then, replacing %N with **exfile** in the path name, the program searches for a file whose path name is **/users/project/test/exfile.cat**.

If the application cannot find a message catalogue with the path names specified in NLSPATH, it searches the default location:

```
/usr/lib/nls/%L/%N.cat
```

This is the location of the message catalogues of all HP-UX commands and HP-supplied applications.

As with LANG, NLSPATH can be set in **/etc/profile** or **/etc/csh.login**, interactively, or in the individual's **.profile** or **.login**. Once again, this information, in a more generic form, has been provided to the end user in the "Using International Applications" chapter. Each end user may have NLSPATH already set. However, set it in **/etc/profile** or **/etc/csh.login** so it can be used globally.

Setting TZ

TZ is a variable that holds time zone information. TZ allows whole number and fractional offsets from GMT (Greenwich Mean Time is the international standard of time). Specification of daylight savings time is taken into account as well as name differences and starting and ending date differences.

It is set by default, when the user logs in, to a value in the **/etc/profile** or **/etc/csh.login** files. For example:

```
TZ=MST7MDT
```

Where MST is Mountain Standard Time, 7 is seven hours greater than GMT and MDT is Mountain Daylight Time. See *environ(5)* and *date(1)* in the *HP-UX Reference* for details.

Configuration

Hardware configuration for 8-bit peripherals changes only slightly for NLS. These are described below. For 16-bit peripherals, there are special considerations; see the *Native Language I/O User's Guide* for details on how to set the peripherals.

Setting 8-bit Configuration

If your user is accessing prelocalized software from an 8-bit terminal, the terminal configuration must be set to 8 bits per character, no parity, and data bits at 8. If the terminal allows it, the user must also select a base language set. Base sets are: KANA8, ARABIC8, TURKISH8, GREEK8, and ROMAN8. This process has been described to your users in the "Using International Applications" chapter and should already be completed.

If the terminal configuration does not support the character set for the language desired, contact the local sales office.

Setting the Terminal Line

The end user has been taught in the previous chapter how to check the "tty line" and execute **stty** to change the terminal line and ensure correct data transmission. See "Using International Applications" for details on this process.

However, for each user accessing prelocalized software from an 8-bit terminal, you must change the global software configuration in **/etc/gettydefs** for correct transmission. The items that need to be set are:

```
-istrip cs8 -parity
```

Either create a new line in **/etc/gettydefs** and change each **tty** in **/etc/inittab** to access the new line or change the line in **/etc/gettydefs** for all tty lines.

Refer to the *getty(1M)* and *inittab(4)* pages in the *HP-UX Reference* for detailed descriptions on how to do this.

Localizing Prelocalized Software

As the system administrator, you can provide a local language interface for your users on applications that have been prelocalized. This involves translating the **n-computer** message catalogue into a local language message catalogue. To determine whether a translated message catalogue already exists see **/usr/lib/nls/***language_name*. If not, check **/usr/lib/nls/n-computer** to see if a translatable **n-computer** catalogue is supplied. Then, follow the steps below to localize the catalogue to the local language. There are two HP-UX commands that help you to localize the **n-computer** files to a local language message catalogues:

- **gencat**— generates a formatted message catalogue file.
- **dumpmsg**— reverses the effect of **gencat**; takes a formatted message catalogue and makes a modifiable message catalogue source file.

These and other message catalogue tools are described in detail in the chapter "Developing International Applications".

Message Catalogue Localization Steps

As an example, create a localized message catalogue for **mailx**. The following steps will show you how to generate the message catalogue:

1. Use the **dumpmsg** command on the **n_computer** message catalogue file found in **/usr/lib/nls/n-computer/wc.cat**[1]. For example:

   ```
   dumpmsg /usr/lib/nls/n-computer/wc.cat > wc.msg
   ```

2. Translate the text file *wc.msg* into the language desired, like German, Japanese, etc. Be sure to let the translator know to preserve the message number, one space following the number, and any format strings.
3. Save the translated messages in the file **wc.msg**.
4. Execute **gencat**; type:

   ```
   gencat /usr/lib/nls/german/wc.cat wc.msg
   ```

   to add the translated message catalogue to the **german** directory.

You now have generated a localized message catalogue which the application accesses when run. For details on these concepts, refer to the chapter "Developing International Applications".

[1] The conventions for suffixes are: **.cat** for final message catalogues, **.msg** for message files or input files to **gencat**, and **.str** for string files or output of **findstr**.

Developing International Software 4

A well-written application program manipulates data and presents it appropriately for its use and user. Users benefit from application programs which interact with them in their native language, process textual data according to the grammar and rules of that native language, and conform to their local customs. However, program interfaces do not always take into account local customs and languages.

The solution to this problem is to design applications that can be easily localized. Traditionally, the user's native language and data processing requirements differed from those in the environment of the software developer. Localization was achieved by modifying program code for each specific country. Prelocalization of applications, by incorporating NLS routines and commands and designing programs with localization in mind, provides a better solution. Localization can then be accomplished with (ideally) no modification of code at all.

An applications designer should write programs with built-in provisions for localization. Functions which are local language or custom-dependent cannot be **hard-coded**. For example, all messages and prompts must be stored in an external file or catalogue. Character comparisons and upshifting must be accomplished by external system-level routines or instructions. Then external files and catalogues can be translated, and the program localized without rewriting or recompiling the application program.

Native Language Support (NLS) provides the tools for an applications designer to produce international 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 (e.g., sorting) as well as its user interface (e.g., messages and formats).

Prelocalization Process

This is an overview of the process needed to internationalize applications. Each is described in detail later in this chapter.

- Design your program so it is capable of reading and writing all the characters (letters, symbols, punctuation marks, etc.) required by the different languages. To do this, you must understand how character data is represented. This is explained in the section "Character Codes and Character Sets".
- Understand the functions offered by the NLS tools, enabling you to know when to use NLS tools and when to use standard C tools. The NLS tools and their function are described in the section "Character Processing Tools".
- Include a call to `nl_init` at the beginning of your program to initialize the program's environment to the user's selected language. This enables the NLS tools to work in a language-sensitive way. The user, the system administrator, or the application programmer must set the LANG environment variable to specify the desired language. The `nl_init` call and the LANG environment variable along with their use are described in the section "Initializing the Environment".
- Design your program to handle characters and strings according to the rules and grammar of the user's language. This involves using NLS tools to perform those character handling tasks that are language sensitive. These tools, their functions, and their use are described in the section "Character Processing Tools".
- Convert input data from its local format (as specified by local convention) to a standard internal representation, so that your program can process the data. Similarly, your program must convert output data from this standard internal representation to a format acceptable by local conventions. NLS provides tools to perform these conversions. The tools are described in the section "Local Conventions".

- Design a local language user interface which is independent of the actual program. This can be done with NLS's message catalogue commands and routines. The section "Creating the User Interface" provides a general overview and description of message catalogues. Then, the following three subsections provide specific guidance for the programmer wishing to utilize message catalogues:
    - The "Creating New C Programs that Access a Message Catalogue" subsection provides step-by-step procedures for writing new C programs that access a message catalogue. It also describes how to create the message catalogue for a new application.
    - The "Incorporating Message Catalogues in Existing C Programs" subsection provides step-by-step procedures for modifying an existing C program to add message catalogue capabilities.
    - The "Updating a C Program and its Message Catalogue" subsection provides step-by-step procedures for updating a C program and its message catalogue.

Character Codes and Character Sets

Written languages differ in many ways. Some languages are ideographic (such as Japanese, Traditional Chinese, and Simplified Chinese). In ideographic languages, a single written symbol can represent a complete thought or idea. In other written languages (such as American, English, German, and Greek), a single symbol or a combination of symbols represent a sound. The written language directly corresponds to the spoken language.

Character data is represented in the computer by numeric codes. The interpretation of the symbols, represented by a character code, is determined by the character set from which the data is assumed to originate.

To enable the computer to distinguish between the symbols used in written languages, each symbol must have a unique identifier or **character code**. Codes for the symbols used by a language (or by a family of related languages) are grouped together into a set, called a **character set**. Codes for American are based on the 7-bit ASCII character set. HP has extended ASCII to an 8-bit character set, that supports Western European languages, called **ROMAN8**. The character set is named ROMAN8 because it provides unique codes for the letters and symbols required to write most Latin derived languages. The **8** specifies that 8 bits (a single byte) are used to provide a unique code for each letter and symbol of these written languages.

ROMAN8 provides character codes for the symbols required to write the following languages:

Table 4-1. ROMAN8 Languages

```
American
Canadian French
Danish
Dutch
English
Finnish
French
German
Italian
Norwegian
Portuguese
Spanish
Swedish
```

Since 8 bits provides 256 (2^8) unique character codes, 256 different symbols can be represented with ROMAN8. However, this number is not sufficient to provide character codes for all languages. So other character sets were created: ARABIC8, GREEK8, and TURKISH8. These character sets provide unique character codes for the symbols required to write the languages indicated in the following table:

Table 4-2. Character Sets

Language	Character Set
Arabic	ARABIC8
Greek	GREEK8
Turkish	TURKISH8
Western Arabic	ARABIC8
Katakana	KANA8

Each character set listed in this table also provides the symbols required to write American English. Therefore, ASCII is a subset of each character set. This ensures that users can always access the symbols required to communicate with their systems.

Each of the character sets can provide at most 256 unique character codes for the symbols required to write a language. However, ideographic languages (such as Japanese, Simplified Chinese, and Traditional Chinese) use many more symbols. For example, Traditional Chinese consists of over 50,000 ideographs. Though the written language consists of a large number of written symbols, only a subset of these symbols are required for normal, daily communication. It was determined two bytes (16 bits) provides enough unique character codes for the ideographic characters required by these languages.

Because operating systems and computer subsystems still require ASCII to communicate, the character coding scheme for ideographic languages must also include ASCII.

To provide enough unique character codes for ideographic languages yet still provide ASCII, HP created a coding scheme in which character codes for ASCII can be intermixed with the 2-byte character codes used to represent ideographs. Now, any data stream can consist of a mix of 1-byte ASCII and 2-byte ideographic characters.

This is possible because the character codes of ASCII characters range from 0x00 through 0x7F (where 0x specifies that the number is a hexadecimal value). However, an 8-bit byte yields valid hexadecimal values 0x00 through 0xFF. Since ASCII requires only the values 0x00 through 0x7F, the values 0x80 through 0xFF can be valid values for the first byte of a 2-byte character sequence. Therefore, the first byte of a two byte character will

never be a ASCII value. Not all values are valid for the second byte of a 2-byte character. For example, the control code range (0x00 through 0x21) is reserved.

By sequentially examining each byte of a byte stream, it can be determined if the byte represents a single ASCII character or is merely the first byte of a 2-byte character sequence:

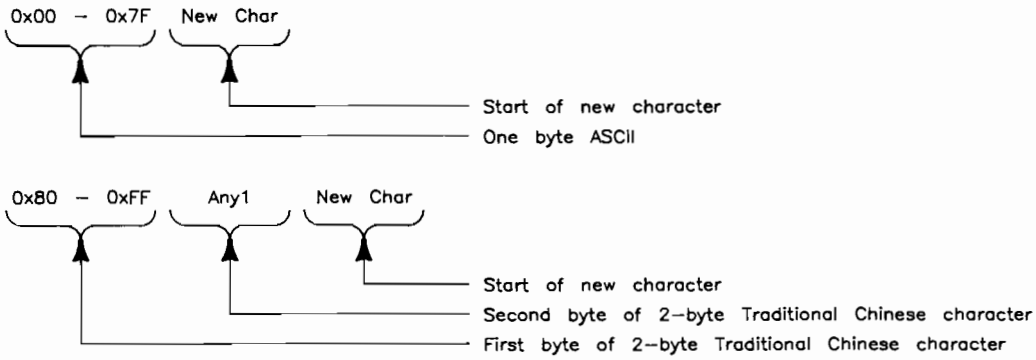


Figure 4-1. Interpreting Traditional Chinese Bytes

Notice that this character encoding scheme, referred to as **HP-15**[2], is not an actual character set. Several different HP supported character sets use this scheme (or a slightly modified version of the scheme) for valid character code ranges, just as several character sets (such as ROMAN8, GREEK8, ARABIC8, and TURKISH8) use an 8-bit encoding scheme. Each character set defines which codes in the range 0x80-0x0FF are single-byte and which are first of two bytes.

The emphasis here is that it is not really important to know the exact code ranges for each 2-byte character set. Rather, it is important to realize that valid characters now may be represented by either one or two bytes and that such data may be mixed in a data stream. The character size identification routines and the character pointer manipulation

[1] Almost any, 0x22-0x7E or 0x80-0xFF. Control codes and the SPACE character are not allowed.

[2] Peripheral devices work with computer data formatted according to a different character encoding scheme, called HP-16. The NLIO subsystem converts between HP-16 and HP-15 character data. This conversion is described in the *Native Language I/O User's Guide*.

routines described in the next section allow your program to examine a byte stream and identify and manipulate characters. The native language specified by the end user tells these routines how to interpret the bytes (i.e., whether it is a stream of 8-bit characters or a possible mix of 1-byte and 2-byte characters).

For example, suppose that the following diagram represents a stream of bytes on standard input:

	byte1	byte2	byte3	byte4	byte5	byte6	byte7	byte8
Hex Value	0x4A	0xE3	0x48	0x65	0x77	0x6C	Ox65	0x74

The letters and symbols these hexadecimal values represent depend on the character set used to interpret them. Examine the symbols which the hexadecimal values represent if the character set is ROMAN8 and then compare these symbols with the representation obtained if using a 2-byte character set, such as Traditional Chinese:

	byte1	byte2	byte3	byte4	byte5	byte6	byte7	byte8
Hex Value	0x4A	0xE3	0x48	0x65	0x77	0x6C	Ox65	0x74
ROMAN8	J	K	H	e	w	l	e	t
Traditional Chinese	J	1st of 2	2nd of 2	e	w	l	e	t

If the data is interpreted as ROMAN8, the bytes represent the symbols shown beside the "ROMAN8" label above. However, if the stream of bytes is interpreted as data in the Traditional Chinese character set (a set of 2-byte characters), the bytes represent the symbols shown beside the "Traditional Chinese" label above. Notice that the symbols represented by bytes 2 and 3 differ between the ROMAN8 and Traditional Chinese interpretations of the data. This is because the Traditional Chinese character set is a set of 2-byte characters. Therefore you must examine the byte stream from the beginning of the stream, testing each byte to determine if the byte represents a single ASCII character or if the byte is ½ of a 2-byte Traditional Chinese character. For example, if you examine the byte stream assuming that the bytes represent characters from the Traditional Chinese character set, the first byte has a value of 0x4A. Since this is less than 0x7F, the byte must represent a ASCII character. The second byte in the stream has a value 0xE3 which is larger than 0x7F. For the Traditional Chinese character set, this byte represents the first byte of a 2-byte character. This means that the third byte, whose value is 0x48, is really the second byte of a 2-byte character and not a 1-byte ASCII character.

NOTE

If you randomly jump into the middle of the byte stream (e.g., at byte 3), you will incorrectly interpret its representation. To correctly identify a character, you must start examining the stream (or string) from its beginning.

Similarly, if you split the stream between the two bytes, you will corrupt the character represented by the 2-byte sequence. These problems, in which a character is corrupted or in which half of a 2-byte character is treated as a whole 1-byte character, is known as **byte redefinition**.

For example, suppose that you want to find all occurrences of the character "/" in string. The character code for "/" is 0x2f. However, 0x2f is also a valid second byte of a 2-byte character in many HP-15 based languages. If your program simply scans a string looking for bytes whose value is 0x2f, it cannot guarantee that it finds only "/" characters. Instead, the string must be examined from its beginning using the character pointer manipulation routines. Then, by dereferencing the pointer and comparing the character with 0x2f, the program will locate only "/" characters.

The application designer whose code deals with character or string data must be aware that in any given string or data stream, a single character may consist of 8 bits or of 16 bits. NLS provides tools to help the software developer distinguish between 8-bit codes and 16-bit codes. These tools are described in the "Character Processing Tools" section.

An awareness of how character data is represented can help an application designer create programs that do not corrupt character data.

Character Processing Tools

Processing of character data is usually a function of the native language in which the data is processed. For example, sorting sequences differ between languages. A German-speaking person collates a word beginning with the letter "å" after words beginning with "a" but before those beginning with "b". However, a Swedish-speaking person collates a word beginning with the letter "å" after words beginning with "z". Character upshifting can be language and locale dependent. In France, the letter "ç" upshifts to "C"; however, in French-speaking portions of Canada, "ç" upshifts to "Ç".

NLS provides a collection of software tools that provide language-dependent processing capabilities for an application program. The language-specific processing information is kept in system language tables, external to the NLS routines and external to the application program.

The language tables contain language-specific information, including collating sequences, character upshift and downshift information, character identification information, time and date formats, currency symbol, and currency symbol position.

The information in the language table is copied to the user's process stack **at run time** by an NLS library call, which is placed at the beginning of the application by the programmer. Then the NLS library routines, header files, and macros access the language table information as the program runs.

Similarly, NLS library calls placed in the application by the programmer are used to retrieve textual information (such as error messages, prompts, softkey labels, and format strings) from external text files, called **message catalogues**. Message catalogues are described in detail in the "Creating the User Interface" section. NLS executable programs allow the application developer to easily build, update, and maintain the message catalogues.

Finally, NLS library calls can be placed in the application to retrieve custom-dependent information, such as:

- the correct format for display time and date
- the currency symbol and its location
- the names and abbreviations of the days of the week
- the names and abbreviations of the months
- yes and no responses
- character size (the number of bytes used to represent a character)

Initializing the Environment

Users of your application use the LANG environment variable to specify the desired native language. Set LANG for the Bourne shell by editing `/etc/.profile` or interactively typing:

```
LANG=Language
export LANG
```

Set LANG for the C shell by editing `/etc/.login` or interactively typing:

```
setenv LANG=Language
```

The valid values for LANG are listed in the text file `/usr/lib/nls/config` and are listed in the following "Languages" table:

Table 4-3. Languages

n-computer
american
c-french
danish
dutch
english
finnish
french
german
italian
norwegian
portuguese
spanish
swedish
katakana
arabic
arabic-w
greek
turkish
japanese

Notice, the first language listed is `n-computer` (native computer) which is an invented, artificial language. You can specify `n-computer` to force the application to work as if NLS has not been included. The `n-computer` language identifies the way the system previously performed those operations that NLS now performs in a language-dependent way.

Once you have specified an operating language by the LANG environment variable, your application must read the value of LANG (for example, with the **getenv** call) and pass it as an argument to NLS's **nl_init** routine:

```
return_value = nl_init(getenv("LANG"));
```

Here **nl_init** copies language processing information into the process space. This information is then accessed by all other NLS routines. If you did not set the LANG environment variable or if LANG is set to an invalid value, **nl_init** initializes the application language environment to **n-computer**, if it is the first call to **nl_init**. Otherwise, the environment defaults to the last valid value set by LANG.

If LANG is set to a valid value, the **nl_init** call succeeds (initializing the process environment to the specified language) and returns the value "0". If LANG is not set or has a null value, the **nl_init** call succeeds (initializing the process environment to language **n-computer**) and returns the value "0".

Tools

The letters, punctuation marks, numbers, and symbols used by different languages have properties that vary with the language. For example, as pointed out earlier, the way in which letters are collated is a function of language. Upshifting or downshifting a letter can produce different results in different languages. Additionally, the size of data object in which a character is stored varies among languages. Asian languages require 16 bits (two bytes) per character; other languages require only 8 bits (one byte) per character. NLS provides tools to:

- identify the size of a character (one byte versus two bytes)
- perform character pointer manipulation, independent of the size of a character (one byte versus two bytes)
- upshift and downshift a character by the rules of particular language and locale
- identify the type of a character (i.e., an alphabetic character, punctuation mark, control character, digit, or hexadecimal digit)
- perform language-sensitive comparisons of characters and stings

Identifying the Size of a Character

The routines **firstof2**, **secof2**, and **byte_status** are used in a program to determine whether a byte value represents an 8-bit character or part of a 16-bit character. The syntax for the routines is:

```
#include <nl_type.h>
int c, laststatus;

firstof2(c);
secof2(c);
byte_status(c,laststatus);
```

firstof2 Accepts an integer value that contains a single byte. Assuming that **nl_init** has been successfully called, it returns a non-zero value if the byte may be the first byte of a 2-byte character; **firstof2** returns the value "0" if the value passed represents a 1-byte character.

When your program executes the **nl_init** statement, it loads information onto the process stack. The **firstof2** routine accesses this information to first determine if the language specified by the user requires two bytes to represent character data. If the language may require two bytes to represent character data, other information placed on the process stack by **nl_init** tells **firstof2** if the byte value passed to it is the first half of a 2-byte character. The range of values that constitute a valid **firstof2** value varies between languages. A value that is a valid **firstof2** for the Traditional Chinese character set is not necessarily a valid **firstof2** for the Japanese character set, even though both use an HP-15 encoding scheme.

secof2 Accepts an integer value that contains a single byte. It returns a non-zero value if the byte may be the second byte of a 2-byte character, and zero if it is not. Both **firstof2** and **secof2** require that a character string or a byte stream be examined from its beginning. The information returned is meaningless if a random byte from the middle of a stream is passed.

Like **firstof2**, **secof2** accesses the information placed on the process stack by **nl_init**. It also checks if the language specified by the user may require two bytes to represent a character. If so, it then uses other information placed on the stack by **nl_init** to determine if the byte is the second half of a 2-byte character. The range of values that constitute a valid **secof2** value varies between languages.

byte_status Combines the function of **firstof2** and **secof2**. If you pass **byte_status** an integer (that contains a byte) and the result of the last **byte_status** call, **byte_status** reports whether its integer argument represents a single-byte character, the first byte of a 2-byte character or the second byte of a 2-byte character. Like **firstof2** and **secof2**, **byte_status** relies on information loaded by **nl_init** to tell it whether the language specified by the user may contain 2-byte character data, what byte values are valid values for the first byte of a 2-byte character, and what byte values are valid values for the second byte of a 2-byte character.

The macros **FIRSTof2**, **SECof2**, and **BYTE_STATUS** provide the same function as the library routines **firstof2**, **secof2**, and **byte_status**, respectively. The syntax for the routines is:

```
#include <nl_ctype.h>
int c, laststatus;

FIRSTof2(c);
SECof2(c);
BYTE_STATUS(c,laststatus);
```

Character Pointer Manipulation

Because a character may now consist of more than a single byte, C pointer manipulation (i.e., `*p`, `*p = c`, `p++`, `*p++`, and `*p++ = c`) can no longer be used in a character-sensitive way. NLS provides a set of macros that help you manipulate character pointers. The syntax for the routines is:

```
int c;
unsigned char *p;

CHARAT(p);
ADVANCE(p);
CHARADV(p);
PCHAR(c,p);
PCHARADV(c,p);
```

CHARAT Accepts an argument which is assumed to be a pointer, pointing at the first byte of either a 1-byte or 2-byte character. In either case, it returns the integer value of the character. The value is less than 255 for a single-byte character, and greater than 255 for a 2-byte character. This is analogous to `*p`.

ADVANCE Accepts an argument which is assumed to be a pointer to the beginning of either a 1-byte or 2-byte character. It advances the pointer past the last byte of the character. This is analogous to `p++`.

CHARADV Combines the functions of **CHARAT** and **ADVANCE** into a single routine. It accepts an argument which is assumed to be a pointer to the beginning of either a 1-byte or 2-byte character. It returns the integer value of the character to which it is pointing and then advances the pointer past the last bye of the character. This is analogous to `*p++`.

PCHAR Accepts an integer argument which is assumed to contain the value of either a 1-byte or 2-byte character. At the position pointed to by argument `p`, it places the individual byte(s) of the character. If the character is a 2-byte character, it places the most significant byte first. This is analogous to `*p=c`.

PCHARADV Accepts an integer argument which is assumed to contain the value of either a 1-byte or 2-byte character. At the position pointed to by argument `p`, it places the individual byte(s) of the character. If the character is a 2-byte character, it places the most significant byte first. It then, advances the value of `p` such that it points past the last byte of the character.

Upshifting/Downshifting Characters

The way a character is upshifted (converted to its uppercase representation) and downshifted (converted to its lowercase representation) varies between languages, and even between locations where one language is spoken. For example, a French speaking resident of France removes the **diacritical** marks when upshifting characters. Thus, the character "ô" upshifts to "O" in French. However, a French speaking resident of Canada will retain the diacritical marks when upshifting characters. Thus, the character "ô" upshifts to "Ô" in Canadian French.

NLS provides routines that upshift and downshift characters in a language-sensitive way. The syntax of the routines is:

```
int c;

toupper(c);
tolower(c);
_toupper(c);
_tolower(c);
```

`toupper(c)` — If `c` represents a valid lowercase letter for the current language, `toupper(c)` returns the integer character code of the upshifted value of `c`; otherwise, `toupper(c)` returns `c` unaltered.

`tolower(c)` — If `c` represents a valid uppercase letter for the current language, `tolower(c)` returns the integer character code of the downshifted value of `c`; otherwise, `tolower(c)` returns `c` unaltered.

`_toupper(c)` — If `c` represents a valid lowercase letter for the current language, `_toupper(c)` returns the integer character code of the upshifted value of `c`; otherwise, `_toupper(c)` returns an undefined value.

`_tolower(c)` — If `c` represents a valid uppercase letter for the current language, `_tolower(c)` returns the integer character code of the downshifted value of `c`; otherwise, `_tolower(c)` returns an undefined value.

The calls `toupper`, `_toupper`, `tolower`, and `_tolower` return the integer character code of the upshifted/downshifted character `c`. These routines have been part of the standard C offering for many years; though in the past, they have worked only for ASCII character codes.

These routines have been modified such that if `nl_init` has been successfully called, they will correctly upshift and downshift the character passed to them, according to the rules of the language specified when `nl_init` was called.

If the call to `nl_init` fails, or if LANG is not set or set to an invalid language or `n-computer` when `nl_init` was called, the routines operate only on ASCII characters. All other character codes passed to these routines are returned unaltered.

NOTE

The routines `nl_toupper` and `nl_tolower` supply similar capabilities, but are provided for historical reasons only. They may not be supported in future releases of HP-UX and should not be used.

Identifying Character Traits

NLS also provides tools to identify traits of a character. The syntax of the routines is:

```
int c;

isalpha(c);
isupper(c);
islower(c);
isdigit(c);
isxdigit(c);
isalnum(c);
isspace(c);
ispunct(c);
isprint(c);
isgraph(c);
iscntrl(c);
isascii(c);
```

`isalpha(c)`	Returns a non-zero integer if `c` is an alphabetic character.
`isupper(c)`	Returns a non-zero integer if `c` is an uppercase alphabetic character.
`islower(c)`	Returns a non-zero integer if `c` is a lowercase alphabetic character.
`isdigit(c)`	Returns a non-zero integer if `c` is digit.
`isxdigit(c)`	Returns a non-zero integer is c is a hexadecimal digit.
`isalnum(c)`	Returns a non-zero integer if `c` is an alphanumeric character.
`isspace(c)`	Returns a non-zero integer if `c` creates "white space" in displayed text (e.g., tab, new-line, and space).
`ispunct(c)`	Returns a non-zero integer if `c` is a punctuation character.
`isprint(c)`	Returns a non-zero integer if `c` is a printing character.
`isgraph(c)`	Returns a non-zero integer if `c` is a character with a visible representation.
`iscntrl(c)`	Returns a non-zero integer if `c` is a control character.
`isascii(c)`	Returns a non-zero integer if `c` is a ASCII character.

The `ctype` package of routines (`isalpha`, `isupper`, `isalnum`, `isspace`, `isgraph`, `iscntrl`, etc.) has been part of the standard C offering for many years. In the past these routines worked only for ASCII character codes. However, they have been modified such that if `nl_init` is successfully called, they will correctly identify the traits of characters in all languages.

If the call to `nl_init` fails, if LANG is set to `n-computer` when `nl_init` is called, or if LANG is not set when `nl_init` is called, the `ctype` routines operate only on ASCII characters. All other character codes passed to `ctype` are treated as undefined and will return "0".

NOTE

The routines `nl_ctype` (`nl_isalpha`, `nl_isupper`, `nl_isalnum`, `nl_isalnum`, `nl_isspace`, `nl_isgraph`, `nl_iscntrl`, etc.) supply similar capabilities, but are provided for historical reasons only. They may not be supported in future releases of HP-UX and should not be used.

Comparing Strings, Comparing Characters

The order in which character strings are sorted is language-dependent. Traditionally, most ordering is done based on ASCII values. With the extension of the ASCII character set to ROMAN8 for support of other languages, the ordering of a character within a character set no longer coincides with the character's collation order. For example, "å" follows "b" in the character set ordering but is sorted before "b" in many cases. This situation makes collation based on each character's code inappropriate when internationalizing software.

In addition, collation based on character code does not provide true dictionary sorting even in the case of the ASCII character set. Dictionary collation sorts "a" after "A" and before "B", whereas ASCII based collation sorts "a" after both "A" and "B". The following is an example of a list of unsorted strings followed by the same list based on the currently available collation, and a language-sensitive collation.

Table 4-4. German/n-computer Sorted Words

Unsorted Words	Sorted n-computer	Sorted German
car	Airplane	Airplane
Airplane	Zebra	åpfel
bird	bird	bird
åpfel	car	car
Zebra	åpfel	Zebra

Beyond the ordering of individual characters, some languages designate that certain characters be treated in a special way. For example, in some languages groups of characters are **clustered** and treated as a **single character**. For example, in Spanish "ll" is treated as a single character; it is sorted after "l" but before "m". Similarly, the "ch" in Spanish is treated as a single character; it is sorted after "c" but before "d":

Table 4-5. Spanish/n-computer Sorted Words

Unsorted Words	Sorted n-computer	Sorted Spanish
maíz	chaleco	cuna
loro	cuna	chaleco
llave	día	día
día	llava	loro
cuna	loro	llava
chaleco	maíz	maíz

When sorting strings in some languages, a single character is **expanded** and treated as if it were really **two characters**. For example, when sorting strings in German, "ß", the sharp S is treated as if it is "ss".

Table 4-6. German/n-computer Sorted Words

Unsorted Words	Sorted n-computer	Sorted German
Roßhaar	Rosselenker	Rosselenker
Rostbratwurst	Rostbratwurst	Roßhaar
Rosselenker	Roßhaar	Rostbratwurst

In some languages, certain characters are **ignored** when collating strings.

Two routines `nl_strcmp` and `nl_strncmp` are available which provide full dictionary sort in a language-sensitive way. The collation is performed based on the rules for the language loaded with the last successful call to `nl_init`. Be sure to include these types in the program before calling the routines. The syntax is:

```
#include <string.h>

unsigned char *s1, *s2;
int n;
nl_strcmp(s1,s2);
nl_strncmp(s1,s2,n);
```

The routines `nl_strcmp` and `nl_strncmp` perform two pass collation. In the first pass the strings are compared with accents and cases and stripped as appropriate for the language. For example, in German "a", "å", and "A" are all given the same weight. In this pass, character groups which are treated as a single unit, characters which are expanded and characters which were ignored are accounted for. If after the first pass the strings are

equal, a second pass is performed. In the second pass, accents and cases are prioritized if there is conflict. For example, in German "A" is given a higher priority then "a", which is higher priority then "å".

The arguments *s1* and *s2* point to strings. Here a string is defined as an array of storage locations, of which each storage location contains the numeric value of a character code. Thus, the string may be an array of 16-bit storage locations, allowing the comparison of strings of 8- or 16-bit characters. Each string is **null** terminated.

An integer greater than, equal to, or less than zero is returned, depending on whether *s1* is, respectively, greater than, equal to, or less than *s2*. The routine `nl_strncmp` makes the same comparisons as `nl_strcmp`, but looks at *n* characters. If *n* is less than or equal to zero, the comparison yields equality.

NOTE

The routines `strcmp8` and `strcmp16` provide capabilities similar to that provided by `nl_strcmp` and `nl_strncmp`, but are provided for historical reasons only. They may not be supported in future releases of HP-UX and should not be used.

Local Customs

Languages differ in the conventions used to represent certain information. These conventions differ not only between languages, but between locales where a single language is spoken.

For example, the way time and date are represented differ not only between languages but between speakers of the same language in different locales. A German speaking person represents the date "July 4, 1987" as 04.07.87. An American represents "July 4, 1987" as 07/04/87. However, an English speaking resident of London, England interprets the date 07/04/87 as "7 April, 1987".

Similarly, languages and locales differ in the way numbers are represented. Some areas use "." as the radix symbol; the symbol that separates the fractional and whole number portions of floating-point numbers. Other areas use "," as the radix symbol.

For example, engineers who speak American, Arabic, English, Hebrew, and Japanese all agree that 3.141 is a reasonable approximation for the value of π. However, engineers who speak Arabic (in the Western regions), Danish, Dutch, Finnish, French, German, Greek, Italian, Portuguese, Spanish, and Swedish interpret 3.141 as **three thousand, one hundred forty-one** which is a terrible approximation for π. Instead, they would approximate π as 3,141.

The symbol that separates the thousands place from the hundreds place also varies from locale to locale. NLS provides routines that allow your program to handle these locale differences.

Numeric Formatting

NLS provides several routines to convert string representation of numeric information into numeric values. The routines correctly handle the different radix characters. By using the NLS routines, your program can convert string representations of numeric values into a standard internal representation (such as a double), allowing the program to use the data in numeric calculations. NLS also provides routines that convert a standard internal representation of a number into a locally formatted string representation of that number. The syntax for these routines is:

```
#include <stdio.h>
double value;
int ndigit;
char *buf, *str, **ptr, *format, format, *s;
FILE *stream;

atof (str)
strtod (str, ptr)
gcvt (value, ndigit, but)
nl_printf (format [,arg]...)
nl_fprintf (stream, format [,arg]...)
nl_sprintf (s, format [,arg]...)
nl_scanf (format [,pointer]...)
nl_fscanf (stream, format [,pointer]...)
nl_sscanf (s, format [,pointer]...)
```

atof — If **nl_init** has been successfully called, **atof** converts a string to a **double** correctly interpreting the radix character for the language specified by LANG.[1]

strtod — If **nl_init** has been successfully called, **strod** converts a string to a **double** correctly interpreting the radix character for the language specified by LANG.[1]

gcvt — If **nl_init** has been successfully called, **gcvt** converts a **double** to a string and places the correct radix character for the language specified by LANG.[1]

nl_printf — If **nl_init** has been successfully called, **nl_printf** will format floating point and decimal numbers with the correct radix character for the language specified by LANG and place it on standard output.[1]

nl_fprintf — If **nl_init** has been successfully called, **nl_fprintf** will format floating point and decimal numbers with the correct radix character for the language specified by LANG and place it on the named output.[1]

Flexible Formatting

Flexible formatting tools provide a way to read and write information in different locales that is order sensitive (such as the order of day and month, or the position of the currency symbol when reading and writing monetary values).

The routines `nl_printf`, `nl_sprintf`, and `nl_fprintf` function identically to their **non-nl_** counterparts. However, the **nl_** versions allow for optional positional indicators in the format string. These positional indicators are of the form *%N$specifier*, where *specifier* is the format conversion specifier (such as `f` for a floating-point value) and *N* is position of the parameter to which specifier applies. Thus, `%2$f` tells `nl_printf` to output the contents of the second argument in its argument list, converting it to a floating-point number. Consider the following example:

```
nl_printf((catgets(nlmsg,NL_SETN,3,"The date is %1$d/%2$d/%3$d")),month,day,yr);
```

Thus, the default message is `The date is %1$d/%2$d/%3$d`. This tells `nl_printf` to take the first argument (`month`), convert it to a signed decimal, and output it, followed by the "/" and so forth.

However, a person localizing this message for German might supply the following in the message catalogue:

```
3 Heute ist:  %2$d.%1$d.$3$d
```

When this message is read by `catgets`, `nl_printf` takes the second argument (`day`), converts it to a signed decimal, and outputs it followed by a period (.) and so forth. In this fashion, the localizer forces the output to be of the form *day.month.year*. The program source code need not be changed.

Like the `printf` family of routines, if the `nl_init` routine has been successfully called, the `nl_printf` family of routines format numeric values with the radix character that is correct for the language specified by LANG.

NOTE

The routines `printmsg`, `fprintmsg`, and `sprintmsg` supply capabilities similar to those supplied by `nl_printf`, `nl_fprintf`, and `nl_sprintf`. However, these routines are provided for historical reasons only. They may not be supported in future releases of HP-UX and should not be used.

Just as the `nl_printf` family of routines allow flexible formatting of output, the `nl_scanf` family of routines allow flexible formatting of input. The routines `nl_scanf`, `nl_fscanf`, and `nl_sscanf` allow positional indicators in the input format string. The function and form of the positional indicators is identical to that of those used with `nl_printf` family of routines.

Placing the format strings of the `nl_scanf` family of routines in a message catalogue allows the localizer to specify an input order that is appropriate for that locale.

Occasionally, the you will build a string by a series of call to `printf`. If this code is being prelocalized, it is better to replace the calls to `printf` with a single call to `nl_printf`. This does add complexity to the program since each segment must be stored in a separate string, but makes localization possible.

Other Local Customs

Using the NLS `nl_langinfo` routine, your application program can retrieve other locale specific information. `nl_langinfo` accepts a single token which identifies the type of information requested, and returns a string containing the requested information. The information is retrieved from the table loaded when `nl_init` executed.

The following information can be accessed with `nl_langinfo`:

- name and abbreviation of each day of the week
- name and abbreviation of each month of the year
- radix symbol
- symbol used for thousands separator
- yes and no string
- currency symbol and its position
- time and date format string
- number of bytes per character

Note that `nl_langinfo` can provide the local currency symbol and its position. In some countries the currency symbol is placed before the amount; in other countries the currency symbol is placed after the amount. In a few places (such as Portugal), the currency symbol actually **replaces** the radix symbol when printing monetary values. For example:

US$1.25	Amount in US currency
1.25$CAN	Amount in Canadian currency
1$25	Amount in Portuguese currency

The routine `nl_langinfo` indicates the position of the currency symbol with:

+	indicates the symbol is placed after the amount
-	indicates the symbol is placed before the amount
.	indicates the symbol replaces the radix symbol

The routine **nl_langinfo** has the following syntax:

```
#include <nl_types.h>
#include <langinfo.h>

char *string;
nl_init (getenv("LANG"));
string=nl_langinfo(item);
```

The *item* parameter is actually one of the following token literals, defined in **langinfo.h**:

D_T_FMT	Date and time format string appropriate for the current language. This is the default format string used by **date**, **nl_cxtime**, and **nl_ascxtime** when no format is specified.
DAY_1	The name of the first day of the week ("Sunday" in English)
. . .	
DAY_7	The name of the seventh day of the week ("Saturday" in English)
ABDAY_1	The abbreviated name of the first day of the week ("Sun" in English).
. . .	
ABDAY_7	The abbreviated name of the seventh day of the week ("Sat" in English).
MON_1	The name of the first month in the Gregorian year.
. . .	
MON_12	The name of the twelfth month in the Gregorian year.

ABMON_1	The abbreviated name of the first month in the Gregorian year.
...	
ABMON_12	The abbreviated name of the twelfth month in the Gregorian year.
RADIXCHAR	Radix character ("decimal point" in English).
THOUSEP	Separator for thousands ("common" in English).
YESSTR	Affirmative response for yes/no questions.
NOSTR	Negative response for yes/no questions.
CRNCYSTR	Symbol for currency preceded by: "-" if it precedes the monetary value, "+" if it follows the monetary value, and "." if it replaces the radix symbol in the monetary value.
BYTES_CHAR	Maximum number of bytes per character for the character set used to represent the language.

Creating the User Interface

A major portion of the prelocalization process is actual design of the user interface. This can be done with NLS's message catalogue commands and routines or message catalogue system.

This section explains how localized message files are created and updated, where they are kept, and naming conventions.

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 errors into the product and reduces the time required to localize.

Localizable programs use text from an external message catalogue. Text includes:

- prompts
- commands
- messages
- softkey definitions
- format strings

This allows text to be translated (part of the localization process) without modifying program source code or recompiling. If the external message catalogue file is inaccessible for any reason (such as accidentally removed or not yet created), a hard-coded default can be provided or the program can default to the `n-computer` message catalogue.

A message catalogue 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 message catalogue system uses HP-UX commands to create the catalogues and C library routines to access those catalogues. Most message catalogue commands work only on C source code, but the library routines can be accessed from C, Pascal, and FORTRAN programs. However, **gencat** and **dumpmsg** accept any programming language, but characters are handled using C escaping conventions.

The message catalogue commands are:

- **findstr** - find strings for inclusion in message catalogues
- **gencat** - generate a formatted message catalogue 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 catalogue (**catgets** calls)
- **findmsg** - copy default messages from C source code (**catgets** calls)
- **dumpmsg** - reverse the effect of **gencat**; take a formatted message catalogue and make a modifiable message catalogue source file.

The C library routines useful with message catalogues are:

- **catgets** - get a message from the catalogue
- **catopen** - open a message catalogue
- **catclose** - close a message catalogue
- **catgetmsg** - get a message from the catalogue
- **nl_printf, nl_fprintf, nl_sprintf, printf, fprintf, sprintf** - print formatted output
- **nl_scanf, nl_fscanf, nl_sscanf, scanf, fscanf, sscanf** - read formatted input

The major steps an applications programmer follows to simplify the localization process is:

1. Write new code using **catopen**, **catgets**, **catgetmsg**, and **catclose**.
2. Modify existing programs using **findstr, insertmsg**.
3. Maintain message catalogues using **findmsg, gencat**, and **dumpmsg**.
4. Extract text from message catalogues using **dumpmsg, gencat**.

Guidelines for Using Message Catalogues

This section covers the overall guidelines you should follow when programming applications to make it easier to localize the software at the users location.

- Place **all** text that needs to be localized in the message catalogue. This includes: prompts, help text, error messages, format strings, softkey definitions, and command names.
- Provide a unique, unambiguous message for each situation. A single message in your native language may appear to cover several different situations. However, when the message is translated, each different situation may require a **different** local language translation.
- Include a message number for all messages that can be referenced. For example, include a message number for each error message. The number can serve as a **virtual language**. For example, suppose that an Italian speaking customer calls a Dutch support office for help. The error message on the customers screen is in Italian, but the Dutch support engineer does not speak Italian. An error message would serve as a **language cross-reference**.
- Allow **at least** 60% extra space in text buffers and screen layouts to allow for text expansion when messages are translated. It may take more space to convey a thought or an idea in another language.
- Decide what to do if a message catalogue cannot be found by your program. If the local language is vital to the operation of the program, you may want the program to issue a default error message and exit or allow the the program to continue to operate with a default language (such as `n-computer`).

Writing Programs that Access Message Catalogues

To write a program that accesses a message catalogue, you must create the source code and the message catalogue. This first section describes how to write a program and how to create the message catalogue that the program will access.

Steps for Creating Prelocalized Source Code

Here is a list of the steps necessary to write a program and then access the created message catalogue. Each step is described in detail in later sections.

1. At the beginning of the program, open the message catalogue using **catopen**.

   ```
   nlmsg_fd=catopen("name",0);
   ```

   Refer to the "Opening Message Catalogues" section for details.

2. Use the **catgets** or **catgetmsg** to retrieve messages from the catalogue.

   ```
   catgets(nlmsg_fd, set_num, msg_num, default_string);
   catgetmsg(nlmsg_fd, set_num, msg_num, buf, buflen);
   ```

   Refer to the section "Retrieving Messages" for details.

3. At the end of the program, include **catclose**.

   ```
   catclose(nlmsg_fd);
   ```

   Refer to the section "Closing Message Catalogues" for details.

4. Collect all messages used by your program in a text file for translation. Refer to the sections "Creating Message Catalogues" and "Translating Catalogues" for details.

5. At this time you have prelocalized the software. These last two steps show the localization process. After translation, execute **gencat** on the translated file to create the message catalogue.

   ```
   gencat  prog.cat  file1.msg  file2.msg ...
   ```

   Refer to the section "Generating Message Catalogues" for details.

6. Compile the application, set your environment variables, and run.

   ```
   cc prog.c -o prog
   LANG=french
   NLSPATH=./%L/%N.cat
   export LANG NLSPATH
   prog
   ```

   Refer to the section "Compiling" for details.

Opening Message Catalogues

To access a message catalogue, the program must first open the message catalogue, just as it must open any file that is to be read. Then, when a hard-coded string would normally be used in the program, an NLS routine is used to retrieve the string from a message catalogue instead. Finally, at the end of the program, the message catalogue must be closed.

Message catalogues are opened with NLS's `catopen` statement:

```
#include <nl_types.h>            /* include the header file where the
                                    nl_catd data type is defined */
nl_catd nlmsg_fd;               /* declaration of the message cat file
                                    descriptor */
nlmsg_fd=catopen("name",0);
```

The parameter *name* is the name of the message catalogue. The parameter *nlmsg_fd* is a message catalogue descriptor of type `nl_catd`, defined in the file `nl_types.h`.

When the program runs, `catopen` first searches at the user's environment to see if an environment variable named NLSPATH is set. NLSPATH is an environment variable that tells `catopen` where to search for a message catalogue. It consists of a series of colon (":") separated path names, which may contain the following wildcards:

Table 4-6. NLSPATH Wildcards

Wildcard	Expansion by catopen
%L	%L is replaced by the current value of the LANG environment variable when catopen executes.
%N	%N is replaced by a *name* which is typically the name of the program passed as an argument to catopen.

Consider the following example where NLSPATH and LANG are set to the values indicated below:

```
LANG=german
NLSPATH=/users/project/localized/%L/%N.cat:/users/project/test/%N.cat
```

The program contains the following `catopen` call:

```
nlmsg_fd=catopen("exfile",0);
```

When the `catopen` statement executes, it first checks to see if NLSPATH is set. Next, replacing `%N` with `exfile` and `%L` with "german" in the first path name specified by NLSPATH, `catopen` searches for a file whose path name is `/users/project/localized/german/exfile.cat`. If this file exists, `catopen` attempts to open it as the message catalogue. If it does not exist, `catopen` checks to see if another path is specified by NLSPATH. Then, replacing `%N` with `exfile` in the path name, `catopen` searches for a file whose path name is `users/project/test/exfile.cat`.

If `catopen` can't find a message catalogue with the path names specified in NLSPATH, it searches the default location:

```
/usr/lib/nls/%L/%N.cat
```

This is the location of the message catalogues for all HP-UX commands and HP supplied applications. If `catopen` cannot find a message catalogue in the locations specified by NLSPATH or in the default location, it returns the value "-1".

Retrieving Messages

Once the message catalogue is open, the program can use either of two NLS routines to retrieve messages from the catalogue, `catgets` or `catgetmsg`:

```
#include <nl_types.h>          /* includes declarations, including data
                                  type nl_catd */

nl_catd nlmsg_fd;              /* declares that nlmsg_fd (a variable to be
                                  used to hold the message catalogue file
                                  descriptor) is of type nl_catd  */

int set_num, msg_num, buflen;  /* set_num identifies the set of messages
                                  within the message catalogue. msg_num
                                  identifies the specific message, within
                                  the set, within the catalogue. buflen
                                  specifies the size of buffer into which
                                  a message will be read with catgetmsg */

char *default_string, *buf;    /* default_string is a pointer to  the default
                                  string that will be displayed by catgets,
                                  should catopen be unable to open the
                                  specified message catalogue. buf is a
                                  pointer to a character buffer into which
                                  catgetmsg will copy the message retrieved
                                  from the catalogue.  */

catgets(nlmsg_fd, set_num, msg_num, default_string);

catgetmsg(nlmsg_fd, set_num, msg_num, buf, buflen);
```

Both **catgets** and **catgetmsg** retrieve a message from a message catalogue. Both use a message catalogue file descriptor returned by **catopen** to identify the message catalogue to be read. Both use a message set number and a message number to identify the specific message in the catalogue to be read. However, these two routines differ in two key ways and these differences will determine which of the two routines should be used in a particular situation.

The **catgets** routine retrieves a message, copies it into a buffer local to the **catgets** routine and returns a pointer to this buffer. This means that each call to **catgets** overwrites the result of a previous **catgets** call. To preserve the message returned by **catgets**, it must be copied to another location. Alternately, each time you want to access the message, another **catgets** call (and another disc access) is required.

The **catgetmsg** routine retrieves a message, copies it into a buffer specified in its argument list and returns a pointer to the buffer. Using the buffer length supplied in its argument list, **catgetmsg** ensures that the message retrieved does not exceed the buffer length. Because the message is copied to a program specified buffer, subsequent use of the message can be made without additional **catgetmsg** calls (or disc accesses).

The **catgetmsg** routine is ideal for retrieving frequently used messages such as prompts and softkey definitions. A string array can be created by the program and the frequently used messages can be initialized into the array with **catgetmsg**. Then whenever one of these frequently used messages is needed, the program can use the message stored in the array; no time is lost accessing the disc.

The **catgets** and **catgetmsg** routines differ in a second key way. If **catopen** can't find and open a message catalogue, **catgetmsg** returns a pointer to the null string and sets **errno** to indicate the error while **catgets** returns a pointer to a default string. The pointer to the default is supplied by the programmer as an argument to **catgets**. Common practice is to hard-code the **n-computer** message as the default message, since logically there is no difference between a string and a pointer to a string. This has the additional advantage of making the program source more readable. Consider the following examples:

```
printf(catgets(nlmsg_fd,NL_SETN,1,"Hello World!\n"));

printf(catgets(nlmsg_fd,NL_SETN,1,default_ptr1));

catgetmsg(nlmsg_fd,NL_SETN,1,message[1],MAX_LENGTH);
printf("%s",message[1]);
```

In the first two **printf** statements, the retrieved message is the format string; in the last **printf**, it is the first argument.

Closing Message Catalogues

Finally, at the end of the program the message catalogue should be closed, just as any open file should be closed. To close a message catalogue, use NLS's **catclose** call:

```
nl_catd nlmsg_fd;

catclose(nlmsg_fd);
```

Creating Message Catalogues

Once the program contains the **catopen**, **catgets**, **catgetmsg**, and **catclose** calls, you are ready to create the message catalogue. The message catalogue is built from text files by using the **gencat** command.

The text file contains logical groups, called **sets**, of messages. At least one set must be defined for each message catalogue. Message sets allow the programmer to group similar messages together within a catalogue. For example, **set 1** might contain all prompts, and **set 2** might contain all error messages. The general format of the text file from which the message catalogue is generated is:

```
$set 1
1 Text of message number 1
2 Text of message number 2
$ A "$" in the first column, followed by a blank, indicates that the\
 text on that line is a comment.  This allows you to document the\
 messages in the message catalogue.
3 Text of message number 3
   ...
$set 2
1 Text of message number 1
$ Comment
2 Text of message number 2
3 Text of message number 3
   ...
```

Messages derive their number from the number on the line and the set from the **$set** declaration which immediately precedes the message line. In the preceding example, **$set 1** specifies that the messages that follow, up to the next **$set** declaration, belong to message **set 1**.

The remainder of the text file consists of comments and messages. A comment line consists of a single **$** character in column 1, followed by a space and the text of the comment. All text up to the next new-line is considered to be part of the comment. Each message is a line of text beginning with a message number, followed by a single space and the text. All text up to the next new-line is considered to be part of the message.

The message text can contain leading and trailing spaces, as well as control characters (such as \n and \t). For example, the following is a portion of a text file used to generate a message catalogue:

```
$set 1
1 Please enter your name:
2 %s, %s %s
3 \nHello %s!
4 Welcome to the Widget Company automated test system!\n\n
   ...
39 As of %s, we have tested a grand total of %d battery powered Widgets!\n
   ...
59 The following capabilities are possible:\n\n
60 o list all battery test results\n
61 o list only battery test failures, or\n
62 o plot percentage of battery failures for a period of the last 12 months.\n\n
$
$ Notice that message 63 is missing.  It was removed during the last update.
$
64 Press a softkey or type a code number to tell me what to do:\n\n
$
$ The following paints the menu for function selection.
$
65 Function Description                          Softkey Label     Softkey #\n\n
66 1. List All Battery Test Results             LIST    RESULTS     f1\n
67 2. List Only Battery Test Failures           LIST    FAILURES    f2\n
68 3. Plot %% of Battery Failures               PLOT    HISTORY     f3\n
69 8. Exit - Return to Main Menu                EXIT                f8\n
   ...
```

Translating Catalogues

At this point, you are finished with the prelocalization process. The rest of this section describes localization procedures.

Once the set declarations, messages, and comments are collected into text files, the files are ready to be translated. For example, suppose that you collected all messages used by a program into the following files: `msg_src.a` and `msg_src.b`. You would give copies of `msg_src.a` and `msg_src.b` to whoever is translating the messages into other languages. Suppose that you are translating into only German, French, and American. Give copies of `msg_src.a` and `msg_src.b` to the French and German translators. Use the suffixies `.fr`, `.sp`, and `.am` to identify the translated French, Spanish, and American message files.

Generating the Message Catalogue

The translated text files can then be converted into the actual message catalogue(s). This is done by executing the **gencat** command:

```
gencat  prog.cat file1.msg  file2.msg ...
```

The **gencat** routine concatenates the *.msg*[1] files into one message catalogue source, formats the messages it finds in these files, and then merges the messages with any existing messages in **prog.cat**. If **prog.cat** does not exist before **gencat** is executed, **gencat** creates it. The **gencat** routine overrides existing messages if a conflict with an already defined message number exists.

The message catalogue must be placed in a location where it will be found by **catopen**; thus, in one of the paths specified by NLSPATH or in the default location **/usr/lib/nls/%L/%N.cat**.

After receiving the translated files, you are ready to generate the message catalogues:

```
cd test_directory
mkdir french spanish american
gencat french/prog.cat msg_src.a.fr msg_src.b.fr
gencat spanish/prog.cat msg_src.a.sp msg_src.b.sp
gencat american/prog.cat msg_src.a.am msg_src.b.am
```

Once **gencat** has generated the message catalogue, the catalogue can be accessed by **catopen**, **catclose**, **catgets**, and **catgetmsg**.

Compiling

Now compile the application (**prog.c**), set the NLSPATH and LANG environment variables, and test the use of message catalogues:

```
cc prog.c -o prog
LANG=french
NLSPATH=./%L/%N.cat
export LANG NLSPATH
prog
```

The application **prog** will find the translated French message catalogue at **./french/prog.cat** and produce messages in French. All other processing will also be in French.

[1] The conventions for suffixes are: **.cat** for final message catalogues, **.msg** for message files or input files to **gencat**, and **.str** for string files or output of **findstr**.

Message Catalogues for Existing C Programs

NLS provides a set of commands that make it easy to modify an existing C program to use message catalogues. These commands search for hard-coded strings, create new programs in which the hard-coded strings are replaced by **catgets** calls, and create the source for the **gencat** command.

findstr

The command **/usr/bin/findstr** searches C source programs for strings enclosed in double quotes. Strings embedded in comments or imbedded in **catgets** calls are ignored. It sends to standard output the following information for each string:

file_name offset strlen "string"

where *file_name* is the name of the C source file, *offset* is the byte offset from the beginning of the file where the *string* is found, *strlen* is the length of the string in bytes, and *string* is the actual string.

insertmsg

The command **/usr/bin/insertmsg** uses the output of **findstr** to create a new C source file in which the hard-coded strings are replaced by **catgets** calls. It writes to standard output the following information, needed by **gencat** to produce a message catalogue:

```
$set n
number message
number message
```

The new application created is named **nl_***originalname*. For example, if the original C source file was named *X*.**c**, **insertmsg** would create a new file named **nl_***X*.**c** in which hard-coded strings were replaced with **catgets** calls.

Using the **-h** option with **insertmsg** causes **insertmsg** to add a header to the beginning of the new file it creates. The header uses **#ifdef** constructs to conditionally add **include** and **define** statements to the new application. If you want to ensure that you always access the message catalog, do not use the **-h** option. Instead, add the following lines to the beginning of the file created by **insertmsg**:

```
#include <nl_types.h>
#define NL_SETN 1
```

The **NL_SETN** is the set number corresponding to the messages.

gencat

The command **/usr/bin/gencat** converts a text file of the form created by **insertmsg** into a message catalogue that is accessible by **catopen**, **catgets**, **catgetmsg**, and **catclose**.

Steps for Adding Message Catalogues to C Programs

If you just want to modify an existing C program to use message catalogues, perform the following steps:

1. Use the **findstr** command to locate all strings in the source file and store in an output file:

   ```
   findstr prog.c > prog.str
   ```

   Refer to the section "Execute findstr" for details.

2. Edit file **prog.str** to remove those strings which should not become part of the message catalog:

   ```
   vi prog.str
   ```

   Refer to the section "Edit the Output File" for details.

3. Execute the **insertmsg** command to create a new file in which hard-coded strings are replaced by calls to **catgets**. Store **insertmsg**'s output in a file:

   ```
   insertmsg prog.str > prog.msg
   ```

   Refer to the section "Execute insertmsg" for details.

4. Edit the file **nl_prog.c**, created by **insertmsg**. Add these statements to the file:

   ```
   #include <nl_types.h>
   #define NL_SETN 1
   ...
   nl_catd nlmsg_fd;            /* define the message cat file descriptor */
   nlmsg_fd=catopen("prog",0);
   ...
   catclose(nlmsg_fd);
   ```

   Refer to the section "Adding NLS Statements" for details.

5. Use the text file **prog.msg** as the source for message translation. After the messages are translated, generate the message catalogue by executing:

```
gencat prog.cat prog.msg
```

   Store the catalogue at a location where it will be found by **catopen**. Refer to the section "Generating the Message Catalogue" for details.

6. Compile your program by executing:

```
cc nl_prog.c -o prog
```

   Refer to the section "Generating the Message Catalogue" for details.

7. Set the LANG and NLSPATH environment variables, and execute the program:

```
LANG=german
NLSPATH=/users/project/test/%N.cat
export LANG NLSPATH
prog
```

   Refer to the section "Generating the Message Catalogue" for details.

Example of Modifying a C Program

The following example illustrates the process of modifying a C program to use message catalogues. This C program is stored in the file `simple.c`:

```
main{
unsigned char *lang;

nl_init((lang=getenv("LANG")));
printf("The language currently selected is %s\n",lang);
printf("Hello World!\n");
}
```

Execute findstr

First, from the shell execute the following command:

```
findstr simple.c > simple.str
```

The file `simple.str` now contains the following:

```
simple.c 42 6 "LANG"
simple.c 60 42 "The language currently selected is %s \n"
simple.c 117 16 "Hello World!\n"
```

NOTE

No editing should be done on the string file (`simple.str` in this example) other than deleting unwanted files.

Edit the Output File

Notice that the first string found was the name of the LANG environment variable passed to `getenv`. This is not a string the localizers should change, and therefore it should not be in the catalogue. So, edit the file and remove the first line such that `simple.str` now contains:

```
simple.c 60 42 "The language currently selected is %s \n"
simple.c 117 16 "Hello World!\n"
```

Execute insertmsg

Next, execute the **insertmsg** command, which creates a new program named **nl_simple.c**. At the same time, store **insertmsg**'s output in a file to be used when generating the catalog:

```
insertmsg simple.str > simple.msg
```

The file **nl_simple.c** is created by **insertmsg** and contains the following:

```
main{
unsigned char *lang;

nl_init((lang=getenv("LANG")));
printf((catgets(nlmsg_fd,NL_SETN,1,"The language currently selected
is %s \n")),lang);
printf((catgets(nlmsg_fd,NL_SETN,2, "Hello World!\n")));
}
```

If the program file, **simple.c**, has been modified between the **findstr** and **insertstr** operations, the resultant new program file, **nl_simple.c**, could be incorrect.

The file **simple.msg**, also created by **insertmsg**, contains the following:

```
$set 1
1 The language currently selected is %s \n
2 Hello World!\n
```

Add NLS Statements

Next, edit the file **nl_simple.c** to add the **include**, **define**, **catopen**, **catclose** statements, and to declare the file descriptor for use by **catopen**. The modified file looks like this:

```
#include <nl_types.h>
#define NL_SETN 1

main{
unsigned char *lang;
nl_catd nlmsg_fd;

nl_init((lang=getenv("LANG")));
nlmsg_fd=catopen("simple",0);
printf((catgets(nlmsg_fd,NL_SETN,1, "The language currently selected
is %s \n")),lang);
printf((catgets(nlmsg_fd,NL_SETN,2, "Hello World!\n")));
catclose(nlmsg_fd);
}
```

Translate the File

Send the message source, **simple.msg**, to be translated; program source file, **nl_simple.c**, need not be sent.

Generate Message Catalogue

Finally, generate the message catalogue, compile the application, and set the environment variables. Then, you are ready to run the application:

```
gencat /users/project/test/simple.cat simple.msg
cc nl_simple.c -o simple
NLSPATH=/users/project/test/%L/%N.cat
export NLSPATH
LANG=german
export LANG
simple
LANG=french
export LANG
simple
```

The file **simple.msg** can now be deleted. It is the source used to generate the message catalogue and can be retrieved if needed by executing either the **dumpmsg** command or the **findmsg** command:

```
dumpmsg simple.cat > simple.msg
```

or

```
findmsg nl_simple.c > simple.msg
```

Updating a C Program and Its Message Catalogue

Once your C program contains message catalogue calls, you will occasionally need to to update the program and maintain the message catalogue. A set of NLS commands can be used to make it easy to update a program and its catalogue.

Steps for Updating a Message Catalogue

These are the steps for updating a program and message catalogue. An explanation of each step follows:

1. Modify the source code as if message catalogues did not exist. Refer to the "Modifying Source" section for details.

2. Determine the greatest message number used by the program.

   ```
   dumpmsg prog.cat
   ```

   Refer to the section "Determining Message Number" for details.

3. Execute `findstr`:

   ```
   findstr  prog.c > prog.str
   ```

   Refer to the section "Executing findstr" for details.

4. Edit `prog.str`. Refer to the section "Edit the Output File" for details.

5. Execute `insertmsg`, where *XX* is the message number with which the new messages should start:

   ```
   insertmsg -nXX prog.str > prog.msg
   ```

   Refer to the section "Execute insertmsg" for details.

6. Execute `gencat`:

   ```
   gencat prog.cat prog.msg
   ```

   Refer to the section "Execute gencat" for details.

7. Compile the new source:

   ```
   cc nl_prog.c -o prog
   LANG=german
   NLSPATH=/users/project/test/%N.cat
   export LANG NLSPATH
   prog
   ```

   Refer to the section "Compile" for details.

Modify Source

To update the program, modify the source code `prog.c` just as if message catalogue calls were not used in the program. Once the source has been modified, you are ready to convert the modified source lines to use message catalogue calls.

Determining Message Number

Each message within a set in a message catalogue must have a unique message number. When adding new messages to a program, such as when updating the program, you must make sure that the new message numbers assigned do not conflict with those already used in the message catalogue.

Find the number of the next available message by executing one of the following:

```
dumpmsg prog.cat
```

or

```
findmsg prog.c
```

Here, `prog.cat` is the message catalogue for the program and `prog.c` is the source file containing the `catgets` calls. Either of these commands will list to standard output the message number and message text of all messages in the catalogue.

Execute findstr

Next, execute the `findstr` command to locate all strings that are not already included in `catgets` calls. This way `findstr` will locate all of the new strings added when you updated `prog.c`:

```
findstr prog.c > prog.str
```

This is recommended when you have added a large number of strings. If this is not the case, adding the `catgets` calls by hand will be easier.

Edit the Output File

Next, edit `prog.str` to remove lines which contain strings that should not be part of the message catalogue.

```
vi prog.str
```

Execute insertmsg

Now, execute **insertmsg** to replace the hard-coded strings with **catgets** calls. Use the **-n** option to tell **insertmsg** the beginning message number for the new messages. For example, if you find that messages 1 through 89 are already used by existing message catalogue calls, execute **insertmsg -n90** ... to begin numbering the new messages with message number 90:

```
insertmsg -nXX prog.str > prog.msg
```

Since **nl_prog.c** (the file created by **insertmsg**) already contains the appropriate declarations as well as **catopen** and **catclose** statements, it should not be necessary to modify **nl_prog.c**. The file **prog.msg** now contains the messages to be added to the catalogue. Copies of this file can be given to the person(s) translating the program messages.

Execute gencat

Once the messages are translated, you are ready to merge them into the existing catalogues by executing the **gencat** command:

```
gencat prog.cat prog.msg
```

If there is a conflict with an already existing message number, **gencat** will override the existing messages. To prevent this make sure each message number is unique.

Compile

Finally, compile the new source, set the environment variables and execute the new program:

```
cc nl_prog.c -o prog
LANG=german
NLSPATH=/users/project/test/%N.cat
export LANG NLSPATH
prog
```

Native Language Support Library and Commands

A

This appendix describes HP-UX library calls and commands relevant to NLS.

Library Routines

The NLS library routines are included in the standard C library `/usr/lib/libc.a`. The "NLS Library" table lists the C library routines for NLS. For more details refer to the appropriate page in sections *3(C)* and *3(S)* of the *HP-UX Reference* or the the chapters "Developing International Applications" and "Task Reference of NLS Routines".

Table A-1. NLS Library

Name	Description
`byte_status`	return `onebyte`, `firstof2`, or `secof2`, indicating single byte character, second byte of 2-byte character, or first byte of 2-byte character as defined in `nl_ctype.h` (refer to the *nl_tools_16(3C)* manual page)
`catopen`	open a message catalogue (refer to *catopen(3C)* manual page)
`catclose`	close a message catalogue
`catgets`	get a program message
`catgetmsg`	get native language message from catalogue
`conv`	character casefolding routines
`ctype`	character classification
`firstof2, secof2`	return true if byte is first (or second) of 2-byte character (refer to the *nl_tools_16(3C)* manual page)
`gcvt`	convert binary numbers to string numerics (refer to *ecvt(3C)* manual page)
`nl_init`	initialize the program with native language processing information (refer to the *nl_init(3C)* manual page)
`nl_langinfo`	get native language information
`nl_ascxtime`	convert a time value to a string (refer to *ctime(3C)* manual page)
`nl_cxtime`	time conversion routines (refer to *ctime(3C)* manual page)
`printf`	print formatted output routines `nl_printf`, `nl_fprintf`, `nl_sprintf` `printf`, `fprintf`, and `sprintf`
`scanf`	formatted input conversion routines `nl_scanf`, `nl_fscanf`, `nl_sscanf` `scanf`, `fscanf`, and `sscanf`
`string`	string comparison routines `nl_strncmp` and `nl_strcmp`
`strtod, atof`	convert string numeric to binary number routines

Other HP-UX system and library calls are 8-bit compatible, with the following exceptions. Localized versions exist for many of these (shown in the "NLS Library" table) and should be used for new program development.

Table A-2. HP-UX System and Library Calls Not Handling NLS

Name	Description
`qsort`	quick sort (refer to *qsort(3C)*)
`regex`	regular expression compile/execute (refer to *regex(3X)*)

Commands

The commands listed in the table "NLS Commands" 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 chapters "Developing International Applications" and "Task Reference of NLS Routines".

Table A-3. NLS Commands

Name	Description
`dumpmsg`	reverse the effect of **gencat**; take a formatted message catalogue and make a modifiable message catalogue source file (refer to the *findmsg(1)* manual page)
`findmsg`	extract strings from prelocalized C programs for inclusion in message catalogues (refer to the *findmsg(1)* manual page)
`findstr`	find strings in programs not previously localized for inclusion in message catalogues (refer to the *findstr(1)* manual page)
`gencat`	generate a formatted message catalogue file (refer to the *gencat(1)* manual page)
`insertmsg`	Uses output from **findstr** to both create a preliminary message file and to create a new C program with calls to the message catalogue (refer to the *gencat(1)* manual page)

Other HP-UX commands may have NLS support to some degree. In the *HP-UX Reference* entry for a command, there is a catogory called "International Support". This category indicates to what level the command supports NLS. The possible values are:

Table A-4. Supported NLS Levels

Support Level	Level's Description
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 filenames	The command accepts files with names written in an 8-bit language. For example, `cut`.
customs	The command formats output appropriate to the user's language. For example, the `date` command formats the output according to the language set in the LANG environment variable.
messages	The command has the capability of accessing a message catalogue. For example, `cat`.
8-bit string	The command correctly parses through 8-bit strings and comments. An example of this is `cc`, which allows source files to have 8-bit strings in comments.
16-bit string	The command correctly parses through 16-bit strings and comments. An example of this is `cc`, which allows source files to have 16-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.

Task Reference of NLS Routines B

This chapter describes the C library routines and HP-UX commands that are used by Native Language Support. Each routine or command is described under the corresponding task you need to perform.

Library Routines

NLS library routines are listed below by task or function. This section is structured to provide quick access to routines by searching alphabetically for the task each performs. For more detail, see the manual pages for these routines in the *HP-UX Reference.*

Classify characters

All these routines have the same parameter:

routine(c)

where *routine* is any of the routines in `ctype`.

`isalpha`	*c* is a letter
`isupper`	*c* is an uppercase letter
`islower`	*c* is a lowercase letter
`isdigit`	*c* is a decimal digit
`isxdigit`	*c* is a hexadecimal digit
`isalnum`	*c* is an alphanumeric (letter or digit)
`isspace`	*c* is a character that creates "white space" in displayed text
`ispunct`	*c* is a punctuation character (neither control nor alphanumeric)
`isprint`	*c* is a printing character
`isgraph`	*c* is a printing character, like `isprint` except false for space
`iscntrl`	*c* is a control character
`isascii`	*c* is an ASCII character

Refer to the *ctype(3C)* manual page for more information. The `nl_ctype` counterparts are supported for historical reasons and should not be used.

Close message catalogue

`catclose(`*catd*`)`

Close the message catalogue identified by *catd*.

Refer to *catopen(3C)*.

Convert date/time to string

`nl_cxtime(`*clock, format*`)`
`nl_ascxtime(`*tm, format*`)`

The `nl_cxtime` 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, month and weekday names in the output string will be in the language of the currently loaded NLS environment (see *nl_init(3C)*). If `nl_init` has not been called successfully, the month and weekday names default to `n-computer`.

The `nl_ascxtime` routine is similar to `asctime`, but like `nl_cxtime` allows the date string to be formatted and the month and weekday names to be in the language currently loaded. However, like `asctime`, it takes *tm* as its argument.

Refer to *ctime(3C)* for these routines. The `nl_ctime` and `nl_asctime` counterparts are supported for historical reasons and should not be used.

Convert floating point to string

`gcvt(`*value, ndigit, buf*`)`

`gcvt` converts *value* to a null-terminated string of *ndigit* digits and returns a pointer *buf*.

Refer to *ecvt(3C)* for this routine. The `nl_gvct` counterpart is supported for historical reasons and should not be used.

Convert string to double precision number

`strtod(`*str, ptr*`)`
`atof(`*str*`)`

`strtod` returns as a double-prescision floating-point number, *ptr*, the value represented by the character pointed to by *str*. `atof` converts a string into a `double`. Refer to *strtod(3C)* for these routines. The `nl_strtod` counterpart is supported for historical reasons and should not be used.

Find strings for message catalogues

`findstr` *file*

`findstr` examines a *file* of C source code for uncommented string constants, which it places on standard output.

Refer to *findstr(1)*.

Get program message

catgets(*catd, set_num, msg_num, def_str*)

catgets reads messages where *catd* is the file descriptor pointing to the catalogue (file) containing the messages, *set_num* is the set number designating a group of messages in the catalogue, *msg_num* is the message number within that set, and *Def_str* is returned if *catd* is invalid.

Refer to *catgets(3C)* for more information.

Get message from catalogue

catgetmsg(*catd, set_num, msg_num, buf, buflen*)

where *catd* is the file descriptor pointing to the catalogue (file) containing the messages, *set_num* is the set number designating a group of messages in the catalogue, *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*.

Refer to *catgetmsg(3C)* for more information. The *getmsg(3C)* counterpart is supported for historical reasons and should not be used.

Initialize NLS environment

nl_init (*langname*)

Initialize the NLS environment of a program to the language specified by **langname**.

Refer to *nl_init(3C)* for more information. The **langinit** counterpart is supported for historical reasons and should not be used.

Insert calls

```
insertmsg [-nNumber] [-h] file
```

insertmsg examines the *file* and inserts calls to **catgets**.

Refer to *insertmsg(1)* for more information.

Open message catalogue

```
catd=catopen("name",0)
```

catopen opens a message catalogue and returns a catalogue descriptor. **catopen** searches the paths specified by NLSPATH when searching for the catalogue. (Refer to *environ(5)*.

Parsing multi-bytes

```
firstof2(c)
secof2(c)
byte_status(c,laststatus)
```

Multi-byte library routines and macros were created to help you parse multi-byte character data.

The routine, **nl_init**, must be called before any of the other 16-bit tools are called. **nl_init** 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 routines.

Print formatted output

```
nl_printf (format [ , arg ] ... )
printf (format [ , arg ] ... )
nl_fprintf (stream, format [ , arg ] ... )
fprintf (stream, format [ , arg ] ... )
nl_sprintf (s, format [ , arg ] ... )
sprintf (s, format [ , arg ] ... )
```

The conversion character *%* used in **printf** is replaced by the sequence *%digit$*, where *digit* is a decimal digit *n* in the range from 1 to 9. The conversion should be applied to the *n*th argument, rather than to the next unused one (you specify which parameter you want this conversion applied to). The conversion specifiers may contain the *%digit$* sequence, and it is your responsibility to make sure the numbering is correct. All parameters must be used exactly once. All routines listed correctly format with local radix if **nl_init** is called successfully.

Refer to *printf(3S)*. The *printmsg(3C)* counterparts are supported for historical reasons and should not be used.

Read/format/convert characters

```
nl_scanf(format [ , pointer]...)
scanf(format [ , pointer]...)
nl_fscanf(stream, format [ , pointer]...)
fscanf(stream, format [ , pointer]...)
nl_sscanf(s, format [ , pointer]...)
sscanf(s, format [ , pointer]...)
```

nl_scanf reads from the standard input stream. **nl_fscanf** reads from the named input *stream*. **nl_sscanf** reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. All routines listed read and convert string representations of numbers that include the local radix character as long as **nl_init** has been called successfully.

The conversion character *%* used in **printf** is replaced by the sequence *%digit$*, where *digit* is a decimal digit *n* in the range from 1 to 9. The conversion should be applied to the *n*th argument, rather than to the next unused one (you specify which parameter you want this conversion applied to). The conversion specifiers may contain the *%digit$* sequence, and it is your responsibility to make sure the numbering is correct. All parameters must be used exactly once. All routines listed correctly format with local radix, if **nl_init** is called successfully.
Refer to *scanf(3C)* for more information.

Retrieve language information

`nl_langinfo(`*item*`)`

where *item* is one of several tokens or parameters. 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`, `nl_cxtime`, and `nl_ascxtime`.
`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 yes/no questions
`NOSTR`	negative response for yes/no questions
`CRNCYSTR`	symbol for currency preceded by "-" if it precedes the number, "+" if it follows the number, "." if it relaces the radix. (e.g., "`-DM`" for German, "`+ Kr`" for Danish.)

The routine `nl_langinfo` retrieves a null-terminated string containing information appropriate for a language or cultural area. The `langtoid`, `idtolang`, `langinfo` and `currlangid` counterparts are supported for historical reasons and should not be used.

Refer to *langinfo(3C)* for more information.

String collation (Non-ASCII)

`nl_strcmp(`*s1, s2*`)`
`nl_strncmp(`*s1, s2, n*`)`

The routine `nl_strcmp` compares string *s1* and *s2* according to the language dependent collating sequences. The routine `nl_strncmp8` makes the same comparison but looks at only *n* characters.

Refer to the *string(3C)* manual page for more information. The *nl_string(3C)* counterparts are supported for historical reasons and should not be used.

Translate characters

`toupper(c)`
`tolower(c)`

`toupper` and `tolower` translate characters from -1 through 255 to uppercase or lowercase appropriately.

Refer to the *conv(3C)* manual page for these routines. The *nl_conv(3C)* counterparts are supported for historical reasons and should not be used.

Character Representation and Sets C

HP-UX NLS is based on many languages, such as Japanese, Greek, and Arabic, and several character sets. Facilities to handle these character sets are built into the operating system. Language tables and files associated with supported languages are 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 ASCII. 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 area characters** (see the table "8-bit Character Set Structure"). New characters are added by defining code values in the range 161 to 254.

Table C-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	USASCII GRAPHIC (printable) CHARACTERS (33-126)				127	EXTENDED CHARACTERS (128-160)	EXTENDED PRINTABLE CHARACTERS (161-254)						
0 0 0 1	1															
0 0 1 0	2															
0 0 1 1	3															
0 1 0 0	4															
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															255

NLS supports six 8-bit character sets: ROMAN8, KANA8, GREEK8, ARABIC8 and TURKISH8. There are also line drawing and math character sets supported by Hewlet-Packard that are not language oriented. These 8-bit character sets are shown later.

HP 8-bit character sets support all ASCII characters in addition to the characters needed to support several Western European-based languages, Mideast African languages, and Katakana. The exception to this is that the graphic (on the keyboard/overlay) for back slash ("\") in KANA8 is yen ("¥")

Since HP represents character data in at least 8 bits; 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; one section contains a German example. Hewlett-Packard peripherals (generally) use the *8-bit Character Set Support Model* to handle multiple character sets).

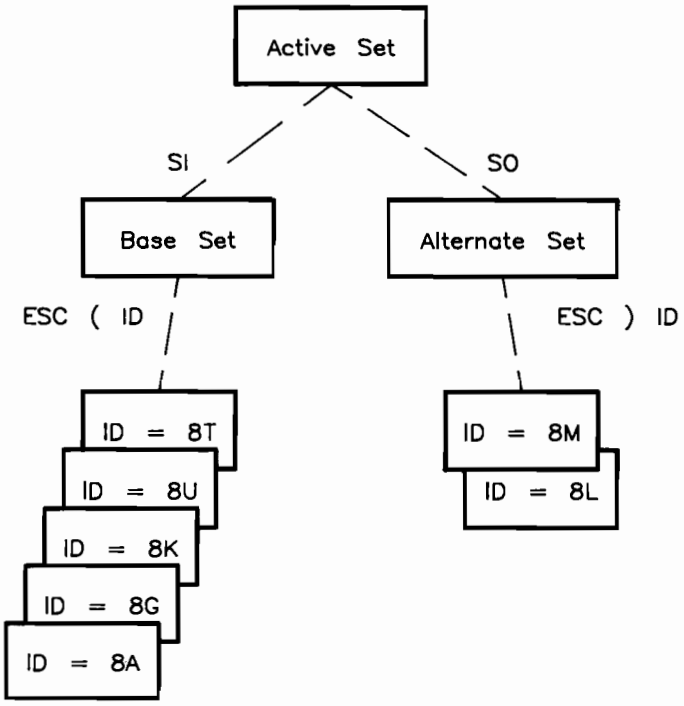


Figure C-1. 8-bit Character Set Support Model for Peripherals

Each rectangle in this figure represents a collection (or set) of 256 character code values in the form shown in the "8-bit Character Set Structure" table).

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 the table on "8-bit Character Set Name" for a list of character sets with their ID Field numbers. All sets in this model are 8-bit character sets.

Table C-2. Character Set ID Numbers

8-bit Character Set Name	ID Field	Active Set
Start up Base/Default Set	@	base
ARABIC8 Character Set	8 A	base
GREEK8 Character Set	8 G	base
KANA8 Character Set	8 K	base
LINEDRAW8 Character Set	8 L	alternate
MATH8 Set	8 M	alternate
ROMAN8 Character Set	8 U	base
TURKISH8 Character Set	8 T	base

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 at least some of its characters. 16-bit character sets are comprised of 2-byte characters.

The tool `firstof2` provides the developer with the ability to easily identify 16-bit Asian characters from ASCII and from 1-byte Asian characters. Refer to the chapters called "Developing International Applications" and "Task Reference for NLS routines" for more details on `firstof2`.

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 the next figure, 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. Peripheral devices use HP-16 for encoding. The conversion between HP-15 and HP-16 is performed by the HP-UX system. Refer to the *Native Language I/O User's Guide* for details.

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.

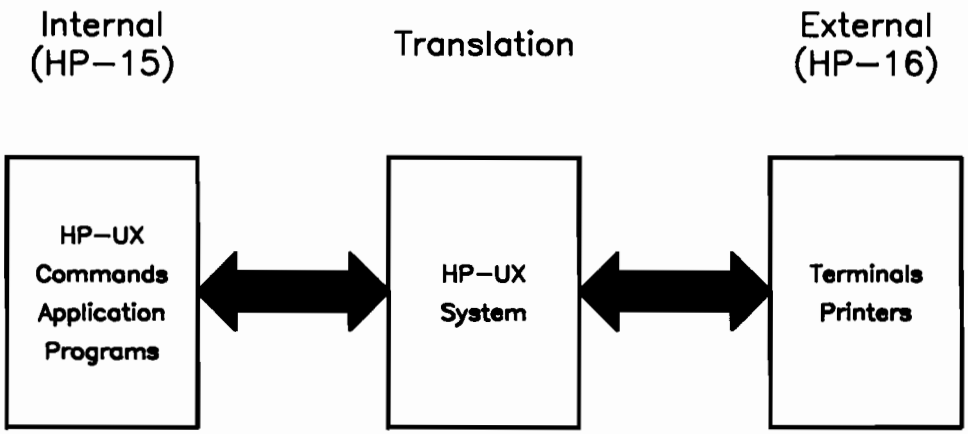


Figure C-2. Use of 16-bit character sets within HP-UX

Hewlett-Packard has developed tools to help parse data so you can determine what characters your data contains. This procedure is described in the "Developing International Applications" chapter.

Native Languages

Each supported native language is based on one of the character sets. 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 the table "Supported Native Languages and Character Sets"). 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** or `n-computer`, represents the way the computers dealt with languages before the introduction of NLS. Whenever `n-computer` is used in a native language function, the result is identical to that of the same function performed before the introduction of NLS. These NLS library calls always work correctly, even if no native languages has been configured on the system.

Table C-3. Supported Native Languages and Character Sets

Language Number	Language Name	Character Set
00	n-computer (native computer)	ASCII
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
51	arabic	ARABIC8
52	arabic-w	ARABIC8
61	greek	GREEK8
81	turkish	TURKISH8
221	japanese	JAPAN15

Character Sets

This section provides tables for the following character sets:

- ASCII
- ROMAN8
- KANA8
- ARABIC8
- GREEK8
- TURKISH8

Table C-4. ASCII Character Set

b_4	b_3	b_2	b_1		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
					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 C-5. ROMAN8 Character Set (ID=8U)

b_4	b_3	b_2	b_1		b_8 0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
					b_7 0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
					b_6 0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
					b_5 0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
					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	♭
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	*f*	ö	ß	Ÿ	±
1	1	1	1	15	SI	US	/	?	O	_	o	DEL			₤	¢	ü	Ô	ÿ	

Table C-6. KANA8 Character Set (ID=8K)

b_8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
b_7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
b_6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
b_5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

b_4	b_3	b_2	b_1		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				—	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	゜		

Table C-7. ARABIC8 Character Set (ID=8A)

b_8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
b_7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
b_6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
b_5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

b_4	b_3	b_2	b_1		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				٠	@	ذ	–	
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q			!	١	ء	ر	ف	ـ
0	0	1	0	2	STX	DC2	"	2	B	R	b	r			"	٢	آ	ز	ق	،
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s				٣	أ	س	ك	
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t				٤	ؤ	ش	ل	
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u			٪	٥	إ	ص	م	
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v				٦	ئ	ض	ن	ة
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w				٧	ا	ط	ه	ى
1	0	0	0	8	BS	CAN	(	8	H	X	h	x			)	٨	ب	ظ	و	٭
1	0	0	1	9	HT	EM	)	9	I	Y	i	y			(	٩	ة	ع	ى	
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z				:	ت	غ	ي	
1	0	1	1	11	VT	ESC	+	;	K	[	k	{			+	؛	ث		،	
1	1	0	0	12	FF	FS	,	<	L	\	l	|			،		ج			
1	1	0	1	13	CR	GS	–	=	M	]	m	}				=	ح			
1	1	1	0	14	SO	RS	.	>	N	^	n	~					خ			
1	1	1	1	15	SI	US	/	?	O	_	o	DEL			/	؟	د	–		

Table C-8. GREEK8 Character Set (ID=8G)

					0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
					0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
					0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
					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						0	ΰ	ο
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q					A	Π	α	π
0	0	1	0	2	STX	DC2	"	2	B	R	b	r					B	Ρ	β	ρ
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s					Γ	Σ	γ	σ
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t					Δ	Τ	δ	τ
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u					E	Υ	ε	υ
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v					Z	Φ	ζ	φ
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w					H		η	ς
1	0	0	0	8	BS	CAN	(	8	H	X	h	x					Θ	Χ	θ	χ
1	0	0	1	9	HT	EM	)	9	I	Y	i	y					I	Ψ	ι	ψ
1	0	1	0	10	LF	SUB	•	:	J	Z	j	z						Ω		ω
1	0	1	1	11	VT	ESC	+	;	K	[	k	{					K	ά	κ	ί
1	1	0	0	12	FF	FS	,	<	L	\	l	|				Ϊ	Λ	ή	λ	ΐ
1	1	0	1	13	CR	GS	−	=	M	]	m	}					M	ό	μ	ώ
1	1	1	0	14	SO	RS	.	>	N	^	n	~				Ϋ	N		ν	'
1	1	1	1	15	SI	US	/	?	O	_	o	DEL					Ξ		ξ	

Table C-10. TURKISH8 Character Set (ID=8T)

b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p						Å	ğ	Þ
0	0	0	0	0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q			Ç	Ý	ê	î	Ã	þ
0	0	0	0	0	0	1	0	2	STX	DC2	"	2	B	R	b	r			Ğ	ý	ô	Ø	ã	·
0	0	0	0	0	0	1	1	3	ETX	DC3	#	3	C	S	c	s			È	·		Æ	Đ	µ
0	0	0	0	0	1	0	0	4	EOT	DC4	$	4	D	T	d	t			Ê		á	å	ð	¶
0	0	0	0	0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u			Ë		é	í	Í	¾
0	0	0	0	0	1	1	0	6	ACK	SYN	&	6	F	V	f	v			Î	Ñ	ó	ø	Ì	-
0	0	0	0	0	1	1	1	7	BEL	ETB	'	7	G	W	g	w			Ï	ñ	ú	æ	Ó	¼
0	0	0	0	1	0	0	0	8	BS	CAN	(	8	H	X	h	x			´	¡	à	Ä	Ò	½
0	0	0	0	1	0	0	1	9	HT	EM	)	9	I	Y	i	y			`	¿	è	ì	Õ	a
0	0	0	0	1	0	1	0	10	LF	SUB	•	:	J	Z	j	z			^	₺	ò		õ	º
0	0	0	0	1	0	1	1	11	VT	ESC	+	;	K	[	k	{			¨	£	ù	İ	Š	ı
0	0	0	0	1	1	0	0	12	FF	FS	,	<	L	\	l	|			~	¥	ä	Ö	š	ö
0	0	0	0	1	1	0	1	13	CR	GS	-	=	M	]	m	}			Ù	§	ë	Ş	Ú	ş
0	0	0	0	1	1	1	0	14	SO	RS	.	>	N	^	n	~			Û	ƒ		Ü	Ÿ	ü
0	0	0	0	1	1	1	1	15	SI	US	/	?	O	_	o	DEL			£	c		ç	ÿ	

Peripheral Configuration D

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, ¥, §, ƒ, Á thru ±.

See the ROMAN8 character set in the appendix "Character Set Representation and Character Sets".

Katakana Character Sets

Many HP peripherals support a base 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 available for 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 D-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 D-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 D-2. 16-Bit Terminals

Peripheral Device	Character Set(s) Support	Ordering Information	Comments
HP 35714A	KANA8 Std. Japanese	NA	Character Mode Kanji Terminal
HP 45945A	KANA8 Std. Japanese	NA	Need AdvanceLink
HP 41041A	KANA8 Std. Japanese	NA	Character/Block Mode Kanji Graphic Terminal

Table D-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 D-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 D-4. 16-Bit Printers

Peripheral Device	Character Set(s) Support	Ordering Information	Comments
HP 35713A	Kanji	NA	
HP 35719A	Kanji	NA	Does not support HP-16
HP 35720A	Kanji	NA	Does not support HP-16
HP 41063A	ROMAN8, KANA8, Japanese	NA	

Table D-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.

application program — a program which performs a specific application.

application programmer — a person who writes programs for an end-user.

ARABIC8 — the Hewlett-Packard supported 8-bit character set for the Arabic language.

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, ARABIC8, or ROMAN8 character.

byte redefinition — corruption of a 2-byte character when one of its bytes is treated as a whole 1-byte character.

character — a language unit, usually represented in 7 bits (ASCII), 8 bits (KANA8, ROMAN8, GREEK8, ARABIC8, TURKISH8), or 16 bits (JAPAN15).

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 read and write 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**, **csh**, and other HP-UX commands apply when searching for a file known by an incomplete path name. It is defined by PATH in **environ**. Log in 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 catalogue 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 impact the operation of your software. It is set by default when you log in and may be changed during the course of a session.

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 internal operating system representation of 16-bit data.

HP-16 — a Hewlett-Packard encoding scheme for 16-bit character sets used for communicating 16-bit data between a peripheral and a computer. By using 16-bit data over 64,000 characters can be represented.

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.

locale a region or nation which shares language, customs, alphabets and special symbols, currency and calendering systems, etc.

localization — the adaptation of prelocalized software for use in different countries or local environments.

message catalogue — the external file containing prompts, responses to prompts, error messages, and mnemonic command names in the user's native language.

message catalogue system — a set of tools developed by Hewlett-Packard to extract print statements from C programs and place them in or retrieve them from the message catalogue.

n-computer — an invented, artifical computer language, native computer, that identifies the way a system previously performed operations NLS now performs in a language-dependent way.

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 and 16-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. See `/usr/lib/nls/config` for supported languages.

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 "/".

Index

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

U

W

Table of Contents

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	C		SP						E							
0 0 0 1 1	O								X							
0 0 1 0 2	N								T							
0 0 1 1 3	T								E							
0 1 0 0 4	R								N							
0 1 0 1 5	O				USASCII				D				EXTENDED			
0 1 1 0 6	L				GRAPHIC				E				PRINTABLE			
0 1 1 1 7					(printable)				D				CHARACTERS			
1 0 0 0 8	C				CHARACTERS								(161-254)			
1 0 0 1 9	O				(33-126)				C							
1 0 1 0 10	D								H							
1 0 1 1 11	E								A							
1 1 0 0 12	S								R							
1 1 0 1 13	(0-31)								A							
1 1 1 0 14									C							
1 1 1 1 15								127	T							255
									E							
									R							
									S							
									(128-							
									160)							

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).

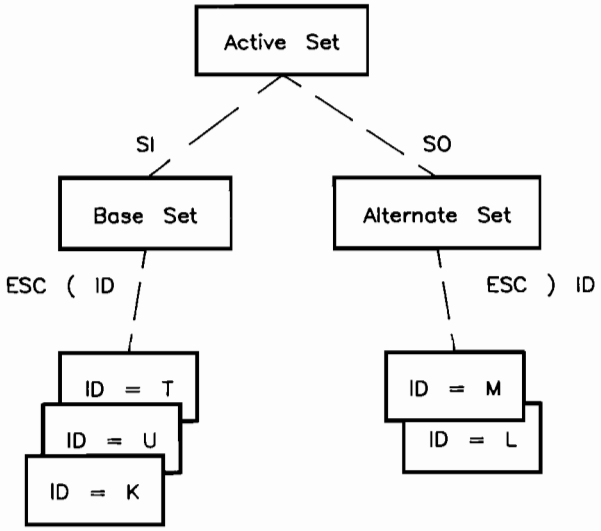


Figure 1-1. 8-bit Character Set Support Model for Peripherals

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.

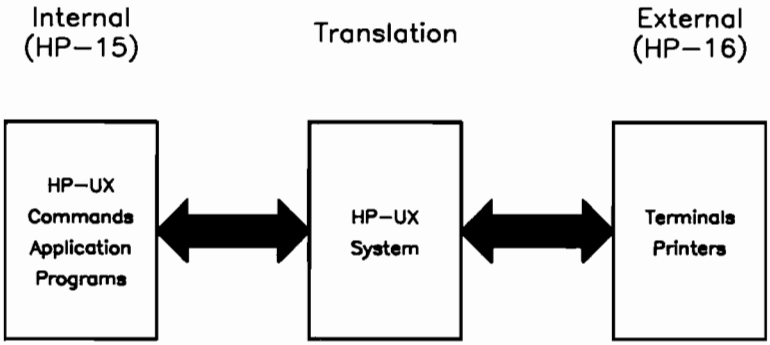
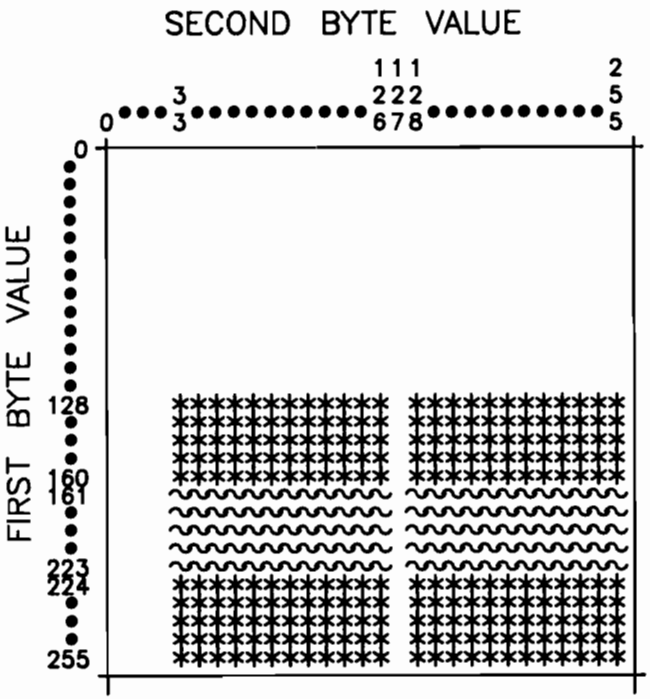


Figure 1-2. Use of 16-bit character sets within HP-UX

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

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,
```

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).

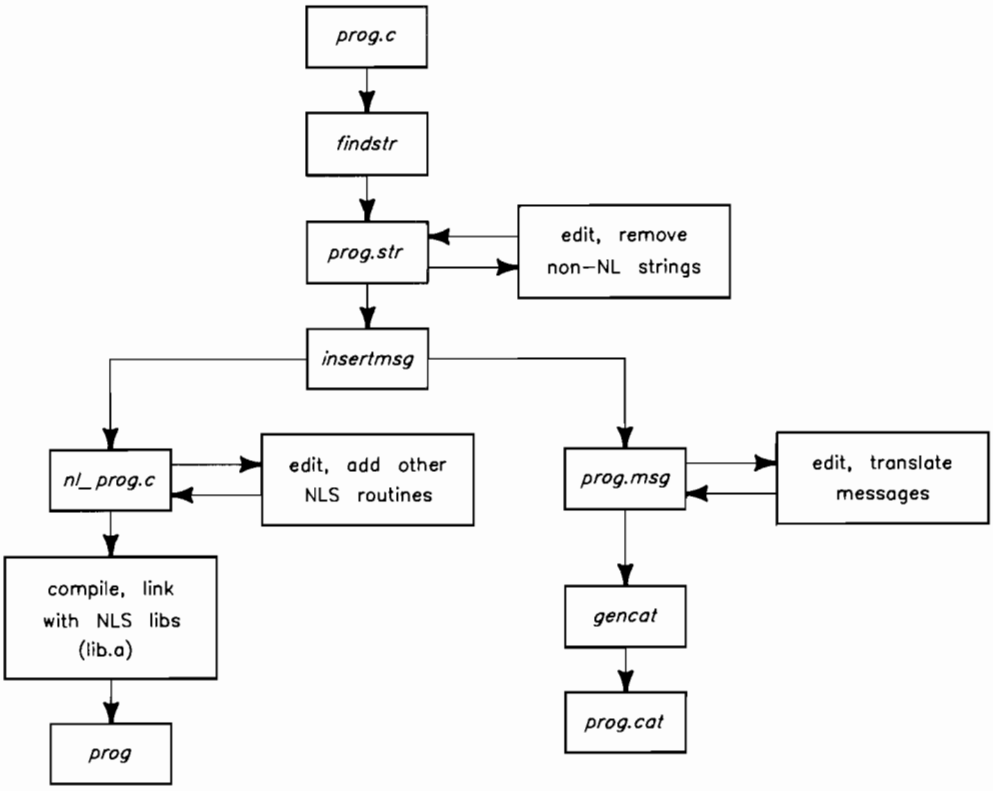


Figure 4-1: Flow of the Message Catalog

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.)

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.

Character Sets

B

This section provides the table for the following character sets:

- ASCII
- ROMAN8
- KANA8

Table B-1. ASCII Character Set

b_4	b_3	b_2	b_1		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
					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)

b_4	b_3	b_2	b_1		b_8 0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
					b_7 0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
					b_6 0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
					b_5 0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
					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	‗
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	*f*	ö	ß	Ÿ	±
1	1	1	1	15	SI	US	/	?	O	_	o	DEL			₤	¢	ü	Ô	ÿ	

Table B-3. KANA8 Character Set (ID=8H)

b_4	b_3	b_2	b_1		b_8 0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
					b_7 0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
					b_6 0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
					b_5 0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
					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				—	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 "/".

Index

a

b

c

d

e

f

g

h

i

j

k

l

m

n

p

r

s

t

u