SCSI Technical Reference

HP 9000 Series 300 Computers

HP Part Number 98265-90010

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.

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.

Februrary 1988...Edition 1

May 1988...Update. This update alters page v of Table of Contents, provides special instructions in front matter for users having HP-UX revision 6.0, and replaces Chapter 1.

Table of Contents

Chapter 9: Miscellaneous Hints

Special Instructions for HP-UX, Revision 6.0

People who have the 6.0 revision of the HP-UX operating system need to account for two situations:

1. The **/source/newfs/usr/lib/drivers** directory contains a file named **Readme** that you need to examine before you use any SCSI software. The "Readme" file:

   a. describes how to organize your work for code development and suggests directories you should make.

   b. defines terms related to the HP SCSI software.

   c. suggests that you archive the makefile. Following this, the README file discusses linking and loading operations and explains how to edit the makefile.

   d. explains how to recompile the source you received.

2. Chapter 8 in this manual describes *The Ioctl Path.* The functionality provided by this path was not provided in revision 6.0 of the HP-UX operating system. For HP-UX 6.0, you need to ignore or remove Chapter 8. If you plan to upgrade your system to revision 6.2 in the forseeable future, just ignore the material.

Once you account for these situations, read the manual as required.

Overview 1

This manual provides user and reference information for the Hewlett-Packard Company Small Computer Systems Interface (HP SCSI) which is based on the SCSI standard. (Differentiating between SCSI and HP SCSI is discussed later.)

This manual helps you complete tasks shown in the following list:

Major Task	Where to get required information
Install or add an unsupported SCSI disc	The section called "Installing or Adding an Unsupported SCSI Disk" in this chapter helps you install or add an unsupported SCSI disk for which `/etc/disktab` has no entry. (You need to be an HP-UX system administrator.)
Get a conceptual view of SCSI	"Describing SCSI" (in this chapter) defines SCSI, shows representative models, and explains how HP SCSI fits into the overall picture. (You need a conceptual grasp of the SCSI standard.)
Determine if a SCSI device might work with HP drivers	Chapter 2 contains this information. Before using a SCSI device with an HP system, you need to determine that the device meets certain requirements. (You need to understand the SCSI standard and be an HP-UX system administrator.)
Test a SCSI device to see if it works with HP SCSI drivers	Chapter 3 describes the procedures for testing a device. After you determine that a device might work (Chapter 2), you need to test it at various levels. (You need expert ability in using HP-UX. You should have a thorough grasp of SCSI.)
Modify HP SCSI drivers so a SCSI device can use them.	Chapters 4 through 9 discuss ways to modify HP SCSI drivers. (You need expert knowledge of HP-UX kernel drivers (e.g. system calls, C programming, I/O subsystems, DMA, HPIB, kernel compiling and debugging, Fujitsu MB87030 chip, ANSI SCSI standard, SCSI I/O subsystem).)

Besides this overview, the next two sections provide information about getting additional information and HP disclaimers.

Notes

Getting Other Related Information

Table 1-1. Materials Related to SCSI, Drivers, and HP-UX

Material	Where to Obtain the Material
ANSI Standard: Small Computer System Interface (SCSI) X3T9.2 Rev 17B	American National Standards Institute, 1430 Broadway, New York, N.Y. 10018
Fujitsu MB87030 User's Manual	Front Range Marketing, 3100 Arapahoe Road, Suite 404 Boulder, CO 80303, (303)-443-4780
The manuals for your SCSI device	Obtain from company making the device.
Manuals for your HP SCSI board/cable	HP Sales Representative.
Source Code for HP SCSI Drivers	`scsi.c` in `/usr/lib/drivers` in HP-UX
HP-UX Driver Development Guide	HP Sales Representative.
HP-UX Installation Manual	HP Sales Representative.
HP-UX System Administrator Manual	HP Sales Representative.

Hewlett-Packard Company Disclaimers

Because you might modify the HP-UX kernel and use nonHP implementations of SCSI, the following items note Hewlett-Packard Company disclaimers:

- HP is committed to industry standards; and for this reason, while the HP SCSI cards and software adhere to the ANSI standard and provide considerable flexibility, their use with unsupported products is at the user's risk.
- No products except the supported products sold by HP have been tested. The operation of nontested products cannot be assured.
- Some nonHP products require commands not included in the HP SCSI software. It might not be possible for nonHP products to have full functionality.
- It is possible for a user to modify the HP SCSI driver, and if the driver is modified, HP waives all responsibility for its proper operation.
- With regard to the SCSI software, HP warrants only that it will not fail to execute its programming instructions due to defects in materials and workmanship as set forth in the HP "Warranty and Installation Terms" applicable to the software.
- HP makes no other warranty for the software, expressed or implied, written or oral. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.
- HP specifically disclaims all responsibility for the operation of the SCSI software according to any specifications for use of the software with nonHP products and for results connected with the use of the software.

In short, you should carefully evaluate a decision to use an unsupported product with the HP SCSI hardware and software.

Installing or Adding and Unsupported SCSI Disk

Recall that this chapter helps you determine if your SCSI device will work with the HP SCSI drivers. In this context, this section helps you answer questions concerning the installation or addition of a SCSI disk.

Supported and Unsupported Disks

From HP's viewpoint, you can have two categories of SCSI disks:

1. **A supported disk** (e.g. the HP 7957S, HP 7958S, and HP 7959S are supported disks). A SCSI disk supported by HP should work, either during installation or as an added disk. No special configuration is required to make a supported disk work.
2. **An unsupported disk** (e.g. a disk other than the disks mentioned above; and in particular, a disk whose vendor claims it meets the SCSI standard). Before you install or add an unsupported disk, be aware that HP does not recommend using such a disk. Variation in the implementation of SCSI among different vendors can cause unpredictable results, including the loss or incorrect recording of data. To ensure proper disk operation, the use of an HP supported disk is strongly recommended. With this in mind, if you do have an unsupported disk, continue reading this section.

Why Read This Section

Two conditions could make it necessary for you to read this section:

1. **During the installation of HP-UX on an unsupported disk**, you could get a message saying that your disk does not appear in `/etc/disktab`. At this point (*Step 6* in the *HP-UX Installation Manual*), you might need to move to *Step 7* to alter the values for the following two file system parameters:
   a. `1024 byte sectors per track`: The default value is `8`
   b. `tracks per cylinder`: The default value is `7`
2. **While adding an unsupported SCSI disk to an installed HP-UX system**, you might need to create an entry in `/etc/disktab` for the disk.

The table on the facing page describes how to deal with these two conditions.

Installing or Adding Unsupported SCSI Disks

Condition	Procedure for Handling the Condition
In *Step 6* of the installation, not having an entry in **/etc/disktab** sent you to *Step 7* to alter values for two file system parameters and that step sent you here.	You need to alter the values for: **1024 byte sectors per track**: (default is **8**) **Tracks per cylinder** (default is **7**) Study your SCSI disk's reference manual, noting that: * Terminology can vary (e.g. tracks/cylinder might be read/write heads) * You might see something like: **Cylinders 680** **Tracks per cylinder 5** **Bytes per track 16,384** To get 1 Kbyte sectors per track, **16,384÷1024 = 16** So you enter **16** for this parameter. For **Tracks per cylinder**, you enter **5** directly.
On adding a disk and using *newfs* to create a file system, you discovered you had no entry for the disk in **/etc/disktab**.	In general, you need to have: * Shutdown your system, added the SCSI disk to the SCSI bus, and powered up the system again. * Used *mknod* to create device files. * Ensured that **/etc/conf/dfile** contains **scsi**. * Used *mediainit* to format the SCSI disk. Before you use *newfs* to create a file system, work through **/etc/disktab** beginning with: **DISK GEOMETRY AND PARTITION LAYOUT TABLES** to create an entry for the unsupported SCSI disk. Then, use *newfs*. If you wish, the **-t** option sets the number of tracks per cylinder. You cannot optimize the number of 1 Kbyte sectors per track unless you reinstall HP-UX (the above condition discusses these parameters).

How to Continue

When you finish accommodating your disk, you have some choices:

- During installation, continue the install process. Then, you should probably work through Part 1 of this manual.
- For an added disk, you can:
  - use the disk without testing it and hope it works.
  - work through Part 1 of this manual.

Describing SCSI

This manual assumes you want to:

1. determine if a SCSI device works with the HP SCSI driver:
2. modify the HP SCSI driver; or
3. perform both tasks.

To provide a common background for discussion, this section examines SCSI in general.

Defining SCSI

SCSI **is** (via an American National Standards Institute (ANSI) standard) an intermediate-level Input/Output bus that sits between a host adapter on the system bus and a controller for device-level interfaces. Figure 1-1 shows a general model with some examples that show the relative location of SCSI.

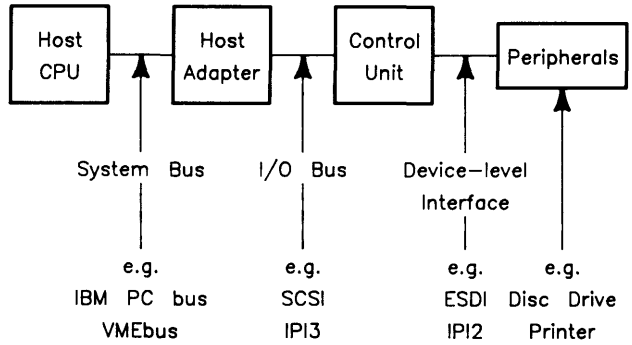


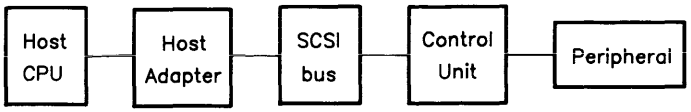
Figure 1-1. A General Model of a System Containing SCSI

While SCSI is an I/O Bus, it can be implemented in several ways and used in several configurations. The point is:

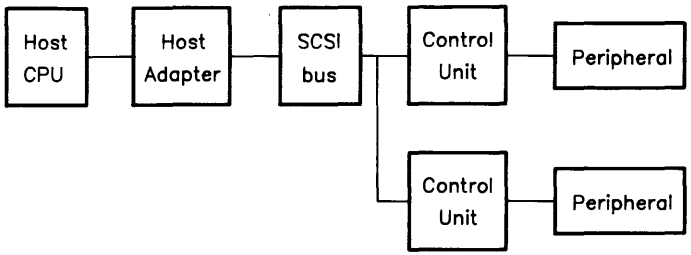
Your SCSI device might **not** work with the HP SCSI driver. You **do not** assume the driver fails to meet the SCSI standard. You might suspect your device has a different implementation of SCSI.

Looking at SCSI requires accounting for the configuration you need in relation to available SCSI implementations. Figure 1-2 on the following page shows three configurations a **complete** SCSI implementation could provide. (Complete implementations are not generally available at present; so beware in selecting your configuration.)

Single Host to Single Controller Configuration



Single Host to Multiple Controller (up to 8) Configuration



Multiple Host to Multiple Controller Configuration

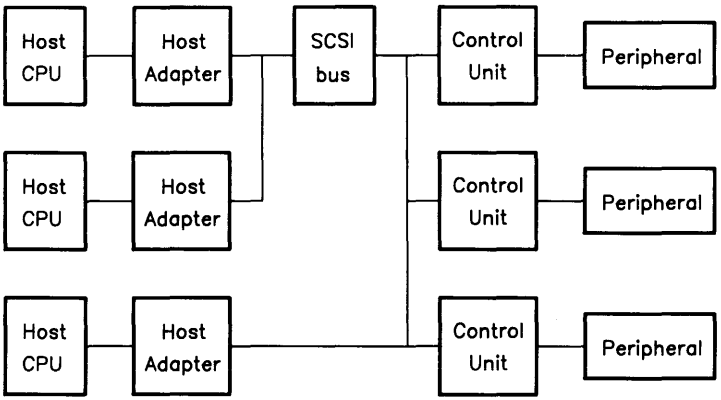


Figure 1-2. Possible SCSI Configurations

As you can see, a complete SCSI implementation allows a range of configurations that increase in complexity as you move beyond a single-host/single-controller setup. (Later, you will see that the HP SCSI allows up to a single-host/multiple-controller configuration.)

What SCSI Is

While you should read the ANSI SCSI standard to get a complete picture, the following items suggest what SCSI is or provides:

- SCSI is an all encompassing specification (standard). It addresses mechanical, electrical, and functional requirements (i.e. SCSI talks about physical connectors, voltage drops, bus timing, and so on).
- Single-ended implementations provide asyncronous data transfer over a maximum distance of 6 meters at up to 1.5 Mbytes per second.
- Differential implementations provide long-distance data transfers over a maximum distance of 25 meters and a synchronous transfer rate of up to 4 Mbytes per second.
- You get a rich command set (e.g. `INQUIRY`, `READ`, `TEST UNIT READY`).
- SCSI can provide messages (e.g. `COMMAND COMPLETE`, `DISCONNECT`, `ABORT`)
- SCSI provides 18 signal lines with parity checking (9 for transfer of data and 9 for control).
- Data transfer is block oriented.

What SCSI Is Not

The following items indicate what SCSI is not:

- It is not a system or device interface.
- It has no hierarchial architecture.
- There are no master-slave relationships.

As you gather information, keep terms such as standard, device, interface, and driver within the context of their use. People can easily use widespread misunderstanding to claim a peripheral is a SCSI device. Before you jump into anything:

- Take time to understand SCSI.
- Examine HP SCSI and its drivers.
- Study the SCSI devices you want to use.
- Determine if the devices are "plug-in" compatible.

When a device is not compatible, you can think about modifying the HP SCSI drivers.

HP SCSI Compatibility Requirements 2

This chapter provides checklists for determining if your SCSI device is sufficiently compatible with the HP SCSI drivers for you to proceed with the testing of the device. Work through the checklists to see if you are ready to test a device; and then work through Chapter 3 to test a device.

At present, the only sure way to determine if a SCSI device works with the HP SCSI drivers is to test it.

Hardware System Checklist

The items in Table 2-1 let you determine that you have appropriate hardware.
Power down your hardware according to its documentation before doing any installation.

Table 2-1. Hardware System Checklist

Component	Checklist Items
Computer System	Series 300 Model 319, 333, or 350 installed according to the manuals that came with the components (e.g. CPU, monitor, disc drive).
Interface Cards/Cables	HP SCSI card installed/tested during powerup according to *User Note, SCSI Interface* pamphlet (HP Part Number: 98265-90601) Set Parity Switch to 1. The Model 319 requires factory installation. All models have options. See your HP Sales Representative about possibilities.
Your SCSI Device	Install your SCSI device to the HP SCSI card according to the documentation for your device and the HP SCSI card.

Go on to the next checklist.

The Software System Checklist

The items in Table 2-2 let you determine that you have appropriate software. Depending on how you work, it might be necessary to reboot HP-UX and otherwise provide for system administration according to the *HP-UX System Administer* manual.

Table 2-2. Software System Checklist

Component	Checklist Items
Operating System	HP-UX revision 6.0 (or newer). Include the **PDRIVERS** fileset which provides the SCSI driver and code.
Device Files	You need block and character device files: **/dev/dsk/** must contain **0s0 and 1s0**. **/dev/rdsk/** must contain files having the same names. If necessary, use *mknod* to make required device files.
Drivers	Make sure **/etc/conf/dfile** contains **scsi**. Your SCSI device needs this driver. If **scsi** is missing, add it to **dfile** and reconfigure the kernel.

Go on to the next checklist.

The Rev C Boot ROM Checklist

The items in Table 2-3 show the features your SCSI device must have for the Rev C boot ROM to load a system from your device.

Table 2-3. Rev C Boot ROM Checklist

Boot ROM Need	Discussion
Operations done only in asyncronous mode	Ignores negotiation of mode
No mid-operation disconnects allowed	No comment
Parity error checking	Must be enabled on the HP SCSI card (switch set to 1)
Your SCSI device must be capable of direct access read operations	Recognized device types include: *Code* *Device* *Class/Description* 0 DAD Direct Access R/W Devices 4 WORM Write Once, Read Many 5 RO Read Only 7 MO Magnito-Optical R/W hex 7F Logical Unit not present
Retries	Attempt **INQUIRY** up to 4 times when boot ROM attempts to identify a device and up to 7 times for other commands
Timeouts	Wait 1 millisecond for **SELECT** to complete; Between phases or after **SELECT**, wait 15 seconds for **REQ**; The boot ROM waits 2 seconds after bus reset before attempting other bus activity.
Messages	Accepts **COMMAND COMPLETE** (code 0). Reads extended messages, but does not examine them. Generates the **ABORT** message (code 6).

Go on to the next checklist.

Required SCSI Commands Checklist

The items in Table 2-4 show the commands (with hexadecimal opcodes) your SCSI device must utilize for the Rev C boot ROM to work with the device.

Table 2-4. SCSI Commands Checklist

Command	Role or Function
`INQUIRY` hex 12	Has two purposes: 1) Determines device type (requests 2 bytes) 2) Obtains vendor id and device name (requests 36 bytes): a) vendor id starts in bytes 4 and occupies 8 bytes; b) device name starts in byte 12 and occupies 16 bytes. Do set the *lunit* and *length* fields; set the others to 0 (zero).
`READ CAPACITY` hex 25	Required for boot ROM translations from internal 256 byte sector address to device block size. Set *lunit* and *length* fields; other command fields are 0. In returned data: Block size is a power of 2 within 32 bytes to 1 Mbyte; No restrictions on the number of blocks.
`REQUEST SENSE` hex 03	Set *lunit* and *length* fields; other command fields are 0. Requests 8 bytes; can send fewer. Check made only on sense-key field. Recognize sense-key codes 0-8 and 11 (hex B); other code causes boot ROM to stop device communications.
`READ` hex 28	Set *lunit*, *block-number*, and *number_of_blocks* fields; other command fields are 0 Boot ROM canot operate reliably with both long and short format reads. Most devices require or can use long format commands rather than short format.

Continuing the Testing

When you determine that your SCSI device meets the requisites shown in Tables 2-1 through 2-4, test the device by working through Chapter 3.

Testing SCSI Devices 3

Assuming you worked through Chapter 2 and determined that your SCSI device (disc) might work, this chapter describes how to verify if a device works with the HP SCSI drivers at some minimal level. Given the nature of SCSI, do not expect a thorough and complete procedure for qualifying a drive. The procedure gives a general guide to testing and demonstrates whether HP-UX can communicate at some minimal level with the device.

Because the tests can vary, procedures are described instead of demonstrated. To some extent, you need to know how to complete each task. Other resources such as the *HP-UX Reference* manual and the *HP-UX System Administrator* manual can provide information.

The items in Table 3-1 show the overall testing process.

3-1. Testing Your SCSI Device

Procedures	Descriptions
Getting Ready	Describes the setup procedure and available diagnostic output.
Low-level Testing	Describes testing the device at low-levels (open a device, look at block sizes, simple reads/writes). Do this on a dedicated bus, if possible.
System or High-level Testing	Describes the testing of timing, bus citizenship, error recovery, and such by stressing the device in an environment that makes it perform complex file system tasks. If possible, put several peripherals known to be good on the bus with all devices active.
Integration Testing	Describes testing a device in an actual environment. Has two phases: 1) Stress testing with a known load. 2) Actual user testing.

Getting Ready for Testing

To test a device, you need to get ready and know where to get diagnostic information.

Table 3-2 describes the process. Become the root user as required.

Table 3-2. Getting Ready for Testing

Task/Information	Procedure/Description
Setting Up	1) Shutdown your HP-UX system (*shutdown -h*). 2) Attach the device (disc drive) according to its documentation and perform required self-tests. 3) Reboot HP-UX and login as the root user. 4) Use *mknod* to make block and character device files for your device (disc drive).
Getting Diagnostic Information	1) The kernel message logs contain diagnostic output from the SCSI drivers. 2) If configured, you can read **usr/adm/messages**. 3) Otherwise, use the utility named *dmesg* to access the kernel message buffer.

The diagnostic information lets you know what happens at all three levels of testing.

Continuing

On completing these tasks, do the low-level testing.

Notes:

Use this page for any notes you want to keep.

Performing Low-level Tests

Having installed your peripheral, powered up the system, and made necessary device files, you can begin low-level testing.

- Table 3-3 describes a series of tests.
- The facing page shows a read/write exerciser shell script for low-level testing.

Table 3-3. Low-level Tests

Test	Description, Information, Procedure

Run `scsi-inquiry`	1) Power cycle your disc drive.
	2) Run `scsi_inquiry` to determine if the device controller handles basic commands and tests:
	`test_unit_ready`
	`read_capacity`
	`inquiry`
	3) Read the `dmesg` buffer to see if it looks all right. This tests `request_sence` (i.e. the HP driver recognizes the extended sense data).

Format Your Disc?	If necessary, use *mediainit* to format your disc.
Access the Drive Mechanism	Use shell scripts (the facing page has an example). The idea is to try simple write/read tests. Vary the test using different block sizes. Test several devices on the bus at the same time, running the tests simultaneously. HP used the `dd` command, but other commands would work. For a more thorough write/read exerciser, write a known pattern to the entire disc, and then read the disk pattern back in, comparing the data with the pattern (use a C language program to do this).

Using a Shell Script for Low-level Testing

The following script performs several low-level tests.

```
#Shell script (A) write/read exerciser
# Parameter supplied is character device file

# Check Parameters and create a 1 MByte image
if [ $# = 1 ]
then
   echo creating image
   dd < /dev/root > /tmp/1MByte bs=64k count=16
else
   echo usage: $0 char_special_file
   exit 1
fi

# 10 Passes of test
for i in 1 2 3 4 5 6 7 8 9 10
do
   dd < /tmp/1MByte > $1  bs=64k count=16
   dd < $1 > /tmp/tmp_copy bs=16k count=64
   if cmp /tmp/tmp_copy /tmp/1MByte
   then
      rm -rf /tmp/tmp_copy
      echo pass
   else
      echo failed
      exit 1
   fi
done
rm -rf /tmp/1MByte
```

Trying Alternative Tests

Another valuable write/read exerciser is a random write/read test using the following algorithm:

1. Write an ascending pattern to the entire disc (e.g. all 0's to block 0, all 1's to block 1, etc.).
2. Using a random number generator to generate logical block numbers,
   a. seek to random locations on the disc,
   b. read in the block, and
   c. compare the data.

When you complete these tests, go on to System-level Tests.

Performing System-level Tests

File system testing is valuable for testing variations in timing, bus citizenship, and error recovery. For using a disc as an ordinary file system, the tests should stress the disc as a root disc under actual file system activity. One standard (basic) procedure has the following steps:

1. Make a file system and mount the disc on a directory.
2. Fire off several write/read exercisers.
3. Umount the disc and use *fsck* to check the integrity of the disc.

This page and the facing one show a script for doing these tests.

Using a Shell Script for System-level Testing

The following script shows system-level tests.

```
# Shell script (C) file system write/read exerciser
# User supplies two parameters:
#    - device special name (Not path name)
#      (Make sure the character file is prefixed with 'r')
#       E.g.   /dev/rscsi.4 is char special file (addr 4)
#              /dev/scsi.4  is block special file
#    - blocksize (in 1KBytes) of device
# Usage: exer dev_name blksz

DEV=$1
SIZE=$2

if [ ! $# = 2 ]
then
   echo usage
   exit 1
fi

mkfs  /dev/r$DEV $SIZE           ||  exit 1
mkdir /misc$$                    ||  exit 1
mount /dev/$DEV /misc$$     ||       exit 1
mkdir /misc$$/lib /misc$$/h ||       exit 1
```

```
for i in 1 2 3 4 5 6 7 8 9 10
do
# Some miscellaneous file system activities
cp /lib/* /misc$$/lib     &
cp /usr/include/*.h /misc$$/h &
cp /hp-ux /misc$$
cmp /hp-ux /misc$$/hp-ux
wait

for file in  /lib/*
do
   cmp $file /misc$$/$file ||
   (echo compare failed;  exit 1)
done
cd /usr/include;

for file in  *.h
do
   cmp $file /misc$$/h/$file ||
   (echo compare failed;  exit 1)
done
echo pass $i
done

wait
umount /dev/$DEV
fsck /dev/r$DEV
```

Performing Additional Tests

The testing becomes more complex, as you trust the integrity of the I/O subsystem.

- The testing harness (using a shell script) fires off a sequence of slave shell scripts that mimic the above test.
- You force a mixture of file activity (copy, remove, move, and variations of file routines).
- Using shell scripts, you should make other devices on the bus active.

Continuing

When you complete these tests, do the integration testing.

Performing Integration-level Tests

At this point, as the root user, make a bootable/rootable disc. Then, test the disc as a root file system. The long and extensive testing includes the following steps where the examples assume a device file in `/dev` named `newdics`:

1. Use *newfs* to make a file system by making an entry in `/etc/disktab` according to instructions in the file.
2. Execute `init s` to get into single-user state.
3. Execute the following list of commands to copy your existing file system from your root disc onto the new disc, making sure files systems get mounted, found, unmounted, and so on.

```
cd /
mount /dev/newdisc /misc.XXXX
find . -print | grep -v 'misc.XXXX' | cpio -pdxlmu /misc.XXXX
umount /dev/newdisc
fsck /dev/newdisc
```

Moving On to User-level Testing

Having completed these tasks, reboot the system, hitting the spacebar during the process. Then, you can select the new disc as the root. For user-level testing, do the following things:

- Set up stress tests that can be fired off regularly.
- Make sure the disk can remain functional for long periods; for example, make some stress tests takes at least 24 hours to complete.
- During the above work, track the diagnostic logs from the operating system.

Measuring Performance

During integration testing, you should have an estimate of the expected performance of your peripheral. For example, does a disc drive perform write operations as-well-as expected. File system tests that compare known (older) drives with the drive being tested can help you evaluate its performance.

Getting an expected performance **is** a critical test. In this regard, for example, the following line shows how to use `dd` to time sequential transfers.

```
# time dd < /dev/r<new_device> > /dev/null bs=64k count=1000
```

The following items suggest some additional measures of performance:

- Time some simple file copy routines.
- Try using a random write/read exerciser that calculates the time.

If these things go well, you should have a functional drive.

Deciding How to Continue

At this point, you probably have one of three situations:

- Your peripheral works fine. You like the way it functions and have no intention of doing additional work.

  Set this manual aside and enjoy using the device.

- Your peripheral works to some degree, but you want to make some modifications.

  You still have work to do, and reading Part 2 can be helpful.

- Your SCSI peripheral does not work adequately.

  You need to decide whether to write a driver, not use the peripheral, or take some other action. If you decide to write a driver for the device, reading Part 2 and looking at the source code for the HP SCSI drivers can help.

Introduction to the SCSI Drivers 4

This chapter begins Part 2 which describes the HP SCSI drivers and how to modify them so they work with your SCSI device. The chapters in Part 2 number 4 through 8. If you modify a driver, you need to work through all of them.

This chapter provides an overall picture. It shows the relationships among such things as hardware and software, HP SCSI and the Fujitsu chip, and the various drivers. Table 4-1 describes the topics.

Table 4-1. Topics in Chapter 4

Topic	Description
Assumptions for Using Part 2	discusses what you need to do beyond working through Part 2.
Identifying Your Situation	discusses legitimate types of driver modifications.
SCSI and HP-UX I/O	provides information about hardware/software layers and drivers.
Overview of Drivers	provides information about the interface driver, HP SCSI implementation, service interrupt routine, device driver, disc transactions, and HP Finite State Machine.
SCSI Requirements for Drivers	mentions requirements for SCSI drivers and a feature (bug) in the Fijutsu chip.

Assumptions for Using Part 2

Using this part assumes you:

1. read Part 1;
2. studied the ANSI SCSI standard;
3. read the manual for the Fujitsu MB87030 chip;
4. read the manual for your SCSI device; and
5. have an expert knowledge of the C programming language and the HP-UX kernel.

Besides assuming the requisite knowledge and skill just mentioned, this part assumes you installed and tested your device according to procedures described in Part 1 and determined that you need to modify the HP SCSI driver.

What If I Write My Own Driver

If you intend to write your own driver, be aware that this manual does not sufficiently address the information required to make major changes to the HP SCSI driver or write a driver from scratch.

Besides this manual and the manuals related to SCSI, the *HP-UX Driver Development Guide* has additional information. Be aware, however, that you are on your own if you decide to write a driver (as opposed to modifying existing HP drivers).

Identifying Your Situation

The organization and content of Part 2 assumes you have situations like those that follow:

- Your peripheral works with the HP SCSI driver, but you want information about the HP SCSI I/O Subsystem.
- You might be a hardware engineer designing a new peripheral, and in this context, you need to understand the software before you integrate software issues into a hardware design.
- Your peripheral does not work with the HP SCSI drivers, but a slight modification will fix the problem.
- You want to make significant modifications (e.g. provide support for a printer or streaming tape drive). To be realistict, you probably should not attempt this unless you have extensive knowledge about writing HP-UX kernel drivers.
- Your peripheral works with the HP SCSI driver, but you need modifications such as;
    - Enhancing support for your peripheral (adding diagnostics, etc.).
    - Adding a new command not currently implemented.
    - Modifying the *ioctl* call to execute a specified routine.
    - Changing certain parameters (e.g. timeout characteristics).
    - Modifying the Finite State Machine to handle message bytes differently.
    - Enforcing some type of protection not anticipated by the current driver.
    - Adding additional fields to SCSI-only data structures.

SCSI and HP-UX I/O

If you decide to modify the HP SCSI driver so your SCSI device will work, you need to know how SCSI fits into the HP-UX Input/Output model. Figure 4-1 shows the model and the location in the model of the components of the HP implementation of SCSI. Looking at the model, notice the following things:

- Each hardware layer has a corresponding software driver.
- HP has one driver per class of devices (e.g. if every device is a disc, there is only one driver.)
- While you may need more than one device driver, the interface driver can work for all classes of devices.

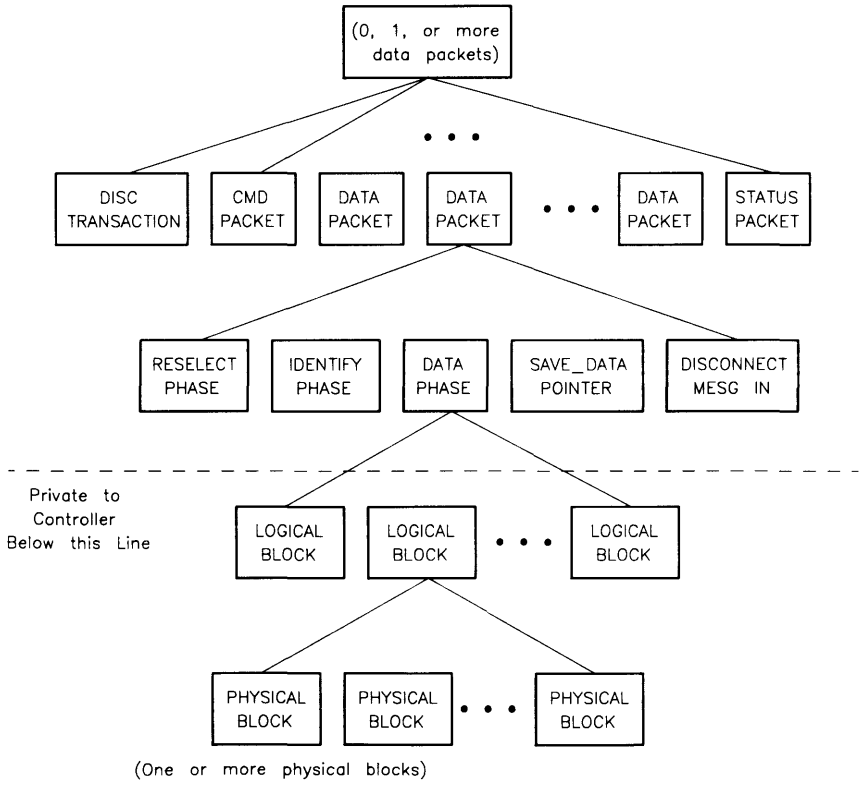


Figure 4-1. The HP Input/Output Model

The model shows that SCSI encompasses the hardware and software layers.

Hardware Layer

The interface is the HP 98265A SCSI card (or equivalent option), which connects a peripheral and the host adapter (this was shown in Part 1 in the general model of SCSI).

The Software Layer

This layer has two components.

- The `scsi_if.c` interface driver handles the activities of the interface card.
  - Activities include: knowing the physical lines on the bus; processing interrupts; selecting devices; and setting up DMA and other types of data transfers.
  - The interface driver interacts directly with the SCSI interface card (i.e. the Fujitsu controller chip) and has detailed knowledge of the characteristics of the Fujitsu controller.
  - The driver knows nothing about commands or the structure of 6-byte or 10-byte commands. It has no device-specific knowledge (i.e. it does not even know if a device is a disc, tape, or something else).
- The higher-level `scsi.c` device driver makes various requests of the interface driver: selecting a device, determining the current bus phase reported by SCSI, and transferring a data buffer, which might be a command or real data.
- The device driver knows about disc-dependent things (e.g. SCSI commands such as `test_unit_ready` and `read_capacity`). It also knows about HP-UX system calls such as *read* and *write*.
- The driver knows about the physical characteristics and geometry of the disc (it assumes a disc). When the driver issues a request to the disc (e.g. a command packet), it asks the interface driver to transfer a specified data buffer to a certain disc and wake the device driver up when done.

  The "heart" of the device driver is a complex mechanism called a "Finite State Machine". At present, no complete description of the FSM is available, but you can see it in use by studying the parts of `scsi.c` related to device drivers.

<The following page has related information.>

The device-driver level within the software layer could have several device drivers. At present, HP provides one device driver called `scsi.c`. which handles direct access devices. To envision requirements for device drivers assuming DEV 0 and DEV 1 in Figure 4-1 are discs, DEV 2 is a printer, and DEV 3 is a 9-track tape; you could use `scsi.c` to access DEV 0 and DEV 1. You would need a printer driver for DEV 2 and a tape driver for DEV 3. While you would need three device drivers, all the device drivers could use the interface driver (`scsi_if.c`). Saying this in a different vein, since the SCSI specification encompasses interface and device driver levels, there could be some confusion about the use of drivers. HP-UX has a driver for the interface and a disc device driver that supports a range of disc-like peripherals (e.g. ordinary discs, WORMs, ROMs, MOs). The disc device driver cannot support printers, plotters, or 9-track tapes but the interface driver can.

(This page is intentionally blank.)

An Overview of the SCSI Drivers

This section provides an overview of the SCSI drivers. By working through the section, you can get an overall picture of how the interface driver, service interrupt routine, disc device driver, disc transactions, and HP Finite State Machine work.

The Interface Driver

The SCSI standard provides various options for implementation. Always remember that the interface driver is strongly dependent on the logical characteristics of the Fujitsu MB87030 SCSI controller chip. To understand the interface driver and the characteristics of the chip, read Chapter 4 of the *Fujitsu MB87030 User's Manual.* Also relative to the interface driver, Table 4-2 on the facing page shows features of the HP SCSI design and implementation (i.e. the table shows what to expect from the interface driver).

Table 4-2. HP SCSI Implementation of Interface Driver

Feature	Description of Implementation
Arbitration	A system option in SCSI implemented by HP, allowing single host/ multiple target environment. You can overlap multiple activities with arbitration.
Single initiator (one host per bus with multiple targets)	The interface driver assumes it is the only initiator on the bus.
Target	The interface driver **cannot** respond as a target.
Parity	HP implements it; every device on the bus checks parity.
Data Transfers	**Asynchronous** (synchronous mode not available). When DMA is available, transfers use DMA based on the availability of the DMA channel and the restrictions described below.
DMA	HP 9000 Models 330/350 SCSI card supports 16 and 32 bit DMA. HP 9000 Model 319 SCSI card uses 16 bit DMA. The SCSI drivers support both 16 bit and 32 bit DMA. The actual DMA path is determined by: availability of DMA the requested size the address of the buffer
DMA Chip Requirements	Buffer is long-word aligned for 32 bit DMA operations. Count is a multiple of four (latter always met since HP driver is for discs). Buffer is word aligned for 16 bit DMA. Buffer is byte aligned when using a processor-controlled fast-handshake routine (HP hardware does not allow byte-wide DMA).
Commands	Transferred by the fast-handshake routine.
Message Bytes	Use manual transfer option provided by Fujitsu chip.
Status Bytes	Use manual transfer option provided by Fujitsu chip.

The Device Driver for Discs

The HP SCSI device driver for discs implements the full "channel" concept of SCSI (i.e. the target can disconnect and reconnect at any point during a disc transaction, allowing multiple activities on the bus to overlap).

The target devices drive the bus phase of SCSI. Messages are used to control the environment of the physical path. One or more messages of one byte each are sent between the host and the target to control a transaction.

How the Driver Works

In a typical read or write transaction:

1. The host selects targets and sends a command.
2. The target disconnects after the command has been issued to allow the device controller to decode the command and, if necessary, to seek the desired track.
3. After the target is ready, a device waits for the bus to become free, arbitrates for the bus, and if successful, reselects the host.
4. After identifying itself, the target then resumes its operation. Disconnect and reconnect may also occur several times during the disc transfer, if a large time delay is anticipated by the target (such as a seek to another cylinder, a seek to a spared cylinder, etc.). At this point the target changes bus phase from `DATA_TRANSFER` to `MESSAGE_IN` and receives a message to disconnect. This methodology provides the potential for a large bus bandwidth.

Table 4-3 on the facing page provides additional information about the disc device driver.

Table 4-3. HP Disc Device Driver Operation

Item	Description or Information
Disc Driver Entry Points	`scsi_open`, `scsi_strategy`, and `scsi_control`.
File System Access	Driver via `scsi_strategy`. `Scsi_ioctl` and `scsi_open` both use `scsi_control`. During `scsi_open` call, driver determines size parameters and who is "out there".
Driver Requests	Usually a request to transfer *n* blocks (read or write). Request takes a standard form: `read, strtng logical blck on disc, no. blcks trnsfrd` `write, strtng logical blck on disc, no. blcks trnsfrd` All requests such as size and offset in terms of blocks.
Basic Transaction	Consists of **Command**, optional **Data_In** or **Data_out**, and **Status**. Message bytes control the environment (See diagram).

Getting More Information

The *HP-UX Driver Development Guide* has more information about device drivers.

Disc Transactions

Figure 4-2 illustrates the relationship between transactions, bus phases, logical blocks, and physical blocks. The figure refers to the SCSI bus phases between a Selection or Reselection and the next Bus Free as a "Packet". This figure also shows disconnects which may or may not be present depending on the current disconnect mode of the target (i.e. allowed by initiator in current transaction, etc.)

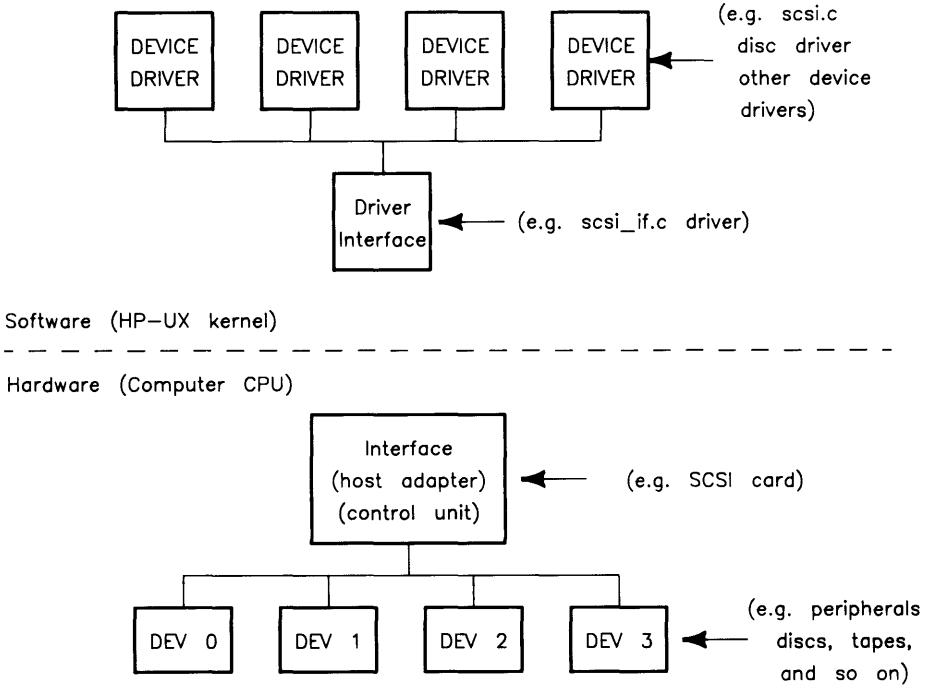


Figure 4-2. HP SCSI Disc Device Driver Transactions

The Service Interrupt Routine

The interface interrupt service routine (ISR), named `scsi_do_isr`, provides interrupts as follows:

Command Complete indicates completion of a data transfer using the Hardware Transfer mode on the Fujitsu SPC or the completion of a `SELECT` comand. Hardware transfers made not using DMA (such as commands) or hardware transfers using the DMA mode both complete via this interrupt. (Manual transfers do not go through the ISR.) The `SELECT` command to the Fujitsu chip (which arbitrates and selects the device) also completes via this interrupt.

Service Required occurs when the Fujitsu chip notices that the target has requested another phase on SCSI. This typically happens during the data transfer: the target will change bus phase to `MESSAGE_IN` to tell the host it wishes it to disconnect. It typically sends two `MESSAGE-IN` bytes: `SAVE_DATA_POINTERS` and `DISCONNECT`.

Disconnect is not needed and not used. It probably should be used in a multi-host environment. In any case, it is unused and you cannot shut it off.

Reselected After a target disconnects from the host (such as for a seek activity) the target will at a later point reselect the host to indicate that it is ready to transfer data.

Timeout indicates that the selected bus device has not responded within a specified time. (The device is probably not present or not powered up.)

Error The Fujitsu chip has detected a hardware failure (e.g. a parity problem).

The HP Finite State Machine (FSM)

To handle the complexity of disc transactions, HP uses the concept of a Finite State Machine (FSM). The FSM is a procedure that is reentered several times during a disc transaction. Use the `ioctl` kernel system call to determine extensive information about the drive and to perform drive specific functions. The call is also the hook into the driver to initialize the disc.

How the FSM Works

The following items provide insights into how the FSM works:

- Local variables are not saved.
- The FSM is stateless (i.e. no assumptions are made from one state to the next state).
- Two methods are used to determine the current state:
  - By setting the state from the previous state.
  - By the phase requested by SCSI (the target controls the bus phases).
- The FSM is driven by the SCSI bus phases (hardware) or by software. Because of the potential for hardware failure, use `START_TIME` and `END_TIME` to trigger timers.

A General Framework for the FSM

The items in Table 4-4 show the general framework for the FSM:

Table 4-4. A Framework for the Finite State Machine (FSM)

Item	Description or Comment
queue up	causes waiting for the select code. The bus might be busy. Some other device might be in the midst of an unrelated transaction. Locks the device.
select device	No comment.
message_in phase	the host sends an identity
issue command	No comment.
disconnect	No comment.
reselected	No comment.
message_in phase	target sends an identity
data_transfer phase	No comment.
message_in	target disconnects
reselect	No comment.
status	No comment.
message_in	command complete (bus free)
queuedone	free up the device

Getting More Information

The Finite State Machine was developed at HP to provide a way to handle disc transactions. Beyond this explanation, you can get additional information by studying the source code for the SCSI drivers.

SCSI Requirements for Drivers

The interface and device drivers for the HP interface impose requirements for a SCSI device. Some requirements are features implicitly made by the drivers. Table 4-5 lists the major required features. The driver still might not work with all software distributed by HP because the list is not inclusive. The intent here is to show the things you consider in conjunction with looking at complete SCSI standard.

Table 4-5. Required SCSI Features for Drivers

Feature	Description/Requirements
Supported Discs	`scsi.c` supports only winchester discs (HP 98575, 85, 95).
Driver Commands	**`extended_read`** and **`extended_write`** **`inquiry`** (a minimum of 36 bytes returned) **`extended_sense`** **`read_capacity`** (return capacity and length of block in bytes) **`test_unit_ready`** **`format_unit`** **`mode_sense`**
Messages	The host can issue: **`identify`** (with bit 6 set) **`no-op`** **`abort`** (on being sent, target goes to bus free and clears the target's internal tables)
Timeouts	Do not allow a device timeout and go to bus free!
Parity	Device should detect **`PARITY`** and respond.

An Important Fijutsu Chip Feature

Besides the features shown in Table 4-4, the Fujitsu chip has a feature (i.e. bug) called the **ATN glitch**. This section merely points out the glitch.

To see this, assume two or more devices on the bus and you allow disconnection/reconnection.

1. After accessing one device, the device disconnects.
2. Later, the host attempts to select the other device on the bus with **ATN**.
    a. If the host and the first device (reselecting the host) both arbitrate for the bus, and the host loses the arbitration, then the Fujitsu chip leaves the **ATN** line asserted. **This is an illegal SCSI bus phase**. Many targets could see this condition and drive the first bus phase (after reselection) to `MESSAGE_OUT`.
    b. If the host immediately sees `MESSAGE_OUT` it responds with a `NO-OP` and then expects the customary `MESSAGE_IN` phase.

The SCSI Drivers 5

This brief chapter describes some functional characteristics of the SCSI interface driver and DMA.

- The Interface Driver section describes the characteristics of the Fujitsu chip and HP DMA.
- The DMA section describes how to accommodate DMA.

Dealing With Other Drivers

The SCSI code has drivers other than the interface driver. In particular, it has a disc device driver that is described elsewhere. Beyond these drivers, you might want to modify a driver so it works with your SCSI device (e.g. a CD-ROM). To some extent, you need to already know how to make the modifications. The trick is to study the existing code and determine which changes to make.

Getting Additional Information

The information in this chapter provides an overview for examining Chapters 6 and 7. The *HP-UX Driver Development Guide* has additional information about drivers and DMA. In particular, it has information about physical addresses and their ranges.

The Interface Driver

The interface driver interacts with the Fujitsu MB87030 chip and the controller for your SCSI device. In this regard, the following sections relate to dealing with the chip and HP DMA. While you read, remember that the SPC host adapter is the only initiator on the bus (HP's design decisions reflect this assumption).

Hardware Characteristics (Fujitsu dependent characteristics)

Assuming you read the *Fujitsu MB87030 User's Manual*, here are essential ideas:

REGISTER	EXPLANATION
SCTL	Functions allow parity, allow arbitration (always set), reset chip, and pull RST on SCSI.
SCMD	Commands to Fujitsu SPC such as **SELECT**, set **ATN** on SCSI, and **TRANSFER**. Bit 2 indicates whether transfer is via DMA mode.
PCTL	Control of bus phase on SCSI.
PMOD	For synchronous transfer.
TEMP	Multi-purpose register for reading the data lines on SCSI. Used by the CPU-controlled fast-handshake routine, and for reading the device ID during reselection.
PSNS	Gives status of control lines SCSI.
INTS	Bitwise indication of the interrupting conditions.
SSTL	Indicates the SPC internal status. It indicates whether the Fujitsu chip is connected with SCSI, actively transferring data, executing the reselection phase, etc.
TCL, TCM, TCH	Transfer count registers.
SERR	Provides details of an error detected in SPC. An SPC hardware error interrupt occurs if an error is indicated at any of bits 3 to 0.

Getting Interrupts

Issuing **SELECT** or **TRANSFER** to the **SCMD** Register results in an interrupt. A device reselecting the Fujitsu SPC also causes an interrupt. A device changing bus phase during a data transfer operation (e.g. to disconnect) causes a service required interrupt.

DMA (16/32 bit)

You will need to accommodate DMA. This section provides minimal information about the HP DMA strategy. To get much more information about DMA and physical addresses, see the *HP-UX Driver Development Guide.*

While DMA is based on physical addresses, the HP drivers work with logical address. Since physical pages usually have little to do with logical pages, the data buffer can span several physical pages (some or all of which may not be contiguous). This leads to the following:

- A physical page is always 4 Kbytes.
- A data buffer can possibly span 17 physical pages.
- A maximum transfer size is 64 Kbytes.
- Since a buffer might not be page aligned, up to 17 chain elements are possible.

The HP strategy passes the logical data buffer address to `dma_build_chain` which creates a linked chain of entries. The DMA chip has two channels which handle the chaining. After creating the chain, fire off the data transaction and expect the DMA routines to handle the chaining (i.e. on physical page boundaries, you get a DMA `COMPLETE` interrupt at level 7 and have the DMA software pick up the next element in the chain array and fire off the request in the chain without involving the interface driver).

DMA has two requirements:

Alignment	long word transfers (32 bit) must start on long word boundaries; short word transfers must start on short word boundaries,
Count	long word transfers (32 bit) must be an integral multiple of long word; short word transfers must have an even count.

All other transfers (byte aligned, or odd count) must be processor fast-handshaked over the bus.

The SCSI Interface Driver 6

This chapter provides a walkthrough of the routines, code, and information related to the interface driver. Table 6-1 lists and describes the topics.

Table 6-1. Interface Driver Topics

Topic	Description
Initialization/Boot-up Routines	The section describes these routines: **scsi_init** (performs certain checks) **scsi_make_entry** (relates to console and card **scsi_link** (places SCSI in linked list) **scsi_msus_for_boot** (relates to msus and devices) **scsi_saved_msus_for_boot** (see above)
Handling Interrupts	The section describes these routines: **scsi_isr** (relates to Fujitsu chip) **scsi_do_isr** (handles INTS register interrupts) **scsi_call_isr** (handles ISR related events)
Selection Process	The section describes these routines: **scsi_select** (lets SCSI transactions commence) **scsi_do_isr** (**cmd_complete** and **srv_reg**)
Data Transfer Circuit	The section describes these routines: **scsi_transfer** (effects transfers) **scsi_dmaisr** (relates to DMA) **scsi_program_xfr** (fast-handshake routine) **scsi_part_prog_xfr** (relates to partial sectors) **scsi_man_xfr** (transfers one data byte)
Message/Status Circuitry	The section describes these routines: **scsi_mesg_out** (relate to messages and status) **scsi_mesg_in** (see above) **scsi_status** (see above)
SCSI Error Handling	The section describes these routines: **scsi_abort** (gets to bus free)

Initialization/Boot-up Routines

This sections describes routines used during the initialization and boot-up of SCSI.

scsi_init

Major functions:

- initialization of card
- initialization of data critical data structures.

This routine is called just once, upon boot up, by the operating system. It also determines whether the interface uses 16/32 bit DMA or just 32 bit DMA. It checks whether the parity option has been selected.

scsi_make_entry

This routine is called just once, upon boot up, by the operating system. It prints the initial information to the console describing the interface. It informs the system that a card is indeed out there.

scsi_link

Places scsi in the linked list of routines to be called when an interrupt at that level occurs.

scsi_msus_for_boot & scsi_saved_msus_for_boot

Both of these are called by `reboot` (msus = "Mass Storage Unit Specifier"). It is the integer passed to the boot ROM to inform the boot ROM which device (if any) to reboot.

Handling Interrupts

This sections describes routines for handling interrupts.

scsi_isr

Interrupts generated by the Fujitsu chip are handled by the low-level interrupt service routine handler for the appropriate interrupt level. The routine eventually calls `scsi_isr`, which determines the corresponding select code structure and calls `scsi_do_isr`.

scsi_do_isr

This routine handles the interrupt(s) indicated by the **INTS** Register of the Fujitsu chip. (See the *Fujitsu MB87030 User's Manual.*) After reading and storing the register, it is reset.

Bit	Interrupt	Explanation
0	Reset Condition	An RST on SCSI has occurred. This is unexpected. Possible problem is a blown fuse.
1	SPC Hardware Error	Parity error is suspected.
2	Timeout	The `SELECT` command to the Fujitsu chip timed out. No device.
3	Service required	The target has changed bus phase. `cmd_complete` handles this interrupt.
4	Command Complete	The command to the Fujitsu chip has completed. Typically it is completion of `SELECT` or data transfer request. Determine the next state, clean up the DMA transaction (if necessary), compute the resid (if necessary), and call the SCSI FSM. Dequeue other activities afterward.
5	Disconnected	Interrupt discarded.
6	Reselected	A target has reselected the SPC. If the Fujitsu chip has glitched ATN, toss the next byte. Call `scsi_call_isr` (see below) to queue up the process waiting for the target to respond.
7	Selected	HP does not support the host being selected as a target.

<Interrupt handler routines continue on the next page.>

scsi_call_isr

To see this procedure, consider a device issuing a `DISCONNECT` message to a peripheral. The message implies:

1. the target drives the bus to bus free; and
2. at a future point, the ISR reselects the host and continues with the transaction.

Looking at the FSM in `scsi.c` that handles this message, and assuming you received the `DISCONNECT` from a disc:

1. start a timeout in case the target never reselects you (e.g. someone power cycles the drive at this instant or the drive dies during a seek activity [more on this later]),
2. free up the bus (select code) for other processes, and
3. set up a queue of processes waiting for a reselect.

An Illustrative Routine

While not all code is used, the following routine shows this:

```
>   HPIB_ppoll_drop_sc(bp, proc, sense)
>   register struct buf *bp;
>   int (*proc)();
>   int sense;
>   {
>   register struct isc_table_type *sc;
>   register unsigned char mask;
>   int s;
>   /* dil needs this to be ok
>   if (bp->b_ba > 7) {
>      panic("bad ppoll bus address");
>      } */
>   sc = bp->b_sc;
>   bp->b_action = proc;
              >
>   bp->av_forw = NULL;
>   s = spl6();
>   if (sc->ppoll_f == NULL) {
>      sc->ppoll_f = bp;
>      sc->ppoll_l = NULL;
>      }
       >   else
>           sc->ppoll_l->av_forw = bp;
```

<The code continues on the next page.>

```
>   bp->av_back = sc->ppoll_l;
>   sc->ppoll_l = bp;
>   if (bp->b_flags & B_DIL) /* handle dil ppoll right */
>         mask = bp->b_ba;
>   else
>         mask = 0x80 >> bp->b_ba;
>   (*sc->iosw->iod_pplset)(sc, mask, sense, 1);
>   unprotected_drop_selcode(bp);
>   splx(s);
>   }
```

Comments

Looking at the routine, you want to place the buf structure, `bp`, on a doubly linked list in the select code structure, `sc`. The pointers:

```
sc->ppoll_f
sc->ppoll_l
```

mark the head and tail of the list. (Ignore references to DIL. `Iod_pplset` is unused by HP, and is set to `no-op` in `scsi.c`). Then, drop the select code and exit. HP protects the reentrant code with the typical spl pair.

Imagine now that you have received a message from the device to disconnect and the bus has been possibly used by other processes during the interim.

1. The device is ready, it arbitrates for the bus, and reselects you.
2. The Fujitsu chip issues a reselect interrupt, reads the mask from the **TMP** Register, and calls `scsi_call_isr` to find out which process is expecting that interrupt.
3. Since only one process can communicate with a controller at a time, you compare via bus addresses. `scsi_call_isr` sets the state to *reselect* and calls the `b_action` routine (typically the driver's FSM) and returns.

Looking at this back in `scsi_do_isr`, you have an interesting problem. Because of the asynchronous nature of the HP operating system (O/S), you can not be guaranteed that the select code is free! When a process obtains a select code and attempts to select another device, the Fujitsu chip may be asynchronously handling the reselect. HP handles this in `scsi_do_isr` by dropping the select code for the other process. Other processes may try to get the select code but will fail because the bus is busy.

The Selection Process

This section discusses the routines used in the selection process.

scsi_select

All SCSI transactions commence in the following way.

1. The Fujitsu chip combines arbitration with selection. (See `SELECT` in the *Fujitsu MB87030 User's Manual.*)
2. On the O/S, a process owns a select code and has locked the *iobuf* associated with the device. The HP strategy allows only one process to communicate with a device at a time. This does NOT preclude other activities from happening on the bus. For example, if HP locks device 0, other processes can be using the bus to communicate with other drives (besides 0) on the bus.

Looking at the FSM in `scsi.c`, state *initial* is used to get the select code. HP enters state *1* when the select is obtained. HP then calls `scsi_select`; and if the bus is not free, HP resets the state to *initial* to retry at a later point and exit. Otherwise, HP issues the `SELECT` command to the bus.

HP marks a flag (indicating issuing the `SELECT` command) and issues the command to the Fujitsu chip. Two possibilities can occur:

1. On being able to select the device, HP gets a **Command Complete** interrupt.
2. Otherwise, HP will timeout.

In the case of the **Command Complete** interrupt, HP determines the phase requested by the target, sets the state accordingly, and proceeds.

In a timeout HP gets a **Timeout** interrupt from the Fujitsu chip. HP sets the state to *select_nodev*, drops the select code, and calls the driver's `b_action` routine.

scsi_do_isr (cmd_complete and srv_reg)

The same code handles both interrupts. If a service required interrupt occurs, reset the data transfer circuit of the Fujitsu SPC.

If the previous phase was `Data_transfer`, there are two tasks: reset the hardware and compute the residual. If the transfer method was a fast handshake, compute the residual data count.

If the transfer was done via DMA, call `dmaend_isr` to reset the DMA hardware and call `scsi_dmaisr` to both reset the Fujitsu hardware and compute the residual.

The residual might not go to zero; so update `b_xadder` and `b_upcount` with the new values. (To get another view of this: the host transfers a portion of the data buffer, and then the target disconnects. The pointers must be modified when the transfer is restarted later.)

If the previous state was `selection`, reset the `ATN` line and indicate that the device is connected.

On being connected, call the driver's FSM. On returning from the FSM, deqeue the activity.

The Data Transfer Circuit

This section describes the routines in the data_transfer circuit.

scsi_transfer

First, determine the type of transfer. If a request for `max_speed` is indicated, negotiate for DMA (`try_dma`). The select code structure holds the buffer address, count, and flag (read / write). It then calls the appropriate transfer routine to kick off the action. HP always indicates `fhs_tfr` for command bytes and usually calls `dma_tfr` for data transfer.

In the case of a DMA transfer, HP:

1. calls `dma_build_chain` to set up the DMA chains;
2. sets up the Fujitsu chip for the transfer; and
3. kicks off the transfer with a call to `dma_start`. (Remember, the DMA routines will handle the chaining (at level 7)).

The line:

```
cp->scsi_scmd = TRANSFER
```

actually starts the Fujitsu transfer, while `dma_starts` starts the DMA chip.

scsi_dmaisr

This routine:

1. drops the DMA channel;
2. resets the SCSI cards buffers (a write to register `scsi_hconf` achieves this); and
3. computes the residual count via the Fujitsu transfer count registers.

scsi_program_xfr

This is the processor controlled "fast-handshake" routine that sets up the transfer count registers, sets up the `PCTL` Register, fires off the command, and starts transferring. The one caveat is that the target may change bus phase, so HP has to check the interrupt register.

scsi_part_prog_xfr

The HP operating system has certain paths that require a write or read from the disc consisting of a "partial sector". SCSI does not allow this per se. The workaround is:

1. if a partial sector is indicated by the driver, the partial sector flag is set.
2. `scsi_transfer` intercepts the call when it detects a partial sector and calls this special fast-handshake routine.
3. Then, HP calls the Fujitsu "Programmed Transfer" and handshakes in or out the requested amount - and then handshakes in or out null bytes.

scsi_man_xfr

This routine transfers one data byte. It uses the standard algorithm specified in the *Fujitsu MB87030 User's Manual.* If the Fujitsu chip has glitched ATN, reset the ATN line. HP does it here the first byte after a reselection is guaranteed to be a message byte.

The Message/Status Transfer Circuitry

This section discusses routines used for messages and status reports.

scsi_mesg_out, scsi_mesg_in, & scsi_status

All three routines are rather similar. Each has a sanity check for the correct bus phase. All message bytes and status bytes are handshaked via `scsi_man_xfr`.

scsi_set_state

Find the next phase requested by SCSI. It may happen that the bus has gone to bus free, in which case you set the state to *defaul.* During a manual transfer, you may experience an error, such as a parity error without getting an interrupt from the chip. Thus, after transferring data in manual mode, and then determining the next bus phase, you check the **SERR** Register to see if an error has occurred. If an error did occur, you print a diagnostic to the message buffer and set the state to `scsi_error` and exit.

SCSI Error Handling

This section discusses error handling routines.

scsi_abort

This awkward and difficult routine provides a way to avoid pulling RST on SCSI. The goal is to get to bus free in case of a problem. Study the code and algorithm in the code for a complete explanation. Notice that you may have to clean up DMA.

The Disc Device Driver 7

This chapter provides a walkthrough of the routines, code, and information related to the disc device driver. Table 7-1 lists and describes the topics.

Table 7-1. Disc Device Driver Topics

Topic	Description
The SCSI Data Structures	A brief description of data structures used by SCSI.
Open and Close Routines	The section describes these routines: **`scsi_open`** (opens a device) **`scsi_close`** (closes a device) **`scsi_nop`** (auxillary procedures for open/close routines)
The Operating System Interface	The section describes these routines: **`scsi_strategy`** (performs read/write to peripheral) **`scsi_read`** (reads character device files) **`scsi_write`** (writes character device files) **`scsi_ioctl`** (controls I/O)
SCSI Control	Describes a utility for setting up activity other than reading or writing.
The Finite State Machine	The section describes these routines: **`scsi_fsm`** (the heart of the device driver)
Writing to a Disc	Describes the process for writing to a disc.
Handling Errors	Describes how to deal with errors.

The SCSI Data Structures

The scsi disc driver maintains a structure per physical device (`scsi_od`). `scsi_od` has an *iobuf* for synchronizing activities per device. Only one process can access a device at a time. In addition, several other fields are maintained for issuing the request sense command, and for the special `ioctl` call.

We also keep a structure (`scsi_olun`) for each opened unit. It maintains some unit specific data (such as the size of a block). We needed a separate structure from `scsi_od` since two units sharing a controller may have different characteristics.

Open and Close Routines

This section discusses the open and close routines.

scsi_open

If a device is not opened, the driver allocates a `scsi_od` structure and a `scsi_olun` for it. The driver then bumps the open counts and calls:

1. `scsi_test_unit` to check that the drive is responding;
2. `scsi_inquiry` to determine "what is out there"; and
3. `scsi_read_capacity` to determine the size of the drive and the size of a logical disc block in bytes. The values for these two last numbers are kept in:

`up->lunsize`	(size in blocks)
`up->log2blk`	(log2 of the block size)

An "open device" structure and an "open logical unit" structure is kept per unit. Only one open device structure is allocated for all opens on individual units. The structure contains the device specific information.

The `scsi_od` structures are allocated by `sunit_open(bp)`. The per unit information (e.g. capacity and blocksize) are kept in `scsi_olun` and are allocated by `scsi_olun`.

scsi_close

This is a simple routine except for the call to:

```
(void)scsi_control(dp, up, scsi_nop, NULL, 0, NULL, 5*HZ, 1, 0);
```

which was inserted because of the asynchronous nature of HP's O/S. Upon shutdown the O/S issues the *reboot* intrinsic, which calls *sync*. It returns immediately and shuts the O/S down. But the driver may not be done! To synchronize, HP places a call to a trivial FSM that assures the request queue is flushed.

sunit_close

The routine deallocates the `scsi_olun` structure. The pool of structures, `scsi_olun` and `scsi_od`, are allocated and reused by possibly differing devices over several invocations of the *open* and *close* system calls.

scsi_nop

Handles auxillary procedures to the drivers open and close routines. It associates `scsi_olun` and `scsi_od` structures on a per device and per unit basis.

The Operating System Interface

This section discusses routines that can invoke an activity on SCSI.

scsi_strategy

This routine is called by the file system to perform a read or write to the peripheral. It is also linked with the special ioctl path to issue a user supplied command.

The routine initializes several values in the buf structure, sets `b_clock_ticks` to the value COMMAND_TIME (this is maximum value a device is allowed to detach after receiving the command). For normal reads and writes it is 5 seconds. HP sets the ATN_REQ flag in `b_flags` to indicate allowing the target to disconnect.

HP calls `bp_check`, which performs several activities on behalf of the driver:

- Is the buffer odd-byte aligned? (it's not allowed)
- Is the start of the transfer device sector aligned?
- Is the length of the transfer a whole number of device sectors?
- Driver strategy for handling end of volume:
  - `b_resid` is set to requested amount (`b_bcount`).
  - `b_bcount` is possibly cut back due to end of volume.
  - The driver attempts to transfer up to `b_bcount` bytes,
  - Decrements `b_resid` as it goes.
  - Afterwards, `b_resid` reflects residual due to either end of volume or error
- Does the request start within range?
- If the request goes beyond the end of volume, cut it back if it is a user raw request not from the pageout deamon.
- Trim count to just the number of bytes remaining on device.
- Return **0** if no error, else call `iodone` and return **1**.

scsi_read & scsi_write

This path (for character device files only) bypasses the file system buffer cache. Both routines call `physio` before calling `scsi_strategy`.

`physio` is used for raw I/O. The arguments are:

- The strategy routine for the device.
- A buffer, which will always be a special buffer.
- The header owned exclusively by the device for this purpose.
- The device number.
- Read/write flag.

Essentially, the work amounts to computing and validating physical addresses. If the user has the proper access permissions, the process is marked *delayed unlock* and the pages involved in the I/O are faulted and locked. After the completion of the I/O, the above pages are unlocked. The routine eventually calls `physstrat`, which in turn calls the driver's strategy routine.

scsi_ioctl

This is a simple routine (except the `CMD_MODE` which is explained in a separate section below). Three parameters are used:

- Buf header
- Flag (to identify the request)
- An address of a buffer.

SCSI Control

The `scsi_control` utility sets up for activities other than read/write. It is used internally for open activities, and also by `ioctl`. It knows SCSI specific things. Items to consider include:

- Open device pointer.
- Open LUN pointer.
- Requested FSM (usually `scsi_fsm`).
- `proc` (command to be issued).
- `dev` (device includes both Major and Minor number).
- `addr` (address of data buffer).
- `clock_ticks` (how long to wait after a command is issued and the device issues a disconnect).
- `atn_flag` - allow for device to disconnect.
- `parm` - parameter to be passed in (such as interleave for the `mediainit` command).

A buf header is obtained via a call to `geteblk`. Several entries in the buf header are initialized and the activity is enqueued. We wait for the activity to complete!

If the device issues synchronous transfer data, mark it at this point. If no error has occurred, do several special activities. If the command (`proc`) was `read_capacity`, initialize the appropriate entries in the `scsi_olun` structure.

Finite State Machine

The FSM provided by the **scsi_fsm** routine is the heart of the entire device driver. Each SCSI bus phase has a corresponding state in the FSM. In fact, the states are driven by both hardware (SCSI bus phase requested by target) or the phase requested by software.

```
states {
        /* States assigned by software */
        initial=0,
        select,
        select_nodev,
        select_TO,
        transfer_TO,
        reselect,
        scsi_error,

        /* Following states assigned by ISR */
        phase_data_out,
        phase_data_in,
        phase_cmd,
        phase_status,
        phase_mesg_out,
        phase_mesg_in,

        defaul  /* default state - should never get there */
        }
```

Writing to a Disc

Consider a typical complete disc transaction for the `WRITE` command:

```
SCSI Bus Phases                 DESCRIPTION                FSM STATE
---------------
  ..... bus free ....
arbitration                \    <handled by Fujitsu        initial
selection device with ATN /     chip as a single           select
                                request>
mesg_out                        identify                   phase_mesg_out
command                         write                      phase_cmd
mesg_in                         disconnect                 phase_mesg_in
          ..... bus free ....
mesg_in                         identify                   phase_mesg_in
data_in/data_out                <data transfer>
mesg_in                         save_data_pointer          phase_mesg_in
mesg_in                         disconnect                 phase_mesg_in
          ..... bus free ....
mesg_in                         identify                   phase_mesg_in
mesg_in                         restore_pointers           phase_mesg_in
data_in/data_out                <data transfer>
mesg_in                         save_data_pointer          phase_mesg_in
mesg_in                         disconnect                 phase_mesg_in
          ..... bus free ....
mesg_in                         identify                   phase_mesg_in
status                          status                     phase_status
mesg_in                         command complete           phase_mesg_in
```

You can trace the above sequence via an HP-UX `write` request. (The non-driver details are left to the reader's imagination.) A request from the operating system to write to a disc is made to **`scsi_strategy`**. The type of request (write or read) is determined by the low-order bit of the **`b_flags`** entry of the buf structure. The buf header is initialized in the strategy routine, **`bpcheck`** checks various parameters and calls **`enqueue`**.

Noting the Queuing Strategy

`Enqueue` enqueues requested activity on the *iobuf* (the per-device queue header). The *iobuf* is in the `scsi_od` structure. If the *iobuf* can be immediately serviced, the action for the queued item is started. All subsequent queuing is on the select code or DMA level, and thus, `selcode_dequeue` and `dma_dequeue` are called.

`Queuestart` is called from `enqueue`, and if the device is free (*iobuf* is not locked) the `b_state` is initialized, and the `b_action` routine is called (the SCSI FSM in our case). Enter `scsi_fsm` with state = *initial*.

Moving to Initial State

In `scsi_fsm` (state = *initial*), you follow the transitions in processing a transaction. Enter the FSM via `queuestart`. The state is *initial*, so set the next state to *select*, call `get_selcode` (which puts you on the queue of processes waiting for the select code, and exit. Completely out of the driver now, you are not sleeping, merely waiting. At some point, someone calls `selcode_dequeue` and your process is called with the state = *select*. Start time, and call `scsi_select`. One of two events should occur:

- If no device responds, the Fujitsu chip will timeout, cause a **Timeout** interrupt, and set our state to *select_nodev*. Then the ISR calls the FSM so you enter the state `select_nodev`, drop the select code, call `queuedone`, and exit.
- If the device responds, you get a **Command Complete** interrupt. The ISR sets the state for you, based on the requested bus phase. If you selected with ATN (usually the case), you expect but do not require the phase to be `mesg_out`.

Continuing

The next entry to the FSM is via the ISR (either a **Timeout** or a **Command Complete** interrupt). If the **command complete** interrupt occurs and you assume going to `phase_mesg_out`, you have a problem concerning the previous state. Since the FSM is stateless, keep the previous state in `b_phase`. If it is `SCSI_SELECT`, issue the `identify` message to set bit 6 on allowing disconnect/reconnect. Call `scsi_mesg_out` to send the byte out to the device and expect the target to respond with a new bus phase. Go to reenter.

If the next phase is the command phase, proceed to that state in the FSM. HP stores the command in `b_action2` (from `scsi_control` or `scsi_strategy`) and calls the corresponding procedure (such as `scsi_xfer_cmd`). This eventually filters down to the interface driver, which pumps the bytes out across the bus. In this state in the FSM, HP finally initializes the *iobuf* structure that holds the running transfer count and buffer address. The call to `scsi_xfer_cmd` initialize the values in the command string.

<The following page continues the disc discussion.>

Exiting

Exit the state machine, and wait for an interrupt. At that time, reenter the FSM with the state set to `mesg_in` and enter that state. This state calls `scsi_mesg_in` and reads all the message bytes available in the `mesg_in` phase (e.g. multiple message byte packets are anticipated). On getting the disconnect message, call `WAIT_for_reselect` and exit. Recall from the above discussion, that `WAIT_for_reselect` puts you on a queue of processes waiting for a reselection, and drops the select code, still having the *iobuf* locked.

The device drives the bus to bus free, and does things private to the drive (e.g. decode the command, seek, internal maintenance, etc.). Eventually (milliseconds later), the device reselects HP. The Fujitsu chip will cause a Reselect interrupt, call `scsi_call_isr` (and find you are waiting for the interrupt), set the `b_state` to Reselect, and call the FSM. Although the bus is being driven by "your" device, you are not guaranteed that the select code is free. Call `scsi_set_state`, and then `get_selcode` and exit. Eventually, you return with the state *mesg_in* (the device will send an identify and possibly other messages) and you eventually enter the FSM with the state set to *data_transfer* (`phase_data_out` since you are writing).

Call `scsi_transfer` (with `MAX_SPEED` specified), which does all all the work in firing off the transfer.

Eventually you get to the status routine. Read the status byte, and if it is non-zero, request status.

Getting Additional Information

While this provides an idea about how the FSM works, details cannot be easily described. Serious kernel hackers should do considerable code-walking and code-reading. The *HP-UX Driver Development Guide* has some hints for code-walking.

ERROR Handling

This section provides some hints for handling errors by discussing the error path.

scsi_req_timeout, scsi_select_timeout, & scsi_dequeue

Imagine you are in the FSM, more specifically, you are in phase `phase_cmd` and intend to write a command out to the unit.

You can `START_TIME` with a timeout parameter of `COMMAND_TIME`, set `b_flags` bit `TO_SET`, call the `b_action2`, and break. The `START_TIME` command which follows,

```
START_TIME (scsi_req_timeout, COMMAND_TIME);
```

is actually a macro that must be paired with `END_TIME`.

The methodology is to call `END_TIME` as soon as you reenter the FSM (if the `TO_SET` bit in `b_flags` is set). The above command sets up a timeout that calls the routine `scsi_req_timeout` after `COMMAND_TIME` has expired. If `END_TIME` is called before `COMMAND_TIME` ticks, the timeout is canceled.

What If the Command Fails

Suppose the command fails for unknown reasons and the timeout goes off (`END_TIME` has not been called). The clock ISR calls the specified routine at level 5. You now execute an error recovery path so that `scsi_req_timeout` prints a diagnostic to the message buffer and calls the macro `TIMEOUT_BODY`.

```
TIMEOUT_BODY (iob->intloc,scsi_dequeue,bp->b_sc->int_lvl,0,transfer_TO)
```

The function of the macro is to set up a software trigger that calls `scsi_dequeue` when the interrupt level drops down to `bp->b_sc->int_lvl` and sets the state to *transfer_TO*. (The **0** means BEFORE a real hardware interrupt at that level. A **1** would imply software trigger `scsi_dequeue` after any pending interrupts at that level.)

You now wait until `software_trigger` triggers `scsi_dequeue` which immediately calls the FSM, drops into state *transfer_TO*, and *escapes* (the recovery routine is now entered). After the recovery routine completes, return to `scsi_dequeue` and loop dequeuing activities on the select code and DMA.

scsi_decode_status

HP issues the command **scsi_request_sense** whenever the status byte is non-zero. When HP receives the data back from the command, HP calls **scsi_decode_status**.

scsi_decode_status decodes the information. The one important point is the strategy of retrying most commands exactly once. That is, if an error of some sort occurs (e.g. a parity error), retry the command once. A bit in the **b_flags** field determines whether the command is being retried and HP keys off of that bit.

The Ioctl Path

8

A functionality called the **ioctl path** accomodates user's requirements for specialized SCSI support. It lets you add support without modification of the driver because the support is provided by **userland** programs alone (i.e. you do not work in the kernel per se).

Table 8-1. Overview of the Ioctl Path

Item	Description
Intended Use	The ioctl path lets users pass a specified command directly to a (disc) device conforming to the SCSI standard for DADs and requirements imposed by the HP-UX SCSI driver.
An Illustrative Situation	You have a disc and want to access certain diagnostic logs. The SCSI device driver does not support such a command. Via the ioctl path, a userland program can package the command and have the driver pass it directly to the drive.
Potential Users	A disc that does not fully work with the HP driver might need support; for example: * A CD-ROM player might need special commands to "play audio". * An autochanger disc unit might require commands to "load/unload" a media from a library. The devices might in all other respects conform to HP's ordinary concept of DAD.
Requirements	The unit must be "disc-like". The support resides solely in a userland program, no kernel modifications are expected. The command must conform to the basic SCSI command formats (6, 10, or 12 byte command; with the standard sequence of SCSI phases). Two special requirements are: * the effective user ID must be root, and * the unit must be locked (i.e. single access). The two requirements provide safety because the path lets a user download THEIR OWN COMMANDS DIRECTLY to a unit. The driver cannot check a call that might be very destructive. For example, a simple error in byte position might convert an innocent inquiry command into a format unit command!

The Header File

The `scsi.h` header file contains the required declarations shown in the following code:

```
include <sys/scsi.h>

#define CMD_LEN 12      /* maximum # data bytes in the cmd message */

struct scsi_cmd_parms {
   char cmd_type;          /* command type (6, 10, or 12 byte) */
   char cmd_mode;          /* environment (select with ATN) */
   long clock_ticks;       /* timeout for data xfr phase */
   char command[CMD_LEN]; /* SCSI Commnd to be sent */
};

This structure is used to specify the command, and establish the
appropriate parameters for the driver.  Specifically,

   'scsi_cmd_parms.cmd_type'   is used to specify whether the command
                               is 6, 10, or 12 bytes in length.

   'scsi_cmd_parms.cmd_mode'   This is currently used only to allow
                               the driver to select the target with ATN
                               (which allows the Target to disconnect).
                               Other  bits  may  be  used  at a future point.

   'scsi_cmd_parms.clock_ticks' Specifies the maximum disconnect time.
                                Specifically, it is the maximum time from
                                the disconnect message following the
                                command phase, until the time the target
                                reselects the host.

   'scsi_cmd_parms.command[]'   The actual command.

The following defines are used to access the special Ioctl feature:

#define SIOC_CMD_MODE   This establishes the environment.  It makes
                        sure the effective ID is super-user, and then
                        locks the device (if not in use), otherwise
                        it returns an error.
#define SIOC_SET_CMD    Sets the command to be issued to SCSI.
#define SIOC_XSENSE     Returns the sense from the last request sense
                        command.  Notice, that if the returned status
                        is 0, the 'request_sense' command is not issued.
```

This page is intentionally blank.

The scsi-ioctl Command

The source code on this and the following page shows the `scsi_ioctl` command.

```
 1  scsi_ioctl(dev, order, addr, flag)
 2  dev_t dev;
 3  int order;
 4  caddr_t addr;
 5  int flag;
 6  {
 7          struct scsi_od *dp;
 8          register struct scsi_olun *up;
 9          register int err=0;
10          int i, blksize=1;
11
12          if ((up = sunit_opened(dev, &dp)) == NULL)
13                  panic("scsi_ioctl: unopened device");
14
15          switch  (order) {
16                  ..............
17              case SIOC_CMD_MODE:
18                     if (!suser()) {
19                          err = EPERM;
20                          break;
21                  }
22                  if (*(int *)addr) {
23                          err = dp->cmd_mode_dev ||
24                          up->lun_open_cnt != 1
25                                  ? EBUSY : 0;
26                          if (!err)
27                                  dp->cmd_mode_dev = dev;
28                  } else {
29                          err = (dp->cmd_mode_dev != dev) ? EPERM : 0;
30                          if (!err)
31                                  dp->cmd_mode_dev = 0;
32                  }
33                  break;
34              case SIOC_SET_CMD:
35                  i = (int)((struct scsi_cmd_parms *)addr)->cmd_type;
36                  err = dp->cmd_mode_dev != dev ? EACCES :
37                          (i != 6 && i != 10 && i != 12) ?
38                                  EINVAL : 0;
39                  if (!err)
40                       bcopy(addr, &dp->cmd_parms,
41                          sizeof(struct scsi_cmd_parms));
42                  break;
```

```
43              case SIOC_XSENSE:
44                      bcopy(&dp->status, addr, sizeof(struct xsense));
45                  break;
46          }
47          return err;
48  }
```

The following table discusses the lines.

Line Nos.	Description
Lines 1-5	In the **scsi_ioctl** declaration: **dev** is the device **order** is the specified ioctl command (e.g. SIOC_SET_CMD) **addr** is the address of buffer **flag** is unused
Line 12	The call to **sunit_opened** initializes the pointer to the **scsi_od** table (open device structure), and returns the pointer to **scsi_olun** (open logical unit structure).
Line 15	Switch on particular request (unrelated ioctl calls deleted here)
Line 17	SIOC_CMD_MODE
Line 18	If not super user, return error (EPERM) else on line 22,
Line 22	If the first element of the buffer is non-zero, you are trying to open the device in command mode. If it is zero, you are shutting off command mode to allow normal access.
Line 23	If the device is already opened in command mode, or the open count is non-zero, return error (EBUSY). Otherwise, open it in command mode.
Line 29	The device you close in command mode better be the same device you opened!
Line 34	SIOC_SET_CMD: (Decode scsi_cmd_parms pointed to by addr)
Lines 35-38	First element of structure is cmd_type. Check the value.
Line 40	Copy the buffer into the internal **scsi_cmd_parms** structure maintained by the driver. Whenever the device is in command mode, the command sent to the drive is found here! This is why HP has exclusive opens!!! Normal reads or writes are not possible.
Line 44	Return the last extended sense returned from that device.

A CD-ROM Case Study

This section contains an example of how to use the ioctl path. The case is supplying complete support for a third-party CD-ROM player. While the case has relative simplicity, it shows a complete study of providing support for a third-party's product.

The Case

A CD-ROM will identify itself as a read-only device that can respond to common SCSI commands such as `test_unit_ready`, `read_capacity`, `inquiry`, and `read`. A CD-ROM will fail for a `write` command and for `format_unit`, but that is to be expected.

To provide full support, the device needs additional functionality; specifically, to:

1. control the audio features via special commands, and
2. provide a way of ejecting the disc via software.

Discussion

In the world of CD-ROM's, there are two types of discs: AUDIO and DATA. The type of disc is determined by a header on the disc, and is accessible only via the unit's controller. When a data disc is inserted into the CD-ROM, the HP command set provides all the required functionality to access the disc except for ejecting the disc via software (the command is naturally called `eject`). (No front panel operation is available.)

Beyond this, recall that you want to also provide a way to turn on and turn off the AUDIO mode when an AUDIO disc is inserted (so the user could plug-in their earphones and use the CD-ROM as a standard CD-ROM player). This means, you need to implement two new commands `play_audio` and `pause`.

Finally, since there are two types of discs, you want a way to determine the type of disc currently in the player (and some additional status). A command called `disc_information` is needed.

Writing the First Program

As a start, you have the program named `eject.c`:

```
#include <stdio.h>
#include <sys/scsi.h>

struct scsi_cmd_parms scsi_cmd = {
        12, 1, 500,
        0xe4, 0x00, 0x00, 0x00, 0x00, 0x00,
        0xe0, 0x00, 0x00, 0x00, 0x01, 0x00
};

main(argc, argv)
int argc;
char *argv[];
{
        int fd, ret, i, flag=1;
        unsigned char *ptr, buf[256];

        if (argc != 2)  {
                fprintf(stderr, "Usage: %s device\n", argv[0]);
                exit(1);
                }
        if ((fd=open(*++argv, 2))<0)    {
                perror("stop: error in open");
                exit(1);
                }

        if((ret = ioctl(fd, SIOC_CMD_MODE, flag))<0)    {
                perror("error in ioctl setup CMD MODE");
                exit(1);
                }

        if (ioctl(fd, SIOC_SET_CMD, &scsi_cmd) <0 ||
                        (ret = read(fd, buf, 0xff)) < 0 )       {
                perror("error in ioctl SET CMD MODE");
                exit(1);
                }
        exit(0);
}
```

Looking at the Code

Lines 4-8 Initialize the structure so that:
`cmd_type` is a 12-byte command;
`cmd_mode` allows disconnect;
`clock_ticks` does timeout for maximum disconnect (10 seconds);
`command` is the hexadecimal encoding of actual command to be passed to drive.

Line 17 Checks for usage;

Line 21 Opens the device;

Lines 26-29 Sets up CMD_MODE

Lines 31-35 Issues the command; the "ioctl" call sends the command down to the driver and the read actually initiates the command. If the ioctl call fails, the read is not performed. A diagnostic is written if either command fails.

Noting the Specifications

You just looked at the code. Table 8-2 shows the description from the Specification Document for the CD-ROM. Study the code, the specifications, and the following comments on the lines to get the picture.

Table 8-2. Eject Command Specifications

Bit Byte	7	6	5	4	3	2	1	0
0	Operation Code = 0xE4							
1	LUN = 0			Reserved = 0				
2	Reserved = 0							
3	Reserved = 0							
4	Reserved = 0							
5	Reserved = 0							
6	Reserved = 0							
7	Reserved = 0							
8	Reserved = 0							
9	Reserved = 0							
10	Reserved = 0							EJC
11	VU = 0		Reserved = 0				0	0

The EJC bit field (Eject) is set. If you desire a check condition status, the CD-ROM has the medium removal condition set.

Writing the Second Program

The second program turns on the AUDIO mode so the user can play an AUDIO disc. The program appears on this and the next two pages. A following section examines the lines.

```
1   #include <stdio.h>
2   #include <sys/scsi.h>
3
4   struct scsi_cmd_parms get_info = {
5           12, 1, 500,
6           0xe3, 0x00, 0x00, 0x00, 0x00, 0x00,
7           0x00, 0x00, 0x00, 0x00, 0x03, 0x00
8   };
9
10  struct scsi_cmd_parms pause = {
11          12, 1, 500,
12          0xe1, 0x00, 0x00, 0x00, 0x00, 0x00,
13          0x00, 0x00, 0x00, 0x00, 0x00, 0x00
14  };
15
16  struct scsi_cmd_parms spinup = {
17          6, 1, 500,
18          0x1b, 0x00, 0x00, 0x00, 0x01, 0x00,
19  };
20
21  struct scsi_cmd_parms play_tracks = {
22          12, 1, 500,
23          0xe2, 0x01, 0x01, 0x00, 0x00, 0x00,
24          0x00, 0x01, 0x00, 0x00, 0x00, 0x00
25  };
26
27
28  main(argc,argv)
29  int argc;
30  char *argv[];
31  {
32          int fd, c, ret, i, flag=1, first_track=1, last_track,
                            num_tracks=0;
33          unsigned char buf[256];
34          char    *name, *begin = NULL, *num = NULL;
35
36          extern  int opterr, optind;
37          extern  char    *optarg;
39          opterr=0;
```

<*Code continues on the following page.*>

```
40          while (( c = getopt(argc, argv, "t:T:n:N:")) != EOF) {
41                  switch (c) {
42                  case 'T':
43                  case 't': begin = optarg;
44                            sscanf(begin,"%d",&first_track);
45                            break;
46                  case 'N':
47                  case 'n': num = optarg;
48                            sscanf(num,"%d",&num_tracks);
49                            break;
50                  default:  usage(argv[0]);
51                  }
52          }
53
54          if ((name = argv[optind++])==NULL || argv[optind] != NULL)
55                  usage(argv[0]);
56          if ((fd=open(name, 2))<0)       {
57                  perror("play: error in open");
58                  exit(1);
59                  }
60          if((ret = ioctl(fd, SIOC_CMD_MODE, flag))<0)    {
61                  perror("error in ioctl setup CMD MODE");
62                  exit(1);
63                  }
64
65          if(num_tracks == 0) {   /* Play to the end */
66              if ((ret = ioctl(fd, SIOC_SET_CMD, &pause))<0)
67                  perror("error in ioctl pause"), exit(1);
68              ret = read(fd, buf, 0xff);
69              sleep(1);
70              if ((ret = ioctl(fd, SIOC_SET_CMD, &get_info)<0))
71                  perror("error in ioctl get_info"), exit(1);
72              ret = read(fd, buf, 0x03);
73              num_tracks = buf[2]-first_track+1;
74          }
75          last_track = first_track + num_tracks-1;
```

<The code continues and ends on the following page.>

```
   76           printf("first_track = %d  last_track =
%d\n",first_track,last_track);
   77           play_tracks.command[2]  = (unsigned char)(first_track & 0xff);
   78           play_tracks.command[7] = (unsigned char)(last_track & 0xff);
   79
   80           ioctl(fd, SIOC_SET_CMD, &pause);
   81           ret = read(fd, buf, 0xff);
   82           ioctl(fd, SIOC_SET_CMD, &spinup);
   83           ret = read(fd, buf, 0xff);
   84           ioctl(fd, SIOC_SET_CMD, &play_tracks);
   85           ret = read(fd, buf, 0xff);
   86
   87           close(fd);
   88           exit(0);
   89   }
   90
   91
   92
   93   usage(name)
   94   char *name;
   95   {
   96       fprintf(stderr,"usage: %s \n\
   97          [-t <first track>] \n\
   98          [-n <number of tracks>] \n",name);
   99       exit(1);
  100   }
```

Looking at the Lines

Line Nos.	Description
Lines 4-25	Specification of the several commands required to implement "play_audio".
Lines 39-52	You are getting fancy here: the user can optionally specify which track they wish to start playing, and how many tracks they want to listen.
Lines 54-56	Error checking for usage.
Lines 60-63	Set up command mode
Lines 65-74	Details of how to compute what tracks to play. The parameters are used in the command play_tracks, and notice that the actual command is modified accordingly on lines 77 & 78.
Lines 80-85	First pause disc (in case it is playing); next spin it up; finally play the disc.

Providing for All Users

The program called `eject` uses `SIOC_SET_CMD`, which requires root permission. Users need access to the command. So execute:

```
% cc eject.c -o eject
```

to compile the program. Become the root user and execute:

```
# chown root eject
# chmod 755  eject
# chmod u+s  eject
```

Miscellaneous Hints 9

This chapter contains assorted information related to modifying, writing, and debugging drivers. The chapter has no particular order or structure. Read it according to your needs.

Some Hints

This section has ideas, hints, and ramblings about kernel development.

After developing kernel code for several years, you learn about the facts of life the hard way. Many hours were lost on really dumb mistakes. Our tools are getting better all the time though . These tools (such as SCCS) help us recover from bad mistakes. Other tools, such as the kernel debugger and `msg_printf` help us develop much, much quicker. (As an example, `scsi.c` and `scsi_if.c` were developed from scratch to its current form in less than 9 months.)

Avoiding Complex Tasks

Avoid the tendency to jump into extremely difficult tasks that lie outside your ability and then call them challenges. Accepting a challenge can become an insurmountable problem. Make every attempt to minimize source changes. Instead of changing source code, use the `ioctl` path whenever possible (see Chapter 8).

Working Effectively and Efficiently

There always seems to be a shortage of tools. Never mind that. Use available tools and time-tested procedures to ease the burden of developing a driver. The following items mention useful ideas or procedures:

- At each point in writing a driver, make extremely small changes and debug them before proceeding.
- Delta each change into an SCCS file (RCS if you prefer).
- Test EACH CHANGE. Have a test procedure or test suite available.

In relation to testing changes, the following items suggest things to do:

- Boot off of the device.
- See if the device "rootable".
- Design a test per change, making a test assure that changes are backward compatible.
- Run a write/read exerciser.
- Run a full disc random write/read exerciser.
- Try non-standard paths (besides the customary write/read) such as *mediainit*.
- Try to force error paths (one technique is to use the debugger to set a breakpoint and force a timeout by using faulty equipment (keep bad tapes, bad microfloppies, discs with no spares available in your desk for QA!)).
- Create a stress test that will run (hopefully) without an error for an extended period of time (try a 48-hour stress tests that thrashes the discs).
- Create a session that imitates what users actually do with discs.

Some of these techniques for debugging were discussed in other places, but in general, the manual assumes you know how to do these things.

Debugging Techniques

You need to use the kernel debugger. If necessary, learn how to boot HP-UX from the debugger and then learn to use the debugger before you start modifying source. If you need more information, the *HP-UX Driver Development Guide* has a chapter on using the kernel debugger.

First Steps in Debugging

The following items mention essential first steps in debugging a driver:

1. setting breakpoints,
2. dumping stacks,
3. looking at registers,
4. examining buffers,
5. single stepping procedures,
6. reading assembly code .

Use printf With Caution

To continue debugging, the kernel *printf* can provide diagnostic data, but it has drawbacks:

- very intrusive
- cannot be easily saved
- difficult to implement in the interface routine.
- do not obey userland rules (e.g. no ^S or ^Q).
- alter the timing and should be used with care.

In general, try not to create new problems with *printf.* Instead, *msg_printf* statements that can be turned off and on by poking registers with the debugger can provide diagnostic information. Such statements are not very intrusive and can be saved. For example, using:

```
> Msgbuf/1000C
```

dumps the message buffer to the debugger screen. Also, by turning the diagnostic on and off, you can avoid timing problems and still save a permanent copy on disc using *dmesg* from userland.

Having Source Code Control

You need to maintain control of your source code. The following items suggest things to do:

- Use SCCS (or RCS) for maintaining revision levels.
- Never make large delta's.
- Test each delta thoroughly.
- Do not use branches unless absolutely necessary.
- Archive the SCCS files regularly. (Think of a worst-case scenario, such as accidentally doing an "rm *" of your files, and think of Murphy's law (imagine the worst case, and it will happen)).

In short, use very systematic means to keep track of and control your source code.

Index

a

b

c

d

e

f

g

h

i

l

m

o

p

q

r

s

t

u

w

MANUAL COMMENT CARD

SCSI Technical Reference

HP Part Number 98265-90010 2/88

Please help us improve this manual. Circle the numbers in the following statement that best indicate how useful you found this manual. Then add any further comments in the spaces below. Thank you.

The information in this manual:

Is poorly organized	1	2	3	4	5	Is well organized
Is hard to find	1	2	3	4	5	Is easy to find
Doesn't cover enough	1	2	3	4	5	Covers everything
Has too many errors	1	2	3	4	5	Is very accurate

fold — —

Particular pages with errors? _____

_____

Comments: _____

_____

_____

_____

_____

_____

_____

Name: _____

Job Title: _____

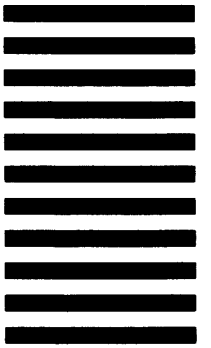
Company: _____

Address: _____

_____

☐ Check here if you wish a reply.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



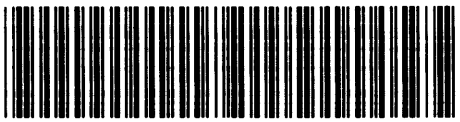
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HP Part Number
98265-90010
Microfiche No. 98265-99010
Printed in U.S.A. 2/88



98265-90610
For Internal Use Only