Chapter 13 Writing SCSI Device Drivers

This chapter presents routines and conceptual material specifically for drivers of SCSI devices. Chapter 4, “Writing a Driver,” describes the general configuration and entry point driver routines, such as driver_open and driver_write. When writing a SCSI driver, provide routines from both Chapter 4, “Writing a Driver,” and this chapter.

The HP-UX 11i v1 Driver Development Reference Guide describes the SCSI Services routines.

SCSI devices can be controlled in two ways, through Kernel drivers and using the scsi-ctl driver, both are supported by the SCSI Services routines. Kernel drivers, following the scsi_disk model, are the traditional method and are described in this chapter and in the scsi_ctl(7) manpage. Many SCSI devices do not need a special driver. Instead, user programs pass ioctl commands to the pass-through driver, scsi_ctl. The pass-through driver is described in HP-UX 11i v1 Driver Development Reference Guide manpage scsi_ctl(7). Refer to mt(7) manpage for discussion of ioctls, device files and driver behavior for tape devices.

The following sections provide the suggested steps for developing a SCSI driver:

- “Step 1: Include Header Files”
- “Step 2: Set Up Structures”
- “Step 3: Create the driver_install Routine”
- “Step 4: Create the driver_dev_init() Routine”
- “Step 5: Analyze Multiprocessor Implications”
- “Step 6: Create the Entry Point Routines”
- “Step 7: Error Handling”
- “Step 8: Underlying Routines”

The examples in this chapter assume the name of the driver is mydriver and are following the routine naming conventions described in Chapter 4, “Writing a Driver.”
Step 1: Include Header Files

See reference pages for each kernel call and data structure the driver uses to find which headers the driver requires.

NOTE

HP recommends that header files the driver doesn’t need are not included. It increases compile time and the likelihood of encountering portability problems. It is not recommended.

General Header Files

/usr/include/sys/buf.h I/O buf structure, buf.
/usr/include/sys/errno.h Defines errors returned to applications.
/usr/include/sys/file.h Defines open flags
/usr/include/sys/io.h isc table structure.
/usr/include/sys/conf.h drv_ops_t structure
/usr/include/sys/kern_svcs.h Kernel services for synchronization
/usr/include/sys/malloc.h Necessary for acquiring and releasing memory.
/usr/include/sys/wsio.h WSIO context data and macro definitions.

Header Files for SCSI Drivers

/usr/include/sys/scsi.h SCSI-specific data definitions and ioctl commands.
/usr/include/sys/scsi_ctl.h SCSI subsystem data and macro definitions.

Header Files for Device Classes

In addition to the header file created for the specific driver, the driver may need other device class specific files.

/usr/include/sys/diskio.h Data definitions for disk ioctl commands (DIOC_XXX). Includes /usr/include/sys/types.h and /usr/include/sys/ioctl.h.
/usr/include/sys/floppy.h Data definitions for floppy ioctl commands.
/usr/include/sys/mtio.h Data definitions for magnetic tape ioctl commands.
Step 2: Set Up Structures

Depending on the characteristics of the driver, it can be set up as a character driver, a block driver, or (as in the case of disk drivers) both.

NOTE

SCSI Services use the locking facilities regardless of whether the driver operates on an Multiprocessor (MP) platform. All drivers using SCSI Services must use the provided data protection routines. It is essential that the C_ALLCLOSES and C_MGR_IS_MP flags in the drv_ops_t structure and the DRV_MP_SAFE flag in the drv_info_t structure are included. See “Step 5: Analyze Multiprocessor Implications” for more information.

Determine the driver’s name and device class, and put this information in the appropriate structures. (See Chapter 4, “Writing a Driver,” for information about these data structures.)

First, declare the driver’s routines that can be called by the kernel. These are used in the following structure:

```c
int mydriver_open();
int mydriver_close();
int mydriver_strategy();
int mydriver_psize();
int mydriver_read();
int mydriver_write();
int mydriver_ioctl();
```

The drv_ops_t structure specifies the “external” driver routines to the kernel. The C_ALLCLOSES and C_MGR_IS_MP flags are required by SCSI Services. See Chapter 4, “Writing a Driver,” for further details.

```c
static drv_ops_t mydriver_ops =
{
    mydriver_open,
    mydriver_close,
    mydriver_strategy,
    NULL,
    mydriver_psize,
    NULL,
    mydriver_read,
    mydriver_write,
    mydriver_ioctl,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    C_ALLCLOSES | C_MGR_IS_MP
};
```
Writing SCSI Device Drivers

Step 2: Set Up Structures

The `drv_info_t` structure specifies the driver's name (`mydriver`) and class (`disk`). Flags define the driver type. The `DRV_MP_SAFE` flag is required by SCSI Services. See Chapter 4, “Writing a Driver,” for further details.

```c
static drv_info_t mydriver_info =
{
    "mydriver",
    "disk",
    DRV_CHAR | DRV_BLOCK | DRV_SAVE_CONF | DRV_MP_SAFE,
    -1,
    -1,
    NULL,
    NULL,
    NULL,
};
```

The `wsio_drv_data_t` structure specifies additional information for the WSIO CDIO. The first field should be “scsi <class>” for SCSI class drivers (for example, “scsi_disk” for the disk class) and “scsi” for SCSI interface drivers. See Chapter 4, “Writing a Driver,” for further details.

```c
static wsio_drv_data_t mydriver_data =
{
    "scsi_disk", /* "scsi_tape" for tape devices */
    T_DEVICE,
    DRV_CONVERGED,
    NULL,
    NULL,
};
```

The `wsio_drv_info_t` structure ties the preceding three together. See Chapter 4, “Writing a Driver,” for further details.

```c
static wsio_drv_info_t mydriver_wsio_info =
{
    &mydriver_info,
    &mydriver_ops,
    &mydriver_data,
    WSIO_DRV_CURRENT_VERSION
};
```
Step 3: Create the driver_install Routine

The `driver_install` routine causes the information that was previously created to be installed into the I/O subsystem, specifically into the WSIO CDIO. External definition of `dev_init` is in `<sys/wsio.h>`.

```c
int (*mydriver_saved_dev_init)();

int mydriver_install()
{
    /* The content of dev_init (a function pointer) needs to be saved here and then must be called by mydriver_dev_init(). This is done to create a linked list of driver_init routines. It is the responsibility of the driver’s init routine to call the init routine of the next driver using this method. */
    mydriver_saved_dev_init = dev_init;
    dev_init = mydriver_dev_init;

    /* register driver with WSIO and return any error */
    return(wsio_install_driver(&mydriver_wsio_info));
}
```
Step 4: Create the driver_dev_init() Routine

Specify the `driver_dev_init` routine from the `driver_install()` routine. The `driver_dev_init` routine calls `scsi_ddsw_init()`, which initializes some fields in the SCSI driver's device switch table (`scsi_ddsw`). This table is independent of the kernel's device switch tables.

```c
mydriver_dev_init()
{
    /*
     * Initialize mydriver_ddsw.blk_major and
     * mydriver_ddsw.raw_major.
     */
    scsi_ddsw_init(mydriver_open, &mydriver_ddsw);

    (*mydriver_saved_dev_init)();
}
```

Setting up the Device Switch Table (scsi_ddsw)

To use SCSI Services effectively, a SCSI driver must define its `scsi_ddsw` device switch structure. This structure contains pointers to special `dd` routines, some of which are executed indirectly by the standard driver routines, such as `driver_read`. The structure is passed to SCSI Services routines from the `driver_open` routine, which calls the `scsi_lun_open()` SCSI Services routine.

SCSI Services has been set up to control the housekeeping and other processing in the SCSI interface. The standard driver routines restrict their operation to calling the appropriate SCSI Services routine, as shown in the examples in “Step 6: Create the Entry Point Routines”. Special processing and customization should all be handled in the special `dd` routines.

For a summary of SCSI Services, see the “SCSI Services Summary” section. For more detailed information, see the HP-UX 11i v1 Driver Development Reference Guide.

The `scsi_ddsw_init()` should be used only for backward compatibility. Do not use it for new drivers.

```c
struct scsi_ddsw {
    u_char       blk_major;
    u_char       raw_major;
    int          dd_lun_size;
    int          (*dd_open)();
    void         (*dd_close)();
    int          (*dd_strategy)();
    int          (*dd_read)();
    int          (*dd_write)();
    int          (*dd_ioctl)();
    struct buf   (*dd_start)();
    int          (*dd_done)();
    int          (*dd_pass_thru_okay)();
    int          (*dd_pass_thru_done)();
    int          (*dd_ioctl_okay)();
    struct status_action *dd_status_list;
    int          dd_status_cnt;
    ubit32       dd_flags;
    wsio_drv_info_t  *wsio_drv;
};
```
The entries are described as follows:

**blk_major, raw_major**
Block and character major numbers; specify them as NODEV. They are initialized by `scsi_ddsw_init()` when it is called from the `driver_dev_init()` routine.

**dd_lun_size**
The number of bytes to be allocated and attached to the `scsi_lun` structure, see Chapter 13, “Writing SCSI Device Drivers,” for the class driver's private data structure. This data structure will be allocated at the first open and deallocated at the last one.

**dd_open(), dd_close(), dd_strategy(), dd_read(), dd_write(), dd_ioctl()**
Pointers to underlying driver specific routines. When the corresponding `driver_routine` is called by the kernel and transfers control to SCSI Services, SCSI Services performs certain overhead operations and calls these routines for driver specific operations.

**dd_start()**
Driver specific start routine.

**dd_done()**
Driver specific post I/O processing.

**dd_pass_thru_okay()**
Driver specific control of pass through I/O.

**dd_pass_thru_done()**
Driver specific notation of pass through I/O.

**dd_ioctl_okay()**
Disallow `ioctl` commands through the pass through driver.

**dd_status_list**
Action list for handling various status/error conditions.

**dd_status_cnt**
Number of entries in `dd_status_list`.

**dd_flags**
Flag bits, currently only `DD_DDG` defined.

**wsio_drv**
Pointer to standard WSIO driver info data structure.

Here is an example of an initialized declaration of the `scsi_ddsw`:

```c
int mydriver_dd_open(); /* dd_open */
void mydriver_dd_close(); /* dd_close */
int mydriver_dd_strategy(); /* dd_strategy */
int mydriver_dd_read(); /* dd_read */
int mydriver_dd_write(); /* dd_write */
int mydriver_dd_ioctl(); /* dd_ioctl */
struct buf mydriver_dd_start(); /* dd_start */
int mydriver_dd_done(); /* dd_done */
int mydriver_dd_pass_thru_okay(); /* dd_pass_thru_okay */
int mydriver_dd_pass_thru_done(); /* dd_pass_thru_done */
int mydriver_dd_ioctl_okay(); /* dd_ioctl_okay */
```
Step 4: Create the driver_dev_init() Routine

The following example shows the scsi_ddsw structure. Specify NULL for routines that are not defined (not using). The first two fields specify the block and character major numbers; they are filled in by the call in driver_dev_init() to the SCSI Services routine scsi_ddsw_init(). The last field points to the wsio_drv_info_t structure that was defined in “Step 2: Set Up Structures”. The first name in each comment is the field name of the scsi_ddsw structure element.

```c
struct scsi_ddsw mydriver_ddsw =
{
    NODEV,    /* blk_major - mydriver_dev_init sets */
    NODEV,    /* raw_major - mydriver_dev_init sets */
    sizeof(struct mydriver_lun),   /* dd_lun_size */
    mydriver_dd_open,           /* dd_open */
    mydriver_dd_close,          /* dd_close */
    mydriver_dd_strategy,       /* dd_strategy */
    NULL,                      /* dd_read */
    NULL,                      /* dd_write */
    mydriver_dd_ioctl,          /* dd_ioctl */
    mydriver_dd_start,          /* dd_start */
    mydriver_dd_done,           /* dd_done */
    mydriver_dd_pass_thru_okay, /* dd_pass_thru_okay */
    mydriver_dd_pass_thru_done, /* dd_pass_thru_done */
    mydriver_dd_ioctl_okay,     /* dd_ioctl_okay */
    mydriver_dd_status_list,    /* dd_status_list */
    sizeof(mydriver_dd_status_list) /
    sizeof(mydriver_dd_status_list[0])
    /* dd_status_cnt */
    mydriver_dd_flags,          /* dd_flag bits DD_DDG */
    &mydriver_wsio_info         /* For Diagnostics Logging; */
    NULL means errors print in dmesg */
};
```
Step 5: Analyze Multiprocessor Implications

Make the class driver **Multiprocessor** (MP) safe, regardless of whether it is to operate an MP platform or not. SCSI Services make use of the kernel's locking facilities, so all drivers that use SCSI Services must use the synchronization routines the kernel provides.

Drivers must do the following:

- Set the `C_MGR_IS_MP` flag in the `d_flags` field of the driver's `drv_ops_t` structure.
- Set the `DRV_MP_SAFE` flag in the `flags` field of the `drv_info_t` structure.
- Use the driver semaphore, driver lock, **Logical Unit Number** (LUN) lock, and target lock as necessary to provide MP protection. Refer to the defines and structures in `/usr/include/sys/scs_ctl.h` for details. This is the largest task, and involves looking at the code and determining whether there are data references that must be protected and which locks and semaphores must be used to protect the references.
- Build a kernel with the driver.
- Test the driver on a **single processor** (UP) system and a **Multiprocessor** (MP) system with a debug kernel if available.
- See Chapter 3, “Multiprocessing,” and the “Synchronization in SCSI Drivers” section of this chapter for additional details.
Step 6: Create the Entry Point Routines

SCSI services provides helper routines to support the various driver entry point routines, which must be called in the driver entry point path.

For example, on open, the SCSI service routine, `scsi_lun_open()` must be called to instantiate the `scsi_lun` structure and other SCSI services data structure. `scsi_lun_open()`, in turn will call back to the driver `dd_open` routine, if not NULL, to perform any subsequent driver-specific actions if needed. The `scsi_lun_open()` routine will simply initialize the inquiry data if no `dd_open` was specified. Similarly, the other entry points have corresponding SCSI services functions (as shown in the following examples) which will call back to the corresponding `dd_xxx` routines if not NULL.

The `scsi_read()` and `scsi_write()` routines will call `physio()` if no corresponding `dd_read`/`dd_write` routines specified (NULL) to complete the request down the stack.

If `dd_strategy` is NULL, `scsi_strategy` will enqueue the `bp` on the `scsi_lun scb` queue for processing down the stack. The `dd_strategy` routine needs to enqueue the passed `bp` on a queue which can be retrieved later during processing of the I/O down the stack.

If `dd_strategy` is provided, one should also provide a `dd_start` routine to dequeue the I/O requests, enqueued by the `dd_strategy` routine, for processing down the stack.

The `dd_ioctl` routine is needed for implementing driver specific `ioctl()` commands.

Examples of each of the entry point routines:

```c

void driver_open()
{
    ....
    return (scsi_lun_open(dev, &mydriver_ddsw, oflags));
    ....
}

void driver_close()
{
    return scsi_lun_close(dev);
}

void driver_read()
{
    return scsi_read(dev, uio);
}

void driver_write()
{
    return scsi_write(dev, uio);
}
```
**driver_strategy()**

The `driver_strategy()` routine does not return anything. It records errors in `bp->b_error`.

```c
mydriver_strategy(struct buf *bp)
{
    scsi_strategy(bp);
}
```

**driver_psize()**

This example assumes that `driver_psize()` is never called when the device is closed. Note the use of the SCSI Services `m_scsi_lun()` function. This routine returns the number of 1K blocks on the device.

```c
mydriver_psize(dev_t dev)
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    int rshift, nblks, size;

    /* Example private fields in llp */
    nblks = llp->nblks;
    rshift = llp->devb_lshift;

    /* Example usage of the files */
    size = rshift > 0 ? nblks >> rshift : nblks << -rshift;

    return size;
}
```

**driver_ioctl()**

```c
mydriver_ioctl(dev_t dev, int cmd, int *data, int flags)
{
    return scsi_ioctl(dev, cmd, data, flags);
}
```
Step 7: Error Handling

The class driver specifies what actions it wants to take on various I/O status conditions via its `dd_status_list` field in `scsi_ddsw` structure. If `dd_status_list` is not specified (`dd_status_cnt` equals 0), then the failed I/O will be completed immediately without any retry.

This status list specifies the retry policy (whether to retry and how long to delay between retrying), an action routine to be called to handle the condition, and an `errno` value to be used on ultimate completion of the I/O in case of failure. The status list keys off of the `cdb_status` field in the `scb` data structure, whose low-order byte is the SCSI status byte from the device and whose high-order byte encodes additional software on the hardware condition.

The SCSI service routine, `scsi_cmdx()`, also allows for a specific status list to be passed in as a parameter.

Data structure definitions for struct `status_action` and struct `sense_action` are available in `<sys/scsi_ctl.h>`.

Structure for each element in the array of status/action pairs attached to `ddsw->dd_status_action_list` and traversed by SCSI services follows.

```c
struct status_action
{
    int status;
    int (*action)();
    intptr_t arg1, arg2, arg3, arg4;
};
```

- **status**: The value to be compared against the `cdb_status` field in `scb` data structure.
- **action**: The action routine to be called if `cdb_status` matches the value specified in the “status” field.
- **arg1, arg2, arg3, arg4**: The arguments to be passed to the action routine when called. The first argument to action routine is always the pointer to the `buf` (bp). The values specified in arg1, arg2, arg3, arg4 would be passed as 2nd, 3rd, 4th and 5th arguments.

For `cdb_status` of CHECK_CONDITION, an additional status list of sense data from the device can be provided to match the sense key, additional sense code, and additional sense code qualifier, to take appropriate actions. The following data structure is used to provide the action list for `cdb_status` of CHECK_CONDITION.

```c
struct sense_action
{
    int status; /* status of the automatic request sense */
    short error_code;
    short sense_key;
    short sense_code;
    short sense_qualifier;
    int (*action)();
    int arg1, arg2, arg3, arg4;
};
```

- **status**: Request sense status byte. The value to be compared against the `sense_status` field in `scb` data structure.
- **error_code**: The value which is to be compared against the “Error code” byte of the request sense data.
sense_key The value which is to be compared against the “sense key” field of the request sense data.
sense_code The value which is to be compared against the “Additional sense code” field of the request sense data.
sense_qualifier The value which is to be compared against the “Additional sense code qualifier” field of the request sense data.
action The action routine to be called if sense_key, sense_code, and sense_qualifier fields match with the request sense data obtained from the device.
arg1, arg2, arg3, arg4 The arguments to be passed to the action routine when called. Please note the first argument to action routine is always the pointer to the buf (bp).

For cdb_status of S_CHECK_CONDITION, the action routine can be SCSI service function, scsi_sense_action(), with the parameters specifying a driver-specific “sense action list”.

In the mydriver_status_list structure, argument one is used as flags, argument two as error no, and argument three is milliseconds between retries (msecs). See the following examples:

```c
struct sense_action mydriver_sense_list[] = {
    { S_GOOD, S_CURRENT_ERROR, S_RECOVERRED_ERROR,
      SA_ANY, SA_ANY, mydriver_check_residue, SA_DONE |
        SA_LOG_IT_ALWAYS, 0 },
    { SA_ANY, SA_ANY, SA_ANY, SA_ANY, scsi_action, 
      SA_DONE + SA_LOG_IT_NEVER, EIO }
};

struct status_action mydriver_status_list[] = {
    { S_GOOD, scsi_action, SA_DONE + 
      SA_LOG_IT_NEVER, 0 },
    { S_CHECK_CONDITION, scsi_sense_action, (int)
      mydriver_sense_list,sizeof(mydriver_sense_list) / 
      sizeof(mydriver_sense_list [0]) },
    { S_CONDITION_MET, scsi_action, SA_DONE + 
      SA_LOG_IT_NEVER, 0 },
    { S_INTERMEDIATE, scsi_action, SA_DONE + 
      SA_LOG_IT_NEVER, 0 },
    { S_I_CONDITION_MET, scsi_action, SA_DONE + 
      SA_LOG_IT_NEVER, 0},
    { SA_ANY, scsi_action, SA_DONE + SA_LOG_IT_ALWAYS, EIO }
};
```

The driver can specify its own routines for handling errors and can break down errors for more granularity. Access the Pass-Thru Driver status using the driver’s dd_pass_thru_done() routine, described in “Step 8: Underlying Routines”.

For tape specific error handling, the head position data returned in MTIOCGET must reflect any changes in head position indicated by sense data. When a Device or Bus reset is done, an error with EPERM is returned on any subsequent media motion commands (for example, read, write, space or write file marker) until the user closes or reopens the device.
Step 8: Underlying Routines

This is where the driver can be as complex as desired, or as the device requires. The `scsi_lun_open()` routine ensures that the bus, target, and LUN of the driver's device are open and able to handle I/O. Specific requirements for the device itself should be addressed in the driver's `ddsw->dd_open()` routine. The same principle applies for `close`, `read`, `write`, and so on.

The call graph in Figure 13-1, “Call Graph of SCSI Routines and Services,” shows how these underlying routines and SCSI services call each other. For a summary list of SCSI Services, see “SCSI Services Summary”. Detailed information on SCSI Services is provided in the HP-UX 11i v1 Driver Development Reference Guide.

Figure 13-1   Call Graph of SCSI Routines and Services
**dd_close**

The `dd_close()` SCSI function, is used to provide driver specific processing during close is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_close` field of the `scsi_ddsw` structure.

If this routine is defined in the `scsi_ddsw` structure, it is called to perform the actual device close processing. For example, for the `scsi_disk` driver, the `sd_close()` function performs the Test Unit Ready and Allow Media Removal commands.

The tape driver supports the following device file attributes:

- No rewind-on-close: Do not rewind media during driver close routine.
- ATT style close: If the device is opened as read only, the driver will space the next file on close if not already at Beginning of Tape (BOT) or End of File (EOF). If the user has written End of Data (EOD) but has not explicitly written an EOD mark, write an EOD in the driver close routine.

Refer to `mt (7)` manpage for more device file attributes supported by a tape driver.

**Conditions**

The `dd_close()` is called from `scsi_lun_close()` in a process context. The open or close LUN semaphore is held when the `dd_close()` function is called. `dd_close()` is not called from within a critical section; it may block.

**Declaration**

```c
void dd_close (dev_t dev);
```

**Parameters**

- `dev` The device number.

**Return Values**

`dd_close()` does not return a value.

**Example**

```c
#include <sys/scsi_ctl.h>
#define ST_GEOM_LOCKED 0x00000002

void mydriver_dd_close(dev_t dev); {
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;

    /*
    * dd_blk_open_cnt() is defined in <sys/scsi_ctl.h>
    * It returns the count of current block opens
    */

    if (dd_blk_open_cnt(lp) == 1) {
        scsi_lun_lock(lp);
        llp->state &= ~ST_GEOM_LOCKED;
        scsi_lun_unlock(lp);
    }
}
```
The \texttt{dd_ioctl()} routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the \texttt{dd_ioctl} field of the \texttt{scsi_ddsw} structure.

If this routine exists in the \texttt{scsi_ddsw} structure it is called by \texttt{scsi_ioctl()} if the ioctl command remains unsatisfied by the choices provided within that SCSI Services procedure. If \texttt{dd_ioctl()} does not exist when called, \texttt{scsi_ioctl()} returns an error.

Examine the ioctl commands provided by SCSI Services in \texttt{scsi_ioctl()} and implement any additional commands needed in the \texttt{dd_ioctl()} routine.

If \texttt{dd_ioctl()} and \texttt{dd_open()} are implemented, some of the more specialized features of SCSI Services may be useful, refer to the following list:

- \texttt{scsi_cmd()}
- \texttt{scsi_init_inquiry_data()}
- \texttt{scsi_mode_sense()}
- \texttt{scsi_mode_fix()}
- \texttt{scsi_mode_select()}
- \texttt{scsi_wr_protect()}

\textit{Table 13-1, “IOCTL Usage by LVM Summary,”} shows a list of IOCTLs that the LVM uses to interface with the class driver.

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIOC_RSTCLR</td>
<td>The LVM uses each of these ioctls on all bus paths.</td>
</tr>
<tr>
<td>SIOC_RESET_BUS</td>
<td></td>
</tr>
<tr>
<td>DIOC_DESCRIBE</td>
<td>The LVM uses each of these ioctl’s for device accessibility, device</td>
</tr>
<tr>
<td>DIOC_BLKLIST_REMAP</td>
<td>type, block size, and control of immediate reporting, and setting the</td>
</tr>
<tr>
<td>DIOC_SET_PF</td>
<td>pftimeout interval on a per LUN path basis.</td>
</tr>
<tr>
<td>TIMEOUTSIOC_GET_IR</td>
<td></td>
</tr>
<tr>
<td>SIOC_INQUIRY</td>
<td></td>
</tr>
<tr>
<td>SIOC_IO</td>
<td></td>
</tr>
<tr>
<td>SIOC_SET_IR</td>
<td></td>
</tr>
</tbody>
</table>

The tape driver must support the ioctl as described in \texttt{mt (7) manpage} in order to interoperate with HP-UX commands, such as \texttt{tar, dd, cpio, mt, st, mc, fbackup} and \texttt{frecover}. The ioctl specific to the tape driver is given in \textit{Table 13-2, “IOCTL Usage by Tape Driver.”}
Writing SCSI Device Drivers

Step 8: Underlying Routines

Table 13-2  IOCTL Usage by Tape Driver

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTIOCGET</td>
<td>The fbackup and frecover commands require that the mt_type field in the MTIOCGET data be filled in with on the following tape types; MT_ISLTO, MT_ISDLT, MT_IS3480, MT_IS3590, MT_ISQIC, MT_ISHPB_REEL, MT_ISSCSI_REEL, MT_ISDDS, MT_ISSMM.</td>
</tr>
<tr>
<td>MTIOCTOP</td>
<td>This ioctl is for media motion operations (for example, space, write file mark, rewind, and so on.) Refer to mt (7) manpage for detailed description on operations. The header file &lt;sys/mtio.h&gt; defines the operations.</td>
</tr>
<tr>
<td>SIOC_SET_BLOCK_SIZE</td>
<td>This ioctl is used for block size settings.</td>
</tr>
<tr>
<td>SIOC_GET_BLOCK_SIZE</td>
<td>This ioctl is used for reading the set block size.</td>
</tr>
</tbody>
</table>

Conditions

The dd_ioctl() is called from scsi_ioctl() in a process context. It is not called from within a critical section; it may block.

Declaration

```c
int dd_ioctl (
    dev_t   dev,
    int     cmd,
    caddr_t data,
    int     flags
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>The command word.</td>
</tr>
<tr>
<td>data</td>
<td>Pointer to the commands arguments.</td>
</tr>
<tr>
<td>dev</td>
<td>The device number.</td>
</tr>
<tr>
<td>flags</td>
<td>The file access flags.</td>
</tr>
</tbody>
</table>

Return Values

The dd_ioctl() is expected to return the following values:

- 0  Successful completion.
- <>0  Error. Value is expected to be an errno.
Example

```c
#include <sys/scsi.h>
#include <sys/scsi_ctl.h>
mydriver_dd_ioctl (dev_t dev,
    int cmd,
    int *data,
    int flags
    )
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    struct scsi_tgt *tp = lp->tgt;
    struct scsi_bus *busp = tp->bus;
    struct inquiry_2 *inq = &lp->inquiry_data.inq2;
    int i;
    switch (cmd)
    { case DIOC_FORMAT:
        case SIOC_FORMAT:
            if (!(flags & FWRITE) && !suser())
                return EACCES;
            if (!(lp->state & L_EXCLUSIVE || tp->state &
                T_EXCLUSIVE || busp->state & B_EXCLUSIVE))
                return EBUSY;
            return mydriver_format(dev, ((struct sioc_format *)
                data)->interleave);
        case SIOC_GET_IR:
            return mydriver_wce(dev, SIOC_GET_IR, data);
        case SIOC_SET_IR:
            if (!(flags & FWRITE) && !suser())
                return EACCES;
            if (*data & ~0x1)
                return EINVAL;
            return mydriver_wce(dev, SIOC_SET_IR, data);
        ... ...
        default:
            return EINVAL;
    }
}
```

**dd_ioctl_okay**

The `dd_ioctl_okay()` SCSI function is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_ioctl_okay` field of the `scsi_ddsw` structure.

The `dd_ioctl_okay()` disallows all `ioctl` commands through the pass-through driver that are not explicitly allowed by any non-pass-through driver that has the device open concurrently.

**Conditions**

`dd_ioctl_okay()` is called from `sctl_ioctl()` in a process context. It is called within a critical section; it may not block.
NOTE
Allow SIOC_INQUIRY for the pass-through driver at all times. SIOC_INQUIRY is allowed by default (if there is no dd_ioctl_okay() routine). SIOC_INQUIRY is always allowed if it will not result in I/O (lp->inquiry_sz > 0), because it does not affect the non-pass-through class driver in any way.

Declaration

```c
int dd_ioctl_okay (
    dev_t dev,
    int cmd,
    caddr_t data,
    int flags
);
```

Parameters

- `cmd`: The command word.
- `data`: Pointer to the commands arguments.
- `dev`: The device number.
- `flags`: The file access flags.

Return Values

`dd_ioctl_okay()` is expected to return the following values:

- `PT_OKAY`: Successful completion.
- `0`: Error

Examples

```c
#include <sys/scsi_ctl.h>

mydriver_dd_ioctl_okay (
    dev_t dev,
    int cmd,
    void *data,
    int flags
)
{
    return PT_OKAY;
}
```

`dd_open`

The `dd_open()` SCSI function is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_open` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, it is called to perform the actual device “open” processing.

As an example, the `scsi_disk` driver's `dd_open` routine, `sd_open()`, calls `disksort_init_queue()` for the LUN’s lun_disk_queue. It calls `scsi_init_inquiry_data()` to set the target state for SDTR and WDTR and send the Start Unit, Test Unit Ready, Prevent Media Removal, and Read Capacity commands, if appropriate, for the type of disk being opened.
This routine can be as complicated as needed to ensure the device is properly open the first time (ensured by checking `dd_open_cnt`). Calling the SCSI Service `scsi_init_inquiry_data()` is reasonable, as is performing Test Unit Ready. Changing state in the `scsi_lun` or target structures requires protection.

The tape driver `O_NDELAY` flag indicates that the device file can be opened even if there is no media loaded.

**Conditions**

The `dd_open()` is called from `scsi_lun_open()` in a process context. The open/close lun_semaphore is held when `dd_open()` is called. `dd_open()` is not called within a critical section; it may block.

**Declaration**

```c
dd_open (  
    dev_t dev,  
    int oflags  
);
```

**Parameters**

- `dev` The device number.
- `oflags` The flags passed in the open call.

**Return Values**

`dd_open()` is expected to return the following values:

- **0** Successful completion.
- **<>0** Error. The value is expected to be an `errno` value.

**Examples**

```c
#include <sys/scsi_ctl.h>

mydriver_dd_open(dev, oflags)
{
    struct scsi_lun   *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    struct scsi_tgt   *tp = lp->tgt;
    struct inquiry_2 *inq = &lp->inquiry_data.inq2;
    struct capacity   cap;
    u_char            cdb[16];
    struct sense_hdr  *hd;
    struct block_desc *bd;
    struct caching_page *c_pd;
    struct error_recovery *e_pd;
    int ret_size, bpb, error, x;

    /*
    * Only first opens are interesting.
    * `dd_open_cnt()` is defined in `<sys/scsi_ctl.h>`.
    *
    * It returns the count of current opens.
    */
    if (dd_open_cnt(lp) > 1)
        return 0;
    ...
    /*
    * Inquiry.
    */
```
* Call the routine provided by services to do any
* necessary synchronization with the pass-through
* driver. Success here does not imply that there is no
* more pending sense data. In fact, the SCSI-2
* standard encourages devices not to give Check
* Condition status on Inquiry, but to defer it until
* a subsequent command. Also, if the inquiry data had
* already been cached as a result of a pass-through
* driver open or SIOC_INQUIRY, this may not even
* result in I/O.
*/

if (error = scsi_init_inquiry_data(dev))
    return error;

...
/*
* Needs protection at LUN and Tgt.
*/
scsi_lun_lock(lp);
scsi_tgt_lock(tp);

tp-h>state |= T_ENABLE_SDTR;
...
scsi_tgt_unlock(tp);
scsi_lun_unlock(lp);
...
bzero(cdb, sizeof(cdb));
cdb[0] = CMDtest_unit_ready;
if (scsi_cmd(dev, SCB_DONT_PRINT, 6, cdb, 0, 0,
    llp->mydriver_msecs, 0,&error))
{
    /*
    * Allow an incomplete open if this is a raw device.
    */
    if (major(dev) == mydriver_ddsw.raw_major)
    {
        scsi_lun_lock(lp);
        lp->state |= L_DISABLE_OPENS;
        scsi_lun_unlock(lp);
        return 0;
    }
    return error;
}
...

dd_pass_thru_done

The dd_pass_thru_done() routine is provided by the driver writer. It can have any unique name. Pass the
name to SCSI Services by specifying it in the dd_pass_thru_done field of the scsi_ddsw structure.

If this routine exists in the scsi_ddsw structure, SCSI Services executes it on completion of a pass-through
I/O. It allows the class driver to make note of any I/Os which have occurred and any resulting status and/or
sense data.
The `dd_pass_thru_done()` function is called from within a critical section; it is not permitted to block.

**Declaration**

```c
int dd_pass_thru_done (
    struct buf *bp
);
```

**Parameters**

bp .buf structure

**Return Values**

`dd_pass_thru_done()` is declared as returning `int`; however, the return is not used by SCSI services.

### dd_pass_thru_okay

The `dd_pass_thru_okay()` routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_pass_thru_okay` field of the `scsi_ddsw` structure.

If a device is opened by a non-pass-through class driver and the driver specifies a `dd_pass_thru_okay()` entry point in its `scsi_ddsw` structure, the driver has complete control over what pass-through I/Os are allowed. If the driver does not specify a `dd_pass_thru_okay()` entry point, then pass-through I/Os are not allowed.

The `dd_pass_thru_okay()` function is called from within a critical section and may not block.

**Declaration**

```c
dd_pass_thru_okay (
    dev_t dev,
    struct sctl_io *sctl_io
);
```

**Parameters**

dev  The device number.

sctl_io  Struct containing ioctl information.

**Return Values**

`dd_pass_thru_okay()` is expected to return the following values:

PT_OKAY  Successful completion.

0  Error

### dd_read

The `dd_read()` routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_read` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, it is called instead of `physio()` by `scsi_read()`.

The `dd_read()` is called in a process context. It is not called from within a critical section; it may block.
Declaration

```c
int dd_read (
    dev_t dev,
    struct uio *uio
);
```

Parameters

- `dev` The device number.
- `uio` Structure containing transfer information.

Return Values

`dd_read()` is expected to return the following values:

- `0` Successful completion.
- `<>0` Error. The value is expected to be an `errno` value.

Example

```c
mydriver_dd_read(
    dev_t dev,
    struct uio *uio
)
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct sf_lun *llp = lp->dd_lun;
    int error;

    scsi_lun_lock(lp);
    while (llp->state & ST_GEOM_SEMAPHORE)
        scsi_sleep(lp, &llp->state, PRIBIO);
    llp->state |= ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);
    sf_update_geometry(dev);
    error = physio(scsi_strategy, NULL, dev, B_READ, minphys, uio);

    scsi_lun_lock(lp);
    llp->state &= -ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);
    wakeup(&llp->state);
    return error;
}
```

**dd_start**

The `dd_start()` routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_start` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, it is called by `scsi_start()` to allow the driver to perform any necessary processing prior to sending the I/O down the SCSI stack.

The `dd_start()` function is called in the process and interrupt context from within a critical section in `scsi_start();` `dd_start()` is not permitted to block.
The critical section in `scsi_start()` from where the `dd_start()` function is called, is protecting the `scsi_lun` structure. It is also guaranteeing that `lp->n_scbs` is consistent with the `dd_start()` function. The critical section also protects the incrementing of `n_scbs` in the `scsi_lun` structure and the incrementing of the SCSI subsystem unique I/O ID `scsi_io_cnt`.

If this routine does not exist, only “special” I/Os (B_SIOC_IO or B_SCSI_CMD) can be performed.

The driver’s `dd_start()` routine must dequeue the I/O from the appropriate list and perform whatever is necessary for the device to operate upon the I/O.

The parameters passed for this purpose are the `lp` and the `scb` parameters. The `scb` has the necessary `cdb[]` array for the SCSI command bytes.

**Declaration**

```c
struct buf *(*d_start) dd_start (  
    struct scsi_lun  *lp,  
    struct scb     *scb  
);
```

**Parameters**

- `lp` The open LUN structure.
- `scb` Extra state information for I/O.

**Return Values**

`dd_start()` is expected to return the following values:

- `struct buf *bp` Successful completion.
- `NULL` Error

**Example**

```c
#include <sys/scsi_ctl.h>
struct buf *mydriver_dd_start(  
    struct scsi_lun *lp,  
    struct scb  
)
{
    struct mydriver_lun *llp = lp->dd_lun;
    struct buf  *bp;
    struct scb     *head_scb, *scb_forw, *scb_back;
    int nblks, blkno, x;
    int lshift = llp->devb_lshift;

    /*
     * We could be more granular with locks, but
     * that would most likely cause too much
     * overhead getting/releasing locks.
     */
    scsi_lun_lock(lp);

    if (bp = mydriver_dequeue(lp)) == NULL)
        {  
            goto start_done;
        }

    nblks = bp->b_bcount >> llp->log2_blk_sz;
```
if (bp->b_offset & DEV_BMASK)
    blkno = (unsigned) bp->b_offset >> llp->log2_blk_sz;
else
    blkno = (unsigned) (lshift > 0
        ? bp->b_blkno << lshift
        : bp->b_blkno >> -lshift);

scb->cdb[0] = (bp->b_flags & B_READ)
    ? CMDread10
    : llp->state & LL_WWV
    ? CMDwriteNverify
    : CMDwrite10;
scb->cdb[1] = 0;
scb->cdb[2] = blkno >> 24;
scb->cdb[3] = blkno >> 16;
scb->cdb[4] = blkno >> 8;
scb->cdb[5] = blkno;
scb->cdb[6] = 0;
scb->cdb[7] = nblks >> 8;
scb->cdb[8] = nblks;
scb->cdb[9] = 0;

/* Immediate Reporting (WCE) ON? */
if (llp->state & LL_IR)
    if ((scb->cdb[0] == CMDwrite10) && (bp->b_flags & B_C))
        scb->cdb[1] |= WRITE_FUA_BIT;

if (lp->state & L_WOE && !(bp->b_flags & B_READ))
    { 
        if (lp->inquiry_data.inq2.dev_type == SCSI_MQ)
            scb->cdb[1] |= 0x04;
        else /* SONY */
            scb->cdb[9] |= 0x40;
    }

scb->cdb_len = 10;

scb->max_msecs = llp->mydriver_msecs;
scb->max_retries = 5;

scb->flags = SCB_4BYTE | SCB_ORDERED_TAG;
if (llp->state & LL_STINGRAY)
    scb->flags &= ~SCB_ORDERED_TAG;

/* Assume that scb->io_id will be set by caller within */
/* this CRIT */

/* Queue this bp into llp->active_bp_list HEAD for */
/* tracking */
if (llp->active_bp_list != NULL)
    { 
        scb->io_forw = llp->active_bp_list;
        head_scb = (void *) llp->active_bp_list->b_scb;
        scb->io_back = head_scb->io_back;

        scb_forw = (void *) scb->io_forw->b_scb;
        scb_back = (void *) scb->io_back->b_scb;
        scb_forw->io_back = bp;
        scb_back->io_forw = bp;

        llp->active_bp_list = bp;
    }
else {
    llp->active_bp_list = bp;
    scb->io_forw = scb->io_back = bp;
}

/* Although redundant with caller, set this in case */
bp->b_scb = (long) scb;

start_done:
    scsi_lun_unlock(lp);
    return bp;
}

**dd_strategy**

The dd_strategy() routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the dd_strategy field of the scsi_ddsw structure.

The dd_strategy() routine is called by scsi_strategy() to perform whatever sorting or queueing the class driver requires for normal I/O. For most drivers, enqueuing to lp->scb_q is necessary; the scsi_disk driver calls disksort_enqueue().

The dd_strategy() is called both in a process and interrupt context; it is not allowed to block.

If the driver invokes scsi_strategy(), dd_strategy() is required. If the dd_read() or dd_write() routines are not specified, SCSI Services will assume physio() is to be used.

---

**NOTE**

The scsi_strategy() calls dd_strategy() holding lun_lock.

**Declaration**

```c
int (*dd_strategy) dd_strategy (
    struct buf *bp,
    struct scsi_lun *lp
);```

**Parameters**

- `bp` transfer buf header.
- `lp` scsi LUN information.

**Return Values**

dd_strategy() is expected to return the following values:

- `0` Successful completion.
- `-1` Error

**Example**

The MP protection is provided for modification of the queues. Here is an example for a tape:

```c
mydriver_dd_strategy(struct buf *bp, struct scsi_lun *lp) {
    struct st_lun *llp = lp->dd_lun;
    struct st_static_lun *sllp = llp->static_data;
```
DB_ASSERT(!(bp->b_flags & B_ERROR));

sllp->head_pos &= ~HEAD_FORWARD;

/* Check for valid request size in fixed block mode */
if (llp->block_size > 0 && bp->b_bcount %
    llp->block_size != 0)
    {
        bp->b_flags |= B_ERROR;
        bp->b_error = ENXIO;
        biodone(bp);
        return -1;
    }

bp->b_resid = bp->b_bcount;
scsi_enqueue(&lp->scb_q, bp, TAIL);
return 0;

A SCSI disk (scsi_disk) driver does not use the lp->scb_q. Instead, dissort() a service from the File System, is used. The following is an example:

mydriver_dd_strategy(
    struct buf *bp,
    struct scsi_lun *lp
)
{
    dev_t          dev = bp->b_dev;
    struct mydriver_lun *llp = lp->dd_lun;

    ASSERT(!(bp->b_flags & B_ERROR));

    return mydriver_enqueue(lp, bp);
}

mydriver_enqueue(
    struct scsi_lun *lp
    struct buf *bp
{
    int x;
    struct mydriver_lun *llp = lp->dd_lun;
    struct buf *dp;

    /* Class driver’s private queue */
    dp = &llp->lun_disk_queue;

    /* set B_FIRST to get queue preference */
    if (bp->b_flags & B_SPECIAL)
        bp->b2_flags |= B2_FIRST;

    /* fake b_cylin 512K per cylinder */
    bp->b_cylin = (bp->b_offset >> 19);
    dissort_enqueue(dp, bp);

    return 0;
}

NOTE The dd_strategy() must exist (be defined as non-NULL in the scsi_ddsw structure) if the driver calls scsi_strategy().
**dd_write**

The `dd_write()` routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_write` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, it is called instead of `physio()` by `scsi_write()`.

This routine is called from `scsi_write()` in a process context. Since it is not called from within a critical section, it may block.

**Declaration**

```c
int dd_write (dev_t dev, struct uio *uio);
```

**Parameters**

- `dev` The device number.
- `uio` Structure containing transfer information.

**Return Values**

`dd_write()` is expected to return the following values:

- `0` Successful completion.
- `errno` Error

**Example**

```c
#include <sys/scsi_ctl.h>
#define ST_GEOM_SEMAPHORE 2

mydriver_dd_write(dev_t dev, struct uio *uio)
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct sf_lun *llp = lp->dd_lun;
    int error;

    scsi_lun_lock(lp);
    while (llp->state & ST_GEOM_SEMAPHORE)
        scsi_sleep(lp, &llp->state, PRIBIO);
    llp->state |= ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);
    sf_update_geometry(dev);
    error = physio(scsi_strategy, NULL, dev, B_WRITE, minphys, uio);

    scsi_lun_lock(lp);
    llp->state &= ~ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);
    wakeup(&llp->state);

    return error;
}
```
Synchronization in SCSI Drivers

When the SCSI Services calls the driver, it takes the appropriate locks to provide MP protection. The driver must provide protection for accessing its own private data and any data under the domain of the SCSI Services such as scsi_lun, scsi_tgt, scsi_bus, or the SCSI subsystem’s data. Locks are defined in <sys/scsi_ctl.h>.

Rules for Ordering Locks

The rules for ordering locks and semaphores help the kernel detect deadlocks in their use. When a thread of execution must hold more than one lock or semaphore, it must acquire them in increasing order. The order of locks and semaphores in the ascending order are as follows:

1. LUN lock
2. Target lock
3. Bus lock
4. Subsystem lock

If a thread of execution must hold both the LUN lock and target lock at the same time, the ordering rules assert that the code must acquire the LUN lock before it acquires the target lock. The class driver should normally not need any additional locks of its own. It should be able to use the LUN lock, for example, for protection of its own per-instance structures. However, if a driver uses a private lock, it must have the highest precedence. It must be acquired first, and must have a lower order number than the LUN lock, SCSI_LUN_LOCK_ORDER.

Subsystem Lock

The subsystem lock protects the SCSI subsystem’s global data. Only SCSI Services access this data, so the driver should have no need for this lock.

Bus Lock

Each scsi_bus structure has a lock associated with it that protects many of the fields in the structure. Most drivers do not need to use the bus lock because they ordinarily do not access the information maintained in the scsi_bus structure.

Some HP class drivers access the B_EXCLUSIVE flag in the state field of the scsi_bus structure.

Target Lock

Each scsi_tgt structure has a lock associated with it that protects some of the fields in the structure. Class drivers can access the open_cnt, sctl_open_cnt, state, and bus fields in this structure. Class drivers may only modify the state field, and must do so under the protection of the target lock. The target lock can also be used to prevent the open_cnt, sctl_open_cnt, or state field from being modified while other conditions are checked or actions are performed.

LUN Lock

Each scsi_lun structure has a lock associated with it that protects the fields in the structure and in the dd_lun private data area. See the following section on the LUN structure to see which fields class drivers can access and modify, and which locks protect those fields.
For the `driver_open()` routine, the class driver does not have any of the locks available until the kernel calls `scsi_lun_open()`, because `scsi_lun_open()` creates the `scsi_bus`, `scsi_tgt`, and `scsi_lun` structures.

For the `driver_close()` routine, the situation is similar. The locks are also available when the `dd_close()` routine is called. When `scsi_lun_close()` returns control to its caller, the locks are no longer available to the driver.

**Handling Concurrency for Tape Devices**

**I/O Access Concurrency**

Make sure that no two outstanding I/O's are on the same device. The driver needs to protect the device from multiple outstanding I/O's from the same initiator. This situation can arise from concurrent I/O's from the same driver or from the pass-thru driver while tape driver I/O is active. Use the `dd_pass_thru_okay` and `dd_pass_thru_done` interfaces to handle concurrent pass_thru and tape driver I/O.

**LUN Open Concurrency**

The driver sets the `L_DISABLE_OPENS` flag on the state field of the `scsi_lun` struct defined in `scsi_ctl.h` to prevent multiple opens on the same tape device.
SCSI Services Summary

SCSI Services are commonly used SCSI functions that allow device and interface drivers to be much smaller and more supportable. In addition to providing most commonly used SCSI functions, WSIO SCSI Services also provide a supported pass-through mechanism. (See scsi_ctl (7) in the HP-UX 11i v1 Driver Development Reference Guide for information on pass-through.)

SCSI Services are summarized in Table 13-3, “SCSI Services.” For more detailed information on these services see the HP-UX 11i v1 Driver Development Reference Guide.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_scsi_lun()</td>
<td>Returns scsi_lun pointer corresponding to the dev_t parameter passed in.</td>
</tr>
<tr>
<td>scsi_action()</td>
<td>Must ultimately be called after each I/O attempt completion, whether retrying the I/O or not, to log errors to the dmesg buffer, determine if the I/O is to be retried, or perform dynamic qdepth handling.</td>
</tr>
<tr>
<td>scsi_cmd(), scsi_cmdx()</td>
<td>For driver generated I/O requests, this routine creates and builds a sctl_io structure and a bp, attaches the sctl_io to the bp, forwards the bp to the scsi_strategy() routine, and cleans up when the I/O is completed.</td>
</tr>
<tr>
<td>scsi_ddsw_init()</td>
<td>Called from class driver’s driver_dev_init() routine. Causes initialization of blk_major and raw_major fields in the driver's switch table (ddsw).</td>
</tr>
<tr>
<td>scsi_dequeue()</td>
<td>Removes I/O requests from queues maintained by SCSI Services.</td>
</tr>
<tr>
<td>scsi_dequeue_bp()</td>
<td>Externally available to dequeue particular bp from circular list. Intended for use with volume manager's B_PFTIMEOUT.</td>
</tr>
<tr>
<td>scsi_enqueue()</td>
<td>Places I/O requests on queues maintained by SCSI Services.</td>
</tr>
<tr>
<td>scsi_init_inquiry_data()</td>
<td>Called from class driver’s ddsw-&gt;dd_open() routine. Performs the SCSI Inquiry request to the device to obtain the Inquiry data if not already done.</td>
</tr>
<tr>
<td>scsi_ioctl()</td>
<td>The ioctl commands that are supported by all drivers are implemented here to ensure consistency among drivers. If the specified ioctl command is not one of the common ones implemented in SCSI services, dd_ioctl is called if it is not NULL.</td>
</tr>
<tr>
<td>scsi_log_io()</td>
<td>Records the I/O attempt and its results in the dmesg buffer.</td>
</tr>
<tr>
<td>scsi_lun_open()</td>
<td>Called from class driver’s driver_open() routine. Performs necessary open operations, including the invocation of the calling driver’s ddsw-&gt;dd_open() routine.</td>
</tr>
<tr>
<td>scsi_read()</td>
<td>Synchronous read routine, which calls dd_read() if not NULL, or physio() if dd_read() is NULL.</td>
</tr>
</tbody>
</table>
### Table 13-3  
SCSI Services (Continued)

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scsi_sense_action()</code></td>
<td>Interprets sense data for SCSI, CCS, or SCSI-2 or SCSI-3 compliance. It requires that the inquiry data for the device has been initialized by <code>scsi_init_inquiry_data()</code> before it can interpret it.</td>
</tr>
<tr>
<td><code>scsi_snooze()</code></td>
<td>Performs a sleep without tying up the processor. Must not be called by a thread of execution that holds any lock and must only be called in a process context (not ICS). Currently, this routine is used only by <code>scsi_disk</code> driver to delay subsequent device access following inquiry to a particular model of Quantum disk drive.</td>
</tr>
<tr>
<td><code>scsi_strategy()</code></td>
<td>The first place in the I/O path that all I/O requests have in common. Its primary purpose is to enqueue the <code>bp</code> to await the necessary resources to allow the request to be sent to the interface driver, and thus, the hardware.</td>
</tr>
<tr>
<td><code>scsi_write()</code></td>
<td>Synchronous write routine, which calls <code>dd_writer()</code> if not NULL, or <code>physio()</code> if <code>dd_write</code> is NULL.</td>
</tr>
</tbody>
</table>