9 Writing SCSI Device Drivers
This chapter presents routines and conceptual material specifically for drivers of SCSI devices. Chapter 5, “Writing a Driver,” describes the general configuration and entry-point driver routines, such as \texttt{driver\_open} and \texttt{driver\_write}. If you are writing a SCSI driver, you must provide routines from both Chapter 5, “Writing a Driver,” and this chapter.

The \textit{HP-UX Driver Development Reference} describes the SCSI Services routines.

SCSI devices can be controlled in two ways, both supported by the SCSI Services routines. Kernel drivers, following the \texttt{scsi\_disk} model, are the traditional method. They are described in this chapter and in \texttt{scsi\_disk(7)}. However, many SCSI devices do not need a special driver. Instead, user-programs pass ioctl commands to the pass-through driver, \texttt{scsi\_ctl}. The pass-through driver is described in \texttt{scsi\_ctl(7)}.

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

- “SCSI Driver Development, Step 1: Include Header Files”
- “SCSI Driver Development, Step 2: Set Up Structures”
- “SCSI Driver Development, Step 3: Create the \texttt{driver\_install} Routine”
- “SCSI Driver Development, Step 4: Create the \texttt{driver\_dev\_init()} Routine”
- “SCSI Driver Development, Step 5: Analyze Multiprocessor Implications”
- “SCSI Driver Development, Step 6: Create the Entry-Point Routines”
- “SCSI Driver Development, Step 7: Error Handling”
- “SCSI Driver Development, Step 8: Underlying Routines”

The examples in this chapter assume that the name of your driver is \textit{mydriver} and that you are following the routine-naming conventions described in “Step 1: Choosing a Driver Name” on page 82, Chapter 5.
SCSI Driver Development, Step 1: Include Header Files

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

**NOTE**

Including header files that your driver does not need 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/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, your 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.
SCSI Driver Development, Step 2: Set Up Structures

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

---

**NOTE**

Whether the driver is to operate on an MP platform or not, SCSI Services makes use of the locking facilities, and all drivers using SCSI Services must use the provided data-protection routines. It is essential that you include 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. See “SCSI Driver Development, 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 “Step 3: Defining Installation Structures” on page 85, Chapter 5, for information about these data structures.)

First, declare your 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 “The `drv_ops_t` Structure Type” on page 85, Chapter 5, for further details.
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,
    NULL,
    C_ALLCLOSES | C_MGR_IS_MP
};

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 “The drv_info_t Structure Type” on page 88, Chapter 5, for further details.

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

The wsio_drv_data_t structure specifies additional information for the WSIO CĐIÖ. The first field should be scsi_disk for SCSI device drivers and scsi for SCSI interface drivers. See “The wsio_drv_data_t Structure Type” on page 90, Chapter 5, for further details.
static wsio_drv_data_t mydriver_data =
{
    "scsi_disk",
    TDEVICE,
    DRV_CONVERGED,
    NULL,
    NULL,
};

The wsio_drv_info_t structure ties the preceding three together. See “The wsio_drv_info_t Structure Type” on page 91, Chapter 5, for further details.

static wsio_drv_info_t mydriver_wsio_info =
{
    &mydriver_info,
    &mydriver_ops,
    &mydriver_data,
};
SCSI Driver Development, Step 3: Create the driver_install Routine

The `driver_install` routine causes the information that you created above to be installed into the I/O subsystem, specifically into the WSIO CDIO.

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

int mydriver_install()
{
    extern int (*dev_init)();

    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));
}
```
SCSI Driver Development, Step 4:
Create the driver_dev_init() Routine

You 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()
{
    dev_t dev = NODEV;
    /*
    * 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)

In order 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. Therefore, you should have the standard driver routines restrict their operation to calling the appropriate SCSI Services routine, as shown in the examples in “SCSI Driver Development, 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 “SCSI Services Summary”. For more detailed information, see the HP-UX Driver Development Reference.

The scsi_ddsw structure is defined as follows in the header file <sys/scsi_ctl.h>:
Writing SCSI Device Drivers

SCSI Driver Development, Step 4: Create the driver_dev_init() Routine

```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 below.

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

- **raw_major**
  - The number of bytes to be allocated and attached to the open device tree when `driver_open()` is first executed.

- **dd_lun_size**
  - 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_open()**
- **dd_close()**
- **dd_strategy()**
- **dd_read()**
- **dd_write()**
- **dd_ioctl()**
- **dd_start()**
- **dd_done()**
- **dd_pass_thru_okay()**
- **dd_pass_thru_done()**
- **dd_ioctl_okay()**
- **dd_status_list**
- **dd_status_cnt**
- **dd_flags**
- **wsio_drv_info_t**
Writing SCSI Device Drivers

SCSI Driver Development, Step 4: Create the driver_dev_init() Routine

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_flags
    Flag bits, currently only DD_DDG defined.

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

The first example is the declaration of your driver's version of the dd routines that can be called by SCSI Services. The routine names are arbitrary. The names in comments are the field names of the scsi_ddsw structure.

```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 */
```

The following example shows the scsi_ddsw structure. Specify NULL for routines that are not defined (that is, that you are 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 you defined in “SCSI Driver Development, 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 */
};
```
Writing SCSI Device Drivers

SCSI Driver Development, Step 4: Create the driver_dev_init() Routine

```c
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 */
```
SCSI Driver Development, Step 5: Analyze Multiprocessor Implications

You need to make your device driver MP safe, regardless of whether it is to operate on 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 data-protection routines the kernel provides.

Your 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, 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. (See “Data Protection for SCSI Drivers” for more details.)
- Build a kernel with your driver.
- Test your driver on a single processor (UP) system with a debug kernel if available. (You can also test it on an MP system.)
SCSI Driver Development, Step 6: Create the Entry-Point Routines

For many of the entry points, SCSI Services perform much of the work. If you use physio(), scsi_strategy() will be called by your driver's driver_strategy routine. Hence, you need not create the underlying dds->dd_read() and dds->dd_write() routines. However, if your driver calls scsi_strategy(), you must specify a dds->dd_strategy() routine.

The scsi_strategy() routine cannot block because it can be called on the Interrupt Control Stack (ICS) by a bp->b_call routine.

driver_open() Routine

mydriver_open(dev, oflags)
dev_t dev;
int oflags;
{
    return (scsi_lun_open(dev, &mydriver_ddsw, oflags));
}

driver_close() Routine

mydriver_close(dev)
dev_t dev;
{
    return scsi_lun_close(dev);
}

driver_read() Routine

mydriver_read(dev, uio)
dev_t dev;
struct uio *uio;
{
    return scsi_read(dev, uio);
}
**driver_write() Routine**

```c
mydriver_write(dev, uio)
dev_t dev;
struct uio *uio;
{
    return scsi_write(dev, uio);
}
```

**driver_strategy() Routine**

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

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

**driver_psize() Routine**

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.

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

    nblks = llp->nblks;
    rshift = llp->devb_lshift;

    size = rshift > 0 ? nblks >> rshift : nblks << -rshift;

    return size;
}
```
**driver_ioctl() Routine**

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

You can specify one optional list in the driver's scsi_ddsw: dd_status_list[]. SCSI services access this optional list when an I/O completion occurs on the driver's SCSI LUN. The SCSI Services internal routine scsi_status_action() determines what to do based upon this list.

The following are examples of very simple lists:

```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, 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 }
};
```

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

This is where the driver can be as complex as you desire, 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 9-1, "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 Driver Development Reference*. 
**dd_close Routine**

The `dd_close()` SCSI function, used to provide driver-specific processing during close is provided by the driver writer. It can have any unique name. You 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.

**Conditions**

`dd_close()` is called from `scsi_lun_close()` in a process context. The open/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);
dev_t dev;
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    if (dd_blk_open_cnt(lp) == 1) {
        scsi_lun_lock(lp);
        llp->state &= ~ST_GEOM_LOCKED;
        scsi_lun_unlock(lp);
    }
}
```
dd_ioctl Routine

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

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

Examine the ioctl commands provided by SCSI Services in scsi_ioctl(), and implement any additional commands needed in your dd_ioctl() routine.

It is in dd_ioctl() and in dd_open(), if implemented, that some of the more specialized features of SCSI Services may be useful, as listed below.

- scsi_cmd()
- scsi_init_inquiry_data()
- scsi_mode_sense()
- scsi_mode_fix()
- scsi_mode_select()
- scsi_wr_protect()

The following is a summary of IOCTLs that the LVM uses to interface with the device driver.

Table 9-1 IOCTL Usage by LVM Summary

<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. For example, DIOC_SET_PFTIMEOUT must be repeated on each bus path for I/O requests.</td>
</tr>
<tr>
<td>DIOC_SET_PFTIMEOUT</td>
<td></td>
</tr>
<tr>
<td>SIOC_RESET_BUS</td>
<td></td>
</tr>
</tbody>
</table>

Conditions

`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

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

Return Values

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

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

Table 9-1 IOCTL Usage by LVM Summary (Continued)

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIOC_DESCRIPT</td>
<td>The LVM uses each of these ioctl's for device accessibility, device type, block size, and control of immediate reporting.</td>
</tr>
<tr>
<td>DIOC_BLKLIST_REMAP</td>
<td></td>
</tr>
<tr>
<td>SIOC_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>
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;
    disk_describe_type *ddt;
    int size = (cmd & IOCSIZE_MASK) >> 16;
    int i;

    switch (cmd & IOCCMD_MASK)
    {
    case DIOC_DESCRIBE & IOCCMD_MASK:
        if (cmd != DIOC_DESCRIBE & size != 32)
            return EINVAL;
        ddt = (void *) data;
        i = inq->dev_type;
        bcopy(inq->product_id, ddt->model_num, 16);
        ddt->intf_type = SCSI_INTF;
        ddt->maxsva = llp->nblks - 1;
        ddt->lgblksz = llp->blk_sz;
        ddt->dev_type = i == SCSI_DIRECT_ACCESS ? DISK_DEV_TYPE
            : i == SCSI_WORM ? WORM_DEV_TYPE
            : i == SCSI_CDROM ? CDROM_DEV_TYPE
            : i == SCSI_MO ? MO_DEV_TYPE
            : UNKNOWN_DEV_TYPE;
        if (HP_MO(lp))
            /* Shark lies; fix it to match Series800 */
        ddt->dev_type = MO_DEV_TYPE;
        if (size == 32)
            return 0;
        /* WRITE_PROTECT for SCSI WORM */
        ddt->flags = (llp->state & LL_WP) ?
            WRITE_PROTECT_FLAG : 0;
        return 0;
    ```
switch (cmd)
{
    case SIOC_CAPACITY:
        {struct capacity *) data)->lba = llp->nblks;
         (struct capacity *) data)->blksz = llp->blk_sz;
        return 0;
    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);
    case SIOC_SYNC_CACHE:
        if (llp->state & LL_IR)
            return mydriver_sync_cache(dev);
        else
            return 0; /* IR not on, just return */
    case DIOC_CAPACITY:
        *data = (llp->devb_lshift > 0 ? llp->nblks >>
            llp->devb_lshift
            : llp->nblks << -(llp->devb_lshift));
        return 0;
    ...
    default:
        return EINVAL;
}

**dd_ioctl_okay Routine**

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

dd_ioctl_okay() disallows all ioctl commands through the pass-through driver that are not explicitly allowed by any nonpass-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

It is desirable to allow `SIOC_INQUIRY` for the pass-through driver at all times. Therefore, `SIOC_INQUIRY` is allowed by default (if there is no `dd_ioctl_okay()` routine). `SIOC_INQUIRY` is also always allowed if it will not result in I/O (`lp->inquiry_sz > 0`), because it does not affect the nonpass-through device 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 Routine

The dd_open() SCSI function is provided by the driver writer. It can have any unique name. You 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 disk driver’s 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 you need 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.

Conditions

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

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>&lt;&gt;0</td>
<td>Error. The value is expected to be an errno value.</td>
</tr>
</tbody>
</table>

**Examples**

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

mydriver_dd_open(dev, oflags)
dev_t dev;
int 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[12];
    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.
    */
    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
```
SCSI Driver Development, Step 8: Underlying Routines

* 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->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 Routine**

The `dd_pass_thru_done()` routine is provided by the driver writer. It can have any unique name. You 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 device 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 Routine**

The `dd_pass_thru_okay()` routine is provided by the driver writer. It can have any unique name. You 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 nonpass-through device driver and the driver specifies a `dd_pass_thru_okay()` entry point in its `scsi_ddsw` structure, then 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

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.

Example

#include <sys/scsi_ctl.h>

mydriver_dd_pass_thru_okay(dev, sctl_io)  
dev_t dev;  
struct sctl_io *sctl_io;  
{  
   return PT_OKAY;  
}

dd_read Routine

The dd_read() routine is provided by the driver writer. It can have any  
unique name. You 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().

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, uio)
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);
    lp->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 Routine**

The `dd_start()` routine is provided by the driver writer. It can have any unique name. You 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 calling `scsi_start_nexus()`.

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 mainly protecting the `scsi_lun` structure and guaranteeing that `lp->n_scbs` is consistent with the `dd_start()` function starting a request or not. The critical section also protects the incrementing of `n_scbs` in the `scsi_tgt` 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
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/scsiCtl.h>

struct buf *mydriver_dd_start(lp, scb)
struct scsi_lun *lp;
struct scb *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_MO)
        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_STRINGRAY)
    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;
Writing SCSI Device Drivers

SCSI Device Development, Step 8: Underlying Routines

```c
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
* completion int */
bp->b_scb = (long) scb;
```

```c
start_done:
    scsi_lun_unlock(lp);
    return bp;
}
```

**dd_strategy Routine**

The `dd_strategy()` routine is provided by the driver writer. It can have any unique name. You 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 device driver requires for normal I/O. For most drivers, enqueuing to `lp->scb_q` is necessary; the `scsi_disk()` driver calls `disksort_enqueue()`.

---

Chapter 9 429
dd_strategy() is called in a process (and possibly, 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

scsi_strategy() calls dd_strategy() holding lun_lock.

Declaration

    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:

    mydriver_dd_strategy(bp)
    struct buf *bp;
    {
        struct scsi_lun *lp = m_scsi_lun(bp->b_dev);
        struct st_lun *llp = lp->dd_lun;
        struct st_static_lun *sllp = llp->static_data;
        DB_ASSERT(!(bp->b_flags & B_ERROR));
A SCSI disk does not use the `lp->scb_q`. Instead, a service from the File System is used, `disksort()`. The following is an example of its use:

```c
def_t dev = bp->b_dev;
struct scsi_lun *lp = m_scsi_lun(dev);
struct mydriver_lun *llp = lp->dd_lun;
ASSERT(!(bp->b_flags & B_ERROR));
if (bpcheck(bp, llp->nblks, llp->log2_blk_sz, 0))
    return -1;
LOG(bp->b_dev, FUNC_QUEUE, bp->b_blkno, "b_blkno");
LOG(bp->b_dev, FUNC_QUEUE, bp->b_offset, "b_offset");
LOG(bp->b_dev, FUNC_QUEUE, bp->b_bcount, "b_bcount");
return mydriver_enqueue(lp, bp);
```
Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

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

    dp = &llp->lun_disk_queue;

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

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

    /* Increment counters within this protection */
    scsi_enqueue_count(lp, bp);

    return 0;
}
```

**Warning**

`dd_strategy()` must exist (be defined as non-NULL in the `scsi_ddsw` structure) if your driver calls `scsi_strategy()`.

### dd_write Routine

The `dd_write()` routine is provided by the driver writer. It can have any unique name. You 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

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

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

mydriver_dd_write(dev, uio)
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, ui o);

    scsi_lun_lock(lp);
    llp->state &= ~ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);
    wakeup(&llp->state);
return error;
};
Data Protection for SCSI Drivers

The SCSI Services your driver calls take the appropriate locks to provide MP protection. One thing your driver must provide is 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 is, in ascending order:

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 spinlocks that are used to implement the LUN, target, bus, and subsystem locks are the normal HP-UX spinlocks.

While a thread of execution holds a lock, the processor's interrupt level is set to SPL6, preventing I/O devices from interrupting that processor. The spinlock associated with `spl*()` services (`spl_lock`) is of lower order than practically all other locks, so code protected by a spinlock cannot call a `spl*()` routine.

Subsystem Lock

The subsystem lock protects the SCSI subsystem's global data. Only SCSI Services access this data, so your 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.

You should be aware that some HP device 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. Device drivers can access the open_cnt, sctl_open_cnt, state, and bus fields in this structure. Device 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 device drivers can access and modify, and which locks protect those fields.

For the driver_open() routine, the device driver does not have any of the locks available until after 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 your driver.
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 provides a supported pass-through mechanism. (See `scsi_ctl(7)` in the *HP-UX Reference* for information on pass-through.)

SCSI Services are summarized in Table 9-2, “SCSI Services.” For more detailed information on these services see the *HP-UX Driver Development Reference*.

Table 9-2  
SCSI Services

<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>disksort_enqueue()</td>
<td>Places I/O requests on queues maintained by SCSI Services.</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 LVM's B_PFTIMEOUT.</td>
</tr>
<tr>
<td>scsi_ddsw_init()</td>
<td>Called from device 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_lun_open()</td>
<td>Called from device driver's driver_dev_init() routine. Performs necessary open operations, including the invocation of the calling driver's ddsw-&gt;dd_open() routine.</td>
</tr>
</tbody>
</table>
**SCSI Services**

Table 9-2  
**SCSI Services (Continued)**

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_init_inquiry_data()</td>
<td>Called from device driver's ddsw-&gt;dd_open() routine. Performs first SCSI Inquiry request to the device.</td>
</tr>
<tr>
<td>scsi_strategy()</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 bp to await the necessary resources to allow the request to be sent to the interface driver, and thus, the hardware.</td>
</tr>
<tr>
<td>scsi_read()</td>
<td>Synchronous read routine, which calls physio().</td>
</tr>
<tr>
<td>scsi_write()</td>
<td>Synchronous write routine, which calls physio().</td>
</tr>
<tr>
<td>scsi_ioctl()</td>
<td>Ioclt commands that are supported by all drivers are implemented here to ensure consistency among drivers.</td>
</tr>
<tr>
<td>scsi_cmd(), scsi_cmdx()</td>
<td>For driver-generated I/O requests. It creates and builds a sctl_io 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_action()</td>
<td>Must ultimately be called after each I/O attempt completion (as in a retry situation). It may log errors to the dmesg buffer, retry the I/O, or disable tags.</td>
</tr>
</tbody>
</table>
### Table 9-2  
**SCSI Services (Continued)**

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_sense_action()</td>
<td>Interprets sense data for SCSI, CCS, or SCSI-2 compliance. It requires that the inquiry data for the device has been initialized by scsi_init_inquiry_data() before it can interpret it.</td>
</tr>
<tr>
<td>scsi_snooze()</td>
<td>Performs a sleep without tying up the processor. Must not be called by a thread of execution that holds any lock. Currently, this routine is used only by scsi_disk to delay subsequent device access following Inquiry to a particular model of Quantum disk drive.</td>
</tr>
<tr>
<td>scsi_log_io()</td>
<td>Records the I/O attempt and its results in the dmesg buffer.</td>
</tr>
</tbody>
</table>