8 Writing a SCSI Interface Driver
This chapter provides information on designing and developing a SCSI interface driver, also known as a Host Bus Adapter driver.

The next section of this chapter describes data structures and interfaces provided by SCSI services and WSIO to an interface driver. Included is the flow of an I/O request as it passes different layers of the SCSI subsystem.

Later in the chapter, steps involved in the interface driver development are detailed. This includes driver installation and initialization in WSIO CDIO, SCSI subsystem specific driver initialization, DMA mapping utilities, and interrupt handling.

While explaining the interface driver development, code snippets from HP-UX sample interface driver for Qlogic's ISP12160 SCSI Ultra3 interface card are provided as an example.
**Overview of HP-UX SCSI Subsystem**

A SCSI interface driver is part of the SCSI subsystem in HP-UX. This is also called a SCSI Host Bus Adapter (HBA) driver in the industry.

The following figure is the Mass Storage Stack in HP-UX. It illustrates the components the SCSI subsystem interacts with.

### Figure 8-1 Mass Storage Stack

<table>
<thead>
<tr>
<th>Customer Applications / I/O Commands</th>
<th>S/W</th>
<th>H/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVM/VxVM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HFS/VxFS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raw I/O Access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCSI Subsystem</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SCSI Peripheral Device Drivers</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCSI Disk Driver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sdidk)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tape, Auto-Chgr Pass-Thru Driver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Driver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCSI Services / SCSI Pass-Thru (sctl)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCSI Interface Driver (HBA Driver)</td>
</tr>
<tr>
<td>HBA Card</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage Devices/Other SCSI Peripheral Devices</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A SCSI pass-through driver (also referred to as pass-through driver) is a character mode device driver. It is provided for customers wishing to integrate special purpose SCSI peripherals without having to worry about the complexities of writing a kernel driver. The SCSI subsystem also uses a pass-through driver interface to discover devices off a SCSI bus during “ioscan”. No special processing is needed at the interface drivers to handle pass-through I/Os. More discussion on pass-through drivers is beyond the scope of this paper. It is mentioned here only to give a complete picture of the SCSI subsystem.

The SCSI services layer is provided to simplify the jobs of, and impose a common structure on, device drivers and interface drivers. SCSI services layer is composed of data structures and functions. All of the functionality which is consistent for all device drivers and/or all interface drivers is implemented by SCSI services. SCSI services also rely on the data specified and set by device and interface drivers on its functionality.

WSIO CDIO is Device Driver Environment (DDE) in which an interface driver is developed. Refer to Chapters 2, 3 and 5 of the DDG for detailed discussion on the WSIO and how a driver fits into the WSIO CDIO.

An interface driver is responsible for managing the SCSI bus hardware. It takes a fully specified SCSI request and manipulates the hardware (including any necessary DMA operations) to make it occur on the SCSI bus. The interface driver is responsible for managing the interconnect between the initiator and targets, I/O timeouts, and other task management functions such as bus reset, abort task, and bus device resets. When re-selection is not completely implemented in the HBA, the interface driver is also responsible for managing the data structures necessary for nexus re-establishment on re-selection.
External Interfaces to an Interface Driver

A SCSI interface driver uses services of various subsystems of HP-UX. This section describes the interfaces a SCSI interface driver interacts with.

WSIO CDIO DDE

Refer to Writing a Driver in chapter 5 of the DDG for an understanding of the data structures and WSIO services. The SCSI subsystem specific driver initialization is discussed later in this chapter.

SCSI Services Interface

SCSI Services Interface includes data structures and functions which are provided by the SCSI subsystem to ease the development of an interface driver. Since SCSI services interact with both the device driver and the interface driver, this chapter discusses only such SCSI services which are visible to an interface driver.

Functions

Of the specified functions, some are provided by the interface driver (driver entry points) for use by services and the others are provided by services (service calls) for use by the interface driver.

Interface Driver Service Entry Points The following is a description of the prototypes and usage for each.

1. Interface Driver Open

```c
int if_open(dev_t dev)
```

The SCSI subsystem allows, but does not require, the interface driver to specify a logical unit open function. On all logical opens, the SCSI subsystem checks the if_open field of the scsi_ifsw structure for the SCSI bus. If if_open field is not NULL, the SCSI subsystem calls it with the device number of the device being opened as its sole argument.

It is never called under interrupt context and is allowed to sleep.
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

The SCSI subsystem provides protection that blocks all other opens and closes to the same logical unit until it returns.

2. Interface Driver Close

```c
void
if_close(dev_t dev)
```

The SCSI subsystem allows, but does not require, the interface driver to specify a logical unit close function. On all logical unit closes, the SCSI subsystem checks the `if_close` field of the `scsi_ifsw` structure for the SCSI bus. If `if_close` field is not NULL, the SCSI subsystem calls it with the device number of the device being closed as its sole argument.

It is never called under interrupt context and is allowed to sleep.

The SCSI subsystem provides protection that blocks all other opens and closes to the same logical unit until it returns.

3. Interface Driver Start

```c
void
if_start(struct isc_table_type * isc)
```

The SCSI subsystem requires the interface driver to specify a start function. Using this entry point, the SCSI subsystem informs the interface driver that it has work to do.

Since `if_start` can be called on the interrupt control stack (ICS) it is not permitted to sleep under any circumstances.

The SCSI subsystem may call `if_start` at any time; for example, when the bus is dormant or not and in a process’ context, or under interrupt. In all cases, the interface driver must continue to execute I/Os that are on the select queue until the bus becomes dormant.

A bus is considered dormant if it has no active I/Os. An I/O is considered to be active from the time it is enqueued on the select queue until `scsi_cbfn` is called for the I/O.

4. Interface Driver Reset Bus

```c
int
if_reset_bus(dev_t dev)
```
The SCSI subsystem allows, but does not require, the interface
driver to specify a bus reset function. When the SCSI subsystem
wants to reset a bus, it checks the if_reset_bus field of the
scsi_ifsw structure for the bus. If if_reset_bus is not NULL, it is
called with a device number identifying the bus as its sole argument.
When if_reset_bus returns, the SCSI bus should have been reset.

I/Os that are disconnected and the I/Os that are connected with the
bus (if any) at the time of the reset should be returned to the SCSI
subsystem with the appropriate status field set to SCTL_INCOMPLETE.
That is, if it was the Request Sense resulting from a check condition
that was terminated by the reset, then scb->sense_action should
be set to SCTL_INCOMPLETE. Otherwise, scb->cdb_status should
be set to SCTL_INCOMPLETE. “struct scb” is described under data
structures later in this section.

The SCSI subsystem makes this call only in response to
SIOC_RESET_BUS ioctl request.

5. Interface Driver Bus Device Reset (BDR)

   int
   if_bdr(dev_t dev)

   The SCSI subsystem allows, but does not require, the interface
driver to specify a BDR function.

   It is intended to serve as a way for the SCSI subsystem to direct the
interface driver to send a SCSI BDR message to the indicated target.

   The SCSI subsystem makes this call only in response to
SIOC_RESET_DEV ioctl request.

6. Interface Driver Abort

   int
   if_abort(dev_t dev)

   The SCSI subsystem allows, but does not require, the interface
driver to specify an abort function.

   It is intended to serve as a way for the SCSI subsystem to direct the
interface driver to send a SCSI ABORT message to the indicated
logical unit.

   The SCSI subsystem makes this call only in response to SIOC_ABORT
ioctl request.
SCSI Service Interface Driver Service Calls  The following is a description of the prototype and usage of each.

1. SCSI Subsystem Callback

   ```c
   void
   scsi_cbfn(struct buf * bp)
   ```

   When the interface driver finishes with an I/O, it returns the I/O to the SCSI subsystem by calling `scsi_cbfn` with the `bp` as its sole argument.

   The interface driver relinquishes all rights to access `bp`, `scb` and `*scb->if_scb` once it calls `scsi_cbfn()`. Of course, the `bp` may be reused later for another I/O, and similarly for the `scb` and `*scb->if_scb`, although they will not necessarily be related in subsequent I/Os.

   If the interface driver has attached a sense buffer to `scb->sense_data`, the `sense_data` buffer must be valid till `scsi_cbfn()` returns. The interface driver is forbidden from accessing it until `scsi_cbfn()` returns. It is important to note that the allocation and management of this buffer for holding `sense_data` is the responsibility of the interface driver.

   This can be called either in process or interrupt context. This must not be called with any locks held since the SCSI services may call the interface driver’s start entry point before it returns.

2. SCSI Subsystem Queueing Functions

   The SCSI subsystem provides three simple routines for managing queues or lists of `bp`'s. They may be used by the interface driver for managing private queues, but they must be used to manage the select queue from which the interface driver gets `bp`'s for execution. It is important to note that the driver has to acquire and release a SCSI bus lock before and after calling one of the queue routines.

   ```c
   #define TAIL 0
   #define HEAD 1
   ```

   ```c
   void scsi_enqueue(struct buf ** qp, struct buf * bp,
                    int where)
   ```
scsi_enqueue simply enqueues bp at the HEAD or TAIL of a circular list; qp is a pointer to the list header which is a pointer to the head of the list. If “where” is HEAD, the bp is inserted ahead of the list, otherwise it is added to the tail of the list.

This must be called with scsi_bus lock held.

struct buf *scsi_enqueue(struct buf ** qp, int where)

scsi_enqueue simply dequeues the bp at HEAD or TAIL of the list *qp based on the value of where and returns the bp. This returns NULL when the queue is empty.

This must be called with scsi_bus lock held.

struct buf *scsi_dequeue(struct buf ** qp, int where)

scsi_dequeue_bp tries to dequeue bp from wherever it may be in the queue *qp. Returns bp when found on the queue. Returns NULL when not found on the queue.

This must be called with scsi_bus lock held.

3. Open Device Tree Access Functions

Functions for acquiring pointers to the open device tree data structures from a device number are provided by services.

struct scsi_bus *m_scsi_bus(dev_t dev)
m_scsi_bus evaluates to a pointer to the scsi_bus structure for dev.

struct scsi_tgt *m_scsi_tgt(dev_t dev)
m_scsi_tgt evaluates to a pointer to the scsi_tgt structure for dev.

struct scsi_lun *m_scsi_lun(dev_t dev)
m_scsi_lun evaluates to a pointer to the scsi_lun structure for dev.

struct isc_table_type *m_scsi_isc(dev_t dev)
m_scsi_isc evaluates to a pointer to the isc_table_type structure for dev.
4. Micro Functions

A few macro functions are provided by the SCSI subsystem for the convenience of the interface driver.

\texttt{m\_bus\_id(dev\_t dev)}

\texttt{m\_bus\_id} evaluates to the bus ID of the SCSI bus of dev. The SCSI bus ID is also same as the card’s instance number.

\texttt{m\_tgt\_id(dev\_t dev)}

\texttt{m\_tgt\_id} evaluates to the target ID, i.e. SCSI bus address of dev.

\texttt{m\_lun\_id(dev\_t dev)}

\texttt{m\_lun\_id} evaluates to the logical unit number of dev.

Data Structures

In addition to the functions, SCSI subsystem also provides data structures to pass data across different layers of the SCSI stack. Some WSIO data structures are also used to pass data across the SCSI subsystem. Of the specified data structures and fields, some are owned by the interface driver i.e., maintained only by the interface driver (after being initialized to zero by services). These exist because some part of the SCSI subsystem other than the interface driver needs access to the information, and source of this information is the interface driver. It could be services or a device driver that needs access to the information, but that is irrelevant to the interface driver.

Other specified data structures and fields are owned by some other part of the SCSI subsystem and are available for use by the interface driver. Those remaining are owned by neither the interface driver nor some other part of the SCSI subsystem, but may be accessed and modified by either.

The bp, scb, \texttt{\*scb->if\_scb} and the data buffer for an I/O are available for use by the interface driver only while the I/O is active.

In this section data structures are present in C syntax and fields that are relevant to the current discussion only are shown.

SCSI Control Block  The buf structure is not large enough to hold all state information associated with a SCSI I/O attempt. An scb is attached to a buf by SCSI services to hold the temporary state information until the I/O is completed. The bp/scb association does persist for retries. “struct buf” is described later in the section.
Some of the fields that are of interest to an interface driver writer are explained below:

```c
struct scb {
    void     *if_scb;
    struct   scsi_lun *lp;
    ubit32   flags;
    ubit32   max_msecs;
    ubit8    cdb[SCSI_MAX_CDB_LEN];
    ubit8    cdb_len;
    ubit32   io_id;
    ubit8    tag;
    ubit32   cdb_status;
    ubit32   data_resid;
    ubit32   sense_status;
    ubit8    sense_bytes;
    ubit8    *sense_data;
} *scb;
```

```c
#define SCB_SDTR    SCTL_INIT_SDTR
#define SCB_WDTR    SCTL_INIT_WDTR
#define SCB_4BYTE   SCTL_4BYTE
#define SCB_2BYTE   SCTL_2BYTE
```

**NOTE**

Interface driver is not allowed to set the following fields:

- lp
- flags
- max_msecs
- cdb
- cdb_len
- io_id
- tag

**NOTE**

If a field is already described earlier, it will only be mentioned here. For more details on such a field, please refer to earlier sections.
scb->if_scb  Is a pointer to ifsw->if_scb_size bytes allocated by SCSI services and reserved for use by the interface driver. The pointer is initialized at scb creation time by services and the data area is bzero'ed by services for each I/O attempt prior to putting the I/O on the select queue. It is not touched by services at any other time. The if_scb area is later freed by the SCSI services along with the scb.

scb->lp  Is a pointer to the scsi_lun structure in the open device tree with which this scb is associated. If the scb belongs to a per-lun pool of scb's as opposed to a per-bus pool, then scb->lp is initialized at scb creation time by SCSI services and never changed. Otherwise, the scb belongs to a per-bus pool, and scp->lp is only valid while the scb is associated with a bp. Then scb->lp points to the scsi_lun structure associated with bp->b_dev.

scb->flags  The interface driver may check these bits in the flag for proper functionality. Bits in scb->flags that are relevant to an interface driver are:

- **SCB_NO_DISC**  This bit indicates that the disconnect privilege should not be granted in the identify message.

- **SCB_SDTR**  If this bit is set and SCB_WDTR is not set, the interface driver should initiate SDTR negotiation immediately following the Selection, Identify or tag message, whichever comes last, and before sending the CDB for the I/O.

- **SCB_WDTR**  This bit directs the interface driver that a wide negotiation should be initiated immediately following the Selection, Identify or tag message, whichever comes last, and before sending any CDB for the I/O. If (tp->state & T_ENABLE_SDTR) or (scb->flags & SCB_SDTR) is also set, the interface driver should initiate
SDTR negotiation immediately following the WDTR negotiation. The wide negotiation should always precede the synchronous negotiation, since a wide negotiation resets the link to asynchronous.

**SCB_4BYTE**

This bit is a hint to the interface driver that the target will never change phase while in data phase on other than a 4-byte boundary at the beginning of the data transfer without subsequently restoring the data pointer (implicitly or explicitly) to a previously aligned value and re-transferring data up to and beyond the point of disconnection to an aligned boundary. The phase change at the end of the I/O need be considered only if the amount of data transferred may be less than that requested in `bp->b_bcount`. Note that **SCB_4BYTE** does not imply that `bp->b_count` is a multiple of four or that `bp->b_un.b_addr` is 4-byte aligned. Note also that the phase change out of data phase if all `bp->b_bcount` bytes have been transferred is not subject to the alignment restructuring.

**SCB_2BYTE**

This bit is the same as **SCB_4BYTE** except that phase changes are only restricted to even boundaries.

**SCB_ORDERED_TAG**

Denotes that ordered tags are intended to be used for this device.
scb->max_msecs Minimum number of milliseconds the interface driver is to allow for this I/O from the time of Selection until Command Complete in parallel SCSI or as close to that as possible within a given HBA architecture. If scb->max_msecs milliseconds elapses and the I/O has not completed, the interface driver is encouraged to abort the I/O with Abort or Abort Tag as appropriate. The interface driver can run a timer routine periodically to watch out for the I/Os that are timed out. A value of zero indicates the interface driver should never abort this I/O based solely on the amount of time since Selection.

scb->cdb Holds the SCSI command bytes for this I/O.

scb->cdb_len The number of bytes in the cdb. This can be a maximum of SCSI_MAX_CDB_LEN.

scb->io_id Is a unique identifier for a SCSI I/O. It is initialized when the scb is associated with a bp and is unique across all SCSI busses.

scb->tag The tag value allocated for this I/O by the SCSI subsystem in accordance with the interface driver’s direction via ifsw->if_max_tag. It is recommended that the interface driver use this value as the tag value for the I/O if the I/O will be tagged, but it is not required. Currently there can only be 256 tags per bus. The tag value may not remain same for retried I/Os.

scb->cdb_status Indicates the status of the I/O command. If the I/O attempt completes with no phase sequencing errors and without being aborted or timing out, the interface driver sets scb->cdb_status to S_GOOD. If the selection phase times out, the interface driver sets cdb_status to SCTL_SELECT_TIMEOUT. If the I/O is not even attempted because of bogus data in the bp or scb, the interface driver sets cdb_status to SCTL_INVALID_REQUEST. If the I/O is not attempted or does not complete for any other reason, cdb_status is set to SCTL_INCOMPLETE. If there is a Contingent Allegiance condition, the cdb_status is set to
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

S_CHECK_CONDITION to request an auto-sense request. `scb->cdb_status` must be set by the interface driver prior to returning the bp via `scsi_cbfn`.

Refer to `scsi.h` for the valid values of `cdb_status`.

`scb->data_resid`

If the I/O attempt completes with no phase sequencing errors and without being aborted or timing out, the interface driver sets `scb->data_resid` such that `bp->b_count - scb->data_resid` is the offset from `bp->b_un.b_addr` of the first byte not transferred by the target, i.e., number of bytes transferred = `bp->b_bcount - scb->data_resid`. Even if the I/O attempt is failed for some reason, it is advisable to set the `scb->data_resid` to indicate the number of bytes that are not yet transferred. Setting this field will have no adverse affect. `scb->data_resid` must be set by the interface driver prior to returning the bp via `scsi_cbfn`.

`scb->sense_status`

Represents the status of the automatic request sense that is performed if `scb->cdb_status` is `S_CHECK_CONDITION`. If the Request Sense completes with no phase sequencing errors and without being aborted or timing out, the interface driver sets `scb->sense_status`. Otherwise, `scb->sense_status` is undefined and will not be referenced by the SCSI subsystem on callback. The possible values for `scb->sense_status` are the same as those for `scb->cdb_status` except that `SCTL_INVALID_REQUEST` cannot be used. `scb->sense_status` represents the result of the automatic Request Sense in the same way that `scb->cdb_status` represents the result of attempting `scb->cdb`. It must be set by the interface driver before returning the bp via `scsi_cbfn`. If there is any sense data, the `sense_status` has to be set to `S_GOOD`.

`scb->sense_bytes`

Number of bytes of data received in response to the automatic request sense if one was performed. It is valid only if `sense_status` is valid and is neither
Writing a SCSI Interface Driver
External Interfaces to an Interface Driver

SCTL_SELECT_TIMEOUT nor SCTL_INCOMPLETE.
scb->sense_bytes is the offset from
scb->sense_data of the first byte of sense data not
transferred by the target. It must be set by the
interface driver prior to returning the bp via
scsi_cbfn.

scb->sense_data
If scb->cdb_status is Check Condition and the
resulting Request Sense completes with no phase
sequencing errors and without being aborted or timing
out, and if scb->sense_status is not zero, the
interface driver sets scb->sense_data. Otherwise,
scb->sense_data is undefined and will not be
referred to by the SCSI subsystem on callback. The
interface driver sets scb->sense_data to point to a
KERNELSPACE buffer containing the sense data; its
size must be at least scb->sense_bytes. It must be set
prior to returning the bp via scsi_cbfn and the
interface driver must not modify the buffer for the
duration of scsi_cbfn. When scsi_cbfn returns, and
not until, the interface driver can reuse the buffer.

ISC Each instance of an interface card has an Interface Select Code
(ISC) entry that the system maintains in an internal table. Each ISC
card, defined as an isc_table_type structure, is used by WSIO to
maintain interface driver information. Refer to Understanding HP-UX
I/O Subsystem Features in Chapter 3 of the DDG for a detailed
discussion on this structure. It is not part of the SCSI subsystem.

Some of the fields of the isc_table_type structure are reserved for use
by the interface driver as specified by the SCSI subsystem. They are
described in this section. Others are reserved for use by the interface
driver at its discretion. These reserved fields are - ppoll_flag,
ppoll_mask and ppoll_sense. However, these fields are not typically
used by the interface driver. A number of these fields have been renamed
using #defines to more accurately reflect their meaning within the
context of the SCSI subsystem.

Fields specific to a SCSI interface driver are explained here.
```c
struct isc_table_type {
    char my_address;
    struct gfsw *gfsw;
    caddr_t *ifsw;

    unsigned char int_enabled;

    unsigned char spoll_byte;
    unsigned char tfr_control;

    struct buf *ppoll_f;
    struct buf *ppoll_l;

    int lock_count;
    struct buf *event_f;
    struct buf *event_l;
    struct buf *status_f;
    struct buf *status_l;

    char ppoll_flag;
    unsigned char ppoll_mask;
    unsigned char ppoll_sense;

    struct buf *owner;
    unsigned int state;
    int *card_ptr;

    unsigned char my_isc;
    char bus_type;
    caddr_t if_reg_ptr;
    caddr_t if_drv_data;
    void *if_isc;

    int if_id;
} *isc;

#define bus_max_width  int_enabled
#define bus_min_sdtr_period  spoll_byte
#define bus_max_reqack_offset  tfr_control
#define tgt_wdtr_done  ppoll_f
#define tgt_wdtr_width  ppoll_l
#define tgt_sdtr_done  lock_count
```
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

```
#define tgt_sdtr_period event_f
#define if_char0 ppoll_flag
#define if_uchar1 ppoll_mask
#define if_uchar2 ppoll_sense
```

**NOTE**

Interface driver is not allowed to set `if_drv_data`.

isc->myaddress  
SCSI bus address of the initiator. It is a binary value from zero to fifteen; it is not a power of two representing the data bit used by the initiator for selection and re-selection. `isc->my_address` is initialized by the interface driver's attach routine.

isc->gfsw  
Pointer to the interface driver's gfsw structure. The SCSI subsystem does not require that the interface driver provide a gfsw structure.

isc->ifsw  
Pointer to the interface driver's scsi_ifsw structure. It is initialized by the interface driver's attach routine.

isc->bus_max_width  
Width of the SCSI data bus. Currently, reasonable values are 8 and 16. It is initialized by the interface driver during the driver's attach routine. This field is later used by the SCSI services while probing for SCSI devices. Not setting this field results in not seeing devices on this SCSI bus.

isc->bus_min_sdtr_period  
Minimum synchronous data transfer period supported by the hardware. This field is expressed in units of 4 ns. It is initialized during the driver's attach routine.

isc->bus_max_reqsck_offset  
Maximum synchronous data transfer REQ/ACK offset supported by the hardware. It is initialized during the driver's attach routine.
isc->tgt_wdtr_done
Indicates whether or not a WDTR negotiation has occurred since the most recently detected event which resets the data transfer width to eight bits, i.e. bus reset or BDR. There is one bit for each target; the least significant bit is for target zero, the next least significant bit is for target one and so on. If the bit is set, a negotiation has occurred, otherwise no negotiation has occurred.

isc->tgt_wdtr_width
Indicates the current width of data transfers. There is one bit for each target; the least significant bit is for target zero, the next least significant bit is for target one and so on. If the bit is set, sixteen-bit data transfers are in effect, otherwise eight-bit transfers are being used.

isc->tgt_sdtr_done
Indicates whether or not an SDTR negotiation has occurred since the most recently detected event which resets the data transfer parameters to asynchronous, i.e., bus reset, BDR or WDTR. There is one bit for each target; the least significant bit is for target zero, the next least significant bit is for target one and so on. If the bit is set, a negotiation has occurred, otherwise no negotiation has occurred.

isc->tgt_sdtr_period
Represents the location of an array of bytes indicating the current synchronous data transfer period as represented in an SDTR message. The address of isc->tgt_sdtr_period is the start of the array. There is one byte for each target; the byte at offset zero is for target zero, the next byte is for target one and so on.

isc->my_isc
Index into the isc_table array that will yield a pointer to this structure.

isc->bus_type
Bus type of the interface card.
isc->if_reg_ptr
This is a virtual address corresponding to a base address register that has already had this virtual mapping done to make it usable by the driver and system.

isc->if_drv_data
Pointer to the scsi_bus structure. It is NULL if and only if the bus is not open. SCSI services set this field and an interface driver is not allowed to write to this field.

isc->if_isc
Pointer to private data structure for use by interface driver.

isc->if_id
Unique ID (device and vendor ID) of the interface.

The SCSI subsystem maintains an array of isc_table_type structures in scsi_isc[]. It is driver’s responsibility to assign its isc structure to an element in scsi_isc[] array indexed by its instance number.

**Interface Driver Switch**  This structure defines SCSI interface driver entry points and parameters as required by SCSI services. The interface driver's attach routine must initialize the ifsw field of the isc_table_type entry to point to a scsi_ifsw_structure. The contents of the scsi_ifsw structure specify the interface driver entry points and operational parameters to the SCSI subsystem. A detailed description of the fields follows.

```c
struct scsi_ifsw {
    ubit8 if_flags;
    ubit8 if_max_tag;
    unsigned int if_scb_size;
    unsigned int if_lun_size;
    unsigned int if_tgt_size;
    unsigned int if_bus_size;
    int (*if_open)(dev_t dev);
    void (*if_close)(dev_t dev);
    void (*if_start)(struct isc_table_type *isc);
    int (*if_abort)(struct buf *bp);
    int (*if_bdr)(struct buf *bp);
    int (*if_reset_bus)(dev_t dev);
} *ifsw;
```
NOTE

If a field is already described earlier, it will only be mentioned here. For more details on such a field, please refer to earlier sections.

`ifsw->if_flags` Interface driver flags convey information to the SCSI services on what it supports and what not. The possible flags are:

- **IF_BUS_TAGS**
  This is a default tag.
- **IF_NO_TAGS**
  Interface driver does not support tags.
- **IF_B2_LIST**
  If set, it indicates the interface driver supports handling of disksort merge buffers.
- **IF_OWNS_TAGS**
  Interface driver owns tagged queueing.

`ifsw->if_max_tag`

One less than the number of per-bus tags supported by the interface driver. A tag is used to differentiate I/O requests. The SCSI subsystem will use tags from zero through `ifsw->if_max_tag`, inclusive. Actually, the interface driver is not required to use the tags allocated by the SCSI subsystem, but the SCSI subsystem will not allow more than `ifsw->if_max_tag+1` active I/Os to the bus at any given time (this includes untagged I/Os).

`ifsw->if_scb_size`

The number of bytes the SCSI subsystem shall allocate and attach to each scb for use by the interface driver. `if_scb` field of scb structure is initialized at scb creation time by services and the data area is bzero'ed by services for each I/O attempt prior to putting the I/O on the select queue. It is not touched by services at any other time.
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

ifsw->if_lun_size
The number of bytes the SCSI subsystem shall allocate and attach to each scsi_lun structure for use by the interface driver. The if_lun field of scsi_lun structure is a pointer to ifsw->if_lun_size bytes for the use of the interface driver.

ifsw->if_tgt_size
The number of bytes the SCSI subsystem allocates and attaches to each scsi_tgt structure for use by the interface driver. The if_tgt field of scsi_tgt structure is a pointer to ifsw->if_tgt_size bytes for the use of the interface driver.

ifsw->if_bus_size
The number of bytes the SCSI subsystem shall allocate and attach to each scsi_bus structure for use by the interface driver. The if_bus field of scsi_bus structure is a pointer to ifsw->if_bus_size bytes for the use of the interface driver.

ifsw->if_open
Pointer to the interface driver’s logical unit close function. This is optional for an interface driver.

ifsw->if_start
Pointer to the interface driver’s start function.

ifsw->if_reset_bus
Pointer to the interface driver’s Bus Reset function. This is optional for an interface driver.

ifsw->if_bdr
Pointer to the interface driver’s Bus Device Reset function. This is optional for an interface driver.

ifsw->if_abort
Pointer to the interface driver’s Abort function. This is optional for an interface driver.

ifsw->if_io_max_size
Maximum size of I/O request supported by the interface driver. A value of zero (0) specifies no limit. If set, I/O requests for more than the supported size will be erred back by the SCSI services.
Interface driver data buffer alignment requirement. These fields must be set to \((n - 1)\) where \(n\) is a power of two (2). SCSI services will ensure that data buffer \((bp->b_un.b_addr)\) is \(n\)-byte aligned. The maximum of both the fields is used for buffer alignment.

**struct buf**  This structure is the header for buffers in the buffer pool and otherwise used to describe a block I/O request. I/O requests are passed to the interface driver in the form of `buf` structure. Some of the fields that are of interest to an interface driver writer are explained here.

```c
struct buf {
    int32_t b_flags;
    struct buf *av_forw;
    struct buf *av_back;
    union { caddr_t b_addr;
    } b_un;
    int32_t b_bcount;
    dev_t b_dev;
    uint16_t b2_flags;
    struct buf * b_merge;
    uint16_t b_merge_cnt;
    space_t b_spaddr;
    long b_s2;
} *bp;
#define b_scb b_s2
```

---

**NOTE**  Interface driver is not allowed to set the following fields:

- `b_flags`
- `b_un.b_addr`
- `b_bcount`
- `b_dev`
- `b2_flags`
- `b_merge_cnt`
- `b_spaddr`
- `b_s2`
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

bp->b_flags  B_READ is the only bit bp->b_flags that is of interest to the interface driver and only if bp->b_bcount is not zero. If B_READ is set, the I/O has a data in phase; if clear, the I/O has a data out of phase.

bp->av_forw  Position on free list of buffers if not busy. This field is used to save a pointer to the buf structure which is passed to scsi_cbfn() after an I/O request is completed.

bp->av_back  Position on free list of buffers if not busy.

bp->b_un.b_addr  Kernel virtual address of the data buffer for the I/O. This is passed to the DMA mapping routines in the interface driver.

NOTE:

This address may not be cache aligned. This has implications for a read request when part of the cacheline is modified by a processor write. The data after the I/O completion will be stale if cache flush occurs after inbound DMA.

Basically interface drivers must do I/Os to a temporary location for non-cache aligned portions and copy data from temporary buffers to actual data buffers after DMA completions. This differs from the buffer alignment requirement of the interface driver.

bp->b_count  Maximum number of bytes that should be transferred for the I/O.

bp->b_dev  Device number of the destination for the I/O. This is used to obtain a pointer to the scsi_bus structure, a pointer to the scsi_tgt structure, a pointer to scsi_lun structure and a pointer to the isc structure when only the buf structure is available.
SCSI services provide the following services to work with the device number:

- `m_scsi_dev()`
- `m_scsi_tgt`
- `m_susi_lun`
- `m_scsi_isc`

**Additional flags to support B2_LIST buffers.** If the B2_LIST flag is set in `bp->b2_flags`, `bp` represents a disksort merge buffer. Interface driver specifies its capability of handling such buffers by setting `IF_B2_LIST` in `ifsw->if_flags`.

**The b_merge field of the first bp represents a linked list of buf structures containing the actual data. The list itself is chained using the b_merge fields of subsequent buf structures.**

The `b_count` field of the first `bp` represents the total data length in all buffers.

If this is a merge buffer, all data buffers (`bp->b_un.addr`) are page aligned and `bp->b_count` will be a multiple of page size (NBPG).

**A count of merged requests.** If this field is non-zero, DMA mapping is done via `bp_dma_setup()` instead of `dma_setup()`.

**Space address of b_un.b_addr.** This is passed to the DMA mapping routines in the interface driver.

**This field is a pointer to the scb associated with this bp.** It is used in an interface driver to obtain the scb struct which will have additional information on the I/O. Refer to the earlier discussion on struct scb for additional details.
Logical Unit Structure  A SCSI Logical Unit (LUN) structure is created per SCSI LUN. This structure is allocated and initialized when the LUN is first opened, and is deallocated on the last close by the SCSI services. This is owned by the SCSI services, but can be accessed by an interface driver. Only the fields relevant to an interface driver are described.

```c
struct scsi_lun {
    struct scsi_tgt *tgt;
    void *if_lun;
    ubit8 lun_id;
    dev_t dev_minor;
    ubit32 open_cnt;
    ubit32 state;
} *lp;
#define L_TAGS 0x20
```

**NOTE**  The interface driver is not allowed to set the following fields:

- `tgt`
- `lun_id`
- `dev_minor`
- `open_cnt`
- `state`

- `lp->tgt`  Pointer to the `scsi_tgt` structure to which this logical unit structure belongs, i.e., it is a pointer to the logical unit structure's parent in the open device tree.

- `lp->if_lun`  Pointer to the data area allocated and bzero'ed on first open, by the SCSI subsystem, for use by the interface driver. Its size is specified by the interface driver in `isc->ifsw->if_lun_size`. This memory is allocated during the first open of the LUN and is freed during the last close of the LUN.

- `lp->lun_id`  Is the logical unit's identification number.

- `lp->dev_minor`  Minor number of the device minus the volume bits, i.e., the device number independent portion of the minor number.
The number of successfully completed opens that have not had a corresponding close. It includes block, character and pass-through driver opens and closes.

L_TAGS is the only bit of lp->state that is defined for the interface driver; all other bits are undefined. The interface driver will send a simple queue tag message when initiating an I/O to the logical unit represented by lp if (lp->state & L_TAGS) is set. The (lp->state & L_TAGS) bit will never change when there are active I/Os on the logical unit.

Target Structure  A SCSI target structure is created per SCSI target. This structure is allocated and initialized when a LUN connected to the target is first opened, and is deallocated on the last close of a LUN connected to the target by the SCSI services. This is owned by the SCSI services, but can be accessed by an interface driver.

```
struct scsi_tgt {
    ubit32 open_cnt;
    ubit32 state;
    ubit8 tgt_id;
    u_char min_sdtr_period;
    struct scsi_bus *bus;
    void *if_tgt;
    struct scsi_lun *[SCSI_MAX_LUN_ID+1];
} *tp;
```

#define T_ENABLE_WDTR 0x40
#define T_ENABLE_SDTR 0x20

NOTE  The interface driver is not allowed to set the following fields:

- open_cnt
- state
- tgt_id
- min_sdtr_period
- bus
- lun
The number of successfully completed opens that have not had a corresponding close. It includes block, character and pass-through driver opens and closes for all logical units of this target.

Two bits are defined. T_ENABLE_WDTR directs the interface driver to initiate SDTR negotiation after any event other than WDTR negotiation that causes the data transfer width to be reset to eight bits, e.g., bus reset or BDR. If T_ENABLE_WDTR is not set, the interface driver is forbidden from initiating WDTR negotiation, but not from responding. T_ENABLE_SDTR directs the interface driver to initiate SDTR negotiation after any event other than SDTR negotiation that causes the synchronous data transfer parameters to be reset to asynchronous e.g., bus reset, BDR or WDTR negotiation. If T_ENABLE_SDTR is not set, the interface driver is forbidden from initiating SDTR negotiation, but not from responding.

SCSI ID of this target.

Lower limit on the transfer period that should be used for synchronous transfers to this target. It may or may not be smaller than what is supported by the hardware. This value may change at any time and the interface driver is responsible for checking it before initiating every I/O to the target. If the currently negotiated transfer period is smaller than what is supported by the hardware. This value may change at any time and the interface driver is responsible for checking it before initiating every I/O to the target. If the currently negotiated transfer period is smaller than tp->min_sdtr_period and (tp->state & T_ENABLE_SDTR) is set, the interface driver should initiate SDTR negotiation to rectify the situation.

Pointer to the target’s parent scsi_bus structure in the open device tree.
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

tp->if_tgt  
Pointer to the data area allocated and bzero'ed on first open, by the SCSI subsystem for use by the interface driver. Its size is specified by the interface driver in isc->ifsw->if_tgt_size. This memory is allocated during the first open of a LUN on this target and is freed during the last close of a LUN on the target.

tp->lun  
Array of pointers to the scsi_lun structures for open logical units of the target. tp->lun[x] is NULL if and only if logical unit x is not open.

**Bus Structure**  
A SCSI bus structure is created per SCSI bus. This structure is allocated and initialized when a LUN connected to a target on the SCSI bus is first opened, and is deallocated on the last close if a LUN connected to the target on the SCSI bus by the SCSI services. This is owned by the SCSI services, but can be accessed by an interface driver.

```c
struct scsi_bus {
    u_int open_cnt;
    struct isc_table_type *isc;
    void *if_bus;
    u_char bus_id;
    scsi_lock_t *lock;
    struct scsi_tgt *tgt[SCSI_MAX_TGT_ID+1];
    struct buf *select_q;
} *busp;
```

**NOTE**  
The interface driver is not allowed to set the following fields:

- open_cnt
- bus_id
- lock
- tgt

---

busp->open_cnt  
The number of successfully completed opens that have not had a corresponding close. It includes block, character and pass-through driver opens and closes for all logical units of all targets of this bus.

busp->isc  
Pointer to the bus' parent isc_table_type structure in the open device tree.
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

`busp->if_bus`  Pointer to the data area allocated and bzero'ed on first open, by the SCSI subsystem, for use by the interface driver. Its size is specified by the interface driver in `isc->ifsw->if_bus_size`. This memory is allocated during the first open of a LUN on this bus and is freed during the last close of a LUN on the bus.

`busp->bus_id`  Index into the `scsi_isc` array that will yield a pointer to `busp->isc`.

`busp->lock`  SCSI bus spinlock. This lock needs to be held by the interface driver when calling `scsi_enqueue/scsi_dequeue/scsi_dequeue_bp`, `scsi_bus_lock()` and `scsi_bus_unlock()` services are used to acquire and release the lock respectively.

`busp->tgt`  Array of pointers to the `scsi_tgt` structures for open targets of this bus. `busp->tgt[x]` is NULL if and only if target x is not open.

`busp->select_q`  A doubly linked list of `buf` structures onto which services places I/Os ready for selection. The interface driver picks up I/O requests from this queue. A detailed discussion of this field follows.

Select_q  This is the only data structure that is shared between the SCSI subsystem and the interface driver in the sense that both may change it. It is the per-bus select queue.

```c
struct scsi_bus {
    struct buf *select_q;
} *busp;
```

After initializing the bp and its scb and bzero'ing `*scb->if_scb`, the SCSI subsystem enqueues the bp onto the select queue using `scsi_enqueue(&busp->select_q, bp, TAIL)` Once the SCSI subsystem enqueues a bp onto the select queue, it will not modify the bp, its scb or the `if_scb` until the interface driver returns the bp via `scsi_cbfn` nor will the SCSI subsystem remove a bp from the queue once it has enqueued it.
The interface driver must treat the select queue as ordered for any one logical unit and execute a Request Sense during Contingent Allegiance in response to any Check Condition. That is, the interface driver must initiate all I/Os for any logical unit in the order that they were enqueued by the SCSI subsystem and it must not initiate any other command to the logical unit after a Check Condition until the associated Request Sense clears the Contingent Allegiance.

The interface driver may dequeue from and enqueue to the head of the select queue with `scsi_dequeue(&busp->select_q, HEAD)` and `scsi_enqueue(&busp->select_q, bp, TAIL)` provided it observes the ordering requirement. It may also dequeue a specific bp using `scsi_dequeue_bp(&busp->select_q, bp)` with the same restriction.

The interface driver should not access the queue in any other way than through the access functions provided by the SCSI subsystem as mentioned in this section.

The interface driver needs to hold the SCSI bus lock while calling `scsi_enqueue/scsi_dequeue/scsi_dequeue_bp`, `scsi_bus_lock()` and `scsi_bus_unlock()` services are used to acquire and release the SCSI bus lock respectively.

**Data Structure Diagram** To complete the discussion on the various data structures that are of relevance to an interface driver, a diagram is given below to illustrate the inter-relationship among different data structures in the SCSI subsystem and other kernel data structures.
Figure 8-2  SCSI Data Structures
I/O Path

An I/O request typically passes through different queues in the SCSI subsystem when it passes from one layer to another. The following is a brief description of these queues.

- **scb queue**: per-LUN queue defined in the device driver that contains requests that are waiting for an scb. Requests get queued into this queue by the device driver's `dd_strategy` routine and get dequeued by the PD's `dd_start` routine.

- **tag queue**: per-bus queue that contains requests that are waiting for a qtag. Requests that are ready to be retried (i.e., requests on the retry queues that have hit their “time-to-retry”) are placed on the tag queue to be restarted because they already have an scb (from before) but need a new qtag assignment.

- **nexus queue**: per-LUN queue that contains requests that have to wait to be started so as to not exceed the LUN’s queue depth (`lp->max_active`).

- **select queue**: per-bus queue that contains requests that are ready to be handed off to the SCSI HBA Driver.

- **retry queue**: per-bus queue that contains requests that are needed to be retried at some point in the future. The requests are ordered in the queue in “timeout order” (i.e., they’re ordered by the time when the request will be retried).

The following figure illustrates all the queues involved in a typical I/O path.
Writing a SCSI Interface Driver

I/O Path

Figure 8-3  I/O Flow Between Various Queues

I/O Within SCSI Device Driver

The SCSI device driver is responsible for enqueueing the incoming I/O requests in its queue. Device drivers typically calls `scsi_strategy()` of SCSI services for this purpose. The device driver specifies a `dd_strategy` entry point in `scsi_ddsw` structure for SCSI services to call the actual device driver routine that does the queueing.
SCSI services, in addition to calling the device driver strategy routine, provide:

- I/O buffer alignment as per the interface driver requirement
- I/O forwarding, to do further processing on the CPU slated to handle the I/O completion.

The device driver's strategy routine should minimally enqueue the buffer on `lp->scb_q`. The I/O start time should be recorded in `bp->b_qstart` if the I/O request is to be timed.

A SCSI device driver owns the following queues:

- `lp->scb_q`     Device driver I/O queue
- `dd_lun->lun_disk_queue`   
  sdisk driver's queue of sorted I/O requests
- `lp->special_scb_q`    sctl pass-through driver I/O request queue
- `lp->priority_scb_q`   Priority mode I/O queue

Device driver strategy routine may return an error if an I/O request encounters any errors while queueing the I/O request. `dd_strategy` returns non-zero status to indicate an error. If I/O is queued successfully, SCSI services act on the I/O for further processing.

**I/O Within SCSI Service Layer**

The SCSI service layer is responsible for:

- Passing I/O requests from device driver to interface driver.
- Implement flow control policies to honor device I/O queue depth
- Ensure fair distribution of shared bus resources between different luns (tags, scbs).
- Keeping track of I/O time.
- Handle I/O completion.
- Retrying I/O requests if needed.
SCSI services allocate the required resources for the I/O. If tag resources are not available, the buffer is queued to \texttt{busp->tag\_q}. If nexus resources are not available, the buffer is queued to \texttt{lp->nexus\_q}. If all the resources are allocated, SCSI services place the buffers to \texttt{busp->select\_q}. The Interface driver is called through the \texttt{ifsw.if\_start} entry point for processing all the I/O requests queued on its \texttt{select\_q}.

Interface drivers return completed I/O requests to SCSI services by calling \texttt{scsi\_cbfn()}. If I/O is not completed successfully, depending on the I/O return status, a device driver can enqueue the I/O to \texttt{busp->retry\_q}.

The SCSI services layer owns the following queues:

- \texttt{busp->tag\_q} \quad SCB is allocated and I/O is initialized. Waiting for per bus resource.
- \texttt{lp->nexus\_q} \quad I/O request got SCB and tag resources; waiting for nexus resource.
- \texttt{busp->retry\_q} \quad I/O is being retried on failure. Tag and nexus resources are fixed. Waiting for the timeout period to be queued.
- \texttt{busp->select\_q} \quad I/O is enqueued to interface layer. I/O is owned by interface driver until it is returned to SCSI services by calling \texttt{scsi\_cbfn()}.  

**I/O Within SCSI Interface Layer**

Interface driver’s \texttt{if\_start} routine must dequeue each I/O on its \texttt{select\_q} and do whatever is required to execute the I/O request. SCSI interface driver is responsible for:

- Executing the I/O request.
- Time the I/O request as requested by upper layers.
- Return I/O to SCSI services by calling \texttt{scsi\_cbfn()}. 

Interface Driver Development

Developing an interface driver includes the following steps:

1. Driver installation and initialization.
2. Register mapping and DMA.
3. Protection and synchronization.
4. Sending SCSI I/O requests to the target.
5. Processing Completed I/O requests.
6. Interrupt Handling.
7. Special Routines.

Driver Installation and Initialization

A SCSI interface driver must have a `driver_install` routine and a `driver_attach` routine. If the interface driver controls and interacts with a hardware device, the driver is also required to have an interrupt service routine to handle the device interrupts. Typically a `driver_init()` routine is assigned to the `isc->gfsw->init` field which will be called by the WSIO.

For a detailed discussion on the `driver_install()`, `driver_attach()`, `driver_init()` and `driver_isr()` routines, refer to Writing a Driver in Chapter 5 of the DDG.

The following sections explain the specific Interface Driver installation and initialization routines.

Calling `driver_install()`

When the HP-UX system is configured through the `config` command, a table of `driver_install()` entry points is created from information in `/stand/system`.

When `driver_install()` is called by the I/O system configuration process through the `driver_install()` entry point configured in the system, the `driver_install()` routine places the `driver_attach()`
entry point in the table of drivers to be called at configuration time.
driver_install() then calls the wsio_install_driver() routine to
register the driver with the I/O subsystem and returns any error.

**SCSI Specific Part** Every interface driver has to register a probe
function with the WSIO to identify the targets connected to the interface
card. This is typically the next step to attaching the driver_attach()
routine. The SCSI subsystem provides the probe function
parallel_scsi_probe to perform the probe and the WSIO provides
wsio_register_addr_probe() to register a probe function with the
WSIO. The SCSI interface driver sets the “class” field of the drv_info_t
to “ext_bus”.

The following example shows a call to driver_install().

```c
static drv_ops_t qlisp_drv_ops = {
    NULL,     /* d_open */
    NULL,     /* d_close */
    NULL,     /* d_strategy */
    NULL,     /* d_dump */
    NULL,     /* d_psize */
    NULL,     /* d_mount */
    NULL,     /* d_read */
    NULL,     /* d_write */
    NULL,     /* d_ioctl */
    NULL,     /* d_select */
    NULL,     /* d_option1 */
    NULL,     /* reserved1 */
    NULL,     /* reserved2 */
    NULL,     /* reserved3 */
    NULL,     /* reserved4 */
    0         /* d_flags */
};
static drv_info_t qlisp_drv_info = {
    "qlisp",  /* name */
    "ext_bus", /* class */
    DRV_SAVE_CONF | DRV_MP_SAFE | DRV_SCAN, /* flags */
    NULL,     /* b_major */
    NULL,     /* c_major */
    NULL,     /* cdio */
    NULL,     /* gio_private */
    NULL,     /* cdio_private */
};
static wsio_drv_data_t qlisp_wsio_data = {
    "scsi",  /* drv_path */
    T_INTERFACE, /* drv_type */
```
Writing a SCSI Interface Driver

Interface Driver Development

Chapter 8

DRV_CONVERGED, /* drv_flags */
NULL, /* dvr_minor_build - field not used */
NULL, /* drv_minor_decode - field not used */
};

static wsio_drv_info_t qlisp_wsio_info = {
    &qlisp_drv_info, /* driver info */
    &qlisp_drv_ops, /* driver ops */
    &qlisp_wsio_data, /* driver data */
    WSIO_DRV_CURRENT_VERSION
};

extern int (*pci_attach)(uint32_t id, 
struct isc_table_type *isc);
/* To attach PCI driver to system */
static int (*qlisp_saved_pci_attach)();

int
qlisp_install(void) {

    /* Link driver attach function into chain */
    qlisp_saved_pci_attach = pci_attach;
    pci_attach = qlisp_pci_attach;

    /* Register the driver probe function to identify
     * the targets */
    wsio_register_addr_probe(parallel_scsi_probe, "qlsip");

    /* Register driver with WSIO */
    if (!wsio_install_driver(&qlisp_wsio_info)) {
        (void)wsio_unregister_probe(WSIO_ADDR_TYPE, "qlsip");

        /* Remove driver from attach chain as cleanup */
        pci_attach = qlisp_saved_pci_attach;

        return 0; /* WSIO driver install failed */
    }

    return (1); /* 1 = WSIO driver successfully installed */
}

WSIO allows a driver to register its own probe function that is used by
the WSIO to scan for devices underneath an interface card. This is
usually required if the “ioscan” can’t reach the device in question on its
own. What it means is that if a device is behind a bus for which there is
Writing a SCSI Interface Driver

Interface Driver Development

no bus nexus CDIO in the OS, it is the responsibility of the driver to let WSIO inform that it can scan by setting the DRV_SCAN flag in drv_info_t structure and provide a handle to the WSIO by registering a probe function with the WSIO. For a detailed discussion on the driver probe function, refer to Writing a Driver in Chapter 5 of the DDG.

Calling driver_attach()

Use the driver_attach() routine to determine whether the product ID passed in matches the driverAttach device and vendor ID. If the IDs do not match, the driver_attach() routine calls the next attach routine in the chain by calling the *driver_saved_attach() routine.

The driver_attach() routine may be called many times before a match is found. For the device in the first slot, the associated driver_attach() routine is called by the number of devices in the PCI backplane. For the device in the last slot of the PCI backplane, the associated driver_attach() routine is called only once.

When the driver_attach() routine recognizes the device ID, it takes actions such as allocating and initializing its driver control blocks and PCI I/O registers. driver_attach() also sets up driver initialization routine and finally calls isc_claim() to claim the device.

SCSI Specific Part  A SCSI interface driver has additional requirements for hardware that it controls. It must inform the SCSI subsystem of this SCSI hardware by assigning the appropriate pointer in the global scsi_isc array to isc. The index of the appropriate pointer in scsi_isc is the bus ID for the SCSI bus. The index into this array can be obtained by calling the WSIO routine wsio_isc_to_instance(isc, NULL).

Another responsibility of the SCSI interface driver is to attach a fully initialized scsi_ifsw structure to the isc structure via isc->ifsw field. This is typically performed in the interface driver init routine. Each bus could have its own scsi_ifsw if necessary, but it's more likely that a single scsi_ifsw could be shared by all buses being managed by the same interface driver.

The interface driver init routine also sets the initiator ID, SCSI transfer rates and bus width fields in the isc structure by calling

SCSI_GET_INITIATOR_PARAMS macro. The bus_max_width field in the isc structure is important. The parallel_scsi_probe() uses this field
to actually scan the SCSI bus. If this field is not set, the SCSI services do not scan on the SCSI bus behind the interface card. More information on this is provided later in the document.

Device initialization, allocating and mapping SCSI command queues, registering device interrupts etc., can be done in the driver_init() routine. In the attach routine, isc->gfsw->init field is assigned with a pointer to the driver_init() routine. This can also be done by calling the macro CONNECT_INIT_ROUTINE(isc, <driver>_init). The HP-UX configuration continues the initialization by calling the driver_init routines after calling the driver attach routine.

An example driver_attach() routine and a driver_init() routine are shown below.

```c
#define QLISP_ID_SUPPORTED(x) ((x) == QLISP_ID_1020 || 
(x) == QLISP_ID_1040 || 
(x) == QLISP_ID_1240 || 
(x) == QLISP_ID_1080 || 
(x) == QLISP_ID_1280 || 
(x) == QLISP_ID_12160 )

static int qlisp_pci_attach(uint32_t id, 
struct isc_table_type *isc)
{
/* Check if it is the correct device */
if (QLISP_ID_SUPPORTED(id)) {
    /* Check that the PCI register set is correctly 
    * mapped */
    if(isc->if_reg_ptr == NULL) {
        msg_printf("qlisp:
            Mapping QLISP registers failed!\n");
        goto recover1;
    }
    /* Allocate driver control structures */
    /* Initialize the fields in the driver control 
    * structures */
    /* Connect the driver init routine */
    CONNECT_INIT_ROUTINE(isc, qlisp_init);
    /* Claim the device */
    isc_claim(isc, &qlisp_wsio_info);
}
return qlisp_saved_pci_attach(id, isc);

recover1:
    /* Error processing */
}

static int
qlisp_init(struct isc_table_type *isc)
{

    /* Perform SCSI specific steps */

    /* 1. Assign the scsi_isc array with the
    * interface’s isc. SCSI interface drivers must
    * check the instance number returned by
    * wsio_isc_to_instance(). If the returned value
    * is greater than SCSI_MAX_BUS_ID, the driver
    * should return WSIO_ERROR */

    if( (instance = wsio_isc_to_instance(isc, NULL)) >
        SCSI_MAX_BUS_ID) return WSIO_ERROR;
    scsi_isc[instance] = isc;

    /* 2. Attach the interface driver switch */

    /* Allocate memory for scsi_ifsw used by SCSI services */

    /* Initialize the ifsw fields */

    /* Attach the struct ifsw to isc */
    isc->ifsw = (caddr_t)qlisp_ifsw;

    /* 3. Call SCSI_GET_INITIATOR_PARAMS macro to get the
    * initiator ID and the SCSI rate. */

    /* Set the following fields of the isc with the correct
    * values: isc->my_address
    *         isc->bus_max_width
    *         isc->bus_min_sdtr_period
    *         isc->bus_max_reqack_offset
    */

    /* Configure PCI config space on the interface card */
/* Allocate and map memory for SCSI command queues */
/* Reset the interface card */
/* Allocate interrupt objects and activate the interrupt */

return 0;

Setting SCSI Parameters

Some of the state information about the SCSI bus and targets on the bus must persist across opens, but the data structures are deallocated when a device is closed. So this information is kept in the following fields of the isc structure:

- isc->my_address
- isc->bus_max_width
- isc->bus_min_sdtr_period
- isc->bus_max_reqack_offset

For additional details on these fields, please refer to the discussion on the isc structure earlier in this document.

The driver has to set these fields in the driver’s init routine. In the current implementation of the SCSI subsystem, it is required to interact with the processor dependent code on PA-RISC platforms to obtain the SCSI parameter values. However, this may change in future platforms. In order to protect drivers from these changes, the DDK includes SCSI_GET_INITIATOR_PARAMS macro. It wraps the processor dependent calls and passes back the values of SCSI parameters. For the correct operation of the macro, it is strongly suggested not to modify anything in the macro.

NOTE
Include the file scsi_params_macro.h that is distributed with the DDK to access this macro.
SCSI_GET_INITIATOR_PARAMS This macro is called with four arguments; a pointer to the isc structure, and place holders for the initiator ID, SDTR period and the REQ/ACK offset. The last three parameters to the macro correspond to the following fields in the isc.

```
my_address
bus_min_sdtr_period
bus_max_reqack_offset
```

**NOTE**

SCSI_GET_INITIATOR_PARAMS does not set the fields in the isc structure. It just returns the values of the corresponding fields.

The following pseudo code of an interface driver's init routine illustrates the calling convention of SCSI_GET_INITIATOR_PARAMS macro and how to set the fields in the isc.

```c
int driver_init()
{
    
    /* Declare local variables to pass to the macro */
    int initiator_id, sdtr_period, reqack;

    /* Call the macro to obtain the SCSI parameters */
    SCSI_GET_INITIATOR_PARAMS(isc, initiator_id, 
                               sdtr_period, reqack);

    /* Set the corresponding fields in the isc with the 
     * values obtained above. */
    isc->my_address = initiator_id;
    isc->bus_min_sdtr_period = sdtr_period;
    isc->reqack_offset = reqack;

    /* Set the SCSI bus width */
    isc->bus_max_width = width; /* where width equals 
                               * to 8 or 16 */

    ....
}
```
SDTR/WDTR Negotiation

SCSI host bus adapters and targets negotiate for synchronous data transfer rates and wide data transfer widths. Synchronous Data Transfer Request (SDTR) and Wide Data Transfer Request (WDTR) commands are used for this. Typically an interface driver need not do anything for this. However in some scenarios like bus reset or when a slower rate device is connected on the bus, the interface driver may force a negotiation on SDTR and WDTR.

Current Negotiated Information  The currently negotiated synchronous data transfer period for each target is stored in one byte of the tgt_sdtr_period array. A pointer to the byte associated with a particular target is kept in the scsi_tgt structure while the device is open so it can be accessed easily and in a way consistent with other target-specific state information. A value of 0xff implies asynchronous data transfers, otherwise the value is the minimum transfer period from the SDTR agreement.

The currently negotiated bus width for each target is stored in one bit of the tgt_wdtr_width bit field. As with tgt_sdtr_period, a pointer to the bit field is kept in the scsi_tgt structure. If the bit is set, 16-bit wide transfers have been agreed upon.

Two other bit fields in the isc_table_type structure associated with SDTR and WDTR are the tgt_sdtr_done and the tgt_wdtr_done bit fields. There is one bit per target in each indicating whether or not SDTR or WDTR negotiation has been done since the most recently recognized event which reset the device parameters to their default value.

When to Negotiate  The interface driver clears the appropriate tgt_sdtr_done and tgt_wdtr_done bits, sets the appropriate tgt_sdtr_period bytes to 0xff and clears the appropriate tgt_wdtr_width bits in the isc_table_type structure when it recognizes a device reset. The interface driver recognizes a device reset when it initiates or detects a SCSI bus reset or sends a Bus Device Reset (BDR) message to the device. The interface driver will also clear the appropriate tgt_sdtr_done bit when WDTR is negotiated. The interface driver will always initiate SDTR or WDTR if the corresponding “done” bit in the isc_table_type structure is clear and the T_ENABLE_SDTR or T_ENABLE_WDTR bit is set in tp->state. It will also initiate WDTR or SDTR whenever SCB_WDTR or SCB_SDTR is set in the scb independently of
the bits in \( tp->state \). It will initiate WDTR and/or SDTR negotiation on every auto-sense if enabled by \( tp->state \). The interface driver may also initiate negotiation at any other time it deems appropriate.

**What to Negotiate** Three parameters are used in determining the minimum transfer period for which to negotiate:

- \( tp->min_sdtr_period \)
- \( isc->bus_min_sdtr_period \)
- the lowest period at which the hardware can operate

When initiating SDTR, the interface driver will send a period equal to the maximum of \( tp->min_sdtr_period \), and the minimum period supported by the hardware. The interface driver will always negotiate for the maximum REQ/ACK offset allowed by the hardware.

Unlike for SDTR, there is no way to control the bus width for which to negotiate. It is all or nothing. Either negotiation is enabled or not. If negotiation is initiated, it will be for the maximum supported by the bus.

**Register Mapping and DMA**

A brief discussion on PCI register mapping is provided in this section. Refer to *Writing PCI Device Drivers* in Chapter 10 of the DDG for a detailed discussion on PCI register mapping services.

For WSIO drivers, the \( if_reg_ptr \) member of the \( isc \) structure is a PA virtual address corresponding to a base address register that has already had this virtual mapping done to make it usable by the driver and system. If \( if_reg_ptr \) is NULL, the driver needs to map the range itself.

WSIO provides a number of services for DMA mapping. Refer to the man pages on WSIO services in the DDR for a detailed discussion of each of these services. In an interface driver, \( dma_setup() \) or \( bp_dma_setup() \) is called to setup DMA mapping, depending on whether one \( buf \) structure or multiple \( buf \) structures merged together.
Fields relevant only to an interface driver:

```c
struct dma_parms {
    int channel;
    int dma_options;
    int flags;
    struct iovec *chain_ptr;
    int chain_count;
    caddr_t addr;
    space_t spaddr;
    int count;
};
```

There is a derived structure from `dma_parms`.

```c
struct bp_dma_parms {
    struct dma_parms dma_parms;
    struct dma_parms *merge_dma_parms;
};
```

The following steps set up the DMA.

1. Initialize the DMA options.
   - If the device can bus master for DMA, the channel field has to be set to `BUS_MASTER_DMA`. `dma_parms->channel = BUS_MASTER_DMA`.
   - If you don’t want to wait until the `dma_setup()` succeeds, `dma_parms->flags = NO_WAIT`.
   - Set the DMA options.
     ```c
     dma_parms->dma_options = ((bp->b_flags & B_READ) ?
           (DMA_8BYTE|DMA_READ) : (DMA_8BYTE|DMA_WRITE))
     ```

2. If `bp->bp_merge_cnt` equal 0, call `dma_setup()`.
   - Set the address of the buffer to be mapped.
     ```c
     dma_parms->addr =
     ```
   - Set the space address of the buffer to be mapped.
     ```c
     dma_parms->spaddr = bp->b_spaddr.
     ```
Set the buffer length to be mapped.

\[
\text{dma_parms->count} = \text{bp->b_count}.
\]

\[
\text{ret_code = dma_setup(isc, dma_parms);}\]

On a successful call to \text{dma_setup()}, \text{chain_ptr} in the \text{dma_parms} structure points to an \text{iovec} structure that contains the mapped address and the length of the mapping. If there are multiple elements involved, \text{chain_count} field can be used to obtain the number of DMA elements.

3. If \text{bp->b_merge_cnt} > 0, call \text{bp_dma_setup()}. Note that \text{bp_dma_setup()} takes \text{bp_dma_parms} as one argument, not \text{dma_parms}.

\[
\text{ret_code = bp_dma_setup(isc, bp, bp_dmaparms);}\]

**Protection and Synchronization**

The major synchronization issue with interface drivers is to avoid data corruption and race conditions when shared structures are accessed by multiple threads in MP systems. Driver data structures also need protection against interrupts. Spinlock scheme may be chosen to provide finer granularity locks, protecting data structures at finer levels. More information on spinlocks is available in *Using Spinlocks and Semaphores for Synchronization in MP Systems* in Chapter 4 of the DDG.

Whenever an interface driver enqueues or dequeues bp's from the \text{select_q}, it has to acquire \text{scsi_bus_lock()} and release the lock after the completion of \text{scsi_enqueue()}, \text{scsi_dequeue()} or \text{scsi_dequeue_bp()} by calling \text{scsi_bus_unlock()}.

For example:

\[
\text{scsi_bus_lock(busp);}\]
\[
\text{scsi_enqueue(&busp->select_q, bp, HEAD);}\]
\[
\text{scsi_bus_unlock(busp);}\]
Similarly,

```c
scsi_bus_lock(busp);
bp = scsi_dequeue(&busp->select_q, HEAD);
scsi_bus_unlock(busp);
```

It is advisable to protect driver control structures with a spinlock to protect across MP access.

A lock order of (SCSI_LOCK_ORDER_BASE + 2) can be used by the interface driver while allocating new spinlocks.

For example:

```c
typedef struct {
  uint32_t    state;
  ubit8       chip_rev;

  ....

  lock_t      *mp_lock;

  ....
} qlisp_shared_isc_t;
```

It is highly recommended that the driver does not hold spinlocks across calls to different subsystems. So, before calling the SCSI services, the driver should release the spinlocks and acquire them if necessary when the interface driver gets the control back.
Sending SCSI I/O Requests to the Target

The SCSI subsystem requires the interface driver to specify a start function. This is the entry point for an I/O request at the interface driver. The following pseudo code is for a typical interface driver start routine.

This routine is called with a pointer to the isc structure.
Obtain the scsi_bus structure from the isc.
(NOTE: isc->if_drv_data field points to the scsi_bus structure)

```
do

acquire scsi_bus lock

Get a bp from the select_q

release scsi_bus lock

if bp is not NULL

get SCSI Control Block from the bp.

if there are bytes to transfer
(bp->b_bcount != 0)

DMA map the buffer(s)

Get the hardware I/O control block(s)

If the I/O request can’t be satisfied
because of insufficient resources at the hardware, set the scb->cdb_status to
SCTL_INVALID_REQUEST and call scsi_cbfn().

If the I/O request can’t be satisfied
because of temporary resource shortage,
then put the bp back at the HEAD
of the select_q.

Build the SCSI command

Inform the hardware of a pending I/O request
done
```
done

until there are bp’s in the select_q

### Processing Completed I/O Requests

I/O requests which are completed are processed when the I/O card either generates an interrupt or informs by some other means.

The interface driver has to inform the SCSI subsystem that an I/O request is completed so the resources allocated for that I/O can be freed. The SCSI subsystem provides the interface driver with the service `scsi_cbfn()` to accomplish this. However before calling the SCSI callback services, fields in the SCB structure have to be set to indicate the result of an I/O request. An I/O might have completed successfully, or it might have resulted in a Check Condition and an auto-sense message is returned. An I/O might have failed because of lack of resources, an invalid command, or an I/O might have timed out. In any of these scenarios, the SCSI subsystem should know the result of the I/O for further processing. It is the responsibility of an interface driver to set the fields in the SCB structure to provide the correct result to the SCSI subsystem.

For a detailed discussion on the fields of the SCB structure, refer to “SCSI Control Block” on page 330. The following pseudo code defines the steps required to set up the fields of the interface driver. In the examples, `scb` is `bp->scb`.

1. If the I/O request is completed successfully, set:
   ```c
   scb->sense_bytes = 0;
   scb->data_resid = 0;
   scb->sense_status = SCTL_INCOMPLETE
   scb->cdb_status = S_GOOD;
   ```

2. If there is a select timeout, set:
   ```c
   scb->sense_status = scb->cdb_status = SCTL_SELECT_TIMEOUT;
   scb->data_resid = <value returned from the device.>
   ```

   If this generates an auto-sense, follow step 3 to set other fields. Otherwise, set:
   ```c
   scb->sense_status = SCTL_INCOMPLETE:
   scb->sense_bytes = 0;
   ```
3. If the I/O request resulted in an auto-sense, set:

- `scb->sense_status = S_GOOD;`
- `scb->sense_bytes =` length of the driver buffer used for sense
- `scb->sense_data =` pointer to the driver buffer used for sense
- `scb->data_resid =` If data transferred and the sense data has at least
  the infor byte (resid) field, check resid count from there. Else set the field with the value returned from the device.

Code example:

```c
    scb->data_resid = iocb->stat.residual;

    if (iocb->stat.state_flags & QLISP_STATE_GOT_SENSE) {
        scb->sense_status = S_GOOD;
        scb->sense_bytes = iocb->stat.req_sense_length;
        scb->sense_data = iocb->stat.req_sense_data;
        /*
        ** If data transferred and the sense data has at
        ** least the info byte (resid) field, check resid
        ** count from there. Some devices report NO_SENSE
        ** check condition with the resid count in the
        ** sense data and the QLogic card reports zero
        ** resid bytes. Need to update scb->data_resid
        ** from there so physio() truncates the data
        ** correctly.
        */
        if (bp->b_bcount &&
            (scb->sense_bytes >= QLISP_MIN_REQ_SENSE_LEN)) {
            struct sense_2_aligned *sense = (void*)
                scb->sense_data;
            int resid;

            if ((sense->error_code == S_CURRENT_ERROR) &&
                (sense->info_valid) &&
                (scb->data_resid == 0)) {
                resid = (sense->info[0] << 24)
                    + (sense->info[1] << 16)
                    + (sense->info[2] << 8)
+ sense->info[3];

/* Negative resid values equate to a
** record size read smaller than what's
** actually returned by the device. Set resid
** count to zero.
*/
scb->data_resid = (resid > 0) ? (uint32_t)
    resid : 0;
}
}

4. If SCSI command transport failed completely, set:

/* Transport failed entirely, residual = requested length */
scb->data_resid = bp->b_bcount;
scb->cdb_status = SCTL_INCOMPLETE;
scb->sense_status = SCTL_INCOMPLETE;
scb->sense_bytes = 0;

5. If the I/O request is invalid, set:

scb->cdb_status = S_BUSY;

6. If a call to dma_setup() / bp_dma_setup() fails, set:

scb->cdb_status = S_BUSY;

When the control returns to the interface driver after calling
scsi_cbfn(), the interface driver should not touch any of the resources
which are part of the completed I/O. Typically, an interface driver forms a
chain of bp's whose I/O has completed. In order to build such a chain, the
interface driver can use av_forw field of the buf structure. The following
code example clarifies this point.

bp->av_forw = bp_chain;
bp_chain = bp;

If initially bp_chain points to NULL, as bp's become available the
buf_chain points to the most recent bp and the chain can be traversed
through the bp->av_forw field.

When calling scsi_cbfn(), bp's from this list are passed to the SCSI
service.
Interrupt Processing

Writing a Driver in Chapter 5 of the DDG discusses the issues involved in writing an interrupt service routine (ISR). An ISR is a device specific routine and different drivers do different things in handling a device interrupt. Some generic steps are provided for processing a device interrupt.

1. Acquire a spinlock for protection on MP-systems
2. Read a device status register.
3. Check if it is a spurious interrupt. If so, release the lock and return.
4. If an I/O request is completed without any errors, process it.
5. If there is an error, look for the specific event that caused the interrupt.
6. Build a bp chain of all the completed I/Os.
7. Set the device register to indicate that an I/O response is processed.
8. Loop if there are some more completed commands.
9. Wakeup any I/O requests that are waiting for device resources.
10. Release the spinlock.
11. Call SCSI callback services on the I/Os that are completed.
12. Return INTR_SERVICED

Special Routines

A SCSI interface driver can optionally implement the following routines to handle special events such as bus reset, I/O abort, etc., A brief description and pseudo code is provided here for each of the routines.

Bus Device Reset

The SCSI subsystem calls the interface driver's bus device reset (bdr) function in response to SIOC_RESET_DEV ioctl request.

This function is passed with a buf structure bp as its sole argument. bp->b_dev field contains the device number of the target. As mentioned earlier, the SCSI subsystem provides m_* services to obtain required SCSI data structures to execute a BDR.
Pseudo code for a typical BDR routine:

1. Obtain device number from `bp->b_dev`.
2. Obtain `scsi_bus` structure from `m_scsi_bus(bp->b_dev)`.
3. Obtain `scsi_tgt` structure from `m_scsi_tgt(bp->b_dev)`.
4. Obtain `isc` structure from the `bp->isc`.
5. If the device is in reset state, reject the request.
6. Get the driver control structures.
7. Issue a BDR command to the device.
8. Set the device to a known state.

Aborting an I/O Request

The SCSI subsystem calls the interface driver’s abort function in response to `SIOC_ABORT` ioctl request. This function is passed with a `buf` structure `bp` as its sole argument. `bp->b_dev` field contains the device number of the destination I/O. The SCSI subsystem provides `m_*` services to obtain required SCSI data structures to execute an abort of the specified I/O request.

Pseudo code for a typical abort routine:

1. Obtain device number from `bp->bp_dev`.
2. Obtain `scsi_bus` structure from `m_scsi_bus(bp->b_dev)`.
3. Obtain `scsi_tgt` structure from `m_scsi_tgt(bp->b_dev)`.
4. Obtain `scsi_lun` structure from `m_scsi_lun(bp->b_dev)`.
5. Obtain `isc` structure from the `bp->isc`.
6. If the device is in reset state, reject the request.
7. Get the driver control structures.
8. Issue a command to abort the LUN device.
9. Set the device to a known state.

Reset Bus

The SCSI subsystem calls the interface driver’s `reset_bus` function to reset the SCSI bus. This function is passed with an `isc` structure as its sole argument.
Pseudo code for a typical reset_bus routine:

1. Get the driver control structures.
2. Acquire a driver spinlock.
3. Check if the device is under reset state. If so, release the spinlock and return. Otherwise, set the device state to RESET state.
4. Release the spinlock.
5. Issue a command to the device to reset SCSI bus.
6. Acquire a driver spinlock.
7. Set the device state to RESET_DELAY state.
8. Set a time-out after 3 seconds.
9. Release the driver lock.
10. After the time-out pops, reset the driver state.
11. Call the driver’s if_start() function to process I/O requests.

**Handling Timeouts**

It is a common practice in driver development to use timeouts to schedule events after a pre-determined time interval. The timeout() kernel service can be used to schedule an event to occur after a specified amount of time. An untimeout() kernel service is also provided to unschedule a timeout. It is particularly important to untimeout() any pending timeouts before unloading a DLKM driver. Refer to *Kernel Services* in Chapter 2 of the DDR for more information on timeout and untimeout services.
Sleep and Wakeup Mechanism

At times, an interface driver may need to wait for an event like availability of resources or until the device establishes a known state. An interface driver typically calls “sleep” if it can from the current context instead of busywaiting and monopolizing the CPU. When a corresponding event occurs, the driver calls a “wakeup” to notify any sleeping threads on this event.

To avoid any racing conditions, the kernel spinlock() services provide get_sleep_lock() call. A get_sleep_lock() obtains a lock before calling sleep and releases after it calls sleep. Similarly, a wakeup() call is also protected with the locks obtained from get_sleep_lock(). For a more detailed explanation of this refer to Multiprocessing in Chapter 4 of the DDG.

Example code for a typical sleep/wakeup pair routine:

```c
/*
 * qlisp_sleep -
 * Sleep a process on an address
 */
STATIC void
qlisp_sleep(lock_t *wlock, caddr_t wchan) {  
    VASSERT(owns_spinlock(wlock));
    (void) get_sleep_lock(wchan);
    spinunlock(wlock); /* wlock is a driver spinlock */
    (void) sleep(wchan, PRIBIO);
    spinlock(wlock); /* wlock is a driver spinlock */
}

/*
 * qlisp_wakeup -
 * Wakeup processes sleeping on an address
 */
STATIC int
qlisp_wakeup(lock_t *wlock, caddr_t wchan) {  
    VASSERT(owns_spinlock(wlock)); /* wlock is a driver spinlock */
    return wakeup(wchan);
}
```
Sample Driver

A sample SCSI interface driver for Qlogic’s ISP12160 is provided along with this document. A review of the earlier sections of this chapter will ensure a better understanding of this driver.

Description of the Device

A brief description of the device and its functionality is included to keep the driver in its correct perspective. However, this is not a complete description of the device. For that, refer to Qlogic’s hardware reference manual.

The ISP12160A supports dual channel, Ultra3 (Fast-80) SCSI functionality. It interfaces the PCI bus to two Ultra3 SCSI buses and contains an on-board RISC processor. The interface between the ISP12160A firmware and drivers consists of two queues; request and response. The queues are located in host memory and are organized as circular fixed-length lists of 64-byte entries (FIFOs). They are I/O mapped and their physical addresses are stored in the device registers.

This section presents the code flow of the sample Qlogic driver. Refer to the sample driver code for details. This section is organized as follows:

- Driver Architecture
- Driver Data Structure
- qlisp_multi driver
  - Driver Installation and Initialization Routines
  - Driver entry points to SCSI Subsystem
  - The main I/O path
  - Device Access and Initialization Routines
  - Interrupt Processing Routines
  - Sending an I/O Request to the target.
  - Processing I/O responses.
  - Miscellaneous routines
Driver Architecture

The sample driver is two drivers; one for claiming the PCI interface card and probing the SCSI channels underneath the PCI card, and another for claiming the SCSI controllers. The first one is "qlisp_multi" and the second driver is "qlisp". It is very important to understand that this kind of two-driver architecture is required only for special devices with multiple SCSI channels behind a PCI function. In such cases, since the HP-UX ioscan can't recognize the SCSI controllers that are underneath a PCI interface, two drivers are required.

The qlisp_multi driver registers a driver specific probe function to scan for SCSI controllers during the driver installation. After the driver claims the card, it sets a `WSIO_IDENTIFY_CHILD` routine to correctly identify the child node. When the driver's scan finds a SCSI controller, the second driver's attach routine claims the SCSI channels. Now, if an interface card has each SCSI channel on a separate PCI function, this kind of two driver approach is not required.

As far as each driver's functionality is concerned, the qlisp_multi driver is involved only in building the I/O tree with the SCSI controllers. The real work is done by the qlisp driver. If a two driver approach is not needed, qlisp driver is the one which is of interest.

Data Structures

The following are the main data structures used in the driver.

- `qlisp_isc_t`: Local interface specific control structure (lisc). This is allocated per device instance.
- `qlisp_shared_isc_t`: Common device control structure. This is allocated when the master port is initialized.
- `qlisp_scb_t`: Interface-specific SCSI Control Block structure. This is used to keep track of the associated bp and scb.
- `qlisp_bus_t`: Interface-specific SCSI bus structure. This is used to locally associate an I/O (based on the tag) with an I/O request.
qlisp_q_ent_t This is a union of different 64-byte structures which can be entries in the request and response queues. These entries can be: Command, Command A64, Extended Command, Continuation, Continuation A64, Marker, Status and Extended Status.

The driver design facilitates traversing from one data structure to another. The inter-relationship among data structures is shown here.

Figure 8-4 Driver Data Structure Inter-relationship

qlisp_multi driver

This section briefly describes the qlisp_multi driver. The objective is to present the code flow of the qlisp_multi driver.

The qlisp_multi driver is a typical WSIO interface driver. The installation and initialization routines of the driver of the qlisp_multi driver are similar to the routines explained in Writing a Driver in Chapter 5 of the DDG. However there are some differences that are specific to the functionality of this driver and they are explained below.

The qlisp_multi driver claims the interface card and scans for SCSI controller chips underneath it. So the driver should be able scan the hardware. This requires three things:

1. It has to inform the WSIO that it can scan for the hardware.
2. It has to provide a device specific probe function to the WSIO to scan the hardware.
3. It has to inform the WSIO of a child driver identifying routine.

The last two steps are typically performed in the
qlisp_multi_install() and qlisp_multi_attach() routines.

To inform the WSIO that the driver can scan, the DRV_SCAN flag is set in
the drv_info_t structure.

```c
static drv_info_t qlisp_multi_drv_info = {
    "qlisp_multi",      /* name */
    "qlispba",             /* class */
    DRV_SAVE_CONF | DRV_MP_SAFE | DRV_SCAN, /* flags */
    NULL,                  /* b_major */
    NULL,                  /* c_major */
    NULL,                  /* cdio */
    NULL,                  /* qio_private */
    NULL,                  /* cdio_private */
};
```

Snippets from the qlisp_multi driver's install and attach routines are
shown here to illustrate the driver steps.

```c
int qlisp_multi_install()
{
    ....
    wsio_register_dev_probe(DRV_NAME, &qlisp_multi_dev_probe,
                             "qlisp_multi");
    wsio_install_driver(&qlisp_multi_wsio_info);
    ....
}
```

The calling semantics for the device probe routine are:

```c
qlisp_multi_dev_probe(this node,
                      drv_info,
                      probe_id,
                      hw_path, isc,
                      probe_type,
                      name, desc);
```
In the driver attach routine qlisp_multi_pci_attach,

```c
int qlispMulti_pci_attach()
{
    ....
    isc_claim(isc, &qlisp_multi_wsio_info);
    wsio_set_parm(isc, WSIO_IDENTIFY_CHILD,
                  (void *)&qlisp_multi_identify_child);
    ....
}
```

The calling semantics for the driver identify child routine is:

```c
qlisp_multi_identify_child(child_name);
```

The driver's probe function is called by the WSIO during the hardware scan, and if a device is found the WSIO calls the driver's child identify routine to verify the driver name.

These routines are required only for a multi-port interface card. A detailed discussion on the device probe function and the `WSIO_CHILD_IDENTIFY` routines follows.

**qlisp_multi_dev_probe**

To understand a device probe routine, it is necessary to have an understanding of the hardware path.

A hardware path is a numerical string of hardware components, notated sequentially from the bus address to the device address. Typically, the initial number is appended by slash (/), to represent a bus converter (if required by your machine), and subsequent numbers are separated by periods (.). Each number represents the location of a hardware component on the path to the device. The hardware path is defined by the following structure:

```c
/* HW path structure (hw_path(GIO4) - This structure is used
 * to define a hardware path. The first/last indices define a
 * window of address elements which are meaningful. If
 * last_index == (first_index-1) then the path is NULL.
 */
#define MAX_ELEMENTS 14
typedef struct hw_path {
    char   first_index;
```
Writing a SCSI Interface Driver

Sample Driver

```c
char last_index; unsigned char
addr[MAX_ELEMENTS];

} hw_path_t;
```

The device probe routine is called with the following arguments:

- a pointer to the calling I/O tree node
- a pointer to the driver info structure
- `probe_id` which is set by the interface driver
- hardware path of the calling device
- a pointer to the isc structure
- `probe_type`
- a pointer to name of the child node, and
- a pointer to description of the child node

When this routine is called for the first time by the WSIO, a probe type of `PROBE_FIRST` is passed and the hardware path structure has the fields set to satisfy the following condition:

```c
hw_path->first_index == hw_path->last_index + 1
```

That way when a device is found, the driver increments the `last_index` and it becomes equal to the `first_index` and the probe function places the hardware address of first device found in the `hw_path->addr` array at an index of `last_index`. Later the probe function is called with the probe type of `PROBE_NEXT` and with `first_index == last_index`; both of which point to the hardware address of the last found device. When probe type is `PROBE_ADDRESS`, the probe routine should retrieve the last element of the hardware path and probe the device at that address. The last element is referenced by `last_index`.

If a device is found, three pieces of information are retained from the device: an “ID” (the product and vendor ID strings), a “description” (driver description) and a “name” (the expected interface/device path to this device - e.g., “scsi_disk”). The WSIO CDIO will use the “name” property to recognize a corresponding driver and match the device when the node is later claimed.

The driver should know when the address range goes out of range and should return an error in such a case.

If a device is found, the routine returns `PROBE_SUCCESS` or `PROBE_UNSUCCESSFUL`.
The following is a sample device probe routine which illustrates the concepts.

```c
static int
qlisp_multi_dev_probe(
    void *this_node,
    drv_info_t *drv_info,
    void *probe_id,
    hw_path_t *hw_path,
    struct isc_table_type *isc,
    int probe_type,
    char *name,
    char *desc)
{
    int found = NO_DEV;
    int target;

    do {
        switch (probe_type) {
            case PROBE_FIRST:
                target = 0;
                probe_type = PROBE_NEXT;
                hw_path->last_index++;
                break;
            case PROBE_NEXT:
                target = hw_path->addr[hw_path->last_index] + 1;
                break;
            case PROBE_ADDRESS:
                target = hw_path->addr[hw_path->last_index];
                break;
            default:
                found = INVAL_TGT;
        }

        if (((target < 0) || (target >= QLISP_MULTI_NUM_PORTS)) ||
            (hw_path->last_index != hw_path->first_index))
            found = INVAL_TGT;
    }

    if (found == NO_DEV) {
        strcpy(name, QLISP_NAME);
        strcpy(desc, QLISP_DESC_BUS);
        *(int *)probe_id = isc->if_id;
        found = VALID_TGT;
    }
```

Writing a SCSI Interface Driver

Sample Driver
Writing a SCSI Interface Driver

Sample Driver

```c
hw_path->addr[hw_path->last_index] = target;
}
} while ((probe_type != PROBE_ADDRESS) && (found == NO_DEV));
return ((found == VALID_TGT) ? PROBE_SUCCESS :
        PROBE_UNSUCCESSFUL);
}
```

qlisp driver

This section briefly describes the qlisp driver. The objective is to present
the code flow of the qlisp driver. qlisp_multi driver is a typical WSIO
interface driver. The installation and initialization routines of the driver
of the qlisp_multi driver are similar to the routines explained in
Writing a Driver, Chapter 5 of the DDG. However, qlisp driver registers
an address probe function to identify all the LUNs connected to the port.

The driver should be able to scan the hardware. This requires two things:

1. It has to inform the WSIO that it can scan for the hardware.
2. It has to provide a device specific probe function to the WSIO to scan
   the hardware.

To inform the WSIO that the driver can scan, DRV_SCAN flag is set in the
drv_info_t structure

```c
static drv_info_t qlisp_drv_info = {
    "qlisp",              /* name */
    "ext_bus",            /* class */
    DRV_SAVE_CONF | DRV_MP_SAFE | DRV_SCAN, /* flags */
    NULL,                 /* b_major */
    NULL,                 /* c_major */
    NULL,                 /* cdio */
    NULL,                 /* gio_private */
    NULL,                 /* cdio_private */
};
```

Snippets from the qlisp driver’s install routine illustrate the previously
mentioned steps in the driver.
int qlisp_install(void)
{
    ....
    wsio_register_addr_probe(parallel_scsi_probe, "qlisp");
    wsio_install_driver(&qlisp_wsio_info);
    ....
}

parallel_scsi_probe is the address probe function for the parallel SCSI drivers. It determines the next address to be probed. Registering this probe function is required to determine the targets and LUNs attached to each port. An explanation of this probe function is given in Writing a Driver in Chapter 5 of the DDG.

All routines that are internal to the qlisp driver are given a brief description and for some, pseudo code is provided to facilitate easy understanding of the driver code.

qlisp_init_pci_cfg
Turn on the flags in the config space control registers that we care about.

qlisp_init_reset_card
- Soft reset the card
- Write into some registers
- Set SHARE_DMA64 flag

qlisp_init_attributes
Set DMA attributes like
- pre-fetch length (in this case, 0)
- DMA width
- Coherency protocol etc.

qlisp_init_alloc_queues
This allocates and maps memory for request & response queues.
Below, “share” is a pointer to qlisp_shared_isc_t data structure.

The “iova’s” are stored in: share->dma_req_queue_p & share->dma_res_queue_p. share->req_max_iocb is set.

qlisp_init interrupts
This allocates an interrupt object and activates the interrupt. It also enables device interrupts by writing into the corresponding device registers.

qlisp_init_firmware
- Load the firmware in case of PA-RISC
- Set firmware features.

qlisp_init_common
- Save the isc in scsi_isc() maintained by the system.
- Call SCSI_GET_INITIATOR_PARAMS macro to obtain SCSI parameter values.
- Set the initiator ID.
- Set the correct transfer period and req/ack offset.
- Call init_ifsw()

qlisp_init_ifsw
- Allocate memory for scsi_ifsw structure used by scsi_ctl
- Initialize fields in the if_sw structure
- Assign isc->ifsw with a pointer to the ifsw structure allocated

qlisp_init_start_queues
- Initialize the request and response queues. These queues hold entries of I/O Control Blocks (iocb)
- Change the card state to (SHARE_RSP_QUEUE | SHARE_REQ_QUEUE)

qlisp_init_master
Call routines qlisp_init_pci_cfg through qlisp_init_start_queues in the order set in this section of the manual.
Writing a SCSI Interface Driver

Sample Driver

**qlisp_init_slave**
- Save slave_isc in master_lisc
- Call qlisp_init_common()

**qlisp_pci_attach**
- Allocate memory for interface specific control structure
- Assign isc->if_isc with the driver control structure (lisp) allocated
- Compare the device ID to verify whether the card can be claimed or not
- Allocate memory for the shared control structure for a dual port card
- Initialize PCI config space values
- Attach the init routines to the list of init routines
- Claim the card instance

**qlisp_mailbox_cmd**
Mailbox commands are not done in performance path, and usually only during the driver initialization time. Send a mailbox command. (details later).

**qlisp_build_cmd (and cmd 64 for __LP64__)**
- Set the iocb fields
- Get the scsi_lun structure associated with the scb
- Get the scsi_tgt structure associated with the lp
- Set the target_id and lun_id fields of the command
- Set the control flags of the command
- Copy the cdb from scb into iocb
- Take the DMA parameters and sync the memory

**qlisp_iocb_cnt**
Calculate number of DMA command entries from DMA scatter/gather list.

**qlisp_get_req_iocb**
- Checks request queue for enough available iocb's for command
- Updates share->req_free and share->req_in
**qlisp_if_start**

Interface entry point called by SCSI services where I/Os are started. This entry point is set in the ID ifsw structure in qlisp_init_ifsw().

- Get the ID specific control structure from isc->if_isc
- Get scsi_bus structure from isc->if_drv_data (this field is set by the scsi_services)
- Get the shared control structure from the lisc

```c
do {
    Acquire scsi bus lock.

    NOTE: Once an I/O has been put on the select queue it is off limits to services until the interface driver calls scsi_cbfn with the I/O.

    Get a bp waiting for interface driver
    (Call scsi_dequeue, a SCSI service to get a bp from the select queue.

    Release scsi bus lock

    if bp is not NULL
    Get SCSI control block.
    Get a pointer to driver’s local scb structure.
    Get a pointer to driver’s local bus structure.

    if there are bytes to transfer (bp->b_bcount != 0)
    Perform mapping for dma pages
    Call get_iocb_cnt (cnt64 for LP64)
    Get required iocbs (this may not always succeed)
    if didn’t get reqd iocbs, then
        if the request can’t be satisfied,
            Set the scb status to invalid
            (scb->cdb_status = SCTL_INVALID_REQUEST)
            and call scsi_cbfn() (SCSI service)
        else (if request can be satisfied later)
            Requeue the bp (scsi_enqueue(&busp->select_q, bp, HEAD))
    /* everything is fine */
    Call build cmd (cmd64 for LP64)
    Increase the activecnt in the interface driver local bus structure.
    /* scb->tag is used as an index into the NexusTable. The
Writing a SCSI Interface Driver

Sample Driver

```c
scb->tag is assigned immediately following scb association in scsi_start and is retained for all retries. */

Save the interface driver local scb in interface driver local bus structure.
Inform the device by moving in the req_in pointer.
}
```

### qlisp_fast_complete

**NOTE**

Used by performance status completion mechanism when status is good and only handle is returned.

Get local scb from the local bus structure’s NexusTable field.
Get bp from local scb.
Get scb from local scb.

if bp->b_count != 0, cleanup the DMA

The next few fields indicate the result of an I/O attempt. The appropriate fields are set by the interface driver prior to calling scsi_cbfn().
Set the following fields (as reqd.)
- scb->cdb_status
- scb->data_resid - number of data bytes that were transferred in response to scb->cdb, i.e.
  number of bytes transferred = bp->b_count - scb->data_resid
- scb->sense_status
- scb->sense_bytes
- scb->sense_data

Decrement activecnt of the outstanding I/O requests
Set the local scb structure in the Nexustable to NULL
Put the bp in the free list (bp->av_forw)
Assign the bp to the list of bps waiting for scsi_cbfn.
qlisp_post_reset_delay
Change the lisc state to ~LISC_RESET.
Call qlisp_if_start() if reset completes.

qlisp_call_cbfn

NOTE
Invoke callback functions for chain of bp’s.

Get the scsi_bus structure from the isc->if_drv_ata
while there is a bp in the cbfn list
  Get scb from the bp
  Set cbfn to bp->av_forw
  Call scsi_cbfn()
done with while

qlisp_process_rsp

NOTE
Loop through response blocks and process

Get scsi_bus structure from isc->if_drv_data.
Get the local bus structure from the scsi_bus structure.

while queue is not empty
  Get the iocb from the response queue.
  Move the rsp_out to the next one with a circular wrap-up if reqd.
  NOTE: Performance status completion for more than 5 I/O’s at one time.

  Get the number of I/O handles in done_cnt.
  while the done_cnt is not 0
    Call qlisp_fast_complete on the corresponding handle
done with while
Signal chip that responses have been processed by moving out pointer

qlisp_isr

Acquire the driver spinlock. Get the driver local control structure (lisc).

Read the device register.

while reg val is SEMA_LOCK

    Check the register for command status
    if the command is completed,
        Read the other mailbox commands
        If the command is completed with no errors,
            just break.
        else print an error message and break.
    else /* An asynch event */
        if the mb_status is one of the port RIO_POST statuses, then
            Get the number of I/Os done
            Call qlisp_fast_complete() on each of them
        else check for possible errors
    done with while

if the mailbox command is not complete, get the rsp_in.
else
    Check the share state if it is in SHARE_RSP_QUEUE and if the response Q is not empty,
    Call qlisp_process_rsp().

if there was a reset earlier, call qlisp_post_reset_delay()
Check for other share->state status values.

if there are any bps whose i/o is completed,
Call qlisp_call_cbfns().

qlisp_if_lun_open

LUN open interface entry point called by SCSI services.
qlisp_if_abort

NOTE
Interface entry point called by SCSI services where an I/O is aborted

- From the buf structure, get the device number, scsi_bus structure,
  scsi_target structure and scsi_lun structure.
- Submit a mailbox command to abort device LUN.
- If the card is not in reset state, call qlisp_send_marker.

qlisp_if_bdr

NOTE
Interface entry point called by SCSI services where a device is reset.

- Submit a mailbox command Bus Device Reset Target
- If the card is not in reset state, call qlisp_send_marker

qlisp_reset_bus

NOTE
Causes a SCSI bus reset with a mp_timeout delay before I/Os are scheduled.

- Set lisc state to LISC_RESET
- Submit a mailbox command to Reset SCSI bus
- Call qlisp_post_reset_delay after 3 secs. (via a time-out)

qlisp_if_reset_bus

NOTE
Interface entry point called by SCSI services where the SCSI bus is reset

- Get scsi_bus structure from the device number
- Get isc from the scsi_bus structure
- Call qlisp_reset_bus
**qlisp_send_marker**

This routine is used to bring the device to a known state. This re-enables the adapter/target/lun request queue.