12 Writing a SCSI Interface Driver

This chapter provides information on designing and developing SCSI transport drivers, also known as a Host Bus Adapter (HBA) drivers. SCSI transport drivers are WSIO interface drivers. See “Overview of Driver Types” in Chapter 2, “HP-UX I/O Subsystem Features,” for more information. SCSI transport drivers include drivers for parallel SCSI (pSCSI) and Fibre Channel (FC) transports.

This chapter details:

• “External Interfaces to a Transport Driver”
• “External Data Structures”
• “I/O Path”
• “Transport Driver Development”
• “Sample Driver”
• “Routines”

Code snippets from a sample HP-UX transport driver for Qlogic's ISP12160 SCSI Ultra160 interface card are provided as an example, during the explanation of transport driver development.
External Interfaces to a Transport Driver

The external interfaces available to a transport driver fall into three general categories:

- Driver exported functions that other modules can use to enter the driver (entry points).
- Service functions exported by other modules for use by drivers (SCSI, WSIO or Kernel Service).
- External data structures needed by the entry points or services.

Entry Points

The driver can only be entered via externally-visible driver routines, called “entry points”. There are five general categories of entry points for a SCSI Transport Driver:

- Configuration entry points.
- Request entry points.
- Interrupt entry points.
- Timeout entry points.
- WSIO Event entry points.

Configuration Entry Points

Include the driver's install routine, attach and init routines, and optionally a probe routine. See “Step 5: Writing Configuration Routines” in Chapter 4, “Writing a Driver.”

Request Entry Points

Defined by the SCSI Services in the scsi_ifsw structure. The driver initializes and registers its scsi_ifsw structure in the driver's init routine (by placing a pointer to it in isc->ifsw). These entry points include optional bus, target, LUN open and LUN close routines, and a mandatory start routine to handle the sending of SCSI Command Blocks (CDBs), and various optional ioctl or SCSI Task Management related entry points (abort, bdr, reset_bus, and ioctl). Optional request entry points can be set to NULL by the driver in its scsi_ifsw structure to indicate that it does not support the entry point. Refer to the “Request Entry Point Details” section for details.

Interrupt Entry Points

Registered from the driver's init routine via WSIO services. For details, refer to “Interrupt Handling” in Chapter 2, “HP-UX I/O Subsystem Features,” and the description of the qlisp_init_interrupts() function in the “Sample Driver” section, in this chapter.

Timeout Entry Points

The timeout callbacks that are registered in the driver when a timer is started. See “Timeout Mechanisms” in Chapter 2, “HP-UX I/O Subsystem Features,” and examples in the Qlisp sample driver code for details.

WSIO Event Entry Points

Registered from the driver's install routine via the WSIO service wsio_install_drv_event_handler(). These entry points currently include suspend and resume entry points for OLA/R with additional entry points to be supported in the future. Refer to Chapter 15, “On-Line Addition / Replacement,” for additional details.
Request Entry Point Details

As previously outlined, the driver's request entry points, which are defined in the `scsi_ifsw` structure, are composed of the following sets of entry points:

- Bus, target, LUN open and LUN close entry points (optional)
- Start entry point (mandatory)
- Abort, `bdr`, `reset_bus`, and `ioctl` entry points (optional)

Details regarding each of these entry points are provided.

Bus, Target, LUN Open/Close Entry Points

The `scsi_ifsw` structure contains the following definitions for these entry points:

```c
int if_open(dev_t dev) /* if lun_open */
int if tgt_open (dev_t dev)
int if bus_open (dev_t dev)
void if_close (dev_t dev)
void if tgt_close (dev_t dev)
```

Whenever a LUN is opened, the bus, target, and LUN entry points, if non-NULL, are called in the corresponding transport driver, passing the LUN's `dev_t` as a parameter, to allow the corresponding transport driver to do any bus, target, or LUN-specific processing that may be needed at open time. The transport driver can obtain a pointer to the corresponding `scsi_bus`, `scsi_tgt`, and `m_scsi_lun` structure via the SCSI Services `m_scsi_bus`, `m_scsi_tgt`, and `m_scsi_lun` respectively, refer to their definitions in the *HP-UX 11i v1 Driver Development Reference Guide*, which can be used to obtain corresponding driver-local structures and to do the processing that is needed for the specific open or close. For example, the driver can ignore non-first opens by checking the `open_cnt` field in the corresponding `scsi_bus/tgt/lun` structure.

These entry points are never called under interrupt context and are allowed to sleep.

The SCSI Services provides protection that blocks all other opens and closes to the same bus, target, or LUN, until the transport driver returns from the entry point.

Start Entry Point

The `scsi_ifsw` structure contains the following definition for this entry point:

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

The transport driver must specify a non-NULL `if_start` function. The SCSI Services calls this entry point to inform the transport driver that a `bp` (pointer to a `buf` structure) has been queued on the driver's Select Queue for processing. This will be a `bp` for the SCSI command that needs to be performed.

The `if_start` entry point can be called on the Interrupt Control Stack (ICS) and is therefore not permitted to sleep.

The SCSI Services can call `if_start` at any time while the specified LUN is open, but has been optimized to only call it if the Select Queue is not empty. The transport driver must be prepared to process new requests on the Select Queue without receiving additional calls to `if_start` as long as the select queue has not gone empty while the driver is holding the `scsi_bus` lock. The Select Queue is synchronized between the SCSI Services and the transport driver via the `scsi_bus` structure's lock (`scsi_bus->lock`).
Abort, bdr, reset_bus, ioctl Entry Points

The `scsi_ifsw` structure contains the following definitions for the entry points:

```c
int if_abort(struct buf *bp)
int if_bdr(struct buf *bp)
int if_reset_bus (dev_t dev)
int if_ioctl(dev_t dev, int cmd, caddr_t data, int_flags)
```

The first three entry points if non-NULL, are called by the SCSI Services upon receipt of an `SIOC_ABORT ioctl`, an `SIOC_RESET_DEV ioctl`, or an `SIOC_RESET_BUS ioctl`. The `if_ioctl` entry point, if non-NULL is called upon receipt of `ioctl`s that can’t be handled at the class driver or SCSI Services level.

The `if_reset_bus` entry point is provided for use in Parallel SCSI (pSCSI) transport drivers to reset the corresponding pSCSI bus. When `if_reset_bus` returns, the SCSI bus should have been reset. Outstanding I/O’s, whether queued in the transport driver, in the Host Bus Adapter (HBA) or in the device at the time of the reset must be cleaned up in hardware or software and returned to the SCSI Services, see `scsi_cbfn`.

The `if_bdr` entry point is intended for use in any of the SCSI transports to cause the transport driver to send a SCSI TARGET RESET to the specified target port.

The `if_abort` entry point is intended for use in any of the SCSI transports to cause the transport driver to send a SCSI ABORT TASK SET to the specified LUN.

The `if_ioctl` entry point is intended for processing of transport-specific `ioctl`s. It is called for `ioctl`s that can’t be handled at the class driver of SCSI Services level to allow the transport driver the opportunity to process the `ioctl`. The `if_ioctl` shall return -1 if the transport driver doesn’t support the requested `ioctl`.

Service Functions

The WSIO and kernel services are described in previous chapters. A description of the transport driver usage of SCSI Services follows.

Categories of SCSI Services functions used by transport drivers:

- I/O Completion Callback Functions
- Queuing Functions
- Dev_t Mapping Functions
- Synchronization Functions
- Miscellaneous Service Functions

In addition, new WSIO Services that can be used for virtual bus, target, LUN management are also discussed:

- I/O Tree Node Management Functions
I/O Completion Callback Functions

When a SCSI transport driver is finished with an I/O, it returns the I/O to the SCSI Services by calling one of the two completion callback functions, `scsi_cbfn` or `scsi_fast_cbfn`, which have the following interface definitions:

```c
void scsi_cbfn(struct buf *bp),
void scsi_fast_cbfn(struct buf *bp),
struct scb *scb,
struct scsi_lun *lp,
struct scsi_bus *busp;
```

The `scsi_cbfn` function can be used by the driver for simplicity in lower performance paths. The `scsi_fast_cbfn` function is intended for use in high performance paths.

The transport driver relinquishes all rights to access the `bp`, `scb` and `*scb->if_scb` once it calls the completion function. The `bp` may be freed or reused by the upper layers later for another I/O, and similarly for the `scb` and `*scb->if_scb`.

If the transport driver has attached a sense buffer to `scb->sense_data`, the `sense_data`, the buffer must remain valid until the completion function returns. The transport driver is forbidden from accessing it, until the completion function returns. The allocation and management of the sense buffer is the responsibility of the transport driver.

The completion functions can be called either in process or interrupt context.

---

**NOTE** The completion functions must be called with any locks held since the SCSI services may call the transport driver's `if_start` entry point before returning.

---

Queuing Functions

The SCSI Services provides routines for managing queues (circularly-linked lists) of `bp`'s; `scsi_enqueue`, `scsi_dequeue`, and `scsi_dequeue_bp`, which have the following definitions:

```c
void scsi_enqueue(struct buf **qp,
  struct buf *bp,
  int where_in_queue);
struct buf *scsi_dequeue (struct buf **qp
  int where_in_queue);
struct buf *scsi_dequeue_bp(struct buf **qp
  struct buf *bp);
```

The `where_in_queue` parameter can have on the following values:

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

The `qp` parameter points to a list header element whose forward and backward pointers in turn point to the HEAD and TAIL element of the list.

The `scsi_enqueue` enqueues the `bp` at the HEAD or TAIL of the circular list pointed to by `qp`.

The `scsi_dequeue` dequeues the `bp` at the HEAD or TAIL of the list pointed to by `qp` and returns the `bp`. NULL is returned when the queue is empty.
The `scsi_dequeue_bp`:

- Dequeues `bp` from wherever it may be in the queue, pointed to by `qp`.
- Returns `bp` when found on the queue.
- Returns NULL when not found on the queue.

**NOTE** Queuing functions must be called with the `scsi_bus` lock held.

**dev_t Mapping Functions**

The SCSI Services provides routines for mapping a `dev_t` into its corresponding bus, target, or LUN instance structure, or into its `isc` structure:

```c
struct scsi_bus *m_scsi_bus(dev_t dev);
struct scsi_tgt *m_scsi_tgt(dev_t dev);
struct scsi_lun *m_scsi_lun(dev_t dev);
struct isc_table_type *m_scsi_isc(dev_t dev);
```

The `m_scsi_bus`, `m_scsi_tgt`, `m_scsi_lun`, and `m_scsi_isc` return a pointer to the `scsi_bus`, `scsi_tgt`, `scsi_lun`, and `isc` structure, that corresponds to `dev`. They return NULL if `dev` is invalid or references a bus, target, or LUN which is not currently open. The `m_scsi_isc` will return a pointer to the `isc` even if the bus isn't open, as long as `dev` refers to a device on a configured bus.

Spinlocks are the most heavily used synchronization mechanism in the HP-UX kernel, and are the basic locking primitive used by the kernel for short-term locks.

**Synchronization Functions**

The SCSI Services provides functions for transport drivers to use to synchronize their access to SCSI Services data structures:

```c
void scsi_tgt_lock(struct scsi_tgt *tp);
void scsi_tgt_unlock(struct scsi_tgt *tp);
void scsi_bus_lock(struct scsi_bus *bustp);
void scsi_bus_unlock(struct scsi_bus *bustp);
```

The `scsi_tgt_lock` obtains the `scsi_tgt` structure's spinlock. The `scsi_bus_lock` obtains the `scsi_bus` structure's spinlock. These locks must be held while accessing the `scsi_tgt` or `scsi_bus` structure. The bus lock must also be held while accessing the “select queue”.

**Miscellaneous Service Functions**

The SCSI Services provides the pSCSI-specific function for use in pSCSI transport drivers:

```c
int scsi_frequency(int sdtr_period);
```

The `scsi_frequency` translates the transfer period factor of an **Synchronous Data Transfer Request** (SDTR) or PPR message into its corresponding Hz frequency. For example, a transfer period factor of 8, which is defined in the SCSI specification (SPI-4) to be Ultra320 speed, returns the value 160,000,000 (160 MHz).

SCSI Services also provides a probe function for use by transport drivers called `parallel_scsi_probe`. The transport driver does not call this routine itself, but can register it as the driver's probe routine with WSIO via a call to `wsio_register_addr_probe` from the driver's install routine, see “Writing a `driver_install()` Routine” and “Writing Driver Probe Routines” in Chapter 4, “Writing a Driver.”
Due to the SCSI-2 paradigm that exists at the SCSI Services level and above in the current HP-UX mass storage stack, the `parallel_scsi_probe` routine is commonly registered as the LUN probing routine by most SCSI transport drivers, (pSCSI or Fibre Channel). See Chapter 11, “Mass Storage Stack Architecture.”

I/O Tree Node Management Functions

The WSIO Services provides functions for a transport driver to create and manage driver-specific I/O Tree nodes. The only expected use for this in HP-UX 11i v1 is for virtual bus, target and LUN management. See Chapter 11, “Mass Storage Stack Architecture,” for additional information on virtual bus, target, and LUN management. See the WSIO Reference Pages in the HP-UX 11i v1 Driver Development Reference Guide, for details on the WSIO I/O Tree Node interfaces, such as `wsio_create_interface` (to register an I/O interface and associated hw path, and return a corresponding `isc`), `wsio_create_attribute` (to create and attach an attribute to an I/O interface), and `wsio_get_relationship` (to get the parents of children of an interface).

To address more than 8 LUNs (for example a SCSI-3 target), a driver has to create virtual SCSI-2 LUNs (under virtual SCSI-2 targets which in turn is under virtual SCSI-2 buses) to represent the large LUN space behind the target. Once the virtual entities are created, communication between the SCSI-2 stack and the SCSI-3 target happens through a virtual SCSI bus layer.

For this scheme to work, a HBA driver has to have an additional virtual bus driver, which intercepts the open or I/O requests from the SCSI-2 stack and converts it to transport specific open or SCSI-3 I/O. The virtual bus driver also has to be scannable (DRV_SCAN set in `drv_info_t` of the driver) and has to have a target probe interface (registered using `wsio_register_dev_probe()`). This probe interface will be responsible for creating the virtual SCSI buses (also referred to as virtual bus nodes) which can be later scanned using HPUX SCSI services. The virtual bus driver for the HBA driver will have to claim the virtual buses created and HPUX SCSI services (```) can be used to scan those virtual SCSI buses.

An example of addressing 1024 LUNs behind a FC SCSI-3 target using the WSIO Interface management functions is given.

In the Fiber Channel world, a target is addressed using a 24 bit address (`N_PortID`). The addressing method which is described in this section is based on the `N_PortID` of the target; in other words, if the `N_PortID` changes for the target, the device file for the LUN also has to change.

As long as the hardware path to the LUN remains the same, the device file corresponding to it will also remain the same. Thus, the hardware path is used to store information. In this scheme, `N_PortID` and `FCP_LUN` is embedded into a hardware path. A LUN hardware path looks like this:

```hlsl
<HBA-hw-path>/<domain>.<area>.<port>.<vbus>.<vtgt>.<vlun>
```

where `vbus`, `vtgt` and `vlun` are taken from the SCC first level LUN ID.

```
->| <- 2 bits addressing

| AD |<------ 14 bit FCP_LUN_0 ----->|

```

```
->| <- 2 bits addressing

9<->7  6<->3  2<->0  Bit position

| AD | vbus | vtgt | vlun |

```

To achieve the previous hardware path for a LUN, HBA driver has to do this.
For each FC target, 3 tiers of nodes are created representing:

- Domain node (upper 8-bits) under the HBA port
- Area node (middle 8-bits) under the domain
- Port node (lower 8-bits) under the area.

Under the port node one or more virtual SCSI bus nodes are created depending upon the number of LUNs behind the target.

Nodes can be created using `wsio_create_interface()`.

Once the virtual bus nodes are created and after `vbus` driver claims it, `parallel_scsi_probe()` takes over the scanning and creates targets and LUNs beneath it.

The HBA driver probe interface is responsible for creating the virtual SCSI buses. Virtual bus driver is responsible for claiming those virtual SCSI bus nodes, intercepting I/Os from SCSI services and converting it to SCSI-3 packets.

In the following code snippets, the interface driver is represented as "scsi_xport" and the virtual SCSI bus driver as "scsi_xport_vbus".

Once the virtual SCSI buses are created, the `scsi_xport_vbus` module would claim those nodes. The `scsi_xport_vbus` driver registers an address probe routine `parallel_scsi_probe()` with WSIO, and the rest of the scanning/discovery (scanning of the virtual SCSI bus) of SCSI targets/LUNs (in this case, virtual targets and virtual LUNS) behind the virtual bus is done by `parallel_scsi_probe()`.

The `scsi_xport` transport driver registers a device probe routine that scans for all the SCSI-3 targets and create virtual SCSI buses (one or more) for all the targets to address the large number of LUNs behind them.

```c
scsi_xport_install()
{
    ...
    if(wsio_register_dev_probe(DRV_NAME, scsi_xport_probe,
                               SCSI_XPORT_NAME) != WSIO_OK) {
        return(0);
    }
    ...
}
```

The following, is a code snippet of the `scsi_xport_probe()` routine that creates virtual buses. The `scsi_xport_vbus` driver manages an attach chain used to claim the virtual bus nodes created by the `scsi_xport_probe` routine.

```
----------
Probe logic
----------
1. Find the number of devices attached to HBA
   (e.g.: from name server query in fabric topologies)
2. For each target
   - Build and send SCSI INQUIRY command.
   - For disk array type of devices,
     . Build and send REPORT_LUNS command
```
. Compute the number of virtual SCSI buses that needs to be created from the response
. Create domain, area and port nodes, if not already created.
. Create virtual buses as needed

---------
static int (*scsi_xport_attach_chain)() = scsi_xport_vbus_attach;

/*
 * Compute the number of virtual SCSI buses that need to be created (by
 * probing the targets/LUNs which can be seen from this HBA port) and
 * return them for every call into this routine; one PROBE_FIRST and
 * multiple PROBE_NEXT calls
 *
 * One possible way of addressing 1024 luns for the SCSI first level LUN
 * (first two bytes of the 64 bit FCP_LUN) is this:
 *
 * ->|    |<- 2 bits addressing
 *    -------------------------------------
 *    | AD |<-------- 14 bit FCP_LUN_0 ------>
 *    -------------------------------------

 * ->|    |<- 2 bits addressing
 *                      9<->7  6<->3  2<->0  Bit position
 *    -------------------------------------
 *    | AD |    VBUS | VTGT | VLUN |
 *    -------------------------------------
 *          |  3   |  4   |  3   | Number of bits
 *
 * where VBUS is the hardware path address for the virtual bus node
 * VTGT is the hardware path address for the virtual target node
 * VLUN is the hardware path address for the virtual lun node
 *
 * VBUS and VLUN nodes are created by parallel_scsi_probe() (the
 * scsi_xport_vbus_if_open() need to map the tgt_id (in struct scsi_tgt) and
 * lun_id (in struct scsi_lun) to bits 3-6 and 0-2 respectively of
 * FCP_LUN_0)
 *
 * The node passed to scsi_xport_probe is the node of the HBA. Initially
 * (PROBE_FIRST) hw_path->last_index is set to point to HBA node element
Writing a SCSI Interface Driver

External Interfaces to a Transport Driver

*(hw_path->addr[hw_path->last_index] == HBA's address) and hw_path->
* first_index is set to hw_path->last_index+1 by the caller.*

```
scsi_xport_probe(
    void                   *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
)
{

    struct scb              *scb;
    struct buf              *bp;
    union inquiry_data      *inq_data;
    wsio_ret_code_t         w_ret;
    struct isc_table_type   *isc_ret;

    /*
    * The isc gotten here is the same isc claimed in scsi_xport_attach().
    * Use isc->if_isc to pass any information from scsi_xport_attach()
    * to scsi_xport_probe()
    */

    bp  = kmalloc(sizeof(struct buf), M_IOSYS, M_WAITOK);
    scb = kmalloc(sizeof(struct scb), M_IOSYS, M_WAITOK);
    inq_data = kmalloc(sizeof(union inquiry_data), M_IOSYS, M_WAITOK);
    /* Allocate and intialize if_scb, if needed */
```

/* sample code for sending SCSI command INQUIRY follows */

bp  = kmalloc(sizeof(struct buf), M_IOSYS, M_WAITOK);
scb = kmalloc(sizeof(struct scb), M_IOSYS, M_WAITOK);
inq_data = kmalloc(sizeof(union inquiry_data), M_IOSYS, M_WAITOK);
/* Allocate and intialize if_scb, if needed */
rsp_len = sizeof(union inquiry_data);
scb->max_msecs = 10*1000; /* 10 seconds timeout */
scb->cdb[0] = CMDinquiry;
scb->cdb[4] = rsp_len;
scb->cdb_len = 6;

bp->b_scb = scb;
bp->b_flags = B_SCSI_CMD | B_READ;
bp->b_bcount = bp->b_resid = rsp_len;
bp->b_un.b_addr = (void *)inq_data;

/* Call interface drivers IO start routine here to send this IO */

/*
 * Once all the targets and LUNs behind the targets are scanned,
 * build an array/tree of data structures containing the N_PortID
 * of all FC targets and for each N_PortID, the number of virtual
 * buses (with its hardware path address) under it.
 *
 * Remember, the objective of this probe routine is to create
 * DOMAIN, AREA, PORT and VBUS nodes only. Once the VBUS node is
 * created, parallel_scsi_probe() takes over the node, once
 * xx_vbus module claims it, and do the rest of the scanning.
 *
 * The order in which the nodes created are as follows:
 * 
 * / parallel_scsi_probe kicks
 * .-- | off as soon as ext_bus
 * |   | nodes are created, which
 * |   | is responsible for the
 * V   | creation of "target" and
 * DOMAIN-1, AREA-1, PORT-1, EXT_BUS-1  \ 'lun' nodes
 * :  EXT_BUS-N
 * :  PORT-N, EXT_BUS-1
 * :  EXT_BUS-N
 * :  AREA-N, PORT-1, EXT_BUS-1
 * :  
*/
Writing a SCSI Interface Driver

External Interfaces to a Transport Driver

*        EXT_BUS-N
*        PORT-N, EXT_BUS-1
*        :
*        EXT_BUS-N
*        :
*        PORT-N, EXT_BUS-1
*        :
*        EXT_BUS-N
*        :
*        PORT-N, EXT_BUS-1
*        :
*        EXT_BUS-N
*        :
*        PORT-N, EXT_BUS-1
*        :
*        EXT_BUS-N
*        :
*        PORT-N, EXT_BUS-1
*        :
*        EXT_BUS-N
*        :
*        AREA-N, PORT-1, EXT_BUS-1
*        :
*        EXT_BUS-N
*        PORT-N, EXT_BUS-1
*        :
*        EXT_BUS-N
*        /

/ *
* Assuming a routine scsi_xport_get_next() returns the elements
* in the above mentioned order, the IO tree nodes corresponding
* to the DOMAIN, AREA, PORT and VBUS (or ext_bus) can be
* created like this.
* *
* #define DOMAIN  1
* #define AREA    2
* #define PORT    3
* #define VBUS    4
*
* Also assume scan_elem is of the following structure type
* *
* struct scan_element {
*     int     nport_id;  /* 24 bit nport id */
*     int     type;      /* DOMAIN, AREA, PORT or VBUS */
*     uint8_t addr;     /* hardware path address for this element */
* } scan_elem;
*/
while(scsi_xport_get_next(&scan_elem)) != NULL) {
    switch(scan_elem.type) {
    case DOMAIN:
        hw_path->last_index = hw_path->first_index;
        hw_path->addr[hw_path->last_index] = scan_elem.addr;

        w_ret = wsio_create_interface(isc, hw_path, WSIO_TRANS,
                                     NULL, NULL, NULL, "", &isc_ret);

        if (w_ret != WSIO_OK) {
            return(PROBE_UNSUCCESSFUL);
        }

        return(PROBE_SUCCESS);
        break;

    case AREA:
        hw_path->last_index = hw_path->first_index + 1;
        hw_path->addr[hw_path->last_index] = scan_elem.addr;
        /* create transparent node as in DOMAIN */

        return(PROBE_SUCCESS);
        break;

    case PORT:
        hw_path->last_index = hw_path->first_index + 2;
        hw_path->addr[hw_path->last_index] = scan_elem.addr;
        /* create transparent node as in DOMAIN */

        return(PROBE_SUCCESS);
        break;

    case VBUS:
        hw_path->last_index = hw_path->first_index + 3;
        hw_path->addr[hw_path->last_index] = scan_elem.addr;
        /*
         * set
         * probe_id as : a 16(max) byte identifier
         * (on subsequent ioscans, if this
* differs, iosc an output shows the
* element as DIFF_HW).
* name : a name for the io tree node
* desc : description for the node (this appears under
*    (Description in iosc an output)
*/

w_ret = wsio_create_interface(isc, hw_path,
    WSIO_INTERFACE, probe_id, name, desc, "", &isc_ret);

w_ret = wsio_sizeof_attribute(isc_ret, "__wsio_card_parms",
    &size);

if (w_ret != WSIO_OK) {
    /* create card_parms property for the node */

    bus_info = kmalloc(sizeof(xx_bus_info_t),
        FCD_WAITOK);

    bzero((caddr_t)bus_info, sizeof(fcd_pb_bus_info_t));
    bzero((caddr_t)&card_parms,
        sizeof(struct wsio_card_parms));

    card_parms.slot_num     = -1;
    card_parms.if_flags     = 0;
    card_parms.bus_info     = (caddr_t)bus_info;
    card_parms.bus_type     = PCI_BUS;
    /* slot_id is passed to scsi_xport_vbus_attach as id */
    card_parms.slot_id      = SCSI_XPORT_VBUS;
    card_parms.attach       = &scsi_xport_attach_chain;

    /*
     * Initialize scsi_xport_bus_info_t here (to pass
     * information to scsi_xport_vbus module
     */
    w_ret = wsio_create_attribute(isc_ret,
        "__wsio_card_parms",
        (uintptr_t)&card_parms,
        sizeof(struct wsio_card_parms),
        WSIO_COPYDATA);

    if (w_ret != WSIO_OK) {
The following code snippets represent the `scsi_xport_vbus()` module that manages the virtual bus nodes.

```c
/* This structure contains pointers to all driver entry points */
static drv_ops_t xx_vbus_ops = /* no interface via [bc]devsw */
{
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    0,
};

/* This structure is used by GIO & WSIO to display properties.
 * The driver here is scsi_xport_vbus driver with class as ext_bus
 * and the device flags are that the scsi_xport_vbus driver supports
 * scanning, deferred scanning and is MP SAFE.
 */
static drv_info_t scsi_xport_vbus_info =
{
    "scsi_xport_vbus",
    "ext_bus", /* the SCSI-2 bus class */
    DRV_SAVE_CONF | DRV_SCAN | DRV_DEFER_SCAN | DRV_MP_SAFE,
};
```
The following routine registers the `scsi_xport_vbus()` driver with WSIO. It also registers an address probe routine that enumerates the SCSI LUNs behind a target.
scsi_xport_vbus_install(void)
{
    int ret;

    /* Call wsio_install driver to register with wsio */
    install_ret = wsio_install_driver(&scsi_xport_vbus_wsio_info);
    if (install_ret != WSIO_OK) {
        return 0;
    }

    /* Register the parallel_scsi_probe as the probe routine. */
    wsio_register_addr_probe(parallel_scsi_probe, "scsi_xport_vbus");
}

The scsi_xport_vbus() module's attach routine is used to claim
the virtual bus nodes created by the scsi_xport driver.

int
scsi_xport_vbus_attach(int id, struct isc_table_type *isc)
{
    /*
     * id is the same as slot_id programmed in struct wsio_card_parms when
     * creating this vbus node. This can be used to identify the type of
     * the virtual scsi_bus node which we have created before (this is
     * needed only if we create multiple types of virtual scsi_buses
     */
    /* do "id" validation here if needed */
    :

    /* isc->bus_info is the same pointer, which was programmed as
     * wsio_card_parms's bus_info pointer, when this node was created
     */
    scsi_xport_bus_info = (scsi_xport_bus_info_t *) isc->bus_info;

    /* Initialize out scsi_xport_vbus_init routine with the isc's gfsw */
    *(isc->gfsw) = scsi_xport_vbus_gfsw;

    /* bind this driver to this node by calling isc_claim */
    isc_claim(isc, &scsi_xport_vbus_wsio_info);

    /* Set the bus_max_width to 16 for SCSI probe */
    isc->bus_max_width = 16;
/ for scsi_frequency() calc */
isc->bus_min_sdtr_period = 2;

/* Call the next driver in the chain, if there are multiple drivers
* which can possibly claim this node. If there is only one driver
* which can claim this node, call wsio_last_attach() directly.
*/
return (wsio_last_attach(id, isc));

The following routine shows the initialization steps for the enumerated virtual bus nodes.

static int
scsi_xport_vbus_init(struct isc_table_type *isc)
{

/*
 * So long as the hardware path to this virtual SCSI bus remains
 * the same, the instance number also would remain the same (actually,
 * the bus part (cXXX) of the instance number) guranteeing the same
 * device file for the lun.
 */
int bus_id = wsio_isc_to_instance(isc, NULL);

if (bus_id > SCSI_MAX_BUS_ID) {
    printf("WARNING: Maximum possible scsi_bus limit has reached\n",
           "No more LUNs will be seen from scsi_xport driver\n");
    return WSIO_ERROR;
}
scsi_isc[bus_id] = isc;

scsi_xport_ifsw = kmalloc(sizeof(struct scsi_ifsw), M_IOSYS, M_WAITOK);
/* initialize our ifsw */
bzero((caddr_t) scsi_xport_ifsw, sizeof(struct scsi_ifsw));

/* set IF_B2_LIST too if capable of disk sort merge buffer request. */
scsi_xport_ifsw->if_flags = IF_BUS_TAGS;

scsi_xport_ifsw->if_max_tag = 254;

scsi_xport_ifsw->if_scb_size = sizeof(scsi_xport_io_ccb_t);
scsi_xport_ifsw->if_lun_size = sizeof(scsi_xport_scsi_lun_t);
Writing a SCSI Interface Driver

External Interfaces to a Transport Driver

```c
scsi_xport_ifsw->if_open = scsi_xport_vbus_if_open;
scsi_xport_ifsw->if_close = scsi_xport_vbus_if_close;
scsi_xport_ifsw->if_start = scsi_xport_vbus_if_start;
scsi_xport_ifsw->if_max_io_size = 1024 * 1024; /* 1MB is the max */

/* Set our scsi_xport_ifsw to isc's ifsw */
isc->ifsw = (caddr_t)scsi_xport_ifsw;

return(WSIO_OK);
}
```

The following routines show the `if_open`, `if_start` and `if_close` entry points in the `if_sw` table.

```c
scsi_xport_vbus_if_open(dev_t dev)
{
    /*
    * Call interface driver's open routine to establish nexus, if
    * it is not already established
    */

    /*
    * Make use of isc->bus_info to pass any information (like per
    * instance interface data structure etc) from probe
    * isc can be retrieved as follows:
    *
    * struct scsi_bus *busp = m_scsi_bus(dev);
    * struct isc_table_type *isc = busp->isc;
    */

    /*
    * Additionally, can convert the SCSI-2 tgt_id and SCSI-3 lun_id to
    * FCP_LUN (the reverse mapping) and could store the information in
    * if_lun and later retrieved for I/Os
    */
}
```

```c
scsi_xport_vbus_if_start(struct isc_table_type * isc)
{
    struct scsi_bus *busp = (struct scsi_bus *) isc->if_drv_data;

    /* Dequeue request packets (bp) from busp->select_q */
```
/* Find the HBA port instance from isc->bus_info */

/* retrieve the FCP_LUN from if_lun */

/* Issue the IO request to the hardware */

}

scsi_xport_vbus_if_close(dev_t dev)
{
    /* Close the nexus on last close to the device */
}
External 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 transport driver, i.e., maintained only by the transport driver (after being initialized to zero by services). These exist because some part of the SCSI subsystem other than the transport driver needs access to the information, and source of this information is the transport driver. It could be services or a device driver that needs access to the information, but that is irrelevant to the transport driver.

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

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

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

**SCSI Control Block**

The \texttt{buf} structure is not large enough to hold all state information associated with a SCSI I/O attempt. An \texttt{scb} is attached to a \texttt{buf} by SCSI services to hold the temporary state information until the I/O is completed. The \texttt{bp/scb} association does persist for retries. The \texttt{struct buf} is described later in the section.

Fields that are of interest to a transport driver writer are:

```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
```

The transport driver is \textit{not allowed} to set the following fields:

- \texttt{lp}
- \texttt{flags}
- \texttt{max_msecs}
Writing a SCSI Interface Driver

External Data Structures

- `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, 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 transport 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 transport driver may check these bits in the flag for proper functionality. Bits in `scb->flags` that are relevant to a transport 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_SDTR` is not set, the transport driver should initiate SDTR negotiation immediately following the Selection, Identify or tag message, whichever comes last, and before sending the **Command Description Block (CDB)** for the I/O.

- **SCB_WDTR**
  This bit directs the transport 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 transport driver should initiate SDTR negotiation immediately following the **Wide Data Transport Request (WDTR)** negotiation. The wide negotiation should always precede the synchronous negotiation, since a wide negotiation resets the link to asynchronous.

- **SCB_4BYTE**
  This bit informs the transport 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 needs to be considered only if the amount of data transferred may be less than that requested in \(bp->b_bcount\). Note that \(SCB_{4\text{BYTE}}\) 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.

\(SCB_{2\text{BYTE}}\)  
This bit is the same as \(SCB_{4\text{BYTE}}\) except that phase changes are only restricted to even boundaries.

\(SCB_{\text{ORDERED\_TAG}}\)  
Denotes that ordered tags are intended to be used for this device.

\(scb->max\_msecs\)  
Minimum number of milliseconds the transport 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 transport driver is encouraged to abort the I/O with Abort or Abort Tag as appropriate. The transport driver can run a timer routine periodically to watch for the I/O’s that are timed out. A value of zero indicates the transport 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 buses.

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

\(scb->cdb\_status\)  
Indicates the status of the I/O command. If the I/O attempt completes with no phase sequencing errors and without being aborted or timed out, the transport driver sets \(scb->cdb\_status\) to \(S\_GOOD\). If the selection phase times out, the transport 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 transport 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 \(S\_CHECK\_CONDITION\) to request an auto-sense request. \(scb->cdb\_status\) must be set by the transport driver prior to returning the \(bp\) via \(scsi\_cbfn\).

Refer to \(scsi\_h\) for the valid values of \(cdb\_status\).
If the I/O attempt completes with no phase sequencing errors and without being aborted or timed out, the transport driver sets `scb->data_resid` so 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, 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 transport 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 timed out, the transport 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 `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 transport 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 `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 transport 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 timed out, and if `scb->sense_status` is not zero, the transport driver sets `scb->sense_data`. Otherwise, `scb->sense_data` is undefined and will not be referenced by the SCSI subsystem on callback. The transport 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 transport driver must not modify the buffer for the duration of `scsi_cbfn`. When `scsi_cbfn` returns, and not until, the transport 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 entry, defined as an isc_table_type structure, is used by WSIO to maintain transport driver information. Refer to the Chapter 2, “HP-UX I/O Subsystem Features,” 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 transport driver as specified by the SCSI subsystem. They are described in this section. Others are reserved for use by the transport driver at its discretion. These reserved fields are - ppoll_flag, ppoll_mask and ppoll_sense. However, these fields are not typically used by the transport 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 transport driver are:

```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_regack_offset tfr_control`

`#define tgt_wdtr_done ppoll_f`

`#define tgt_wdtr_width ppoll_l`

`#define tgt_sdtr_done lock_count`

`#define tgt_sdtr_period event_f`
Writing a SCSI Interface Driver

External Data Structures

#define if_char0 ppoll_flag
#define if_uchar1 ppoll_mask
#define if_uchar2 ppoll_sense

NOTE

The transport 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 transport driver's attach routine.

isc->gfsw

Pointer to the transport driver’s gfsw structure. The SCSI subsystem does not require that the transport driver provide a gfsw structure.

isc->ifsw

Pointer to the transport driver's scsi_ifsw structure. It is initialized by the transport 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 transport 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; that is, 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 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 the bus is not open. SCSI services set this field and a transport driver is not allowed to write to this field.

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

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

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

Transport Driver Switch

This structure defines SCSI transport driver entry points and parameters as required by SCSI services. The transport 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 transport driver entry points and operational parameters to the SCSI subsystem. A detailed description of the fields are:

```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  
Transport 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**  
  Transport driver does not support tags.
- **IF_B2_LIST**  
  If set, it indicates the transport driver supports handling of disk sort merge buffers.
- **IF_OWNS_TAGS**  
  Transport driver owns tagged queueing.

ifsw->if_max_tag  
One less than the number of per-bus tags supported by the transport 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 transport 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/O's to the bus at any given time (this
includes untagged I/O's).

ifsw->if_scb_size
The number of bytes the SCSI subsystem shall allocate and attach to each
scb for use by the transport driver. The if_scb field of the 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.

ifsw->if_lun_size
The number of bytes the SCSI subsystem allocates and attaches to each
scsi_lun structure for use by the transport driver. The if_lun field of
scsi_lun structure is a pointer to ifsw->if_lun_size bytes for the use
of the transport driver.

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

ifsw->if_bus_size
The number of bytes the SCSI subsystem allocates and attaches to each
scsi_bus structure for use by the transport driver. The if_bus field of
scsi_bus structure is a pointer to ifsw->if_bus_size bytes for the use
of the transport driver.

ifsw->if_open
Pointer to the transport driver's logical unit close function. This is optional
for a transport driver.

ifsw->if_start
Pointer to the transport driver's start function.

ifsw->if_reset_bus
Pointer to the transport driver's Bus Reset function. This is optional for a
transport driver.

ifsw->if_bdr
Pointer to the transport driver's Bus Device Reset function. This is
optional for a transport driver.

ifsw->if_abort
Pointer to the transport driver's Abort function. This is optional for a
transport driver.

ifsw->if_io_max_size
Maximum size of I/O request supported by the transport driver. A value of
zero specifies no limit. If set, I/O requests for more than the supported size
will be erred back by the SCSI services.

ifsw->if_beg_align, ifsw->if_end_align
Transport 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 Structure
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 transport driver in the form of buf structure. Some of the fields that are of
interest to a transport driver writer are explained here:

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;
devid_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

The transport 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

bp->b_flags

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

---

**NOTE**

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

Transport drivers must do I/O’s 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 transport 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

bp->b2_flags

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

bp->b_merge

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 Number of Bytes per Page (NBPG).

bp->b_merge_cnt

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

bp->b_spaddr

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

bp->b_scb (or b_s2)

This field is a pointer to the scb associated with this bp. It is used in a transport 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. It is owned by the SCSI services, but can be accessed by a transport driver. Only the fields relevant to a transport 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
```
The transport driver is *not allowed* to set the following fields:

- **tgt**
- **lun_id**
- **dev_minor**
- **open_cnt**
- **state**

```c
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.

```c
lp->if_lun
```

Pointer to the data area allocated and zeroed on first open, by the SCSI subsystem, for use by the transport driver. Its size is specified by the transport 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.

```c
lp->lun_id
```

The logical unit’s identification number.

```c
lp->dev_minor
```

Minor number of the device minus the volume bits i.e., the device number independent portion of the minor number.

```c
lp->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.

```c
lp->state
```

`L_TAGS` is the only bit of `lp->state` that is defined for the transport driver; all other bits are undefined. The transport 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/O's 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 a transport driver:

```c
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 *lun[SCSI_MAX_LUN_ID+1];
} *tp;
```

### Definitions

```c
#define T_ENABLE_WDTR 0x40
#define T_ENABLE_SDTR 0x20
```
The transport driver is *not allowed* to set the following fields:

- `open_cnt`
- `state`
- `tgt_id`
- `min_sdtr_period`
- `bus`
- `lun`

**tp->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 this target.

**tp->state**

Two bits are defined. `T_ENABLE_WDTR` directs the transport driver to initiate SDTR negotiation after any event other than WDTR negotiation that causes the data transfer width to be reset to eight bits that is, bus reset or BDR. If `T_ENABLE_WDTR` is not set, the transport driver is forbidden to initiate WDTR negotiation, but not to respond.

`T_ENABLE_SDTR` directs the transport driver to initiate SDTR negotiation after any event other than SDTR negotiation that causes the synchronous data transfer parameters to be reset to asynchronous that is, bus reset, BDR or WDTR negotiation. If `T_ENABLE_SDTR` is not set, the transport driver is forbidden to initiate SDTR negotiation, but not to respond.

**tp->tgt_id**

SCSI ID of this target.

**tp->min_sdtr_period**

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 transport 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 transport 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 transport driver should initiate SDTR negotiation to correct the situation.

**tp->bus**

Pointer to the target's parent `scsi_bus` structure in the open device tree.

**tp->if_tgt**

Pointer to the data area allocated and zeroed on first open by the SCSI subsystem for use by the transport driver. Its size is specified by the transport 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 for each 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 a transport driver:

```
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;
```

The transport 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. This 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.
- `busp->if_bus` Pointer to the data area allocated and zeroed on first open by the SCSI subsystem for use by the transport driver. Its size is specified by the transport 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 this 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 transport driver when calling scsi_enqueue/scsi_dequeue/scsi_dequeue_bp. The 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/O's ready for selection. The transport driver picks up I/O requests from this queue.
select_q Structure

This is the only data structure that is shared between the SCSI subsystem and the transport driver — both may change it. It is the per-bus select queue as shown:

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

After initializing the bp and its scb and zeroing *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 transport driver returns the bp via `scsi_cbfn`, nor will the SCSI subsystem remove a bp from the queue once it has enqueued it.

The transport 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 transport driver must initiate all I/O's for any logical unit in the order 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 transport 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 transport driver does not access the queue in any other way than through the access functions provided by the SCSI subsystem as mentioned in this section.

The transport 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 a transport driver, the diagram in Figure 12-1, “SCSI Data Structures,” illustrates the inter-relationship between different data structures in the SCSI subsystem and other kernel data structures.
Figure 12-1  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 are enqueued by the device driver's `dd_strategy` routine and dequeued by the PD's `dd_start` routine.

- **tag queue**: Per-bus queue that contains requests waiting for a qtag. Requests ready to be retried (that is, requests on the retry queues that have hit their "time-to-retry") are placed on the tag queue to be restarted; 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 to avoid exceeding 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" (that is, they're ordered by the time when the request will be retried).

The following Figure 12-2, "I/O Flow Between Various Queues," illustrates all the queues involved in a typical I/O path.
I/O Flow Between Various Queues

I/O Winin SCSI Device Driver

The SCSI device driver is responsible for enqueueing the incoming I/O requests in its queue. Device drivers typically call `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 required by the transport driver.
- 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
The 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 nonzero status to indicate an error. If I/O is queued successfully, SCSI services act on it for further processing.

### I/O Within SCSI Service Layer

The SCSI service layer is responsible for:

- Passing I/O requests from device driver to transport driver.
- Implementing flow control policies to honor device I/O queue depth.
- Ensuring fair distribution of shared bus resources between different LUNs (tags, scbs).
- Keeping track of I/O time.
- Handling 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 `busp->tag_q`. If nexus resources are not available the buffer is queued to `lp->nexus_q`. If all the resources are allocated SCSI services place the buffers to `busp->select_q`. The transport driver is called through the `ifsw.if_start` entry point for processing all the I/O requests queued on its `select_q`.

Transport drivers return completed I/O requests to SCSI services by calling `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 `busp->retry_q`.

The SCSI services layer owns the following queues:

- `busp->tag_q`: SCB is allocated and I/O is initialized; waiting for per bus resource.
- `lp->nexus_q`: I/O request got SCB and tag resources; waiting for nexus resource.
- `busp->retry_q`: I/O is being retried on failure; tag and nexus resources are fixed. Waiting for the timeout period to be queued.
- `busp->select_q`: I/O is enqueued to the interface layer. I/O is owned by the transport driver until it is returned to SCSI services by calling `scsi_cbfn()`.

### I/O Within SCSI Interface Layer

The transport driver’s `if_start` routine must dequeue each I/O on its `select_q` and do whatever is required to execute the I/O request. The SCSI transport driver is responsible for:

- Executing the I/O request.
- Timing the I/O request as requested by upper layers.
- Returning I/O to SCSI services by calling `scsi_cbfn()`.
Transport Driver Development

Developing a transport 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 transport driver must have a `driver_install` routine and a `driver_attach` routine. If the transport 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 Chapter 4, “Writing a Driver.”

The following sections explain the specific transport 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 transport 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 in 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 transport 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 qllsp_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 */
```
Writing a SCSI Interface Driver

Transport Driver Development

```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 */
};
static wsio_drv_data_t qlisp_wsio_data = {
    "scsi", /* drv_path */
    T_INTERFACE, /* drv_type */
    DRV_CONVERGED, /* drv_flags */
    NULL, /* dvr_minor_build - field not used */
    NULL, /* dvr_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 used by 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. If a device is behind a bus for which there is no bus nexus CDIO in the OS, it is the responsibility of the driver to let WSIO notify 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 Chapter 4, “Writing a Driver.”
Calling driver_attach()

Use the driver_attach() routine to determine whether the product ID passed in matches the
driver_attach 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 a driver
initialization routine and finally calls isc_claim() to claim the device.

SCSI Specific Part

A SCSI transport 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 transport driver is to attach a fully initialized scsi_ifsw structure to the
isc structure via isc->ifsw field. This is typically performed in the transport 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 transport driver.

The transport 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 is provided
later in this chapter.

Device initialization, allocating and mapping SCSI command queues, registering device interrupts, and so
forth, 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 driver_init() routine are shown:

```c
#define QLISP_ID_SUPPORTED(x) ((x) == QLISP_ID_1020 ||
(x) == QLISP_ID_1040 ||
(x) == QLISP_ID_1080 ||
(x) == QLISP_ID_1240 ||
(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 */
  /* Allocate driver control structures */
```

Chapter 12 323
/ * 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. This information is kept to the following fields of the \texttt{isc} structure:

- \texttt{isc->my_address}
- \texttt{isc->bus_max_width}
- \texttt{isc->bus_min_sdtr_period}
- \texttt{isc->bus_max_reqack_offset}

For additional details on these fields, refer to the discussion on the \texttt{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. To protect drivers from these changes, the DDK includes the \texttt{SCSI\_GET\_INITIATOR\_PARAMS} macro. It wraps the processor dependent calls and passes back the values of the SCSI parameters. For correct operation of the macro, do not make any modifications to it.

\textbf{NOTE} Include the file \texttt{scsi\_params\_macro.h} distributed with the \textit{Driver Development Kit} to access this macro.

\texttt{SCSI\_GET\_INITIATOR\_PARAMS}

This macro is called with four arguments: a pointer to the \texttt{isc} structure, and place holders for the initiator ID, SDTR period, and REQ/ACK offset. The last three parameters to the macro correspond to the following fields in the \texttt{isc}:

- \texttt{my_address}
- \texttt{bus_min_sdtr_period}
- \texttt{bus_max_reqack_offset}

\textbf{NOTE} \texttt{SCSI\_GET\_INITIATOR\_PARAMS} does not set the fields in the \texttt{isc} structure; it just returns the values of the corresponding fields.

The following pseudo code of a transport driver's init routine illustrates the calling convention of the \texttt{SCSI\_GET\_INITIATOR\_PARAMS} macro and how to set the fields in the \texttt{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 *t o8o r1 6*/

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. A transport 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 transport driver may force a negotiation on SDTR and WDTR.

Current Negotiated Information

The currently negotiated synchronous data transfer period per target is stored in one byte of the tgt_sdtr_period array. A pointer to the byte associated with a target is kept in the scsi_tgt structure while the device is open for easy access, and 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 transport 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 transport 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 transport driver also clears the appropriate tgt_sdtr_done bit when WDTR is negotiated. The transport driver always initiates 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 also initiates WDTR or SDTR whenever SCB_WDTR or SCB_SDTR is set in the scb independently of the bits in tp->state. It initiates WDTR and/or SDTR negotiation on every auto-sense if enabled by tp->state. The transport driver initiates 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:

1. tp->min_sdtr_period
2. isc->bus_min_sdtr_period
3. The lowest period at which the hardware can operate.
When initiating SDTR, the transport driver sends a period equal to the maximum of \( tp->\text{min_sdtr\_period} \) and the minimum period supported by the hardware. The transport driver always negotiates for the maximum \( \text{REQ/ACK} \) offset allowed by the hardware.

Unlike for SDTR, there is no way to control bus width negotiation; 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**

The following is a brief discussion of PCI register mapping. See *Chapter 14, “Writing PCI Device Drivers,”* for a detailed description.

For WSIO drivers, the `if_reg_ptr` member of the `isc` structure is a **Precision Architecture** (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 maps the range itself.

WSIO provides a number of services for DMA mapping. Refer to the manpages in the *HP-UX 11i v1 Driver Development Reference Guide* on WSIO services for a detailed discussion of each. In a transport driver, `dma_setup()` or `bp_dma_setup()` is called to set up DMA mapping, depending on whether one `buf` structure or multiple `buf` structures merged together.

### Fields relevant only to a transport 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`.
     ```c
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. `dma_parms->dma_options` = `((bp->b_flags & B_READ) ? (DMA_8BYTE|DMA_READ) : (DMA_8BYTE|DMA_WRITE))`

2. If `bp->bp_merge_cnt` equal zero, call `dma_setup()`.
   - Set the address of the buffer to be mapped. `dma_parms->addr`
   - Set the space address of the buffer to be mapped. `dma_parms->spaddr` = `bp->b_spaddr`.
   - Set the buffer length to be mapped. `dma_parms->count` = `bp->b_count`.
     ```c
     ret_code = dma_setup(isc, dma parms);
     ```
On a successful call to `dma_setup()`, the `chain_ptr` in the `dma_parms` structure points to an `iovec` structure that contains the mapped address and the length of the mapping. If there are multiple elements involved, the `chain_count` field is used to obtain the number of DMA elements.

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

On a successful call to `bp_dma_setup()`, `bp_dma_parms->merge_dma_parms` will point to a chain of the `dma_parms` structures, and for each of these `dma_parms` structures, the fields explained previously for `dma_setup()` are applicable.

```c
ret_code = bp_dma_setup(isc, bp, bp_dma_parms);
```

**Protection and Synchronization**

The major synchronization issue with transport 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. A spinlock scheme may be chosen to provide finer granularity locks, protecting data structures at finer levels. More information on spinlocks refer to Chapter 3, “Multiprocessing.”

Whenever a transport driver enqueues or dequeues `bp`'s from the `select_q`, it has to acquire `scsi_bus_lock()` and release the lock after the completion of `scsi_enqueue()`, `scsi_dequeue()` or `scsi_dequeue_bp()` by calling `scsi_bus_unlock()`.

For example:

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

Similarly,

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

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 transport driver while allocating new spinlocks.

For example:

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

    ....

    lock_t *mp_lock;

    ....
} qlisp_shared_isc_t;
```

The driver does not hold spinlocks across calls to different subsystems. Before calling the SCSI services, the driver should release the spinlocks and acquire them if necessary when the transport driver reestablishes control.
Sending SCSI I/O Requests to the Target

The SCSI subsystem requires the transport driver to specify a start function. This is the entry point for an I/O request at the transport driver. The following pseudo code is for a typical transport 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 transport 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 transport 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 the transport driver to set the fields in the SCB structure to provide the correct result to the SCSI subsystem.

For details of the fields in the SCB structure, refer to the “SCSI Control Block” section. The following pseudo code defines the steps required to set up the fields of the transport 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:
   ```c
   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 ther, 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_GOTSENSE) {
     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))
        (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 transport driver after calling scsi_cbfn(), the transport driver does not touch any of the resources of the completed I/O. The transport driver forms a chain of bp’s whose I/O has completed. To build such a chain, the transport driver uses the av_forw field of the buf structure. The following code example demonstrates this point:

- bp->av_forw = bp_chain;
- bp_chain = bp;

If the bp_chain initially 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

Refer to Chapter 4, “Writing a Driver,” as it discusses the issues involved in writing an Interrupt Service Routine (ISR). An ISR is a device specific routine and different drivers do different things when handling a device interrupt. Some generic steps are provided for processing a device interrupt:

1. Acquire a spinlock for protection on management processor systems.
2. Read a device status register.
3. Check for 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 the completed I/O's.
7. Set the device register to indicate if an I/O response is processed.
8. Loop if there are more completed commands.
9. Wake up any I/O requests waiting for device resources.
10. Release the spinlock.
11. Call SCSI callback services on completed I/O's.
12. Return INTR_SERVICED

Special Routines

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

Bus Device Reset

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

This function is passed with a buf structure bp as its sole argument. The bp->b_dev field contains the device number of the target. The SCSI subsystem provides m_* services to obtain the required SCSI data structures for executing a BDR.

Pseudo code for a typical BDR routine:

1. Obtain the device number from bp->b_dev.
2. Obtain the scsi_bus structure from m_scsi_bus(bp->b_dev).
3. Obtain the scsi_tgt structure from m_scsi_tgt(bp->b_dev).
4. Obtain the isc structure from the bp->isc.
5. If the device is in reset state, reject the request.
6. Obtain 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 transport driver's abort function in response to an \texttt{SIOC_ABORT} ioctl request. This function is passed with a \texttt{buf} structure \texttt{bp} as its sole argument. The \texttt{bp->b_dev} field contains the device number of the destination I/O. The SCSI subsystem provides \texttt{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 the device number from \texttt{bp->bp_dev}.
2. Obtain the \texttt{scsi_bus} structure from \texttt{m_scsi_bus(bp->b_dev)}.
3. Obtain the \texttt{scsi_tgt} structure from \texttt{m_scsi_tgt(bp->b_dev)}.
4. Obtain the \texttt{scsi_lun} structure from \texttt{m_scsi_lun(bp->b_dev)}.
5. Obtain the \texttt{isc} structure from the \texttt{bp->isc}.
6. If the device is in reset state, reject the request.
7. Acquire 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 transport driver's \texttt{reset_bus} function to reset the SCSI bus. This function is passed with an \texttt{isc} structure as its sole argument.

Pseudo code for a typical \texttt{reset_bus} routine:

1. Obtain 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 \texttt{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 \texttt{RESET_DELAY} state.
8. Set a \texttt{timeout} after 3 seconds.
9. Release the driver lock.
10. After the \texttt{timeout} sets, reset the driver state.
11. Call the driver's \texttt{if_start()} function to process I/O requests.

Handling Timeouts

It is a common practice in driver development to use time-outs to schedule events after a pre-determined time interval. The \texttt{timeout()} kernel service can be used to schedule an event to occur after a specified amount of time. An \texttt{untimeout()} kernel service is also provided to unschedule a time-out. It isparticularly important to \texttt{untimeout()} any pending time-outs before unloading a DLKM driver. Refer to Chapter 2, "HP-UX I/O Subsystem Features," for more information on the \texttt{timeout()} and \texttt{untimeout()} services.
Sleep and Wakeup Mechanism

At times, a transport driver may need to wait for an event such as availability of resources or device establishment of a known state. The transport driver typically calls “sleep”, if it can, from the current context, instead of busy waiting and monopolizing the CPU. When a corresponding event occurs, the driver calls “wakeup” to notify any sleeping threads of this event.

To avoid any race conditions, the kernel spinlock() services provide the get_sleep_lock() call. A get_sleep_lock() obtains a lock before calling sleep and releases the lock after it calls sleep. Similarly, a wakeup() call is also protected with the locks obtained from get_spin_lock(). For a more detailed explanation, refer to Chapter 3, “Multiprocessing.”

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 transport driver for Qlogic's ISP12160 is provided along with this document as part of the Driver Development Kit. 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 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 First In First Out (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 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". This kind of two-driver architecture is required only for special devices with multiple SCSI channels behind a PCI function. 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 an 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. 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, the 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; 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 in Figure 12-3, "Driver Data Structure Inter-Relationship."

---

**Figure 12-3**  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 transport driver. The installation and initialization routines of the *qlisp_multi* driver are similar to the routines explained in Chapter 4, “Writing a Driver,” of this manual. There are some differences that are specific to the functionality of this driver and they are explained in the following sections.

The *qlisp_multi* driver claims the interface card and scans for SCSI controller chips underneath it. The driver should be able to 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 performed in the `qlisp_multi_install()` and `qlisp_multi_attach()` routines. The `DRV_SCAN` flag is set in the `drv_info_t` structure to inform the WSIO that the driver can scan.

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

Fragments 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_NAME_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 qlisp_multi_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 are:

```c
qlisp_multi_identify_child(child_name);
```
Writing a SCSI Interface Driver

Routines

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

These routines are required only for a multi-port interface card.

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 the 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;
    char    last_index; // unsigned
    char    addr[MAX_ELEMENTS];
} hw_path_t;
```

The device probe routine is called with the following arguments:

- Pointer to the calling I/O tree node
- Pointer to the driver info structure
- `probe_id`, which is set by the transport driver
- Hardware path of the calling device
- Pointer to the `isc` structure
- `probe_type`
- Pointer to name of the child node.
- 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:

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

When a device is found, the driver increments the `last_index`, and it becomes equal to the `first_index`; the probe function places the hardware address of the 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`. 

---

338

Chapter 12
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, for example, “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 sense when the address range is out of range and should return an error.

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;
        hw_path->addr[hw_path->last_index] = target;
    }

    GENLOG (probe_type, target, found, hw_path->last_index, 0);
} while ((probe_type != PROBE_ADDRESS) && (found == NO_DEV));
return ((found == VALID_TGT) ? PROBE_SUCCESS :
        PROBE_UNSUCCESSFUL);
}
```
qlisp Driver

This section presents the code flow of the qlisp driver. The qlisp_multi driver is a typical WSIO transport driver. The installation and initialization routines of the driver of the qlisp_multi driver are similar to the routines explained in Chapter 4, “Writing a Driver.” 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. To inform the WSIO that it can scan for the hardware.
2. 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",    /* 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 */
};
```

Fragments from the qlisp driver's install routine illustrate the previously mentioned steps in the driver.

```c
int qlisp_install(void)
{
    ....
    wsio_register_addr_probe(parallel_scsi_probe, "QLISP_NAME");
    wsio_install_driver(&qlisp_wsio_info);
    ....
}
```

The 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 Chapter 4, “Writing a Driver.”

All routines internal to the qlisp driver are given a brief description and for some, pseudo code is provided to illustrate the code concepts.

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.

NOTE The “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_pdk_load_nvram_values()
- Retrieves 256 byte NVRAM region from ATMEL 93C56/66 serial EEPROM on Qlogic card.

qlisp_pdk_nvram_read()
- Programs serial EEPROM to read a 16-bit value from NVRAM.

qlisp_pdk_nvram_send_bits()
- Send bits to serial EEPROM.

qlisp_pdk_init_common()
- Initialization routine. Common to master and slave ports.

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.

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/O's 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`

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 icobs (this may not always succeed)
if didn’t get reqd icobs, 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 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.
} until bp != NULL

qlisp_fast_complete

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_bcount - 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_cbfns

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 cbfns list
Get scb from the bp
Set cbfns to bp->av_forw
Call scsi_cbfn()
done with while
qlisp_process_rsp

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

The qlisp_isr is the driver interrupt routine. It handles interrupts on the Qlisp 12160 card.

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 async 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

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

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

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

qlisp_reset_bus

Causes a SCSI bus reset with a mp_timeout delay before I/O's 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

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 or LUN request queue.