# The Design and Implementation of the RPC Device Drivers

Ian A Penny
(email: iap@ukc.ac.uk)

16th December 1993

## 1 Overview

The RPC project group is investigating high performance communication network interface structures which are compatible with existing operating systems, in this instance SunOS 4.1 Unix. The use of parallel processing in the marshalling and unmarshalling of RPC arguments together with direct I/O to and from the user's data area and early scheduling of user processes, are expected to give a higher throughput than more traditional implementations.

The network front end comprises PC based TRAM's. The Unix machine is a Sun SPARC1+ running SunOS 4.1.3. The interconnection between the two systems is by the SCSI bus. To implement this structure requires a kernel device driver to act as a bridge between the Unix environment on the SPARC station and the TRAM's in the PC.

## 2 The Kernel

The structure of a device driver is closely tied to the run time structure of the Unix kernel. A short description of the essentials is given here to provide background before the structure of SCSI device drivers is discussed.

The kernel is divided into a top half and a bottom half. The top half of the kernel provides services to applications in response to system calls or traps. The top half of the kernel executes in a privileged execution mode in which it has access to both kernel memory and the context in which the user process is executing, its *user area* or *u dot* structure.

The bottom half of the kernel contains the routines that are invoked to handle hardware interrupts and traps that are not related to the current process. These interrupts may be delivered by hardware devices such as a real-time clock or I/O device. Activities in the bottom half of the kernel occur asynchronously and cannot rely on having a specific process executing at the time of interrupt.

Whilst executing the code for a system call the kernel is never preempted, it must either run to completion or voluntarily give up the processor to wait for a resource. It may, however, be interrupted by activities in the bottom half of the kernel. The top half may block entry into routines in the bottom half of the kernel by setting the processor priority level to a value which blocks out the appropriate interrupts.

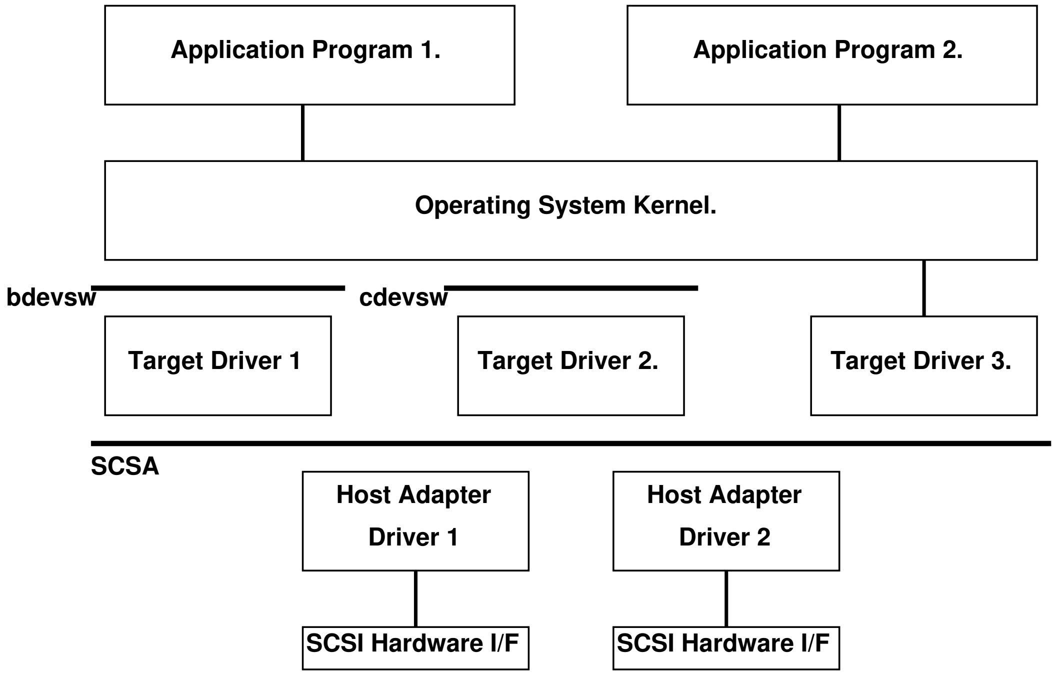


Figure 1: Block Diagram of Sun Common SCSI Architecture.

# 3 Sun Common SCSI Architecture

The SCSA provides a library of support routines for the transmission of SCSI commands from a Target driver[1] to a Host Adapter driver[2]. The interface is independent of the host adapter hardware being used, and the SCSI command set being transported. The SCSA separates those functions related to SCSI protocol management, which are included in the Host Adapter Driver, from the functions related to SCSI command content, which are included in the Target Driver.

# 4 SCSI Character Device Drivers

Unix supports access to a hardware device through either the block or character interface. Block oriented device interfaces use the buffer cache which resides in the kernel's address space. All I/O operations are block oriented and done to or from these kernel buffers. This approach requires at least one memory-to-memory copy to satisfy a user request.

Character oriented device interfaces have one of two styles, depending on the characteristics of the underlying hardware. For block oriented devices such as a disk, a character device interface may provide unstructured or raw access to the device. True character devices provide a byte stream oriented interface. A processor device, such as the SCSI Tram, requires this type of interface because it is not limited to sending or receiving fixed size blocks of data.

Classically any device driver has three major sections.

1. Autoconfiguration and Initialization routines.
2. Routines for servicing I/O requests.

[1] A Target driver is responsible for the generation of SCSI commands and interpretation of data arising from the transmission of such commands.

[2] The Host Adapter driver is responsible for the management of the SCSI protocol chip and DMA resources, together with virtual address mapping.

3. Routines for servicing interrupts.

## 4.1 Autoconfiguration Process

The autoconfiguration of the driver takes place when the system is initialized. It is responsible for determining if a hardware device is present and to initialize both the device, and any software state associated with it.

For SCSI device drivers the autoconfiguration process proceeds as follows. The host adapter is configured prior to any search for target drivers. Once the host adapter is configured it is required to call a SCSA routine `scsi_config()` which identifies it to the SCSA library. `scsi_config` then traverses the `config(8)` generated array of `scsi_conf` structures looking for target devices which have been declared to exist on a SCSI bus connected to this adapter. For each match, a `scsi_device` structure is generated and the appropriate target drivers `slave()` routine is called with the `scsi_device` structure as it argument.

A target driver's `slave()` routine is responsible for validating the presence of the hardware. It should return 0 for failure or 1 for success, and set the `sd_present` field of the `scsi_device` to reflect the presence of the device. If the `slave()` routine returns 1 and the `sd_present` field is set then the SCSA library will call the target driver's `attach` routine. The `attach` routine must decide whether the addressed hardware is ready for service.

For systems with openproms, the semantics of the `dev_info` structure state that a device is considered present when the `devi_driver` field is initialized to point to the device's `dev_ops` structure. The SCSA implementation requires that this be filled in the same time as the `sd_present` field.

### 4.1.1 dev_ops structure

The `dev_ops` structure is designed to identify the entry points to a device driver, it is however largely redundant with the `cdevsw` and `bdevsw` structures. The paragraphs below describe the relevant fields of the structure. The other elements of the `dev_ops` structure do not need to be initialized as they are unused under the present implementation.

**devo_rev** The device driver revision level, which must be set to one when programming for SunOS 4.1. Under revision level one, the only valid elements of the `dev_ops` structure are the identify and attach routines.

**devo_identify** The function of the `identify` routine is to determine if this driver works with the device. For drivers which do not occupy a physical slot on the SBus, such as a SCSI target driver, no `identify` routine needs to be declared. Instead the target drivers `slave` routine must be declared to check the devices availability.

**devo_attach** For a SCSI target driver the `attach` routine has two main responsibilities.

1. To allocate any state memory (data structures) the device driver needs in order to function. This keeps the size of the object files and, more importantly, the kernel to a minimum if the device is not present.
2. To perform any device initialization needed.

The `attach` routines of drivers which occupy a physical slot, such as the host adapter, have more responsibilities than just these two.

## 4.2 The Top Half.

The top half of the driver is responsible for servicing I/O requests, as a result of a system call or on behalf of the virtual memory system. It executes synchronously in the top half of the kernel and is permitted to block by calling `sleep()`.

### 4.2.1 Entry Points.

Device drivers are connected to the rest of the kernel by the entry points recorded in the array for their class, their use of a common buffering system, and their use of low level hardware support routines. For character devices the array of entry points consists of `cdevsw` structures, the array is declared in `/usr/src/sys/sun/conf.c` and the structure in `/usr/src/sys/sys/conf.h`. To index into this array each device is distinguished by a *major device number*. A device's *minor device number* is interpreted at the driver level, and is hence unimportant at this stage.

The relevant fields of the `cdevsw` structure are described below.

**open** Prepare the device for I/O operations. This routine will be called for each `open` system call on a character special device file. Typically it should validate the `minor device number` and do other device specific error checking, such as ensuring the device was identified correctly in the autoconfiguration phase. If there are no errors it should then proceed to initialize the device, allocating any resources which are needed on a per-minor device basis, and wait for the device to come online.

**close** Shutdown the device. This routine is called when the last client using the device either explicitly calls the `close` system call, or terminates. Consequently it must clean up for all clients which have had the device open, typically releasing or unlocking any resources which were used by the device and indicate that data cannot be transferred until the device has been reopened.

**read** Read data from the device. A read request requires that the driver copy the data from the device to the application's virtual address space. It must check that the `minor device number` passed to it is within its range, but subsequent actions are dependent on the type of device.

**write** Write data to the device. This entry point is similar to the `read` routine except the direction of data flow is reversed.

**ioctl** Perform an operation other than a read or write. These operations differ for each type of device, but can provide a means for setting device parameters and for checking the status of a device. This entry point is intended to implement any commands which are not covered by the rest of the device-driver entry points.

**select** Check the device to see if any data is available for reading, or space is available for writing. This entry point is used by the `select` system call to check file descriptors associated with this device. The driver must maintain a local per-device structure that can associate a process with each device, and keep track of state information to enable this check to be made.

**mmap** Map a device's contents into memory. This entry point is used by the `mmap()` system call to provide page table entry (PTE) information about pages of the device's memory. This information is needed by the kernel to map the page to a virtual address.

### 4.2.2 Typical Operation

When an I/O request is received by the Target Driver it must prepare a SCSI command that, when passed to the Target Device, will perform the desired function. There is a relatively simple sequence of events which must be performed in order for the transfer to take place.

1. Target Driver executes functions to obtain a block of memory.
2. Target Driver fills in the blanks in the block of memory to describe the command for the host adapter.

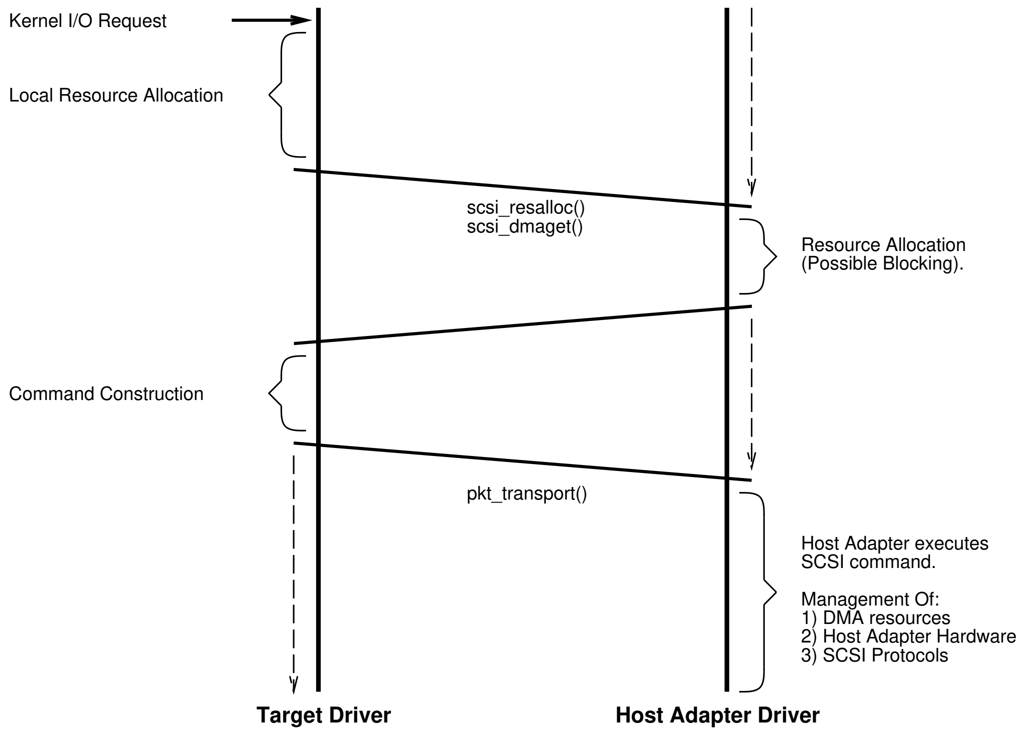


Figure 2: Response to a Kernel I/O request.

3. Target Driver calls Host Adapter Driver functions to obtain the other resources necessary to execute the desired command.

4. Host Adapter Driver manages the resources and can either allocate them now or later at its discretion.

5. Target Driver passes the descriptive information to the Host Adapter Driver using the transport function.

6. Target Driver is free to prepare other SCSI commands for the same target and for other attached targets.

Figure 2 illustrates this sequence with the function names of the routines used by the Target Driver to request resources from the Host Adapter Driver. There are two separate ways to allocate resources for a command requiring a DMA transfer. The first is to allocate all the resources at one go using `scsi_resalloc` this allocates not only the `scsi_pkt` but also the DMA resources required to perform the transfer. The second way to perform this allocation is to call `scsi_pktalloc`, this routine allocates the necessary `scsi_pkt` but does not allocate any DMA resources for it. Subsequently the Target Driver may call `scsi_dmaget` to promote the command to one with associated DMA resources. It is usual for a Target Driver to allocate a request sense packet in the autoconfiguration phase, this packet may then be used in response to error conditions without the need to go through resource allocation again.

## 4.3 The Bottom Half.

The bottom half of the driver consists of the routines for servicing interrupts. These routines execute in the bottom half of the kernel and as a consequence cannot rely on any per-process state and cannot block by calling the `sleep()` routine.

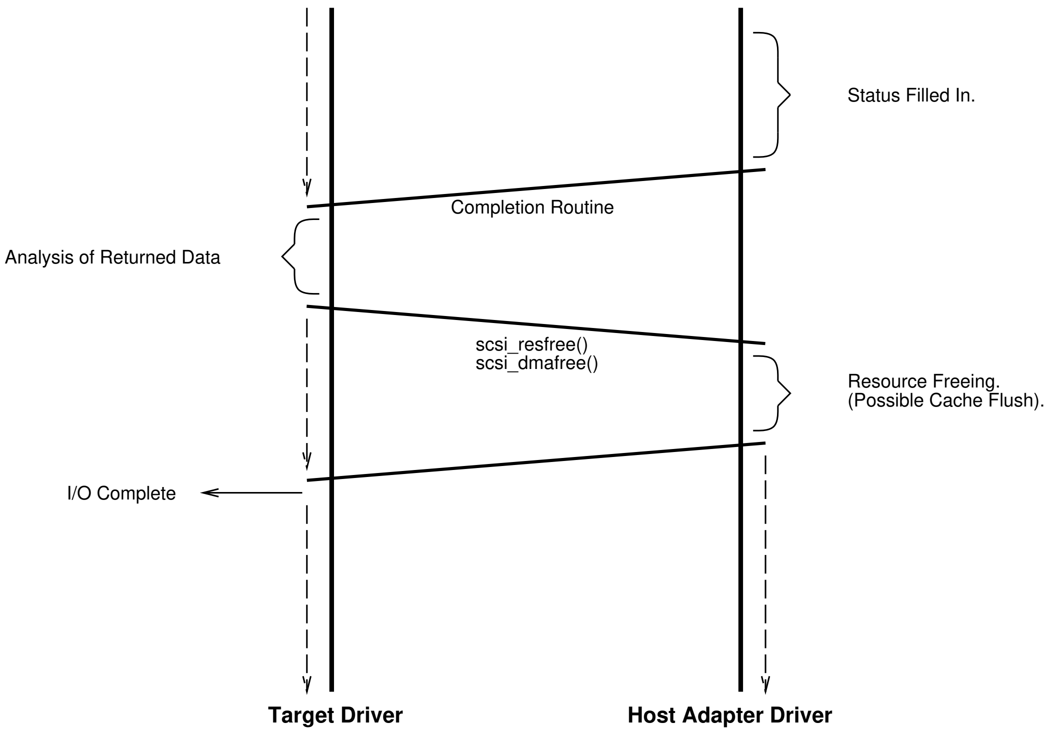


Figure 3: Command completion.

The two halves of the driver communicate by using I/O queues. When an I/O request is received by the top half of the driver it must record the command on a per-device queue for processing. When a command completes the target driver will receive an interrupt from the host adapter. The interrupt service routine must notify the requester that the command has completed and then initiate a new command from the device's queue. The I/O queue is the primary means of communication between the top and bottom halves of a target driver.

As the I/O queues are shared by the two halves of the driver, access to them must be protected. The top half of the driver must raise the priority level of the processor, by using `splx()`, to prevent the bottom half of the driver from being entered whilst the top half is manipulating the queues.

### 4.3.1 Typical Operation

When the command has completed successfully or unsuccessfully, the Host Adapter Driver is responsible for calling the completion routine of the Target Driver. The sequence of steps involved are described below.

1. Host Adapter Driver fills in remaining status information in allocated memory area.
2. Host Adapter Driver calls the Target Drivers completion routine.
3. Host Adapter Driver is no longer responsible for the command.
4. Target Driver analyses the returned status information.
5. Target Driver requests that the Host Adapter Driver releases the resources allocated for this command.
6. Target Driver notifies the process that originally requested the transaction that it is complete.

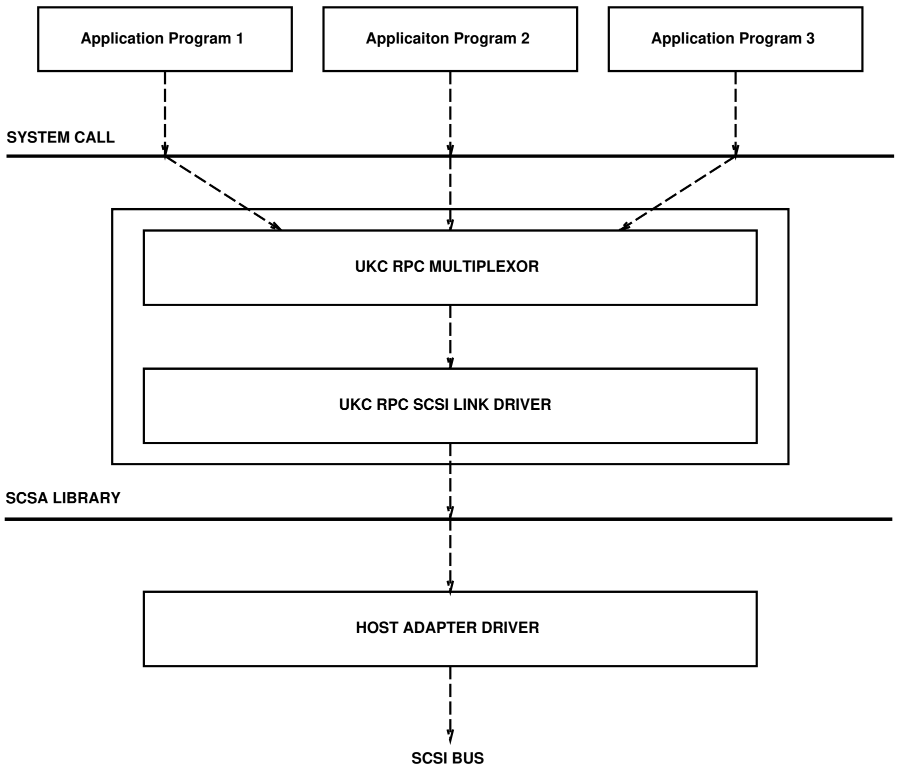


Figure 4: Structure of RPC Drivers.

# 5 UKC RPC Drivers

Having dealt with the general case Target Drivers the rest of this document concentrates on the RPC Target Drivers, their structure and operation.

## 5.1 Structure

The RPC project calls for a kernel device driver to act as a link between the Unix environment on the SPARCstations and the parallel processing environment of the TRAM's. The driver is required to perform two tasks, the first is to provide multiple RPC interfaces to the application environment. The second is to communicate with the scsi TRAM. This suggests using two drivers, a multiplexor and a SCSI Link Driver.

The RPC Multiplexor provides a number of character-special devices, each one offering an autonomous RPC slot. The limit on the number of these devices is 256 as minor(dev) is 8 bits. It multiplexes these RPC interfaces onto the link provided by the SCSI Link Driver. It is the multiplexors responsibility to match up outbound RPC's with their replies, and provide a system call interface for user processes.

The RPC Link Driver provides an asynchronous send / receive interface across the SCSI bus to the TRAM. This interface should provide reliable delivery of the commands to the TRAM, the multiplexor is informed only when writes complete or network input has arrived.

## 5.2 Autoconfiguration Process

The RPC Multiplexor does not need to go through autoconfiguration as it does not have any associated hardware. The resources it needs are assigned when the first open system call is processed, see description of open entry point.

The RPC Link Driver must determine if the TRAM is present and running the RPC Tram Driver. Its autoconfiguration process proceeds as follows;

`ukcsslave` is called once the host adapter has been configured. It calls `scsi_slave()` which sends a `TEST UNIT READY` to the Target/Lun address for this device. If this succeeds it will send an `INQUIRY` command. If the inquiry succeeds then the `sd_inq` field in the device structure will be filled in. `ukcsslave` then examines the device type to ensure it is a processor and attempts to match the vendor and product identifiers with those returned by the RPC Tram Driver. If any of the steps of the `scsi_slave` routine fail, or `ukcsslave` cannot match up the device type and vendor/product identifiers with those of the RPC Tram Driver then it will return 0, indicating to the host adapter that the hardware failed the autoconfiguration process. At this point the Link Driver has determined that the unit is present and that it is executing the code for the RPC Tram Driver.

The next step is to allocate a request sense packet, this is used if a SCSI command generates a `CHECK CONDITION` status. It is allocated here so that the Link Driver can transport a `REQUEST SENSE` command without the need to perform new allocations. When a command completes with a `CHECK CONDITION`, the Host Adapter Driver assumes that the next command transmitted to that Target/Lun will perform the recovery. The Target/ Lun is frozen by the Host Adapter Driver until another packet is transmitted for that Target/Lun.

The last stage is to allocate and fill in the Link Drivers private data structure for this Target/Lun. The `sd_present` field of the `scsi_device` is filled in to indicate that the device is present, and the `dev_ops` entry is filled in.

`ukcsattach()` is called if `ukcsslave` returns 1 and the `sd_present` field is set. As `ukcsslave` has already allocated the private data structure for this Target/Lun, `ukcsattach` only needs to perform device initialization. If the driver state is NIL it marks it as CLOSED and prints the Target/Lun and the Vendor identifier. The last stage is to fill in the private data structure `un_attached` field to indicate that the device has been attached.

The `ukcsattach` routine has far less to do than its counterparts in other scsi target drivers. This is because the real initialization of the device is done by the `ukct_setup` routine, which is called by the multiplexor open routine (see open entry point).

## 5.3 The Top Half.

The following section deals with the top half of both the Multiplexor and Link driver. Although the Link driver does not provide a system call interface its routines are called by the Multiplexor in response to system calls and hence are classified as part of the top level.

When an application emits an RPC request a fixed size structure is passed over the read/write system call interface. This structure describes the RPC request to the RPC drivers and the TRAM. The RPC drivers use an extended version of this user structure in order to keep track of the owning process and form the ioqueues. This extended version is passed across the SCSI bus to the TRAM, hence some of the fields of the structure have no meaning to the RPC drivers. These two structures are described below.

### 5.3.1 ukcrpccmd_t

```
typedef struct {
    int     ur_cmd;         /* command type */
    caddr_t ur_ibp;         /* input buffer addr */
    int     ur_ibytes;      /* max/actual input length */
    caddr_t ur_obp;         /* output buffer addr */
    int     ur_obytes;      /* mac/actual output length */
```

```
        int     ur_stag;        /* RPC service number */
        int     ur_utag;        /* chosen by user (client) */
        int     ur_ktag;        /* chosen by kernel (server) */
    } ukcrpccmd_t;
```

**ur_cmd** This field identifies the purpose of the RPC request, and is interpreted by the TRAM. The RPC drivers are designed as a bridge and hence have no knowledge of the implementation details of the RPC requests themselves.

**ur_ibp** This identifies the client's input buffer to the RPC drivers. The input buffer is "locked down" whilst the RPC request is in flight, allowing the RPC drivers to DMA directly to the input buffer.

**ur_ibytes** When passed by the client to the RPC drivers, this field specifies the maximum amount of data that may be DMA'ed into the clients input buffer. When passed by the RPC drivers to the client, this field specifies the actual amount of data which was DMA'ed into the clients input buffer.

**ur_obp** Analogous to `ur_ibp`, this field identifies the client's output buffer to the RPC drivers. It is also locked down whilst the RPC request is in flight to allow DMA directly from the buffer.

**ur_obytes** This field specifies the maximum amount of data which can be DMA'ed from the client's output buffer.

**ur_stag** This field identifies which service the RPC request is destined for. A service handler is responsible for associating servers and clients, it does this through the service number (`ur_stag`) mechanism.

**ur_utag** This field is unique to the RPC request and is chosen by the user. If a client has multiple requests outstanding and a reply returns it may use this field to determine which request has completed.

**ur_ktag** This field is used by the kernel to match up RPC requests with their replies at the service handler level (see `ur_stag` above). It must be unique within the context of a service, but two different services may use the same kernel tag (`ur_ktag`). When a request or reply is received by a service handler, the service number identifies which service it is destined for and the kernel tag identifies it within that service.

### 5.3.2 ukckcmd_t

```
    typedef struct ukckcmd_node *ukckcmd_ptr;
    typedef struct ukckcmd_node {
        ukckcmd_ptr     uc_next;        /* Command Chain */
        ukcrpc_ptr      uc_uk;          /* Back ptr to minor(dev) struct */
        struct proc     *uc_uproc;      /* Owners proc table entry */
        int             uc_flag;        /* Flags */
        int             uc_mtag;        /* minor(dev) + uniqueness bits */
        ukcrpccmd_t     uc_rpccmd;      /* User command */
    } ukckcmd_t;
```

**uc_next** This field identifies the next entry in the ioqueue. The RPC drivers ioqueue consists of a linked list of `ukckcmd_t` structures.

**uc_uk** Used by the Multiplexor to identify which minor device (RPC slot) this command is associated with. Primarily this is used when the Link driver calls the Multiplexor completion routines for read or write, the command can then be queued on the minor device's completion queue.

**uc_uproc** Required by the multiplexor to lock down the user's input and output buffers. It is also required by some of the SCSA library support routines.

**uc_flag** This field may be used by the RPC drivers or TRAM to indicate any special activities which must be performed for this command.

**uc_mtag** If RPC replies arrive 'late' and the client has issued a close system call it is possible that another client has opened the slot with outstanding traffic. The `uc_mtag` field protects clients from RPC traffic from a previous existence.

**uc_rpccmd** This holds a copy of the user's `ukcrpccmd_t` for use by the RPC drivers.

### 5.3.3 Entry Points

The Multiplexor implements a subset of the full character device entry points, its `cdevsw` entry is listed below. The Link driver does not have a `cdevsw` entry, as it does not implement any system call interface. The Link driver top level routines are accessed solely by the Multiplexor.

```
{
    xx_open,        xx_close,       xx_read,        xx_write,
    xx_ioctl,       xx_reset,       xx_select,      xx_mmap,
    (struct streamtab *)xx_str,     xx_segmap,
},
{
    ukcr_open,      ukcr_close,     ukcr_read,      ukcr_write,     /*105*/
    nulldev,        nulldev,        ukcr_select,    0,
    0,              0,
},
```

`nulldev` is placed in insignificant entries in the `cdevsw` structure. These entries are not invalid, they are merely not configured. `nodev` should be placed in illegal entries in the `cdevsw` structure, it will return `ENODEV` if a process attempts to use the entry point.

**ukcr_open** The responsibilities of the open entry point for the Multiplexor are extended to encompass the Link driver validation, as the Link driver has no open entry point. The RPC driver open process proceeds as follows;

Upon the first open or subsequently after the Link driver has informed the Multiplexor that the TRAM has crashed, `ukcr_init` will be called. `ukcr_init` initializes the ioqueue and fills in a structure containing pointers to the Multiplexor's completion routines for reading and writing, and the reset routine. It passes this structure to the Link Driver by calling the `ukct_setup` routine.

`ukct_setup` checks that the autoconfiguration process succeeded and sets up the logical send and receive channels to the TRAM. These channels are implemented by using two logical units at the same target. It will also interrogate the Host Adapter to see if it supports synchronous mode transfers and if so sets up the channels to use it. `ukct_setup` will send a `TEST UNIT READY` command to each logical unit to check they are both available, and allocate a command and data packet for each channel. Assuming no problems arise the Multiplexor completion structure is stored and a message will be printed to indicate that the channels have been activated.

Once both channels have been initialized `ukct_setup` sends a read to the TRAM. The TRAM is expected to disconnect from the bus and reconnect when data becomes available. This allows the TRAM to interrupt the SPARCstation and provides a means of asynchronous communication down the SCSI bus. Once the read has been sent, `ukct_setup` fills in the

Link Driver priority level so that the Multiplexor can block the Link Driver and returns. `ukcr_init` stores this priority before returning.

If any errors occurred whilst trying to initialize the ioqueue or validate the Link Driver then `ukcr_open` returns the error generated by the lower level routine. `ukcr_open` then ensures that the requested slot is not being used by anyone else, it will return `EBUSY` if this is the case. A process receiving `EBUSY` from an open is expected to increment the minor device number and reattempt the open. Because parts of the process area will be used for direct DMA it must not be swapped out, this is done by overloading the meaning of `SPHYSIO` in the process table entry. This does mean that physio on other devices is forbidden, but does allow the Multiplexor to regain control on exit before the user's virtual machine is released. Because of this setting of process table entries the Multiplexor must ensure only the opener may use the slot. Children may inherit the file descriptor associated with an RPC slot from their parent, but they are forbidden to use it.

**ukcr_close** The close entry point is responsible for shutting down the slot and tidying up. `ukcr_close` verifies that the device is open returning `ENXIO` if it is not. It must ensure that the process attempting the close is the one which opened the device. If it is not `ukcr_close` returns 0, indicating that the close was successful. In this case the Multiplexor will close the device when the "real" owner calls close.

`ukcr_close` must attempt to deal with any outstanding commands. Any currently active DMA operation must be allowed to complete. Any commands on the write queue for the TRAM must be sent to ensure little or no traffic is lost. Any items on the read pending queue are placed on the read queue, from where the associated buffers are unlocked. The device state is cleared to indicate that it is closed.

Once any commands have been dealt with `ukcr_close` checks to see if this was the last outstanding RPC slot open for this process. If it was then `SPHYSIO` is cleared in the process table entry. `ukcr_close` then returns.

`ukcr_forceclose` is called from exec and exit and allows the user's pages to be unlocked before the user's virtual machine is discarded. It cycles through the valid minor device numbers, attempting to close each one. The behavior of `ukcr_close` ensures that the device will only be closed by the user process which called open. It then attempts to clear `SPHYSIO`. It will `panic()` the kernel should `SPHYSIO` not be set when it attempts to clear it.

**ukcr_read** The read entry point reads a `ukcrpccmd_t` from the user and fills it in with the next command on the read queue. It ensures that the process has this slot open and that the correct number of bytes have been requested. If the Link Driver has reported that the TRAM has crashed then `UKF_TRAMDOWN` will be set, in this case any open RPC slots must be closed. Attempting to read from a slot whilst `UKF_TRAMDOWN` is set is considered an error.

The next `ukckcmd_t` is taken off the read queue, if no commands are available `ukcr_read` will return `EWOULDBLOCK`. The users memory which was locked down for DMA is released and the user command is copied into the user supplied `ukcrpccmd_t`. The kernel command (`ukckcmd_t`) is placed on the free message list for reuse.

**ukcr_write** The write entry point reads a `ukcrpccmd_t` from the user and transports it to the TRAM. It must ensure that the process has this slot open and that the correct number of bytes have been requested. As with the read entry point, if the TRAM has crashed any RPC slots must be closed. Attempting to write to a slot whilst `UKF_TRAMDOWN` is set is considered an error.

`ukcr_write` gets a `ukckcmd_t` from the free list and clears it, if no commands are available `ukcr_write` will return `EWOULDBLOCK`. The user's command is loaded into the kernel command structure and the user's buffers are locked down to enable DMA transfers from/to them. The kernel command is placed on the tram write queue and the start routine (`ukcr_start`) is called.

If the Link driver is not busy, `ukcr_start` takes the first kernel command off the tram write queue and passes it to the Link driver by calling the routine `ukcr_wtram`.

`ukcr_wtram` ensures that no work is outstanding and that the TRAM hasn't crashed before it constructs two `buf` structures to describe the command to the low level SCSA routines. It links the second buf, containing the data to the first, containing the kernel command and places them on the send channel queue. It calls the routine `ukcsstart` to begin the transfer.

`ukcsstart` checks to see if resources have already been claimed for this transfer, if not it will call `ukcs_make_cmd`. `ukcs_make_cmd` calls the SCSA routine `scsi_dmaget` to allocate the necessary DMA resources for this SCSI packet, and then uses the MAKECOM_G0_P[3] macro to create a SCSI command descriptor block. If resources were not available `ukcsstart` marks this device as waiting for resources. The callback mechanism implemented by the SCSA will call the routine `ukcs_retry` when resources "may" have become available. `ukcs_retry` just attempts to restart any devices waiting for resources. The SCSA requires that it must return 1 if it was either able to allocate resources or chose not to attempt it, else it must return 0. If resources were available `ukcsstart` moves the command from the pending queue to the active queue and calls the SCSA routine `pkt_transport` to transport the command. Once `pkt_transport` returns the command is no longer the responsibility of the Link driver, the Host Adapter will call the Link driver completion routine `ukcsintr` when it has completed the command or can proceed no further. Execution unwinds until `ukcr_write` returns.

**ukcr_select** The select entry point checks that the process has the requested device open and that the TRAM has not crashed. If there is data available in the case of a read select, or if data can be sent in the case of a write select, `ukcr_select` will return 1. If a select has already blocked for this device and subsequently another select occurs which will also block, then the collision flag is set. `ukcr_select` will cause the process to be blocked, returning 0.

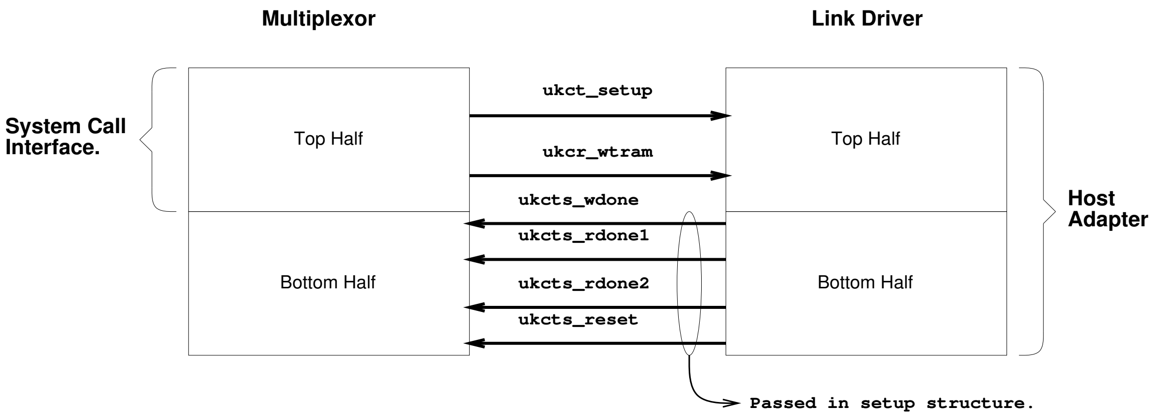


Figure 5: Interaction between Multiplexor and Link Driver.

## 5.4 The Bottom Half.

The bottom half of the Multiplexor is invoked by the Link driver when it has completed operations either successfully or unsuccessfully, on behalf of the Multiplexor or RPC traffic has been received from the TRAM. The bottom half of the Link driver is invoked by the Host Adapter driver when it has completed a scsi command on behalf of the Link driver, or resources have become available when the Link driver is blocked waiting for them. The latter case is described in the `ukcr_write`

[3] This macro fills out many of the fields of a `scsi_pkt` for group 0 commands that require the special bits associated with processor device types.

entry point. In the former case, the host adapter interrupts the Link driver by calling the `ukcsintr` routine. It is the responsibility of `ukcsintr` to examine the status of the returned command and figure out what to.

### 5.4.1 ukcsintr()

The command completion routine `ukcsintr` begins by ensuring that the scsi packet is valid. Invalid packets are those that have requested a non-interrupting service, or that have no private data area or device associated with them. If the command is incomplete `ukcs_incomplete` is called.

If the command was a request sense `ukcs_incomplete` will return `QUE_SENSE` if retrying it will not exceed the retry count. If the command was not a request sense `ukcs_incomplete` will return `QUE_COMMAND` if retrying it will not exceed the retry count. If the host adapter got the bus and the command cannot be retried then the target has not responded to selection, in which case `ukcs_shutdown` is called.

`ukcs_shutdown` detaches both the send and receive channels and marks them as closed. It will call the Multiplexor's reset routine via the setup structure it received in `ukct_setup` (see open entry point). The Multiplexor reset routine `ukcr_reset` places all pending writes onto the read pending queue, and flags all open slots that the tram has crashed (`UKF_TRAMDOWN`) the only operation which may be performed on such a slot is a close. `ukcs_shutdown` and `ukcs_incomplete` return.

If the command was completed and the device is `SENSING` then `ukcs_handle_sense` is called, otherwise `ukcs_check_error` will be called. `ukcs_handle_sense` decodes and checks the information returned by a request sense. Based on the decoded information it may mark the command as done, or done but with errors or that it should be queued for retransmission. `ukcs_check_error` determines if any errors occurred on completed commands. If the command returned with a busy status then it will be queued for retransmission if retrying it does not exceed the retry count, if it would then it is marked as done with errors. If the command returned with a check condition then the Link driver must send a request sense command.

Once the command has been checked by the lower level routines, `ukcsintr` decides what to do with the command based upon their return values. If the command needs to be retransmitted, `ukcsintr` calls `pkt_transport`. Otherwise the command completed either successfully or with errors, in which case `ukcsdone` is called.

### 5.4.2 ukcsdone

`ukcsdone` cleans up after the last command and starts the next one. If it receives a command which has returned with an error condition it frees the resources for the scsi packet and returns. In later implementations it is envisaged that the Link driver will be required to recover from a number of different errors, each possibly requiring a slightly different strategy. If the command completes without errors `ukcsdone` must determine which channel (send or receive) this command arrived on.

If the command arrived on the send channel and there is still outstanding work, then the command which arrived is the `ukckcmd_t` in which case the `ukcsstart` routine is called to send the data. Otherwise the Multiplexor is informed that the write completed. The resources associated with this write command are released.

If the command arrived on the receive channel and there is an outstanding packet on the queue, then the command which has completed is the `ukckcmd_t`. The Multiplexor is informed which read command has partially completed. This two part sequence is needed in case a process dies before any outstanding RPC's are completed, in this case it would is crucial that the DMA transfer direct into the processes memory not be allowed to continue. If the Multiplexor okay's the second part of the read, the Link driver fills in the waiting buf structure and calls `ukcsstart` to begin the second part of the read. If the Multiplexor refuses the read, the resources for this command are released and the `ukcs_read` routine is called to set up the next disconnecting read (see open entry point). Otherwise, if nothing is on the pending queue the command which has arrived is the data. In this case the Multiplexor is informed that the second part of the read has completed. The resources for this command are released and `ukcs_read` is called to set up the next disconnecting read.

In both cases the Multiplexor is informed through the setup structure which was passed to the Link driver by `ukcr_init` when it calls the `ukct_setup` routine. This design ensures that if the Multiplexor is loadable in future implementations then it can inform the Link driver of its command completion routines.

## 6 Conclusion

The multiplexor provides for multiple character special device files, each of which acts as an autonomous RPC slot. It has no knowledge of the underlying SCSI structure, but instead uses a simple interface to communicate with the SCSI link driver. The link driver manages the generation of the necessary SCSI commands to transmit the data over the bus to the PC based TRAM's. RPC requests and replies are viewed as opaque data about which neither driver has any knowledge.

Futher work in this area of the project will involve timing the drivers and writing a local service handler which does not require SCSI interactions for those RPC's from clients whose server executes on the same machine.

## References

[Sun, 1991] Sun microsystems, Inc. "Writing SBus Device Drivers." Part No. 800-5322-10, (January 1991).

[Sun, 1990] Sun microsystems, Inc. "SCSA : SUN Common SCSI Architecture" Part No. 800-4701-10, (March 1990).

[Sun, 1989] Sun microsystems, Inc. "Implementation Guide : SUN Common SCSI Architecture" Part No. 800-4700-10 (November 1989).

[Leffler et al., 1989] S.J. Leffler et al. "The Design and Implementation of the 4.3BSD Unix Operating System." Addison Wesley (1989).

[ANSI, 1986] American National Standards Institute, Inc. "Small Computer System Interface (SCSI)" ANSI X3.131-1986, (June 1986).