Implementing a Transputer SCSI Interface

Andrew Smith,
Computing Laboratory, University of Kent,
Canterbury, Kent. CT2 7NF

Abstract

The work described in this report is part of a project to investigate high performance communication network interface structures which are compatible with existing operating systems. The aim of the project is to increase the effective speed of Remote Procedure Calls (RPC) for an application by adding parallel processing power to the communication subsystem without making major changes to the application programming environment.

This document describes the Transputer interface to the host. This interface provides an asynchronous bidirectional communication path between a SCSI bus and a number of Transputer links. A mechanism for supporting Remote Procedure Calls between the host and a network interface is provided as part of the interface.

1 Coral SCSI TRAM Structure

The work described in this report is part of a project to investigate high performance communication network interface structures which are compatible with existing operating systems. The aim of the project is to increase the effective speed of Remote Procedure Calls (RPC) for an application by adding parallel processing power to the communication subsystem without making major changes to the application programming environment (figure 1).

The Coral HPT04 TRAM provides a Transputer/SCSI interface via the NCR 53C710 SCSI controller [1] (figure 2). The NCR processor manages all hardware and software SCSI bus transactions and interfaces to the shared Transputer memory through a DMA interface. This intelligent SCSI controller is able to handle most SCSI transactions, thus allowing the Transputer to simultaneously perform other computational tasks.

The NCR 53C710 is programmed via a microcoded engine in a language called "SCSI Scripts" [2]. The scripts language allows the device to be programmed to handle the majority (if not all) of the SCSI transactions. Communication with the host processor (the Transputer) is via the Transputer interrupt mechanism. The NCR SCSI controller is thus able to operate fully in parallel with the Transputer.

2 SCSI TRAM Device Driver

The SCSI driver implements the SCSI target processor model as defined in the SCSI I [5] and SCSI II standards [6]. This model provides a processor target with simple send/receive attributes to an initiator (host). The initiator is able to send and receive blocks of data of variable size to a maximum of 64K bytes.

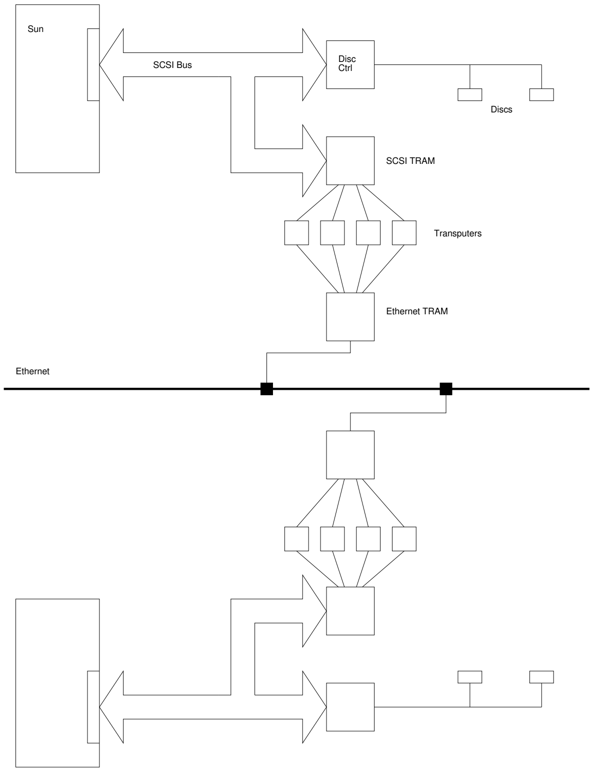

Figure 1: SUN/Transputer Architecture

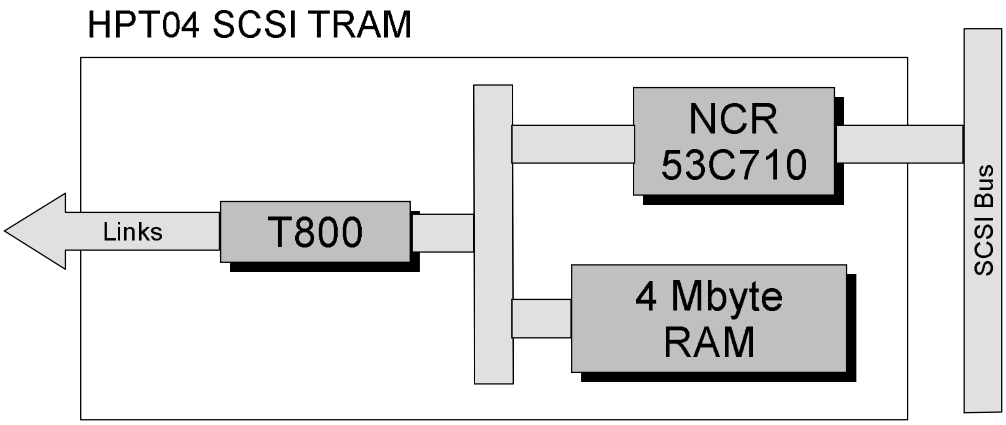

Figure 2: SCSI TRAM Architecture

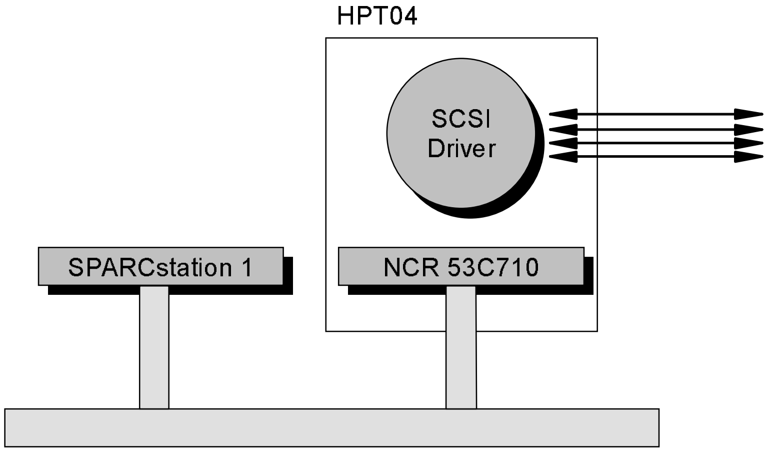

Figure 3: SCSI Driver Architecture Design – model 1

2.1 Architecture Design

A number of models were proposed and evaluated for the implementation of a bi-directional interface between the initiator and target devices. The main obstacle in designing the interface was the requirement that when the initiator was blocked, waiting for data from the target during a receive command, that the initiator should still be able to send data to the target.

The first model was based on the initiator controlling the flow of data to and from the target (figure 3). Sending data to the target is a simple case of issuing a SCSI *snd* command and waiting for the acknowledgment from the target for the data. Receiving of data from the target is more complicated as targets are not allowed to initiate transfers to an initiator. Thus the initiator has to poll the target by sending it requests to inquire if data is available for transfer. If the target confirms that data is available, the initiator issues a *rec* command to receive the data from the target. Because of this polling operation and the load on both the initiator and target controllers and SCSI bus, it was felt that this solution would not be practicable.

The second model was based on multiple initiators. The same mechanism of sending data to the target was used as in the first model. When the target required data to be sent to the initiator the target would swap rolls to become an initiator and issue commands to the Sun. This mode of operation is allowed for in the SCSI standard but early experiments showed that the Sun SCSI controller was intolerable of multiple initiators on the bus. Another disadvantage became apparent due to the uncertainty as to whether the Sun SCSI software support could operate in target mode.

The third, and implemented, model was based on the target controller having multiple logical units. Two logical units were implemented for the *snd* and *rec* channels between the initiator and

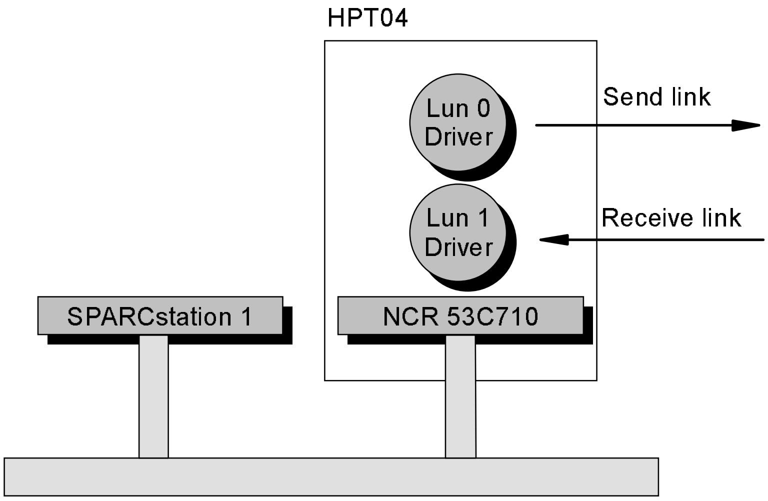

Figure 4: SCSI Driver Architecture Design – model 3

targets. When the initiator has data to send to the target it simply issues the SCSI *snd* command on the designated channel and passes the data to the target. Receiving data from the target operates in a similar manner, the initiator issues a *rec* command and waits for the target to return the data. The case when the target is unable to supply data is handled easily via the target disconnecting from the bus, thus freeing the bus for other traffic, and reconnecting when data is available. The splitting of the data transfers onto separate channels (logical units) provides deadlock free communication between the devices. Multiple SCSI commands on the same channel are not permitted in this implementation thus avoiding potential areas of deadlock and complicated command queueing management techniques. Figure 4 shows this model.

3 Operation of the SCSI Driver

Figure 5 shows the structure of the SCSI driver. A master process provides management of the SCSI bus (via the NCR SCSI processor). The other processes are duplicated in pairs for the receive and send channels of the communications. These processes manage the Transputer links, service requests, and SCSI bus. The auxiliary processes manage access to the shared data buffers and service handler buffer.

A single dedicated process manages the SCSI processor. Communication between the Transputer and NCR processor is via a shared set of registers and a hardware interrupt mechanism. The management of the Transputer links is controlled by two dedicated processes, one for each link. These processes have no control over the SCSI bus. Access to the buffer is controlled via a process which manages a set a semaphores in order to resolve conflicts in updating buffers and associated pointers. The use of software channels for the transfer of data between the control

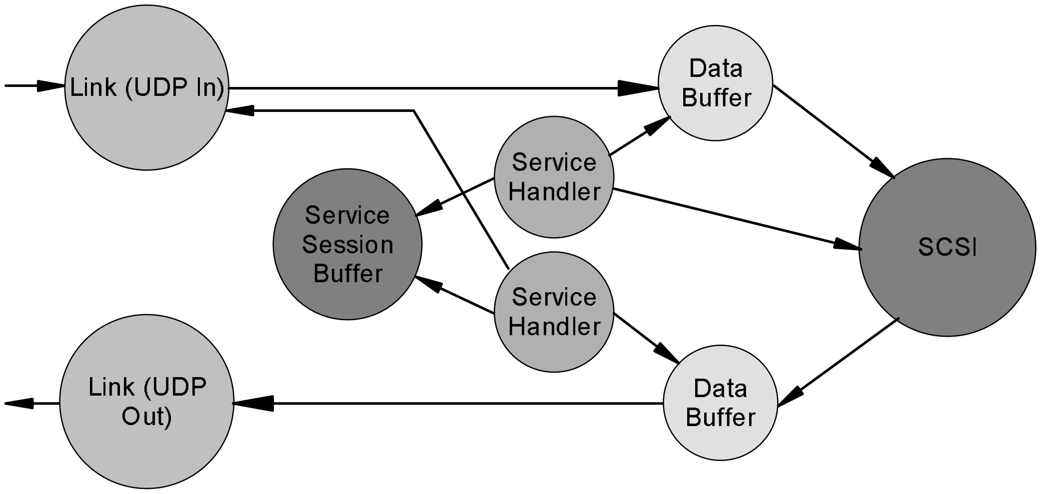

Figure 5: Architecture of TRAM SCSI Driver

processes would allow for a much simpler design but the overhead is considered too great and would introduce latency between each process which would not be tolerable. The nature of the operation of the Coral driver disallowed the standard occam technique of sharing buffers via the movement of the process to the data [3].

3.1 Controlling the SCSI BUS

The SCSI bus is under the total control of the NCR SCSI processor. The majority of the control of the SCSI hardware can be handled via the SCSI script language without intervention from the Transputer. The SCSI process interrupts the Transputer only when data is either required by or is available for the processor. Figure 6.

The SCSI driver is structured so that all data from the initiator is sent to the TRAM on logical unit zero, and all data read from the TRAM is on unit one. Sending data to the TRAM on unit one and reading from unit zero will fail.

3.1.1 NCR SCSI Scripts

The SCSI bus is controlled via the SCSI script driver which is configured, loaded into the NCR 53C710, and initiated when the Transputer is first booted. These scripts are programmed to interrupt the Transputer (by asserting the Transputer event line) at a number of stages during a SCSI transaction:

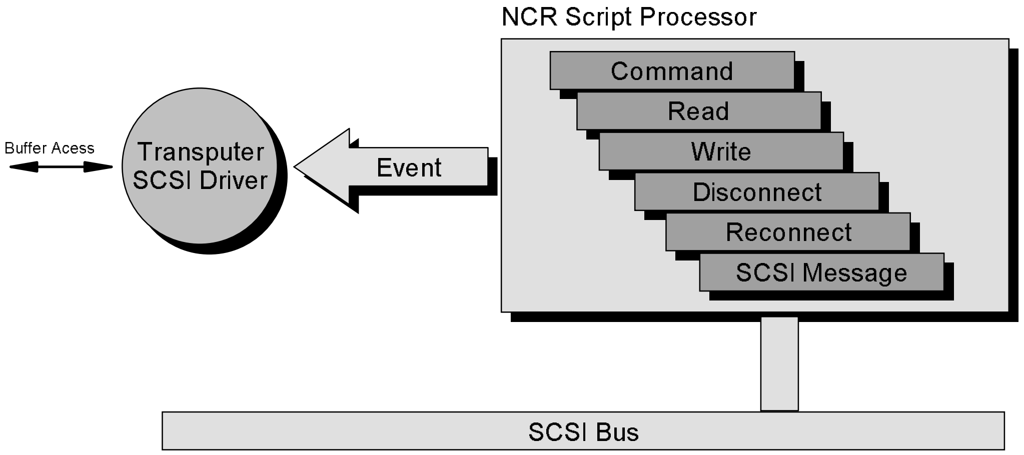

Figure 6: Managing the SCSI Bus

- Bus selection and SCSI command processing. This signals the completion of the SCSI command phase – the NCR processor has managed the SCSI bus negotiation, connection, message processing, and command decoding;
- Data transfer. The data phase is complete – the data has been transferred successfully to or from the initiator;
- Bus disconnection. The current SCSI command has either completed successfully or partially (due to lack of data or buffer space);
- Message processing. A SCSI message requiring host intervention has been received from the initiator;
- Bus reconnection. Reconnection to the initiator has been successful and data transfer can now take place;
- Error processing. Parity errors and other hardware problems are notified to the host via the interrupt mechanism.

Figure 7 shows a small section of a script for processing a connection with the initiator.

A number of NCR SCSI processor registers are initialised on startup of the Transputer. There are only three main registers used during the processing of a SCSI transaction, the SCSI script instruction pointer (DSP), the interrupt status (ISTAT), and an index register to the shared data space (DSA).

```
target_script:
     SET TARGET
     WAIT SELECT REL(reconnect)        ; wait for selection

message_out_phase:
     CALL REL(message_phase), IF ATN

command_phase:
     MOVE 0, PTR cmd_buf, WITH CMD     ; move the command bytes in

;    Check for further messages
     CALL REL(message_phase), IF ATN

command_decode:
;    interrupt host for supported commands
     JUMP REL(inquiry_cmd),   IF 0x12  ; inquiry
     JUMP REL(receive_cmd),   IF 0x08  ; receive
     JUMP REL(send_cmd),      IF 0x0A  ; send
     JUMP REL(test_unit_cmd), IF 0x00  ; test unit ready
     JUMP REL(send_diag_cmd), IF 0x1D  ; send diagnostic

;    else we reject the command, send CHECK CONDITION status,
;    and disconnect
     MOVE 1, cmd_cmplt, WITH MSG_IN
     DISCONNECT
     INT command_rejected
```

Figure 7: Part of a Script for SCSI connection processing

3.1.2 Operation of the SCSI manager

After initialisation of the SCSI processor the Transputer waits for either an event from the SCSI processor, or a request from the buffer manager. The buffer manager requests inform the SCSI manager that data is available in the buffer for the host. Handling of the SCSI processor events is more complicated and requires considerable effort to ensure the process does not deadlock due to a mismatched state between the Transputer and SCSI processor. SCSI processor events are one of two forms, hardware and software. The hardware events typically signify SCSI hardware of bus errors, and these are handled at present by displaying an error message and resetting the processor.

Specific software events are signalled, via the ISTAT register, by the currently executing SCSI script. The software events are decoded and additional information, taken from the state of the SCSI processor, is used to invoke one of the following actions:

send: the SCSI controller has received a send command from the initiator. If space is available in the buffer then the command can continue with the data phase, else the SCSI processor is told to disconnect from the bus;

receive: the SCSI controller has received a receive command from the initiator. If data is available from the buffer the command continues with the transfer of data, else the SCSI processor is told to disconnect from the bus;

sent data: the data has arrived in the buffer and is available for processing;

received data: all data in the buffer has been transferred to the initiator;

command complete: the execution of the SCSI command is now complete and the processor has disconnected from the bus;

internal command: a SCSI command which does not require Transputer intervention has completed;

disconnected: the SCSI processor has completed the requested disconnection from the bus;

send reconnect: the SCSI processor has reconnected to the bus and is waiting to complete the send command;

receive reconnect: the SCSI processor has reconnected to the bus and is waiting to complete the receive command.

3.2 Managing the Transputer Links

Two processes handle the Transputer links, one for input and one for output. The operation of these processes is very simple and is no more than a loop with the operations (i) request buffer space, (ii) transfer data via the link, and (iii) free buffer space. If no data is available or the buffer is full, the process just waits until the buffer controller signals otherwise.

These processes run at high priority in order to ensure the links are operating at full capacity.

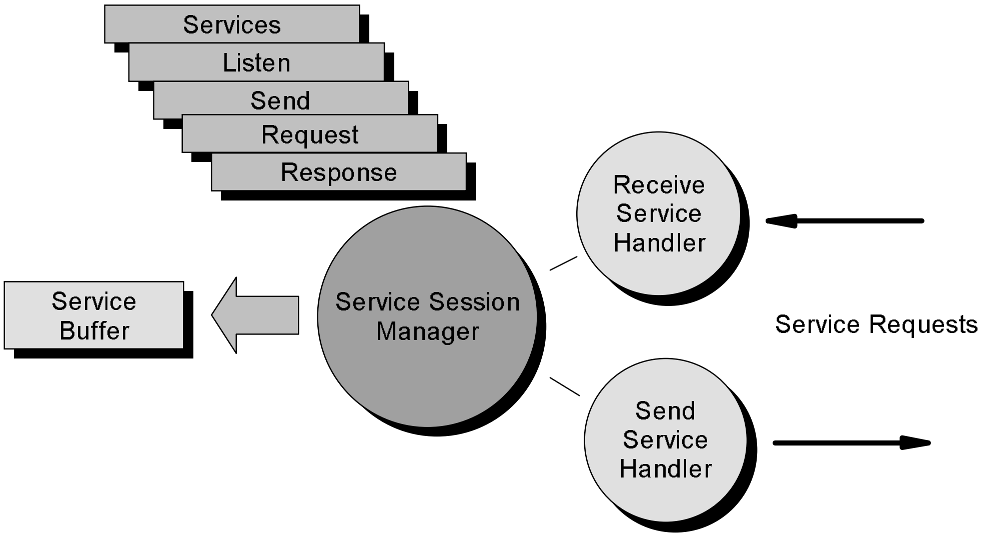

Figure 8: Service Handler

3.3 Service Handler

The service handler [4] manages the service requests from the host processor. Two processors (figure 8) manage the internal and external requests as defined in [4], and a common buffer manager holds the state information which is held during the lifetime of the request or service.

The service buffer manager keeps the state of each service type when necessary. All requests to update the service buffer are sent to this process. This ensures that the state of the service buffer is constant to both of the service handlers – no partial state of a service can be seen by any process. For example an incoming *request service* will either see a valid service or not during a concurrent *delete service* operation from the host.

3.4 Buffer Management

Some form of buffering is required in order to account for discrepancies in throughput of data at either of the data sources (Transputer links or SCSI bus). This buffer is shared by the link process, service handler, and SCSI driver (figure 9). A manager is required so that potential conflicts in accesses to the buffer by the SCSI controller and the link engines can be avoided.

Both the send and receive buffers operate in an identical manner. The buffer is made from three contiguous blocks. Thus at full capacity it is possible to input from a link, process a service request, and output on the SCSI bus simultaneously. The buffer process handles requests for buffer access via request/grant pairs of links. An internal semaphore mechanism protects access to shared resources such as the internal buffer position pointers.

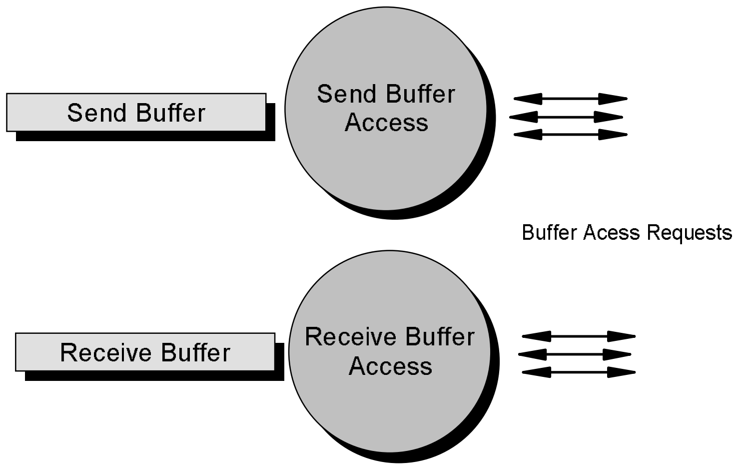

Figure 9: Buffer Manager

3.4.1 Buffer Access Control

Simple access to a shared buffer by a number processes is easily achieved by semaphores controlling the updating of buffer pointers, and a procedure based on these techniques was developed. The requirement of the SCSI controller to disconnect when data could not be transferred on the bus made the management technique more difficult and hierarchies of semaphores were developed to control the access. These techniques were eventually disregarded in favor of having a small management processes that handled the allocation of buffer space via requests through channels.

Figure 10 outlines the architecture for two processes sharing a buffer with a third process managing the buffer. To show how this management operates take the case of the SCSI driver receiving a *snd* command. When the command is first received the Buffer Manager is sent a message requesting a place in the buffer to receive the data. If the manager returns a valid address the transfer can take place without the target disconnecting from the bus. If an invalid address (NULL) is returned the target disconnects from the bus and waits for the manager to return a valid address. This case will happen only when the buffer is full.

The link process operates in a similar manner but requests for addresses always result in valid addresses of data to transfer.

The buffer manager handles two pointers to the top and bottom of the buffers and updates these as the SCSI driver inserts items into the buffer and the link process removes items. If the buffer is empty the process waits only for SCSI requests. If the buffer is full it will only return NULL to requests from the SCSI driver. The code fragment in figure 11 below shows this.

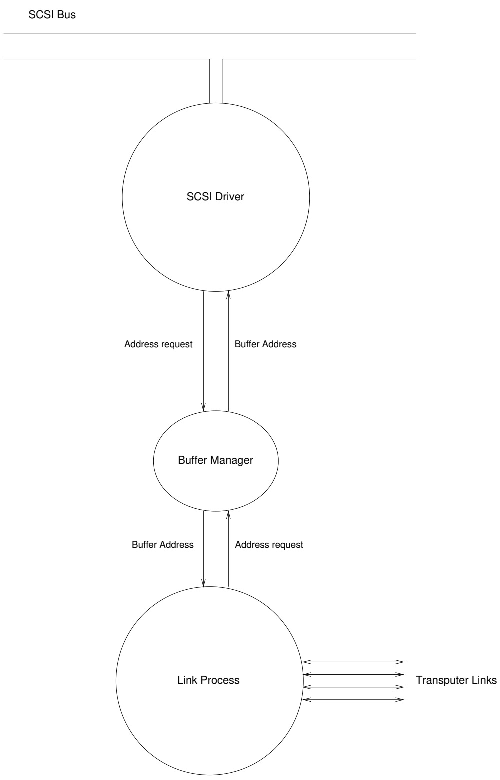

Figure 10: Buffer Management Using Concurrent Processes

```
WHILE TRUE
   IF buffer empty
       wait for address request from SCSI driver
       return valid address to SCSI driver
   ELSE IF buffer full
       send NULL addresses to SCSI driver requests
       send address to link driver request
   ELSE
       handle link driver and SCSI driver requests
```

Figure 11: Algorithm for Handling Address Requests for Buffer Access

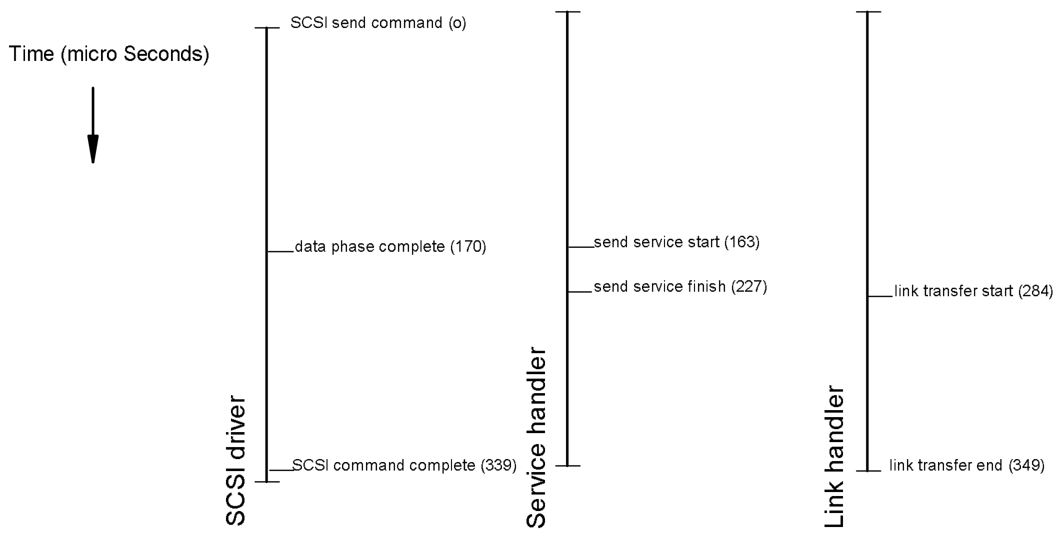

Figure 12: Client Performance for SCSI send transaction

The solutions provided above were developed for the SCSI *snd* command where the initiator sends data to the target device. For the reception of data (*rec* command) the nature of the code is simply reversed with the link driver filling the buffer, and the SCSI driver emptying it. The SCSI driver will disconnect from the bus when the buffer is empty rather than full.

4 SCSI Driver Performance

Timing figures for a client application are shown in figure 12. The size of the data buffer sent (send service) from the host and onto the Transputer links, is 26 bytes (2 bytes data, 24 bytes header).

It can be seen that between the completed data phase and the end of the link transfer that the service request has been processed and the data transferred to the neighboring Transputer. There is a time saving of 169 micro seconds due to not having to wait for the SCSI transaction to complete. This saving has been gained due to the parallelism offered by the NCR SCSI processor.

References

[1] NCR Corporation *NCR 53C710 SCSI I/O Processor Data Manual (rev 2.0).* NCR Corporation, Dayton, Ohio, USA, 1991.

[2] NCR Corporation *NCR 53C710 SCSI I/O Processor Programmer's Guide (rev 1.0).* NCR Corporation, Dayton, Ohio, USA, 1990.

[3] Geraint Jones *Efficient Multiple Buffering in occam.* occam User Group Newsletter, No 11, July 1989.

[4] Ian Penny *A connection orientated service handler.* UKC RPC Project Working Report, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF.

[5] ANSI. *Small Computer Systems Interface (SCSI).* X3-131-1986, American National Standards Institute, Inc, 1986.

[6] ANSI. *Small Computer Systems Interface (SCSI-II).* X3T9.2/86-109 Rev. 10h, American National Standards Institute, Inc, 1991.