NCR 53C710 SCSI I/O Processor

NCR

Programmer's Guide

Printing History

Revision No.	Print Date
Draft 0.1	9/90
1.0	10/90

SCSI SCRIPT™ is a registered trademark of NCR Corporation

NCR is the name and mark of NCR Corporation. While the information presented herein has been checked for accuracy, NCR assumes no responsibility for either its use or any damages resulting from its use. NCR reserves the right to make any changes or discontinue altogether without notice any hardware or software product or the technical content herein.

Table of Contents

Appendices

List of Figures

I/O Performance

The demands on today's I/O interfaces are being pushed by increased performance of personal computers and workstations. Extremely fast CPU's, both CISC and RISC, only provide marginal system performance if their I/O interfaces are not properly designed. Faster processors do not equal higher performance. Amdahl's Law describes this situation: "Assume I/O represents 10% of the system activity and its performance is kept constant. If CPU power is increased by a factor of 10:1, the net improvement is only 5:1. A 100:1 increase in CPU power is valueless if the net improvement in systems performance is only 10:1."

Interrupt service routines often take more than several hundred microseconds to execute and can be a large source of performance delays. Interrupts may be generated for exception conditions, I/O completion, saving/restoring buffer data pointers (for system check-point/restart), or low probability events available as options in todays SCSI definition. Interrupts can be reduced by using programmed I/O; however, this can be time consuming and requires much of the host computer cycle time. Therefore, programmed I/O is not an adequate solution for multi-tasking operations.

Scatter/Gather

Another performance issue is the scatter-gather operation. With virtual storage immensely common today, many I/O's gather the data from several physical addresses in system memory. Latencies inherent in the re-instruct DMA operation can cause serious performance degradation by allowing the disk drive to slip a latency while the DMA is being re-instructed.

I/O Flexibility

Options in bus protocol allow increased I/O flexibility. Need for I/O flexibility is partially responsible for the popularity of the SCSI standard. I/O flexibility allows configuration of systems for a wide range of peripherals (from high performance disk drives to hand held scanners). Additionally, I/O flexibility supports command queueing, asynchronous or synchronous data transfers, caching controllers, peer to peer communication, etc.. Unfortunately, this implies firmware complexity. If these options are not carefully implemented, performance will suffer.

A Better Solution

First generation (NCR 5380) SCSI devices are register oriented and require processor intervention to make the most fundamental protocol decisions. Users like the flexibility of these devices because the low-level firmware interface provides specific real time information about the SCSI bus and improved testability of the SCSI device. This generation of devices typically requires in excess of 4,000 lines of code to specify a SCSI-1 device implementation.

Second generation (NCR 53C90) SCSI devices provide on-chip state machines. Some complex SCSI sequences can be performed automatically which reduces protocol overhead. However, these devices have no decision making capability, because the internal sequences are fixed in hardware at VLSI design time. This generation of devices typically requires in excess of 2,500 lines of driver software to support this class of SCSI device.

The flexibility of the SCSI bus creates a dilemma for system integrators and OEM's alike. The dilemma is: should first and second generation SCSI devices be used as non-intelligent, stand-alone devices or should they be integrated into intelligent host adapter boards. Non-intelligent SCSI host ports or host bus adapters require a fair amount of processor intervention, but are inexpensive to implement. Intelligent host adapters are more expensive than non-intelligent adapters. They provide slower decision making capabilities (less powerful CPU's), experience interpretation delays (2-8 msec required to start any I/O), and suffer from interprocessor communication delays. In systems not requiring a complex buffering scheme, non-

intelligent host adapters outperform their intelligent counterparts. For peripheral controllers, space is at a premium and complex peripheral interfaces require powerful microprocessors to transfer data at the high rates used by the peripheral interface. Therefore, SCSI chips requiring intense firmware can overwork the controller microprocessor making it unable to perform required tasks. Limited available space usually excludes adding an extra processor or replacing it with a more powerful one.

With MIPs increasing in the system CPU, the delays caused by intelligent host adapter cards and slow peripheral controllers pose problems for the system integrator. The simplest solution is to build complex, versatile hardware sequences inside the SCSI components or to add additional CPU power in the SCSI device board. Both solutions are costly (space and component cost) and do not adequately address the problem.

Third Generation Requirements

To accommodate the flexibility requirements of the SCSI bus (reducing interrupts and controlling board cost), an additional level of intelligence and integration is required for next generation SCSI devices. Third generation SCSI devices must make execution decisions based on phase changes on the SCSI bus and compare specific incoming data values which will result in a minimum number of interrupts to the external processor.

A programmable SCSI device that executes SCSI oriented commands is required. These new devices must reduce interrupt service routine complexities by providing unique status values to the external processor for any interrupts that do occur. Additionally, a fully integrated DMA channel would allow full use of available host bus bandwidth. This is the key to overall I/O performance given current use of virtual memory schemes which require the ability to support scatter-gather memory operations without processor intervention.

Third generation SCSI devices require only a few hundred lines of driver code. This code is required for exception conditions and for passing addresses of the user data buffer to the device. Error recovery occurs at the high level interface. In second generation chips, the firmware is required to manage every detail of the error recovery mechanism, because the high level interface is fixed and has only one entry point. Programmable SCSI chips allow error recovery using the high level interface, because the algorithm can be entered at any command and error specific SCSI SCRIPTS™ can be developed.

Chapter 1
Introduction

This chapter introduces the NCR 53C710 SCSI I/O Processor (SIOP).

The NCR SCSI I/O Processor (SIOP)

The NCR 53C710 is the second member of a family of intelligent SCSI chips. A high-performance reusable SCSI core and an intelligent 32-bit bus master DMA have been integrated with a SCSI SCRIPTS processor to accommodate the flexibility requirements of SCSI-1, SCSI-2, and eventually SCSI-3. This flexibility is supported while solving the protocol performance problems that have plagued both intelligent and non-intelligent adapter designs.

SCSI Component

In addition to the reliability components of NCR's other SCSI chips:

- 10K volts ESD protection
- >350 mV Bus Hysteresis
- Immunity to bus reflections due to impedance mismatches
- Controlled bus assertion times which reduces generated RFI, improves reliability, and increases the chances for FCC approval
- Latch-up protection >100 mA
- Voltage feed-through protection

The SCSI core in the 53C710 is reusable and designed to migrate to SCSI-2 wide and fast requirements. It offers synchronous transfers up to 10 MBytes/sec with asynchronous transfers greater than 5 MBytes/sec. Synchronous offsets up to 8 MBytes/sec are supported.

The SCSI core offers low-level register access as well as the high-level control interface. Similar to first generation SCSI devices, the 53C710 SCSI core can be accessed as a register-oriented chip. The ability to sample and assert any signal on the SCSI bus can be used for manufacturing test and diagnostic procedures. Loopback diagnostics are supported; the SCSI core can perform self-selection and can operate as both an initiator and a target to verify that internal data paths are operational. The 53C710 can test the SCSI pins for physical connection to the board or the SCSI bus.

Unlike previous generation devices, the 53C710 SCSI core is controlled by the integrated DMA through a high-level logical interface. High-level programming language commands controlling the SCSI core may be chained from main host memory. These commands instruct the SCSI core to select, reselect, disconnect, wait for a disconnect, transfer user data, transfer SCSI information, change bus phases, and implement all aspects of the SCSI protocol.

Also, the SCSI SCRIPTS processor will transfer execution control (jump, call, return, and interrupt) based on SCSI bus phase comparisons. A value in the SCSI SCRIPTS command can be compared to the actual data value on the SCSI bus, allowing the same transfer of control based on input data compares. The SCSI SCRIPTS processor is a special 2MIPS processor located on the SCSI chip.

DMA Component

The DMA component is a bus master DMA chip that attaches easily to various processor buses and is designed to be externally adapted to ISA (AT), EISA, Micro Channel™, etc.

The 53C710 supports 32-bit memory and automatically supports misaligned DMA transfers. Data bus enables are provided for each byte lane. An on-chip, 64-byte FIFO allows 2,4, or 8-long words to be burst across the memory bus interface, providing memory transfer rates in excess of 66 MBytes/sec.

Sixteen bytes at a time can be burst into the FIFO using the cache line burst feature.

The DMA is tightly coupled to the SCSI core through the SCSI SCRIPTS processor, which supports uninterrupted scatter-gather memory operations with only a 500 nanosecond delay between memory segment transfers.

A Watchdog Timer provides a "bus safety" feature. The flexible arbitration scheme allows daisy chained or 'OR'ed memory bus request implementations.

SCSI SCRIPTS™ Processor

The SCSI SCRIPTS processor is a specially designed 2 MIPS processor that allows both DMA and SCSI instructions to be fetched from host memory. Algorithms written in the SCSI SCRIPTS language and then compiled control the SCSI and DMA cores and are executed from 32-bit system memory. Complex SCSI bus sequences are executed independently of the host CPU.

Using relative jumps and the Table Indirect Mode for fetching data values, SCSI SCRIPTS can be executed from a PROM.

The SCSI SCRIPTS processor can begin a SCSI I/O operation in 500 nsec. This compares to the 2-8 msec required for traditional intelligent adapters. The SCSI SCRIPTS processor offers performance and customization. By designing your own algorithms, you can tune SCSI bus performance, adjusting it to new bus device types (that is, scanners, communication gateways, etc.), or changes in the SCSI logical bus definitions; or you can quickly incorporate new or popular options.

The SCSI SCRIPTS processor is how the 53C710, the NCR third generation SCSI chip, implements flexibility without sacrificing I/O performance.

NCR SCSI SCRIPTS Description

SCSI SCRIPTS are independent of the CPU and system bus. SCRIPTS for an EISA bus implementation of a 80386 can, therefore, be identical to the SCRIPTS for a Micro Channel implementation.

After power up and initialization of the 53C710, the chip may be operated in one of two modes:

1) Low-level register interface

2) SCSI SCRIPTS mode.

In the low level register interface, you have access to the DMA control logic and the SCSI bus control logic and can operate the chip like an NCR 53C80. Access by an external processor to the SCSI bus signals and the low-level DMA signals, allows use of a complicated board level test algorithm. The interface provides backwards compatibility with SCSI chips requiring unique timings or bus sequences to operate properly. Another low-level feature is loop back testing. In loop back mode, the SCSI core can be directed to talk to the DMA core; this allows the internal data paths to be tested all the way to the chip's pad.

Operating in the SCSI SCRIPTS chained mode, the 53C710 requires *only* a SCSI SCRIPTS start address. All subsequent commands are fetched from external memory. Four bytes at a time are fetched across the DMA interface and loaded into the command register. Command fetch and decode time is minimal at about 500 nanoseconds.

In the Table Indirect Mode, data values (for example, byte count and address) are fetched after the command bytes are in the chip.

A Data Structure Address (**DSA**) register is provided for the data structure base address, and a 24-bit signed value is in the SCSI SCRIPT. Therefore, a complete context switch involves loading a new **DSA** value and then starting SCSI SCRIPT execution.

Commands are fetched until an interrupt command is encountered or until an external, unexpected event (e.g. hardware error detected) causes an interrupt to the external processor. The full set of SCSI features in the command set allow re-entry of the SCSI algorithm at any point. A high level interface is required for both normal and exception conditions. Therefore, switching to a low-level mode for error recovery, as is the case with today's second generation SCSI VLSI, is never necessary.

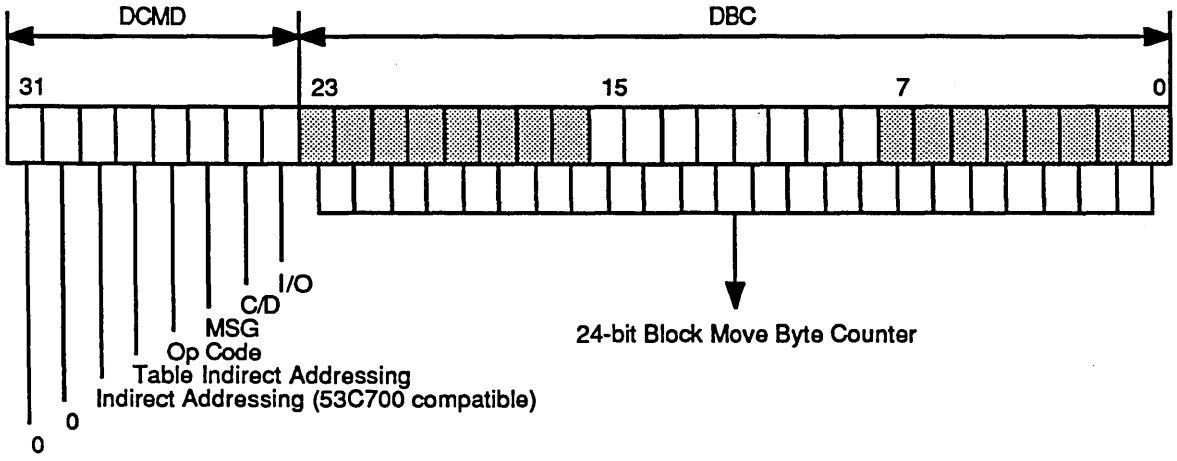
Chapter 2
SCSI SCRIPTS Machine Language Description

This chapter describes each SCSI SCRIPT in detail, at the programming and bit level. Normally, you will use the SCSI SCRIPTS compiler as described in the following sections, but for debugging purposes, each command is described in detail. Each command description consists of a bit diagram of the command, a brief overview of the command, and a description of each field within the command.

Bits 31-30 are SCSI I/O Processor opcodes:

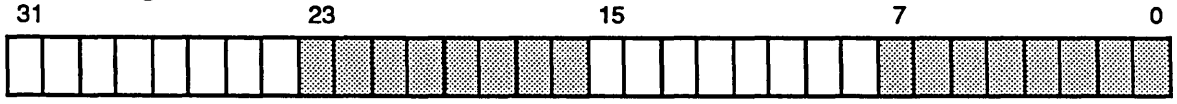
00 equals Block Move Command
01 equals I/O Command and Read/Write
10 equals Transfer Control Command
11 equals Memory Move Command

BLOCK MOVE COMMAND



First 32-bit word of the Block Move Instructions

DNAD Register



Second 32-bit word of the Block Move Instructions

Figure 1. Block Move Instructions

Overview

The Block Move command transfers data to(from) user memory from(to) the SCSI bus. No distinction is made between user data and SCSI information, such as command or message bytes. A series of SCSI SCRIPTS is written to move all types of data, with no requirement for separate firmware to distinguish between user and SCSI data.

Note that the data may come from any memory address, so scatter/gather operations for user data are transparent to the chip and the external processor. One simply writes a separate Block Move for each piece of data to be moved. Use the 32-byte DMA data buffer to speed data transfers between user memory and the I/O Processor. Synchronous SCSI data in transfers can use the 8-byte FIFO.

Note: The possible values for each field are given in binary.

BLOCK MOVE COMMAND FIRST SCRIPTS WORD

Block Move opcode -- 00 Bits 31-30

Indirect data address flag (I), Bit 29

0 SCSI or user data is moved to (from) the 32-bit data start address for the Block Move. The value is loaded into the chip's address register and incremented as data is transferred.

1 The 32-bit SCSI or user data start address for the Block Move is the address of a pointer to the actual data buffer address. The value at the 32-bit data start address is loaded into the chip's **DNAD** register via a second long word (four-byte transfer across the host computer bus).

This option implies three DMA long word transfers, rather than two transfers. Once the data buffer address is loaded, it is executed as if the chip operates in the direct mode. This indirect feature allows specification of a table of data buffer addresses. Using the NCR SCSI SCRIPTS compiler, the table offset is placed in the script at compile time. Then at the actual data transfer time, the offsets are added to the base address of the data address table by the external processor. This allows the logical I/O driver to build a structure of addresses for an I/O rather than treating each address individually.

Table Indirect Field, Bit 28

0 SCSI or user data is moved as described previously. This option allows compatibility with existing 53C700 SCSI SCRIPTS.

1 The 32-bit start address is treated as a 24-bit signed value. After the command is moved into the 53C710, the 24 bits are added to the Data Structure Address (DSA) register to form a 32-bit physical address.

From this new address, the byte count (24 bits of count, plus 8 bits of high-order zeros), and the Data Buffer Address (32 bits of address) are fetched.

There are several programming implications of this feature.

First, a standard SCSI data structure can be designed with values in predefined areas. The SCSI SCRIPT does not require the actual 32-bit address or 24-bit count to be in the SCRIPT itself. At the start of the an I/O, once the actual data structure is built, no more firmware intervention is required except loading the data table base address into the **DSA** register.

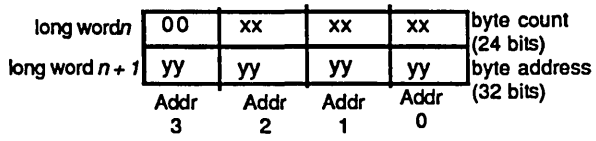
Second, the SCRIPT may be placed in a PROM because no dynamic alteration is required at the start of an I/O.

Finally, there is a requirement for only one copy of the main SCSI SCRIPT for all I/O, using a fast context switch to change to another I/O.

In the Table Indirect mode, the user must have stored the byte count and data address in memory formatted as shown in the illustration following this description.

The data must begin on a 4-byte boundary and must be located at the 24-bit signed offset from the address contained in the Data Structure Address register.

If the data is written to memory, four bytes at a time from the processor, then the user need not be concerned about big or little Endian mode because the low order byte will automatically be at the low order address. If this is not the case, the user must ensure that the bytes are in the proper order (that is, low order byte at address zero; next byte at address 1, etc.)



Block Move Opcodes, Bit 27

The SCSI role (target or initiator) causes the chip to react differently, with respect to the phase line values. A primary difference between roles is whether the SCSI phase lines are sensed or driven. There are also major differences between the two roles in the command phase. Therefore, the Block Move functions are described for each SCSI role - target and initiator.

Target Role Function--0

The target role allows DMA of user or SCSI data. First the chip determines whether the previous command has completed, or a reselect has occurred. The SCSI phase bits are asserted to the value requested by the Block Move command.

In all phases, the chip will react one of several ways, after the SCSI SCRIPT is loaded.

If the Indirect Addressing bit is 1, the 53C710 fetches the Data Buffer Start Address from the location pointed to by the DMA Next Address (**DNAD**) register. This fetched value is then stored in the **DNAD**, and execution begins.

If the Table Indirect bit is 1, then the byte count is fetched, and the buffer address is fetched.

An address for these values is generated using the 24-bit signed value in the start address field of the SCSI SCRIPT, and the value of the **DSA** register.

Note:
Setting both the Indirect Addressing and Table Indirect bits to 1 causes an illegal instruction.

If the command phase has been requested, the chip will:

- Wait for the first byte received.
- Decode the byte to determine the number of SCSI command bytes to receive.
- Write the command length into the **DBC** register.

  An invalid group code value causes the chip to use the original value in the **DBC** register. A zero value stops processing, creates an interrupt with the first byte, and stops transferring command bytes.
- Transfer the correct number of bytes into the address designated by the Block Move command.

If any phase (other than command) is requested, the chip transfers the number of bytes requested to(from) the address requested. Should the initiator turn on attention at any time during the transfer, the transfer will optionally complete, and then an interrupt will occur.

Target Role Function--1

This is an illegal value and will generate an invalid command interrupt if the chip is in the target role.

Initiator Role Function--00

This is an illegal value and will generate an invalid command interrupt if the chip is in the initiator role.

Initiator Role Function--1

In the initiator role, this operation waits for a valid phase and DMA data. After verification that the previous command is complete or a

reselect has occurred, the chip waits for a previously unserviced phase before executing the Block Move command. You can program the 53C710 to pause until the SCSI device it is communicating with goes to the next phase, using the Transfer Control commands or the Move instructions.

A comparison is made between the expected phase bits in the SCSI SCRIPTS and the latched phase value. If the two values are not equal, the chip issues a phase mismatch interrupt and halts execution. This wait capability is normally used to allow the target to pace the chip in the initiator role. When a phase change is expected, the wait synchronizes the expected phase with the Block Move for that phase.

To eliminate the possibility of these interrupts, use the compare and jump features to verify the phase before issuing the Block Move command.

Please refer to the previous discussion of how the table indirect or indirect address features cause the chip to load byte count and buffer address.

SCSI Phase Lines, Bits 26-24

These three SCSI phase lines perform comparisons to the actual SCSI bus phase lines. The SCSI bus phase value is latched when REQ goes active. The value is stored in SSTAT2 (bit 2 through bit 0 -- MSG, C/D, & I/O). Before any data is moved, the chip compares the expected value with the actual value.

24-bit Byte Count, Bits 23-00

This count value specifies the exact number of data bytes to be moved between the SCSI bus and system memory. As the SCSI SCRIPTS command is decoded, the value is moved into the **DBC** register. When the user specified burst size of data is available in the DMA FIFO, the SCSI I/O Processor will:

- Gain access to the system bus.
- Transfer the burst size.
- Decrement the byte counter (byte count).
- Increment the next address register (data address).

The process will continue until the byte count is zero. At that time, the next SCSI SCRIPTS command will be fetched.

If the chip is in Table Indirect mode, the byte count will be fetched from the memory address formed by adding the Data Structure Address (DSA) register to the 24-bit signed value in the Start Address register.

BLOCK MOVE SECOND SCRIPTS WORD

Data Start Address for the Block Move Bits 31-00

This value specifies the address of data in memory (direct mode), the address of the actual address (indirect mode), or the 24-bit signed offset from the Data Structure Address register (Table Indirect mode). The **DNAD** register is updated with the address of the actual data and is incremented with each chip DMA transfer.

The Block Move command is very powerful for several reasons.

1) No distinction is made between user data and SCSI command, message, or status data.

2) Data can be stored in any area of system memory with little performance impact (one command fetch).

3) The indirect feature allows a table of addresses instead of requiring the address to be in the command.

4) A scatter/gather operation has little performance impact, because the only overhead is 500 nanoseconds (direct mode) or 750 nanoseconds (indirect mode). Thus, one Block Move

command for each segment of data in memory is economical with the SCSI I/O processor architecture.

The Table Indirect mode allows both byte count and Data Buffer address to be fetched from system memory. Having this information brought into the chip, in the indirect mode, causes 8 more bytes of information to be fetched and separates data from SCRIPTS code.

In the initiator role, the Block Move wait feature is useful for high performance SCSI SCRIPTS that do not compare for any unexpected phases before executing a Block Move command. If the phase does not match, then an external interrupt is generated.

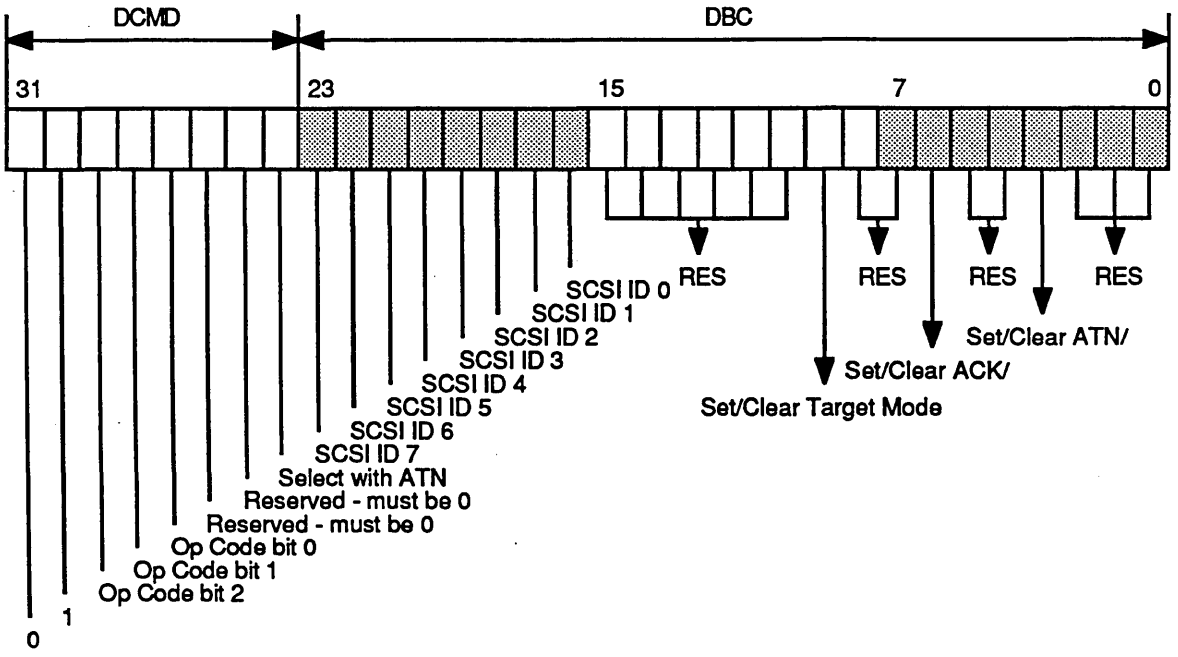
For the high performance SCSI SCRIPTS algorithm, exceptions are abnormal and are handled by the external processor. Normally, the Conditional Transfer command (see I/O Command) compares actual to expected phase. The first Conditional Transfer command must have the "wait" option on (to synchronize the commands with the actual bus phase), and each subsequent command should have the "wait" option turned off.

With the Table Indirect mode, I/O data structures can be fetched directly, eliminating one more level of system software translation normally required to start an I/O.

In this mode, SCRIPTS do not need to be patched at the start of an I/O. Once a standard I/O descriptor has been described by a SCSI SCRIPT, it can effectively execute the data structure with no processor intervention.

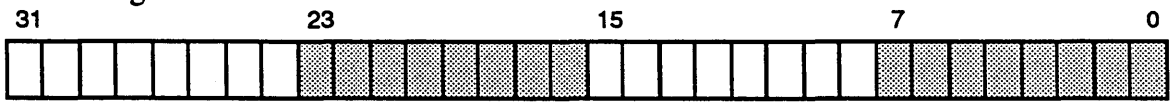
For another method of placing a 32-bit address into this instruction, refer to the PASS option available in the SCSI SCRIPTS compiler.

I/O COMMAND



First 32-bit word of the I/O Instructions

DSPS Register



Second 32-bit word of the I/O Instructions

Figure 2. I/O Instructions

Overview

The I/O command performs SCSI operations such as select and reselect. Each function defined is a direct command to the SCSI portion of the 53C710. The functions vary if the chip is in the target or initiator role, so that the functions are described separately for each role.

A new set of register-to-register operations has been defined for this opcode.

I/O COMMAND FIRST SCRIPTS WORD

SCSI I/O Processor opcode -- 01 Bits 31-30

I/O Command Opcodes Bits 29-27

Five functions are defined for target and initiator role, three are used in register operations.

Target Role -- function 000
Perform reselection -- The chip arbitrates for the SCSI bus and then performs a reselection. Arbitration continues until the chip is successful, unless there is a bus initiated interrupt (e.g. selection). If arbitration terminates because of a bus initiated interrupt (selection or reselection) the chip will use the 32-bit jump address value to fetch the next instruction and begin execution at that address.

If the relative addressing bit is 1, then the 24-bit signed value in the DMA Next Address register is used as a relative displacement from the DMA SCRIPTS pointer. If the command is successful, then the next sequential instruction is fetched and executed.

Note that the target/initiator role automatically changes to reflect what is actually occurring on the bus, unless bit 0 (COM) of the DCNTL register is set.

If the Table Indirect mode bit is 1, the 24-bit signed value in the DMA Byte Count register is used as an offset relative to the Data Structure Address register. The SCSI destination device ID, the synchronous offset, and the synchronous period are loaded from the formed address. Using this indirect mode, the SCRIPTS program can set the values stored with the I/O data structure and not require the user to alter SCRIPTS instructions at the start of an I/O. Upon reselect, the synchronous offset and period can be set using register writes, with no need to cause an external interrupt.

After completion of the bus initiated interrupt processing (sequence goes to bus free), the chip reverts to the role set by the user in the registers. Some caution is required here. If the chip is set to an initiator role, gets selected, changes to the target role automatically, disconnects, does some processing, and then issues a reselect command (without being set to the target role by the external processor or by a SCRIPTS register write), a selection will occur. Because the chip was in the initiator role (at the time of selection), it reverts to that role after the disconnect and bus free. See the description of the COM bit (DCNTL, bit 0) for a mechanism to turn off auto-switching.

Target Role -- function 001
Perform disconnect -- The chip physically disconnects from the SCSI bus.

Target Role -- function 010
Wait for select -- The chip waits for a SCSI selection by another device on the SCSI bus. If the chip is already selected, then the next SCSI SCRIPTS is fetched and executed. When a bus initiated interrupt or reselect occurs, the chip optionally changes to the initiator role and fetches the next command from the address pointed to by the 32-bit jump address, and continues execution.

If the relative addressing bit is 1, then the 24-bit signed value in the DMA Next Address register is used as a relative displacement from the DMA SCRIPTS pointer.

Target Role -- function 011
Assert bit -- The chip asserts the latches in the SCSI output data register, but nothing is driven onto the SCSI bus. Consequently, this function should not be used in the target role.

Target Role -- function 100
Reset bit -- The chip resets the latches in the SCSI output data register, but nothing is reset on the SCSI bus. Consequently, this function should not be used in the target role.

Initiator Role -- 000
Perform selection -- The chip arbitrates for the SCSI bus and then performs a selection. Arbitration continues until the chip is

successful or a bus initiated interrupt (e.g., reselection) occurs. If arbitration terminates because of a bus initiated interrupt (as a result of a select or reselect), the chip uses the 32-bit jump address to fetch the next instruction and begin execution at that address.

If the relative addressing bit is 1, then the 24-bit signed value in the DMA Next Address register is used as a relative displacement from the DMA SCRIPTS pointer.

If the command is successful, then the next sequential instruction is fetched and executed.

Optionally, the target/initiator role automatically changes to reflect bus actions (see the description of the DCNTL COM bit).

If the Table Indirect mode bit is 1, the 24-bit signed value in the DMA Byte Count register is used as an offset relative to the Data Structure Address register. The SCSI destination device ID, the synchronous offset, and the synchronous period are loaded from the formed address. Using this indirect mode, the SCRIPTS program can set the values stored with the I/O data structure and not require the user to alter SCRIPTS instructions at the start of an I/O. Upon reselect, the synchronous offset and period can be set using register writes, with no need to cause an external interrupt.

After completion of the bus initiated interrupt processing (sequence goes to bus free), the chip reverts to the role set by the user. If the selection is successful, the next instruction is fetched and executed. If bit 24 (the attention flag) is set, then the chip performs a select with attention.

Note:
Because the chip automatically changes roles and jumps to an alternate address if the select or reselect fails, a bus initiated interrupt can be processed by the chip with no external intervention. The alternate jump address should contain the address of an algorithm for a selection or reselection. Include in the address a wait for selection (target role) command. That command's alternate address is the reselection algorithm (initiator role). The 53C710 can determine exactly what happened and transfer control to the appropriate SCSI SCRIPTS algorithm. See Appendix C for another solution to this problem.

Initiator Role -- 001
Wait for disconnect -- The initiator waits for a disconnect from the SCSI bus. A legal disconnect is defined as a loss of busy and select for the specified bus free time following a DISCONNECT message or a COMMAND COMPLETE message. If the disconnect is legal, the next SCSI SCRIPTS command will be executed, otherwise an unexpected disconnect interrupt will be generated.

Note that the interrupt will occur on an abort message to a target to abort a SCSI command when the aborted target goes to bus free.

Initiator Role -- 010
Wait for reselection -- The initiator waits for a reselection from a previously selected SCSI device. If the operation completes as expected, then the next instruction is fetched and executed by the 53C710. However, if the chip is selected, then the alternate jump address should contain the address of an algorithm for a selection. Include in the address a wait for selection (target role) command. That command's alternate address is the error recovery algorithm (for initiator role -- reselect). The chip can determine exactly what happened and transfer control to the appropriate SCSI SCRIPTS algorithm.

If the relative addressing bit is 1, then the 24-bit signed value in the DMA Next Address register is used as a relative displacement from the DMA SCRIPTS pointer. If the command is successful, then the next sequential instruction is fetched and executed.

Note:
With the 53C710 byte compare capability of the transfer control command, the SCSI SCRIPTS algorithm can determine which target reselected the initiator and can jump to the correct algorithm for that particular target. SCSI SCRIPTS can be tuned for the various types of targets available and executed with no external processor intervention.

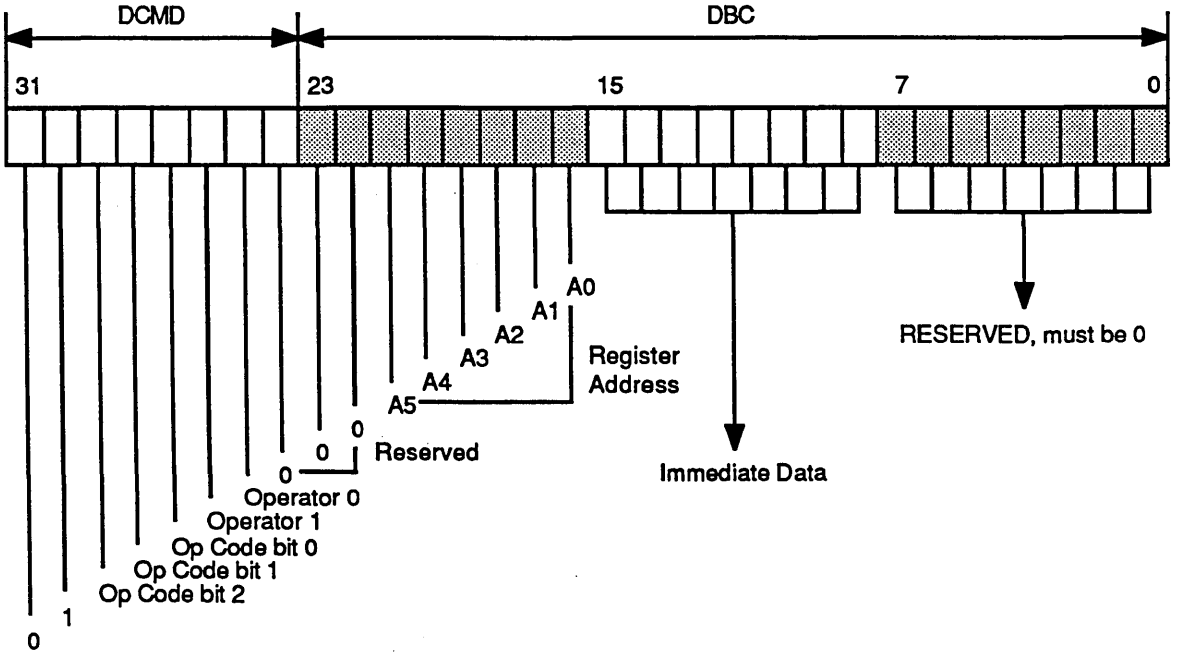
See chapter 10, "Multi-tasking I/O" for more discussion of this subject.

Initiator Role -- function 011
Assert bit -- The chip asserts the SCSI bus bits requested in the flags field. Currently three bits are defined, allowing the SCSI ACK target role and ATN bits to be set. Bit 10 is for target, bit 6 is for Acknowledge, and bit 3 is for Attention.

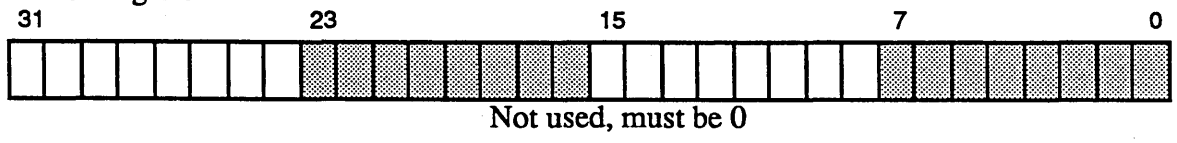
Initiator Role -- function 100
Reset bit -- The chip resets the SCSI bus bits requested in the flags field. Currently two bits are defined, allowing the SCSI ACK target role and ATN bits to be reset. Bit 10 is for target, bit 6 is for Acknowledge, and bit 3 is for Attention.

Note that these bits can also be set or reset with the read/write register functions.

READ/WRITE REGISTER COMMAND



First 32-bit word of the RD/WR Register Instructions



Second 32-bit word of the RD/WR Register Instructions

Figure 3. Read/Write Register Instructions

Overview

In either initiator or target role, the opcode bits 101, 110, and 111 are for a set of register operations. The three opcodes are modified by the operator field (bits 26-25).

The opcode bit operations are:

Function 101

Move the SCSI First Byte Received register (**SFBR**) to the specified register. Four operator field values alter the meaning of the function. They are:

(Function 101, continued)

00 Move immediate data value to the destination register value.

01 Or the immediate data value with the **SFBR,** and write the result to the destination register.

10 And the immediate data value with the **SFBR** and write the result to the destination value.

11 Add the immediate data value with the **SFBR** and write the result to the destination register.

Function 110

Move the specified register value to the SCSI First Byte Received (**SFBR**) register. Four operator field values alter the meaning of the function. They are as follows:

00 Move immediate data to the **SFBR**

01 Or the immediate data value with the specified register and write the result to the **SFBR**

10 And the immediate data value with the specified register and write the result to the **SFBR.**

11 Add the immediate data value to the specified register and write the result to the **SFBR.**

Function 111

Read a specified register, modify it, and write the result back into the register. Four operator field values alter the meaning of the function.

00 Move immediate data to the specified register.

01 Or the immediate data value with the specified register and write it back to the specified register.

10 And the immediate data value with the specified register and write it back to the specified register.

11 Add the immediate data value to the specified register and write it back to the specified register.

The following table is a summary of the possible operations allowed.

Operator Field	OpCode 7 Read modify Write	OpCode 6 Move to SFBR	OpCode 5 Move from SFBR
00	Immediate data to destination register	Immediate data to SFBR	Immediate data to destination register
01	Immediate data or'ed with destination register	Immediate data OR register to SFBR	Immediate data or'ed with SFBR to destination register
10	Immediate data and'ed with register	Immediate data AND register to destination register	Immediate data and'ed with SFBR to destination SFBR
11	Immediate data added to destination register	Immediate data added with register to SFBR	Immediate data added with SFBR to destination register

Register Address Field, Bits 21-16

These bits select one of the 8-bit registers in the 53C710 to serve as source, destination, or immediate register.

Immediate Data Field, Bits 15-8

These bits contain any immediate data that is to be used in the operation specified by the instruction.

The second 32-bit register in the instruction is not used in the operations, but it should be zero to ensure compatibility with future instructions that may be defined.

Having a read/write register capability in the 53C710 adds a new dimension of SCRIPTS programming capability.

Several examples of how useful this capability are explained in the following.

1) Set synchronous offset and period for a target upon reselection. This operation will typically require an interrupt to an external processor. A 53C710 SCRIPT will be able to write an immediate value to the correct register once the reselecting device ID is decoded, and resume data transfer immediately.

2) Write an interrupt service routine in the SCSI SCRIPTS. After the external interrupt is serviced, the processor SCRIPTS program can determine the number of bytes left in the chip-check status bits, and in general, can clean up after an interrupt.

3) Keep a loop counter. Using the Add instruction, the number of times through a loop can be counted and stored. Thus, a Do Loop construction can be programmed using SCRIPTS.

Many other uses can be discovered. With the 53C710, a user can write a SCRIPTS program that will perform most of the operations done in external processor firmware.

Relative Addressing Mode, Bit 26

When this bit is set to 1, the 24-bit signed value in the DMA Next Address register is used as a relative displacement from the current DMA SCRIPTS Pointer register.

Using this mode, the 32-bit physical address is formed at execution time, and there is no need to relocate a SCRIPT at system power-up. This bit may be used with select, reselect, wait_select, and wait_reselect commands.

Table Indirect Mode, Bit 25

When this bit is set to 1, the 24-bit signed value in the DMA Byte Count register is used as an offset relative to the value of the Data Structure Address register. Using this feature allows the SCSI device ID synchronous offset and synchronous period to be fetched from an I/O data structure that is built at start I/O. Thus, an I/O can begin with no requirement to write the values into the chip or into the actual SCRIPT in memory. In the I/O data structure the user must have written a four-byte value of:

00	Device i.d.	Period & offset	00
Addr 3	Addr 2	Addr 1	Addr 0

Device ID is the same format as SCSI destination ID register (02), and the period offset must be the same as the SCSI Transfer register (05).

The data must begin on a four-byte boundary and must be located at the 24-bit signed offset from the address contained in the Data Structure Address register.

If the four bytes are written from the processor into memory as a unit (one long word), then the user need not be concerned about big or little Endian mode. The low order byte must be at address zero, next byte at address one, and so forth.

Only one bit may be on in the Device ID byte. Synchronous offset is a 4-bit value, and synchronous period is a 3-bit value with the high order bit equal to zero. Refer to the

SCSI Transfer register (address 05) for a complete description of these bits.

SELECT WITH ATN - Bit 24

If bit 24 is set, then the initiator SELECT command will cause the SCSI attention line to be set during the SELECT operation. Attention on is valid only during the initiator function 000. The bit is invalid for all other functions and will cause an interrupt.

SCSI ID 7-0 - Bits 23-16

This eight bit field is the ID for the SCSI chip to be selected in the initiator role and reselected in the target role. Set only one bit for either of the functions requested. These bits are not used for any function other than select or reselect.

Flags Field - Bits 15-00

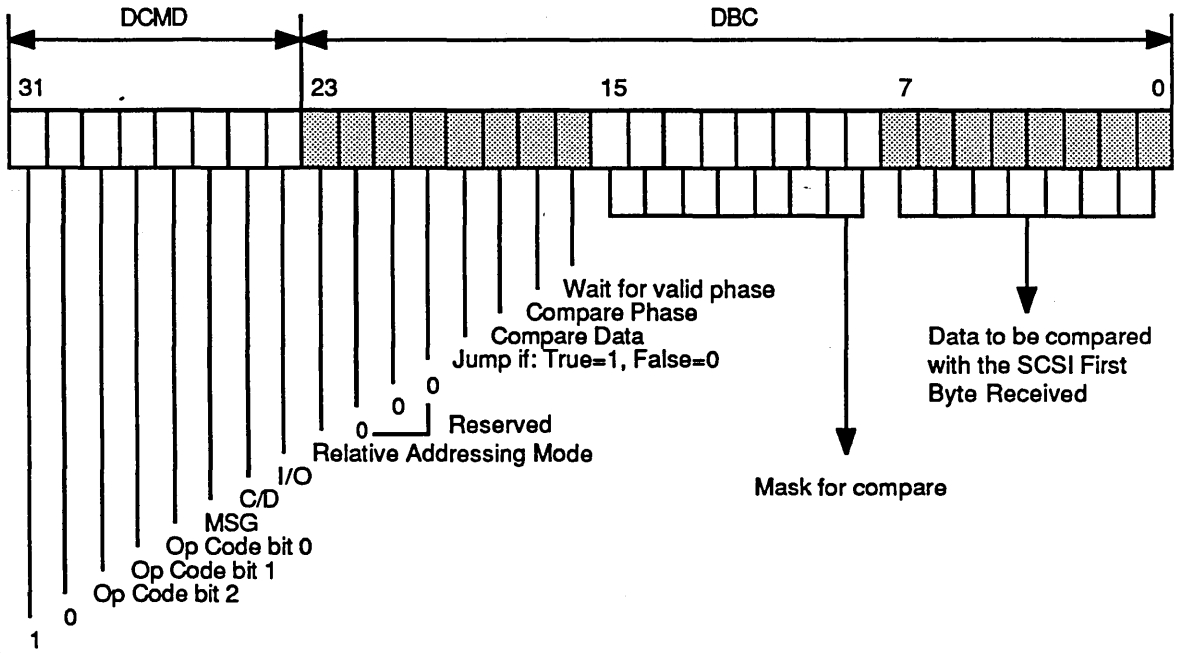
These bits are used during the set or clear command. Bit 10, on places the chip in the target/initiator role. Bit 6, on sets/resets the SCSI acknowledge. Bit 3, on sets/resets the SCSI attention. Use the clear ACK command after the last target message-in byte has been verified for each separate message data Block Move command. The initiator has the opportunity to set attention before acknowledging the last message byte of a Block Move command. On each byte, if a parity error was detected on the message in operation, the ASSERT SCSI ATN is issued before the clear acknowledge is issued to accept the message. Use Set Acknowledge to handshake bytes across the SCSI bus.

I/O COMMAND SECOND SCRIPTS WORD

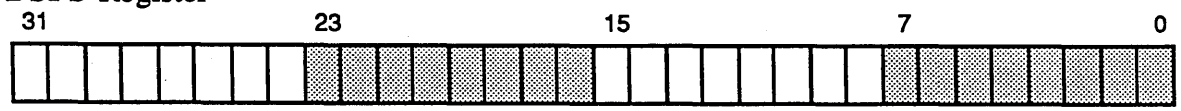
Jump Address - Bit 31-00

If the select, wait reselect, or reselect command fails, this thirty-two bit field specifies from which memory address to fetch the next SCSI SCRIPTS for execution. Normally, the next instruction is fetched in sequence if the requested operation completes with no bus initiated interrupt.

Transfer Control Command



First 32-bit word of the Transfer Control Command



Second 32-bit word of the Transfer Control Command

Figure 4. Transfer Control Command

Overview

The Transfer Control Command contains the JUMP, CALL, RETURN, and INTERRUPT operation codes. Each opcode is conditionally performed based on compare of SCSI phase values and incoming SCSI data values.

The Transfer control command allows comparisons of current phase values on the SCSI bus or the first byte of data on any incoming bytes and transfers control to another address depending on the results of the test.

These commands allow SCSI algorithms to be written in SCSI SCRIPTS and give the 53C710 characteristics of a general purpose SCSI processor. With transfer control commands, you can program the chip, rather than simply buffering commands to be serially executed with no real-time decision making capabilities.

TRANSFER CONTROL COMMAND, FIRST SCRIPTS WORD

SCSI I/O Processor opcode -- 10 Bits 31-30

Transfer Opcodes - Bits 29-27

Four opcodes are currently defined that allow a transfer of control in the SCSI SCRIPTS language. All undefined opcodes cause an interrupt of illegal command.

JUMP Command -- 000
If the condition evaluates according to the sequence control bits so the jump must be taken, the next instruction is fetched from memory at the 32-bit jump address. Otherwise, the next sequential address will be used as the instruction fetch address.

CALL Command -- 001
If the condition evaluates according to the sequence control bits so the call must be taken, the next instruction is fetched from memory at the 32-bit call address. Otherwise, the next sequential address will be used as the instruction fetch address.

The address of the next sequential command is stored in the chip's TEMP register in anticipation of a subsequent return address. If two CALL instructions are executed without any intervening RETURN instruction, then the first return address in the chip's TEMP register is overwritten by the second CALL.

Note that a call to an exit point, followed by an interrupt at the exit point, will supply the address (in the Temp register) of which execution path led to the exit.

RETURN Command -- 010
If the condition evaluates according to the sequence control bits so the return must be taken, the next instruction will be fetched from memory at the 32-bit address contained in the TEMP register, where it was stored by the previous call instruction. Otherwise, the next sequential address will be used as the instruction fetch address. The contents of the TEMP register may be undefined if a call instruction was not previously executed.

INTERRUPT Command -- 011
If the condition evaluates according to the sequence control bits so the software interrupt must be taken, the chip halts execution and issues an interrupt request to the external processor. Otherwise, the next sequential address will be used as the instruction fetch address.

The 32-bit jump address in the instruction is available in the chip's command register at the time of the interrupt. You can post a four byte, user unique error status to be used by the external processor's interrupt service routine. Thus, the cause of the interrupt can be easily decoded by firmware which reduces interrupt service routine overhead. Also, the value could be a 32-bit firmware (or a SCRIPT) address.

SCSI Phase Bits - Bits 26-24

In the SCSI initiator role, these bits compare the actual SCSI lines (MSG, C/D, and I/O), if the phase compare bit is set in the sequence control field. Actual SCSI lines are a copy of the last valid SCSI phase line values. These bits are set in the SCSI SCRIPTS command to compare with the current SCSI bus phase lines, then branch to the SCSI SCRIPT™ that processes the particular phase that is currently active. Bit 26 is SCSI MSG, bit 25 is SCSI C/D, and bit 24 is SCSI I/O. In the target role, these bits are ignored.

Relative Addressing, Bit 23

For the JUMP command or the CALL command, the chip can execute a relative transfer. The 24-bit signed value in the DSPS register is used as a relative offset from the DMA SCRIPTS Pointer register.

Bits 22-20

These bits are reserved for future use and must be zero.

Sequence Control Bits - Bits 19-16

SCSI SCRIPTS can use the current conditions on the SCSI bus to determine where to transfer control and execute alternative algorithms using the sequence control bits. The bits are defined as follows:

- Bit 19 -- Transfer if True/False.

  If the bit is set to 1, a transfer of control occurs if the phase or data values in the instruction are equal to the actual phase value on the SCSI bus or the first byte of the most recent asynchronous in phase. The byte could be a message in, data in, or status for the initiator and message out, command, or data out for the target role. When the bit is set to zero, the transfer control will occur if the comparison yields a false.

- Bit 18 -- Compare the data byte value (bit 7 - bit 0 in the instruction) to the first byte of the most recent data, message, command, or status byte received.

  The user's SCSI SCRIPTS program can determine what routine to execute next, based on actual data values received across the SCSI bus. For example, the chip can compare for specific message values and process an extended message in SCSI SCRIPTS, with no external interrupt to the external processor.

- Bit 17 -- In the initiator role, compare the SCSI phase line value (bit 26 - bit 24) to the recent valid SCSI phase line values saved in the chip.

  Using this feature, the chip can react to actual bus conditions and determine which routines to execute next based on SCSI bus phase line values. Unexpected phase values can be compared for and error conditions or low probability events can be processed by SCSI SCRIPTS inside the chip.

  In the Target role, bit 17 ON causes the chip to test for the attention line on. If the initiator has set attention, the chip (in the target role) can jump to a message out routine to determine what the initiator needs. This is normally placed after each SCSI phase to allow the initiator to turn on attention if an error is detected during the transfer.

- Bit 16 -- In the initiator role, wait for a previously unserviced phase change.

  You can program the chip to pause until the SCSI device it is communicating with has proceeded to the next phase. One normally uses this wait capability to pace the chip in the initiator role. When a phase change is expected, the wait is used to synchronize the expected phase with the actual phase detected on the SCSI bus. If both data and phase compare bits are set, the compare must be both true or both false for the transfer to occur.

Mask Bits - Bits 15-8

The mask bits allow selective comparison of bits within the data byte using SCRIPTS. During the compare, any bits that are on will cause the corresponding bit in the data byte to be ignored for the comparison. A user can code a binary sort to quickly determine the value of a byte.

For instance, a mask of '7F' and data compare of '80' allows the SCRIPTS processor to determine whether or not the high order bit is on.

Data Byte - Bits 7-0

Compare this data byte value to the first byte of the most recent asynchronous data, message, command, or status byte received. The user's SCSI SCRIPTS program can determine what routine to execute next based on actual data values received. Using a series of these compares, the algorithm can process complex sequences with no intervention required by the external processor.

TRANSFER CONTROL COMMAND, SECOND SCRIPTS WORD

Data Jump Address - Bit 31-00

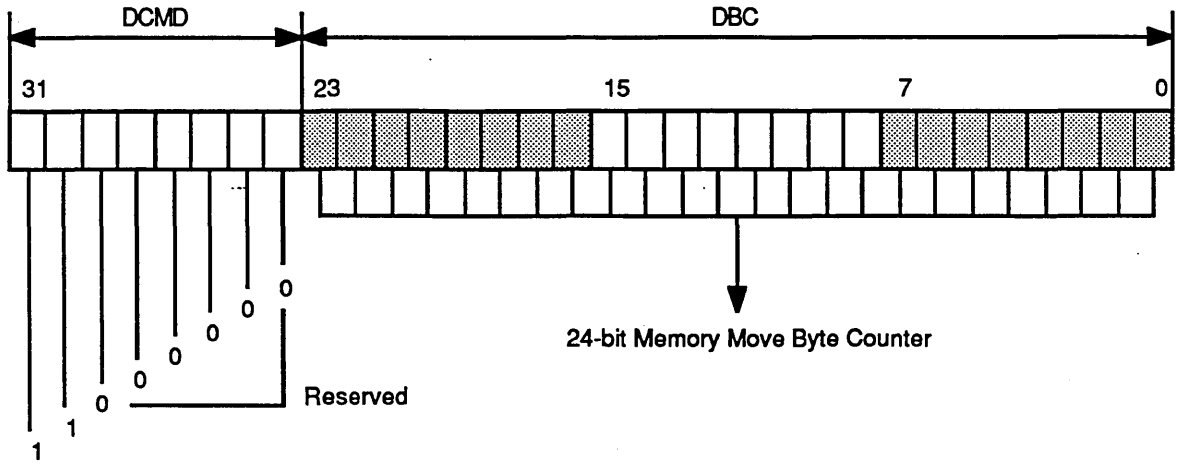
This value specifies the address of the next instruction in memory to transfer control. It is either a 32-bit physical address, or a 24-bit signed value, used as an offset from the DMA SCRIPTS Pointer register. The value is ignored in both return and interrupt commands. However, the address is loaded into the chip's command register and is available to be read by firmware in the case of an interrupt command.

If both data compare and phase compare bits are set, then both comparisons must be true or both must be false before the requested transfer will occur. There is no way to test one for false and the other for true.

If neither the phase or data bit are set, and if the true/false bit is 1, the operation is executed unconditionally.

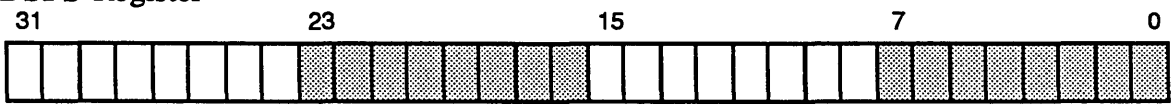
If neither the phase nor the data bit is set and the true/false bit is 0, then the command has no operation assignment and can be used as a delay function, or to reserve a SCSI SCRIPTS patch area.

Memory-to-Memory Move Instruction



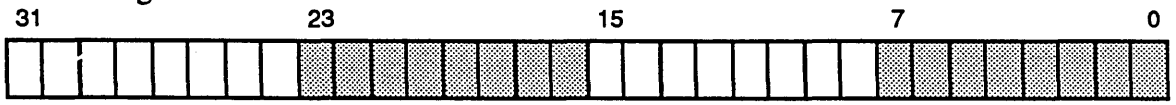
First 32-bit word of the Memory Move instruction

DSPS Register



Second 32-bit word (source address) of the Memory Move instruction

TEMP Register



Third 32-bit word (destination address) of the Memory Move Instruction

Figure 5. Memory-to-Memory Move Instruction

Overview

The Memory Move command is able to transfer data from one 32-bit memory location to another. A 24-bit byte counter allows large moves to occur with no intervention required by the processor.

If both addresses are in system memory, then the 53C710 functions as a high-speed DMA controller, able to move data at speeds of (up to) 40 MBytes/sec without using the processor or its cache memory. If the source address is within the 53C710's address space, then the instruction is a write to external memory. To perform a read from memory, make the destination address be within the 53C710.

MEMORY-TO-MEMORY MOVE, FIRST SCRIPTS WORD

SCSI I/O Processor Opcode, Bits 31-30

Reserved Section, Bits 29-24

These bits should always be zero.

24-bit Byte Count, Bits 23-00

This count value specifies the exact number of bytes to be moved from the source address and the destination address. As the SCSI SCRIPTS command is decoded, the value is moved into the DMA Byte Counter register. The SCSI I/O Processor will:

- Gain access to the system bus.
- Transfer the burst size into the DMA FIFO
- Decrement the byte count.
- Increment the source address.
- Gain access to the system bus.
- Transfer the burst size from the DMA FIFO into system memory.
- Increment the destination address.

The process will continue until the byte count is zero at the start of a byte transfer into the DMA FIFO. At that time, the next SCSI SCRIPTS command will be fetched.

The Indirect Mode is not allowed for the Memory Move command; therefore, the byte count must be in the actual SCRIPT. A byte count can be any value; thus, an odd number of bytes can be transferred.

MEMORY MOVE SECOND SCRIPTS WORD

Source Address of the Memory Move, Bits 31-00

This value specifies the address from which data will be moved. An address must be the full 32-bit physical address of the data source. The indirect mode is not allowed in the Memory Move instruction. The DMA Next Address register holds this source address and is incremented with each chip DMA transfer. If the value placed in the chip is a 53C710 register address, data can be moved from the 53C710 to a destination address. Only one byte, or multiples of four bytes, can be moved out of the chip. A register-to-register move is possible if both source and destination addresses are within the 53C710's register address space.

For another method of placing a 32-bit address in the instruction without resorting to patching SCSI SCRIPTS, please refer to the PASS option available in the SCSI SCRIPTS compiler.

MEMORY MOVE THIRD SCRIPTS WORD

Destination Address of the Memory Move, Bits 31-00

This value specifies the address to which data will be moved. An address must be the full 32-bit physical address of the data destination. The indirect mode is not allowed in the Memory Move instruction. The TEMP register holds this destination address and is incremented with each chip DMA transfer. If the value placed in the chip is a 53C710 register address, then data can be moved to the 53C710 from a source address. One byte, or multiples of four bytes, can be moved into the chip. A register-to-register move is possible if both source and destination addresses are within the 53C710's register address space. For another method of placing a 32-bit address in the instruction without resorting to patching SCRIPTS, please refer to the PASS option available in the SCSI SCRIPTS compiler.

There is one restriction on addresses that the 53C710 can process. The low order two bits must be equal; thus, the the source address must be on the same byte offset within a longword as the destination. An illegal instruction results if the two addresses are not byte aligned. The 53C710 supports burst sizes of 1, 2, 4, or 8 longwords.

During this instruction's execution, the DMA SCRIPTS Pointer Save register and the Data Structure Address register are used (along with the DNAD and TEMP) and will be destroyed. These registers should be saved before a Memory Move command and then later restored, if the contents are significant. To save the contents of a register, move its contents to the scratch register and then move the information into memory. Any register not used by the Memory Move command can be written directly to memory. Because the moving of data to the 53C710 is the last event performed by the command, any register can be written, including the ones used by the command.

Chapter 3
Developing NCR SCSI SCRIPTS

To develop an executable SCSI SCRIPT, first define the SCSI functions required. Identify what functions will be executed in SCRIPTS and what functions must be contained in system firmware. Then design the specific algorithms for the functions that will be executed in the SCSI SCRIPTS portion of the SCSI logical I/O driver.

Use the SCRIPTS compiler to code the algorithms SCRIPTS. Then compile to create the object code required as input by the 53C710. The compiler output is like an object module, it includes relocation information required to load the SCRIPTS object module into main memory, if any relocation is required.

At load time, the SCRIPTS absolute jump addresses must be resolved using one of the utilities furnished in the software package. At start I/O time, another utility is used to patch in the correct buffer addresses, byte counts, destination I.D., and so forth, if the Table Indirect mode is not used.

Writing a logical I/O driver is an easy task for the 53C710. This is illustrated in the first SCSI SCRIPTS example. This code will perform a read or write function using the 53C710 in the high-level chained mode. Because SCSI algorithms are so simple when written in SCSI SCRIPTS, you can rapidly prototype SCSI algorithms for a proof of concept and concentrate later on more complicated, realistic algorithms.

A SCSI SCRIPTS is comprised of two parts, or areas:

1) Definition area
2) SCRIPT area

In the following example, the definition area is comprised of variable and absolute values. These values may describe a variable memory address location, variable byte count or a fixed status byte value.

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
;* The following are variable data values provided  *
;* external to the compiler and resolved at run-time *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *

;       Definition area INITIATOR ROLE

;       Target Device I.D. offset in the data table.
EXTERNAL device

;       status_adr
EXTERNAL status_adr

;       Ten byte buffer address offset.
EXTERNAL sendmsg

;       Ten byte buffer address offset.
EXTERNAL rcvmsg

;       Buffer address offset for the SCSI command
EXTERNAL cmd_adr

;       Address of user data buffer
EXTERNAL data_adr


;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
;* Absolute values are stored in DSPS Register     *
;* for purposes of interrupt processing             *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *

;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
;* Note that 0X0 precedes the interrupt status      *
;* values and designates a hex value                *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
;       Error -- not message out after selection

ABSOLUTE err1 = 0x0ff01

;       Error -- unexpected SCSI phase before command phase
ABSOLUTE err2 = 0x0ff02

;       Error -- unexpected SCSI phase after a command transfer
ABSOLUTE err3 = 0x0ff03

;       Error -- expected status phase
ABSOLUTE err4 = 0x0ff04

;       No Error -- good I/O
ABSOLUTE ok = 0x0ff00

;       Error -- expected message outphase
ABSOLUTE err5 = 0x0ff05

;       Error -- expected message command complete
ABSOLUTE err6 = 0x0ff06

;*************************************************
; The following shows how you can use the PASS capability   *
; of the compiler to pass C code to the output file         *
;*************************************************
PASS(include "NCR.h")
PASS(extern char line[];)
```

Single-Tasking SCSI Example

The following is a simple SCSI SCRIPT that performs a single-tasking SCSI operation without disconnecting.

If an unpredictable event occurs on the SCSI bus, a unique interrupt vector value is stored in the 53C710's **DSPS** register and is available for interrupt processing.

```
PROC sample:
; select device with attention on
select atn from device, REL (resel_adr)

;  if the next phase is not message out, interrupt
int err1 when not MSG_OUT

; sent the i.d. message out to the target
move FROM sendmsg, when MSG_OUT

; if the next phase is not command, interrupt
int err2 when not CMD

; send the command bytes
move FROM cmd_adr, when CMD

; go to process cleanup if status phase
jump REL (end) when STATUS

; process data in phase
jump REL (input_data) if DATA_IN

; or data out phase
jump REL (output_data)  if DATA_OUT

; unexpected phase if here
int err3

; process the data in phase
input_data:
move FROM data_adr, when DATA_IN

; and go process status
jump REL (end)

; process the data out phase
output_data:
move FROM data_adr, when DATA_OUT

; interrupt if not status phase
end:
int err4 when not STATUS
```

```
; move the status byte into memory
move FROM status_adr, when STATUS

; interrupt if message in is not next
int err5 when not MSG_IN

; move the command complete byte in
move FROM rcvmsg, when MSG_IN

; interrupt if it is not a command complete message
int err6 if not 00

; accept the message if there are no problems
clear ack

; wait for a physical disconnect
wait disconnect

; interrupt with an I/O complete
int ok

resel_adr:
int ok
```

Chapter 4
SCSI SCRIPTS Compiler

SCSI SCRIPTS Compiler

The SCSI SCRIPTS Compiler takes a source file and generates a C file which may then be used in other C programs. The source file may be created using any standard text editor that creates ASCII file output.

To provide portability this compiler does not support directory paths. The compiler and the files to be compiled must reside in the same directory.

Invoking the SCSI SCRIPTS Compiler

In the following examples, items enclosed in double brackets "{ }" are optional. The following format is used to invoke the compiler.

scc sourcefile {options}

Options:

-o {OutputFilename}

This option determines if a C output file will be generated and if so what the name of the file will be. If the -o is given without a filename following, then the filename will default to sourcefile.out.

-l {ListFilename}

This option determines if a listfile will be generated and if so what the name of the filename will be. If the -l option is given without a filename following, then the filename will default to sourcefile.lis. For every instruction the listfile lists

an offset from the beginning of the script,
the long word instruction,
the long word address, and
the corresponding ASCII source instruction

Labels appear on a line by themselves as they are encountered in the SCRIPT.

Next is a list consisting of absolute or relative variables, and their location in the SCRIPT. This is followed by a list of labels and label locations that appear in the SCRIPT. The location is an offset from the beginning of the SCRIPT.

The final list gives the label patches. Label patches are offsets into the SCRIPT where a label is referenced. They are called patches because the absolute address of the labels must be patched into the SCRIPT at runtime.

-z {debugfilename}

This option will generate a file that is necessary if the SCRIPT debugger is to be used. If the debugger is used, this is the file that is loaded to begin the debug process. If the -z option is given without a filename following, then the filename will default to sourcefile.sod. The file produced when this option is set is compatible with the pass 1 output file of the C700 compiler.

-e {errorfilename}

This option will generate an error file where all the error information will be stored. If the -e option is used without a filename following, then the filename will default to sourcefile.err.

-v

This option will print all relevant information about the compilation process to the screen for the user to view.

-u

When this option is set the define INSTRUCTIONS and define PATCHES statements in the output file is suppressed. This option is necessary if two or more output files are being linked together.

SCSI SCRIPTS Compiler Output

When the compiler is writing to an output file, it will generate instruction array(s) first unless the pass option is used before any instructions are given. If the first instruction is not preceeded by a proc label: statement, then the instruction array name will default to "SCRIPT" . The first column in the instruction array contains the long word instruction and the remaining columns contain corresponding long word addresses. An example is given below:

Source Code:

```
PASS(#include "NCR.h")
int 7
PROC first:
int 8
```

Compiled Output:

```
#include "NCR.h"
ULONG SCRIPT[] = {
        0x98080000,  0x00000007
};

ULONG first[] = {
        0x98080000,  0x00000008
};
```

The variable name prefix will have an "A_" for absolute or an "R_" for relative. The value of the variable is used in a define statement. The define statement is followed by an array which contains the long word offsets into the SCRIPT where the variable is used. The array name is the variable name appended with "_Used".

Example:

```
#define R_DATA_BUF 0x00000020
ULONG R_DATA_BUF_Used[] ={
};
```

Then the SCRIPT entry label values are defined with a prefix of "Ent_".

Example:

```
#define Ent_alt_addr 0x00000078
```

The SCRIPT entry labels values are followed by an array of long word offsets for labels in the SCRIPT. These offsets are used to patch in the absolute addresses at runtime.

Example:

```
ULONG LABELPATCHES[] = {
        0x00000001,  0x00000019,
        0x0000001b
};
```

The last item produced is the number of instructions and patches in the SCRIPT. Note that if the the undefine option is set "-u" when invoking the compiler, these statements will not be produced.

Example:

```
ULONG INSTRUCTIONS =
0x00000011;
ULONG PATCHES = 0x00000003;
```

Appendix D shows the source file, the list file, the debug file and the output file from the initiator script of the previous chapter. This script was named sample and these files resulted from the following invocation:

```
scc sample -l -z -o
```

Chapter 5
NCR SCSI SCRIPTS Utilities

The following utilities are part of the Software Development package which includes the SCSI SCRIPTS Compiler.

Initialize_IOP()

Sets up the 53C710 for operation after power up.

Save_IOP_State(savearea*)

Takes the pointer as an argument to a save area and stores the status of the 53C710 at that address. Information saved includes SCSI SCRIPT pointer, current data counter, buffer address, and all registers required to allow the state of the chip to be restored later. This routine is used during SCSI disconnect handling, or save data pointer operations.

Restore_IOP_State(savearea*)

Takes the pointer as an argument to the save area where Save_IOP_State has stored the 53C710 status data. Restores the chip state so an interrupted I/O can be resumed after a reselection or restore pointers operation.

IOP_Interrupt_Status()

After the 53C710 experiences an interrupt, this routine is called to decode the chip's interrupt status information and report the reason for the interrupt.

Relocate_Script_Address(rel_info,base)

Relocates all transfer control addresses in a SCSI SCRIPT to the specified base address. Rel_info is a data structure produced by the back end of the SCSI SCRIPTS compiler.

Patch_Script(rel_info,symbol,value)

Using the symbol name for byte count, device i.d., or data address, and the rel_info data structure, this routine will patch the locations where the symbol is used with the input value.

InitSIOP()

Declaration:
  void InitSIOP(struct SIOP*)

InitSIOP() accepts a pointer to an SIOP struct or NULL. If NULL then all the members in __SCSIREGS__ are assigned a value of 0. Otherwise copy the value from each member of the passed struct into __SCSIREGS__ and put those into the chip.

NOTE:
This function will, by default, assign one structure to another. This is ANSI C compatible, but older compilers may not support it.

SetPhaseMMInt()

Declaration:
  void SetPhaseMMInt(BOOL)

SetPhaseMMInt() turns the phase mismatch interrupt on or off.

SetCompInt(BOOL)

Declaration:
  void SetCompInt(BOOL)

SetCompInt() turns the function complete interrupt on or off.

SetSelTimeoutInt()

Declaration:
  void SetSelTimeoutInt(BOOL)

SetSelTimeoutInt() turns the select time out interrupt on or off.

SetSelInt()

Declaration:
  void SetSelInt(BOOL)

SetSelInt() turns the select interrupt on or off.

SetGrossErrInt()

Declaration:
  void SetGrossErrInt(BOOL)

SetGrossErrInt() turns the SCSI gross error interrupt on or off.

SetUXDiscInt()

Declaration:
  void SetUXDiscInt(BOOL)

SetUXDiscInt() turns the Unexpected disconnect interrupt on or off.

SetRSTInt()

Declaration:
  void SetRSTInt(BOOL)

SetRSTInt() turns the RST/ interrupt on or off.

SetParInt()

Declaration:
  void SetParInt(BOOL)

SetParInt() sets the parity error interrupt on or off.

Set286Mode()

Declaration:
  void Set286Mode(BOOL)

Set286Mode() puts the chip in 80286 mode when ON, otherwise it is in the 80386 mode.

ClearDMAFifo()

Declaration:
  void ClearDMAFifo()

ClearDMAFifo() clears the DMA FIFO.

SetIO()

Declaration:
  void SetIO(BOOL)

SetIO() tells the 53C710 to transfer data to an I/O mapped device when ON, otherwise transfers are to memory mapped devices.

Set16BitDBus()

Declaration:
  void Set16BitDBus(BOOL)

Set16BitDBus() causes the 53C710 to perform transfers 16-bits at a time when ON, otherwise transfers are 32-bits at a time.

SetFixedAddr()

Declaration:
  void SetFixedAddr(BOOL)

SetFixedAddr() disables the address pointer in the DNAD register so that it is ON, it will not increment.

SetAbortInt()

Declaration:
  void SetAbortInt(BOOL)

SetAbortInt() makes the 53C710 drive the INT/ signal when an abort condition is encountered and it is set to ON.

SetINTInstInt()

Declaration:
  void SetINTInstInt(BOOL)

SetINTInstInt() allows the 53C710 to drive the INT/ signal when it encounters an INT instruction in a script and it is set to ON.

SetWatchDogInt()

Declaration:
  void SetWatchDogInt(BOOL)

SetWatchDogInt() allows the 53C710 to drive the INT/ signal when the watch dog timer decrements to 0 and it is set to ON.

SetIllegalInstInt()

Declaration:
  void SetIllegalInstInt(BOOL)

SetIllegalInstInt() allows the 53C710 to drive the INT/ signal when an illegal instruction is encountered in a SCRIPT and it is set to ON.

Set16BitScripts()

Declaration:
  void Set16BitScripts(BOOL)

Set16BitScripts() makes the 53C710 fetch script instructions 16-bits at a time when set to ON. Otherwise fetches are 32-bits at a time.

SelClkFreq()

Declaration:
```
void SelClkFreq(UBYTE)
```

SetClkFreq() send the clock speed being used by the system to the 53C710. It accepts 1 of 3 values; SLOW, MED, or FAST.

```
/* 0 FAST 37.51 to 50 MHz
/* 1 MED 25.01 to 37.5 MHz
/* 2 SLOW 16.67 to 25 MHz
```

SetHOSTID()

Declaration:
```
BOOL SetHOSTID(UBYTE)
```

SetHOSTID() accepts a byte value to be placed into the SCID register. It will not allow a value of 255 (FF hex) to be placed into this register since the 53C710 cannot talk to itself.

SetParity()

Declaration:
```
void SetParity(BOOL)
```

SetParity(), when ON, the 53C710 checks the data bus for odd parity when receiving across the SCSI bus.

SetAutoATN()

Declaration:
```
void SetAutoATN(BOOL)
```

SetAutoATN(), the 53C710 asserts the ATN/ signal when a parity error is detected and it is ON.

SetSlowBus()

Declaration:
```
void SetSlowBus(BOOL)
```

SetSlowBus(), the 53C710 adds 1 extra clock cycle to the data setup time when it is ON.

GetPhysAddr()

Declaration:
```
ULONG GetPhysAddr(UBYTE far *)
```

GetPhysAddr() accepts a far pointer in the 80x86 format. Then It takes the segment portion, multiplies it by 16 and adds it to the offset portion to return a physical address.

PatchLabels()

Declaration:
```
void PatchLabels(Base, PatchArray, Count)
        ULONG Base[], PatchArray[];
        ULONG Count;
```

PatchLabels() patches a script that references labels within that script. Three ULONGs are passed to it.

The first ULONG is a pointer to the ULONG array SCRIPT that is going to be manipulated.

The second is a pointer to the ULONG PatchArray (LABELPATCHES), the array whose elements contain the offsets into the script to be manipulated.

The third ULONG is the count of the number of elements in the patch array.

PatchRelative()

Declaration:
```
void PatchRelative (ScriptBase, RelBase,
RelArray, Count)
        ULONG ScriptBase[ ], RelArray[ ];
        ULONG RelBase, Count;
```

PatchRelative() requires a little programmer input. It is very similar to PatchLabels(). Passed to it are

a pointer to the ULONG Script array,

the physical relative data base address,

a pointer to the ULONG relative Data array, and

a count of the number of elements in the relative array.

PatchID()

Declaration:
```
void PatchID(Instructions, Location, Value)
        ULONG far *Instructions;
        ULONG Location, Value;
```

PatchPhase()

Declaration:
```
void PatchPhase (Instructions, Location, Value)
        ULONG far *Instructions;
        ULONG Location, Value;
```

Chapter 6
The NCR SCSI SCRIPTS Language Syntax

Notation

[ ] Items enclosed in brackets are optional.

[ ]"..." The item enclosed in the brackets can be repeated as often as necessary.

KEYWORD A word in all upper case is a keyword. Case is ignored by the compiler when looking for keywords.

Phase must be replaced with only one of the following keywords:

MSG_IN,
MSG_OUT,
DATA_IN,
DATA_OUT,
CMD,
STATUS,
RES4,
RES5

The word **'address'** means a 32-bit number.

The word **'value'** means a 32-bit number.

The word **'count'** means a 24 bit number.

The word **'id'** means an eight bit number that has exactly one bit set.

The word **'data'** means an eight bit number.

The word **'expression'** denotes a mathematical expression with the form:

<identifier> [<addop> <identifier>]*

<identifier> is any valid variable name or a numeric constant.

<addop> is the '+' or '-' character to denote addition or subtraction respectively.

An 'expression' may be used in any place that a numeric value would normally be used. The value of all 'expressions' are automatically extended to 32-bits. When expressions are used in a context where the evaluated value is less then 32-bits, the least significant bits will be used. For instance, if an 'expression' is used to represent a count for a move instruction, the evaluated value will be truncated to 24 bits. Notification that the expression has been truncated will occur if the value of the expression is changed.

The word **'name'** represents a string of one or more consecutive characters chosen from the letters, the numbers, the underscore, and the dollar sign. Names used for labels, externals, and variables in the relative data area are passed on to the Host development system.

If the Host development system has restrictions on the format of such names, it is the responsibility of the SCSI SCRIPTS writer's to avoid using such names. For example, Turbo C, which is used as the Host development system for this application, does not allow names to begin with a digit or to contain a dollar sign. Therefore, the SCSI SCRIPTS writer for DOS and Turbo C should avoid using names of this form.

Input Format

SCSI SCRIPTS consist of a series of lines. Blank lines, lines containing only white space, and anything after a semi-colon on an input line are ignored by the front end.

The compiler is "token" oriented. It reads the input stream and splits it up into tokens. White space and anything from a semicolon to the end of the line is not part of any token, and is ignored by the compiler.

There are two types of tokens. A ***token*** is any string of consecutive letters, numbers, dollar signs, or underscores; a character can be part of ONLY one token. The input stream is split into tokens to minimize the number of tokens. For example, the string "abc" would be treated as one token ("*abc*") rather than multiple tokens ("*a*" and "*bc*").

The second type of token consists of characters that are not part of other tokens. Anything that is not a letter, a digit, an underscore, or a dollar sign becomes a token. For example, the string

```
xxx=0x123 ; assign value to xxx
```

contains three tokens.

```
xxx
=
0x123
```

Numeric values may be specified in decimal, hexadecimal, octal, or binary.

Decimal numbers are specified by a string of digits not beginning with zero.

Hex numbers are specified by a string consisting of "0x" or "0X" and the hex digits of the number. Both upper and lower case are allowed.

A binary number is similar to a hex number, except that "0b" or "0B" is used instead of "0x" or "0X".

An octal number is specified by a "0" followed by the octal digits.

Language Directives

Several keywords provide information to the front end about the compilation of the SCSI SCRIPTS. They define symbolic names and indicate things to be passed to the output of the compiler.

ENTRY label [,label...]

The ENTRY keyword indicates that the specified labels are SCSI SCRIPTS entry points. Their names and values are defined at the back end, which will also make them available to the Host development system.

ABSOLUTE name = expression [,name = expression...]

This declares symbolic names for numeric values. For example,

```
ABSOLUTE bad_cmd = 0x1200
```

allows the name

```
bad_cmd
```

to be used instead of a number in the SCSI SCRIPTS. The SCSI SCRIPTS will be compiled as if the number 0x1200 had been specified instead of the name "bad_cmd" in every instruction that uses "bad_cmd".

EXTERNAL name [,name...]

This tells the compiler that the SCSI SCRIPTS will refer to variables with specified names that are declared outside of the SCSI SCRIPTS. Some host development systems are not able to support use of this word and SCSI SCRIPTS requiring this feature may not be portable to all hosts.

RELATIVE name = expression [,name = expression...]

Use to declare relative data variables.

name	the variable name.
expression	the offset from the start of the relative data area where the variable is located.

A name followed by a colon signifies a label. Use a label name wherever there is a call for an address.

The SCSI SCRIPTS Instructions

When an instruction call specifies a count, use a 24-bit number or a symbolic constant (declared using the ABSOLUTE keyword).

When an instruction requires an address, use

a 32-bit number,

the label name,

the variable name in the relative data area (previously declared with the RELATIVE keyword) , or

the external variable name (previously declared with the EXTERNAL keyword).

Labels, external variables, and relative variables all share the same name space. If a name is declared more than once, the front end resolves the conflict. If a problem possibly exists, a warning will be issued.

If the address field of an instruction contains an undefined name, then the front end assumes that it refers to a label that will be defined later. This is called forward referencing. If the name is defined later as an external or relative variable, this will create a name conflict and the front end will resolve it. A possible problem warning is issued.

Anywhere a 32-bit address can be used in a SCRIPT, the PASS option can be substituted. This option allows the user to pass through a literal value to the output and thus to be input into the C compiler. Any valid C Name (for example, label, structure element) can be passed through for final resolution by the C compiler.

Even though the SCRIPTS compiler cannot recognize the name, or resolve the value, it can preserve it as a literal for the C output.

BLOCK MOVE Command

There are several forms of the Block Move instruction.

address count — Specify the address and byte count fields of the instruction. If the optional keyword PTR is present, then the indirect bit will be set.

Phase — Specifies the phase field of the instruction

WITH or WHEN — Specify the Block Move function codes

WITH signals the target role which sets the phase values

WHEN is the initiator "test for phase" feature

FROM — Specifies that the byte count and address are to be fetched from memory. The 24-bit value is combined with the Data Structure Address register to form the 32-bit fetch address.

The 53C710 waits for a valid phase (initiator) or drives the phase lines (target). In the initiator role, it performs a compare by looking for a match between the phase specified in the SCRIPT and the actual value on the bus. If the phases do not match, an external interrupt occurs. Data is then transferred in or out according to the phase lines. When the count goes to zero, the next sequential SCRIPTS instruction is fetched.

```
MOVE count, [ PTR ] address, WITH Phase
MOVE count, [ PTR ] address, WHEN Phase

MOVE FROM address, WITH Phase
MOVE FROM address, WHEN Phase
```

Address can be replaced with PASS(C Name) for the MOVE command.

JUMP Command

The conditional JUMP instructions all have the same general form.

Address — The SCSI SCRIPTS address that will be transferred to if the JUMP is taken. Limited to 24 bits if the

REL — option is used.

WHEN — Sets the Wait bit in the SEQ CNTL field.

IF — Do not set the Wait bit.

If NOT follows WHEN or IF, then the True/False bit of the SEQ CNTL field is not set; otherwise, the bit will be set.

Phase — When present, the compare Phase bit of SEQ CNTL will be set; otherwise, it will be cleared.

data — When present, the compare Data bit of SEQ CNTL will be set; otherwise, it will be cleared.

If both 'Phase' and 'data' are specified, they must be in that order and they must be separated by the keyword AND.

ATN — The target role version which is required to test whether the initiator has set ATN on the bus.

NOT — Used for the inverse test of WHEN and IF. "NOT Phase OR data" is the negation of "Phase AND data".

MASK — Always use with an 'AND' or 'OR' keyword. The data following the keyword 'MASK' allows a SCRIPT to selectively compare the bits within the data byte.

REL — Used if the jump is to be relative to the current program counter.

Note that the address values can be replaced with a REL (Address). The value of the address must be a signed 24 value. Also PASS (Any valid C Name) can replace an address.

Any bits that are ON eliminate the corresponding bit in the data byte at the time of the compare. Use this 'binary sort' to quickly determine the value of incoming bytes. For example, a mask of '7F' and a data compare of '80' allows the SCRIPTS processor to determine if the high order bit is ON.

```
NOP
JUMP  address
JUMP  address,  IF ATN
JUMP  address,  IF Phase
JUMP  address,  IF data
JUMP  address,  IF data, AND MASK data
JUMP  address,  IF ATN AND data
JUMP  address,  IF ATN AND data, AND MASK data
JUMP  address,  IF Phase AND data
JUMP  address,  IF Phase AND data, AND MASK data
JUMP  address,  WHEN Phase
JUMP  address,  WHEN data
JUMP  address,  WHEN data ,AND MASK data
JUMP  address,  WHEN Phase AND data
JUMP  address,  WHEN Phase AND data, AND MASK data
JUMP  address,  IF NOT ATN
JUMP  address,  IF NOT Phase
JUMP  address,  IF NOT data
JUMP  address,  IF NOT data, AND MASK data
JUMP  address,  IF NOT ATN OR data
JUMP  address,  IF NOT ATN OR data, AND MASK data
JUMP  address,  IF NOT Phase OR data
```

```
JUMP  address,  IF NOT Phase OR data, AND MASK data
JUMP  address,  WHEN NOT Phase
JUMP  address,  WHEN NOT data
JUMP  address,  WHEN NOT data, AND MASK data
JUMP  address,  WHEN NOT Phase OR data
JUMP  address,  WHEN NOT Phase OR data, AND MASK data
JUMP  REL (Address)      (An option for any "address" above)
```

PASS (Any valid C Name) can replace an address in the JUMP instruction.
REL (Address) can replace an address in the JUMP instruction.

CALL Command

All conditional CALL instructions have the same general form.

Address — The SCSI SCRIPTS address transferred to if the JUMP is taken.

WHEN — Set the Wait bit in the SEQ CNTL field.

IF — Do not set the Wait bit.

If WHEN or IF are followed by NOT, then the True/False bit of the SEQ CNTL field is not set. Otherwise, the bit will be set.

Phase — When present, the compare Phase bit of SEQ CNTL will be set; otherwise, it will be cleared.

data — When present, the compare Data bit of SEQ CNTL will be set; otherwise, it will be cleared.

If both Phase and data are specified, they must be in that order and they must be separated by the keyword AND; that is ..WHEN Phase AND data...

ATN — The target role version which is required to test whether the initiator has set ATN on the bus.

NOT — Used for the inverse test of WHEN and IF.

"NOT Phase OR data" is the negation of "Phase AND data".

MASK — Always use with an 'AND' or 'OR' keyword. The data following the keyword MASK allows a SCRIPT to selectively compare the bits within the data byte.

REL — Used if the jump is to be relative to the current program counter.

Note that the address values can be replaced with a REL(Address). The value of the address must be a signed 24 value. Also PASS (Any valid C Name) can replace an address.

Any bits that are ON eliminate the corresponding bit in the data byte at the compare. Use this 'binary sort' to quickly determine value of incoming bytes. For example, a mask of '7F' and a data compare of '80' allows the SCRIPTS processor to determine if the high order bit is ON.

```
CALL address
CALL address,  IF ATN
CALL address,  IF Phase
CALL address,  IF data
CALL address,  IF data, AND MASK data
CALL address,  IF ATN AND data
CALL address,  IF ATN AND data, AND MASK data
CALL address,  IF Phase AND data
CALL address,  IF Phase AND data, AND MASK data
CALL address,  WHEN Phase
CALL address,  WHEN data
CALL address,  WHEN data, AND MASK data
CALL address,  WHEN Phase AND data
CALL address,  WHEN Phase AND data, AND MASK data
CALL address,  IF NOT ATN
CALL address,  IF NOT Phase
CALL address,  IF NOT data
CALL address,  IF NOT data, AND MASK data
CALL address,  IF NOT ATN OR data
CALL address,  IF NOT ATN OR data, AND MASK data
CALL address,  IF NOT Phase OR data
CALL address,  IF NOT Phase OR data, AND MASK data
CALL address,  WHEN NOT Phase
CALL address,  WHEN NOT data
CALL address,  WHEN NOT data, AND MASK data
```

```
CALL address,   WHEN NOT Phase OR data
CALL address,   WHEN NOT Phase OR data, AND MASK data
CALL REL (address)      (An option for any "address" above)
```

Pass (Any valid C Name) can replace an address in the CALL instruction.
REL (address) can replace an address in the CALL instruction.

RETURN Command

All conditional RETURN instructions have the same general form.

Address — The SCSI SCRIPTS address that will be transferred to if the JUMP is taken.

WHEN — Set the Wait bit in the SEQ CNTL field.

IF — Do not set the Wait bit.

If WHEN or IF are followed by NOT, then the True/False bit of the SEQ CNTL field is not set. Otherwise, the bit will be set.

Phase — When present the compare Phase bit of SEQ CNTL will be set; otherwise, it will be cleared.

data — When present, the compare Data bit of SEQ CNTL will be set; otherwise, it will be cleared.

If both Phase and data are specified, they must be in that order and they must be separated by the keyword AND.

ATN — The target role version which is required to test whether the initiator has set ATN on the bus.

NOT — Used for the inverse test of WHEN and IF. "NOT Phase OR data" is the negation of "Phase AND data".

MASK — Always use with an 'AND' or 'OR' keyword. The data following the keyword 'MASK' allows a SCRIPT to selectively compare the bits within the data byte.

Note that the address values can be replaced with a REL(Address). The value of the address must be a signed 24 value. Also PASS (Any valid C Name) can replace an address.

Any bits that are ON eliminate the corresponding bit in the data byte at the time of the compare. Use this 'binary sort' to quickly determine value of incoming bytes. For example, a mask of '7F' and a data compare of '80' allows the SCRIPTS processor to determine if the high order bit is ON.

```
RETURN
RETURN, IF ATN
RETURN, IF Phase
RETURN, IF data
RETURN, IF data, AND MASK data
RETURN, IF ATN AND data
RETURN, IF ATN AND data, AND MASK data
RETURN, IF Phase AND data
RETURN, IF Phase AND data, AND MASK data
RETURN, WHEN Phase
RETURN, WHEN data
RETURN, WHEN data, AND MASK data
RETURN, WHEN Phase AND data
RETURN, WHEN Phase AND data, AND MASK data
RETURN, IF NOT ATN
RETURN, IF NOT Phase
RETURN, IF NOT data
RETURN, IF NOT data, AND MASK data
RETURN, IF NOT ATN OR data
RETURN, IF NOT ATN OR data, AND MASK data
RETURN, IF NOT Phase OR data
RETURN, IF NOT Phase OR data, AND MASK data
RETURN, WHEN NOT Phase
RETURN, WHEN NOT data
RETURN, WHEN NOT data, AND MASK data
```

RETURN, WHEN NOT Phase OR data
RETURN, WHEN NOT Phase OR data, AND MASK data

INTERRUPT Command

All conditional INT instructions have the same general form.

value — The data value that will be placed in the DSPS register if the INT condition is evaluated as true.

WHEN — Set the Wait bit in the SEQ CNTL field.

IF — Do not set the Wait bit.

If WHEN or IF is followed by NOT, then the True/False bit of the SEQ CNTL field is not set. Otherwise, the bit will be set.

Phase — When present, the compare Phase bit of SEQ CNTL will be set; otherwise, it will be cleared.

data — When present, the compare Data bit of SEQ CNTL will be set; otherwise, it will be cleared.

If both Phase and data are specified, they must be in that order and they must be separated by the keyword AND.

ATN — The target role version which is required to test whether the initiator has set ATN on the bus.

NOT — Used for the inverse test of WHEN and IF. "NOT Phase OR data" is the negation of "Phase AND data".

MASK — Always use with an AND or OR keyword. The data following the keyword MASK allows a SCRIPT to selectively compare the bits within the data byte.

Any bits that are ON eliminate the corresponding bit in the data byte at the compare. Use this 'binary sort' to quickly determine value of incoming bytes. For example, a mask of '7F' and a data compare of '80' allows the SCRIPTS processor to determine if the high order bit is ON.

```
INT value
INT value, IF ATN
INT value, IF Phase
INT value, IF data
INT value, IF data, AND MASK data
INT value, IF ATN AND data
INT value, IF ATN AND data, AND MASK data
INT value, IF Phase AND data
INT value, IF Phase AND data, AND MASK data
INT value, WHEN Phase
INT value, WHEN data
INT value, WHEN data, AND MASK data
INT value, WHEN Phase AND data
INT value, WHEN Phase AND data, AND MASK data
INT value, IF NOT ATN
INT value, IF NOT Phase
INT value, IF NOT data
INT value, IF NOT data, AND MASK data
INT value, IF NOT ATN OR data
INT value, IF NOT ATN OR data, AND MASK data
INT value, IF NOT Phase OR data
INT value, IF NOT Phase OR data, AND MASK data
INT value, WHEN NOT Phase
INT value, WHEN NOT data
INT value, WHEN NOT data, AND MASK data
INT value, WHEN NOT Phase OR data
INT value, WHEN NOT Phase OR data, AND MASK data
```

PASS (Any valid C Name) can replace a value in the INT instruction.

SCSI I/O Commands

SELECT [ATN] ID, REL (Address)
SELECT [ATN] FROM Address, REL (Address)
SELECT [ATN] ID, Address
SELECT [ATN] FROM Address, Address

Initiator mode function 0.
If ATN is present, the "select with ATN" bit is turned on. 'id' specifies the destination SCSI ID. REL allows a relative jump and FROM allows a table indirect fetch of device id, offset and period for synchronous transfers.

RESELECT ID, address
RESELECT ID, REL (Address)
RESELECT FROM Address REL (Address)
RESELECT FROM Address, Address

Target mode function 0

WAIT DISCONNECT

Initiator mode function 1

DISCONNECT

Target mode function 1

WAIT RESELECT Address
WAIT RESELECT REL (Address)

Initiator mode function 2

REL allows the alternate address to be relative.

WAIT SELECT address
WAIT SELECT REL (Address)

Target mode function 2

If the 53C710 is connected as a target, the following set and clear commands will have no meaning. (The SCSI target role is active) and should not be used.

Note:

Any address value following FROM or REL must be a 24-bit signed value.

SET TARGET

Function 3 with the target bit set in the flags field.

SET ACK

Function 3 with the ACK bit set in the Flags field.

SET ATN

Function 3 with the ATN bit set in the Flags field.

SET ACK and ATN and TARGET

Function 3 with ACK, ATN, and TARGET bits set in the flag field. All three or any two of the keywords (ACK, ATN, or TARGET) may be used.

CLEAR TARGET

Function 4 with the target bit set in the flags field.

CLEAR ACK

Function 4 with the ACK bit set in the Flags field.

CLEAR ATN

Function 4 with the ATN bit set in the Flags field.

CLEAR ACK and ATN and TARGET

Function 4 with ACK, ATN, and TARGET bits set in the Flags field. All three or any two of the keywords (ACK, ATN, or TARGET) may be used.

REGISTER WRITE COMMAND

This instruction allows a read, modify, write, or a move to SCSI First-byte Received Register (SFBR) or a move from SFBR.

register: One of the registers must be SFBR if the instruction allows two register names (register to register move). Both registers must be the same for a read modify write.

Valid register names are:

SCNTL0	SCNTL1	SDID	SIEN
SCID	SXFER	SODL	SOCL
SFBR	SIDL	SBDL	SBCL
DSTAT	SSTAT0	SSTAT1	SSTAT2
DSA0	DSA1	DSA2	DSA3
CTEST0	CTEST1	CTEST2	CTEST3
CTEST4	CTEST5	CTEST6	CTEST7
TEMP0	TEMP1	TEMP2	TEMP3
DFIFO	ISTAT	CTEST8	LCRC
DBC0	DBC1	DBC2	DCMD
DNAD0	DNAD1	DNAD2	DNAD3
DSP0	DSP1	DSP2	DSP3
DSPS0	DSPS1	DSPS2	DSPS3
SCRATCH0	SCRATCH1	SCRATCH2	SCRATCH3
DMODE	DIEN	DWT	DCNTL
ADDER0	ADDER1	ADDER2	ADDER3

REG(n), where n is a value from 0 to hexadecimal 03f.

Only 8 bits of a register can be operated on at one time.

data 8: An 8-bit data value or name of an 8-bit value.

operator: Valid operators are OR, AND, addition and subtraction.

Register writes are very useful, but caution must be exercised when this mode is used. Writing to certain registers could have disastrous effects on the SCSI bus or operation of the chip. When a register is written or read, side effects may occur; the degree and possibility of these effects must be clearly understood.

A register-to-register move can be accomplished by moving data from the source register to the SFBR and then from the SFBR to the destination register.

To compare for a value in a register (or a bit ON), move the value to the SFBR (AND off unwanted bits); then execute a COMPARE and JUMP instruction.

In the following instructions, the two register keywords in each line must be identical, or one must be SFBR.

The Add or Subtract operator can be used for an event or loop counter.

```
MOVE register TO register
MOVE data8 TO register
MOVE register | data8 TO register
MOVE register & data8 TO register
MOVE register + data8 TO register
MOVE register - data8 TO register
```

MEMORY-TO-MEMORY MOVE Command

This instruction allows the 53C710 to become a high-speed DMA chip. DATA is moved from the source address into the chip's DMA FIFO and then out to the destination address. There is no indirect capability with this command so that the physical 32-bit address must be in the SCRIPT. The PASS option can be used with either or both addresses to allow the user to designate a C Name that can be resolved when the C code is compiled.

count A 24-bit byte count for the number of bytes to the transferred by the MOVE command.

address A 32-bit physical address; source is first, followed by the destination data buffer address.

The last two bits of the source and destination must be equal, but there are no other restrictions on the address values. Note that if a 53C710 register address is the source or destination, then this instruction can be used to read or write system memory from SCRIPTS. This capability is very useful for saving the state of an I/O in a multi-threaded I/O environment.

MOVE MEMORY count, source address, destination address

PASS Option

To allow the SCSI SCRIPTS compiler more flexibility in the C environment, an option is included that allows the programmer to pass 32-character literal strings through to the C compiler. This feature allows the programmer to use any C Name that will be resolved when the output is compiled. Strings can be placed on a single line, or used in place of a 32-bit address.

PASS (literal string)

This statement can be used to send an include statement to the C compiler. Note that this allows the two levels of include capability. The first level is implemented by using include statements in the SCRIPTS code and using a C preprocessor to bring in the desired code. The second level uses the PASS option. Everything between the left and right parenthesis is sent to the output file of the C compatible SCSI SCRIPTS compiler. The literal string must be placed before the SCSI SCRIPTS instruction area.

```
Wait  Reselect  PASS(&alt_addr)
Move  Memory 4,   PASS(&buf.save),
                  PASS(&buf.restore)
```

These two SCRIPTS instructions illustrate how the PASS option can be used to defer the fixing of addresses until link time. Any C Name can be referred to (limited to 32 characters) if it will be converted to a 32-bit address by the C compiler.

Addressing External Labels

A SCRIPTS programmer may want to write modular code instead of one large routine. To have modules, some type of external reference must be allowed. Because the SCRIPTS compiler does not have a link editor capability, another mechanism allows the same feature with minimal changes to the compiler. Encountering the keyword PROC causes the compiler to close out the current longword array and generate a new array with the label following the keyword PROC. Thus, the name "label" can be referenced by other SCSI SCRIPTS in other modules. For example, a JUMP instruction can transfer to an external name, using the PASS option. At C compile time, the reference will be resolved.

Chapter 7
SCRIPTS SCSI Use of Scatter-Gather

Virtual memory schemes are common in today's systems; they are used to keep user data in small, manageable pages in main memory. Memory management units track actual, physical locations. This memory scheme is called scatter-gather because user data is scattered through memory and gathered for a write to disk. One I/O may include several pages; therefore, current SCSI ports must re-instruct the DMA controller at the beginning of each user-data page.

The extra time required to re-instruct for each page causes some delay for the external processor interrupt and DMA set-up time. A potentially undesirable side effect occurs when the delay makes the disk slip a revolution because there is no place to put data coming off the media, or the data is not yet available for writing to the media.

The 53C710 has an efficient solution to the scatter-gather performance degradation problem. Each page of user data is represented by a Block Move command. The only overhead required to move to the next page of data is a SCSI SCRIPTS fetch. No firmware interrupt is required (normally a ***minimum*** of 80 microseconds in a system environment). Nor is firmware required to re-instruct a DMA controller.

Chapter 7 contains one possible SCSI SCRIPTS model for the scatter-gather situation. First, separate the set of Block Move commands that are required to process the user data and code the SCSI SCRIPTS to call this user data section to move data. Determine a maximum number of pages per I/O and code one SCSI SCRIPTS Block Move for each possible page. At the start I/O time, the logical I/O routine determines exactly how many block moves are required and writes a return command over the next SCSI SCRIPTS command after the last required Block Move command. The group of Block Move commands is called, the correct number of moves is performed, and the return is executed. At the completion of the I/O, the return is overwritten with a Block Move to prepare the set of Block Move commands for the next I/O.

With the read/write capability of the 53C710, another possible solution exists for the scatter gather problem. The following SCSI SCRIPT uses the increment register feature and the table indirect feature to update the address and count values in the chip's registers. By fetching these values indirectly and adding 8 to the DSA register each time through the loop, the SCRIPT can continue to fetch user data from various locations. The actual SCRIPT is:

```
Loop:
; Perform the move
        Move from ADDR when DATA_OUT
; Increment to the next DSA entry
        Move DSA (0) +8 to SFBR
; Check for Wraparound
        Jump L1 if not 00
; handle a one-byte overflow
        Move DSA(1)+1 to DSA(1)

L1:

; update the DSA register
        Move SFBR to DSA(0)
; repeat until a phase change
        Jump rel(Loop) when DATA_OUT
```

This SCRIPTS algorithm allows for a large number (8196) of Data Structure Table entries in the scatter- gather list. An alternative to simply waiting for a phase change is to use a counter in the loop and exit on zero. To allow for disconnects in the loop, save the Data Structure Address (DSA) register value when processing the disconnect message.

The 53C710 can process scatter/gather requests in a very simple manner and simultaneously, dramatically reduce I/O overhead.

```
;       Definition area INITIATOR ROLE

;       Target Device I.D. offset in the Data structure
Absolute device=0

;       Send message offset for count and address
Absolute sendmsg=8

;       Receive message offset for count and address
Absolute rcvmsg=0X010

;       SCSI command offset for count and address
Absolute cmd_adr=0X018

;       User data buffer offset for count and address
Absolute data_adr=0X020

;       Error -- not message out after selection
ABSOLUTE err1 = 0x0ff01

;       Error -- unexpected SCSI phase before command phase
ABSOLUTE err2 = 0x0ff02

;       Error -- unexpected SCSI phase after a command transfer
ABSOLUTE err3 = 0x0ff03

;       Error -- not msg in phase after status phase
ABSOLUTE err4 = 0x0ff04

;       No Error -- good I/O
ABSOLUTE ok = 0x0ff00

;       SCSI status returned is check condition
ABSOLUTE check_cond = 0x0fffe

;       SCSI status returned is busy
ABSOLUTE busy = 0x0fffd

;       SCSI status returned is reservation conflict
ABSOLUTE reserved = 0x0fffc
```

```
;       SCSI status returned is unknown
ABSOLUTE bad_status = 0x0fffb

;       Error -- unexpected phase after a data transfer
ABSOLUTE err5 = 0x0ff05

;       Error -- unexpected msg in phase before command phase
ABSOLUTE err6 = 0x0ff06

;       Error -- extended msg present before a command phase
ABSOLUTE err7 = 0x0ff07

;       Error -- save data pointers before a command phase
ABSOLUTE err8 = 0x0ff08

;       Error -- disconnect before command phase
ABSOLUTE err9 = 0x0ff09

;       Error -- save data pointers after the command phase
ABSOLUTE err10 = 0x0ff10

;       Error -- unexpected msg after command phase
ABSOLUTE err11 = 0x0ff11

;       Error -- extended message present after the command phase
ABSOLUTE err12 = 0x0ff12

;       Error -- disconnect after a command phase
ABSOLUTE err13 = 0x0ff13

;       Error -- save data pointers after a data transfer
ABSOLUTE err14 = 0x0ff14

;       Error -- unexpected message after a data transfer
ABSOLUTE err15 = 0x0ff15

;       Error -- extended message after a data transfer
ABSOLUTE err16 = 0x0ff16

;       Error -- disconnect after a data transfer
ABSOLUTE err17 = 0x0ff17

;       Error -- Message in not received after reselection
ABSOLUTE err18 = 0x0ff18

;       Error -- Data in phase after reselection and i.d. msg rcvd
ABSOLUTE err19 = 0x0ff19

;       Error -- Data out phase after reselection and i.d. msg rcvd
ABSOLUTE err20 = 0x0ff20
```

```
;       Error -- Msg in phase after reselection and i.d. msg rcvd
ABSOLUTE err21 = 0x0ff21

;       Error -- Status phase after reselection and i.d. msg rcvd
ABSOLUTE err22 = 0x0ff22

;       Error -- Msg out phase after reselection and i.d. msg rcvd
ABSOLUTE err23 = 0x0ff23

;       Error -- Unknown phase after reselection and i.d. msg rcvd
ABSOLUTE err24 = 0x0ff24

;       Error -- Selected as a target
ABSOLUTE err25 = 0x0ff25

;       Error -- Unexpected message rcvd instead of command complete
ABSOLUTE err26 = 0x0ff26

;       SCSI I/O entry point. This address must be loaded into the
;       53C710 before initiating a SCSI I/O.
ENTRY start_up


;       SCRIPTS AREA

;       ************************************
;       This is the entry point for a SCSI I/O
;       ************************************
start_up:

;       This is the SCRIPT for a standard SCSI I/O

;       First, select the device with attention and go to an
;       alternate reselect address. If a reselection or selection
;       happens before the selection can execute, the chip will
;       change roles if required.
SELECT ATN FROM device, PASS(&Resel)

;       If the next phase is status, go to end. Wait for valid
;       phase before performing the comparison.
JUMP REL(end), WHEN STATUS

;       If not msg out phase, interrupt. Do not wait for phase.
INT err1, IF NOT MSG_OUT
```

```
;       ***********************************************
;       Label for retry loop to resend I.D. msg on error
;       ***********************************************
retry:

;       The expected case after selection is I.D. message out to the
;       device.  Move the I.D. message from the send message buffer.
;       Do not wait for a phase change.
MOVE FROM sendmsg, WHEN MSG_OUT

;       If the target remains in the message out phase after the
;       initial messages have been sent to the device, retransfer
;       the messages.  Wait for a valid phase (req asserted).
JUMP REL(retry), WHEN MSG_OUT

;       Now check for all expected phases.
JUMP REL(end), IF STATUS

;       Process a message in before the command phase here
JUMP REL(msg1), IF MSG_IN

;       If it is not status, msg in, or command, stop
;       Interrupt if not command phase
INT err2, IF NOT CMD

;       Transfer command bytes to the host
MOVE FROM cmd_adr, WHEN CMD

;       Determine what is coming next.  Is there a message in after
;       the command phase?
JUMP REL(msg2), WHEN MSG_IN

;       Status phase after the command?
JUMP REL(end), IF STATUS

;       Check for data in phase
JUMP REL(input_data), IF DATA_IN

;       Is this a data out phase?
JUMP REL(output_data), IF DATA_OUT

;       Error -- an unexpected phase after a command transfer
INT err3
```

```
;       **********************************
;       Label to process the status phase
;       **********************************
end:

;       Move the status byte in to the buffer area
MOVE FROM status_adr, WHEN STATUS

;       NOTE: an alternative at this point is to determine what the
;       status byte is and jump to a set of routines that will
;       process the command complete message, physical disconnect,
;       and then interrupt with the appropriate status byte error
;       value. Here, the algorithm interrupts if good I/O is not
;       the status byte returned by the target.

;       Was there a check condition
INT check_cond, IF 0x02

;       Is the device busy
INT busy, IF 0x08

;       Is the device reserved
INT reserved, IF 0x018

;       Interrupt for unknown state
INT bad_status, IF NOT 0x00

;       Status value is good I/O, so process the command complete
;       Stop if the next phase is not message in.
INT err4, WHEN NOT MSG_IN

;       Message in if here. It should be a command complete.
MOVE FROM rcvmsg, WHEN MSG_IN

;       Process the message if it is not a command complete
INT err26, IF NOT 0x00

;       At this point, instead of interrupting, the best course
;       would be to examine the message received and react, or to
;       interrupt with a more specific error code.

;       Command complete was received, acknowledge it
CLEAR ACK

;       A physical disconnect should be next
WAIT DISCONNECT

;       Good I/O if here
INT ok
```

```
;       ****************************************
;       This the data out section of the algorithm
;       ****************************************
output_data:

MOVE FROM data_adr, WHEN DATA_OUT

;       If a scatter/gather requirement exists, then this section
;       can be multiple block moves to allow for multiple segments
;       of data.  Also, this section could actually be a jump to a
;       group of block moves that can be patched appropriately at
;       start I/O for the number of segments needed.  The overhead
;       between segment block moves is 500-600 nanoseconds.


;       ****************************************
;       Process what comes after the data transfer
;       ****************************************
check_out:

;       Status phase is the normal next step
JUMP REL(end), WHEN STATUS

;       Is there a message in phase after data transfer
JUMP REL(msg3), IF MSG_IN

;       Unexpected phase detected after data transfer
INT err5


;       ************************************************
;       This is the data in phase portion of the algorithm
;       ************************************************
input_data:


;       If a scatter/gather requirement exists, then this section
;       can be multiple block moves to allow for multiple segments
;       of data.  Also, this section could actually be a jump to a
;       group of block moves that can be patched appropriately at
;       start I/O for the number of segments needed.  The overhead
;       between segment block moves is 500-600 nanoseconds.

MOVE FROM data_adr, WHEN DATA_IN

;       Go check the phase after data in
JUMP REL(check_it)
```

```
;       ***********************************************
;       Process a message in before the command phase
;       ***********************************************
msg1:

MOVE FROM rcvmsg, WHEN MSG_IN

;       Is this an extended message?
JUMP REL(ext_msg1), IF 0x01

;       Is this save data pointers?  Interrupt with ACK set.
INT err8, IF 0x02

;       Is this a disconnect?
JUMP PASS(&disc_proc), IF 0x04

;       Interrupt if any other message with ACK set
INT err6

;       Message is an extended message
ext_msg1:
;       Acknowledge the message just received
CLEAR ACK

;       Move two more messages into the buffer to get the extended
;       message length and opcode for the processor to have
;       available on the interrupt.
MOVE FROM ext_buf, WHEN MSG_IN

;       Interrupt the processor
INT err7

;       Message is a disconnect
disc1:
;       Acknowledge the disconnect message
CLEAR ACK

;       Disconnect before the command if here
WAIT DISCONNECT

;       Interrupt the processor on a disconnect
INT err9
```

```
;       ************************************
;       Message in after the command phase
;       ************************************
msg2:
MOVE FROM rcvmsg, WHEN MSG_IN

;       Is this an extended message?
JUMP REL(ext_msg2), IF 0x01

;       Is this save data pointers?  Interrupt with ACK set.
INT err10 IF 0x02

;       Is this a disconnect?
JUMP REL(disc2), IF 0x04

;       Interrupt if any other message with ACK set
INT err11

;       Message is an extended message
ext_msg2:
;       Acknowledge the message just received
CLEAR ACK

;       Move two more messages into the buffer to get the extended
;       message length and opcode for the processor to have
;       available on the interrupt.
MOVE FROM ext_buf, WHEN MSG_IN

;       interrupt the processor
INT err12

;       Message is a disconnect
disc2:
;       Acknowledge the message
CLEAR ACK

;       Disconnect after the command if here
WAIT DISCONNECT

;       Interrupt the processor on a disconnect
INT err13
```

```
**********************************************
;       Message in after the data transfer phase
**********************************************
msg3:
MOVE FROM rcvmsg, WHEN MSG_IN

;       Is this an extended message?
JUMP REL(ext_msg3), IF 0x01

;       Is this save data pointers?  Interrupt with ACK set.
INT err14, IF 0x02

;       Is this a disconnect?
JUMP PASS(&disc_PROC1), IF 0x04

;       Interrupt if any other message with ACK set
INT err15

;       Message is an extended message
ext_msg3:
;       Acknowledge the message just received
CLEAR ACK

;       Move two more messages into the buffer to get the extended
;       message length and opcode for the processor to have
;       available on the interrupt.
MOVE FROM ext_buf, WHEN MSG_IN

;       Interrupt the processor
INT err16

;       Message is a disconnect
disc3:
;       Acknowledge the message
CLEAR ACK

;       Disconnect before the data transfer if here
WAIT DISCONNECT

;       Interrupt the processor on a disconnect
INT err17
```

```
;       ********************************************************
;       This is the section of code to process a reselect or select
;       when a select I/O command was executed
;       ********************************************************
;
resel_adr:

;       Wait for reselect as the most probable event
WAIT RESELECT select_adr

;       The initiator was reselected, so process the possibilities
INT err18, WHEN NOT MSG_IN

;       I.D. message in is the only expected SCSI phase here
MOVE FROM rcvmsg, WHEN MSG_IN

;       At this point, if the system integrator knows the possible
;       SCSI device I.D.'s possible, the algorithm can compare for
;       each known I.D. and react accordingly.  An I/O could even be
;       restarted if the SCSI bus configuration is exactly known.

;       Data in phase after reselection and i.d. transfer
INT err19, WHEN DATA_IN

;       Data out phase after reselection and i.d. transfer
INT err20, IF DATA_OUT

;       Message in phase after reselection and i.d. transfer
INT err21, IF MSG_IN

;       Status phase after reselection and i.d. transfer
INT err22, IF STATUS

;       Message out phase after reselection and i.d. transfer
INT err23, IF MSG_OUT

;       Unknown phase after reselection and i.d. transfer
INT err24
```

```
;       ************************************************************
;       The chip was in an initiator role, but it has been selected
;       by another device on the SCSI bus.  It is now in the target
;       role.  One could implement the complete SCSI SCRIPTS target
;       algorithm here, or simply interrupt with an error message.
        ************************************************************
select_adr:

INT err25

;       Definition Area TARGET ROLE

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;* The following are variable data values provided  *
;* external to the compiler and resolved at run-time *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


;       offset for count and address
ABSOLUTE msg_buf=8

;       Command byte offset for count & address
ABSOLUTE cmd_buf=0x010

;       Input message offset for count and address
ABSOLUTE msg_buf2=0x018

;       Buffer offset for the initiator i.d.
ABSOLUTE initiator=0

;       user data buffer offset for count and address
ABSOLUTE data_addr=0x020

;       Status buffer offset for count and address
ABSOLUTE stat_adr=0x0400
```

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;* Absolute values are stored in DNAD Register*
;* for purposes of interrupt processing          *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

ABSOLUTE error1 = 0x0ff01

;       ATN is on after the i.d. message is sent in to the initiator
ABSOLUTE error2 = 0x0ff02

;       ATN is on after the command bytes are sent to the initiator
ABSOLUTE error3 = 0x0ff03

;       Atn is on after the disconnect message is sent to the ;initiator
ABSOLUTE error4 = 0x0ff04

;       ATN on after i.d. message sent to the initiator after a
;       reselect operation is complete
ABSOLUTE error5 = 0x0ff05

;       ATN is on after user data is sent into the initiator
ABSOLUTE error6 = 0x0ff06

;       ATN is on after the status byte is sent
ABSOLUTE error7 = 0x0ff07

;       ATN is on after the command complete message is sent
ABSOLUTE error8 = 0x0ff08

;       Entry Point for the target role
ENTRY start_up

;       Entry point for a target reselect
ENTRY resel_in
```

```
;       SCRIPTS AREA

;       ********************************************
;       This is the entry point for a SCSI target I/O
;       ********************************************
start_up:

;       First wait for a selection by the initiator and jump to the
;       alternate address if reselected.
WAIT SELECT rel(resel_adr)

;       Move the i.d. message into the message buffer
retry_id:
MOVE FROM msg_buf, WITH MSG_OUT

;       If the initiator sets ATN, go process that condition
JUMP Rel(id_atn), IF ATN

continue_id:

;       Move the command bytes in to the target buffer
MOVE FROM cmd_buf, WITH CMD

;       Note that though a 1 is in the command count field, the chip
;       will automatically transfer in the correct number of bytes
;       based on the SCSI command op code.
;       If the initiator sets ATN, go process that condition
JUMP REL(cmd_atn), IF ATN

continue_cmd:

;       In this algorithm, an automatic disconnect is assumed after
;       the SCSI command is received into the buffer.  However, the
;       first byte of the command may be compared against a set of
;       opcode values to determine if this specific command should
;       disconnect or not.

;       Send in the disconnect message
MOVE FROM msg_buf2, WITH MSG_IN

;       If the initiator sets ATN, go process that condition
JUMP REL(disc_atn), IF ATN

continue_disc:

;       Now get off the bus
DISCONNECT
```

```
;       ****************************************
;       Entry point for reselecting the initiator
;       ****************************************
;
resel_in:

;       Perform the reselect and jump to resel_adr if a reselection
;       happens while trying to do the reselect
RESELECT FROM initiator REL(resel_adr)

;       Move the reselect i.d. message into the initiator
retry_resel:
MOVE FROM msg_buf2, WITH MSG_IN

;       If the initiator sets ATN, go process that condition
JUMP REL(resel_atn), IF ATN

continue_resel:

;       Now move the data bytes into the initiator
MOVE FROM data_adr, WITH DATA_IN

;       Note that this could easily be changed to a data out command
;       by patching the phase section of the command, or using a
;       jump command that can be patched to transfer control to a
;       section of code that is either the data out or data in algorithm.
;       If the initiator sets ATN, go process that condition.
JUMP REL(data_atn), IF ATN

continue_data:
```

```
;       ************************************************************
;       If a scatter/gather requirement exists, then this data
;       transfer section can be multiple block moves for the
;       multiple segments of data.  Also, the section could be a
;       jump to a group of block moves that had been patched
;       appropriately at start I/O for the exact number of segments desired.
;       ************************************************************

;       Now move in the status byte
MOVE FROM stat_adr, WITH STATUS

;       If the initiator sets ATN, go process that condition
JUMP Rel(stat_atn), IF ATN

continue_stat:

;       Move the command complete message in
MOVE FROM msg_buf2, WITH MSG_IN

;       If the initiator sets ATN, go process that condition
JUMP REL(cc_atn), IF ATN

continue_cc:

;       Now physically disconnect
DISCONNECT

;       ******************************************************
;       If the wait for select or reselect fails, this is the label
;       for the alternate address
;       ******************************************************
resel_adr:

INT error1
```

```
;       ********************************************************
;       If the initiator turns on ATN after the i.d. message comes
;       out, this is the code for processing what comes next.
;       ********************************************************
id_atn:

;       Move the message byte from the initiator out to the message buffer
MOVE FROM msg_buf WITH MSG_OUT

;       At this point, the user may decide to use scripts to program
;       at a very detailed level or simply interrupt with one user
;       error code.  Scripts may be used to check for:
;               • no-op message -- ignore and jump to continue
;               • initiator detected error -- jump to retry
;               • message parity error -- jump to retry
;               • extended message -- as a minimum, get the opcode and
;                 byte count before interrupting the processor

INT error2


;       All the ATN subroutines have the same basic function

cmd_atn:
MOVE FROM  msg_buf, WITH MSG_OUT
INT error3


disc_atn:
MOVE FROM msg_buf, WITH MSG_OUT
INT error4


resel_atn:
MOVE FROM msg_buf, WITH MSG_OUT
INT error5


data_atn:
MOVE FROM msg_buf, WITH MSG_OUT
INT error6


stat_atn:
MOVE FROM msg_buf, WITH MSG_OUT
INT error7


cc_atn:
MOVE FROM msg_buf, WITH MSG_OUT
INT error8
```

Chapter 9
Unique Initiator Sequences for the 53C710

Disk Drive Initiator Sequence

Arbitrate and Select With Atn
Transfer the I.D. message
Transfer the command bytes
Accept the message in -- DISCONNECT
Reselected -- I.D. message in
Data transfer of 1 - 4 user data blocks
Accept SCSI status byte, COMMAND COMPLETE message and wait for bus free

53C710 Strengths in the Disk Drive Environment

- A large number of commands are typically issued to the disk, and the 53C710 offers very little SCSI bus overhead and a minimum of time to initiate an I/O in the host computer.
- The 53C710 can continue to the next scheduled SCSI I/O within SCRIPTS with no interrupt to the external processor for the following:

  Compare for Good I/O status byte

  Interrupt if non-zero

  Jump to the next scheduled I/O if the status is zero (Good I/O)

- The 53C710 can mask certain disk idiosyncrasies.

  For example, if the disk does a SAVE DATA POINTERS before the first DISCONNECT message after the command bytes are transferred, the 53C710 can be programmed to absorb this message with no interrupt to the external processor.

- The 53C710 can process a disconnect message from the disk. See Chapter 11 for a complete description.

Tape Drive Initiator Sequence

Arbitrate and Select With Atn
Transfer the I.D. message
Transfer the command bytes
Accept the message in -- DISCONNECT
Reflected -- I.D. message in
Data transfer of 16K of user data
Accept the message in -- SAVE DATA POINTERS followed by DISCONNECT.
Reselected -- I.D. message in
Data transfer of 16K of user data
*
*
*
Reselected -- I.D. message in
Data transfer of 16k of user data
Accept SCSI status byte, COMMAND COMPLETE message and wait for bus free

Each disconnect (on a 16k boundary) causes an interrupt to the external processor if there are multiple SCSI devices on the SCSI bus. Reselect causes an interrupt in the general case. If this were a single device bus or the system was designed to perform tape only activity on the SCSI bus during backup, then the 53C710 could be programmed specifically for this system. Knowing the tape drive was alone on the bus, the 53C710 could be programmed to:

1) Absorb the SAVE DATA POINTERS.

2) Execute a SCRIPTS command of wait for reselect.

3) Process the SCSI reselect sequence with no interrupts.

4) Initiate the next 16k user data block move.

5) If there is ever a restore pointers, the 53C710 interrupts to allow the external processor to restart the tape I/O.

The 53C710 allows systems integration designs using the SCSI bus with no performance impact to I/O throughput.

See Chapter 7 for another possible algorithm for large blocks of data that use a SCSI SCRIPTS loop.

SCSI Character Oriented Device in the Initiator Role

A SCSI port can be dedicated by the system designer for terminal control. First, a SCSI read command is transferred to the target terminal controller. A stream of user data typed in at the terminals, plus the inserted control bytes in the stream comes back to the initiator. A SCRIPT can be written which looks at the byte stream coming in and sends line control bytes to the processing buffer and data bytes to the data buffer. When certain control bytes are received, the 53C710 can terminate the READ operation and generate a unique interrupt to the external processor.

Writes to the terminal controller can begin automatically when a certain read threshold is reached. The 53C710 can process the READ command cleanup, jump to the WRITE command portion of the SCRIPTS, and automatically start sending data to the terminal controller. The 53C710 can be used in unusual areas to offload any processor and improve performance.

Another implementation of the 53C710 is SCSI printer design, where WRITE is the only operation and control characters also play an important role.

Chapter 10
Special SCSI SCRIPTS Situations

Transferring Large Blocks of User Data

CASE 1

An unexpected Phase change occurs in the middle of a data transfer.

The Block Move command was developed to transfer 4K of user data, but anomalies such as an unexpected phase change after transferring 2K of the data, must be handled by the processor.

Data may be left in the chip on a data out phase, so an interrupt is required to:

1) Clean up the chip on Data Out Phase

2) Change the data address and byte count in the active SCSI SCRIPTS or the Indirect Data Table.

3) Receive the message byte via SCSI SCRIPTS (e.g., load the new entry point for resumption of the message in operation).

After the message byte has been received, verify that the message byte is a SAVE DATA POINTERS (if not, interrupt the external processor, or process that message), and jump to the SCSI SCRIPTS entry point that will resume the data transfer previously interrupted.

CASE 2

The expected burst size is known ahead of time and is extremely predictable.

At systems integration time, set this burst size, so that each Block Move command can equal the burst size. The SCSI SCRIPTS logic becomes the following:

- Block Move of burst size.
- Call subroutine (after waiting) if the next phase is not a data phase. (The subroutine should process the SAVE DATA POINTERS message in and return.)
- Block Move of burst size
- Call subroutine (after waiting) if the next phase is not a data phase.

Using this logic, all phase changes are assumed to come on a Block Move command boundary, so no bytes will be left in the chip when a phase change occurs. There is a small penalty for fetching the call subroutine command (500 nanoseconds per SCSI SCRIPTS). But a system interrupt (minimum 80 microseconds) will be saved by avoiding the extra interrupt.

CASE 3

The expected burst size is NOT known ahead of time.

Use the same logic as in Case 2, but make the Block Move byte count equal to the device block size. The assumption is that a phase change will come only on the device's block boundary. The SCSI SCRIPTS fetching overhead depends on the ratio of the device block size to the burst size. However, an extra 10 microseconds is small when compared to the external processor interrupt time of at least 80 microseconds. Refer to Chapter 7 for another way of writing the SCSI SCRIPTS to implement **CASE 3.**

How a SAVE DATA POINTERS Can be Processed by the Initiator

CASE 1

A message received during a Block Move command offers 2 possibilities:

1. Data in phase
2. Data out phase

Data-in Phase

During the data in phase, all bytes in the 53C710 are sent to the DMA core and into system memory. When no bytes are left in the chip, all execution stops and an interrupt is generated to the external processor. To save the I/O state, update the current SCSI SCRIPTS with the memory address and byte count located in the 53C710. Save a pointer to this SCSI SCRIPTS in the system I/O structure so that the I/O can easily be rescheduled. The chip's SCSI SCRIPTS pointer value is actually the current SCSI SCRIPTS address plus eight. So the saved value must be the SCSI SCRIPTS pointer value minus eight.

Data-out Phase

If the phase is data out, the 53C710 is full of data bytes going out to the SCSI bus. Execution stops after the phase change and an interrupt is generated to the external processor. At that time, the processor should calculate the number of bytes in the chip, add this value to the chip's byte count, subtract from the chip's memory address pointer, and store these values in the current SCSI SCRIPTS. A pointer to the SCSI SCRIPTS (minus eight) must be saved in some I/O structure for rescheduling. This saved value is the entry point for resuming the data transfer portion of the I/O, depending on the outcome of the phase change.

CASE 2

A message comes in on a Block Move command boundary.

If no test for data phase was placed between Block Move commands, then the 53C710 will fetch the next command and start processing it. When the phase change actually occurs, the 53C710 may have data in it, so the processing is exactly the same as CASE 1 above.

If a wait and test for data phase command is inserted between each Block Move (burst size is known or the block size is used in each Block Move command), then an interrupt is generated to signal the processor to save a pointer to the next Block Move command. A SCSI SCRIPTS to receive message bytes is executed, and the I/O can be resumed by reloading the saved SCSI SCRIPTS pointer.

Alternatively, the message processing SCSI SCRIPTS could have a jump command as its last command. The jump to address would be the entry point of the resume SCSI SCRIPTS pointer so that the interrupted I/O can start up again easily.

This JUMP command must have been updated by the external processor or by the 53C710 at the time of a SAVE DATA POINTERS message.

Chapter 11
Multi-Tasking I/O Using SCSI SCRIPTS

Multi-Threaded I/O Using SCSI SCRIPTS

A design goal of the 53C710 is to allow the user to perform multi-threaded I/O with no external processor intervention.

Four distinct parts exist in a multi-threaded SCSI SCRIPTS algorithm:

- A main SCRIPT.
- A scheduler SCRIPT.
- A reselect SCRIPT.
- A disconnect SCRIPT.

All are involved during multi-threaded I/O. Some of the command areas must be written by the 53C710; thus, key SCRIPTS must be stored in random access memory (RAM).

Main SCRIPT

Only one copy of this SCRIPT is required to service any number of outstanding I/Os. This SCRIPT performs the standard operations associated with a SCSI command (for example, transfer messages, commands, data, and so forth).

A context switch from one I/O to another is performed by loading the Table Indirect Data Structure Address into the Data Structure Address (**DSA**) register and then loading the SCRIPTS entry point into the 53C710.

Note that the entry point address is loaded with a simple transfer control (JUMP or CALL) instruction. Because a SCRIPT register read or write can load the DSA address, and the chip can perform a JUMP SCRIPT, the context switch can easily start an I/O or begin a new I/O or switch to a different one. In the main SCRIPT, numerous ***resume*** points exist. When coding the algorithm, each resume point must be identified as the SCRIPT is coded. An answer to the question "If a disconnect message arrived from the target, where must the I/O resume?" must be known throughout the main SCRIPT. In the following paragraphs, which discuss multi-threaded I/O, the importance of this major point will become quite clear.

Scheduler SCRIPTS

This algorithm is executed when an I/O completes, or the target changes to message-in phase and sends in a disconnect message. There is an entry in the scheduler for every possible I/O the system allows to be outstanding to the SCSI bus. Therefore, there is an entry for every Indirect Data Structure Table (that is, one per I/O allowed by the operating system). Each entry in the scheduler consists of the following SCRIPT:

```
Move 4, memory_Address1, DSA
Jump entry_Point
```

or:

```
move 4, memory_Address1, DSA
NOP
```

An I/O is scheduled when the system processor writes an entry to the scheduler SCRIPT. The 53C710 driver routines must identify an unused entry in the scheduler SCRIPT and move a pointer to the data structure into the appropriate memory address of the unused entry. Then a JUMP command must be written to the next line of code. When the 53C710 has no more SCSI SCRIPTS to execute for an I/O, it will jump to the scheduler SCRIPT. For a scheduled I/O, the value at a memory address will be moved into **DSA** and then the chip will transfer to the main SCRIPT entry point. A NOP is then written to the jump just taken so that the same I/O will not be restarted by the 53C710 before it completes. Because the system will not re-use the entry until the I/O is complete, the I/O runs until completion. If there are no I/Os scheduled, the 53C710 should interrupt or wait for reselect if outstanding I/Os exist.

Disconnect SCRIPT

The target device can change phases on the SCSI bus at any time to save state or to disconnect temporarily. If a move command is executing during a phase change and the byte count is not zero, an external interrupt occurs. However, if the 53C710 has completed the move operation, no external interrupt is required and the chip can handle the phase change using SCRIPTS. To automatically process this phase change, the programmer must identify the resume points in the SCSI SCRIPTS as the algorithm is being developed.

The disconnect routine assumes that the chip is completely in the data indirect mode and that an I/O data structure table exists for each possible I/O. Each data structure has the following entries in RAM:

Address	SCRIPT
-16	write synchronous values to 53C710
-8	Jump to the resume point
0	Label: move 4, SCRATCH, Label-4
+8	Jump Scheduler
+16	I/O data structure values

The significance of these SCRIPTS will become clear as the complete multi-threaded SCRIPT is described as follows:

To implement the disconnect, determine the necessary action if a disconnect message comes into the chip. Choose the SCSI SCRIPTS label that should be jumped-to upon the subsequent reselect operation. The following SCRIPT illustrates this principle and how several lines of extra code in the main SCRIPT allows a save state upon receipt of the disconnect message:

```
; jump around the resume label
                Jump  resume1
resume1_base:
; Place the resume address in TEMP
                Call  save_resume

resume1:
        .
        .
        .
        .
; DISCONNECT Message was just received
; resume1 is the restart label
                Jump  resume1_base
```

As this area of the code was written, the label resume1 is recognized as the restart point for SCSI disconnects. When the DISCONNECT message is received, the chip transfers to one statement before the resume point. A CALL instruction at this address will place the address of resume1 into TEMP and transfer control to save_resume. At this routine, the value in TEMP is moved to SCRATCH with the following SCRIPT:

```
save_resume:
; Address of the resume point is in
; TEMP

    Move TEMP (0) to SFBR
    Move SFBR to SCRATCH (0)

    Move TEMP (1) to SFBR
    Move SFBR to SCRATCH(1)

    Move TEMP(2) to SFBR
    Move SFBR to SCRATCH (2)

    Move TEMP(3) to SFBR
    Move SFBR to SCRATCH (3)

; the resume address is now in
; SCRATCH
```

Because any register-to-register move must go through the SCSI First Byte Received (**SFBR**) register, eight SCRIPTS are required to move one 32-bit register to another. Because SCRATCH is not used by any SCRIPT operation, the resume address must be placed there before being written to memory by the Memory Move SCRIPT.

Next, the resume address must be written to memory by the 53C710. At the address pointed to by the **DSA** register is a Memory MOVE command that moves the value in SCRATCH (now the resume point) into the second four bytes of the JUMP command, eight bytes above. The next step is for the SCRIPT to jump to the address in the **DSA** register.

```
; Move contents of DSA to TEMP
   Move DSA (0) to SFBR
   Move SFBR to TEMP(0)
            .
            .
            .
   Move DSA (3) to SFBR
   Move SFBR to TEMP(3)


; now Return to the data structure
      Return
```

Note that a return SCRIPT simply jumps to the address in the **TEMP** register. At this address, the resume address is saved, and the execution continues at the scheduler SCRIPT. Now a SCRIPT is all set to begin at the correct resume point when the correct reselect occurs.

RESUME SCRIPT

In SCSI terminology, the ***nexus*** is a combination of device i.d., logical unit number, and queue tag value. Upon reselection, the 53C710 will decode the nexus, using COMPARE and JUMP SCSI SCRIPT instructions. Upon reselection, the device i.d. is in the **SFBR** or optionally in the Longitudinal Parity Register (**LCRC**).

After a series of COMPARE and JUMP instructions, based on the unique nexus value, the 53C710 will transfer to a unique Memory MOVE command.

```
      Move 4, address, DSA
      Jump set_up
; DSA Register is now correct
```

For each possible nexus allowed in the system, there is one entry. "Address" points to the memory location where the I/O's data structure address is kept. At power-up, the value of address is initialized after all data structures are allocated, and the addresses are fixed in a nexus address table. There is not necessarily a one-to-one correspondence between possible I/Os and possible nexus values. However, if the values are not all fixed the memory-to-memory MOVE instruction must be updated with the correct address at start I/O rather than at power-up. The system designer can decide how to allocate based on requirements.

Before resuming the I/O execution, only one more step is required. At the set_up routine, **DSA** is moved to **TEMP**, and a return is executed to the DSA pointer, minus 16.

```
set_up:
   Move DSA (0) to SFBR
   Move SFBR to TEMP (0)
      .
      .
      .
   Move DSA (3) to SFBR
   Move SFBR to TEMP (3)
   Move TEMP (0) -16 to TEMP (0)
   Return
```

At the data structure, minus sixteen is an instruction that writes the synchronous offset and period to the 53C710; there is then a jump to the resume point.

Upon completion of an I/O, the programmer may want to signal the system processor by one of several mechanisms allowed by the 53C710:

1) Execute an interrupt instruction. A simple method, but it terminates the execution of SCSI SCRIPTS.

2) Write a value to system memory. Termination is unnecessary; yet the processor must poll a software semaphore. With some periodic I/O timer interrupt followed by a read of I/O status areas, this method can work well.

3) Set the semaphore as in 2), but then write to a user-defined pin (first on, then off) to cause an external interrupt. This allows completely interrupt-driven I/O software.

Compared to a system interrupt, fetching SCRIPTS is very fast. More importantly, the programmer is in control of the tradeoffs and can allow the processor more or less work depending on requirements. If system bus latencies are large, then SCRIPTS can also be stored in local memory on a host bus adapter to eliminate the fetch times. There are enough optional features in the 53C710 to allow optimization of many configurations.

Appendix A
53C710 High Performance Considerations Compared to 53C90

This chapter compares firmware required for the 53C710 and the 53C90 to determine how much of a performance boost the 53C710 can offer at a system level (I/Os per second). One microsecond is the time assumed for execution of each external processor instruction.

Sample Input Data Structure

The following data structure is typical at the SCSI hardware driver level when performing an I/O.

```
Device id, Period & Offset
Byte count
Data address
Byte count
Data address
        *
        *
Byte count
Data address'
```

Initializing SCSI SCRIPTS for an I/O and Starting I/O Operations

53C710 Algorithm Description

Refer to the sample initiator SCSI, SCSI SCRIPTS for details about the exact sequence and values to be updated. At the firmware level, the Initiator SCSI SCRIPTS must be updated with the address and count for the various SCSI data and user data required to perform an I/O. In the sample initiator algorithm, 15 values must be fetched indirectly during execution of the SCRIPT. Assuming the user data structure is in the format required by the SCSI SCRIPT for indirect fetching, there is no overhead associated with starting the I/O. Using the multi-threaded SCRIPTS algorithm, there is no host processor interrupt upon disconnect or at completion of the I/O.

	Approx time in µs
Executing the initiator algorithm takes about 30 SCSI SCRIPTS fetches and indirect data fetches and decodes.	
The total overhead is	**30**
The total time is	**30**

Using the interrupt and continue feature allowed by user programmable bits, in a multi-threaded environment, the next I/O can proceed while the previous I/O complete interrupt is processed by the system. Thus, the overhead of this interrupt is ignored because work is proceeding.

53C90 Algorithm Description.

The firmware begins the sequence by preloading the 53C90 FIFO with the SCSI i.d. message followed by a 10-byte SCSI command. The firmware sequence involved requires:

```
Loop:
        Read Next Byte
        Write Next Byte
        Go To Loop If Count Not Zero
```

For 11 bytes, the above sequence requires about 33 microseconds. Once the SCSI operation begins, the 53C90 requires the overhead listed below. (Note that each interrupt requires some reads and processing to determine the exact cause of the chip's interrupt.) Assume that an extra 20 microseconds is required for each interrupt for a total of 100 (80 + 20) microseconds.

The following sequence is required to perform a SCSI operation.

	µsec
Send the SCSI command	033
Interrupt -- msg in phase	100
Interrupt -- msg accepted	100
Interrupt -- physical disc	100
Interrupt -- reselected	100
Initialize DMA Logic	025
Interrupt -- transfer complete	100
Interrupt -- completion seq	100
Interrupt -- msg accepted	100
Interrupt -- physical disc	100
Total time	858 microseconds

Conclusion

The 53C710 requires less than 5 % of the normal firmware overhead associated with a 53C90, in the simplest case. To further compare the chips, note that a SAVE DATA POINTER operation in the 53C90 requires two processor interrupts (200 µsec) and ***no*** interrupts using the 53C710

Each data segment in a scatter-gather situation requires 125 µsec on the 53C90 (one interrupt plus DMA initialize), but only 1 µsec on the 53C710 (500 nanosecond instruction fetch, plus indirect data fetch). Thus, an I/O that required four data segments in a scatter-gather mode would require 500 µsec on the 53C90 and 4 µsec on the 53C710 for user data transfer. These factors translate into a four-segment data transfer as follows.

53C90
(858) + (3x125) = (858+375)
1233 µsec per I/O

53C710
34 µsec per I/O

To translate this improvement into I/O's per second, assume a 4K data transfer size, consisting of four 1K segments in host memory, a target overhead of one millisecond (excluding seek times), and a 4-megabyte per second user data transfer rate on the SCSI bus.

Function	53C90 In msec	53C710 In msec
Data Transfer Time	1.00	1.00
Target overhead	1.00	1.00
Host Overhead	1.25	0.034
Total times	3.25	2.034
I/O's Per Second	**307**	**491**

In this projected environment, a system can increase its throughput rate by ***sixty percent*** by using the 53C710 and reducing host computer firmware overhead. With the types of buffered SCSI disk drives currently available, the 53C710 eliminates the host computer firmware as the high performance bottleneck.

Remember that a 125 µsec delay between user data segments may cause a disk drive to slip a revolution translating into a dramatic decrease in data throughput.

Without the 53C710, to increase system level performance, designers must eliminate each delay. The 53C710 can remove much of the host overhead associated with each I/O.

Appendix B
53C710 System Bus Utilization

The 53C710, in the laboratory environment transfers 512 bytes of user data at the rate of 6,666 transfers per second (150 microseconds per I/O). The synchronous SCSI burst rate is set at 5 Mbytes per second. This I/O's per second rate is a limit for the 53C710, because no firmware intervention is required.

A real concern is host bus utilization, or "Does the 53C710 affect host computer performance significantly?" This appendix provides information about host bus usage when the SCSI bus is saturated at a block size of 512 bytes.

Host Bus Time To Fetch A SCSI SCRIPTS Command

80 nsec -- Arbitrate and bus settle
80 nsec -- Fetch 4 bytes
80 nsec -- Fetch 4 bytes
40 nsec -- Bus settle time
280 nsec -- Total time

Completing an I/O requires 14 SCSI SCRIPTS.

```
select with ATN
jump error, when not MSG_OUT
move FROM msg_buf, when MSG_OUT
jump error, when not CMD phase
move FROM cmd_buf, when CMD
jump error, when not DATA_IN
move FROM data_buf, when DATA_IN
jump error, when not STATUS
move FROM status_buf, when STATUS
jump error, when not MSG_IN
move FROM msg_buf, when MSG_IN
clear ack
wait disconnect
int 0x001
error:
int 0x0ff
```

The time required to execute the SCSI SCRIPTS with no exception conditions is as follows.

Indirect fetch 5x280 =1.40µsec
SCRIPT fetch 14x 280 =5.32µsec

Total:

6,666 x 5.32 = 35.4 B/sec
(total fetch time per second)

The fetch time is **3.5%** of the available system bus time (one second).

Fetching data across the system bus requires:

Time in nsec	Instruction	
200	I.D. msg fetch =	80 (data fetch) + 80 (arbitrate) + 40 (settle)
360	command fetch =	240 (three data fetches) + 120 (arbitrate + settle)
200	Status byte fetch	
200	COMMAND COMPLETE message	
960	**Total time per SCSI command**	

Total SCSI-related data fetch time is:

6,666 X 960 = 6.4 msec

which is 0.64% of the available system time (one second).

Total overhead time is:

0.64% + 3.5% = 4.14% of the time available

The effective user data transfer rate is 3,333 Mbytes per second, or about 6.66% of the available system bandwidth. Including time for bus arbitration, the available system bandwidth being absorbed by user data transfer is about 8%.

Conclusion

Therefore, the total time to saturate the SCSI bus takes **12.2%** of a processor bus available with a block size of 512 bytes per SCSI command.

Using larger block sizes lowers SCSI command overhead (fewer commands per second) and increases the data transfer rates. As the block size increases, the SCSI overhead per byte of user data decreases.

Appendix C
Use of the Sig_p Bit

Use of the Sig_p Bit in the 53C710

Use of the standard commands to route a bus initiated interrupt, assuming that the 53C710 compatibility bit is on, and the device is in the initiator role. The assumption is that sig_p is only used to signal that an I/O is ready for execution, and has already been scheduled. If selection is in progress or a select/reselect happens, then sig_p can be reset, because the new I/O will be executed when the scheduler function gets to it. The system processor will check the connected bit before setting the sig_p bit to signal that an I/O is to be executed immediately.

```
    SELECT FROM buffer, alternate1
; selection happened if execution gets here
                .
                .
alternate1:
; assume a reselect if here
    WAIT RESELECT, alternate2
; reselected if here, proceed with processing
                .
                .
alternate2:
; got here because of a sig_p bit set or was
; selected. Did the sig_p bit get set after the
; sel/resel occurred and just before the wait?
    MOVE ISTAT and sig_bit to SFBR
; reset it and do the wait again
    Move CTEST3 to SFBR
    JUMP alternate1 if sig_bit
alt2:
; can only have been selected if here
    WAIT SELECT, alternate3
    SET TARGET
; selected if here, proceed with processing in
; target mode
                .
                .

alternate3:
; got here because of a sig_p bit set or error
; Did the sig_p bit get set after the select
; occurred and just before the wait select?
    MOVE ISTAT and sig_bit to SFBR
; reset it and do the wait again
    Move CTEST3 to SFBR
    JUMP alt2 if sig_bit

; should never get here
    INT big_error
```

Aborting a Wait Reselect or Wait Selection SCSI SCRIPT, assuming that the 53C710 compatibility bit has been set and the device is in the initiator role.

```
reselect_entry:
    WAIT RESELECT, alt_sig_p1
; if here, got reselected
                .
                .
                .

select_entry:
    WAIT SELECT, alt_sig_p1
    SET TARGET
; if here, got selected -- change to target
                .
                .
                .

alt_sig_p1:
    MOVE ISTAT and connect_bit to
SFBR
; test the SCSI connected bit
    JUMP alt_sig_p2, if connect_bit
; either the chip got selected, reselected, or the
; sig_p bit was set
    MOVE ISTAT and sig_bit to SFBR
; test the sig_p bit first
    JUMP sig_p_set, if sig_bit
; big error if here -- not connected and sig_p was
; not set
    INT big_error1
alt_sig_p2:
; Bus initiated interrupt occurred if here --
; connected bit is on. First reset the sig_p bit, so
; the alternate jump is NOT taken.
    MOVE CTEST2 to SFBR
    WAIT RESELECT, alt_sig_p3
; process the reselection
                .
                .
                .
```

```
alt_sig_p3:

; got selected
    SET  TARGET

                •
                •
                •

sig_p_set:
; System processor has set the sig_p bit.
; Reset it and service the system request.

    MOVE  CTEST2  to  SFBR

                •
                •
                •
```

Appendix D
Compiler Script Examples

SAMPLE SCSI SCRIPTS Scource File

```
;*******************************************************
;* The following are variable data values provided     *
;* external to the compiler and resolved at run-time   *
;*******************************************************

; Definition area INITIATOR ROLE

; Target Device I.D. offset in the data table.
EXTERN device

EXTERN status_adr

; Ten byte buffer address offset.
EXTERN sendmsg

; Ten byte buffer address offset.
EXTERN rcvmsg

; Buffer address offset for the SCSI command
EXTERN cmd_adr

; Address of user data buffer
EXTERN data_adr

;*******************************************************
;* Absolute values are stored in DSPS Register         *
;* for purposes of interrupt processing                *
;*******************************************************

;*******************************************************
;* Note that 0x0 precedes the interrupt status         *
;* values and designates a hex value                   *
;*******************************************************

ABSOLUTE err1 = 0x0ff01

; Error -- unexpected SCSI phase before command phase
ABSOLUTE err2 = 0x0ff02

; Error -- unexpected SCSI phase after a command
ABSOLUTE err3 = 0x0ff03

; Error -- expected status phase
ABSOLUTE err4 = 0x0ff04

; No Error -- good I/O
ABSOLUTE ok = 0x0ff00

; Error -- expected message outphase
ABSOLUTE err5 = 0x0ff05

; Error -- expected message command complete
ABSOLUTE err6 = 0x0ff06
```

```
;*******************************************************
;* The following shows how you can use the PASS        *
;* capability of the compiler to pass C code to the    *
;* output file                                         *
;*******************************************************

PASS(#include "NCR.h")
PASS(extern char line[];)

PROC sample:
; select the device with attention on
select atn from device, REL (resel_adr)

; if the next phase is not msg_out, interrupt
int err1 when not MSG_OUT

; sent the i.d. message out to the target
move FROM sendmsg, when MSG_OUT

; if next phase is not command, interrupt
int err2 when not CMD

; send the command bytes
move FROM cmd_adr, when CMD

; go to process cleanup if status phase
jump REL (end) when STATUS

; process data in phase
jump REL (input_data) if DATA_IN

; or data out phase
jump REL (output_data) if DATA_OUT

; unexpected phase if here
int err3

; process the data in phase
input_data:
move FROM data_adr, when DATA_IN

; and go process status
jump REL (end)

; process the data out phase
output_data:
move FROM data_adr, when DATA_OUT

; interrupt if not status phase
end:
int err4 when not STATUS

; move the status byte into memory
move FROM status_adr, when STATUS

; interrupt if message in is not next
int err5 when not MSG_IN
```

```
; move the command complete byte in
move FROM rcvmsg, when MSG_IN

; interrupt if not command complete
int err6 if not 00

; accept the message if there are no problems
clear ack

; wait for a physical disconnect
wait disconnect

; interrupt with an I/O complete
int ok
resel_adr:
int ok
```

SAMPLE LIST FILE

```
1   ;*******************************************************
2   ;* The following are variable data values provided     *
3   ;* external to the compiler and resolved at run-time   *
4   ;*******************************************************
5
6   ; Definition area INITIATOR ROLE
7
8   ; Target Device I.D. offset in the data table.
9   EXTERN device
10
11  EXTERN status_adr
12
13  ; Ten byte buffer address offset.
14  EXTERN sendmsg
15
16  ; Ten byte buffer address offset.
17  EXTERN rcvmsg
18
19  ; Buffer address offset for the SCSI command
20  EXTERN cmd_adr
21
22  ; Address of user data buffer
23  EXTERN data_adr
24
25  ;*******************************************************
26  ;* Absolute values are stored in DSPS Register         *
27  ;* for purposes of interrupt processing                *
28  ;*******************************************************
29
30  ;*******************************************************
31  ;* Note that 0X0 precedes the interrupt status         *
32  ;* values and designates a hex value                   *
33  ;*******************************************************
34
35  ABSOLUTE err1 = 0x0ff01
36
37  ; Error -- unexpected SCSI phase before command phase
38  ABSOLUTE err2 = 0x0ff02
39
40  ; Error -- unexpected SCSI phase after a command
41  ABSOLUTE err3 = 0x0ff03
42
43  ; Error -- expected status phase
44  ABSOLUTE err4 = 0x0ff04
45
46  ; No Error -- good I/O
47  ABSOLUTE ok = 0x0ff00
48
49  ; Error -- expected message outphase
50  ABSOLUTE err5 = 0x0ff05
51
52  ; Error -- expected message command complete
53  ABSOLUTE err6 = 0x0ff06
54
55  ;*************************************************
56  ; The following shows how you can use the PASS     *
```

```
; capability of the compiler to pass C code to the *
; output file                                       *
;***************************************************
#include "NCR.h" PASS(#include "NCR.h")
extern char line[]; PASS(extern char line[];)

00000000: PROC sample:
                                ; select the device with attention on
00000000: 47000000 00000098 select atn from device, REL (resel_adr)

                                ; if next phase is not msg_out,interrupt
00000008: 9E030000 0000FF01 int err1 when not MSG_OUT

                                ; sent the i.d. message out to the target
00000010: 1E000000 00000000 move FROM sendmsg, when MSG_OUT

                                ; if next phase is not command, interrupt
00000018: 9A030000 0000FF02 int err2 when not CMD

                                ; send the command bytes
00000020: 1A000000 00000000 move FROM cmd_adr, when CMD

                                ; go to process cleanup if status phase
00000028: 838B0000 00000030 jump REL (end) when STATUS

                                ; process data in phase
00000030: 818A0000 00000010 jump REL (input_data) if DATA_IN

                                ; or data out phase
00000038: 808A0000 00000018 jump REL (output_data) if DATA_OUT

                                ; unexpected phase if here
00000040: 98080000 0000FF03 int err3

                                ; process the data in phase
00000048: input_data:
00000048: 19000000 00000000 move FROM data_adr, when DATA_IN

                                ; and go process status
00000050: 80880000 00000008 jump REL (end)

                                ; process the data out phase
00000058: output_data:
00000058: 18000000 00000000 move FROM data_adr, when DATA_OUT

                                ; interrupt if not status phase
00000060: end:
00000060: 9B030000 0000FF04 int err4 when not STATUS

                                ; move the status byte into memory
00000068: 1B000000 00000000 move FROM status_adr, when STATUS

                                ; interrupt if message in is not next
00000070: 9F030000 0000FF05 int err5 when not MSG_IN

                                ; move the command complete byte in
00000078: 1F000000 00000000 move FROM rcvmsg, when MSG_IN

```

```
115                                 ; interrupt if not command complete
116 00000080: 98040000 0000FF06 int err6 if not 00
117
118                                 ; accept the message if there are no problems
119 00000088: 60000040 00000000 clear ack
120
121                                 ; wait for a physical disconnect
122 00000090: 48000000 00000000 wait disconnect
123
124                                 ; interrupt with an I/O complete
125 00000098: 98080000 0000FF00 int ok
126 000000A0: resel_adr:
127 000000A0: 98080000 0000FF00 int ok

Symbol Name             Value       Type
-------------------------------------------------------------
device                  00000000    EXTERNAL
status_adr              00000000    EXTERNAL
sendmsg                 00000000    EXTERNAL
rcvmsg                  00000000    EXTERNAL
cmd_adr                 00000000    EXTERNAL
data_adr                00000000    EXTERNAL
err1                    0000FF01    ABSOLUTE
err2                    0000FF02    ABSOLUTE
err3                    0000FF03    ABSOLUTE
err4                    0000FF04    ABSOLUTE
ok                      0000FF00    ABSOLUTE
err5                    0000FF05    ABSOLUTE
err6                    0000FF06    ABSOLUTE
include "NCR.h"         00000000    PASS_LABEL
extern char line[];     00000000    PASS_LABEL
sample                  00000000    PROC_LABEL
resel_adr               000000A0    LABEL (REL)
end                     00000060    LABEL (REL)
input_data              00000048    LABEL (REL)
output_data             00000058    LABEL (REL)
```

SAMPLE OUTPUT FILE

```
include "NCR.h"
extern char line[];
typedef unsigned long ULONG;

ULONG   sample[] = {
        0x47000000,     0x00000098,
        0x9E030000,     0x0000FF01,
        0x1E000000,     0x00000000,
        0x9A030000,     0x0000FF02,
        0x1A000000,     0x00000000,
        0x838B0000,     0x00000030,
        0x818A0000,     0x00000010,
        0x808A0000,     0x00000018,
        0x98080000,     0x0000FF03,
        0x19000000,     0x00000000,
        0x80880000,     0x00000008,
        0x18000000,     0x00000000,
        0x9B030000,     0x0000FF04,
        0x1B000000,     0x00000000,
        0x9F030000,     0x0000FF05,
        0x1F000000,     0x00000000,
        0x98040000,     0x0000FF06,
        0x60000040,     0x00000000,
        0x48000000,     0x00000000,
        0x98080000,     0x0000FF00,
        0x98080000,     0x0000FF00

};

#define E_device        0x00000000
ULONG   E_device_Used[] = {
        0x00000000
};

#define E_status_adr    0x00000000
ULONG   E_status_adr_Used[] = {
        0x0000001b
};

#define E_sendmsg       0x00000000
ULONG   E_sendmsg_Used[] = {
        0x00000005
};

#define E_rcvmsg        0x00000000
ULONG   E_rcvmsg_Used[] = {
        0x0000001f
};

#define E_cmd_adr       0x00000000
ULONG   E_cmd_adr_Used[] = {
        0x00000009
};

#define E_data_adr      0x00000000
ULONG   E_data_adr_Used[] = {
```

```
        0x00000013,
        0x00000017
};

ULONG   INSTRUCTIONS        = 0x00000015;
ULONG   PATCHES             = 0x00000000;
```

Appendix E
SCRIPTS™ Compiler Error Messages

Fatal Error: . . .

Fatal Error: No memory. Aborting compiler ...:

There is not enough available memory to read the SCRIPT into RAM.

Fatal Error: Local stack overflow. Aborting compile...:

Please contact NCR immediately, you have an obsolete version of SCRIPTS.

Fatal Error: Cannot open file:

The SCRIPT file cannot be opened or one of the output files (.ERR or .XRF) are corrupt. Compilation is terminated.

Fatal Error: Cannot read file:

The file was opened, but could not be read. Compilation is terminated.

Error: . . .

Error: Expected digit:

While evaluating a number, a character other than a legal digit was encountered.

Error: Expected a separator:

A separator was expected, insert a comma, EOL character or any other legal separator.

Error: Numeric constant has too many digits:

A number, either decimal, hex or binary contains too many digits.

Error: Expected a value:

A value was expected, but instead an operator, pseudo-op, or instruction was encountered.

Error: Undefined variable:

A variable was encountered that was not defined at the beginning of the SCRIPT.

Error: Unknown identifier:

An identifier was encountered that was not a "+", "-", or any other expected separator.

Error: Expected an identifier:

A reserved word was encountered where there should have been an identifier.

Error: Expected a variable:

A pseudo op, instruction, or reserved word was encountered where a variable was expected.

Error: Expected an expression:

A mathematical expression was expected but not found. If you encounter this error message, contact NCR, you have an old version of SCRIPTS.

Error: Expected a reserved word:

A reserved word was expected (WITH, WHEN, IF, etc.) but was not encountered.

Error: Expected a PHASE:

An instruction was used in which a phase was expected and but was not found in the instructions.

Error: Cannot use a RELATIVE in a non address field:

A relative variable was used in a field that was not an address field.

Warning: . . .

Warning: Identifier truncated:

An identifier, such as a label contained more than 32 characters and was truncated.

Warning: Redefinition of variable:

A variable was defined two or more times.

Warning: Duplicate ATN:

ATN has already been set and you are attempting to set it again.

Warning: Duplicate ACK:

ACK has already been set and you are attempting to set it again.

Warning: Undefined label used as entry point:

The label was not defined as an entry point.

Warning: Unused variable:

A variable was defined but not used in the SCRIPT.

Warning: Lost resolution:

A number encountered was too large. For example, using 8 as a SCSI ID. SCSI ID numbers can be no larger than 7.

Warning: Duplicate label:

A label was defined more than once.

UNKNOWN ERROR!

You have just experienced a phenomenon known as cosmic ray bombardment. This is believed to be associated with increased solar flare activity. Fortunately, the effects are not permanent, try again.

Appendix F
Miscellaneous Design Topics

The following paragraphs detail design topics.

Design Topics

The following design topics are discussed.

- SCSI Activity Timer
- Longitudinal Parity Register
- Big/Little Endian Support
- SCRIPTS in a host adapter

SCSI Activity Timer

Some SCSI systems have a system requirement with respect to activity on the SCSI bus.

If there are long periods with no SCSI activity then the SCSI driver must notify the system software that a timeout has occurred. The 53C710 has a built-in 250 millisecond timer that will cause an interrupt when the time expires. If the interrupt is turned off, the system does nothing when the timer expires.

Bit one of the DMA Interrupt Enable register can turn on the timer. When the timer is enabled, all SCSI activity is covered by the timer. If there is no assertion of Transfer/Acknowledge on the bus within 250 milliseconds after selection through disconnect then the interrupt will occur.

Longitudinal Parity Register (LCRC)

For a simple error check of any data passing through the 53C710, there is an 8-bit register that contains a continual exclusive OR of the data. The value in the chip is cleared by any write to the register. A designer can use the information by performing the following:

1) Clear the value with a SCRIPTS write.

2) Move data through the 53C710.

3) Move the generated byte to the SCSI target to be stored with the data.

4) Read in the extra byte on a read, and compare it to the byte generated during the move.

All the extra moves and compares can be done by the 53C710 or by the system processor, depending on the designer's preference.

Note that the **LCRC** doubles as the **SFBR** during a select or reselect. The device i.d. is always written into the **LCRC**. Because a SCRIPT could be writing to the **SFBR** during a SCSI bus-initiated interrupt, the value could be destroyed. Optionally, therefore, the chip can be set to write the device i.d. only to the **LCRC**.

Big/Little Endian Support

There is some support for both big and little Endian in the 53C710. Five areas must be considered when discussing the byte ordering.

SCRIPTS Order

To ensure that all SCSI SCRIPTS are in the correct order, each SCRIPT must be compiled in the target architecture. The C output is a longword value, which will be stored in the memory by the processor and in the correct order for the subsequent execution. If a little Endian SCRIPT is to be executed on a big Endian machine, the bytes will need to be reversed before execution by the 53C710 (in big Endian mode). Note that a PROM cannot be moved from one environment to another without re-ordering bytes within each word.

53C710 Register Access from Firmware

There is a big Endian and a little Endian address mode for the registers. To develop code that works in either mode, simply use equates with an Endian switch that includes the appropriate set of address values. Note that the change is only for byte access. If 32 bits are accessed, there is no change from big to little Endian.

53C710 Register Access from SCSI SCRIPTS

The compiler offers a set of logical names that can be used to access registers. Names do not change when the mode changes, and the binary code required to access a register does not change either.

User Data Byte Ordering

Data transfers to/from system memory from/to the SCSI bus always start at the beginning address and continue until the last byte is sent. No internal re-ordering of the data for either mode occurs. A serial stream of data is assumed, and the first byte on the SCSI bus is associated with the lowest address in system memory.

SCRIPTS in a Host Adapter

Some designs require that SCSI SCRIPTS be fetched from a local ROM rather than from system memory across the bus. Typically, this requirement comes from the desire to avoid traffic on the bus or is caused by large overheads associated with bus arbitration. The 53C710 allows several options in the placement of SCRIPTS and table indirect data.

SCRIPTS and data structures can be placed in system memory.

Using the FETCH pin, external system bus interface hardware can read SCRIPTS locally and all other data from system memory. During SCRIPT fetches, the pin is active, and thus, the access can go locally rather than across the system bus.

In the CTEST8 register is the fetch mode bit. When set, the FETCH pin will deassert during indirect and table indirect read operations. FETCH will be active during SCRIPT fetches only. Thus, external hardware can drive the opcode fetch to one memory area (local ROM) and table indirect fetches to to another area (system RAM). If the bit is not set, then fetch is asserted throughout the instruction fetch.

Thus, the designer can place SCRIPTS, user data, and table indirect data in the most convenient area of memory. Note that the options can be changed dynamically by writing to the registers from SCRIPTS.

Low-level SCSI Code

Pseudocode examples of selection, message out, command, data in, status, and message in.

```
**********************************************************
Selection:                                                *
**********************************************************
      parity check, generation
SCNTL0=0XCC

      C700 i.d.=7,  target i.d.=2
SODL=0X84

      assert BSY
SOCL=0X20

      assert SODL, connected; if not connected, ATN cannot be asserted
SCNTL1=0X50

      low-level mode (Note: Disable low-level mode before starting the
      SCRIPTS' processor.)

DCNTL=0X08

      assert SEL, ATN, BSY
SOCL=0X38

      deassert BSY, keep SEL, ATN
SOCL=0X18

      wait for BSY, asserted by Target
(SBCL & 0X20)=0X20

      deassert SEL, keep ATN
SOCL=0X08

**********************************************************
Message Out                                               *
**********************************************************

      look for REQ and message out
(SBCL & 0X87)=0X86

      identify message
SODL

      message-out phase; a phase match asserts SODL onto the SCSI bus
SOCL=0X0E

      assert ACK, message out, keep ATN
SOCL=0X4E

      wait for REQ deasserted
wait for (SBCL & 0X80)=0X00
```

```
      deassert ACK, ATN; keep message out
SOCL=0X06

*************************************************************
Command                                                      *
*************************************************************

      look for REQ and command
(SBCL & 0X87)=0X82

      initialize command byte
SODL=command byte
      assert ACK, command
SOCL=0X42
      wait for REQ deasserted
wait for (SBCL & 0X80)=0X00
      deassert ACK, keep command
SOCL=0X02
      repeat until last command byte


*************************************************************
Data In                                                      *
*************************************************************
      look for REQ and data in
(SBCL & 0X87)=0X81

SBDL=data byte
      assert ACK, data in
SOCL=0X41
      wait for REQ deasserted
wait for (SBCL & 0X80)=0X0
      deassert ACK, keep command
SOCL=0X02
      repeat until last data byte


*************************************************************
Status                                                       *
*************************************************************

      look for REQ and status
(SBCL & 0X87)=0x83

      ACK, status phase
SOCL=0X43

      SBDL contains status byte
status=SBDL

      wait for REQ deasserted
wait for (SBCL & 0X80)=0X00

      deassert ACK; keep status phase
SOCL=0X03
```

```
**************************************************************
Message in                                                    *
**************************************************************

      look for REQ and message in
(SBCL & 0X87)=0X87

      ACK, message in phase
SOCL=0X47

      SBDL contains message byte
message in=SBDL

      wait for REQ deasserted
wait for (SBCL & 0X80)=0X00

      deassert ACK; keep message-in phase
SOCL=0X07
```

NCR

READER'S COMMENT FORM

F-8763 0687

BOOK TITLE	BOOK NO.	PRINT DATE

To help us plan future editions of this document, please take a few minutes to answer the following questions. Explain in detail using the space provided. Include page numbers where applicable.

Are there any technical errors or misrepresentations in the document?

Is the material presented in a logical and consistent order?

Is it easy to locate specific information in the document?

Is there any information you would like to have added to the document?

Are the examples relevant to the task being described?

Could parts of the document be deleted without affecting the document's usefulness?

Did the document help you to perform your job?

Any general comments?

NAME _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

TELEPHONE NO. ( ) _____

Thank you for your evaluation of this document.
Fold the form as indicated and mail to NCR. No postage is necessary in the U.S.A.

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 3 DAYTON, OHIO

POSTAGE WILL BE PAID BY ADDRESSEE

NCR Corporation
ATTENTION: Publication Services
WHQ-4
1700 S. Patterson Blvd.
Dayton, Ohio 45409

fold

NCR Microelectronic Products Division – Sales Locations

For literature on any NCR product or service call the NCR hotline toll-free

1-800-334-5454

Worldwide Sales Headquarters
1731 Technology Drive, Suite 600
San Jose, CA 95110
(408) 453-0303

Division Plant Locations

NCR Microelectronic Products Division
2001 Danfield Court
Fort Collins, CO 80525
(303) 226-9500

Commercial ASIC Products
Customer Owned Tooling Products
Communication Products
Disk Array Products

NCR Microelectronic Products Division
1635 Aeroplaza Drive
Colorado Springs, CO 80916
(719) 596-5795

Automotive Products
SCSI Products
Graphics Products

NCR Microelectronic Products Division
2850A North El Paso Street
Colorado Springs, CO 80907
(719) 578-3400

Multi-Chip Module Products

North American Sales Offices

Northwest Sales
1731 Technology Drive, Suite 600
San Jose, CA 95110
(408) 441-1080

Southwest Sales
3300 Irvine Avenue, Suite 255
Newport Beach, CA 92660
(714) 474-7095

North Central Sales
8000 Townline Avenue, Suite 209
Bloomington, MN 55438
(612) 941-7075

South Central Sales
17304 Preston Road, Suite 635
Dallas, TX 75252
(214) 733-3594

Northeast Sales
500 West Cummings Park, Suite 4000
Woburn, MA 01801
(617) 933-0778

Southeast Sales
1051 Cambridge Square, Suite C
Alpharetta, GA 30201
(404) 740-9151

International Sales Offices

European Sales Headquarters
Westerndstrasse 193
8000 Munchen 21
Post fach 210370
Germany
49 89 57931199

Asia/Pacific Sales Headquarters
35th Floor, Shun Tak Centre
200 Connaught Road
Central
Hong Kong
852 859 6044

J04921I
0892-5HX