Microelectronic Products Division
Colorado Springs

NCR 53C700

SCSI I/O Processor

Programmer's Guide Rev. 1.0

PRINTING HISTORY

While the information herein presented has been checked for both accuracy and reliability, NCR Corporation assumes no responsibility for either its use or any inaccuracies.

Prepared by NCR Microelectronics Division

Revision No.	Print Date	NCR internal Part No(s). covered by each manual
Preliminary	10/89	609-3400616 only 609-3400625/631 (up to and including)
1.0	3/90	609-3400634 (includes all previous revisions)

Additional Information

NCR 53C700 Data Manual

Trademarks

SCRIPTS is a registered trademark of NCR Corporation
IBM, Micro Channel are registered trademarks of International Business Machines Corporation

Table of Contents

Appendices

List of Figures

Chapter 1
Introduction

NCR SCSI I/O Processor (53C700)

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.

Another performance issue is the scatter/gather operation. With virtual storage so common today, many I/O's gather the data from several physical addresses in system memory. Latencies inherent in the reinstruct 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 (NCR5380) 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 (NCR53C90) 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 H/W 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.

The NCR SCSI I/O Processor (SIOP)

The NCR 53C700 is the first intelligent SCSI host adapter on a chip. A high-performance re-usable 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 Hystersis
- 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-thru protection

The SCSI core in the 53C700 is reusable and designed to migrate to SCSI-2 wide and fast requirements. It offers synchronous transfers up to 6.25 MBytes/sec with asynchronous transfers greater than 5 MBytes/sec. Synchronous offsets up to 8 are supported.

The SCSI core offers low-level register access as well as the high-level control interface. Like first generation SCSI devices, the 53C700 SCSI core can be accessed as a register oriented chip. The ability to sample and assert any signal on the SCSI bus can used for manufacturing test and diagnostic procedures. Loopback diagnostics are supported, the SCSI core may perform self-selection and operate as both an initiator and a target to verify that internal data paths are operational. The 53C700 can test the SCSI pins for physical connection to the board or the SCSI bus.

Unlike previous generation devices, the 53C700 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 2 MIPS processor located on the SCSI chip.

DMA COMPONENT

The DMA component is a bus master DMA chip that attaches easily to the 80486, 80386, 80286, 80386SX, and 80376 processors. It is designed for 25 Mhz 80386 bus timings and may be externally adapted to ISA (AT), EISA, Micro Channel™ , etc..

The 53C700 supports 16 or 32-bit memory and automatically supports misaligned DMA transfers. As with the 80386, data bus enables are provided for each byte lane. An on-chip, 32 byte FIFO allows 2,4, or 8 long words to be burst across the memory bus interface, providing memory transfer rates in excess of 50 MBytes/sec.

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 "ored" 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 16 or 32-bit system memory. Complex SCSI bus sequences are executed independently of the host CPU.

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 (i.e. scanners, communication gateways, etc.) or changes in the SCSI logical bus definitions, or quickly incorporate new or popular options.

The SCSI SCRIPTS processor is how the 53C700, 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 implementation of a 80386 can therefore be identical to the scripts for a 80386SX Micro Channel™ implementation.

After power up and initialization of the 53C700, the chip may be operated in one of two modes:

1) Low level register interface

2) SCSI SCRIPTS chained 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, 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 53C700 requires only a SCSI SCRIPTS start address. All subsequent commands are fetched from external memory. Four bytes (or optionally two) at a time are fetched across the iAPX 286/386 DMA interface and loaded into the command register. Command fetch and decode time is minimal at about 500 nanoseconds. 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.

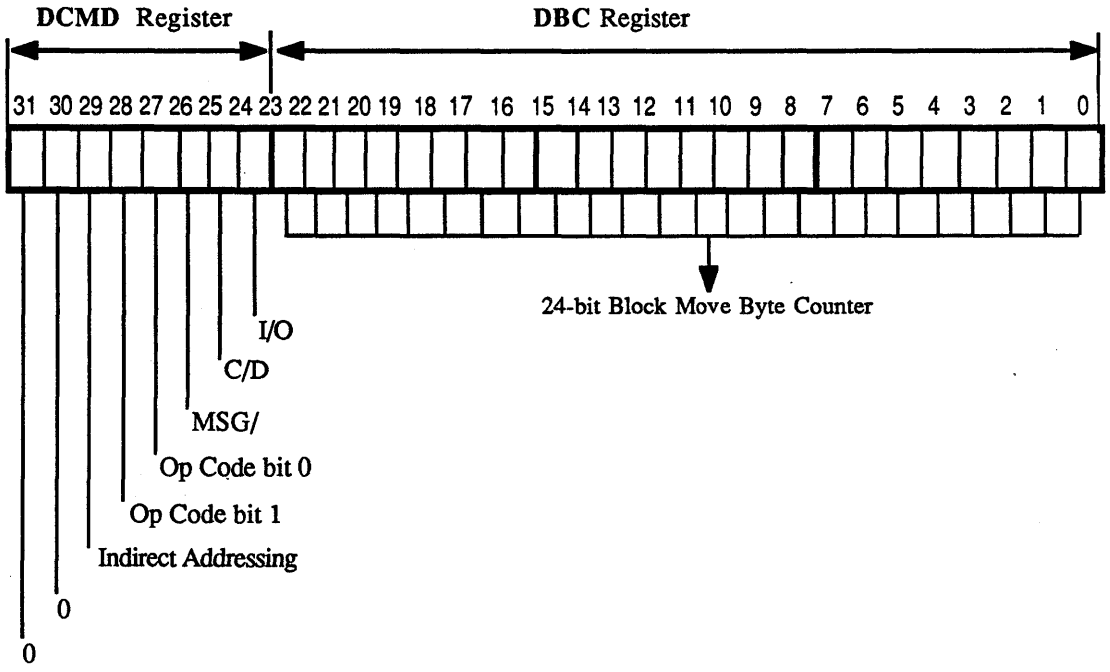
Chapter 2
SCSI SCRIPTS ™ Machine Language Description

This chapter describes each SCSI SCRIPT™ command at the programmer, detailed, bit level. Normally you will use the SCSI SCRIPTS compiler described in 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.

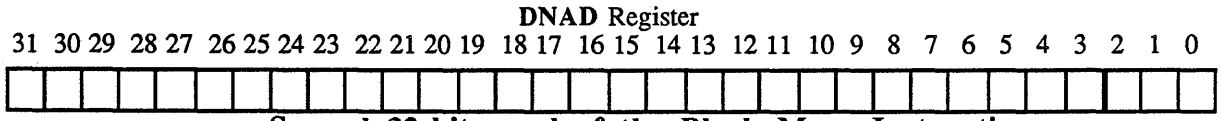
Bits 31-30 are SCSI I/O Processor opcodes.

00 equals Block Move Command
01 equals I/O Command
10 equals Transfer Control Command
11 equals NCR Reserved

BLOCK MOVE COMMAND (00)



First 32-bit word of the Block Move Instructions



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. Write 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.

Also, this feature makes it possible to locate SCSI SCRIPTS in a PROM.

Block Move Opcodes Bit 28-27

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--00

The target role allows DMA 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. 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 be completed, and then an interrupt will occur.

Target Role Function--01,10, or 11

These are illegal values and will generate an invalid command interrupt if the chip is in the target role.

Initiator Role Function--00

Reserved

Initiator Role Function--01

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, and the chip waits for a previously unserviced phase before executing the Block Move command. You can program the 53C700 to pause until the SCSI device it is communicating with goes to the next phase.

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.

Initiator Role Function--10, or 11

These are illegal values and will generate an invalid command interrupt if the chip is in the initiator role.

SCSI Phase Lines Bit 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 (or waits for a new phase and compare).

24-Bit Byte Count, Bit 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.

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) or the address of the actual address (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).

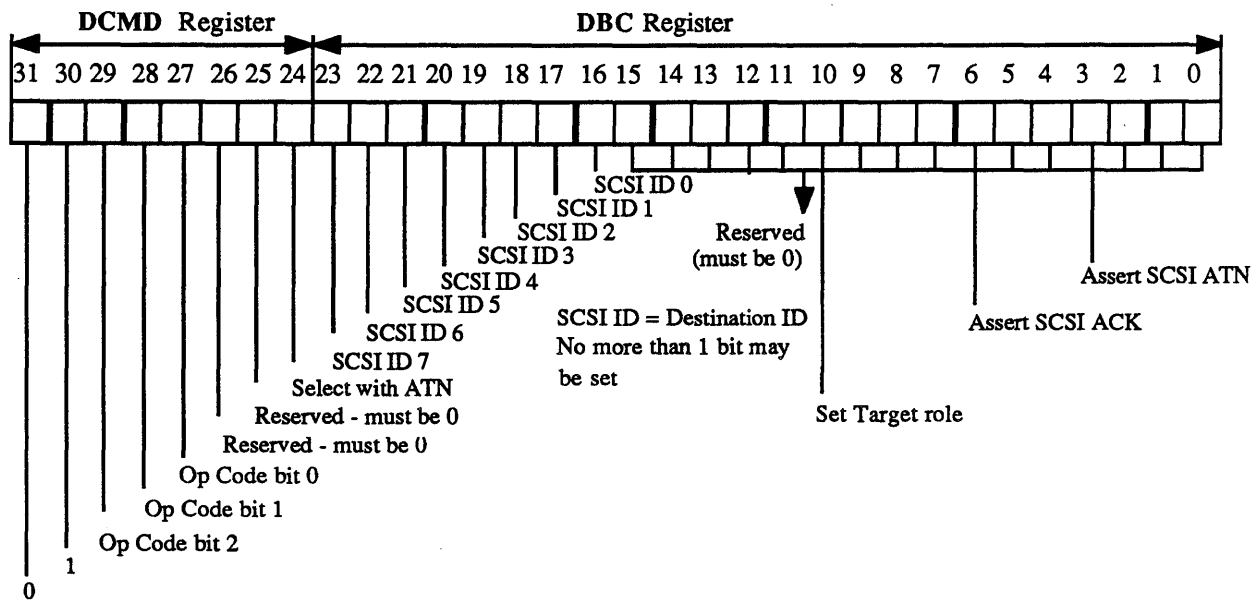
3) The indirect feature allows a table of addresses instead of requiring the address to be in the command.

4) A scatter/gather operation has little performance impact, because the only overhead is 500 nanoseconds (direct mode) or 750 nanoseconds (indirect mode). So, one Block Move command for each segment of data in memory is economical with the SCSI I/O processor architecture.

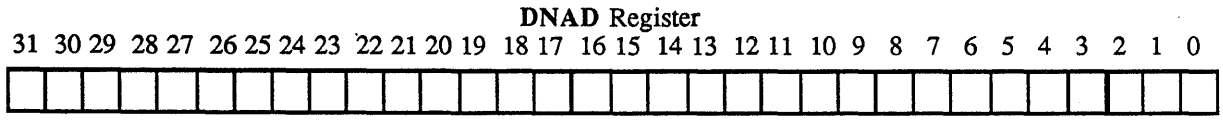
In the initiator role, the Block Move wait feature is useful for high performance SCSI SCRIPTS that do not compare for any unexpected phases before executing a Block Move command. If the phase does not match, then an external interrupt is generated.

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.

I/O Command



First 32-bit word of the I/O Instructions



Second 32-bit word of the I/O Instructions

Figure 2. I/O Instructions

Overview

The I/O command performs select and reselect SCSI operations. Each function defined is a direct command to the SCSI portion of the 53C700. The functions vary if the chip is in the target or initiator role, so the functions are described separately for each role.

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 reserved for future expansion. Using the reserved function codes generates an illegal command interrupt stopping execution .

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 command is successful, then the next sequential instruction is fetched and executed. Note that the target/initiator role automatically change to reflect what is actually happening on the bus.

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), 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.

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 changes to the initiator role and fetches the next command from the address pointed to by the 32-bit jump address, and continues execution.

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.

Target Role -- function 101, 110, 111
These are not currently defined and will cause an illegal command interrupt if used.

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. The target/initiator role automatically changes to reflect bus actions.

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 53C700 can determine exactly what happened and transfer control to the appropriate SCSI SCRIPTS algorithm.

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.

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 53C700. 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.

Note:
With the 53C700 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.

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.

Initiator Role -- function 101, 110, 111
These are not currently defined and will cause an illegal command interrupt if used.

SELECT WITH ATN - Bits 26-24

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 I.D. 7-0 - Bits 23-16

This eight bit field is the I.D. 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

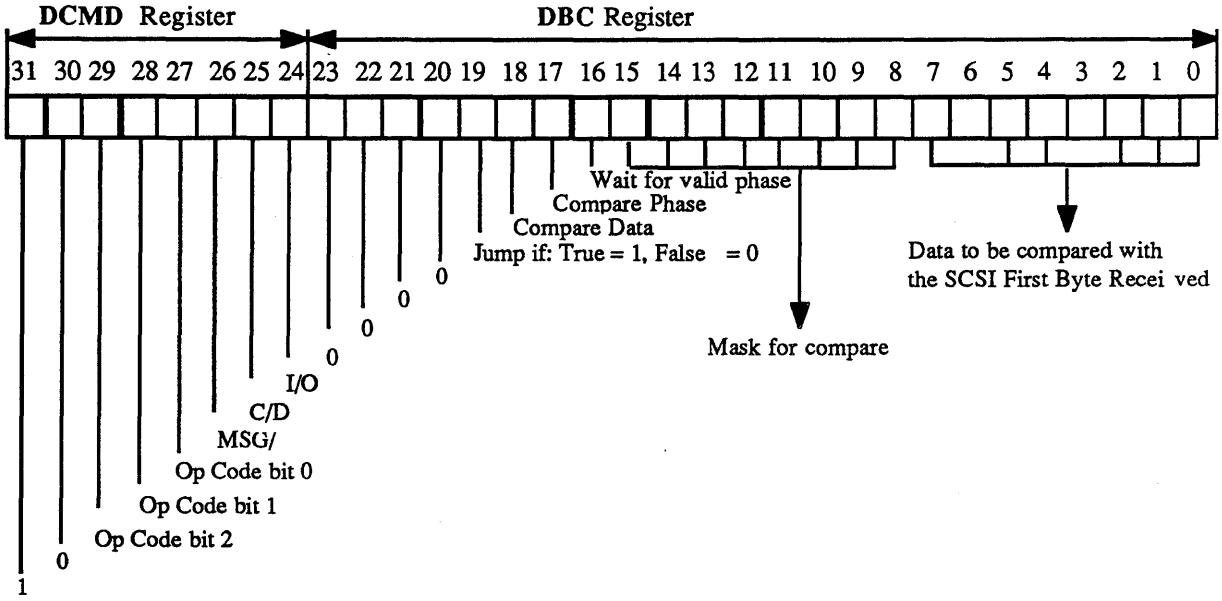
These bits are used during the set or clear command. Bit 10, on places the chip in the target/initiator role. Bit 6, on sets/resets the SCSI acknowledge. Bit 3, on sets/resets the SCSI attention. Use the clear ACK command after the last target message-in byte has been verified for each separate message 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. Clear attention should be issued after the target has serviced the request for a message out by the initiator.

I/O Command Second SCRIPTS Word

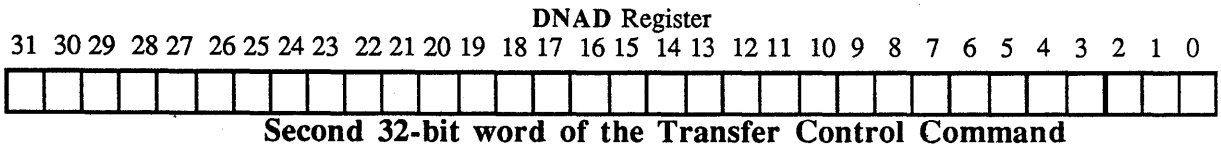
Jump Address - Bit 31-00

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 3. Transfer Control Command

Overview

The Transfer Control Command contains the JUMP, CALL, RETURN, and INTERRUPT operation codes. Each opcode is conditionally performed based on compares of SCSI phase values and incoming SCIS 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 53C700 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.

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 firmware interrupt service routine overhead.

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.

Bits 23-20

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. The value is ignored 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 SCSI SCRIPTS patch area.

Chapter 3
Developing NCR SCSI SCRIPTS™

NCR Microelectronics has a 53C700 Software Development kit which supports the development of SCSI SCRIPTS. This kit includes:

- Sample SCRIPTS
- SCRIPTS Utilities
- Test/Diagnostic SCRIPTS
- A SCRIPTS compiler
- Hardware Test Support

Your local NCR Sales Office or Factory Representative will assist you with ordering information and current board level options.

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 53C700. The compiler output is like an object module, it includes relocation information required to load the SCRIPTS object module into main memory.

At load time, the SCRIPTS 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., etc.

Writing a logical I/O driver is an easy task for the 53C700. This is illustrated in the first SCSI SCRIPTS example. This code will perform a read or write function using the 53C700 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 on more complicated, realistic algorithms.

A SCSI SCRIPTS is comprised of two parts or areas:

1) Definition area
2) SCRIPT area

In this 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. to be fixed at Start I/O time.
EXTERNAL device

;       Ten byte buffer for sending messages
EXTERNAL sendmsg

;       Ten byte buffer for receiving messages
EXTERNAL rcvmsg
```

```
;       Number of message bytes to send after selection
EXTERNAL idcount

;       Number of command bytes
EXTERNAL cmd_count

;       Buffer for the SCSI command
EXTERNAL cmd_adr

;       Number of user data bytes
EXTERNAL data_count

;       Address of user data buffer
EXTERNAL data_adr
;       Error -- not message out after selection


;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*  Absolute values are stored in DNAD 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 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
```

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 status value is DNAD stored in the 53C700's register and is available for interrupt processing.

```
; select the device with attention on
select atn device 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 idcount sendmsg when MSG_OUT

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

; send the command bytes
move cmd_count cmd_adr when CMD

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

; process data in phase
jump input_data if DATA_IN

; or data out phase
jump output_data if DATA_OUT

; unexpected phase if here
int err3

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

; and go process status
jump end

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

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

```
; move the status byte into memory
move 1 status_adr when STATUS

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

; move the command complete byte in
move 1 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
```

Chapter 4
NCR SCSI SCRIPTS ™ Utilities

The development package includes these utilities and the SCSI SCRIPTS Compiler.

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 struct to another. This is ANSI compatible, but older compilers may not support it. Therefore by defined KR to 1 each member will be assigned in turn.

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 SIOP 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 SIOP 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 SIOP drive the INT/ signal when an abort condition is encountered and it is set to ON.

SetINTInstInt()

Declaration:
void SetINTInstInt(BOOL)

SetINTInstInt() allows the SIOP 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 SIOP 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 SIOP 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 SIOP 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 SIOP. 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 SIOP cannot talk to itself.

SetParity()

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

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

SetAutoATN()

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

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

SetSlowBus()

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

SetSlowBus(), the SIOP 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 5
The NCR SCSI SCRIPTS™ Language Syntax

Notation

{ } Something enclosed in curly braces is optional.

{ }"... " The item enclosed in the curly braces can be repeated as often as desired.

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 first pass of 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 second pass 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 calls 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.

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

The 53C700 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
```

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.

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.

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
```

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.

ie. ...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.

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
```

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.

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.

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

SCSI I/O Commands

SELECT {ATN} ID, Address

Initiator mode function 0.
If ATN is present, the "select with ATN" bit is turned on. 'id' specifies the destination SCSI ID.

RESELECT id, address

Target mode function 0

WAIT DISCONNECT

Initiator mode function 1

DISCONNECT

Target mode function 1

WAIT RESELECT address

Initiator mode function 2

WAIT SELECT address

Target mode function 2
If the 53C700 is connected as an initiator, the following set and clear commands will have no meaning (the SCSI target role is not active) and should not be used.

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.

Chapter 6
SCSI SCRIPTS ™ to Support Use of Scatter/Gather

Virtual memory schemes are common in todays 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, so 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 setup 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.

The 53C700 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 (500 nanoseconds). No firmware interrupt is required (normally a ***minimum*** of 80 microseconds in a system environment). Nor is a firmware instruction required to re-instruct a DMA controller.

Chapter 7 contains a 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. 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 patches 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.

The 53C700 can process scatter/gather requests in a very simple manner and simultaneously, dramatically reduce I/O overhead.

Chapter 7

NCR SCSI SCRIPTS™ for an Initiator

```
;       Definition area INITIATOR ROLE

;       Target Device I.D. to be fixed at Start I/O time.
EXTERNAL device

;       Ten byte buffer for sending messages
EXTERNAL sendmsg

;       Ten byte buffer for receiving messages
EXTERNAL rcvmsg

;       Number of message bytes to send after selection
EXTERNAL idcount

;       Number of command bytes
EXTERNAL cmd_count

;       Buffer for the SCSI command
EXTERNAL cmd_adr

;       Number of user data bytes
EXTERNAL data_count

;       Address of user data buffer
EXTERNAL data_adr

;       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
;       53C700 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 device resel_adr

;       If the next phase is status, go to end.  Wait for valid
;       phase before performing the comparison.
JUMP 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 idcount 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 retry WHEN MSG_OUT

;       Now check for all expected phases.
JUMP end IF STATUS

;       Process a message in before the command phase here
JUMP 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 cmd_count cmd_adr when CMD

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

;       Status phase after the command?
JUMP end IF STATUS

;       Check for data in phase
JUMP input_data IF DATA_IN

;       Is this a data out phase?
JUMP 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 1 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 1 rcvmsg when MSG_IN

;       Process the message if it is not a command complete
INT 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 data_count 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 end WHEN STATUS

;       Is there a message in phase after data transfer
JUMP 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 data_count data_adr when DATA_IN

;       Go check the phase after data in
JUMP check_it
```

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

MOVE 1 rcvmsg when MSG_IN

;       Is this an extended message?
JUMP ext_msg1 IF 0x01

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

;       Is this a disconnect?
JUMP disc1 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 2 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 1 rcvmsg when MSG_IN

;       Is this an extended message?
JUMP ext_msg2 IF 0x01

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

;       Is this a disconnect?
JUMP 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 2 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 1 rcvmsg when MSG_IN

;       Is this an extended message?
JUMP ext_msg3 IF 0x01

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

;       Is this a disconnect?
JUMP disc3 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 2 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 1 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 *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

;       Buffer area where the initiator device i.d. is kept
EXTERNAL device

;       Message out buffer area
EXTERNAL msg_buf

;       Command byte buffer area
EXTERNAL cmd_buf

;       Input message buffer
EXTERNAL msg_buf2

;       Buffer address for the initiator i.d.
EXTERNAL initiator

;       Count of user data bytes to be moved
EXTERNAL data_count

;       Address of the user data buffer
EXTERNAL data_addr
;       Target got reselected

;       Address of the status buffer
EXTERNAL stat_adr
```

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;* 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 resel_adr

;       Move the i.d. message into the message buffer
retry_id:
MOVE 1 msg_buf WITH MSG_OUT

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

continue_id:

;       Move the command bytes in to the target buffer
MOVE 1 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 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 1 msg_buf2 WITH MSG_IN

;       If the initiator sets ATN, go process that condition
JUMP 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 initiator resel_adr

;       Move the reselect i.d. message into the initiator
retry_resel:
MOVE 1 msg_buf2 WITH MSG_IN

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

continue_resel:

;       Now move the data bytes into the initiator
MOVE data_count 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 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 1 stat_adr WITH STATUS

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

continue_stat:

;       Move the command complete message in
MOVE 1 msg_buf2 WITH MSG_IN

;       If the initiator sets ATN, go process that condition
JUMP 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 1 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 1 msg_buf WITH MSG_OUT
INT error3


disc_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error4


resel_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error5


data_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error6


stat_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error7


cc_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error8
```

Chapter 8
Unique Initiator Sequences for the 53C700

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

53C700 strengths in the disk drive environment

- A large number of commands are typically issued to the disk, and the 53C700 offers very little SCSI bus overhead and a minimum of time to initiate an I/O in the host computer.
- The 53C700 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 53C700 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 53C700 can be programmed to absorb this message with no interrupt to the external processor.

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 53C700 could be programmed specifically for this system. Knowing the tape drive was alone on the bus, the 53C700 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 53C700 interrupts to allow the external processor to restart the tape I/O.

The 53C700 allows systems integration designs using the SCSI bus with no performance impact to I/O throughput.

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 53C700 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 53C700 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 53C700 can be used in unusual areas to offload any processor and improve performance.

Another implementation of the 53C700 is SCSI printer design, where WRITE is the only operation and control characters also play in important role.

Chapter 9
Special Scripts™ Situations

SAVE DATA POINTERS message that can be ignored.

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, as well as anomalies such as an unexpected phase change after transferring 2K of the data.

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

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

The routine described in steps 1-3 will receive the message byte, verify that the message byte is a SAVE DATA POINTERS (if not, interrupt the external processor), 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 nsec 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.

SAVE DATA POINTERS message that must 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 53C700 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 53C700. 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 53C700 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 53C700 will fetch the next command and start processing it. When the phase change actually occurs, the 53C700 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.

Chapter 10
Multi-Tasking I/O Using SCSI SCRIPTS™

To accommodate multi-tasking I/O entirely within SCSI SCRIPTS, some special techniques are required. A standard SCSI SCRIPTS algorithm (the I/O descriptor) must be developed for each concurrent I/O. I/O's can be linked together by making the last SCSI SCRIPTS command of each scheduled I/O descriptor a jump to the next scheduled I/O descriptor. This last command address is effectively a mailbox for communication between the host computer and the 53C700. The external processor can patch the last command to a jump command if the next I/O descriptor has been scheduled by the logical I/O, or patch an interrupt command if it has not been scheduled. The 53C700 will fetch a complete SCSI SCRIPTS (all 8 bytes), blocking out the processor. The iAPX 286/386 can write 4 bytes, blocking out the 53C700. The patch must be to the four byte opcode to allow a test/set capability. Thus, the second four bytes must be the SCSI SCRIPTS jump address and the interrupt value. All of the SCSI SCRIPTS algorithms are arranged in memory in a linked list. To schedule an I/O, the host processor must:

- Find the address of the first open I/O descriptor (SCSI SCRIPTS program).
- Update the variable addresses of user and SCSI data within the I/O descriptor.
- Change the last command of the I/O descriptor (currently an interrupt command) to an interrupt with the I/O descriptor I.D. as the last four bytes. At interrupt time, this information allows a fast identification of the I/O just completed (when no other I/O is scheduled).
- Change the last command of the previous I/O descriptor (currently functioning as an interrupt command) to a jump to the beginning of the newly scheduled I/O descriptor.

A series of I/O's can be scheduled by the host processor. When an I/O completes, if there is no other scheduled I/O, then the 53C700 interrupts the command register using the I/O descriptor I.D.; otherwise it jumps to the next scheduled I/O. The processor knows that an I/O has completed when:

- An interrupt occurs for a given I/O because of an error.
- The address of the SCSI SCRIPTS command being executed is outside the address space of the I/O descriptor being tested for completion. Wrap around must be considered in this test.

  The host processor can trigger this test through a timer interrupt or a polling scheme.
- The SCSI status byte is written into a known address to flag that the I/O is complete. The host processor can use a timer interrupt or a polling scheme.
- Some types of hardware assists generate an interrupt at the completion of the I/O. The interrupt occurs when SCSI status byte being written into a main memory address interrupts the host processor.

When an I/O is completed, the last SCSI SCRIPTS command of the I/O descriptor must be changed to an interrupt command to initialize it for the next I/O to be scheduled (the I.D. value is set to an invalid value). The I/O driver must take care that an infinite loop is NOT established with the SCSI SCRIPTS jump command. This simple software mechanism allows the 53C700 to schedule I/O requests without a sophisticated host bus adapter.

Using SCSI SCRIPTS to Implement Multi-Threaded I/O

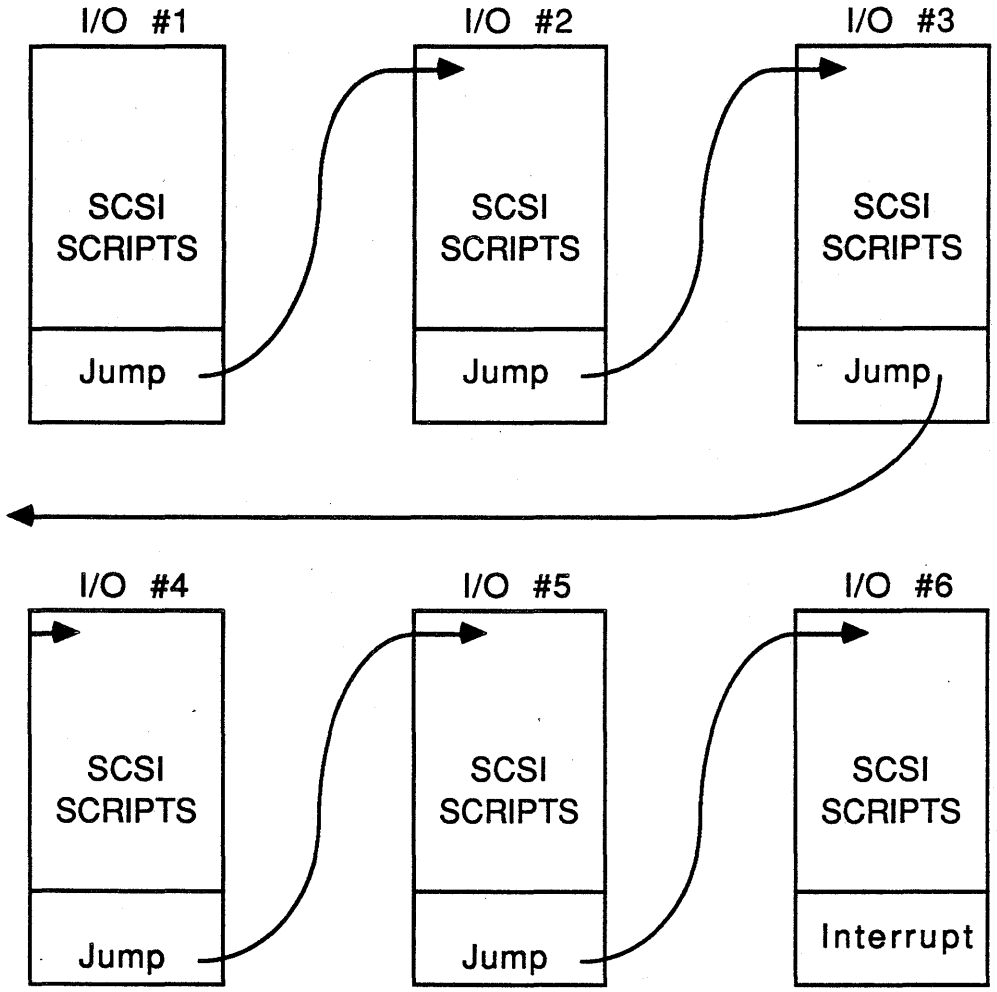


Figure 4. Using SCSI SCRIPTS to Implement Multi-Threaded I/O

In this example, all six I/O's have been scheduled by patching the last SCSI SCRIPT in each I/O descriptor to jump to the next scheduled I/O descriptor. When each I/O is complete, the linked list is broken by patching out the jump instruction to the next I/O descriptor.

Appendix A
High Performance Considerations for 53C700 vs 53C90

This chapter compares firmware required for the 53C700 and the 53C90 to determine how much of a performance boost the 53C700 can offer at a system level (I/O's 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 H/W driver level when performing an I/O.

I.D. message byte count
Input message buffer address
Output message buffer address
SCSI command byte count
SCSI command buffer address
User data byte count 1
User data buffer address 1
*
*
User data byte count 'n'
User data buffer address 'n'
SCSI status buffer address
Command Complete message address

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

53C700 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, 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, fifteen values must be updated. Basically, all the Block Move commands must be altered. The firmware sequence requires:

Load Address Of Data/Count In Main Memory
Load Value Desired From Main Memory
Load SCSI SCRIPTSTM Offset
Store Value in SCSI SCRIPTS™ Offset

	Approx time in µs
Assuming that the sequence above takes about four microseconds.	
the total time is	60
Executing the initiator algorithm takes about 30 SCSI SCRIPTS fetches and decodes.	
the total overhead is	15
If disconnect from the target after transferring the SCSI command, there will be two interrupts to the host processor. Each interrupt takes about 80 micro seconds.	
the total time is	160

60 + 15 + 160 = 235 µsec overhead.

Note that the SCSI portion of the interrupt service routine is only two or three lines of code because both interrupting situations are controlled by SCSI SCRIPTS, requiring only a read of the interrupt code.

53C90 Algorithm Description

The firmware begins the sequence by preloading the 53C90 FIFO with the SCSI I.D. message followed by a ten byte SCSI command. The firmware sequence involved requires:

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

For eleven 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 53C700 requires about 25% 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 only one interrupt using the 53C700 (80 - 90 µsec). Each data segment in a scatter/gather situation requires 125 µsec on the 53C90 (one interrupt plus DMA initialize) but only .5 µsec on the 53C700 (500 nanosecond instruction fetch). So, an I/O that required four data segments in a scatter/gather mode would require 500 µsec on the 53C90 and 2 µsec on the 53C700 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

53C700
237 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	53C700 in msec
Data Transfer Time	1.00	1.00
Target overhead	1.00	1.00
Host Overhead	1.25	0.25
Total times	3.25	2.25
I/O's Per Second	**307**	**444**

In this projected environment, a system can increase its throughput rate by fifty percent by using the 53C700 and reducing host computer firmware overhead. With the types of buffered SCSI disk drives currently available, the 53C700 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 53C700, to increase system level performance, designers must eliminate each delay. The 53C700 can remove much of the host overhead associated with each I/O.

Appendix B
53C700 System Bus Utilization

The 53C700, 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 53C700, because no firmware intervention is required.

A real concern is host bus utilization, or "Does the 53C700 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, msg_buf, when MSG_OUT
jump error, when not CMD phase
move, cmd_buf, when CMD
jump error, when not DATA_IN
move, data_buf, when DATA_IN
jump error, when not STATUS
move, status_buf, when STATUS
jump error, when not MSG_IN
move, 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.

14 X 280 = 3.92 µsec

6,666 X 3.92 = 26.13 µsec
(total fetch time per second)

The fetch time is **2.6%** 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% + 2.6% = 3.24% 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 increased to about 8%.

Conclusion

So the total time to saturate the SCSI bus takes 11.2% of the iAPX 286/386 system 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. For example, a 1K block implies 250 µsec per I/O (50 µsec SCSI overhead as measured in a lab environment and 200 µsec for user data). This is 4000 I/O's per second or 4 Mbytes per second. The total 386 bus overhead is reduced to about 1.95% of the available time (4000/6666 X 3.24%). As the block size increases, the overhead decreases.

Appendix C
SCSI SCRIPTS™ Compiler

The SCSI SCRIPTS Compiler is a two pass compiler.

The first pass compiler creates an output file in generic format from a SCRIPTS source file. This may be included in the user's source code for any language.

The second pass compiler takes the pass1 file and creates a C file which may be used in any C program.

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

SCSI SCRIPTS Compiler Pass 1 (ss.exe)

A SCRIPTS source file may be created using any standard text editor that can create an ASCII file output. In the example below, the SCRIPT source file is SCRIPT.ss. To compile SCRIPT.ss use the following format:

ss<input SCRIPT.ss> <output SCRIPT FILE> <options>

Example:

```
ss SCRIPT.ss  SCRIPT.ps1 /s
```

This creates two output files:

SCRIPT.ps1	compiled output of the SCRIPT source file
SCRIPT.xrf	cross reference listing.

If there are syntax errors in the SCRIPTS source file, the line number and error message are displayed on the screen. The error messages can be saved into a file by adding the /s option to your compile command. If there were errors in this example, the file name generated would be SCRIPT.ERR. Appendix D lists the compiler error messages.

Output File Description

The pass1 compiled output, SCRIPT.ps1 is in a generic format . The SCRIPT instruction list is always generated first. The first instruction column contains the long word instruction. The second column contains the corresponding long word address.

Example:

```
INSTRUCTIONS
    90008000        0000000A
    90008017        00000023
```

The next list contains a relative or absolute variable followed by its value. The next line contains the long word offsets in the SCRIPT where the variable is used.

Example:

```
R_data_buf        00000020
                  0000000d
```

The variable entry is followed by the SCRIPT entry label values.

Example:

```
Ent_alt_adr = 00000078
```

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

Example:

```
LABELPATCHES
        00000001
        00000019
        0000001b
```

The last item is a count of the number of instructions and patches in the SCRIPT.

Example:

```
INSTRUCTIONS      00000011
PATCHES           00000003
```

SCRIPT.xrf File Description (SCRIPT cross-reference file)

For every instruction, the cross reference file (.xrf) lists
- an offset from the beginning of the script,
- the long word instruction,
- the long word address, and
- the corresponding source SCRIPT instruction.

Labels appear on a line by themselves as they are encountered in the SCRIPT. After the cross reference is a list consisting of the absolute or relative variable, the variable name and 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 last information 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 program runtime. Use the Patch Utilities described in Section 4 of the NCR 53C700 Programmer's Guide.

SCSI SCRIPTS Compiler Pass2 (pass2.exe)

Pass2 creates a C format file from the pass1 compiler output. If the pass1 output file is SCRIPT.ps1, then the format for using Pass2 will be as follows.

```
pass2 <pass1 file> <Cfile>
```

Example:

```
pass2 SCRIPT.ps1 SCRIPT.ps2
```

When you run the compile, the following message will be displayed on your screen.

```
SCSI Scripts (TM) Compiler Pass2,
V0.01Beta
Copyright (C) 19989 NCR

Requires 0x3e7 bytes of memory.
```

The memory requirement is the number of bytes in the SCRIPT.ps1 file. Without this amount of memory, the compiler will not execute properly.

Pass2 reads in clusters of characters called tokens until a separator is encountered, such as space, tab, line feed, or end of file. Tokens can be identifiers, variables, or numbers. When an identifier is encountered, an array of unsigned long elements is generated in the C output file. First, a value is defined for a variable, then an array of unsigned long elements is generated. These elements indicate where in the SCRIPT the variable is used. Numeric values are given an "0x" prefix for hexadecimal numbers in C format.

The SCRIPT instructions array is always generated first. The first column contains the long word instruction and the second the corresponding long word address.

Example:

```
ULONG SCRIPT[ ] = {
        0x90008000, 0x0000000A,
        0x90008017, 0x00000023
};
```

The variable name prefix will have an "A_" for absolute or an "R_ "for relative. This is followed by the variable value. 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:

```
#defineR_DATA_BUF 0X00000020
ULONG R_data_buf_Used[ ] = {
};
```

Next define the SCRIPT entry label values that are prefixed with an "Ent_".

Example:

```
#define Ent_alt_adr 0x00000078
```

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

Example:

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

Next is the last item, the number of instructions and patches in the SCRIPT.

Example:

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

Since an error file was generated with the Pass1 compile, Pass2 does little error checking. Do not give Pass2 a file in a different format (i.e. other than Pass1).

Appendix D
Compiler Script Examples

SCSI SCRIPTS Compiler Pass1 SCRIPT Scource File (sample.ss)

```
RELATIVE msg_buf=0, cmd_buf=msg_buf+1, stat_buf=cmd_buf+10
RELATIVE msg_in_buf=stat_buf+1, data_buf = msg_buf+32

ENTRY  alt_adr, alt2, s_write


;INITIATOR WRITE SCRIPT

s_write:
        select ATN 01, alt_adr                  ; select the target

        int 6, when not MSG_OUT                 ; check for message out phase next
        move 1, msg_buf, when MSG_OUT           ; move the message byte in

        int 7, when not CMD                     ; check for command phase next
        move 6, cmd_buf, when CMD               ; move the command bytes out next

        int 75, when not DATA_OUT               ; interrupt if not data out phase
        move 512, data_buf, when DATA_OUT       ; move data bytes

        int 8, when not STATUS                  ; check for status phase next
        move 1, stat_buf, when STATUS           ; move the status byte in

        int 9, when not MSG_IN                  ; check for message in phase
        move 1, msg_in_buf, when MSG_IN         ; move the command complete message in
        int 95, if not 00                       ; interrupt if not a command complete message
        clear ACK alt2                          ; accept the message byte

        wait disconnect alt2                    ; wait for the bus free interrupt

        int 10                                  ; interrupt when the command is completed

; end Initiator Write SCRIPT


alt_adr:
        int 11      ;interrupt if jump to alt_adr during selection

alt2:
        int 12      ;interrupt if jump to alt2

;end of SCRIPT
```

SCSI SCRIPTS Compiler Pass 1
Pass1 Compiler Output (smple.ps1)

```
INSTRUCTIONS
        41010000        00000078
        9E030000        00000006
        06000001        00000000
        9A030000        00000007
        02000006        00000001
        98030000        0000004B
        00000200        00000020
        9B030000        00000008
        03000001        0000000B
        9F030000        00000009
        07000001        0000000C
        98040000        0000005F
        60000040        00000080
        48000000        00000080
        98080000        0000000A
        98080000        0000000B
        98080000        0000000C

R_msg_buf       00000000
        00000005

R_cmd_buf       00000001
        00000009

R_stat_buf      0000000B
        00000011

R_msg_in_buf    0000000C
        00000015

R_data_buf      00000020
        0000000d

Ent_alt_adr     00000078
Ent_alt2        00000080
Ent_s_write     00000000
LABEL PATCHES
        00000001
        00000019
        0000001b

COUNTS
        INSTRUCTIONS            00000011
        PATCHES                 00000003
```

SCSI SCRIPTS Compiler Pass2

Pass2 Compiler Output (sample.ps2)

```
typedef unsigned long ULONG;

ULONG           SCRIPT[] = {
        0x41010000,     0x00000078,
        0x9E030000,     0x00000006,
        0x06000001,     0x00000000,
        0x9A030000,     0x00000007,
        0x02000006,     0x00000001,
        0x98030000,     0x0000004B,
        0x00000200,     0x00000020,
        0x9B030000,     0x00000008,
        0x03000001,     0x0000000B,
        0x9F030000,     0x00000009,
        0x07000001,     0x0000000C,
        0x98040000,     0x0000005F,
        0x60000040,     0x00000080,
        0x48000000,     0x00000080,
        0x98080000,     0x0000000A,
        0x98080000,     0x0000000B,
        0x98080000,     0x0000000C
};

#define  R_msg_buf      0x00000000
ULONG           R_msg_buf_Used[] = {
        0x00000005
};

#define  R_cmd_buf      0x00000001
ULONG           R_cmd_buf_Used[] = {
        0x00000009
};

#define  R_stat_buf     0x0000000B
ULONG           R_stat_buf_Used[] = {
        0x00000011
};

#define  R_msg_in_buf   0x0000000C
ULONG           R_msg_in_buf_Used[] = {
        0x00000015
};

#define  R_data_buf     0x00000020
ULONG           R_data_buf_Used[] = {
        0x0000000d
};

#define  Ent_alt_adr    0x00000078
#define  Ent_alt2       0x00000080
#define  Ent_s_write    0x00000000
ULONG           LABELPATCHES[] = {
        0x00000001,     0x00000019,
        0x0000001b
};
ULONG           INSTRUCTIONS            = 0x00000011;
ULONG           PATCHES                 = 0x00000003;
```

SCSI SCRIPTS Cross Reference File Listing (sample.xrf)

SCRIPT FILE: sample.ps1

```
s_write:
0000 41010000 00000078   select ATN 01, alt_adr                ; select the target
0008 9E030000 00000006   int 6, when not MSG_OUT               ; check for message out phase next
0010 06000001 00000000   move 1, msg_buf, when MSG_OUT         ; move the message byte in
0018 9A030000 00000007   int 7, when not CMD                   ; check for command phase next
0020 02000006 00000001   move 6, cmd_buf, when CMD             ; move the command bytes out next
0028 98030000 0000004B   int 75, when not DATA_OUT             ; interrupt if not data out phase
0030 00000200 00000020   move 512, data_buf, when DATA_OUT                ; move data bytes
0038 9B030000 00000008   int 8, when not STATUS                ; check for status phase next
0040 03000001 0000000B   move 1, stat_buf, when STATUS         ; move the status byte in
0048 9F030000 00000009   int 9, when not MSG_IN                ; check for message in phase
0050 07000001 0000000C   move 1, msg_in_buf, when MSG_IN       ; move the command complete message in
0058 98040000 0000005F   int 95, if not 00                     ; interrupt if not a command complete message
0060 60000040 00000080   clear ACK alt2                        ; accept the message byte
0068 48000000 00000080   wait disconnect alt2                  ; wait for the bus free interrupt
0070 98080000 0000000A   int 10                                ; interrupt when the command is completed
alt_adr:
0078 98080000 0000000B   int 11                                ;interrupt if jump to alt_adr during selection
alt2:
0080 98080000 0000000C   int 12                                ;interrupt if jump to alt2


RELATIVE msg_buf = 00000000
        00000010

RELATIVE cmd_buf = 00000001
        00000020

RELATIVE stat_buf = 0000000b
        00000040

RELATIVE msg_in_buf = 0000000c
        00000050

RELATIVE data_buf = 00000020
        00000030

ENTRY LABEL: alt_adr at address 00000078

ENTRY LABEL: alt2 at address 00000080

ENTRY LABEL: s_write at address 00000000

LABEL PATCH at      00000001
LABEL PATCH at      00000019
LABEL PATCH at      0000001b
```

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 phenomenona known as cosmic ray bombardment. This is believed to be associated with increased solar flare activity. Fortunately, the effects are not permanent, try again.

NCR Microelectronic Products Division – Sales Locations

For literature on any NCR Microelectronics product or service call the NCR hotline toll-free:

1-800-334-5454

NCR Microelectronic Products Division
Worldwide Sales Headquarters
3130 De La Cruz Boulevard, Suite 209
Santa Clara, CA 95054
(408) 980-6200

Division Plants

NCR Microelectronic Products Division
2001 Danfield Court
Fort Collins, CO 80525
(303) 226-9500

Commercial ASIC Products
Customer Owned Tooling
Communications Products
Memory Products
Software Products

NCR Microelectronic Products Division
1635 Aeroplaza Drive
Colorado Springs, CO 80916
(719) 596-5611
1-800-525-2252

High Reliability ASIC
Military Products
Automotive Products
Logic Products
SCSI Products
Internal ASIC

North American Sales Offices

Northwest Sales
3130 De La Cruz Boulevard, Suite 209
Santa Clara, CA 95054
(408) 727-6575

Southwest Sales
3300 Irvine Avenue, Suite 255
Newport Beach, CA 92660
(714) 474-7095

1940 Century Park East
Los Angeles, CA 90067
(213) 556-5231

North Central Sales
8000 Townline Avenue, Suite 209
Bloomington, MN 55438
(612) 941-7075
(612) 941-6340

South Central Sales
400 Chisholm Place, Suite 100
Plano, TX 75075
(214) 578-9113

Northeast Sales
500 West Cummings Parkway, Suite 4000
Woburn, MA 01801
(617) 933-0778

Southeast Sales
700 Old Roswell Lakes Parkway, Suite 250
Roswell, GA 30036
(404) 587-3136

International Sales Offices

Europe
Gustav-Heinemann-Ring 133
8000 Munchen 83
West Germany
49 89 632202

Asia/Pacific
2501 Vicwood Plaza
199 Des Voeux Road
Central Hong Kong
852 5 859 6888