

# **NCR 53C720 SCSI I/O Processor**



**Programmer's Guide**

**Copyright © 1991  
By NCR Corporation,  
Dayton, Ohio, U.S.A.  
All Rights Reserved,  
Printed in U.S.A.**

**This information has been checked for both accuracy and reliability. NCR assumes no responsibility for either its use or any damages resulting from its use. NCR reserves the right to make any changes or discontinue altogether without notice, any hardware or software product, or the technical content herein.**

## Preface

This manual is a programming guide for the NCR 53C720 SCSI I/O Processor chip. The 53C720 SCSI I/O Processor can be programmed using either high level NCR SCSI SCRIPTS™ or using low level register interface. This manual contains a syntax level description of the NCR SCSI SCRIPTS instructions for high level programming and a bit level instruction description for those who require more details for debugging. Examples of SCSI SCRIPTS and how to use features are provided. To successfully use this manual the programmer should be familiar with the 'C' programming language.

## Trademarks

NCR is the name and mark of NCR Corporation.  
SCSI SCRIPTS™ is a registered trademark of NCR Corporation

## Additional Information

NCR 53C720 SCSI I/O Processor Data Manual  
NCR SCSI Engineering Note 831, Comparison of 53C720 to 53C710

NCR SCSI SCRIPTS Examples  
Logic Products Electronic Bulletin Board  
(719) 596 - 1649  
Communications parameters:  
300/1200/2400 baud, 8 data bits, no parity, 1 stop bit

## SCSI Specifications

### ANSI

1430 Broadway  
New York, NY 10018  
(212) 642-4900  
Ask for document number; X3.131-1986 (SCSI-1)

### Global Engineering Documents

2805 McGaw  
Irving, CA 92714  
(800) 854-7179 or (714) 261-1455  
Ask for document number: x3.131-198X (SCSI-2)

### ENDL Publications

14426 Black Walnut Curt  
Saratoga, CA 95070  
(408) 867-6642  
Document name: SCSI Bench Reference

### Prentice Hall

Englewood Cliffs, New Jersey 07632  
(201) 767-5937  
Ask for document number: ISBN 0-13-796855-8  
Document name: SCSI – Understanding the Small Computer System Interface



## Revision Record

Revision	Date	Affected Pages/Remarks
	June 91	Preliminary – First Printing



# Contents

## **Chapter 1** **The NCR SCSI SCRIPTS**

A SCSI Solution .....	1-1
NCR 53C720 SCSI I/O Processor Chip Block Diagram .....	1-2
The NCR SCSI I/O Processor .....	1-3
DMA Component .....	1-3
SCSI SCRIPTS™ Processor .....	1-3
NCR SCSI SCRIPTS™ Description .....	1-4
How SCSI SCRIPTS becomes part of a C Language Program .....	1-5
Example of a SCRIPTS Operation .....	1-6

## **Chapter 2** **Developing NCR SCSI SCRIPTS**

Single-Tasking SCSI Example .....	2-4
-----------------------------------	-----

## **Chapter 3** **SCSI SCRIPTS Compiler**

Invoking the SCSI SCRIPTS Compiler .....	3-1
SCSI SCRIPTS Compiler Output .....	3-2

## **Chapter 4** **The NCR SCSI SCRIPTS Language Syntax**

SCSI SCRIPTS .....	4-1
SCRIPTS Keywords .....	4-1
Instruction Keywords .....	4-1
Phase Keywords .....	4-2
Register Keywords .....	4-2
Miscellaneous Keywords .....	4-3
Compiler Directives .....	4-3
SCRIPTS Notation .....	4-4
Compiler Directives Syntax .....	4-5
Instruction Keywords .....	4-7
MOVE Instructions .....	4-7
1. Block Move .....	4-7
Direct Chained Block Move .....	4-7
Indirect Chained Block Move .....	4-7
Table Indirect Chained Block Move .....	4-8

2. Chained Block Move .....	4-8
Direct Chained Block Move .....	4-9
Indirect Chained Block Move.....	4-9
Table Indirect Chained Block Move .....	4-9
3. Memory -to-Memory Move .....	4-10
JUMP Instruction .....	4-11
CALL Instruction .....	4-12
RETURN Instruction.....	4-14
Interrupt Instruction .....	4-15
Interrupt on the Fly Instruction .....	4-16
Miscellaneous Instructions.....	4-17
Select Instruction .....	4-17
Reselect Instruction .....	4-17
Wait Disconnect Instruction.....	4-17
Disconnect Instruction.....	4-17
Wait Reselect Instruction .....	4-17
Wait Select Instruction.....	4-18
Set Instructions .....	4-18
Clear Instructions .....	4-18
Register Read/Write Instruction .....	4-19

**Chapter 5**  
**SCSI SCRIPTS Use of Scatter/Gatter**

**Chapter 6**  
**CSI SCRIPTS for an Initiator and Target**

SCRIPTS for the Initiator Role .....	6-1
SCRIPTS for the Target Role.....	6-7

**Chapter 7**  
**Unique Initiator Sequences for the 53C720**

Disk Drive Initiator Sequence .....	7-1
53C720 Strengths in the Disk Drive Environment .....	7-1
Tape Drive Initiator Sequence.....	7-1
SCSI Character Oriented Device in the Initiator Role .....	7-2

**Chapter 8**  
**Special SCSI SCRIPTS Situations**

Transferring Large Blocks of User Data .....	8-1
Case 1 .....	8-1
Case 2 .....	8-1
Case 3 .....	8-1



How a SAVE DATA POINTERS Can be Processed by the Initiator .....	8-2
Case 1 .....	8-2
Data-in Phase .....	8-2
Data-out Phase .....	8-2
Case 2 .....	8-2

## Chapter 9

### Multi-Tasking I/O Using SCSI SCRIPTS

Multi-Threaded I/O Using SCSI SCRIPT .....	9-1
Main SCSI SCRIPT .....	9-1
Scheduler SCSI SCRIPT .....	9-1
Disconnect SCSI SCRIPT .....	9-2
Resume SCSI SCRIPT .....	9-3

## Chapter 10

### SCSI SCRIPTS Machine Language Description

Block Move Overview .....	10-2
Block Move Instruction (First SCRIPTS Word) .....	10-2
Bits 31-30 Block Move (00) .....	10-2
Bit 29 Indirect data address flag .....	10-2
Bit 29 = 0 Direct Addressing .....	10-2
Bit 29 = 1 Indirect Addressing .....	10-2
Bit 28 Table Indirect Field .....	10-2
Bit 28 = 0 Table Direct Mode .....	10-2
Bit 28 = 1 Table Indirect Mode .....	10-2
Bit 27 Block Move Opcode .....	10-3
Target Mode Bit 27 = 0 (MOVE) .....	10-3
Target Mode Bit 27 = 1 (CHMOV) .....	10-4
Initiator Mode Bit 27 = 0 (CHMOV) .....	10-4
Initiator Mode Bit 27 = 1 (MOVE) .....	10-4
Bits 26-24 SCSI Phase Lines .....	10-4
Bits 23-0 Block Move Byte Count .....	10-4
Block Move Instruction (Second SCRIPTS Word) .....	10-5
Bits 31-0 Data Start Address .....	10-5
Chained Move Feature .....	10-5
I/O Instruction Overview .....	10-7
I/O Instruction (First SCRIPTS Word) .....	10-7
Bits 31-30 I/O Instruction (01) .....	10-7
Bits 29-27 I/O Instruction Opcodes .....	10-7
Target Mode Bits 29-27 = 000 (Reselect) .....	10-8

Target Mode Bits 29-27 = 001 (Disconnect) .....	10-8
Target Mode Bits 29-27 = 010 (Wait Select) .....	10-8
Target Mode Bits 29-27 = 011 (Set) .....	10-8
Target Mode Bits 29-27 = 100 (Reset) .....	10-8
Initiator Mode Bits 29-27 = 000 (Selection) .....	10-8
Initiator Mode Bits 29-27 = 001 (Wait Disconnect) .....	10-9
Initiator Mode Bits 29-27 = 010 (Wait Reselect) .....	10-9
Initiator Mode Bits 29-27 = 011 (Set) .....	10-9
Initiator Mode Bits 29-27 = 100 (Reset) .....	10-9
Bit 26 Relative Addressing Mode .....	10-10
Bit 25 Table Indirect Mode .....	10-10
Bit 24 SELECT With ATN .....	10-10
Bits 23-16 SCSI ID 7-0 .....	10-10
Bits 15-0 Flags Field .....	10-10
Read/Write Register Instructions (First SCRIPTS Word) .....	10-11
Bits 31-30 Read/Write Instruction (01) .....	10-11
Read/Write Overview .....	10-11
Bits 29-27 = 101 (Move from SFBR) .....	10-12
Bits 26-25 = 00 .....	10-12
Bits 26-25 = 01 .....	10-12
Bits 26-25 = 10 .....	10-12
Bits 26-25 = 11 .....	10-12
Bits 29-27 = 110 (Move to SFBR) .....	10-12
Bits 26-25 = 00 .....	10-12
Bits 26-25 = 01 .....	10-12
Bits 26-25 = 10 .....	10-12
Bits 26-25 = 11 .....	10-12
Bits 29-27 = 111 (Read-Modify-Write) .....	10-13
Bits 26-25 = 00 .....	10-13
Bits 26-25 = 01 .....	10-13
Bits 26-25 = 10 .....	10-13
Bits 26-25 = 11 .....	10-13
Bits 21-16 Register Address Field .....	10-13
Bits 15-8 Immediate Data Field .....	10-13
I/O Instruction (Second SCRIPTS Word) .....	10-13
Bit 31-0 Jump Address .....	10-13
Transfer Control Instructions (First SCRIPTS Word) .....	10-14
Transfer Control Overview .....	10-14
Bits 31-30 SCSI I/O Processor (10) .....	10-15
Bits 29-27 Transfer Opcodes .....	10-15
Bits 29-27 = 000 (JUMP) .....	10-15
Bits 29-27 = 001 (CALL) .....	10-15

Bits 29-27 = 010 (RETURN) .....	10-15
Bits 29-27 = 011 (Interrupt) .....	10-15
Bits 26-24 SCSI Phase Bits .....	10-15
Bit 23 Relative Addressing .....	10-16
Bits 22-20 Reserved Bits .....	10-16
Bits 19-16 Sequence Control Bits .....	10-16
Bit 19 .....	10-16
Bit 18 .....	10-16
Bit 17 .....	10-16
Bit 16 .....	10-16
Bits 15-8 Mask Bits .....	10-16
Bits 7-0 Data Byte .....	10-16
Transfer Control Instruction (Second SCRIPTS Word) .....	10-18
Bit 31-00 Data Jump Address .....	10-18
Memory-to-Memory Move Instructions (First SCRIPTS Word) .....	10-19
Memory Move Overview .....	10-19
Bits 31-30 SCSI I/O Processor Opcode .....	10-20
Bits 29-24 Reserved Section .....	10-20
Bits 23-00 24-bit Byte Count .....	10-20
Memory Move (Second SCRIPTS Word) .....	10-20
Bits 31-00 Source Address of the Memory Move .....	10-20
Memory Move (Third SCRIPTS Word) .....	10-21
Bits 31-00 Destination Address of the Memory Move .....	10-21

## **Appendix A**

### **53C720 Performance Compared to 53C90**

Sample Input Data Structure .....	A-1
Initializing SCSI SCRIPTS for an I/O and Starting I/O Operations .....	A-1
53C720 Algorithm Description .....	A-1
53C90 Algorithm Description .....	A-1
Conclusion .....	A-2

## **Appendix B**

### **53C720 System Bus Utilization**

Host Bus Time To Fetch A SCSI SCRIPTS Command .....	B-1
Conclusion .....	B-2

**Appendix C**  
**Use of the Sig\_p Bit in the 53C720**

**Appendix D**

**Compiler SCRIPTS Examples**

Sample SCSI SCRIPTS Scource File .....	D-1
Sample List File .....	D-3
Sample Output File .....	D-6

**Appendix E**  
**53C720 Test SCRIPTS Examples**

**Appendix F**  
**SCRIPTS™ Compiler Error Messages**

Fatal Errors .....	F-1
Errors .....	F-2
Warning .....	F-3

**Appendix G**  
**Miscellaneous Design Topics**

Design Topics .....	G-1
SCSI Timers .....	G-1
Longitudinal Parity Register (SLPAR) .....	G-1
Big/Little Endian Support .....	G-1
1. SCRIPTS Order .....	G-1
2. 53C720 Register Access from Firmware .....	G-1
3. 53C720 Register Access from SCSI SCRIPTS .....	G-1
4. User Data Byte Ordering .....	G-2
SCRIPTS in a Host Adapter .....	G-2

**Appendix H**  
**Using the 53C720 in Low Level Mode**

Low-level SCSI Code .....	H-1
---------------------------	-----

**Index**

# The NCR SCSI SCRIPTS

## A SCSI Solution

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

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

The flexibility of the SCSI bus creates a dilemma for system integrators and OEM's alike. The dilemma is: whether first and second generation SCSI devices should be used as non-intelligent, stand-alone devices or should 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. SCSI chips requiring intense firmware can overwork the controller microprocessor, making it unable to perform required tasks.

Limited space usually excludes adding an extra processor or replacing it with a more powerful one.

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

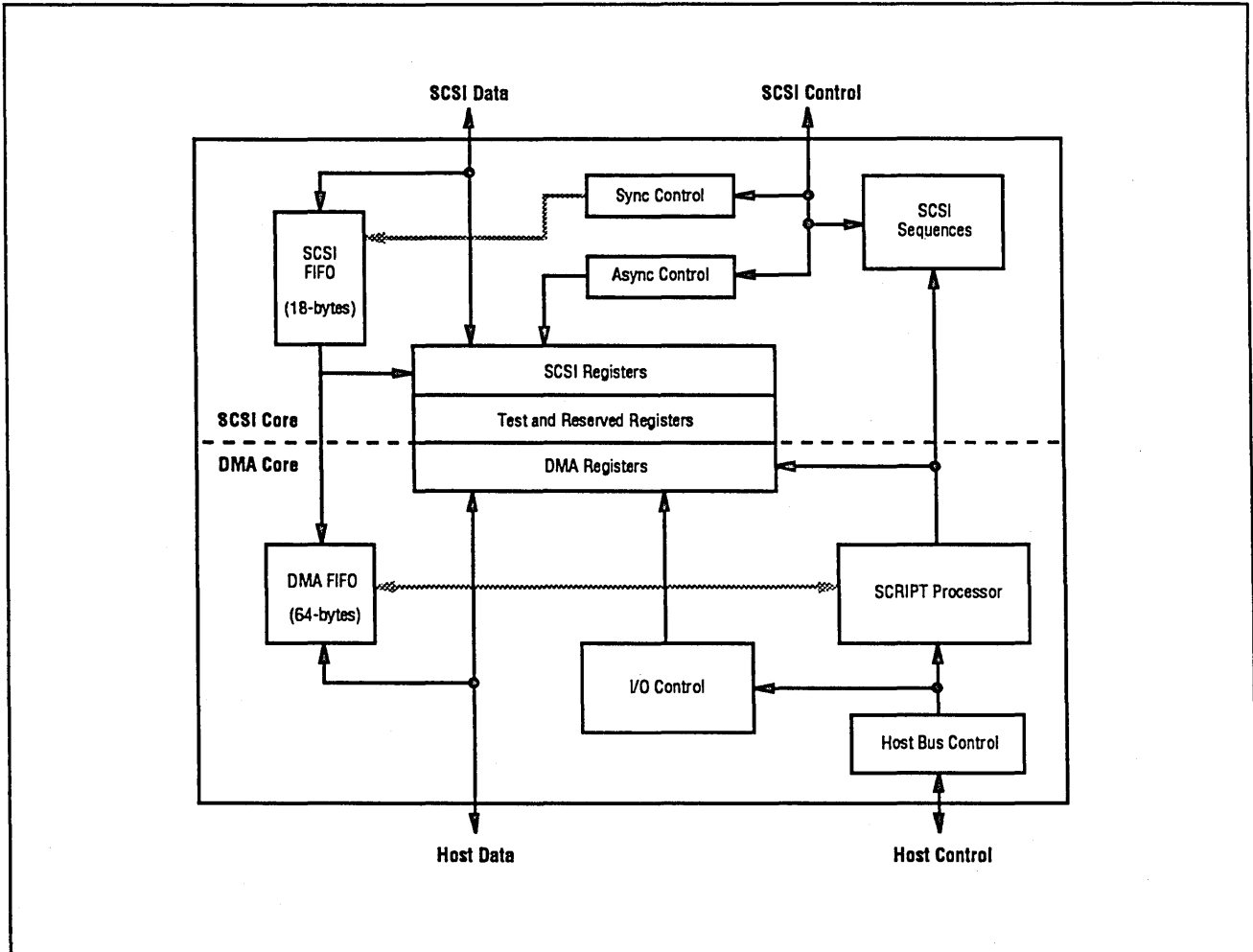
*Third generation (NCR 53C700/710/720)* SCSI devices provide an additional level of intelligence and integration as required for the next generation of SCSI devices. Third generation SCSI devices make execution decisions based on phase compares on the SCSI bus and incoming data value compares which will result in a minimum number of interrupts to the external processor. The third generation of SCSI devices reduces the cost of controller boards by relieving or eliminating the requirement for an external processor.

The third generation SCSI device is a programmable SCSI device that executes SCSI oriented commands. It reduces interrupt service routine complexities by providing unique status values to the external processor for any interrupts that do occur. Additionally, it has a fully integrated DMA channel that allows 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. Also, integration reduces cost and increases reliability.

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 buffers to the device. In second generation chips, the firmware is required to manage every detail of the error recovery mechanism, because the

high level sequence in the user 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. Following is a block drawing of the NCR 53C720 SCSI I/O Processor chip.

### NCR 53C720 SCSI I/O Processor Chip Block Diagram



## The NCR SCSI I/O Processor

The NCR 53C720 is an intelligent SCSI host adapter on a chip. A high-performance reusable SCSI core and an intelligent 32-bit bus master DMA controller have been integrated with the SCSI SCRIPTS processor to accommodate the flexibility requirements of SCSI-1, SCSI-2, and eventually SCSI-3. NCR products support these requirements while solving the protocol performance problems that have plagued both intelligent and non-intelligent adapter designs.

Unlike previous generation devices, the 53C720 SCSI core is controlled by an integrated DMA through a high-level logical interface. High level programming language commands controlling the SCSI core are fetched from external 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 (initiator or target).

Also, the SCSI SCRIPTS processor will transfer execution control (jump, call, return, carry, interrupt and interrupt on the fly) based on SCSI bus phase comparisons. Alternatively, a value in the SCSI SCRIPTS command can be compared to the input data value on the SCSI bus, allowing transfer of control based on input data comparison.

Using the wide SCSI option, data can be transferred over the SCSI bus at 20 Mbytes per second over a 16-bit single REQ/ACK cable (P-Cable). Also, 16 devices can be attached to the wide SCSI bus.

## DMA Component

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

The 53C720 supports 32-bit memory and automatically supports misaligned DMA transfers. Data bus enables are provided for

each byte lane. An on-chip, 64-byte FIFO permits 2, 4, 8 or 16-long words to be burst across the memory bus interface, providing memory transfer rates in excess of 66 Mbytes per second.

Sixteen bytes at a time can be burst into the FIFO using the cache line burst feature, supporting burst speeds in excess of 97 Mbytes per second. One, two, or four cache line bursts can occur before the chip gets off the system bus.

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.

Two other features of the 53C720 are the programmable timer that provides a "bus safety" feature and the flexible arbitration scheme that allows daisy chained or 'OR'ed memory bus request implementations.

## SCSI SCRIPTS™ Processor

The SCSI SCRIPTS processor is a specially designed 2 MIPS processor, located in the chip, that permits instructions to be fetched from external memory. Algorithms written in the SCSI SCRIPTS language and then compiled, control the SCSI and DMA modules and are executed from 16 or 32-bit system memory. Complex SCSI bus sequences are executed independently of the host CPU.

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

The SCSI SCRIPTS processor can begin a SCSI I/O operation in 500 nanoseconds. This compares to the 2-8 milliseconds 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.), changes in the SCSI logical definitions, or quickly incorporate new or popular options.

The 53C720 SCSI SCRIPTS processor is the latest member of the NCR third generation of SCSI chips. The 53C720 implements flexibility without sacrificing I/O performance.

## NCR SCSI SCRIPTS™ Description

SCSI SCRIPTS is a high level language used by the NCR 53C720 to execute SCSI sequences. The processor in the 53C720 chip executes the SCRIPTS. Therefore, SCRIPTS are independent of the CPU and system bus. For example, SCRIPTS for an EISA implementation on an 80386 or an 80386SX Micro Channel™ computer can therefore be identical to the SCRIPTS for a Motorola 68030 implementation.

After power up and initialization of the 53C720, the chip may be operated in one of two modes:

### 1. Low level register interface

### 2. SCSI SCRIPTS chained mode.

- Operating in the low level register interface mode, the user has access to the DMA control logic and the SCSI bus control logic and can operate the chip much 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 backward compatibility with SCSI chips requiring unique timings or bus sequences to operate properly. Another low level feature is loopback testing. In loopback mode, the SCSI core (controlled by a processor) can be directed to talk to the DMA core (controlled by SCRIPTS), allowing the internal data paths to be tested all the way to the chip's pads.

- Operating in the SCSI SCRIPTS chained mode, the 53C720 requires *only* a SCSI SCRIPTS start address. All subsequent commands are fetched from exter-

nal memory. Four bytes (or optionally two) at a time are fetched across the DMA interface and loaded into the command register. Command fetch and decode time is minimal at about 500 nanoseconds, when the chip is operating at its highest frequency.

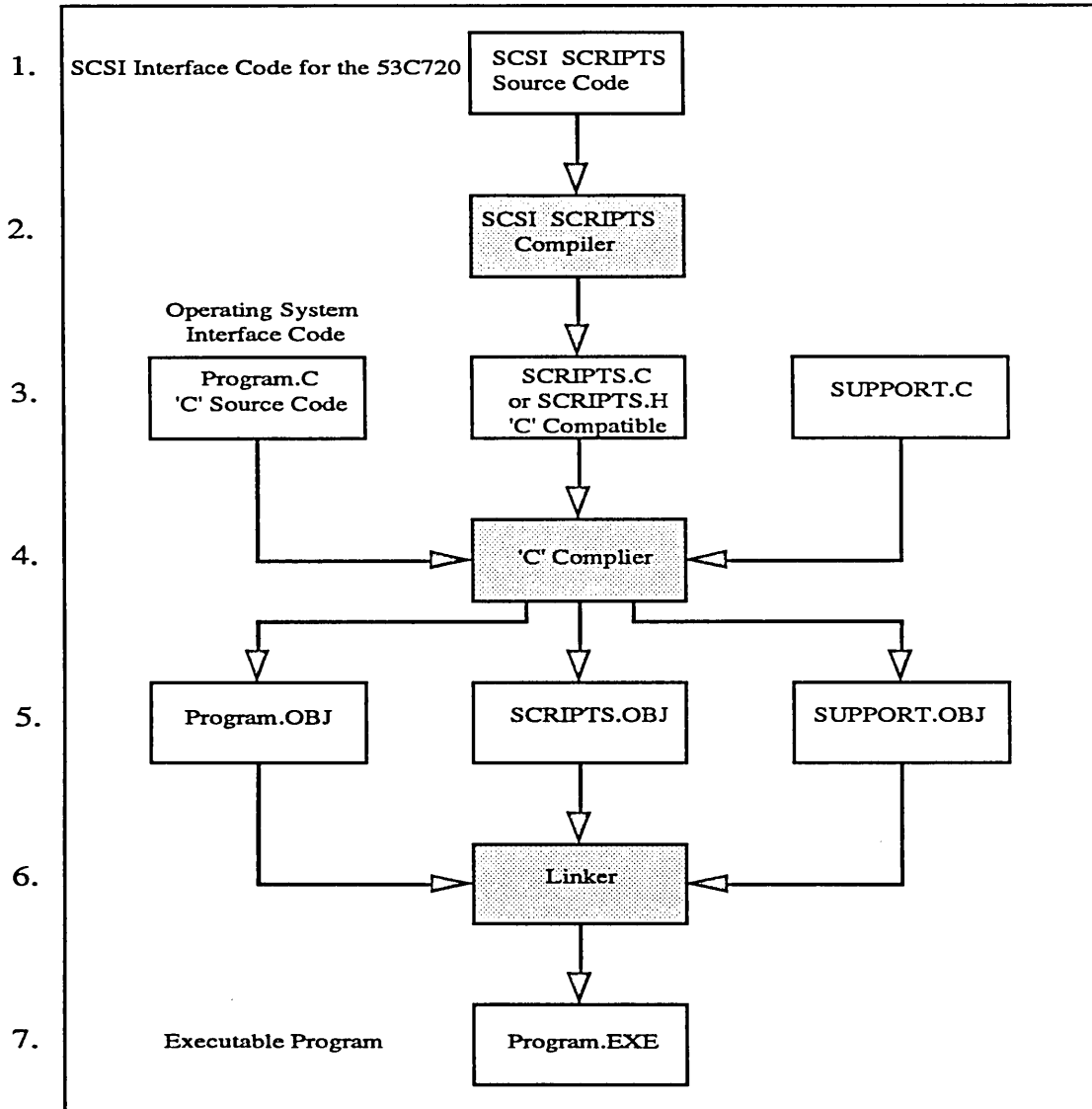
In the Table Indirect Mode the Data Structure Address (DSA) register must be loaded with the address of the data tables, and then data values (for example, byte count and address) are fetched after the SCSI SCRIPT instruction bytes are in the chip.

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

Commands are fetched until an interrupt 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 allows re-entry of the SCSI algorithm at any point. A high level interface can be used for both normal and exception conditions. Therefore, switching to a low level mode for error recovery is not normally required.

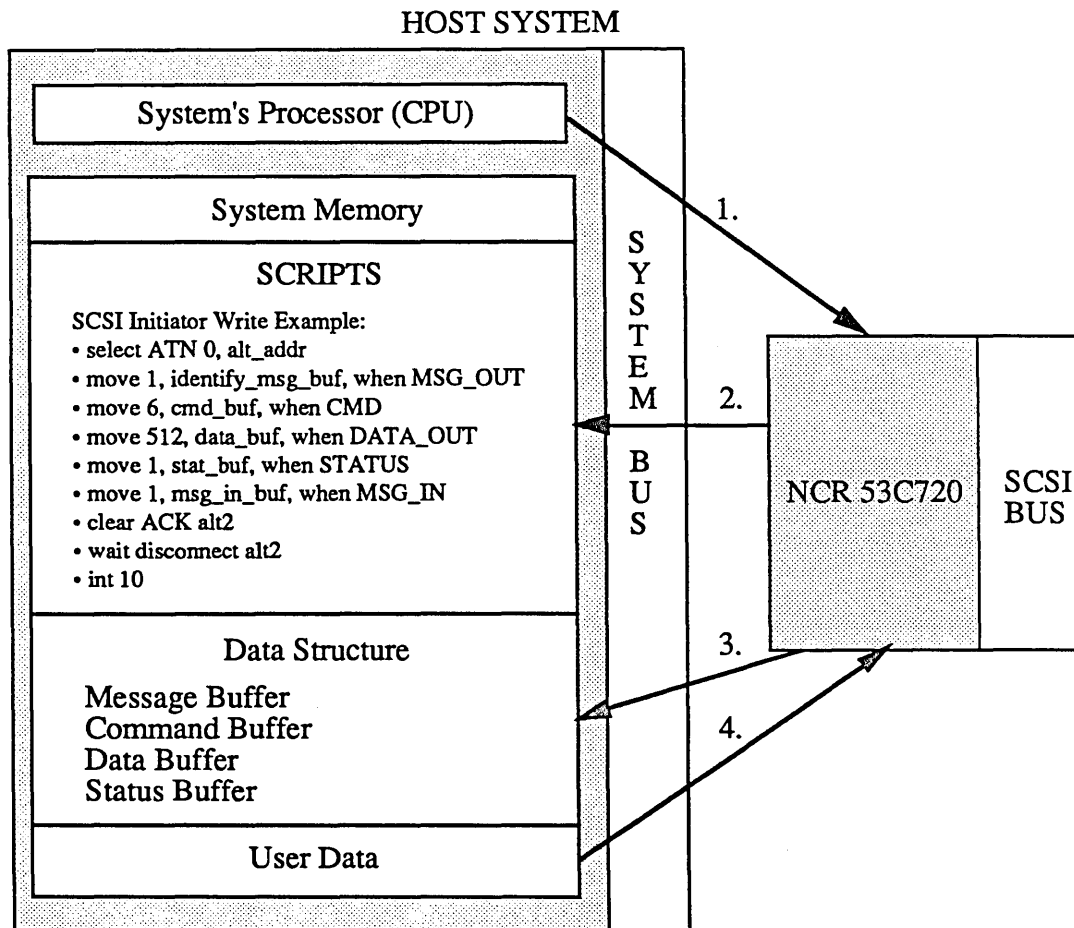


### How SCSI SCRIPTS becomes part of a C Language Program



1. Write SCSI SCRIPTS source code
2. Compile the SCSI SCRIPTS source code using the SCRIPTS compiler  
Output is a C static array whose contents are the instructions that are executed by the SCRIPTS processor
3. Write C Language source code
4. Compile all code using the C language compiler
5. Results are object (.obj) code
6. Link all the object modules together
7. Results are an executable program.

## Example of a SCRIPTS Operation



1. System processor writes Data Structure Address (DSA) value and SCRIPTS address into the 53C720 which starts the chip running
2. 53C720 becomes the bus master and fetches a SCRIPT
3. If required, the indirect data fetch gets address, byte counts, etc. for the execution

### SCSI initiator write example:

- select the target
- move the message out
- move the command bytes
- move the data bytes out
- move the status byte in
- move the command complete message in
- accept the message byte
- wait for the bus free interrupt
- interrupt when the command is complete

4. User data is moved from memory out to the SCSI bus (on a WRITE instruction), during the "move the data bytes out" operation, the main processor is freed up

## Chapter 2

# Developing NCR SCSI SCRIPTS

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

Use the SCRIPTS language to write SCRIPTS algorithms. Then compile them to create the object code required as input by the 53C720. The compiler output is like an object module, it includes relocation information required to load the SCRIPTS object module into main memory, if any relocation is required, and it can be directly included in firmware written in the C language.

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

Writing a logical I/O driver is an easy task for the 53C720. This is illustrated in the first SCSI SCRIPTS example. This code will perform a read or write function using the 53C720 in the high-level chained mode. Because SCSI algorithms are so simple when written in SCSI SCRIPTS, you can rapidly prototype SCSI algorithms for a proof of concept and concentrate later on more complete algorithms.

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

1. Definition area
2. SCRIPT area

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

```

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

;      Definition area INITIATOR ROLE

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

EXTERNAL status_adr     ; status_adr

EXTERNAL sendmsg        ; Ten byte buffer address offset.

EXTERNAL rcvmsg         ; Ten byte buffer address offset.

EXTERNAL cmd_adr        ; Buffer address offset for the SCSI command

EXTERNAL data_adr       ; Address of user data buffer

;*****
;* Absolute values are stored in the DSPS Register after an interrupt *
;* SCRIPT is executed for purposes of interrupt processing          *
;*****

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

```

```

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

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

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

ABSOLUTE err4 = 0x0ff04 ; Error — expected status phase

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

ABSOLUTE err5 = 0x0ff05 ; Error — expected message outphase

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

ABSOLUTE err7 = 0x0ff07 ; Error — got selected or reselected on select SCRIPT
    
```

```

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

```

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

```

;*****
; All of the text inside the parenthesis is passed directly to the *
; C compatible output. *
;*****
    
```

## Single-Tasking SCSI Example

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

If an unpredictable event occurs on the SCSI bus, a unique interrupt vector value is stored in the 53C720's DSPS register and is available for interrupt processing.

PROC sample:

```

SELECT atn from device, REL (resel_adr) ; select device with attention on
INT err1 when not MSG_OUT ; if the next phase is not message out,
                           interrupt
MOVE from sendmsg, when MSG_OUT ; sent the i.d. message out to the
                                target
INT err2 when not CMD ; if the next phase is not command,
                      interrupt
MOVE from cmd_adr, when CMD ; send the command bytes
JUMP rel (end1) when STATUS ; go to process cleanup if status phase
JUMP rel (input_data) if DATA_IN ; process data in phase
JUMP rel (output_data) if DATA_OUT ; or data out phase
INT err3 ; unexpected phase if here

```

INPUT\_DATA:

```

MOVE from data_adr, when DATA_IN ; process the data in phase
JUMP rel (end) ; and go process status

```

OUTPUT\_DATA:

```

MOVE from data_adr, when DATA_OUT ; process the data out phase

```

END:

```

INT err4 when not STATUS ; interrupt if not status phase

```

END1:

```

MOVE from status_adr, when STATUS ; move the status byte into memory
INT err5 when not MSG_IN ; interrupt if message in is not next
MOVE from rcvmsg, when MSG_IN ; move the command complete byte in
INT err6 if not 00 ; interrupt if it is not a command
                    complete message
CLEAR ack ; accept the message if there are no
          problems
WAIT disconnect ; wait for a physical disconnect
INT ok ; interrupt with an I/O complete

```

RESEL\_ADR:

```

INT err7

```

## SCSI SCRIPTS Compiler

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

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

### Invoking the SCSI SCRIPTS Compiler

In the following examples, items enclosed in double brackets “[ ]” are optional. The following format is used to invoke the compiler.

```
scc sourcefile [options]
```

#### Options:

##### **-o [OutputFilename]**

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

##### **-l [ListFilename]**

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

```
the long word instruction,  
the long word address, and  
the corresponding ASCII source  
instruction.
```

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

Next is a list consisting of absolute or relative variables, and their location in the SCRIPTS. This is followed by a list of labels and label locations that appear in

the SCRIPTS. The location is an offset from the beginning of the SCRIPTS.

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

##### **-z [debugfilename]**

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

##### **-e [errorfilename]**

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

##### **-v**

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

##### **-u**

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

##### **-w**

When this option is set the compiler uses the 53C720 mapping instead of the regular mapping for the 53C700 and the 53C710.

## SCSI SCRIPTS Compiler Output

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

### Source Code:

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

### Compiled Output:

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

};

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

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

### Example:

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

Then the SCRIPTS entry label values are defined with a prefix of "Ent\_".

### Example:

```
#define Ent_alt_addr 0x00000078
```

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

### Example:

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

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

### Example:

```
ULONG INSTRUCTIONS =
0x00000011;

ULONG PATCHES = 0x00000003;
```

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

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



# The NCR SCSI SCRIPTS Language Syntax

## SCSI SCRIPTS

NCR SCSI SCRIPTS is a high level language used to control NCR's line of intelligent processors (53C7X0). SCSI SCRIPTS consist of a series of lines. Blank lines and anything after a semi-colon on an input line are ignored by the NCR SCSI SCRIPTS Compiler. The front-end of the compiler receives SCSI SCRIPTS, compiles them and the back-end of the compiler outputs "C" compatible code.

The compiler is "*token*" oriented. The compiler reads SCSI SCRIPTS and splits each line up into tokens. White space and anything from a semicolon to the end of the line are ignored by the compiler and are not part of a token.

A token is any string of consecutive letters, underscores, dollar signs, or numbers.

A token that has a numeric value may be specified in decimal, hexadecimal, binary, or octal.

- Decimal numbers are specified by a string of digits not beginning with zero.
- Hexadecimal (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" (zero) followed by the octal digits.

## SCRIPTS Keywords

A reserved word is a token that has a particular and specific meaning when used in a SCRIPTS program. Reserved words are often called keywords because they "key" the translator to what follows.

Keywords the SCSI SCRIPTS compiler recognizes are classified as instruction keywords, phase keywords, register keywords, miscellaneous keywords, and compiler directives.

### Instruction Keywords

These keywords initiates a specific instruction.

**CALL** — Initiates a CALL instruction. CALL transfers control to an address location if given conditions are met. Differs from JUMP instruction in that the return address will be stored in the TEMP register.

**CHMOV** — Chained Block Move handles odd byte scatter/gather on wide SCSI.

**CLEAR** — This instruction clears specific bits in registers.

**DISCONNECT** — In the target mode this instruction causes the target to disconnect from the SCSI bus.

**INTFLY** — The Interrupt on the Fly instruction causes an interrupt that will not halt the SCRIPTS processor.

**INT** — Interrupt initiates an INTERRUPT instruction. Halts the SCRIPTS processor.

**JUMP** — Initiates a JUMP instruction. This transfers control to an address location if given conditions are met. Differs from CALL instruction in that the return address is not stored in the TEMP register.

**MOVE** — Initiates a Block Move instruction.

**MOVE MEMORY** — Initiates a move of a specified number of bytes from the source address to the destination address.

**NOP** — Initiates a no operation instruction.

**RESELECT** — Initiates a Reselect I/O instruction.

**RETURN** — Initiates a RETURN instruction. This will transfer control to an address location that is stored in the TEMP register. This is usually issued in response to an earlier CALL instruction (one deep on the stack).

**SELECT** — Initiates a SELECT I/O instruction.

**SET** — This instruction sets specific bits in registers.

### Phase Keywords

These keywords specify specific SCSI bus phases

**CMD** — Command indicates the instruction phase.

**DATA\_IN** — Indicates the data-in phase.

**DATA\_OUT** — Indicates the data-out phase.

**MSG\_IN** — Indicates the message-in phase.

**MSG\_OUT** — Indicates the message-out phase.

**RES4** — Reserved phase 4.

**RES5** — Reserved phase 5.

**STATUS** — Indicates status phase.

### Register Keywords

The following keywords represent the 53C720 register set. See the 53C720 Data Manual for more information.

**ADDER0** — Internal Adder Register 0

**ADDER1** — Internal Adder Register 1

**ADDER2** — Internal Adder Register 2

**ADDER3** — Internal Adder Register 3

**CTEST0** — Chip Test Register 0

**CTEST1** — Chip Test Register 1

**CTEST2** — Chip Test Register 2

**CTEST3** — Chip Test Register 3

**CTEST4** — Chip Test Register 4

**CTEST5** — Chip Test Register 5

**CTEST6** — Chip Test Register 6

**DBC0** — DMA Byte Count Register 0

**DBC1** — DMA Byte Count Register 1

**DBC2** — DMA Byte Count Register 2

**DCMD** — DMA Command Register

**DCNTL** — DMA Control Register

**DFIFO** — DMA FIFO Register

**DIEN** — DMA Interrupt Enable Register

**DMODE** — DMA Mode Register

**DNAD0** — DMA Next Address for Data Register 0

**DNAD1** — DMA Next Address for Data Register 1

**DNAD2** — DMA Next Address for Data Register 2

**DNAD3** — DMA Next Address for Data Register 3

**DSA0** — Data Structure Address Register 0

**DSA1** — Data Structure Address Register 1

**DSA2** — Data Structure Address Register 2

**DSA3** — Data Structure Address Register 3

**DSP0** — DMA Scripts Pointer Register 0

**DSP1** — DMA Scripts Pointer Register 1

**DSP2** — DMA Scripts Pointer Register 2

**DSP3** — DMA Scripts Pointer Register 3

**DSPS0** — DMA Scripts Pointer Save Register 0

**DSPS1** — DMA Scripts Pointer Save Register 1

**DSPS2** — DMA Scripts Pointer Save Register 2

**DSPS3** — DMA Scripts Pointer Save Register 3

**DSTAT** — DMA Status Register

**DWT** — DMA Watchdog Timeout Register

**GPREG** — General Purpose

**ISTAT** — Interrupt Status Register

**SBCL** — SCSI Bus Control Lines Register

**SBDL0** — SCSI Bus Data Lines Register 0

**SBDL1** — SCSI Bus Data Lines Register 1

**SCID** — SCSI Chip ID Register

**SCNTL0** — SCSI Control Register 0

**SCNTL1** — SCSI Control Register 1

**SCNTL2** — SCSI Control Register 2

**SCNTL3** — SCSI Control Register 3

**SCRATCHA0** — Scratch Pad A Register 0

**SCRATCHA1** — Scratch Pad A Register 1

**SCRATCHA2** — Scratch Pad A Register 2

**SCRATCHA3** — Scratch Pad A Register 3

**SCRATCHB0** — Scratch Pad B Register 0

**SCRATCHB1** — Scratch Pad B Register 1

**SCRATCHB2** — Scratch Pad B Register 2

**SCRATCHB3** — Scratch Pad B Register 3

**SDID** — SCSI Destination ID Register

**SFBR** — SCSI First Byte Received Register

**SIDL0** — SCSI Input Data Latch Register 0

**SIDL1** — SCSI Input Data Latch Register 1

**SIEN0** — SCSI Interrupt Enable Register 0

**SIEN1** — SCSI Interrupt Enable Register 1

**SIST0** — SCSI Interrupt Status 0

**SIST1** — SCSI Interrupt Status 1

SLPAR — SCSI Longitudinal Parity  
 SOCL — SCSI Output Control Latch Register  
 SODL0 — SCSI Output Data Latch Register 0  
 SODL1 — SCSI Output Data Latch Register 1  
 SSID — SCSI Selector ID  
 SSTAT0 — SCSI Status Register 0  
 SSTAT1 — SCSI Status Register 1  
 SSTAT2 — SCSI Status Register 2  
 STEST0 — SCSI Test 0  
 STEST1 — SCSI Test 1  
 STEST2 — SCSI Test 2  
 STEST3 — SCSI Test 3  
 STIME0 — SCSI Timer 0  
 STIME1 — SCSI Timer 1  
 SWIDE — SCSI Wide Residue Data  
 SXFER — SCSI Transfer Register  
 TEMP0 — Temporary Stack Register 0  
 TEMP1 — Temporary Stack Register 1  
 TEMP2 — Temporary Stack Register 2  
 TEMP3 — Temporary Stack Register 3

### Miscellaneous Keywords

The following keywords place conditions on the instructions.

ACK — Acknowledge manipulates the acknowledge flag bit in I/O instructions.  
 AND — Conditional AND operation.  
 ATN — Attention manipulates the attention flag bit in I/O instructions.  
 CARRY — Decisions are made based on the SCRIPTS processor carry bit.  
 FROM — Signifies that table indirect addressing is to be used in a block move instruction or an I/O instruction.  
 IF — Conditional IF operation. When used to compare phases in transfer control instructions, the current latched phase will be the phase that is evaluated. This keyword should not be used for block move operations. WHEN is the correct keyword for block move operations.

MASK — Performs masking operations on data.  
 MEMORY — Signifies the operation is to be from host memory.  
 NOT — Causes the negation of the following condition.  
 OR — Conditional OR operation.  
 PTR — Pointer signifies indirect addressing is to be used in a block move instruction.  
 REG — Register allows the user to access registers by a register number.  
 REL — Relative signifies that a transfer of control will be relative to the current program counter.  
 TARGET — Sets the processor to the target mode of operation.  
 TO — Indicates a move instruction from register TO register.  
 WAIT — Signifies that the SCSI processor should remain idle until a condition is met. (i.e. WAIT DISCONNECT)  
 WHEN — Conditional WHEN operation. When used to compare phases, the phase of the next assertion of the REQ line is the phase that is evaluated. This is the keyword that should be used for block move operations.  
 WITH — Used for target move operations. The phase that follows the WITH keyword will be asserted by the target device.

### Compiler Directives

These keywords are used as compiler directives and do not result in instructions being generated.

ABSOLUTE — Declares symbolic names for numeric values. Similar to the define statement in the C programming language.

**ENTRY** — Declares that the variable(s) following are entry points into the SCSI SCRIPTS.

**EXTERN** or **EXTERNAL** — Declares the variable(s) following are defined external to the SCRIPTS program. The assembler will keep an array of offsets where the variable(s) are used to facilitate replacing these external variable(s) with their absolute values.

**PASS** — Declares that the characters between the parenthesis ( ) will be passed unaltered to the output file. This permits the programmer to pass C code through to the output file.

**PROC** — Instructs the compiler to close out the current SCRIPTS instruction array and generate a new instruction array with the name of the array being the name following the PROC instruction. This allows for multiple SCRIPTS arrays within the same SCRIPTS file.

**RELATIVE** — Declares that the buffer names following are to be relative to one another.

## SCRIPTS Notation

[ ] Items enclosed in brackets are optional.

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

**KEYWORD** — A keyword is often called a reserved word. 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 '**offset**' means a signed 24 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 than 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 letter numbers, underscores, and the dollar sign. Names used for labels, externals, and variables 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 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 digit or to contain a dollar sign. Therefore, the SCSI SCRIPTS writer for DOS and Turbo C should avoid using names of this form.

## Compiler Directives Syntax

Compiler Directives do not produce opcodes, but rather inform the compiler of certain types of tokens and how these tokens will be treated in SCRIPTS instructions.

**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".

**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.

**EXTERNAL name [,name...]**

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. The compiler outputs an array that contains a list of instruction offsets where the external name is used. Use of the PASS option can help in this situation.

### PASS Option

To allow the SCSI SCRIPTS compiler more flexibility in the C environment, an option is included that allows the programmer to pass

64-character literal strings through to the C compiler. This feature allows the programmer to use any C expression that will be resolved when the output is compiled. Strings can be placed on a single line, or used in place of a 32-bit address.

**PASS (literal string)**

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

The following two SCRIPTS instructions illustrate how the PASS option can be used to defer the fixing of addresses until link time. Any C expression can be referred to (limited to 64-characters) if it will be converted to a 32-bit address by the C compiler.

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

A complete line of C code can be included in a SCRIPTS program and transferred intact to the C source output. For example,

```
PASS (#include "NCR.h")
```

results in

```
#include "NCR.h"
```

appearing in the final output.

### PROC label

PROC is a way of addressing External Labels. A SCRIPTS programmer may want to write modular code instead of one large routine. To have modules, some type of

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

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

Declares 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.

**KEYWORD count,**

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

**KEYWORD count, address**

When an instruction requires an **address**, use a 32-bit number, the label name, the variable name in the relative data area (previously declared with the RELATIVE keyword), or the external variable name (previously declared with the EXTERNAL keyword).

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

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

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

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

## Instruction Keywords

### MOVE Instructions

- Block Move
- Chained Block Move
- Memory to Memory Move

There are three types of move instructions; Block Move, Chained Block Move, and Memory to Memory Move. Block Move instruction transfers data to (from) user memory from (to) the SCSI bus. Chained Block Move is for handling wide SCSI, odd byte scatter/gather situations. Memory to Memory Move is for copying a specified number of bytes from a source address to the destination address. Both Block Move and Chained Block Move can use the three addressing modes.

- Direct Block Move
- Indirect Block Move
- Table Indirect Block Move

The 53C720 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 SCRIPTS and the actual value on the bus. If the phases do not match, an external interrupt occurs. If the phase matches, then data is transferred in or out according to the phase lines. When the count goes to zero, the next sequential SCRIPTS instruction is fetched.

## 1. Block Move

### ◦ Direct Block Move

In a Direct Block Move instruction the 32-bit SCSI or user data start address is uniquely specified in the Block Move instruction.

**Syntax** MOVE count, address, WITH Phase  
MOVE count, address, WHEN Phase

**count** Count is a 24-bit value specifying the number of bytes to transfer.

**address** 32-bit start address specifies the location where the Block Move is to take place. Note: Address can be replaced with "PASS (C expression)". See PASS option for more details.

**WITH/** Specify the Block Move  
**WHEN** function codes

WITH signals the target role which sets the phase values

WHEN is the initiator "test for phase" feature

**Phase** Specifies the phase field of the instruction

### ◦ Indirect Block Move

In an Indirect Block Move instruction the 32-bit SCSI or user data start address is the address of a pointer to the actual data buffer address.

**Syntax** MOVE count, PTR address, WITH Phase  
MOVE count, PTR address, WHEN Phase

**count** Count is a 24-bit value specifying the number of bytes to transfer.

**PTR** The PTR (pointer) keyword indicates the Block Move is to be an Indirect Block Move.

**address** Specifies the address of a pointer that points to the actual data buffer address. Note: Address can be replaced with "PASS (C expression)". See PASS option for more

details.

*WITH/* Specify the Block Move  
*WHEN* function codes.

*WITH* signals the target role which sets the phase values.

*WHEN* is the initiator “test for phase” feature.

*Phase* Specifies the phase field of the instruction.

• **Table Indirect Block Move**

In a Table Indirect Block Move instruction the 32-bit start address is treated as a 24-bit signed value. After the instruction is moved into the 53C720, the 24 bits are added to the Data Structure Address (DSA) register to form a 32-bit physical address.

*Syntax* MOVE FROM offset, WITH Phase  
 MOVE FROM offset, WHEN Phase

*FROM* The FROM keyword indicates the Block Move is to be a Table Indirect Block Move.

*offset* A 24-bit signed value is combined with the Data Structure Address (DSA) register to form a 32-bit physical address where the Block Move is to take place. Note: PASS option can not be used.

*WITH/* Specify the Block Move  
*WHEN* function codes.

*WITH* signals the target role which sets the phase values.

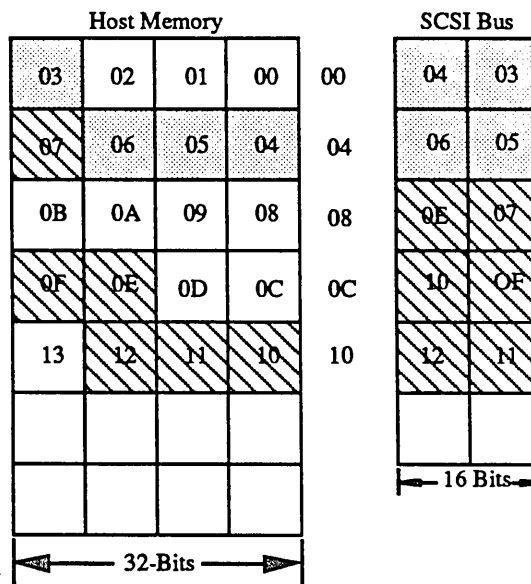
*WHEN* is the initiator “test for phase” feature.

*Phase* Specifies the phase field of the instruction.

**2. Chained Block Move**

Chained Block Move (CHMOV) is a second type of Block Move. There are several problems that can arise when the wide SCSI bus is combined with odd byte scatter/gather buffers in system memory. A simple example can show one situation, from a system perspective.

- 10 byte transfer with two 5 byte scatter/gather pieces
- first move is a CHMOV to allow for the odd byte size
- four bytes go on to the SCSI bus
- fifth byte is saved in the 53C720
- next move instruction is fetched (A MOVE)
- sixth byte is matched with fifth byte for a word transfer and sent out
- last four bytes are sent out



CHMOV 5, 0x03, WHEN DATA\_OUT

move 5 bytes from address 03 in the host memory to the SCSI bus (bytes 03, 04, 05 and 06 are moved and byte 07 remains in the low order byte of the SCSI Output Data Latch register and is married with the first byte of the following move instruction byte 0E)



```
MOVE 5, 0x0E, WHEN DATA_OUT
; move 5 bytes from address 0E in the
host memory to the SCSI bus
```

Because the user is relieved of the odd byte problems, the 53C720 is much simpler to use in the wide SCSI environment. A user that expects any odd byte transfers should use the CHMOV instruction for a list of move instructions, all but the last move should be CHMOV, and the last one should be a Block Move.

#### ◦ Direct Chained Block Move

In a Direct Chained Block Move instruction the 32-bit SCSI or user data start address is uniquely specified in the Chained Block Move instruction.

**Syntax** CHMOV count, address, WITH Phase  
CHMOV count, address, WHEN Phase

**count** Count is a 24-bit value specifying the number of bytes to transfer.

**address** 32-bit start address specifies the location where the Chained Block Move is to take place. Note: Address can be replaced with PASS (C Name) (see PASS option for more details).

**WITH/** Specify the Chained Block  
**WHEN** Move function codes.

WITH signals the target role which sets the phase values.

WHEN is the initiator “test for phase” feature.

**Phase** Specifies the phase field of the instruction.

#### ◦ Indirect Chained Block Move

In an Indirect Chained Block Move instruction the 32-bit SCSI or user data start address is the address of a pointer to the actual data buffer address.

**Syntax** CHMOV count, PTR address, WITH Phase  
CHMOV count, PTR address, WHEN Phase

**count** Count is a 24-bit value specifying the number of bytes to transfer.

**PTR** The PTR (pointer) keyword indicates the Block Move is to be an Indirect Block Move.

**address** Specifies the address of a pointer that points to the actual data buffer address. Note: Address can be replaced with PASS (C Name) (see PASS option for more details).

**WITH/** Specify the Chained Block  
**WHEN** Move function codes.

WITH signals the target role which sets the phase values.

WHEN is the initiator “test for phase” feature.

**Phase** Specifies the phase field of the instruction.

#### ◦ Table Indirect Chained Block Move

In a Table Indirect Chained Block Move instruction the 32-bit start address is treated as a 24-bit signed value. After the instruction is moved into the 53C720, the 24 bits are added to the Data Structure Address (DSA) register to form a 32-bit physical address.

**Syntax** CHMOV FROM offset, WITH Phase  
CHMOV FROM offset, WHEN Phase

**FROM** The FROM keyword indicates the Chained Block Move is to be a Table Indirect Chained Block Move.

**offset** A 24-bit signed value is combined with the Data Structure Address (DSA) register to form a 32-bit physical address where the Block Move is to take place. Note: PASS option can not be used.

**WITH/** Specify the Chained Block  
**WHEN** Move function codes.

**WITH** signals the target role which sets the phase values.

**WHEN** is the initiator “test for phase” feature.

**Phase** Specifies the phase field of the instruction.

### 3. Memory-to-Memory Move

The Memory-to-Memory Move instruction is used to copy the specified number of bytes from the source address to the destination address. This instruction allows the 53C720 to become a high-speed DMA chip. DATA is moved from the source address into the chip’s DMA FIFO and then out to the destination address. This instruction does not destroy the TEMP or DSA register. There is no indirect capability with this instruction so that the physical 32-bit address must be in the SCRIPTS. The PASS option can be used with either or both addresses to allow the user to designate a C Name that can be resolved when the C code is compiled. A 24-bit byte counter permits large moves to occur with no intervention required by the processor.

**Syntax** MOVE MEMORY count, source address, destination address

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

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

The last two bits of the source and destination must be equal, but there are no other restrictions on the address values. For example, you can move a long-word or a byte from 0 to 4, 1 to 5, 2 to 6, or from 3 to 7, etc. But you can not move from 0 to 5 or 2 to 7, etc. The source data and the destination data needs to start on the same byte lane. Note that if a 53C720 register address is the source or destination, then this instruction can be used to read or write system memory from SCRIPTS. This capability is very useful for saving the state of an I/O in a multi-threaded I/O environment.

## JUMP Instruction

JUMP transfers control to an address location if given conditions are met. JUMP is different from a CALL instruction because the return address is not stored in the TEMP register. The conditional JUMP instructions all have the same general form.

### Syntax

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

*address* The SCSI SCRIPTS address that will be transferred to if the JUMP is taken. Limited to 24 bits if the REL option is used. Pass option can be used.

*WHEN* Sets the Wait bit in the Transfer Control Instruction.

*IF* Do not set the Wait bit.

If NOT follows WHEN or IF, then the True/False bit of the Transfer Control Instruction is not set. Otherwise, the bit will be set.

*Phase* When present, the instruction will compare the phase value to the phase bits stored in the chip to determine which SCRIPTS to execute next.

*data* When present, the instruction will compare the data value to the value in the SCSI First Byte Received (SFBR) register to determine which SCRIPTS to execute next.

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

*CARRY* When present (it is not valid with phase or data), the instruction will check the carry bit in the chip to determine which SCRIPTS to execute next.

*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 & CARRY. "NOT Phase OR data" is the negation of "Phase AND data" or "Phase OR data".

*MASK* Always use with an 'AND' keyword. The data following the keyword 'MASK' allows a SCRIPTS to selectively compare the bits within the SCSI First Byte Received (SFBR) register.

Any bits that are on in the MASK value 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.

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

Note that the address values can be replaced with a REL (Address). The value of the address must be a signed 24 bit value.

REL (Address) can replace an address in the JUMP instruction for a relative rather than absolute JUMP.

PASS (any valid C expression) can replace an address in the JUMP instruction.

For low level or bit information on the Jump, Call, Return, Interrupt and Interrupt on the Fly instructions see "Transfer Control Instruction" in Chapter 10.

## CALL Instruction

CALL instruction transfers control to an address location if given conditions are met. CALL is different from a JUMP instruction in that the return address will be stored in the TEMP register. All conditional CALL instructions have the same general form. A CALL can only be one level deep on the stack.

### Syntax

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

**address** The SCSI SCRIPTS address transferred to if the CALL is taken.

**WHEN** Set the Wait bit in the Transfer Control Instruction.

- IF** Do not set the Wait bit.
- If WHEN or IF are followed by NOT, then the True/False bit of the Transfer Control Instruction is not set. Otherwise, the bit will be set.
- Phase** When present, the instruction will compare the phase value to the phase bits stored in the chip to determine which SCRIPTS to execute next.
- data** When present, the instruction will compare the data value to the value in the SCSI First Byte Received (SFBR) register to determine which SCRIPTS to execute next.
- If both 'Phase' and 'data' are specified, they must be in that order and they must be separated by the keyword AND or OR.
- CARRY** When present (it is not valid with phase or data), the instruction will check the carry bit in the chip to determine which SCRIPTS to execute next.
- 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 OR CARRY. "NOT Phase OR data" is the negation of "Phase AND data" or "Phase OR data".
- MASK** Always use with an 'AND' keyword. The data following the keyword 'MASK' allows a SCRIPTS to selectively compare the bits within the SCSI First Byte Received (SFBR) register.
- Any bits that are on in the MASK value 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.
- REL** Used if the jump is to be relative to the current program counter.
- Note that the address values can be replaced with a REL (Address). The value of the address must be a signed 24 bit value. Also PASS (Any valid C expression) can replace an address.
- Pass (any valid C expression) can replace an address in the CALL instruction. REL (address) can replace an address in the CALL instruction.

## RETURN Instruction

The RETURN instruction transfers control to an address location that is stored in the TEMP register. This is usually issued in response to an earlier CALL instruction (one deep on the stack). All conditional RETURN instructions have the same general form.

### Syntax

```

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
RETURN, IF CARRY
RETURN, IF NOT CARRY

```

**Address** The SCSI SCRIPTS address that will be transferred to if the RETURN is taken.

**WHEN** Set the Wait bit in the Transfer Control Instruction.

**IF** Do not set the Wait bit.

If WHEN or IF are followed by

NOT, then the True/False bit of the Transfer Control Instruction is not set. Otherwise, the bit will be set.

**Phase** When present, the instruction will compare the phase value to the phase bits stored in the chip to determine which SCRIPTS to execute next.

**data** When present, the instruction will compare the data value to the value in the SCSI First Byte Received (SFBR) register to determine which SCRIPTS to execute next.

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

**CARRY** When present (it is not valid with phase or data), the instruction will check the carry bit in the chip to determine which SCRIPTS to execute next.

**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 OR CARRY. "NOT Phase OR data" is the negation of "Phase AND data" or "Phase OR data".

**MASK** Always use with an 'AND' keyword. The data following the keyword 'MASK' allows a SCRIPTS to selectively compare the bits within the SCSI First Byte Received (SFBR) register.

Any bits that are on in the MASK value 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.

## Interrupt Instruction

The Interrupt (INT) instruction interrupts the SCRIPTS processor. All conditional INT instructions have the same general form.

### Syntax

```

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

```

*value* The 32-bit data value that will be placed in the DSPS register if the INT condition is evaluated as true.

*WHEN* Set the Wait bit in the Transfer Control Instruction.

*IF* Do not set the Wait bit.

If *WHEN* or *IF* is followed by *NOT*, then the True/False bit of the Transfer Control Instruction is not set. Otherwise, the bit will be set.

*Phase* When present, the instruction will compare the phase value to the phase bits stored in the chip to determine which SCRIPTS to execute next.

*data* When present, the instruction will compare the data value to the value in the SCSI First Byte Received (SFBR) register to determine which SCRIPTS to execute next.

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

*CARRY* When present (it is not valid with phase or data), the instruction will check the carry bit in the chip to determine which SCRIPTS to execute next.

*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 OR CARRY*. "NOT Phase OR data" is the negation of "Phase AND data" or "Phase OR data".

*MASK* Always use with an 'AND' keyword. The data following the keyword 'MASK' allows a SCRIPTS to selectively compare the bits within the SCSI First Byte Received (SFBR) register.

Any bits that are on in the *MASK* value 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.

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

## Interrupt on the Fly Instruction

The Interrupt on the Fly (INTFLY) instruction causes an interrupt that will not halt the SCRIPTS processor. All conditional INTFLY instructions have the same general form.

### Syntax

```

INTFLY
INTFLY, IF ATN
INTFLY, IF Phase
INTFLY, IF data
INTFLY, IF data, AND MASK data
INTFLY, IF ATN AND data
INTFLY, IF ATN AND data,
    AND MASK data
INTFLY, IF Phase AND data
INTFLY, IF Phase AND data,
    AND MASK data
INTFLY, WHEN Phase
INTFLY, WHEN data
INTFLY, WHEN data,
    AND MASK data
INTFLY, WHEN Phase AND data
INTFLY, WHEN Phase AND data,
    AND MASK data
INTFLY, IF NOT ATN
INTFLY, IF NOT Phase
INTFLY, IF NOT data
INTFLY, IF NOT data,
    AND MASK data
INTFLY, IF NOT ATN OR data
INTFLY, IF NOT ATN OR data,
    AND MASK data
INTFLY, IF NOT Phase OR data
INTFLY, IF NOT Phase OR data,
    AND MASK data
INTFLY, WHEN NOT Phase
INTFLY, WHEN NOT data
INTFLY, WHEN NOT data,
    AND MASK data
INTFLY, WHEN NOT Phase OR data
INTFLY, WHEN NOT Phase OR data,
    AND MASK data
INTFLY, IF CARRY
INTFLY, IF NOT CARRY

```

**WHEN** Set the Wait bit in the Transfer Control Instruction.

**IF** Do not set the Wait bit.

If WHEN or IF is followed by NOT, then the True/False bit of the Transfer Control Instruction is not set. Otherwise, the bit will be set.

**Phase** When present, the instruction will compare the phase value to the phase bits stored in the chip to determine which SCRIPTS to execute next.

**data** When present, the instruction will compare the data value to the value in the SCSI First Byte Received (SFBR) register to determine which SCRIPTS to execute next.

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

**CARRY** When present (it is not valid with phase or data), the instruction will check the carry bit in the chip to determine which SCRIPTS to execute next.

**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 OR CARRY. "NOT Phase OR data" is the negation of "Phase AND data" or "Phase OR data".

**MASK** Always use with an 'AND' keyword. The data following the keyword 'MASK' allows a SCRIPTS to selectively compare the bits within the SCSI First Byte Received (SFBR) register.

Any bits that are on in the MASK value 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.

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



## Miscellaneous Instructions

The following miscellaneous instructions are I/O instructions. These instructions are either executed in the initiator or target mode. Target mode instructions are:

- Reselect
- Disconnect
- Wait Select

Initiator mode instructions are:

- Select
- Wait Disconnect
- Wait Reselect
- Set
- Clear

## Select Instruction

Select initiates a SELECT I/O instruction. Select causes the 53C720 to arbitrate for the SCSI bus by asserting the SCSI ID stored in the SCID register. When the 53C720 wins arbitration, it attempts to select the SCSI device whose ID is defined in the destination ID field of the instruction. If arbitration fails, jumps to address.

### Syntax

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

If ATN is present, the "SELECT with ATN" bit is turned on.

'ID' specifies the destination SCSI ID. REL allows a relative jump and FROM allows a table indirect fetch of device ID, offset and period for synchronous transfers. Address can use Pass.

## Reselect Instruction

Reselect initiates a RESELECT I/O instruction. Reselect causes the 53C720 to arbitrate for the SCSI bus by asserting the SCSI ID stored in the SCID register. When the 53C720 wins arbitration, it attempts to reselect the SCSI device whose ID is defined in the destination ID field of the instruction. If arbitration fails, jumps to address.

### Syntax

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

'ID' specifies the destination SCSI ID. REL allows a relative jump and FROM allows a table indirect fetch of device ID, offset and period for synchronous transfers. Address can use Pass.

## Wait Disconnect Instruction

The Wait Disconnect instruction causes the 53C720 to initiate a wait for the target to perform a "legal" disconnect from the SCSI bus. A "legal" disconnect occurs when BSY/ and SEL/ are inactive for a minimum of one Bus Free Delay (400 ns), after the 53C720 has received a Disconnect Message or a Command Complete Message.

### Syntax

```
WAIT DISCONNECT
```

## Disconnect Instruction

The Disconnect instruction causes the 53C720 to physically disconnect from the SCSI bus.

### Syntax

```
DISCONNECT
```

## Wait Reselect Instruction

The Wait Reselect instruction causes the 53C720 to initiate a wait 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 53C720. However, if the chip is selected, then the alternate jump address should contain the address of an algorithm for a selection.

### Syntax

```
WAIT RESELECT Address
WAIT RESELECT REL (Address)
```

REL allows the alternate address to be relative.

## Wait Select Instruction

The Wait Select instruction causes the 53C720 chip to wait for a SCSI selection by another device on the SCSI bus. If the chip is already selected, then the next SCSI SCRIPTS is fetched and executed. When a bus initiated interrupt or reselect occurs, the chip optionally changes to the initiator mode and fetches the next instruction from the address pointed to by the 32-bit jump address, and continues execution.

### Syntax

```
WAIT SELECT address
WAIT SELECT REL (Address)
```

## Set Instruction

When the ACK/ or ATN/ bits are set, the corresponding bits in the SCSI Output Control Latch (SOCL) register are set. ACK/ or ATN/ should not be set except for testing (diagnostic) purposes. When the target bit is set, the corresponding bit in the SCSI Control 0 (SCNTLO) register is also set. When the carry bit is set the corresponding bit in the ALU is set.

If the 53C720 is connected as a target, the following set and clear instructions will have no meaning (the SCSI target role is active) and should not be used.

### Syntax

```
SET TARGET
SET ACK
SET ATN
SET CARRY
SET ACK and ATN and TARGET
and CARRY
```

All four or three or any two of the keywords (ACK, ATN, TARGET, or CARRY) may be used.

## Clear Instruction

When the ACK/ or ATN/ bits are set, the corresponding bits in the SCSI Output Control Latch (SOCL) register are cleared. ACK/ or ATN/ should not be cleared

except for testing (diagnostic) purposes. When the target bit is cleared, the corresponding bit in the SCSI Control 0 (SCNTLO) register is cleared. When the carry bit is cleared the corresponding bit in the ALU is cleared.

### Syntax

```
CLEAR TARGET
CLEAR ACK
CLEAR ATN
CLEAR CARRY
CLEAR ACK and ATN and TARGET
and CARRY
```

All four or three or any two of the keywords (ACK, ATN, TARGET, or CARRY) may be used. Clear target can be used in both the initiator and the target mode. In the target mode, use CLEAR TARGET to get back to the initiator mode. CLEAR CARRY can be used in either the initiator or the target mode.

For low level or bit information on the Reselect, Disconnect, Wait Select, Select, Wait Disconnect, Wait Reselect, Set, and Clear instructions see "I/O Instruction" in Chapter 10.

## Register Read/Write Instruction

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

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

Valid register names are:

ADDER0	ADDER1	ADDER2	ADDER3
CTEST0	CTEST1	CTEST2	CTEST3
CTEST4	CTEST5	CTEST6	DBC0
DBC1	DBC2	DCMD	DCNTL
DFIFO	DIEN	DMODE	DNAD0
DNAD1	DNAD2	DNAD3	DSA0
DSA1	DSA2	DSA3	DSP0
DSP1	DSP2	DSP3	DSPS0
DSPS1	DSPS2	DSPS3	DSTAT
DWT	GPREG	ISTAT	SBCL
SBDL0	SBDL1	SCID	
SCNTL0	SCNTL1	SCNTL2	SCNTL3
SCRATCHA0	SCRATCHA1	SCRATCHA2	
SCRATCHA3			
SCRATCHB0	SCRATCHB1	SCRATCHB2	
SCRATCHB3			
SDID	SFBR	SIDL0	SIDL1
SIEN0	SIEN1	SIST0	SIST1
SLPAR	SOCL	SODL0	SODL1
SSID	SSTAT0	SSTAT1	SSTAT2
STEST0	STEST1	STEST2	STEST3
STIME0	STIME1	SWIDE	SXFER
TEMP0	TEMP1	TEMP2	TEMP3

REG(n), where n is a value from 0 to hexadecimal 05f.

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

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

**operator:** – Valid operators are OR (|), AND (&), addition (+) and subtraction (-).

Register writes are very useful, but caution must be exercised when this mode is used. Writing to certain registers could

have disastrous effects on the SCSI bus or operation of the chip. When a register is written or read, side effects may occur; the degree and possibility of these effects must be clearly understood.

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

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

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

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

```

MOVE register TO register
MOVE data8 TO register
MOVE register | data8 TO register
MOVE register & data8 TO register
MOVE register + data8 TO register
MOVE register + data8 TO register WITH CARRY
MOVE register - data8 TO register

```

**Notes**

A large, empty rectangular box with a thin black border, occupying most of the page. It is intended for the user to write notes.

## SCSI SCRIPTS Use of Scatter/Gather

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

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

The 53C720 has an efficient solution to the scatter/gather performance degradation problem. Each page of user data is represented by a Block Move command. The only overhead required to move to the next page of data is a SCSI SCRIPTS fetch. No firmware interrupt is required (normally a *minimum* of 80 microseconds in a system environment). Nor is firmware required to re-instruct a DMA controller.

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

With the read/write capability of the 53C720, another solution exists for the scatter/gather problem. The following SCSI SCRIPT uses the increment register feature and the table indirect feature to update the address and count values in the chip's registers. By fetching these values indirectly and adding 8 to the Data Structure Address DSA register each time through the loop, the SCRIPT can continue to fetch user data from various locations. The actual SCRIPT is:

```

Loop:
    MOVE from ADDR when DATA_OUT
        ; Perform the move
    MOVE DSA (0) +8 to SFBR
    CLEAR CARRY
        ; Increment to the next DSA entry
    JUMP L1 IF NOT CARRY
        ; Check for Wraparound
    MOVE DSA(1)+1 to DSA(1)
        ; handle a one-byte overflow

L1:
    MOVE SFBR to DSA(0)
        ; update the DSA register
    JUMP rel(Loop) when DATA_OUT
        ; repeat until a phase change

```

This SCRIPTS algorithm allows for a large number (8192) of Data Structure Table entries in the scatter/gather list. An alternative to simply waiting for a phase change is to use a counter in the loop and exit on zero. To allow for disconnects in the loop, save the Data Structure Address (DSA) register value when processing the disconnect message.

The 53C720 can process scatter/gather requests in a very simple manner and simultaneously, dramatically reduce I/O overhead.

**Notes**

A large, empty rectangular box with a thin black border, occupying most of the page. It is intended for the user to write notes.

# SCSI SCRIPTS for an Initiator and Target

## SCRIPTS for the Initiator Role

```

=====
;
;                                     Definition area INITIATOR ROLE
;
=====

ABSOLUTE    device = 0x000    ; Target Device id offset in the Data structure
ABSOLUTE    sendmsg = 0x008   ; Send message offset for count and address
ABSOLUTE    rcvmsg = 0x010   ; Receive message offset for count and address
ABSOLUTE    cmd_adr = 0x018   ; SCSI command offset for count and address
ABSOLUTE    data_adr = 0x020  ; User data buffer offset for count and address

ABSOLUTE    ok = 0x0ff00     ; No Error — good I/O
ABSOLUTE    check_cond = 0x0fffe ; SCSI status returned is check condition
ABSOLUTE    busy = 0x0fffd   ; SCSI status returned is busy
ABSOLUTE    reserved = 0x0fffc ; SCSI status returned is reservation conflict
ABSOLUTE    bad_status = 0x0fffb ; SCSI status returned is unknown

ABSOLUTE    err1 = 0x0ff01    ; Error — not message-out after selection
ABSOLUTE    err2 = 0x0ff02    ; Error — unexpected SCSI phase before command phase
ABSOLUTE    err3 = 0x0ff03    ; Error — unexpected SCSI phase after a command transfer
ABSOLUTE    err4 = 0x0ff04    ; Error — not msg-in phase after status phase
ABSOLUTE    err5 = 0x0ff05    ; Error — unexpected phase after a data transfer
ABSOLUTE    err6 = 0x0ff06    ; Error — unexpected msg-in phase before command phase
ABSOLUTE    err7 = 0x0ff07    ; Error — extended msg present before a command phase
ABSOLUTE    err8 = 0x0ff08    ; Error — save data pointers before a command phase
ABSOLUTE    err9 = 0x0ff09    ; Error — disconnect before command phase
ABSOLUTE    err10 = 0x0ff10   ; Error — save data pointers after the command phase
ABSOLUTE    err11 = 0x0ff11   ; Error — unexpected msg after command phase
ABSOLUTE    err12 = 0x0ff12   ; Error — extended message present after the command phase
ABSOLUTE    err13 = 0x0ff13   ; Error — disconnect after a command phase
ABSOLUTE    err14 = 0x0ff14   ; Error — save data pointers after a data transfer
ABSOLUTE    err15 = 0x0ff15   ; Error — unexpected message after a data transfer
ABSOLUTE    err16 = 0x0ff16   ; Error — extended message after a data transfer
ABSOLUTE    err17 = 0x0ff17   ; Error — disconnect after a data transfer
ABSOLUTE    err18 = 0x0ff18   ; Error — Message-in not received after reselection
ABSOLUTE    err19 = 0x0ff19   ; Error — Data-in phase after reselection and id msg rcvd
ABSOLUTE    err20 = 0x0ff20   ; Error — Data-out phase after reselection and id msg rcvd
ABSOLUTE    err21 = 0x0ff21   ; Error — Msg-in phase after reselection and id msg rcvd
ABSOLUTE    err22 = 0x0ff22   ; Error — Status phase after reselection and id msg rcvd
ABSOLUTE    err23 = 0x0ff23   ; Error — Msg-out phase after reselection and id msg rcvd
ABSOLUTE    err24 = 0x0ff24   ; Error — Unknown phase after reselection and id msg rcvd
ABSOLUTE    err25 = 0x0ff25   ; Error — Selected as a target
ABSOLUTE    err26 = 0x0ff26   ; Error — Unexpected message rcvd instead of command
;                               complete

ENTRY start_up                ; SCSI I/O entry point. This address must be loaded into the
;                               53C720 before initiating a SCSI I/O.

```

**; SCRIPTS AREA**

```

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

```

start\_up:

```

SELECT ATN FROM device, PASS(&Resel) ; 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.
JUMP REL(end), WHEN STATUS ; If the next phase is status, go to end. Wait
; for valid phase before performing the
; comparison.
JUMP REL(cmd_phase), IF CMD
INT err1, IF NOT MSG_OUT ; If not msg-out phase, interrupt. Do not wait
; for phase.

```

```

;*****
;Label for retry loop to resend id msg on error
;*****

```

retry:

```

MOVE FROM sendmsg, WHEN MSG_OUT ; The expected case after selection is id
; message-out to the device. Move the id
; message from the send message buffer.
; Do not wait for a phase change.
JUMP REL(retry), WHEN MSG_OUT ; If the target remains in the message-out phase
; after the initial messages have been sent to th
; device, retransfer the messages. Wait for a
; valid phase (req asserted).
JUMP REL(end), IF STATUS ; Now check for all expected phases.
JUMP REL(msg1), IF MSG_IN ; Process a message-in before the command
; phase here
INT err2, IF NOT CMD ; If it is not status, msg-in, or command, stop
; Interrupt if not command phase

```

cmd\_phase:

```

CLEAR ATN
MOVE FROM cmd_adr, WHEN CMD ; Transfer command bytes to the host
JUMP REL(msg2), WHEN MSG_IN ; Determine what is coming next. Is there a
; message-in after the command phase?
JUMP REL(end), IF STATUS ; Status phase after the command?
JUMP REL(input_data), IF DATA_IN; Check for data-in phase
JUMP REL(output_data), IF DATA_OUT ; Is this a data-out phase?
INT err3 ; Error — an unexpected phase after a commar
; transfer

```



```

; *****
; Label to process the status phase
; *****
end:
  MOVE FROM status_adr, WHEN STATUS ; Move the status byte in to the buffer area
; 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.

  INT check_cond, IF 0x02 ; Was there a check condition
  INT busy, IF 0x08 ; Is the device busy
  INT reserved, IF 0x018 ; Is the device reserved
  INT bad_status, IF NOT 0x00 ; Interrupt for unknown state
  INT err4, WHEN NOT MSG_IN ; Status value is good I/O, so process the
; command complete Stop if the next phase is
; not message-in.

  MOVE FROM rcvmsg, WHEN MSG_IN ; Message-in if here. It should be a command
; complete.
  INT err26, IF NOT 0x00 ; Process the message if it is not a command
; complete

; 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.

  MOVE SCNTL2 &0x7F TO SCNTL2 ; Allow the disconnect to occur.
  CLEAR ACK ; Command complete was received,
; acknowledge it
  WAIT DISCONNECT ; A physical disconnect should be next
  INT ok ; Good I/O if here

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

```

```

; *****
;
; Process what comes after the data transfer
; *****
check_out:
    JUMP REL(end), WHEN STATUS           ; Status phase is the normal next step
    JUMP REL(msg3), IF MSG_IN           ; Is there a message-in phase after data
                                        ; transfer
    INT err5                             ; Unexpected phase detected after data
                                        ; transfer

; *****
;
; This is the data-in phase portion of the algorithm
; *****
input_data:
                                        ; If a scatter/gather requirement exists, then thi
                                        ; 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 appropriatel
                                        ; at start I/O for the number of segments neede
                                        ; The overhead between segment block moves i
                                        ; 500-600 nanoseconds.
    MOVE FROM data_adr, WHEN DATA_IN
    JUMP REL(check_out)                 ; Go check the phase after data-in

; *****
;
; Process a message-in before the command phase
; *****
msg1:
    MOVE FROM rcvmsg, WHEN MSG_IN
    JUMP REL(ext_msg1), IF 0x01         ; Is this an extended message?
    INT err8, IF 0x02                  ; Is this save data pointers?
                                        ; Interrupt with ACKset.
    JUMP PASS(&disc_proc), IF 0x04     ; Is this a disconnect?
    INT err6                             ; Interrupt if any other message with ACK set
ext_msg1:
    MOVE SCNTL2 &0x7F TO SCNTL2       ; Message is an extended message
    CLEAR ACK                            ; Allow the disconnect to occur.
    MOVE FROM ext_buf, WHEN MSG_IN     ; Acknowledge the message just received
                                        ; Move two more messages into the buffer to g
                                        ; the extended message length and opcode for
                                        ; the processor to have available on the interrupt
    INT err7                             ; Interrupt the processor
discl:
    MOVE SCNTL2 &0x7F TO SCNTL2       ; Message is a disconnect
    CLEAR ACK                            ; Allow the disconnect to occur.
    WAIT DISCONNECT                     ; Acknowledge the disconnect message
    INT err9                             ; Disconnect before the command if here
                                        ; Interrupt the processor on a disconnect

```

```

; *****
; Message-in after the command phase
; *****

msg2:
  MOVE FROM rcvmsg, WHEN MSG_IN
  JUMP REL(ext_msg2), IF 0x01
  INT err10 IF 0x02
  JUMP REL(disc2), IF 0x04
  INT err11
ext_msg2:
  MOVE SCNTL2 &0x7F TO SCNTL2
  CLEAR ACK
  MOVE FROM ext_buf, WHEN MSG_IN
  INT err12
disc2:
  MOVE SCNTL2 &0x7F TO SCNTL2
  CLEAR ACK
  WAIT DISCONNECT
  INT err13

; *****
; Message-in after the data transfer phase
; *****

msg3:
  MOVE FROM rcvmsg, WHEN MSG_IN
  JUMP REL(ext_msg3), IF 0x01
  INT err14, IF 0x02
  JUMP PASS(&disc_PROCl), IF 0x04
  INT err15
ext_msg3:
  MOVE SCNTL2 &0x7F TO SCNTL2
  CLEAR ACK
  MOVE FROM ext_buf, WHEN MSG_IN
  INT err16
disc3:
  MOVE SCNTL2 &0x7F TO SCNTL2
  CLEAR ACK
  WAIT DISCONNECT
  INT err17

; Is this an extended message?
; Is this save data pointers?
; Interrupt with AC set.
; Is this a disconnect?
; Interrupt if any other message with ACK set
; Message is an extended message
; Allow the disconnect to occur.
; Acknowledge the message just received
; Move two more messages into the buffer to get
; the extended message length and opcode for
; the processor to have available on the
; interrupt.
; interrupt the processor
; Message is a disconnect
; Allow the disconnect to occur.
; Acknowledge the message
; Disconnect after the command if here
; Interrupt the processor on a disconnect

; Is this an extended message?
; Is this save data pointers?
; Interrupt with ACKset.
; Is this a disconnect?
; Interrupt if any other message with ACK set
; Message is an extended message
; Allow the disconnect to occur.
; Acknowledge the message just received
; Move two more messages into the buffer to get
; the extended message length and opcode for
; the processor to have
; available on the interrupt.
; Interrupt the processor
; Message is a disconnect
; Allow the disconnect to occur.
; Acknowledge the message
; Disconnect before the data transfer if here
; Interrupt the processor on a disconnect

```

```

; *****
;
; This is the section of code to process a reselect or select
; when a select I/O command was executed
; *****
resel_adr:
    WAIT RESELECT select_adr          ; Wait for reselect as the most probable event
    INT err18, WHEN NOT MSG_IN        ; The initiator was reselected, so process the
                                        ; possibilities
    MOVE FROM rcvmsg, WHEN MSG_IN     ; id message-in is the only expected SCSI phase
                                        ; here

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

    INT err19, WHEN DATA_IN          ; Data-in phase after reselection and id transfer
    INT err20, IF DATA_OUT           ; Data-out phase after reselection and id transfe
    INT err21, IF MSG_IN              ; Message-in phase after reselection and id
                                        ; transfer
    INT err22, IF STATUS              ; Status phase after reselection and id transfer
    INT err23, IF MSG_OUT             ; Message-out phase after reselection and id
                                        ; transfer
    INT err24                         ; Unknown phase after reselection and id
                                        ; transfer

; *****
; 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

```

**SCRIPTS for the Target Role**

```

=====
;
;
;
=====

```

**Definition Area TARGET ROLE**

```

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

```

```

ABSOLUTE initiator = 0x000 ; Buffer offset for the initiator id
ABSOLUTE msg_buf = 0x008 ; offset for count and address
ABSOLUTE cmd_buf = 0x010 ; Command byte offset for count & address
ABSOLUTE msg_buf2 = 0x018 ; Input message offset for count and address
ABSOLUTE data_addr = 0x020 ; user data buffer offset for count and address
ABSOLUTE stat_adr = 0x400 ; Status buffer offset for count and address

```

```

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

```

```

ABSOLUTE error1 = 0x0ff01
ABSOLUTE error2 = 0x0ff02 ; ATN is on after the id message is sent in to the
; initiator
ABSOLUTE error3 = 0x0ff03 ; ATN is on after the command bytes are sent to
; the initiator
ABSOLUTE error4 = 0x0ff04 ; Atn is on after the disconnect message is sent
; to the initiator
ABSOLUTE error5 = 0x0ff05 ; ATN on after id message sent to the initiator
; after a reselect operation is complete
ABSOLUTE error6 = 0x0ff06 ; ATN is on after user data is sent into the
; initiator
ABSOLUTE error7 = 0x0ff07 ; ATN is on after the status byte is sent
ABSOLUTE error8 = 0x0ff08 ; ATN is on after the command complete
; message is sent

```

```

ENTRY start_up ; Entry Point for the target role

```

**; SCRIPTS AREA**

```

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

```

```

start_up:
    WAIT SELECT rel(resel_adr) ; First wait for a selection by the initiator and
                                ; jump to the
                                ; alternate address if reselected.
retry_id:
    MOVE FROM msg_buf, WITH MSG_OUT ; Move the id message into the message buffer

    JUMP Rel(id_atn), IF ATN ; If the initiator sets ATN, go process that
                                ; condition
continue_id:
    MOVE FROM cmd_buf, WITH CMD ; Move the command bytes in to the target
                                ; buffer
    JUMP REL(cmd_atn), IF ATN ; Note that though a one is in the command
                                ; count field, the chip will automatically
                                ; transfer in the correct number of bytes
                                ; based on the SCSI command opcode.
                                ; If the initiator sets ATN, go process that
                                ; condition
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.

    MOVE FROM msg_buf2, WITH MSG_IN ; Send in the disconnect message
    JUMP REL(disc_atn), IF ATN ; If the initiator sets ATN, go process that
                                ; condition
continue_disc:
    DISCONNECT ; Now get off the bus

; *****
; Entry point for reselecting the initiator
; *****
resel_in:
    RESELECT FROM initiator REL(resel_adr) ; Perform the reselect and jump to resel_adr if
                                ; reselection happens while trying to do the
                                ; reselect
retry_resel:
    MOVE FROM msg_buf2, WITH MSG_IN ; Move the reselect id message into the initiato
    JUMP REL(resel_atn), IF ATN ; If the initiator sets ATN, go process that
                                ; condition
continue_resel:
    MOVE FROM data_adr, WITH DATA_IN ; Now move the data bytes into the initiator

```

JUMP REL(data\_atn), IF ATN

; 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.

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.
; *****

```

MOVE FROM stat\_adr, WITH STATUS  
JUMP Rel(stat\_atn), IF ATN

; Now move in the status byte  
; If the initiator sets ATN, go process that  
; condition

continue\_stat:

MOVE FROM msg\_buf2, WITH MSG\_IN  
JUMP REL(cc\_atn), IF ATN

; Move the command complete message-in  
; If the initiator sets ATN, go process that  
; condition

continue\_cc:

DISCONNECT

; Now physically 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 id message comes
;out, this is the code for processing what comes next.
;*****

```

```

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

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

INT error2

; All the ATN subroutines have the same basic
; function

```

```

cmd_atn:
  MOVE FROM msg_buf, WITH MSG_OUT
  INT error3

```

```

disc_atn:
  MOVE FROM msg_buf, WITH MSG_OUT
  INT error4

```

```

resel_atn:
  MOVE FROM msg_buf, WITH MSG_OUT
  INT error5

```

```

data_atn:
  MOVE FROM msg_buf, WITH MSG_OUT
  INT error6

```

```

stat_atn:
  MOVE FROM msg_buf, WITH MSG_OUT
  INT error7

```

```

cc_atn:
  MOVE FROM msg_buf, WITH MSG_OUT
  INT error8

```



# Unique Initiator Sequences for the 53C720

## Disk Drive Initiator Sequence

Arbitrate and Select with ATN  
 Transfer the id message  
 Transfer the command bytes  
 Accept the message-in — DISCONNECT  
 Reselected — id message-in  
 Data transfer of 1 - 4 user data blocks  
 Accept SCSI status byte, COMMAND  
 COMPLETE message and wait for  
 bus free

### 53C720 Strengths in the Disk Drive Environment

- A large number of commands are typically issued to the disk, and the 53C720 offers very little SCSI bus overhead and a minimum of time to initiate an I/O in the host computer.
- The 53C720 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)
  - Use the interrupt on the fly (INTFLY) instruction to signal the system processor that the current I/O is complete.
- The 53C720 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 53C720 can be programmed to absorb this message with no interrupt to the external processor.

- The 53C720 can process a disconnect message from the disk without interrupting the system processor. See the Chapter "Multi-Tasking I/O Using SCSI SCRIPTS" for a complete description.

- Because there can be a requirement for a very high performance system disk driver, a minimal algorithm can be developed that requires only a small number of SCSI SCRIPTS. Other disks can use more complex SCRIPTS. The designer can decide where to put the I/O logic (in firmware or in SCRIPTS) using this architecture.

## Tape Drive Initiator Sequence

Arbitrate and Select with ATN  
 Transfer the id message  
 Transfer the command bytes  
 Accept the message-in — DISCONNECT  
 Reflected — id message-in  
 Data transfer of 16K of user data  
 Accept the message-in — SAVE DATA  
 POINTERS followed by DISCON-  
 NECT.  
 Reselected — id message-in  
 Data transfer of 16K of user data  
 \*  
 \*  
 \*  
 Reselected — id 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 53C720 could be programmed specifically for this system. Knowing the tape drive was alone on the bus, the 53C720 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 53C720 interrupts to allow the external processor to restart the tape I/O.

The 53C720 allows systems integration designs using the SCSI bus with no performance impact to I/O throughput. See the Chapter "SCRIPTS SCSI Use of Scatter/Gather" for another possible algorithm for large blocks of data that use a SCSI SCRIPTS loop.

### **SCSI Character Oriented Device in the Initiator Role**

A SCSI port can be dedicated by the system designer for terminal control. First, a SCSI read command is transferred to the target terminal controller. A stream of user data typed in at the terminals, plus the inserted control bytes in the stream comes back to the initiator. A SCRIPT can be written which looks at the byte stream coming in and sends line control bytes to the processing buffer and data bytes to the data buffer. When certain control bytes are received, the 53C720 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 53C720 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 53C720 can be used in unusual areas to offload any processor and improve performance.

## Special SCSI SCRIPTS Situations

### Transferring Large Blocks of User Data

#### Case 1

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

The Block Move command was developed to transfer large amounts of user data, but anomalies such as an unexpected phase change after transferring 16K of the data, must be handled by the processor.

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

- 1) Clean up the chip on Data-Out Phase using the external processor
- 2) Change the data address and byte count in the active SCSI SCRIPTS or the Indirect Data Table, using SCRIPTS on the processor
- 3) Receive the message byte via SCSI SCRIPTS and make the appropriate changes for the subsequent reselect.

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

#### Case 2

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

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

- Block Move of burst size.
- Call subroutine (after waiting) if the

next phase is not a data phase. (The subroutine should process the SAVE DATA POINTERS message in and return.)

- Block Move of burst size
- Call subroutine (after waiting) if the next phase is not a data phase.

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

#### Case 3

*The expected burst size is NOT known ahead of time.*

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

Note that the overall penalty of this situation is not great for many SCSI devices, because the unexpected phase change is a low probability situation. When the interrupt occurs, the external processor decodes the chip status (two register reads) and then loads in the appropriate SCRIPT address for handling data-in or data-out.

## Processing a SAVE DATA POINTERS Message

### 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 53C720 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 53C720. Save a pointer to this SCSI SCRIPTS in the system I/O structure so that the I/O can easily be re-scheduled. 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 53C720 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 53C720 will fetch the next command and start processing it. When the phase change actually occurs, the 53C720 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 a SCRIPT is executed 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.

## Multi-Tasking I/O Using SCSI SCRIPTS

### Multi-Threaded I/O Using SCSI SCRIPTS

A design goal of the 53C720 is to allow the user to perform multi-threaded I/O with no external processor intervention.

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

- Main SCSI SCRIPTS
- Scheduler SCSI SCRIPTS
- Disconnect SCSI SCRIPTS
- (Reselect) Resume SCSI SCRIPTS

All are involved during multi-threaded I/O. Some of the command areas must be written by the 53C720; thus, some script code must be stored in random access memory (RAM).

#### Main SCSI SCRIPTS

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

A context switch from one I/O to another is performed by loading the Table Indirect Data Structure Address into the Data Structure Address (DSA) register and then loading the SCSI SCRIPTS entry point into the 53C720 (a JUMP instruction).

Note that the entry point address is loaded with a simple transfer control (JUMP or CALL) instruction. Because a SCSI SCRIPTS Memory to Memory MOVE can load the DSA address, and the chip can perform a JUMP SCRIPT, the context switch can easily start an I/O or begin a new I/O or switch to a different one. In the Main SCSI SCRIPTS, numerous *resume* points exist. When coding the algorithm, each resume point must be identified as the script

is coded. An answer to the question “*If a disconnect message arrived from the target, where must the I/O resume?*” must be known throughout the Main SCSI SCRIPTS. In the following paragraphs, which discuss multi-threaded I/O, the importance of this major point will become quite clear.

#### Scheduler SCSI SCRIPTS

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

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

or:

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

An I/O is scheduled when the system processor writes an entry to the Scheduler. The 53C720 driver routines must identify an unused entry in the Scheduler SCSI SCRIPTS and move a pointer to the data structure into the appropriate memory address of the unused entry. Then a JUMP command must be written to the next line of code. When the 53C720 has no more SCSI SCRIPTS to execute for an I/O, it will jump to the Scheduler SCSI SCRIPTS. For a scheduled I/O, the value at a memory address will be moved into the DSA register and then the chip will transfer to the main SCRIPT entry point. A NOP is then written to the jump just taken so that the same I/O will not be restarted by the 53C720 before it completes. Because the system will not re-use the entry until the I/O is complete, the I/O runs until completion. If there are no I/Os scheduled, the 53C720 should interrupt or wait for reselect if outstanding I/Os exist.

To conserve RAM space, there may be fewer

entries in the Scheduler. Once the NOP is written by the 53C720, the entry can be reused. Then the number of entries is the maximum number of I/O's scheduled but not started. After the Select with ATN SCRIPT, the scheduler entry is no longer needed. Using a MOVE memory instruction, a NOP can be written to the scheduler entry just executed, leaving it open for the system to reuse.

### Disconnect SCSI SCRIPTS

The target device can change phases on the SCSI bus at any time to save state or to disconnect temporarily. If a MOVE command is executing during a phase change and the byte count is not zero, an external interrupt occurs. However, if the 53C720 has completed the move operation, no external interrupt is required and the chip can handle the phase change using SCSI SCRIPTS. To automatically process this phase change, the programmer must identify the resume points in the SCSI SCRIPTS as the algorithm is being developed.

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

#### Address SCRIPT

- 16 write synchronous values to 53C720
- 8 Jump to the resume point
- 0 Label: move 4, SCRATCH, Label-4
- +8 Jump Scheduler
- +16 I/O data structure values

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

To implement the disconnect, determine the necessary action if a disconnect message

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

```

Jump resume1
    ; jump around the resume label

resume1_base:
    ; Place the resume address in TEMP

Call save_resume

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

```

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

```

ABSOLUTE TEMP = 0xde01c

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

    Move Memory 4, TEMP, SCRATCH

    ; the resume address is now in
    ; SCRATCH

```

Next, the resume address must be written to memory by the 53C720. At the address pointed to by the DSA register is a Memory MOVE command that moves the value (now the resume point) into the second four bytes of the JUMP command, eight bytes above. The next

step is for the SCRIPT to jump to the address in the DSA register.

```
Move 4, DSA, TEMP
; Move contents of DSA to TEMP
```

```
Return
; now Return to the data structure
```

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

### Resume SCSI SCRIPTS

In SCSI terminology, the *nexus* is a combination of device id, logical unit number, and queue tag value. Upon reselection, the 53C720 will decode the nexus, using COMPARE and JUMP SCSI SCRIPTS instructions. Upon reselection, the device id is in the SFBR or optionally in the Longitudinal Parity Register (SLPAR).

After a series of COMPARE and JUMP instructions, based on the unique nexus value, the 53C720 will transfer to a unique Memory MOVE command.

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

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

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

```
set_up:
Move 4, DSA, TEMP
Move TEMP0 -16 to TEMP0
Return
```

At the data structure, minus sixteen is an instruction that writes the synchronous offset and period to the 53C720; there is then a jump to the resume point.

Upon completion of an I/O, the programmer may want to signal the system processor by one of several mechanisms allowed by the 53C720:

1. Execute an interrupt instruction.
2. Execute an interrupt on the fly.
3. Write a value to system memory. Termination is unnecessary; yet the processor must poll a software semaphore. With some periodic I/O timer interrupt followed by a read of I/O status areas, this method can work well.
4. Set the semaphore as in 3), but then write to a user-defined pin (first on, then off) to cause an external interrupt. This allows completely interrupt-driven I/O software.
5. Set the semaphore as in 3) and then execute an interrupt on the fly. Compared to a system interrupt, fetching SCRIPTS is very fast. More importantly, the programmer is in control of the tradeoffs and can allow the processor more or less work depending on requirements. If system bus latencies are large, then SCRIPTS can also be stored in local memory on a host bus adapter to eliminate the fetch times. There are enough optional features in the 53C720 to allow optimization of many configurations.

**Notes**

A large, empty rectangular box with a thin black border, occupying most of the page. It is intended for the user to write notes.



# SCSI SCRIPTS Machine Language Description

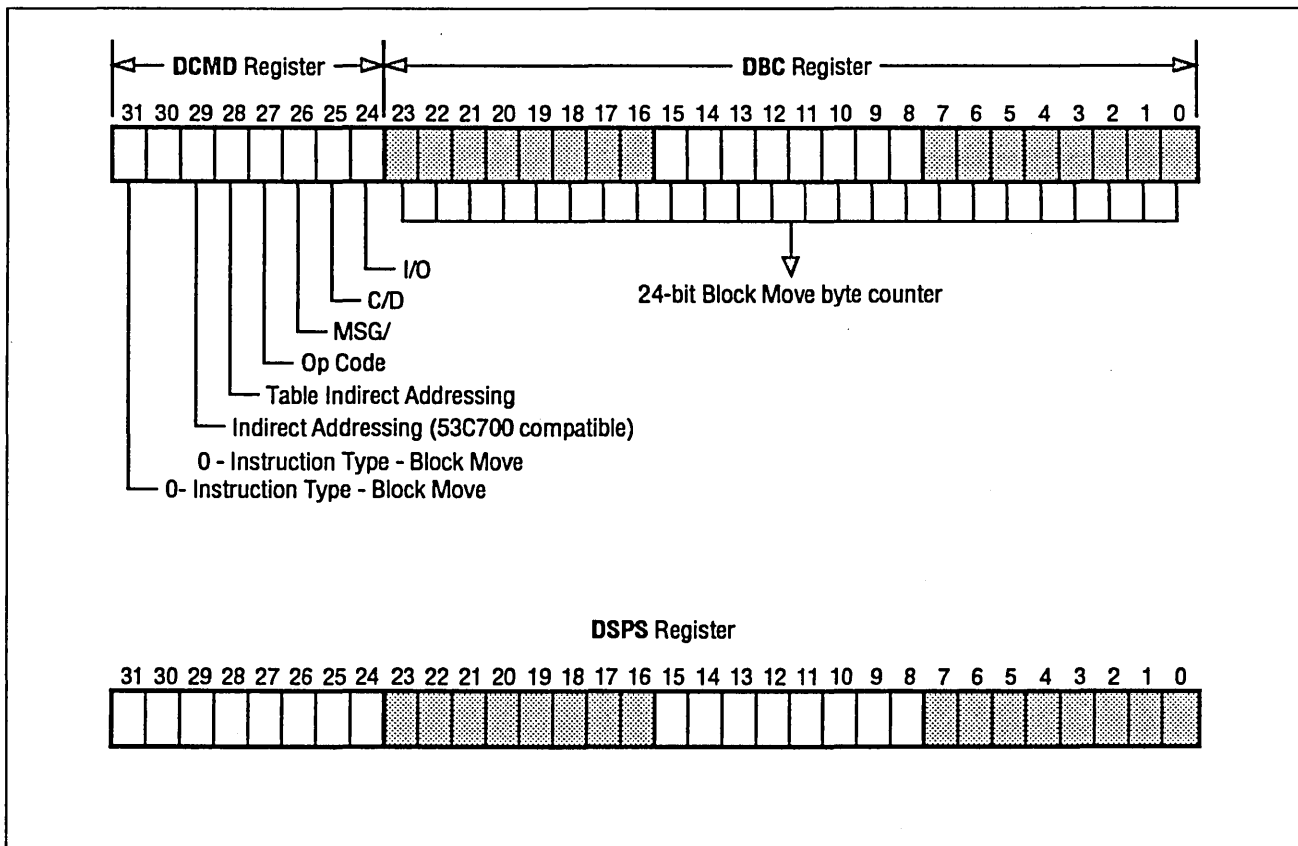
This chapter describes in detail each SCSI SCRIPTS instruction. Normally, you will use the SCSI SCRIPTS compiler as described in the previous chapters for programming the 53C720, but for debugging purposes, this chapter contains a detail description of each instruction. Each instruction consists of a bit diagram, a brief overview, and a description of each field within the instruction.

The 53C720 implements four types of instructions; Block Move, I/O or Read/Write, Transfer Control, and Memory Move. Bits 31-30 of the first word of each instruction define the SCSI I/O Processor instruction type. Depending on instruction type, bits 29-0 have different functions. For example, if bits 31-30 = 01 (I/O or Read/Write), bits 29-27 becomes opcode bits defining either I/O or Read/Write.

Opcode	Instruction Type
00	Block Move Instruction
01	I/O Instruction or Read/Write
10	Transfer Control Instruction
11	Memory Move Instruction

Each instruction consists of two or three 32-bit words. The first 32-bit word is always loaded into the DMA Command (DCMD) register and the DMA Byte Counter (DBC) register and the second 32-bit word is loaded into the DMA SCRIPTS Pointer Save (DPS) register. The third word, only used by the Memory Move instruction, is loaded into a shadowed Temporary Stack (TEMP) register.

## Block Move Instructions



## Block Move Overview

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

Note that the data may come from any memory address, so Scatter/Gather operations for user data are transparent to the chip and the external processor. One simply writes a separate Block Move for each piece of data to be moved. Use the 64-byte DMA data buffer to speed data transfers between user memory and the I/O Processor. Synchronous SCSI data-in transfers uses the 8-byte FIFO.

When the 53C720 executes several Chained Move (CHMOV) instructions and one ends on an odd byte boundary, the 53C720 temporarily stores the residual byte. It then takes the first byte from the subsequent CHMOV or MOVE instruction and line it up with the residual byte in order to complete a wide transfer and maintain a continuous data flow on the SCSI bus.

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

## Block Move Instruction

(First SCRIPTS Word)

### Bits 31-30 Block Move (00)

An instruction type of 00 equates to a Block Move.

### Bit 29 Indirect data address flag

#### Bit 29 = 0 Direct Addressing

SCSI data 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.

#### Bit 29 = 1 Indirect Addressing

The 32-bit SCSI data 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 DSPS 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 were 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 32 bit address is fetched from memory and data is transferred from this address. This allows the logical I/O driver to build a structure of addresses for an I/O rather than treating each address individually.

### Bit 28 Table Indirect Field

#### Bit 28 = 0 Table Direct Mode

SCSI or user data is moved as described previously. This option allows compatibility with existing 53C700/710 SCSI SCRIPTS.

#### Bit 28 = 1 Table Indirect Mode

The 32-bit start address is treated as a 24-bit signed value. After the instruction is moved into the 53C720, the 24 bits are added to the Data Structure Address (DSA) register to form a 32-bit physical address.

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

There are several programming implications of this feature.

First, a standard SCSI data structure can be designed with values at predefined offsets. The SCSI SCRIPT does not require the actual 32-bit address or 24-bit count to be in the SCRIPT itself. At the start of the an I/O,

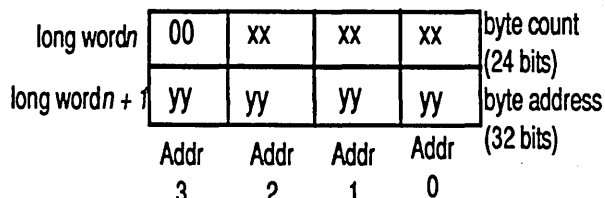
once the actual data structure is built, no more firmware intervention is required except loading the data table base address into the DSA register.

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

Finally, there is a requirement for only one copy of the main SCSI SCRIPT for all I/O, using a fast context switch to change to another I/O. Only the data structure is unique to each I/O, and the SCRIPT is re-entrant.

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

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



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

**Bit 27 Block Move Opcode**

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

**Target Mode Bit 27 = 0 (MOVE)**

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

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

If the Indirect Addressing bit is 1, the 53C720 fetches the Data Buffer Start Address from the location pointed to by the DMA Next Address (DSPS) register. This fetched value is then stored in the DSPS, and execution begins.

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

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

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

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

- Wait for the first byte received.
- Decode the byte to determine the number of SCSI instruction bytes to receive
- Write the instruction 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 instruction bytes.

- Transfer the correct number of bytes into the address designated by the Block Move instruction.

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

**Target Mode Bit 27 = 1 (CHMOV)**

Refer to the Chained Move Feature discussion that follow.

**Initiator Mode Bit 27 = 0 (CHMOV)**

Refer to the Chained Move Feature discussion that follow.

**Initiator Mode Bit 27 = 1 (MOVE)**

In the initiator mode, this operation waits for a valid phase and DMA data. After verification that the previous instruction is complete or a reselect has occurred, the chip waits for a previously unserviced phase before executing the Block Move instruction. You can program the 53C720 to pause until the SCSI device it is communicating with goes to the next phase, using the Transfer Control instructions or the Move instructions.

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

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

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

**Bits 26-24 SCSI Phase Lines**

These three SCSI phase lines are used to compare to the actual SCSI bus phase lines. The SCSI bus phase value is latched when REQ goes active. The value is stored in

SSTAT1 (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. The following table describes the possible combinations and the corresponding SCSI phase.

MSG	C/D	I/O	SCSI Phase
0	0	0	Data Out
0	0	1	Data In
0	1	0	Command
0	1	1	Status
1	0	0	Reserved Out
1	0	1	Reserved In
1	1	0	Message Out
1	1	1	Message In

**Bits 23-0 Block Move Byte Count**

This count value specifies the exact number of data bytes to be moved between the SCSI bus and system memory. As the SCSI SCRIPTS instruction 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 instruction will be fetched.

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

## Block Move Instruction

(Second SCRIPTS Word)

### Bits 31-0 Data Start Address

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

The Block Move instruction is very powerful for several reasons.

- No distinction is made between user data and SCSI instruction, message, or status data.
- Data can be stored in any area of system memory with little performance impact (one instruction fetch) to switch data buffer addresses.
- The indirect feature allows a table of addresses in stead of requiring the address to be in the instruction.
- A Scatter/Gather operation has little performance impact, because the only overhead is 500 nanoseconds (direct mode) or 750 nanoseconds (indirect mode). Thus, one Block Move instruction for each segment of data-in memory is economical with the SCSI I/O processor architecture.

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

In the initiator mode, 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 instruction. If the phase does not match, then an external interrupt is generated.

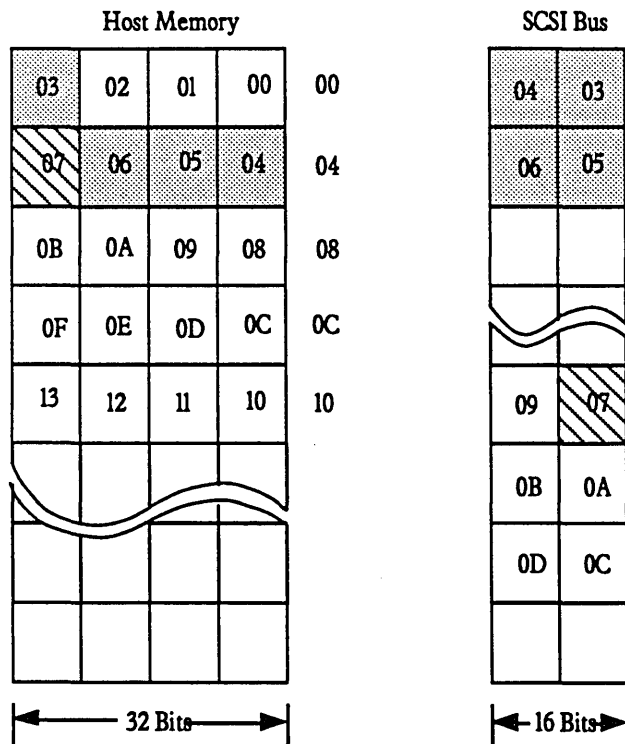
For the high performance SCSI SCRIPTS algorithm, exceptions are abnormal and are handled by the external processor. Normally, the Conditional Transfer instruction (see I/O Instruction) compares actual to expected phase before executing any Block Move. The first Conditional Transfer instruction must have the “wait” option on (to synchronize the instructions with the actual bus phase), and each subsequent instruction should have the “wait” option turned off.

With the Table Indirect mode, I/O data structures can be fetched directly, eliminating one more level of system software translation normally required to start an I/O. In this mode, SCRIPTS do not need to be patched at the start of an I/O. Once a standard I/O descriptor has been built by a SCSI SCRIPT, the 53C720 can effectively execute the data structure without processor intervention.

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

## Chained Move Instruction

Because Wide SCSI transfers two bytes at a time across the SCSI bus, rather complicated combinations of DMA and SCSI odd byte transfers can be envisioned. Because the 53C720 supports arbitrary DMA Scatter/Gather, all odd byte handling must be resolved inside the 53C720, by the DMA portion of the chip. A Chained Move SCRIPT instruction (CHMOV) was defined to solve the odd byte problem. The following examples illustrate the possibilities. The following example is from a system memory perspective.



- Ten byte transfer with two five byte pieces (two block moves of 5 bytes each)
- First four bytes of data go onto the SCSI bus
- Fifth byte is saved in the chip, because a Chained Move is used
- A second Move SCRIPT is fetched by the 53C720 (a regular Move)
- The sixth byte is match with the fifth and sent out to the SCSI bus.
- The last 4 bytes are then sent out .

Note that if an odd byte count is used and the move is not in a series of Moves then the odd byte is sent to the SCSI target which must disregard the last byte, based on the original byte count associated with the SCSI instruction.

When moving data through the 53C720 and odd byte transfers is a possibility (from the SCSI bus or system bus) all but the last instruction should be a Chained Move instruction. The cases are:

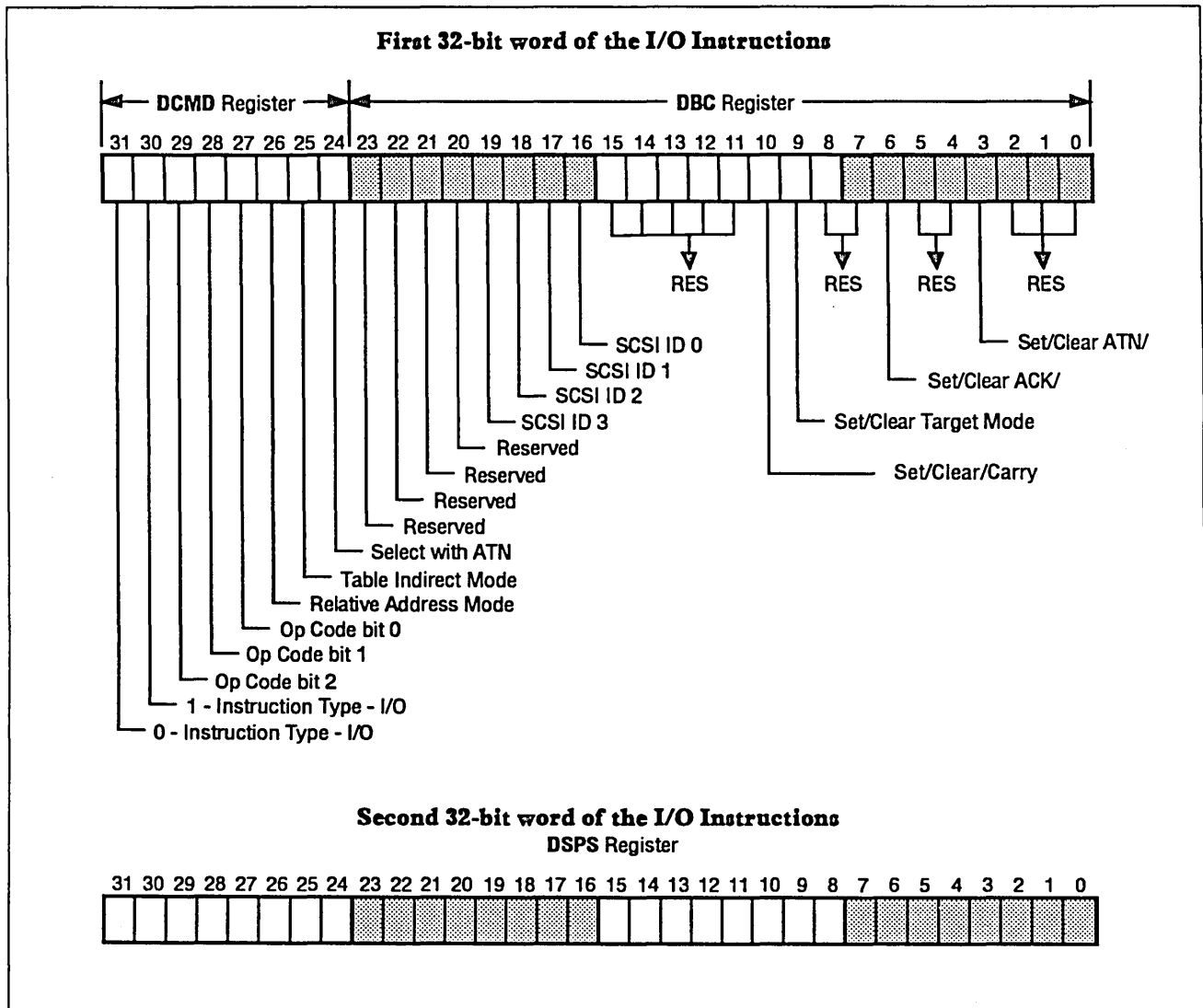
#### 1. Data-Out From System Memory.

Chained Moves handle all the intermediated moves, by saving an odd byte in the chip until the subsequent move is fetched. The last move is not chained, so it will match up with any byte in the chip at start of execution, but if one exists at the end, the byte will be sent on to the target device, which knows which bytes are good by the original byte count.

#### 2. Data-In To System Memory.

Two odd byte situations exist for this case. First, the byte count in a Scatter/Gather list may be odd, so a byte will be held in the chip until the next move is fetched. This implies that a Chained Move instruction must also be used until the last Scatter/Gather list entry when a non Chained Move must be used. On the last move, there may be a SCSI odd byte situation in which the target changes phases to Message-In and sends the Ignore Wide Residue Message. This non wide transfer will cause the 53C720 to drop the odd byte and continue executing SCSI SCRIPTS. The SCSI SCRIPTS algorithm must be prepared to verify that after an odd byte transfer that the next step is a message of Ignore Wide Residue from the target. In the target mode, the message must be sent any time one byte of user data is sent to the initiator (Data-In). On a Data-Out phase, the target will use the byte count to determine whether the last byte is good data.

## I/O Instructions



### I/O Instructions Overview

The I/O instruction performs SCSI operations such as select and reselect. Each function defined is a direct instruction to the SCSI portion of the 53C720. The functions vary if the chip is in the target or initiator mode, so that the functions are described separately for each mode.

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

### I/O Instructions

(First SCRIPTS Word)

#### Bits 31-30 I/O Instruction (01)

An instruction type of 01 equates to an I/O Instruction or Read/Write. Bits 29-27 define I/O instruction or Read/Write operation.

#### Bits 29-27 I/O Instruction Opcodes

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

**Target Mode Bits 29-27 = 000 (Reselect)**

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 uses the 32-bit jump address value to fetch the next instruction and begin execution at that address.

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

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

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

**Target Mode Bits 29-27 = 001 (Disconnect)**

The chip physically disconnects from the SCSI bus.

**Target Mode Bits 29-27 = 010 (Wait Select)**

The chip waits for a SCSI selection by another device on the SCSI bus. If the chip is already selected, then the next SCSI SCRIPTS is fetched and executed. When a bus initiated interrupt or reselect occurs, the chip optionally changes to the initiator mode and fetches the next instruction from the

address pointed to by the 32-bit jump address, and continues execution.

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

**Target Mode Bits 29-27 = 011 (Set)**

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 mode.

**Target Mode Bits 29-27 = 100 (Reset)**

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 mode.

**Initiator Mode Bits 29-27 = 000 (Selection)**

The chip arbitrates for the SCSI bus and then performs a selection. Arbitration continues until the chip is successful or a bus initiated interrupt (e.g., reselection) occurs. If arbitration terminates because of a bus initiated interrupt (as a result of a select or reselect), the chip uses the 32-bit jump address to fetch the next instruction and begin execution at that address.

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

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

If the Table Indirect mode bit is 1, the 24-bit signed value in the DMA Byte Count (DBC) register is used as an offset relative to the Data Structure Address register. The SCSI destination device ID, the synchronous offset, the synchronous period, the wide/narrow bit, and the clock conversion bits are loaded from the formed address. Using this indirect mode, the SCRIPTS program can set the values stored with the I/O data structure and not require the user



to alter SCRIPTS instructions at the start of an I/O. Upon reselect, the synchronous offset, the synchronous period, the clock conversion and the wide/narrow bit can be set using register writes, with no need to cause an external interrupt.

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 modes 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 mode) instruction. That instruction's alternate address is the reselection algorithm (initiator mode). The 53C720 can determine exactly what happened and transfer control to the appropriate SCSI SCRIPTS algorithm. See Appendix C for another solution to this problem.*

**Initiator Mode Bits 29-27 = 001 (Wait Disconnect)**

The initiator waits for a disconnect from the SCSI bus. In the SCSI Control #2 register, there is the disconnect bit that can be set to allow any disconnect to be legal and not cause an interrupt. If it is reset, every disconnect (loss of busy and select for the specified bus free time) causes an interrupt. The user must alter the disconnect bit when legal disconnects are expected, and change it back so any disconnect should be illegal. If the SCSI bus goes bus free and then the 53C720 is reselected before the Wait for Disconnect SCRIPT is fetched, an infinite wait will not occur.

**Initiator Mode Bits 29-27 = 010 (Wait Reselect)**

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 53C720. 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 mode) instruction. That instruction's alternate address is the error recovery algorithm (for initiator mode — reselect). The chip can determine exactly what happened and transfer control to the appropriate SCSI SCRIPTS algorithm.

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

*Note: With the 53C720 byte compare capability of the transfer control instruction, 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. Examine the SCSI selector valid ID bit (SSID, bit 7) which automatically sets when two SCSI ID's are detected on the bus during a bus-initiated selection or reselection. The encoded destination ID bits (SSID, bits 3-0) contain the ID of the initiator, selecting the 53C720, or the ID bit of the target, reselecting the 53C720. See "Multi-tasking I/O" for more discussion of this subject.*

**Initiator Mode Bits 29-27 = 011 (Set)**

The chip asserts the SCSI bus bits requested in the flags field. Currently four bits are defined, allowing the SCSI ACK, target mode, Arithmetic Carry and ATN bits to be set. Bit 10 is for the Arithmetic Carry, bit 9 is for target, bit 6 is for Acknowledge, and bit 3 is for Attention.

**Initiator Mode Bits 29-27 = 100 (Reset)**

The chip resets the SCSI bus bits requested in the flags field. Currently four bits are defined, allowing the SCSI ACK, carry bit, target mode and ATN bits to be reset. Bit 10 is for the Arithmetic Carry, bit 9 is for target, bit 6 is for Acknowledge, and bit 3 is for Attention.

Note that these bits can also be set or reset with the read/write register functions, except for the Arithmetic Carry bit which can not be (re)set directly by writing a register.

### Bit 26 Relative Addressing Mode

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

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

### Bit 25 Table Indirect Mode

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

00	Device ID	Peroid&Offset	00
----	-----------	---------------	----

Byte	Byte	Byte	Byte
Lane	Lane	Lane	Lane
3	2	1	0

Information in byte lane 3 is mapped into the SCSI Control 3 (SCNTL3) register (03). Device ID is mapped into the SCSI destination ID (SDID) register (02), and period and offset is mapped into the SCSI Transfer (SXFER) register (05).

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

If the four bytes are written from the processor into memory as a unit (one long word), then the user need not be concerned about Big or Little Endian mode. The low order

byte must be at lane byte zero, next byte at lane byte one, and so forth.

The SCNTL3 register contains Synchronous Clock Conversion Factor (SCF2-0), Enable Wide SCSI (EWS), and Clock Conversion Factor (CCF2-0). The SDID register contains Enable Response to Reselection (RRE), Enable Response to Selection (SRE), and Encoded 53C720 chip SCSI ID. The SXFER register contains SCSI Synchronous Transfer Peroid (TP2-0), and Max SCSI Synchronous Offset (MO3-MO0).

### Bit 24 SELECT With ATN

If bit 24 is set, then the initiator SELECT instruction 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.

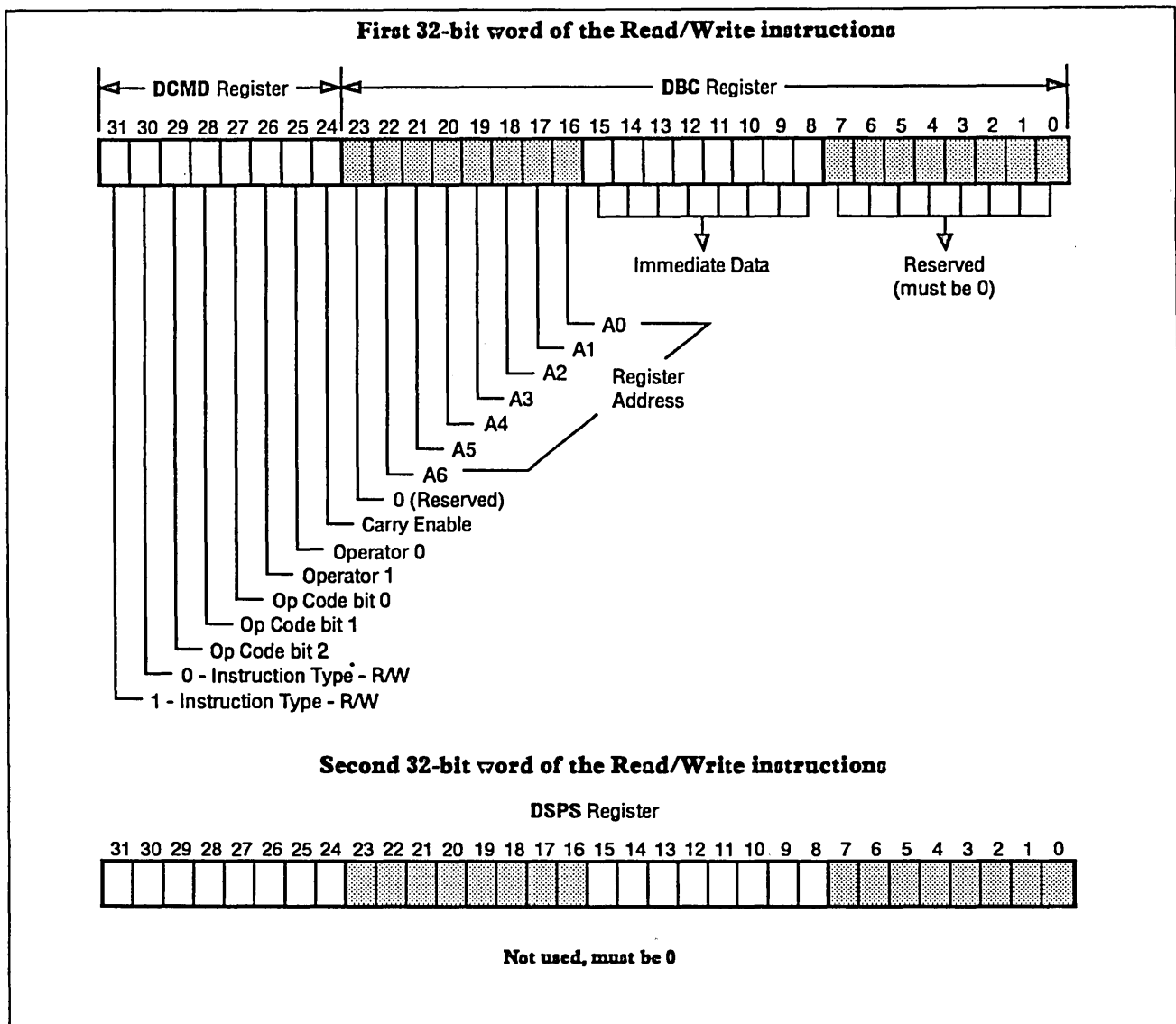
### Bits 23-16 SCSI ID 7-0

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

### Bits 15-0 Flags Field

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

### Read/Write Register Instructions



### Read/Write Register Instructions

(First SCRIPTS Word)

#### Bits 31-30 Read/Write Instructions (01)

An instruction type of 01 equates to a Read/Write or I/O Instruction. Bits 29-27 define Read/Write operation or I/O instruction. Opcode bits of 101, 110, and 111 are for Read/Write operations. Read/Write operations are modified by operator bits 26-25. Operator bits define four modes.

#### Read/Write Overview

In either initiator or target mode, the opcode bits 29-27 (opcode 101, 110, and 111) are for a set of register operations. The three opcodes are modified by the operator field (bits 26-25). The opcode bit operations are:

**Bits 29-27 = 101 (Move from SFBR)**

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

Bits 26-25 = 00

Move immediate data value to the destination register value.

Bits 26-25 = 01

OR the immediate data value with the SFBR, and write the result to the destination register.

Bits 26-25 = 10

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

Bits 26-25 = 11

Add the immediate data value with the SFBR and write the result to the destination register with or without carry.

**Bits 29-27 = 110 (Move to SFBR)**

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

Bits 26-25 = 00

Move immediate data to the SFBR

Bits 26-25 = 01

OR the immediate data value with the specified register and write the result to the SFBR

Bits 26-25 = 10

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

Bits 26-25 = 11

Add the immediate data value to the specified register and write the result to the SFBR with or without carry.

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

(Bits 26-25) Operator Field	(Bits 29-27) Opcode 7 (111) Read modify Write	(Bits 29-27) Opcode 6 (110) Move to SFBR	(Bits 29-27) Opcode 5 (101) Move from SFBR
00	Immediate data to destination register	Immediate data to SFBR	Immediate data to destination register
01	Immediate data or'ed with destination register	Immediate data OR register to SFBR	Immediate data or'ed with SFBR to register destination
10	Immediate data and'ed with register	Immediate data AND register to destination register	Immediate data and'ed with SFBR to destination SFBR
11	Immediate data added to destination register with or without carry	Immediate data added with register to SFBR with or without carry	Immediate data added SFBR to destination register with or without carry

**Bits 29-27 = 111 (Read-Modify-Write)**

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

Bits 26-25 = 00

Move immediate data to the specified register.

Bits 26-25 = 01

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

Bits 26-25 = 10

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

Bits 26-25 = 11

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

**Bit 24 Carry Enable**

When this bit is set it allows the previous carry value to be used by the present add instruction. The carry value remains intact unless it is modified by an add, set carry or clear carry instruction. All other instructions do not affect carry. If carry Enable is not set, no carry in will be used during the present add instruction.

**Bit 23 Reserved****Bits 22-16 Register Address Field**

These bits select one of the 8-bit registers in the 53C720 to serve as source, destination, or immediate register.

**Bits 15-8 Immediate Data Field**

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

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

instructions that may be defined.

Having a read/write register capability in the 53C720 adds a new dimension of SCRIPTS programming capability.

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

1. Set synchronous offset and period for a target upon reselection. This operation will typically require an interrupt to an external processor. A 53C720 SCRIPT will be able to write an immediate value to the correct register once the reselecting device ID is decoded, and resume data transfer immediately.

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

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

Many other uses can be discovered. With the 53C720, a user can write a SCRIPTS program that will perform most of the operations done in external processor firmware.

**Bits 7-0 Reserved**

These bits should always be zero.

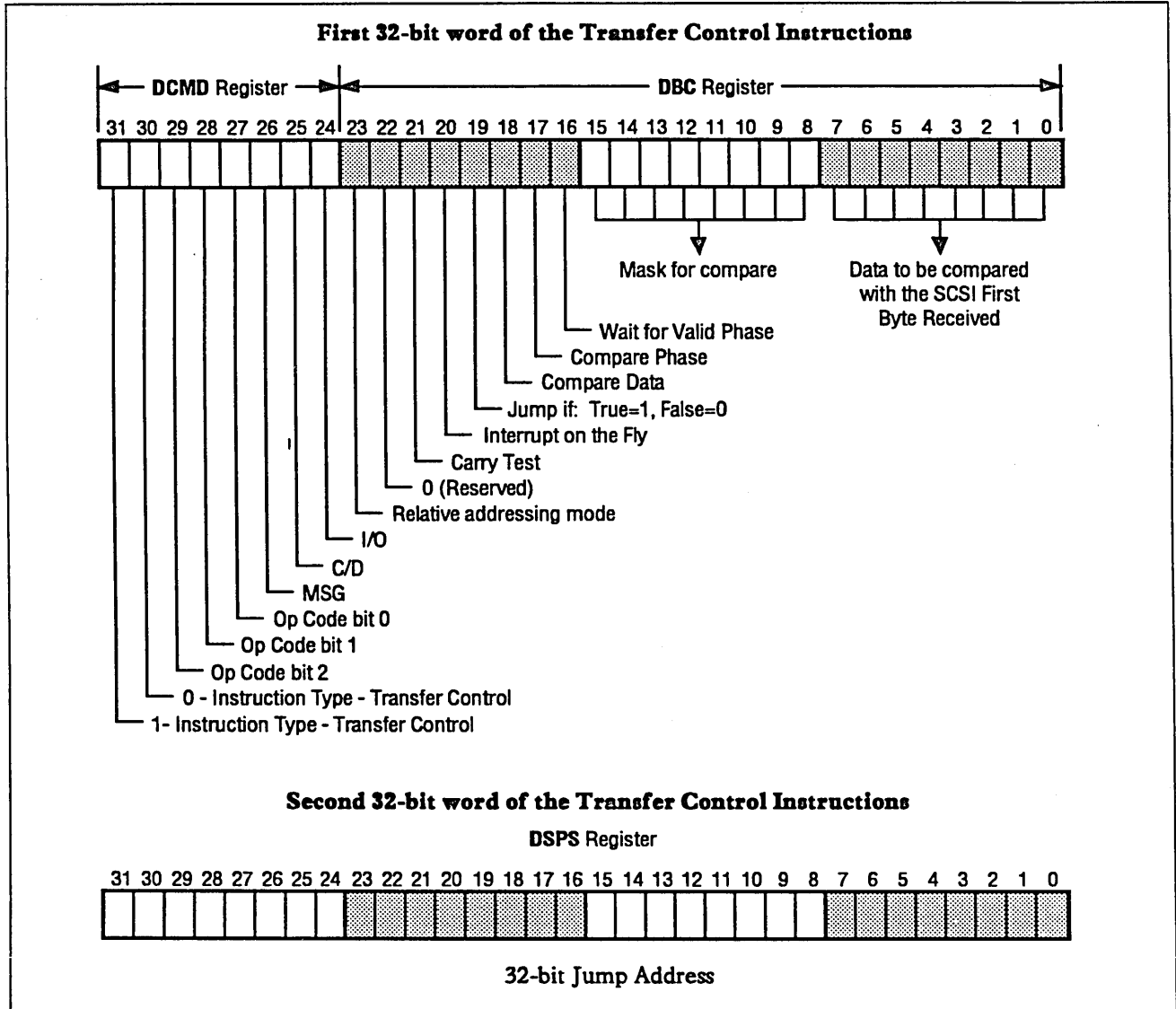
**I/O Instruction**

(Second SCRIPTS Word)

**Bits 31-0 Jump Address**

If the select, wait reselect, or reselect instruction 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 Instructions



## Transfer Control Overview

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

The Transfer control instruction 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 instructions allow SCSI algorithms to be written in SCSI SCRIPTS and give the 53C720 characteristics of a general purpose SCSI processor. With transfer control instructions, you can program the chip, rather than simply buffering instructions to be serially executed with no real-time decision making capabilities.

## Transfer Control Instruction (First SCRIPTS Word)

**Bits 31-30 SCSI I/O Processor (10)**

**Bits 29-27 Transfer Opcodes**

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

**Bits 29-27 = 000 (JUMP)**

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.

**Bits 29-27 = 001 (CALL)**

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

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

**Bits 29-27 = 010 (RETURN)**

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.

**Bits 29-27 = 011 (Interrupt)**

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

**Bits 26-24 SCSI Phase Bits**

In the SCSI initiator mode, 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 instruction 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 mode, these bits are ignored.

**Bit 23 Relative Addressing**

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

**Bit 22 Reserved**

This bit is reserved and must be zero.

**Bit 21 Carry Test**

When set, decisions based on the ALU carry bit can be made. True/False comparisons are legal, but Data Compare and Phase Compare are illegal.

**Bit 20 Interrupt on the Fly (INTFLY)**

When this bit is asserted, the interrupt instruction will not halt the SCRIPTS processor.

**Bits 19-16 Sequence Control Bits**

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 Jump If**

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, instruction, or data-out for the target mode. When the bit is set to zero, the transfer control will occur if the comparison yields a false.

**Bit 18 Compare Data**

Compare the data byte value (bit 7 - bit 0 in the instruction) to the first byte of the most recent data, message, instruction, 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 Compare Phase**

In the initiator mode, 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 mode, bit 17 ON causes the chip to test for the attention line on. If the initiator has set attention, the chip (in the target mode) 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 Wait for Valid Phase**

In the initiator mode, 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 mode. 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.

**Bits 15-8 Mask Bits**

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.

**Bits 7-0 Data Byte**

Compare this data byte value to the first byte of the most recent asynchronous data, message, instruction, 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 Instruction**

**(Second SCRIPTS Word)**

### **Bits 31-0 Data Jump Address**

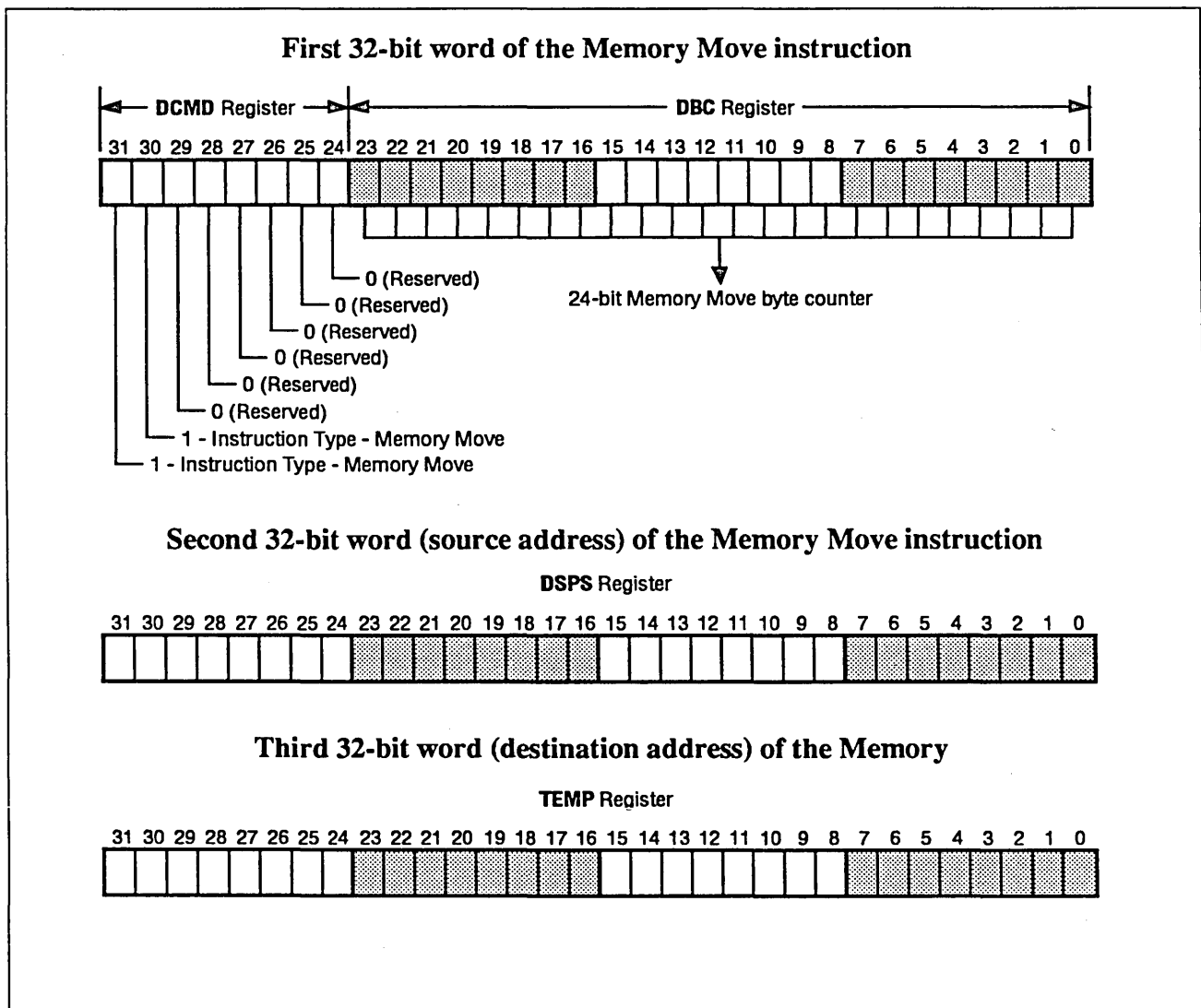
This value specifies the address of the next instruction in memory to transfer control. It is either a 32-bit physical address, or a 24-bit signed value, used as an offset from the DMA SCRIPTS Pointer register. The value is ignored in both return and interrupt instructions. However, the address is loaded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

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 instruction has no operation assignment and can be used as a delay function, or to reserve a SCSI SCRIPTS patch area.

## Memory-to-Memory Move Instructions



### Memory Move Overview

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

If both addresses are in system memory, then the 53C720 functions as a high-speed DMA controller, able to move data at speeds of (up to) 53 MBytes/sec without using the processor or its cache memory. If the source address is within the 53C720's address space, then the instruction is a write to external memory. To perform a read from memory, make the destination address be within the 53C720.

**Memory-to-Memory Move****(First SCRIPTS Word)****Bits 31-30 SCSI I/O Processor Opcode (11)****Bits 29-24 Reserved Section**

These bits should always be zero.

**Bits 23-00 24-bit Byte Count**

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

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

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

The indirect mode is not allowed for the Memory Move instruction; therefore, the byte count must be in the actual SCRIPT. A byte count can be any value; thus, an odd number of bytes can be transferred. The DSA and the TEMP registers are not destroyed.

**Memory Move****(Second SCRIPTS Word)****Bits 31-00 Source Address of the Memory Move**

This value specifies the address from which data will be moved. An address must be the full 32-bit physical address of the data source. The indirect mode is not allowed in the Memory Move instruction. The DMA Next Address register holds this source address and is incremented with each chip DMA transfer. If the value placed in the chip is a 53C720 register address, data can be moved from the 53C720 to a destination address. Only one byte, or multiples of four bytes, can be moved out of the chip. A register-to-register move is possible if both source and destination addresses are within the 53C720's register address space.

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

## Memory Move

(Third SCRIPTS Word)

### Bits 31-00 Destination Address of the Memory Move

This value specifies the address to which data will be moved. An address must be the full 32-bit physical address of the data destination. The indirect mode is not allowed in the Memory Move instruction. The TEMP register holds this destination address and is incremented with each chip DMA transfer. If the value placed in the chip is a 53C720 register address, then data can be moved to the 53C720 from a source address. One byte, or multiples of four bytes, can be moved into the chip. A register-to-register move is possible if both source and destination addresses are within the 53C720's register address space.

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

There is one restriction on addresses that the 53C720 can process. The low order two bits must be equal; thus, the source address must be on the same byte offset within a longword as the destination. An illegal instruction results if the two addresses are not byte aligned. The 53C720 supports burst sizes of 2, 4, 8, or 16 longwords.

During this instruction's execution, the DMA SCRIPTS Pointer Save register and the Data Structure Address register are used (along with the DNAD and TEMP) and will be destroyed. These registers should be saved before a Memory Move instruction and then later restored, if the contents are significant. To save the contents of a register, move its contents to the scratch register and then move the information into memory. Any register not used by the Memory Move instruction can be written directly to memory. Because the moving of data to the 53C720 is the last event performed by the instruction, any register can be written, including the ones used by the instruction.

## Appendix A

# 53C720 Performance Compared to 53C90

This appendix compares firmware required for the 53C720 and the 53C90 to determine how much of a performance boost the 53C720 can offer at a system level (I/Os per second). One microsecond is the time assumed for execution of each external processor instruction.

### Sample Input Data Structure

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

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

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

#### 53C720 Algorithm Description

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

Executing the initiator algorithm takes about 30 SCSI SCRIPTS fetches and indirect data fetches and decodes.

	<u>Approx time in <math>\mu</math>s</u>
The total overhead is	30 $\mu$ sec
The total time per I/O is	30 $\mu$ sec

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

#### 53C90 Algorithm Description.

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

Loop:

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

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

The following sequence is required to perform a 53C90 SCSI operation.

	<u>μsec</u>
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
<hr/> Total time	<hr/> 858 microseconds

## Conclusion

The 53C720 requires less than 5 % of the normal firmware overhead associated with a 53C90, in the simplest case. To further compare the chips, note that a SAVE DATA POINTER operation in the 53C90 requires two processor interrupts (200 μsec) and *no* interrupts using the 53C720. Each data segment in a scatter-gather situation requires 125 μsec on the 53C90 (one interrupt plus DMA initialize), but only 1 μsec on the 53C720 (500 nanosecond instruction fetch, plus indirect data fetch). Thus, an I/O that required four data segments in a scatter-gather mode would require 500 μsec on the 53C90 and 4 μsec on the 53C720 for user data transfer. These factors translate into a four-segment data transfer as follows.

### 53C90

$$(858) + (3 \times 125) = (858 + 375)$$

**1233 μsec per I/O**

### 53C720

**34 μsec per I/O**

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

Function	53C90 msec	53C720 msec
Data Transfer Time	1.00	1.00
Target overhead	1.00	1.00
Host Overhead	<u>1.25</u>	<u>0.034</u>
<i>Total times</i>	3.25	2.034
I/O's Per Second	307	491

In this projected environment, a system can increase its throughput rate by **sixty percent** by using the 53C720 and reducing host computer firmware overhead. At 20 Mbyte/sec SCSI bus rates, the data transfer time is reduced to 200 μsec giving an even more dramatic performance boost. With the types of buffered SCSI disk drives currently available, the 53C720 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 53C720, to increase system level performance, designers must eliminate each delay. The 53C720 can remove much of the host overhead associated with each I/O.

## 53C720 System Bus Utilization

The 53C720, 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 53C720, because no firmware intervention is required.

A real concern is host bus utilization, or *"Does the 53C720 affect host computer performance significantly?"* This appendix provides information about host bus usage when the SCSI bus is saturated at a block size of 512 bytes.

### Host Bus Time To Fetch A SCSI SCRIPTS Command

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

Completing an I/O requires 14 SCSI SCRIPTS.

- select with ATN
- jump error, when not MSG\_OUT
- move FROM msg\_buf, when MSG\_OUT
- jump error, when not CMD phase
- move FROM cmd\_buf, when CMD
- jump error, when not DATA\_IN
- move FROM data\_buf, when DATA\_IN
- jump error, when not STATUS
- move FROM status\_buf, when STATUS
- jump error, when not MSG\_IN
- move FROM msg\_buf, when MSG\_IN
- clear ack
- wait disconnect
- int 0x001
- error:
- int 0x0ff

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

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

Total:

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

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

Fetching data across the system bus requires:

Time in nsec	Instruction
200	ID 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	<b>Total time per SCSI command</b>

Total SCSI-related data fetch time is:

$$6,666 \times 960 = 6.4 \text{ msec}$$

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

Total overhead time is:

$$0.64\% + 3.5\% = 4.14\% \text{ of the time available}$$

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

## Conclusion

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

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



## Use of the Sig\_p Bit in the 53C720

Use of the standard commands to route a bus initiated interrupt, assuming that the 53C720 compatibility bit is on, and the device is in the initiator role. The assumption is that sig\_p is only used to signal that an I/O is ready for execution, and has already been scheduled. If selection is in progress or a select/reselect happens, then sig\_p can be reset, because the new I/O will be executed when the scheduler function gets to it. The system processor will check the connected bit before setting the sig\_p bit to signal that an I/O is to be executed immediately.

```

SELECT FROM buffer, alternate1
; selection happened if execution gets here
    .
    .
    .

```

```

alternate1:
; assume a reselect if here
    WAIT RESELECT, alternate2
; reselected if here, proceed with processing
    .
    .
    .

```

```

alternate2:
; got here because of a sig_p bit set or was
; selected. Did the sig_p bit get set after the
; sel/resel occurred and just before the wait?
    MOVE ISTAT and sig_bit to SFBR
; reset it and do the wait again
    Move CTEST3 to SFBR
    JUMP alternate1 if sig_bit

```

```

alt2:
; can only have been selected if here
    WAIT SELECT, alternate3
    SET TARGET
; selected if here, proceed with processing in
; target mode
    .
    .
    .

```

```

alternate3:
; got here because of a sig_p bit set or error
; Did the sig_p bit get set after the select
; occurred and just before the wait select?
    MOVE ISTAT and sig_bit to SFBR
; reset it and do the wait again
    Move CTEST3 to SFBR
    JUMP alt2 if sig_bit
; should never get here
    INT big_error

```

Aborting a Wait Reselect or Wait Selection SCSI SCRIPT, assuming that the 53C720 compatibility bit has been set and the device is in the initiator role.

```

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

```

```

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

```

```

alt_sig_p1:
    MOVE ISTAT and connect_bit to SFBR
; test the SCSI connected bit
    JUMP alt_sig_p2, if connect_bit
; either the chip got selected, reselected, or the
; sig_p bit was set
    MOVE ISTAT and sig_bit to SFBR
; test the sig_p bit first
    JUMP sig_p_set, if sig_bit
; big error if here — not connected and sig_p was
; not set
    INT big_error1

```

**alt\_sig\_p2:**

; Bus initiated interrupt occurred if here —  
; connected bit is on. First reset the sig\_p bit, so  
; the alternate jump is NOT taken.

**MOVE CTEST2 to SFBR**

**WAIT RESELECT, alt\_sig\_p3**

; process the reselection

•  
•  
•

**alt\_sig\_p3:**

; got selected

**SET TARGET**

•  
•  
•

**sig\_p\_set:**

; System processor has set the sig\_p bit.

; Reset it and service the system request.

**MOVE CTEST2 to SFBR**

•  
•  
•

# Appendix D

## Compiler SCRIPTS Examples

### SAMPLE SCSI SCRIPTS Source File

```

;*****
;
;* The following are variable data values provided          *
;* external to the compiler and resolved at run-time        *
;*****
;
; Definition area INITIATOR ROLE
; Target Device ID offset in the data table.
EXTERN device
EXTERN status_adr
EXTERN sendmsg
EXTERN rcvmsg
EXTERN cmd_adr
EXTERN data_adr
; Ten byte buffer address offset.
; Ten byte buffer address offset.
; Buffer address offset for the SCSI command
; Address of user data buffer

;*****
;
;* Absolute values are stored in DSPS Register            *
;* for purposes of interrupt processing                    *
;*****
;
;*****
;
;* Note that 0x0 precedes the interrupt status            *
;* values and designates a hex value                      *
;*****
ABSOLUTE err1 = 0x0ff01
ABSOLUTE err2 = 0x0ff02
ABSOLUTE err3 = 0x0ff03
ABSOLUTE err4 = 0x0ff04
ABSOLUTE ok = 0x0ff05
ABSOLUTE err5 = 0x0ff05
ABSOLUTE err6 = 0x0ff06
; Error — unexpected SCSI phase before command phase
; Error — unexpected SCSI phase after a command
; Error — expected status phase
; No Error — good I/O
; Error — expected message outphase
; Error — expected message command complete

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

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

PROC sample:
select atn from device, REL (resel_adr)
int err1 when not MSG_OUT
move FROM sendmsg, when MSG_OUT
int err2 when not CMD
move FROM cmd_adr, when CMD
jump REL (end) when STATUS
jump REL (input_data) if DATA_IN
jump REL (output_data) if DATA_OUT
int err3
; select the device with attention on
; if the next phase is not msg_out, interrupt
; sent the id message out to the target
; if next phase is not command, interrupt
; send the command bytes
; go to process cleanup if status phase
; process data-in phase
; or data-out phase
; unexpected phase if here

```

```
input_data:
move FROM data_adr, when DATA_IN
jump REL (end)
; process the data-in phase
; and go process status

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

end:
int err4 when not STATUS
move FROM status_adr, when STATUS
int err5 when not MSG_IN
move FROM rcvmsg, when MSG_IN
int err6 if not 00
clear ack
wait disconnect
int ok
; interrupt if not status phase
; move the status byte into memory
; interrupt if message-in is not next
; move the command complete byte in
; interrupt if not command complete
; accept the message if there are no problems
; wait for a physical disconnect
; interrupt with an I/O complete

reset_adr:
int ok
```

### SAMPLE LIST FILE

```

1      ;*****
2      ;* The following are variable data values provided          *
3      ;* external to the compiler and resolved at run-time      *
4      ;*****
5
6      ; Definition area INITIATOR ROLE
7
8      ; Target Device ID offset in the data table.
9  EXTERN device
10
11 EXTERN status_adr
12
13      ; Ten byte buffer address offset.
14 EXTERN sendmsg
15
16      ; Ten byte buffer address offset.
17 EXTERN rcvmsg
18
19      ; Buffer address offset for the SCSI command
20 EXTERN cmd_adr
21
22      ; Address of user data buffer
23 EXTERN data_adr
24
25      ;*****
26      ;* Absolute values are stored in DSPS Register          *
27      ;* for purposes of interrupt processing                  *
28      ;*****
29
30      ;*****
31      ;* Note that 0x0 precedes the interrupt status          *
32      ;* values and designates a hex value                    *
33      ;*****
34
35 ABSOLUTE err1 = 0x0ff01
36
37      ; Error — unexpected SCSI phase before command phase
38 ABSOLUTE err2 = 0x0ff02
39
40      ; Error — unexpected SCSI phase after a command
41 ABSOLUTE err3 = 0x0ff03
42
43      ; Error — expected status phase
44 ABSOLUTE err4 = 0x0ff04
45
46      ; No Error — good I/O
47 ABSOLUTE ok = 0x0ff00
48
49      ; Error — expected message outphase
50 ABSOLUTE err5 = 0x0ff05
51

```

```

52                                     ; Error — expected message command complete
53 ABSOLUTE err6 = 0x0ff06
54
55     ;*****
56     ; The following shows how you can use the PASS          *
57     ; capability of the compiler to pass C code to the      *
58     ; output file                                           *
59     ;*****
60 #include "NCR.h" PASS(#include "NCR.h")
61 extern char line[]; PASS(extern char line[];)
62
63 00000000: PROC sample:
64                                     ; select the device with attention on
65 00000000: 47000000 00000098          select atn from device, REL (resel_adr)
66
67                                     ; if next phase is not msg_out,interrupt
68 00000008: 9E030000 0000FF01          int err1 when not MSG_OUT
69
70                                     ; sent the id message out to the target
71 00000010: 1E000000 00000000          move FROM sendmsg, when MSG_OUT
72
73                                     ; if next phase is not command, interrupt
74 00000018: 9A030000 0000FF02          int err2 when not CMD
75
76                                     ; send the command bytes
77 00000020: 1A000000 00000000          move FROM cmd_adr, when CMD
78
79                                     ; go to process cleanup if status phase
80 00000028: 838B0000 00000030          jump REL (end) when STATUS
81
82                                     ; process data-in phase
83 00000030: 818A0000 00000010          jump REL (input_data) if DATA_IN
84
85                                     ; or data-out phase
86 00000038: 808A0000 00000018          jump REL (output_data) if DATA_OUT
87
88                                     ; unexpected phase if here
89 00000040: 98080000 0000FF03          int err3
90
91                                     ; process the data-in phase
92 00000048: input_data:
93 00000048: 19000000 00000000          move FROM data_adr, when DATA_IN
94
95                                     ; and go process status
96 00000050: 80880000 00000008          jump REL (end)
97
98                                     ; process the data-out phase
99 00000058: output_data:
100 00000058: 18000000 00000000          move FROM data_adr, when DATA_OUT
101
102                                     ; interrupt if not status phase
103 00000060: end:

```

```

104 00000060: 9B030000 0000FF04      int err4 when not STATUS
105
106                                     ; move the status byte into memory
107 00000068: 1B000000 00000000      move FROM status_adr, when STATUS
108
109                                     ; interrupt if message in is not next
110 00000070: 9F030000 0000FF05      int err5 when not MSG_IN
111
112                                     ; move the command complete byte in
113 00000078: 1F000000 00000000      move FROM rcvmsg, when MSG_IN
114
115                                     ; interrupt if not command complete
116 00000080: 98040000 0000FF06      int err6 if not 00
117
118                                     ; accept the message if there are no problems
119 00000088: 60000040 00000000      clear ack
120
121                                     ; wait for a physical disconnect
122 00000090: 48000000 00000000      wait disconnect
123
124                                     ; interrupt with an I/O complete
125 00000098: 98080000 0000FF00      int ok
126 000000A0: resel_adr:
127 000000A0: 98080000 0000FF00      int ok
    
```

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

**SAMPLE OUTPUT FILE**

```

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

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

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

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

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

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

```



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

#define E_data_adr    0x00000000
ULONG E_data_adr_Used[] = {
    0x00000013,
    0x00000017
};

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

### Notes

A large, empty rectangular box with a thin black border, occupying most of the page. It is intended for the user to write notes.

## APPENDIX E

# 53C720 Test SCRIPTS Examples

```
*****  
; Filename: 720TEST.SS  
; This sample SCRIPT shows the Memory to Memory Move, Carry, and  
; Register Read/Write capabilities of the 53C720.  
*****
```

```
; Memory to Memory Move instructions can be done to/from the 53C720's  
; internal registers if the address given decodes to the memory mapped  
; address of the 53C720. Below are the addresses for the SCRATCHA and SCRATCHB  
; registers in little endian mode when the base address of the card is D000.  
; Converting D000:6034 to an absolute address gives 0xD6034. Converting D000:  
; 605C to an absolute address gives 0xD605C.
```

```
ABSOLUTE ScratchA_Zero_Addr = 0x0D6034  
ABSOLUTE ScratchA_One_Addr  = 0x0D6035  
ABSOLUTE ScratchA_Two_Addr  = 0x0D6036  
ABSOLUTE ScratchA_Three_Addr = 0x0D6037  
ABSOLUTE ScratchB_Zero_Addr = 0x0D605C  
ABSOLUTE ScratchB_One_Addr  = 0x0D605D  
ABSOLUTE ScratchB_Two_Addr  = 0x0D605E  
ABSOLUTE ScratchB_Three_Addr = 0x0D605F
```

```
; Note: When doing memory to memory moves to/from the chip's address space,  
; the buffers must be long word aligned. This is because the chip's  
; registers are long word aligned, and memory to memory move instructions  
; require that the last two address bits of the source and destination  
; addresses be the same. If the software doesn't load the SCRIPTS starting  
; at a long word boundary, then the relative buffers will not be long word  
; aligned and illegal instruction interrupts will occur when executing the  
; memory to memory moves to/from the chips address space. A simple way to  
; fix this is to add a small (1, 2, or 3) byte buffer to the beginning of  
; the relative buffers that will force the rest of the buffers to be  
; long word aligned. A better way to fix this is to force your software  
; to load the SCRIPTS starting at a long word boundary.
```

```
; Relative buffers in memory
```

```
RELATIVE Temp_Buff1 = 0  
RELATIVE Temp_Buff2 = Temp_Buff1 + 4  
RELATIVE Byte_0 = Temp_Buff2 + 4  
RELATIVE Byte_1 = Byte_0+1  
RELATIVE Byte_2 = Byte_1+1  
RELATIVE Byte_3 = Byte_2+1
```

ENTRY start

**start:**

**; Use Register Write to put 0xFF's in SCRATCHA register**

```
move 0xFF to SCRATCHA0
move 0xFF to SCRATCHA1
move 0xFF to SCRATCHA2
move 0xFF to SCRATCHA3
```

**; Use Memory-to-Memory Move instruction to move 1 byte into the SCRATCHA  
; register from memory.**

```
move memory 1, Temp_Buff1, ScratchA_Zero_Addr
```

**; Now move that byte back out of the SCRATCHA register**

```
move memory 1, ScratchA_Zero_Addr, Temp_Buff2
```

**; Use 2 Register Read/Write instructions to copy one register to another.  
; When moving from one register to another, the SFBR must be used as an  
; intermediate step.**

```
move SCRATCHB0 to SFBR
move SFBR to SCRATCHA0
```

**; Move 4 bytes from memory to the SCRATCHA register using 4 separate  
; byte wide moves to show different alignments**

```
move memory 1, Byte_0, ScratchA_Zero_Addr
move memory 1, Byte_1, ScratchA_One_Addr
move memory 1, Byte_2, ScratchA_Two_Addr
move memory 1, Byte_3, ScratchA_Three_Addr
```

**; Move 4 bytes out of the SCRATCHA register and put them in memory**

```
move memory 1, ScratchA_Zero_Addr, Byte_0
move memory 1, ScratchA_One_Addr, Byte_1
move memory 1, ScratchA_Two_Addr, Byte_2
move memory 1, ScratchA_Three_Addr, Byte_3
```

**; Re-initialize the SCRATCHA register to all 0xFF's.**

```
move 0xFF to SCRATCHA0
move 0xFF to SCRATCHA1
move 0xFF to SCRATCHA2
move 0xFF to SCRATCHA3
```

```
; Now move 4 bytes at a time from memory into the SCRATCHA register
; to show 32 bit accesses. Note that the address's of Temp_Buff1 and
; ScratchA_Zero_Addr must have the same long word alignment (A0-A1
; must be the same).
```

```
    move memory 4, Temp_Buff1, ScratchA_Zero_Addr
```

```
; Now move the data back out of the SCRATCHA register using a long
; word access
```

```
    move memory 4, ScratchA_Zero_Addr, Temp_Buff2
```

```
; Move 1 byte from memory to memory
```

```
    move memory 1, Temp_Buff1, Temp_Buff2
```

```
; Move 2 bytes from memory to memory
```

```
    move memory 2, Temp_Buff1, Temp_Buff2
```

```
; Move 3 bytes from memory to memory
```

```
    move memory 3, Temp_Buff1, Temp_Buff2
```

```
; Move 4 bytes from memory to memory
```

```
    move memory 4, Temp_Buff1, Temp_Buff2
```

```
; The next section implements a counter that counts from 0 to 0xFFFF. It
; shows how the SFBR register can be used in conjunction with a transfer
; control instruction to compare for certain data values.
```

```
; Use Register Write to put 0x00's in SCRATCHA register
```

```
    move 0x00 to SCRATCHB0
```

```
    move 0x00 to SCRATCHB1
```

```
    move 0x00 to SCRATCHB2
```

```
    move 0x00 to SCRATCHB3
```

```
addbyte0:
```

```
    move SCRATCHB0 + 0x01 to SCRATCHB0
```

```
    move SCRATCHB0 to SFBR
```

```
    jump addbyte0 if not 0xff
```

```
    move SCRATCHB1 + 0x01 to SCRATCHB1
```

```
    move SCRATCHB1 to SFBR
```

```
    jump addbyte0 if not 0xff
```

```
; End of 0 to 0xFFFF counter routine
```

```
; The rest of this SCRIPT demonstrates how the carry can be used to make  
; a 32 bit counter. As is, this SCRIPT will count from 0 to  $2^{32}$ . This  
; will most likely take hours depending on the hardware surrounding the  
; 53C720. To execute this SCRIPT in a reasonable amount of time, I  
; recommend you put an interrupt instruction after the first two loops.  
; This will make it count from 0 to  $2^{16}$  which doesn't take long at all.
```

```
; Use Register Write to put 0x00's in SCRATCHA register
```

```
    move 0x00 to SCRATCHB0  
    move 0x00 to SCRATCHB1  
    move 0x00 to SCRATCHB2  
    move 0x00 to SCRATCHB3  
    clear carry
```

```
addest:
```

```
    move SCRATCHB0 + 0x01 to SCRATCHB0  
    jump addest if not carry
```

```
    move SCRATCHB1 + 0x00 to SCRATCHB1 with carry  
    jump addest if not carry
```

```
; int 0x0b
```

```
    move SCRATCHB2 + 0x00 to SCRATCHB2 with carry  
    jump addest if not carry
```

```
    move SCRATCHB3 + 0x00 to SCRATCHB3 with carry  
    jump addest if not carry
```

```
; Interrupt saying we are all done
```

```
    int 0x0A
```

## SCRIPTS™ Compiler Error Messages

The NCR SCSI SCRIPTS compiler diagnostic messages fall into four classes: Fatal Errors, Errors, and Warnings.

### Fatal Errors

When a fatal error occurs, compilation immediately stops. You must take appropriate action and then restart compilation.

#### **No memory. Aborting compiler**

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

#### **Local stack overflow. Aborting compile**

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

#### **Cannot open file**

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

#### **Cannot read file**

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

## Errors

Errors indicate program syntax errors, disk or memory access errors, and command line errors.

### Expected digit

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

### Expected a separator

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

### Numeric constant has too many digits

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

### Expected a value

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

### Undefined variable

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

### Unknown identifier

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

### Expected an identifier

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

### Expected a variable

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

### 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.

### Expected a reserved word

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

### Expected a PHASE

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

### Cannot use a RELATIVE in a non address field

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



## Warning

Warnings do not prevent the compilation from finishing.

### Identifier truncated

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

### Redefinition of variable

A variable was defined two or more times.

### Duplicate ATN

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

### Duplicate ACK

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

### Undefined label used as entry point

The label was not defined as an entry point.

### Unused variable

A variable was defined but not used in the SCRIPT.

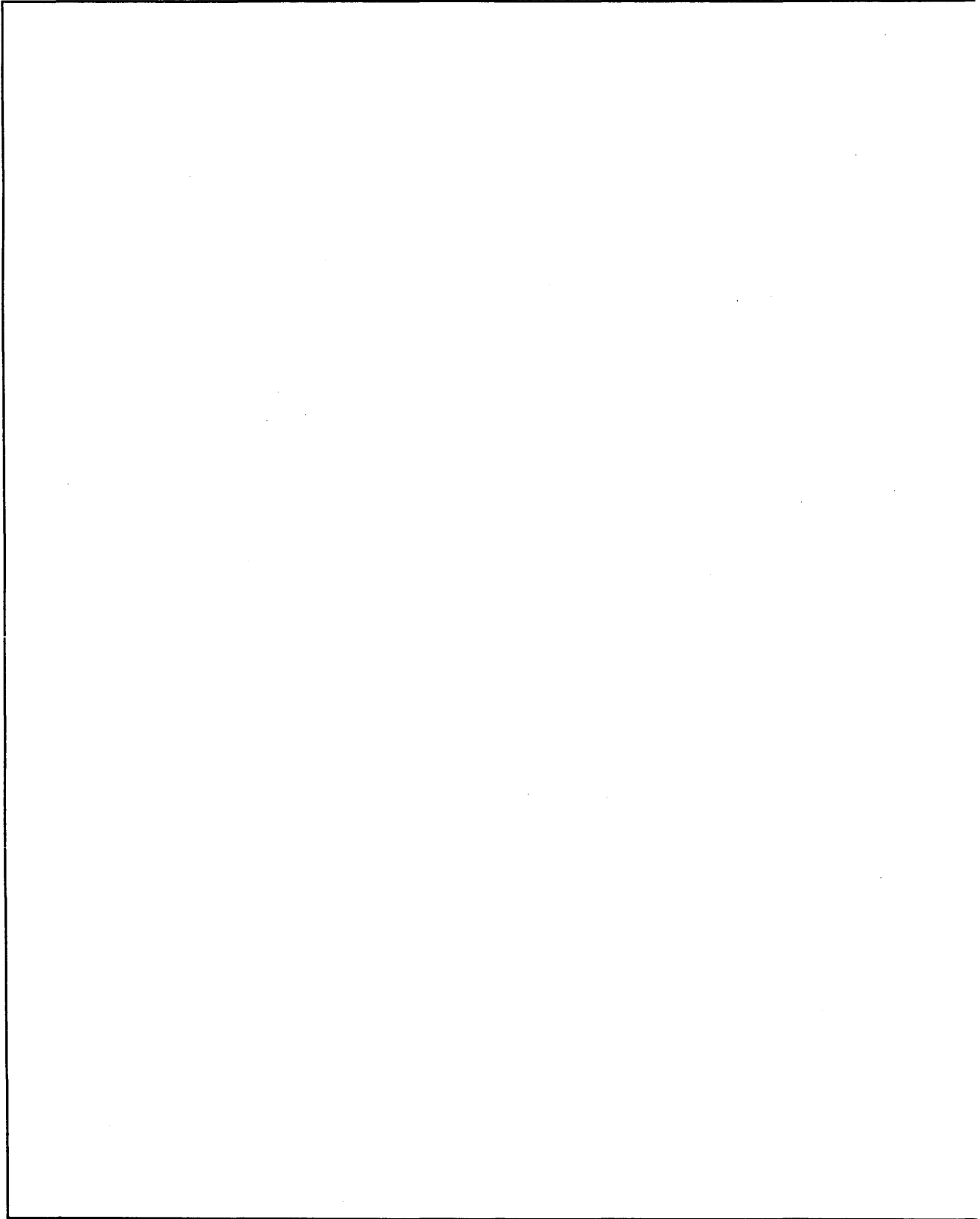
### Lost resolution

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

### Duplicate label

A label was defined more than once.

**Notes**



## Appendix G

# Miscellaneous Design Topics

The following paragraphs detail design topics.

## Design Topics

The following design topics are discussed.

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

### SCSI Timers

Some SCSI systems have a system requirement with respect to activity on the SCSI bus.

If there are long periods with no SCSI activity then the SCSI driver must notify the system software that a time-out has occurred. The 53C720 provides programmable select/reselect, handshake to handshake, and general purpose timers. The time-out period is programmable from 100  $\mu$ sec to greater than 1.6 seconds. A maskable interrupt is available for each of the timers. Timers are masked in the SIENI register and status of the timers are checked in the SISTI register.

### Longitudinal Parity Register (SLPAR)

For a simple error check of any data passing through the 53C720, there is an 8-bit register that contains a continual exclusive OR of the data. The value in the chip is cleared by any write to the register. A designer can use the information by performing the following:

1. Clear the value with a SCRIPTS write.
2. Move data through the 53C720.
3. Move the generated byte to the SCSI target to be stored with the data.
4. Read in the extra byte on a read, and compare it to the byte generated during the move.

All the extra moves and compares can be done by the 53C720 or by the system processor, depending on the designer's preference.

Note that the SLPAR doubles as the SFBR during a select or reselect. The device id is always written into the SLPAR. Because a SCRIPT could be writing to the SFBR during a SCSI bus-initiated interrupt, the value could be destroyed. Optionally, therefore, the chip can be set to write the device id only to the SLPAR.

### Big/Little Endian Support

There is some support for both Big and Little Endian in the 53C720. Four areas must be considered when discussing the byte ordering.

#### 1. SCRIPTS Order

To ensure that all SCSI SCRIPTS are in the correct order, each SCRIPT must be compiled in the target architecture. The C output is a longword value, which will be stored in the memory by the processor and in the correct order for the subsequent execution. If a little Endian SCRIPT is to be executed on a big Endian machine, the bytes will need to be reversed before execution by the 53C720 (in big Endian mode). Note that a PROM cannot be moved from one environment to another without re-ordering bytes within each word.

#### 2. 53C720 Register Access from Firmware

There is a big Endian and a little Endian address mode for the registers. To develop code that works in either mode, simply use equates with an Endian switch that includes the appropriate set of address values. Note that the change is only for byte access. If 32 bits are accessed, there is no change from big to little Endian.

#### 3. 53C720 Register Access from SCSI SCRIPTS

The compiler offers a set of logical names that can be used to access registers. Names do not change when the mode changes, and the binary code required to access a register does not change either.

#### **4. User Data Byte Ordering**

Data transfers to/from system memory from/to the SCSI bus always start at the beginning address and continue until the last byte is sent. No internal re-ordering of the data for either mode occurs. A serial stream of data is assumed, and the first byte on the SCSI bus is associated with the lowest address in system memory.

#### **SCRIPTS in a Host Adapter**

Some designs require that SCSI SCRIPTS be fetched from a local ROM rather than from system memory across the bus. Typically, this requirement comes from the desire to avoid traffic on the bus or is caused by large overheads associated with bus arbitration. The 53C720 allows several options in the placement of SCRIPTS and table indirect data.

SCRIPTS and data structures can be placed in system memory.

Using the FETCH pin, external system bus interface hardware can read SCRIPTS locally and all other data from system memory. During SCRIPT fetches, the pin is active, and thus, the access can go locally rather than across the system bus.

In the CTEST8 register is the fetch mode bit. When set, the FETCH pin will deassert during indirect and table indirect read operations. FETCH will be active during SCRIPT fetches only. Thus, external hardware can drive the opcode fetch to one memory area (local ROM) and table indirect fetches to another area (system RAM). If the bit is not set, then fetch is asserted throughout the instruction fetch.

Thus, the designer can place SCRIPTS, user data, and table indirect data in the most convenient area of memory. Note that the options can be changed dynamically by writing to the registers from SCRIPTS.

# Using the 53C720 in Low Level Mode

## Low-level SCSI Code

Pseudocode examples of selection, message-out, command, data-in, status, and message-in.

\*\*\*\*\*  
**Selection:** \*

parity check, generation  
SCNTL0=0XCC  
C700 id=7, target id=2  
SODL=0X84  
assert BSY  
SOCL=0X20  
assert SODL, connected; if not connected, ATN cannot be asserted  
SCNTL1=0X50  
low-level mode (Note: Disable low-level mode before starting the SCRIPTS' processor.)  
DCNTL=0X08  
assert SEL, ATN, BSY  
SOCL=0X38  
deassert BSY, keep SEL, ATN  
SOCL=0X18  
wait for BSY, asserted by Target  
(SBCL & 0X20)=0X20  
deassert SEL, keep ATN  
SOCL=0X08

\*\*\*\*\*  
**Message-Out** \*

look for REQ and message-out  
(SBCL & 0X87)=0X86  
identify message  
SODL  
message-out phase; a phase match asserts SODL onto the SCSI bus  
SOCL=0X0E  
assert ACK, message-out, keep ATN  
SOCL=0X4E  
wait for REQ deasserted  
wait for (SBCL & 0X80)=0X00  
deassert ACK, ATN; keep message-out  
SOCL=0X06

\*\*\*\*\*  
**Command** \*

look for REQ and command  
(SBCL & 0X87)=0X82  
initialize command byte  
SODL=command byte  
assert ACK, command  
SOCL=0X42  
wait for REQ deasserted  
wait for (SBCL & 0X80)=0X00  
deassert ACK, keep command  
SOCL=0X02  
repeat until last command byte

\*\*\*\*\*  
**Data-In** \*

look for REQ and data-in  
(SBCL & 0X87)=0X81  
SBDL=data byte  
assert ACK, data-in  
SOCL=0X41  
wait for REQ deasserted  
wait for (SBCL & 0X80)=0X0  
deassert ACK, keep command  
SOCL=0X02  
repeat until last data byte

\*\*\*\*\*  
**Status** \*

look for REQ and status  
(SBCL & 0X87)=0x83  
ACK, status phase  
SOCL=0X43  
SBDL contains status byte  
status=SBDL  
wait for REQ deasserted  
wait for (SBCL & 0X80)=0X00  
deassert ACK; keep status phase  
SOCL=0X03

\*\*\*\*\*  
**Message-in** \*  
\*\*\*\*\*

look for REQ and message-in  
(SBCL & 0X87)=0X87  
ACK, message-in phase  
SOCL=0X47  
SBDL contains message byte  
message-in=SBDL  
wait for REQ deasserted  
wait for (SBCL & 0X80)=0X00  
deassert ACK; keep message-in phase  
SOCL=0X07

**Notes**

A large, empty rectangular box with a thin black border, occupying most of the page. It is intended for the user to write notes.



# Index

-e[errorfilename] - 3-1  
-l[ListFilename] - 3-1  
-o[OutputFile name] - 3-1  
-u - 3-1  
-v - 3-1  
-w - 3-1  
-z[debugfilename] - 3-1

53C7X0 - 4-1  
53C90 Algorithm Description. - A-1  
53C720 Algorithm Description - A-1  
53C720 Performance Compared to 53C90 - A-1  
53C720 Register Access from Firmware - G-1  
53C720 Register Access from SCSI SCRIPTS - G-1  
53C720 Strengths in the Disk Drive Environment - 7-1  
53C720 System Bus Utilization - B-1  
53C720 Test SCRIPTS Examples - E-1  
68030 - 1-4  
80386 - 1-4  
80386SX - 1-4

## A

A SCSI Solution - 1-1  
ABSOLUTE - 2-3, 4-3, 4-5  
ACK - 4-3  
AND - 4-3  
array - 1-5  
ASCII - 3-1  
ATN - 4-3

## B

back-end - 4-1  
Big/Little Endian Support - G-1  
binary - 4-1  
Bit 16 Wait for Valid Phase - 10-16  
Bit 17 Compare Phase - 10-16  
Bit 18 Compare Data - 10-16  
Bit 19 Jump If - 10-16  
Bit 20 Interrupt on the Fly (INTFLY) - 10-16  
Bit 21 Carry Test - 10-15

Bit 22 Reserved - 10-15  
 Bit 23 Relative Addressing - 10-15  
 Bit 23 Reserved - 10-13  
 Bit 24 Carry Enable - 10-13  
 Bit 24 SELECT With ATN - 10-10  
 Bit 25 Table Indirect Mode - 10-10  
 Bit 26 Relative Addressing Mode - 10-10  
 Bit 27 Block Move Opcode - 10-3  
 Bit 28 = 0 Table Direct Mode - 10-2  
 Bit 28 = 1 Table Indirect Mode - 10-2  
 Bit 28 Table Indirect Field - 10-2  
 Bit 29 = 0 Direct Addressing - 10-2  
 Bit 29 = 1 Indirect Addressing - 10-2  
 Bit 29 Indirect data address flag - 10-2  
 Bits 7-0 Data Byte - 10-16  
 Bits 7-0 Reserved - 10-13  
 Bits 15-0 Flags Field - 10-10  
 Bits 15-8 Immediate Data Field - 10-13  
 Bits 15-8 Mask Bits - 10-16  
 Bits 19-16 Sequence Control Bits - 10-16  
 Bits 22-16 Register Address Field - 10-13  
 Bits 23-0 Block Move Byte Count - 10-4  
 Bits 23-00 24-bit Byte Count - 10-19  
 Bits 23-16 SCSI ID 7-0 - 10-10  
 Bits 26-24 SCSI Phase Bits - 10-15  
 Bits 26-24 SCSI Phase Lines - 10-4  
 Bits 29-24 Reserved Section - 10-19  
 Bits 29-27 = 000 (JUMP) - 10-15  
 Bits 29-27 = 001 (CALL) - 10-15  
 Bits 29-27 = 010 (RETURN) - 10-15  
 Bits 29-27 = 011 (INTERRUPT) - 10-15  
 Bits 29-27 = 101 (Move from SFBR) - 10-12  
 Bits 29-27 = 110 (Move to SFBR) - 10-12  
 Bits 29-27 = 111 (Read-Modify-Write) - 10-13  
 Bits 29-27 I/O Instruction Opcodes - 10-7  
 Bits 29-27 Transfer Opcodes - 10-15  
 Bits 31-0 Data Jump Address - 10-17  
 Bits 31-0 Data Start Address - 10-5  
 Bits 31-0 Jump Address - 10-13  
 Bits 31-00 Destination Address of the Memory Move - 10-20  
 Bits 31-00 Source Address of the Memory Move - 10-19  
 Bits 31-30 Block Move (00) - 10-2  
 Bits 31-30 I/O Instruction (01) - 10-7

Bits 31-30 Read/Write Instructions (01) - 10-11  
Bits 31-30 SCSI I/O Processor (10) - 10-14  
Bits 31-30 SCSI I/O Processor Opcode (11) - 10-19  
Block Move Instruction - 10-2  
Block Move Instruction - 10-5  
Block Move Instructions - 10-1  
Block Move Overview - 10-2  
Block Move - 4-7  
brackets - 3-1

## C

C Source Code - 1-5  
CALL Instruction - 4-12  
CALL - 4-1  
Cannot open file - F-1  
Cannot read file - F-1  
Cannot use a RELATIVE in a non address field - F-2  
CARRY - 4-3  
Chained Block Move - 4-8  
Chained Move Instruction - 10-5  
chained mode - 1-4  
CHMOV - 4-1  
CLEAR - 4-1  
Clear Instruction - 4-18  
Compiled Output - 3-2  
Compiler Directives Syntax - 4-5  
Compiler Directives - 4-3  
Compiler SCRIPTS Examples - D-1  
compiler - 1-5  
count - 4-4  
CPU - 1-1, 1-6

## D

data - 4-4  
Data-in Phase - 8-2  
Data-out Phase - 8-2  
debugger - 3-1  
Decimal - 4-1  
Definition area - 2-1  
Design Topics - G-1  
Direct Block Move - 4-7  
Direct Chained Block Move - 4-9

DISCONNECT - 4-1  
Disconnect Instruction - 4-17  
Disconnect SCSI SCRIPTS - 9-2  
Disk Drive Initiator Sequence - 7-1  
DMA Component - 1-3  
DMA - 1-1,1-3  
DSA - 1-4  
Duplicate ACK - F-3  
Duplicate ATN - F-3  
Duplicate label - F-3

## **E**

ENTRY - 4-4,4-5  
Errors - F-2  
Example of a SCRIPTS Operation - 1-6  
Expected a PHASE - F-2  
Expected a reserved word - F-2  
Expected a separator - F-2  
Expected a value - F-2  
Expected a variable - F-2  
Expected an expression - F-2  
Expected an identifier - F-2  
Expected digit - F-2  
expression [,name = expression...] - 4-5  
expression [,name = expression...] - 4-6  
expression - 4-4  
EXTERNAL name [,name...] - 4-5  
EXTERNAL - 2-2

## **F**

Fatal Errors - F-1  
First 32-bit word of the I/O Instructions - 10-7  
First generation - 1-1  
FROM - 4-3  
front-end - 4-1

## **H**

hex - 4-1  
Hexadecimal - 4-1  
high level - 1-4  
Host Bus Time To Fetch A SCSI SCRIPTS Command - B-1  
Host System - 1-6  
How SCSI SCRIPTS becomes part of a C Language Program - 1-5  
Index - 4

## I

I/O Instructions Overview - 10-7  
I/O Instructions - 10-7,10-13  
id - 4-4  
Identifier truncated - F-3  
IF - 4-3  
Indirect Block Move - 4-7  
Indirect Chained Block Move - 4-9  
Initiator Mode Bit 27 = 0 (CHMOV) - 10-4  
Initiator Mode Bit 27 = 1 (MOVE) - 10-4  
Initiator Mode Bits 29-27 = 000 (Selection) - 10-8  
Initiator Mode Bits 29-27 = 001 (Wait Disconnect) - 10-9  
Initiator Mode Bits 29-27 = 010 (Wait Reselect) - 10-9  
Initiator Mode Bits 29-27 = 011 (Set) - 10-9  
Initiator Mode Bits 29-27 = 100 (Reset) - 10-9  
Instruction Keywords - 4-1, 4-7  
INT - 4-1  
INTERRUPT Instruction - 15  
INTERRUPT on the FLY Instruction - 4-16  
INTFLY - 4-1  
Invoking the SCSI SCRIPTS Compiler - 3-1

## J

JUMP Instruction - 4-11  
JUMP - 4-1

## K

KEYWORD count, address - 4-6  
KEYWORD - 4-1, 4-4

## L

Labels - 3-1  
lines of code - 1-1  
Linker - 1-5  
Local stack overflow. Aborting compile - F-1  
Longitudinal Parity Register (SLPAR) - G-1  
Lost resolution - F-3  
Low level - 1-4  
Low-level SCSI Code - H-1

## **M**

Main SCSI SCRIPTS - 9-1  
MASK - 4-3  
MEMORY MOVE - 10-19, 10-20  
MEMORY - 4-3  
Memory Move Overview - 10-18  
Memory to Memory Move - 4-10, 10-19, 10-18  
MIPS - 1-1  
Miscellaneous Design Topics - G-1  
Miscellaneous Instructions - 4-16  
Miscellaneous Keywords - 4-3  
MOVE Instructions - 4-7  
MOVE MEMORY - 4-1  
MOVE - 4-1  
Multi-Tasking I/O Using SCSI SCRIPTS - 9-1  
Multi-Threaded I/O Using SCSI SCRIPTS - 9-1

## **N**

name - 4-4  
NCR 53C720 SCSI I/O Processor Chip Block Diagram - 1-2  
NCR SCSI SCRIPTS™ Description - 1-4  
No memory. Aborting compiler - F-1  
NOP - 4-1  
NOT - 4-3  
Numeric constant has too many digits - F-2

## **O**

octal - 4-1  
offset - 4-4  
OR - 4-3

## **P**

P-Calbe - 1-3  
PASS (#include "NCR.h") - 4-5  
PASS (literal string) - 4-5  
PASS Option - 4-5  
PASS - 2-3, 4-4  
Phase Keywords - 4-2  
Phase - 4-4  
PROC label - 4-5

PROC - 4-4  
Processing a SAVE DATA POINTERS Message - 8-2  
PTR - 4-3

## R

Read/Write Overview - 10-11  
Read/Write Register Instructions - 10-11  
Redefinition of variable - F-3  
REG - 4-3  
Register Keywords - 4-2  
Register Read/Write Instruction - 4-18  
REL - 4-3  
RELATIVE name = - 4-6  
RELATIVE - 4-4  
RESELECT - 4-1  
Reselect Instruction - 4-17  
reserved word - 4-1  
Resume SCSI SCRIPTS - 9-3  
RETURN Instruction - 4-14  
RETURN - 4-1

## S

SAMPLE SCSI SCRIPTS Source File - D-1  
Sample Input Data Structure - A-1  
Scheduler SCSI SCRIPTS - 9-1  
SCRIPT area - 2-1  
SCRIPTS for the Initiator Role - 6-1  
SCRIPTS for the Target Role - 6-7  
SCRIPTS in a Host Adapter - G-2  
SCRIPTS Keywords - 4-1  
SCRIPTS Notation - 4-4  
SCRIPTS Order - G-1  
SCRIPTS™ Compiler Error Messages - F-1  
SCSI Character Oriented Device in the Initiator Role - 7-2  
SCSI SCRIPTS Compiler Output - 3-2  
SCSI SCRIPTS Compiler - 1-5  
SCSI SCRIPTS Machine Language Description - 10-1  
SCSI SCRIPTS - 4-1  
SCSI SCRIPTS™ Processor - 1-3  
SCSI Timers - G-1  
Second 32-bit word of the I/O Instructions - 10-7  
Second generation - 1-1

SELECT [ATN] FROM offset, Address - 4-17  
SELECT [ATN] FROM offset, REL (Address) - 4-17  
SELECT [ATN] ID, Address - 17  
SELECT [ATN] ID, REL (Address) - 4-17  
SELECT - 4-1  
Select Instruction - 4-17  
Set Instruction - 4-18  
Source Code - 3-2  
SSC Sourcefine Options - 3-1  
Syntax - 4-18

## T

Table Indirect Block Move - 4-8  
Table Indirect Chained Block Move - 4-9  
Tape Drive Initiator Sequence - 7-1  
TARGET - 4-3  
Target Mode Bit 27 = 0 (MOVE) - 10-3  
Target Mode Bit 27 = 1 (CHMOV) - 10-4  
Target Mode Bits 29-27 = 000 (Reselect) - 10-8  
Target Mode Bits 29-27 = 001 (Disconnect) - 10-8  
Target Mode Bits 29-27 = 010 (Wait Select) - 10-8  
Target Mode Bits 29-27 = 011 (Set) - 10-8  
Target Mode Bits 29-27 = 100 (Reset) - 10-8  
The NCR SCSI I/O Processor - 1-3  
The NCR SCSI SCRIPTS Language Syntax - 4-1  
Third generation - 1-1  
token - 4-1  
TRANSFER CONTROL INSTRUCTION - 10-14, 10-17  
Transfer Control Instructions - 10-14  
Transfer Control Overview - 10-14  
Transferring Large Blocks of User Data - 8-1

## U

Undefined label used as entry point - F-3  
Undefined variable - F-2  
Unknown identifier - F-2  
Unused variable - F-3  
Use of the Sig\_p Bit in the 53C720 - C-1  
User Data Byte Ordering - G-2  
Using the 53C720 in Low Level Mode - H-1

## V



value - 4-4

VLSI - 1-1

## W

WAIT - 4-3

Wait Disconnect Instruction - 4-17

Wait Reselect Instruction - 4-17

Wait Reselect PASS(&alt\_addr) - 4-5

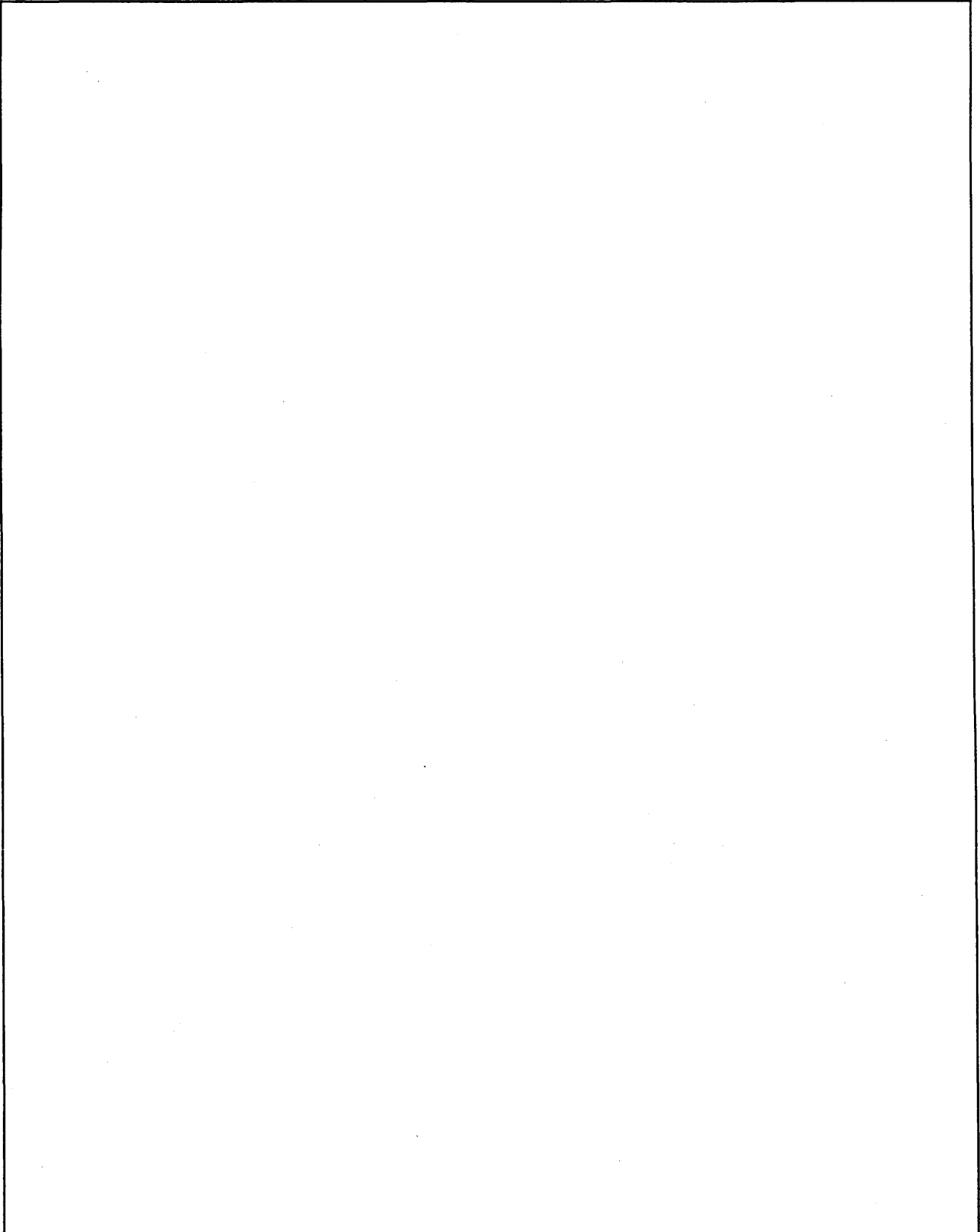
Wait Select Instruction - 4-17

Warning - F-3

WHEN - 4-3

WITH - 4-3

## Notes

A large, empty rectangular box with a thin black border, occupying most of the page. It is intended for the user to write their notes.



# READER'S COMMENT FORM

F-8763 0687

BOOK TITLE	BOOK NO.	PRINT DATE

To help us plan future editions of this document, please take a few minutes to answer the following questions. Explain in detail using the space provided. Include page numbers where applicable.

Are there any technical errors or misrepresentations in the document?

---



---



---

Is the material presented in a logical and consistent order?

---



---



---

Is it easy to locate specific information in the document?

---



---



---

Is there any information you would like to have added to the document?

---



---



---

Are the examples relevant to the task being described?

---



---



---

Could parts of the document be deleted without affecting the document's usefulness?

---



---



---

Did the document help you to perform your job?

---



---



---

Any general comments?

---



---



---

NAME \_\_\_\_\_

TITLE \_\_\_\_\_

COMPANY \_\_\_\_\_

ADDRESS \_\_\_\_\_

TELEPHONE NO. (        ) \_\_\_\_\_

Thank you for your evaluation of this document.  
Fold the form as indicated and mail to NCR. No postage is necessary in the U.S.A.

fold

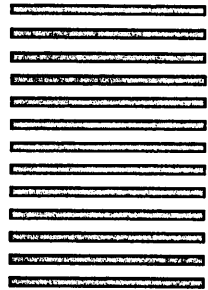


NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 3 DAYTON, OHIO

POSTAGE WILL BE PAID BY ADDRESSEE

**NCR Corporation**  
ATTENTION: Publication Services  
WHQ-4  
1700 S. Patterson Blvd.  
Dayton, Ohio 45409



fold

## **NCR Microelectronic Products Division – Sales Locations**

---

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

**1-800-334-5454**

---

### **Worldwide Sales Headquarters**

731 Technology Drive, Suite 600  
San Jose, CA 95110  
(408) 453-0303

### **Division Plant Locations**

#### **NCR Microelectronic Products Division**

2001 Danfield Court  
Fort Collins, CO 80525  
(303) 226-9500

*Commercial ASIC Products*  
*Customer Owned Tooling Products*  
*Communication Products*  
*Disk Array Products*

#### **NCR Microelectronic Products Division**

1635 Aeroplaza Drive  
Colorado Springs, CO 80916  
(719) 596-5795

*Automotive Products*  
*SCSI Products*  
*Graphics Products*

#### **NCR Microelectronic Products Division**

2850A North El Paso Street  
Colorado Springs, CO 80907  
(719) 578-3400

*Multi-Chip Module Products*

### **North American Sales Offices**

#### **Northwest Sales**

1731 Technology Drive, Suite 600  
San Jose, CA 95110  
(408) 441-1080

#### **Southwest Sales**

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

#### **North Central Sales**

8000 Townline Avenue, Suite 209  
Bloomington, MN 55438  
(612) 941-7075

#### **South Central Sales**

17304 Preston Road, Suite 635  
Dallas, TX 75252  
(214) 733-3594

#### **Northeast Sales**

500 West Cummings Park, Suite 4000  
Woburn, MA 01801  
(617) 933-0778

#### **Southeast Sales**

1051 Cambridge Square, Suite C  
Alpharetta, GA 30201  
(404) 740-9151

### **International Sales Offices**

#### **European Sales Headquarters**

Westendstrasse 193  
8000 Munchen 21  
Postfach 210370  
Germany  
49 89 57931199

#### **Asia/Pacific Sales Headquarters**

35th Floor, Shun Tak Centre  
200 Connaught Road  
Central  
Hong Kong  
852 859 6044

NCR Corporation  
Microelectronic Products Division  
Colorado Springs, CO 80916  
J0592II  
0192-15MB