PROGRAMMING GUIDE

SCSI SCRIPTS™ Processors

Version 2.3

October 2000

LSI LOGIC®

S14044.A

DB15-000159-01, Second Edition (October 2000)
This document describes the LSI Logic Corporation SCSI SCRIPTS™ Processors and will remain the official reference source for all revisions/releases of this product until rescinded by an update.

To receive product literature, visit us at http://www.lsilogic.com.

Ultra SCSI is the term used by the SCSI Trade Association (STA) to describe Fast-20 SCSI, as documented in the SCSI-3 Fast-20 Parallel Interface standard, X3.277-199X.

Ultra2 SCSI is the term used by the SCSI Trade Association (STA) to describe Fast-40 SCSI, as documented in the SCSI Parallel Interface-2 standard, (SPI-2) X3710-1142D.

Preface

This book is the primary reference and programming guide for the LSI Logic PCI to SCSI I/O Processors. It contains a complete functional description for the LSI Logic PCI to SCSI I/O Processors and includes complete physical and electrical specifications for the LSI Logic PCI to SCSI I/O Processors.

Audience

This manual is written for users who are familiar with the SCSI and PCI specifications, and have a working knowledge of computer architectures and programming. It is specifically designed for use with programming the LSI Logic SCSI SCRIPTS™ processor in the following chip families:

- LSI53C7XX
- LSI53C8XX
- LSI53C10XX (up to the LSI53C1010 and LSI53C1010R)

Organization

This document has the following chapters and appendixes:

- Chapter 1, **Using the Programming Guide**, introduces the SCRIPTS processor features and functions, and the parts of the PCI to SCSI system that are involved in operating the chip.
- Chapter 2, **Programming with SCRIPTS**, describes the SCRIPTS processor and programming language in depth, including how SCRIPTS programs are integrated with “C” code to execute SCSI commands.
- Chapter 3, **The SCSI SCRIPTS Processor Instruction Set**, describes the SCRIPTS processor instruction set, along with detailed

functional descriptions and usage guidelines for all of the instructions supported.

- Chapter 4, **Using the LSI Logic Assembler NASM™**, describes use and operation of the LSI Logic Assembler (NASM).
- Chapter 5, **The NASM Output File**, describes the LSI Logic Assembler (NASM) output file.
- Chapter 6, **Using the Registers to Control Chip Operations**, contains functional and address information on the LSI53C7XX/8XX/10XX family chips register set.
- Chapter 7, **Integrating SCRIPTS Programs into "C" Language Drivers**, illustrates the relationship between the SCRIPTS program and the "C" language device driver.
- Chapter 8, **Writing Device Drivers with SCRIPTS**, addresses specific kinds of driver applications, with code samples for all applications discussed.
- Chapter 9, **SCRIPTS Programming Topics**, addresses specific kinds of driver applications, with code samples for all applications discussed.
- Chapter 10, **Multithreaded I/O**, contains guidelines for writing SCRIPTS for multithreaded applications.
- Chapter 11, **Using the SCRIPTS Processor in Target Applications**, provides guidelines that are specific to using the SCRIPTS processor in a target device.
- Chapter 12, **Debugging the SCRIPTS Processor**, provides information on debugging SCRIPTS programs.
- Chapter 13, **New SCRIPTS Processor Features**, provides information on the new 64-bit features of the latest version of this chip family.
- Appendix A, **NASM Error Messages**, provides a list of NASM error messages.
- Appendix B, **Multithreaded SCRIPTS Example**, provides example SCRIPTS code.
- Appendix C, **Glossary of Terms and Abbreviations**, provides definitions of terms and abbreviations.

Related Publications

LSI53C770 SCSI I/O Processor with Ultra SCSI Data Manual, Version 2.0, LSI Logic Corporation, Order Number T18962I

LSI53C810A PCI-SCSI I/O Processor Data Manual, Version 2.0, LSI Logic Corporation, Order Number T07962I

LSI53C815 PCI-SCSI I/O Processor with Local ROM Interface Data Manual, Version 2.0, LSI Logic Corporation, Order Number T10962I

LSI53C825A/825AE PCI-SCSI I/O Processor Data Manual, Version 3.0, LSI Logic Corporation, Order Number T40937I

LSI53C860 PCI-Ultra SCSI I/O Processor Data Manual, Version 2.0, LSI Logic Corporation, Order Number T09962I

LSI53C875/875E PCI-Ultra SCSI I/O Processor Data Manual, Version 4.0, LSI Logic Corporation, Order Number T42984I

LSI53C895 PCI to Ultra2 SCSI I/O Processor with LVD Link™ Universal Transceivers Technical Manual, Version 3.1, LSI Logic Corporation, Order Number S14030

LSI53C895A PCI to Ultra2 SCSI Controller Technical Manual, Version 2.2, LSI Logic Corporation, Order Number S14028.B

LSI53C896 PCI to Dual Channel Ultra2 SCSI Multifunction Controller Technical Manual, Version 3.1, LSI Logic Corporation, Order Number S14015.B

LSI53C1000 PCI to Ultra3 SCSI Controller Technical Manual, Version 2.0, LSI Logic Corporation, Order Number S14050

LSI53C1000R PCI to Ultra160 SCSI Controller Technical Manual, Version 1.0, LSI Logic Corporation, Order Number S14052

LSI53C1010-33 PCI to Dual Channel Ultra3 SCSI Multifunction Controller Technical Manual, Version 3.1, LSI Logic Corporation, Order Number S14025.C

LSI53C1010-66 PCI to Dual Channel Ultra3 SCSI Multifunction Controller Technical Manual, Version 2.0, LSI Logic Corporation, Order Number S14049

LSI53C1010R PCI to Dual Channel Ultra160 SCSI Multifunction Controller Technical Manual, Version 1.0, LSI Logic Corporation, Order Number S14053

Conventions Used in This Manual

The following is a list of notational conventions used throughout this programming guide:

Notation	Example	Meaning and Use
square braces [ ]	`CALL [REL] Address, [{IF | WHEN} [NOT] CARRY]`	Optional items in instruction examples.
courier font	`program.exe`	Used for code samples, file names, command line information, prompts, etc. that appear in body text.
All Caps	`JUMP [REL] Address, [{IF | WHEN} [NOT] CARRY]`	Keywords.
Curly braces {}	`SELECT [ATN] {FROM Address | ID}, [REL] Address`	Choose between items enclosed in curly braces.
{} "..."	`SET {ACK|ATN|TARGET|CARRY} [and {ACK|ATN|TARGET|CARRY}...]`	The character enclosed in the curly braces can be repeated as often as desired.
|	`INTFLY int_value, [{IF | WHEN} [NOT] CARRY]`	OR, select one item from a list.
\	`RELATIVE baselabel \`	Line continuation.

Revision Record

Revision	Date	Remarks
2.0	8/96	Initial release.
2.1	6/97	Added chapter on programming multifunction controllers.
2.2	6/00	Miscellaneous updates/format changes.
2.3	10/00	All product names changed from SYM to LSI.

Contents

Tables

Chapter 1
Using the Programming Guide

This chapter provides an overview of the LSI Logic SCSI SCRIPTS processor. It also provides brief descriptions for some of the chips containing the processor and their features. The chapter contains the following sections:

1.1 Product Overview

The LSI Logic SCSI SCRIPTS processor is based on the LSI53C7XX SCSI I/O Processor family architecture, with a host interface to the Peripheral Component Interconnect (PCI) bus. The SCRIPTS processor connects to the PCI bus without glue logic.

Several LSI Logic product families contain the SCRIPTS processor.

- LSI53C7XX
- LSI53C8XX
- LSI53C10XX (up to the LSI53C1010 and LSI53C1010R)

Tables 1.1 and 1.2 list currently available chips using the SCRIPTS processor and their basic specifications. More detailed information is available in the respective chip technical manuals, listed in Related Publications on page v.

Table 1.1 Features and Functions of LSI53C7XX/8XX/10XX Family Chips (part 1)

	LSI53C770	LSI53C810A	LSI53C860	LSI53C815	LSI53C825A LSI53C825AJ
Maximum Transfer Rate	20 Mbytes/s synchronous (with Wide SCSI)	5 Mbytes/s asynchronous 10 Mbytes/s synchronous	5 Mbytes/s asynchronous 20 Mbytes/s synchronous (with Ultra SCSI)	5 Mbytes/s asynchronous 10 Mbytes/s synchronous	10 Mbytes/s asynchronous 20 Mbytes/s synchronous
DMA FIFO Size (bytes)	96	80	80	64	88 or 536
Synchronous Offset (levels)	16	8	8	8	16
SCRIPTS RAM	4 Kbytes	None	None	None	4 Kbytes
Differential SCSI	No	No	No	No	HVD
Wide SCSI	Yes	No	No	No	Yes
External Memory Interface	No	No	No	Yes	Yes
Instruction Prefetch	Yes	Yes	Yes	No	Yes
Load/Store Instructions	No	Yes	Yes	No	Yes
Enhanced Move Register Capability	No	No	No	No	Yes
SCSI Selected As ID Bits	No	Yes	Yes	No	Yes
Number of 32-bit SCRATCH Registers	1	2	2	2	10
PCI Caching	No	Yes	Yes	No	Yes
Selectable IRQ Disable	No	Yes	Yes	No	Yes
Big/Little Endian Support	Big or Little Endian	Little Endian	Little Endian	Big or Little Endian	Big or Little Endian (except LSI53C825AJ)
PCI Data Bus	N/A	32-Bit	32-Bit	32-Bit	32-Bit

Table 1.1 Features and Functions of LSI53C7XX/8XX/10XX Family Chips (part 1) (Cont.)

	LSI53C770	LSI53C810A	LSI53C860	LSI53C815	LSI53C825A LSI53C825AJ
PCI Addressing	N/A	32-Bit	32-Bit	32-Bit	32-Bit
Package	208 PQFP	100 PQFP	100 PQFP	128 PQFP	160 PQFP

Table 1.2 Features and Functions of LSI53C7XX/8XX/10XX Family Chips (part 2)

	LSI53C875 LSI53C875A LSI53C875J LSI53C875JB LSI53C875N	LSI53C895 LSI53C895A	LSI53C896	LSI53C1000 LSI53C1000R	LSI53C1010 LSI53C1010R
Maximum Transfer Rate	10 Mbytes/s asynchronous 40 Mbytes/s synchronous (with Ultra SCSI)	10 Mbytes/s asynchronous 80 Mbytes/s synchronous (with Ultra2 SCSI)	10 Mbytes/s asynchronous 80 Mbytes/s synchronous per channel for 160 Mbytes/s	10 Mbytes/s asynchronous 160 Mbytes/s synchronous (with Ultra2 SCSI)	10 Mbytes/s asynchronous 160 Mbytes/s synchronous per channel for 320 Mbytes/s
DMA FIFO Size (bytes)	88 or 536	112 or 816 (LSI53C895) 112 or 944 (LSI53C895A)	112 or 944	896 to 920	896 to 920
Synchronous Offset (levels)	16	31	31	62	62
SCRIPTS RAM	4 Kbytes	4 Kbytes (LSI53C895) 8 Kbytes (LSI53C895A)	8 Kbytes	8 Kbytes	8 Kbytes
Differential SCSI	High Voltage Differential (HVD)	Low Voltage Differential (LVD) and HVD	LVD and HVD	LVD and HVD	LVD and HVD
Wide SCSI	Yes	Yes	Yes Dual Channel	Yes	Yes Dual Channel
External Memory Interface	Yes	Yes	Yes	Yes	Yes

Table 1.2 Features and Functions of LSI53C7XX/8XX/10XX Family Chips (part 2) (Cont.)

	LSI53C875 LSI53C875A LSI53C875J LSI53C875JB LSI53C875N	**LSI53C895 LSI53C895A**	**LSI53C896**	**LSI53C1000 LSI53C1000R**	**LSI53C1010 LSI53C1010R**
Instruction Prefetch	Yes	Yes	Yes	Yes	Yes
Load/Store Instructions	Yes	Yes	Yes	Yes	Yes
Enhanced Move Register Capability	Yes	Yes	Yes	Yes	Yes
SCSI Selected As ID Bits	Yes	Yes	Yes	Yes	Yes
Number of 32-bit SCRATCH Register	10	10 (LSI53C895) 18 (LSI53C895A)	18	18	18
PCI Caching	Yes	Yes	Yes	Yes	Yes
Selectable IRQ Disable	Yes	Yes	Yes	Yes	Yes
Big/Little Endian Support	Big or Little Endian (except LSI53C875J, LSI53C875JB)	LSI53C895 Big or Little Endian LSI53C895A Little Endian	Little Endian	Little Endian	Little Endian
PCI Data Bus	32-Bit	32-Bit	64-Bit	64-Bit	64-Bit
PCI Addressing	32-Bit	32-Bit (LSI53C895) 64-Bit (LSI53C895A)	64-Bit	64-Bit	64-Bit
Package	160 PQFP, 169 BGA, 208 PQFP	208 PQFP	329 BGA	329 BGA	329 BGA

The LSI Logic SCSI SCRIPTS processors are the first products to concentrate the functions of an intelligent SCSI adapter board onto a single chip. These products integrate a high-performance SCSI core, a PCI bus master DMA core, and the SCSI SCRIPTS processor to meet

the flexibility requirements of SCSI-3 and future SCSI standards. It executes multithreaded I/O algorithms with minimum host processor intervention, reducing the protocol overhead required for SCSI operations to as little as one interrupt per SCSI I/O. The SCRIPTS language, a high level instruction set, provides complete programmability of I/O operations and supports the flexibility needed for multithreaded I/O algorithms. SCRIPTS provides:

- Phase sequencing without processor intervention
- Automatic bus arbitration
- Data or phase comparison for independent SCSI algorithm decisions
- DMA interface control

All LSI53C7XX/8XX/10XX family chips are also supported by LSI Logic software for connecting SCSI devices. This includes BIOS support for LSI Logic SCSI processors and drivers for most types of SCSI peripherals under the major operating systems. These chips also feature:

- On-chip Single-Ended (SE) drivers
- Synchronous and asynchronous transfer capabilities
- LSI Logic TolerANT® driver and receiver technology
- Bus mastering
- Automatic selection/reselection time-outs
- 32-bit memory addressing
- 32-bit data bus
- PCI bursting

Newer chips, including the LSI53C895, LSI53C895A, LSI53C896, LSI53C1000, LSI53C1010, LSI53C1010R, and LSI53C1000R also have these features:

- On-chip LVD
- 64-bit memory addressing
- 64-bit data bus

Note: For specific information on the features and functions of the various chips supporting SCRIPTS, refer to their respective technical manuals. You must have the appropriate technical

manual in order to effectively program SCRIPTS for each chip.

Figures 1.1 and 1.2 are block diagrams of the single and dual channel LSI Logic chips that support SCRIPTS, with a map of SCSI data and control paths through the chips.

Figure 1.1 Single Channel Block Diagram

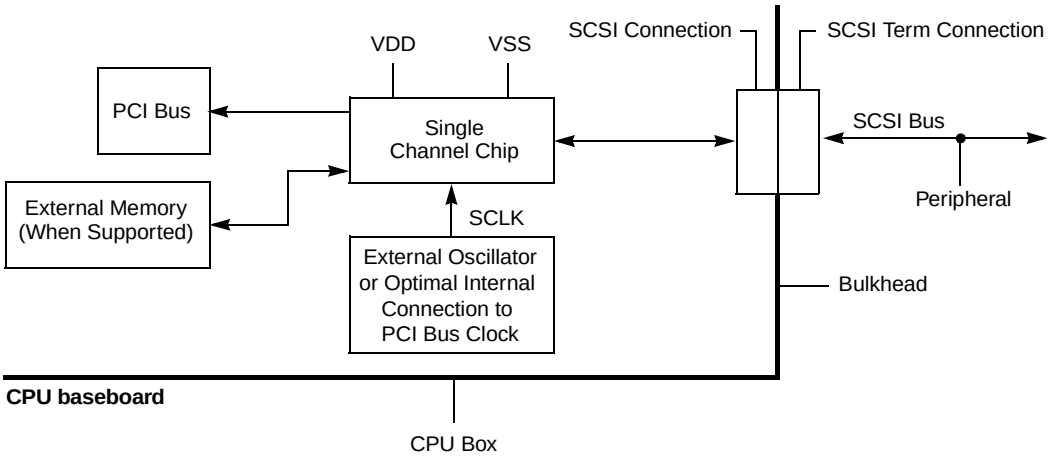
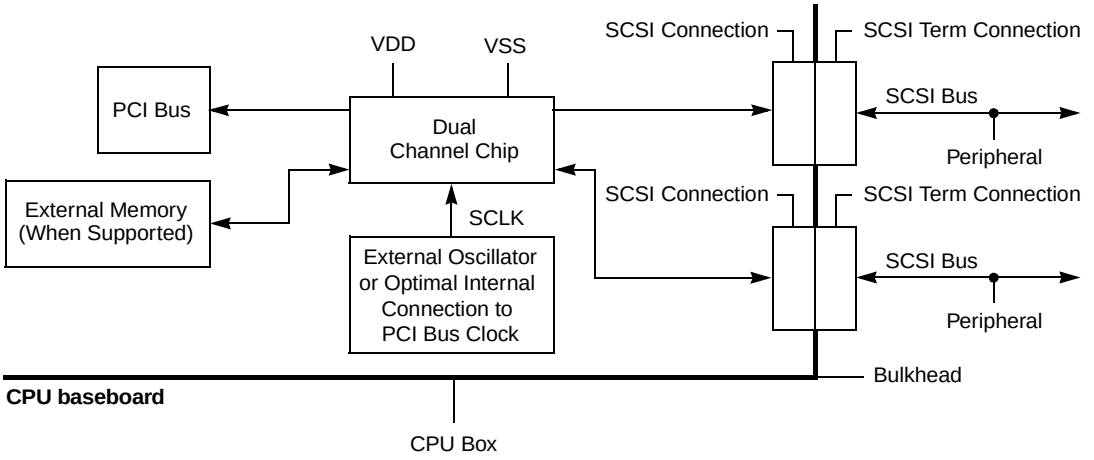


Figure 1.2 Dual Channel Block Diagram



1.2 Benefits of Ultra, Ultra2, and Ultra3 SCSI

Ultra SCSI is an extension of the SCSI-3 standard that expands the bandwidth of the SCSI bus and allows faster synchronous SCSI transfer rates. When enabled, Ultra SCSI performs 20 megatransfers per second, which results in approximately doubling the synchronous transfer rates of Fast SCSI-2. The LSI53C860 and LSI53C875 can perform 8-bit or 16-bit Ultra SCSI synchronous transfers as fast as 20 Mbytes/s or 40 Mbytes/s.

Ultra2 SCSI extends SCSI performance beyond Ultra SCSI rates, up to 40 megatransfers per second. It also defines a new physical interface, LVD SCSI, that retains the reliability of HVD SCSI while allowing a longer cable and more devices on the bus than Ultra SCSI. The LSI53C895 can perform 16-bit, Ultra2 SCSI synchronous transfers as fast as 80 Mbytes/s.

Ultra3 SCSI delivers data up to two times faster than Ultra2 SCSI. Ultra3 SCSI is an extension of the SPI-3 draft standard. When enabled, Ultra3 SCSI performs 80 megatransfers per second. Ultra3 data transfer speed is accomplished using Double Transition (DT) clocking. Data is clocked on both rising and falling edges of the request and acknowledge signals, doubling data transfer speeds without increasing the clock rate.

The advantages of Ultra/Ultra2/Ultra3 SCSI are most noticeable in heavily loaded systems, or large block size applications such as video on demand and image processing. Not only does it significantly improve SCSI bandwidth, it also preserves existing hardware and software investments. LSI Logic Ultra/Ultra2/Ultra3 SCSI chips are all compatible with Fast SCSI software; the only changes required are to enable the chip to negotiate for the faster synchronous transfer rates.

Some changes to existing cabling or system designs may be needed to maintain signal integrity at Ultra SCSI synchronous transfer rates. These design issues are discussed in the chip technical manuals.

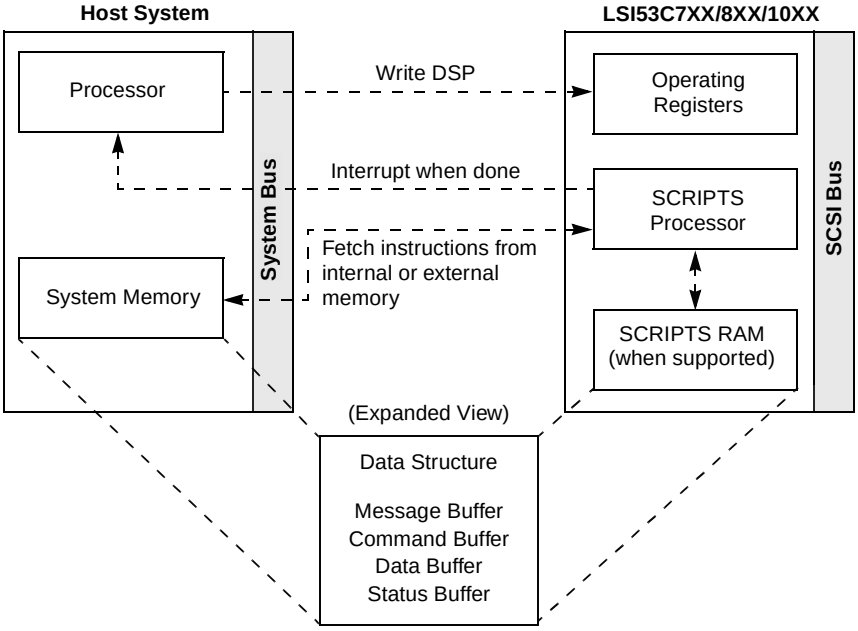
1.3 System Overview

To execute SCSI SCRIPTS programs, only a SCSI SCRIPTS starting address is required. All subsequent instructions are fetched from external memory or internal SCRIPTS RAM (when supported). Depending on the chip, up to eight Dwords at a time are fetched across the DMA interface and loaded into the internal chip registers. When the chip is operating at its highest frequency, instruction fetching and decoding take as little as 500 ns. The chip fetches instructions until a SCRIPTS interrupt occurs or until an external, unexpected event (such as a hardware error) causes an interrupt. The full set of SCSI features in the instruction set allows re-entry to the algorithm at any point. This high level interface can be used for both normal operation and exception conditions.

A typical SCRIPTS operation is illustrated in Figure 1.3. Before SCRIPTS operation begins, the host processor writes the Data Structure Address register value to initialize the pointer for table indirect operations. To begin SCRIPTS operation, the host processor writes the starting address of the SCRIPTS instructions into the chip's DMA SCRIPTS Pointer Register. Once it receives this address, the chip becomes a bus master and fetches the first SCRIPTS instruction. The chip executes all steps of the instruction, moving through the appropriate bus phases, interrupting only on completion of SCRIPTS operation or when service from the external processor is required. This leaves the host processor free for other tasks.

Software developers can create SCSI SCRIPTS source code in any text editor. The LSI Logic Assembler, NASM, is discussed in Chapter 4, "Using the LSI Logic Assembler NASM™." NASM assembles SCRIPTS code into an array of assembled SCRIPTS instructions that can be included in the main "C" language program and linked together to create an executable driver. When compiled, these programs control chip operation.

Figure 1.3 Typical SCRIPTS Operation



Chapter 2
Programming with SCRIPTS

This chapter contains the following sections:

2.1 The SCSI SCRIPTS Processor

The SCSI SCRIPTS processor permits instructions to be fetched from internal or external memory. Algorithms written in the SCSI SCRIPTS language are assembled to control the SCSI and DMA modules. Complex SCSI bus sequences, including multiple SCRIPTS instructions, execute independently of the host processor.

The SCSI SCRIPTS reside in host computer memory or internal SCRIPTS RAM during system operation, allowing for fast execution. If instructions reside in external memory, the chip fetches SCRIPTS programs from memory using bus master DMA transfers. If instructions reside in SCRIPTS RAM, they are fetched directly from RAM without generating PCI bus traffic. The SCRIPTS processor allows you to fine tune SCSI operations such as adjusting to new device types, adapting to changes in SCSI logical definitions, or quickly incorporating new options, such as vendor unique commands or new SCSI specifications. The SCRIPTS processor fetches SCRIPTS instructions from system memory to control chip operation. The SCRIPTS processor does not compile code; SCRIPTS programs must be assembled for execution by the NASM assembler and then compiled with a standard “C” compiler as part of a “C” program. Third generation SCSI devices can be programmed

with SCRIPTS using only a few hundred lines of SCRIPTS code. SCRIPTS are independent of the CPU, operating system, or system bus being used, so they are portable across platforms.

Important: The SCRIPTS processor is not used in chip families subsequent to the LSI53C1010 and LSI53C1010R.

2.2 SCRIPTS and the SCSI Bus Phases

One important advantage of SCSI SCRIPTS is that the SCRIPTS language corresponds directly to SCSI protocol. In conjunction with the high level language syntax, it provides an excellent vehicle to master the complexity of SCSI. The one-to-one relationship between protocol phases and SCRIPTS instructions means that SCRIPTS can be customized to specific operations on the SCSI bus, and that SCSI software development is simplified by using SCRIPTS. SCSI uses the bus phases in the order shown in Table 2.1. This table also shows the SCSI SCRIPTS instructions that correspond to the SCSI bus phases for initiator and target roles.

Table 2.1 SCSI Protocol and SCRIPTS Instructions

Bus Phase	Definition	SCRIPTS Instruction (Initiator role)	SCRIPTS Instruction (Target role)
Bus Free	This phase indicates that the SCSI bus is available.	N/A	N/A
Arbitration	This phase allows the initiator to gain control of the SCSI bus.	SELECT ATN	RESELECT
Selection	During this phase, the initiator selects a target device to perform the desired function. The Attention option notifies the target that upon successful selection the initiator desires to send further messages.	SELECT ATN	WAIT SELECT
Reselection	The target reselects with the initiator during this phase.	WAIT RESELECT	RESELECT
Message-Out	During this phase, the initiator can send messages to the target, such as queuing or error recovery information.	MOVE WHEN MSG_OUT	MOVE WITH MSG_OUT

Table 2.1 SCSI Protocol and SCRIPTS Instructions (Cont.)

Bus Phase	Definition	SCRIPTS Instruction (Initiator role)	SCRIPTS Instruction (Target role)
Command	During this phase, the initiator can send a command in the form of a command descriptor block (CDB) to the target buffer.	MOVE WHEN CMD	MOVE WITH CMD
Data In/Out	Data In and Data Out phases are used to send data to the initiator or to the target and are used dependent on the information transferred during the Command phase. This phase is optional. For example, a Test Unit Ready command does not require a data transfer.	MOVE	MOVE
Status	During this phase, the initiator receives status information from the target about the previously executed CDB.	MOVE WHEN STATUS	MOVE WITH STATUS
Message-In	During this phase, the initiator will receive messages from the target. These messages can acknowledge or reject previously sent initiator messages. They also can provide other information like queuing, disconnect, or parity errors.	MOVE WHEN MSG_IN	MOVE WITH MSG_IN
Disconnect	This phase is used to end the initiator's connection with the bus. After successful completion of an I/O operation and a request for disconnect, the bus returns to the Bus Free state, indicating that it is now available.	WAIT DISCONNECT WAIT DISCONNECT	DISCONNECT DISCONNECT

2.3 Assembling SCSI SCRIPTS

The SCSI SCRIPTS are assembled with the LSI Logic Assembler (NASM), a DOS command line driven assembler that supports LSI Logic SCSI SCRIPTS processors. NASM assembles SCSI SCRIPTS for inclusion in SCSI device driver software programs. NASM is described in detail in Chapter 4, “Using the LSI Logic Assembler NASM™.”

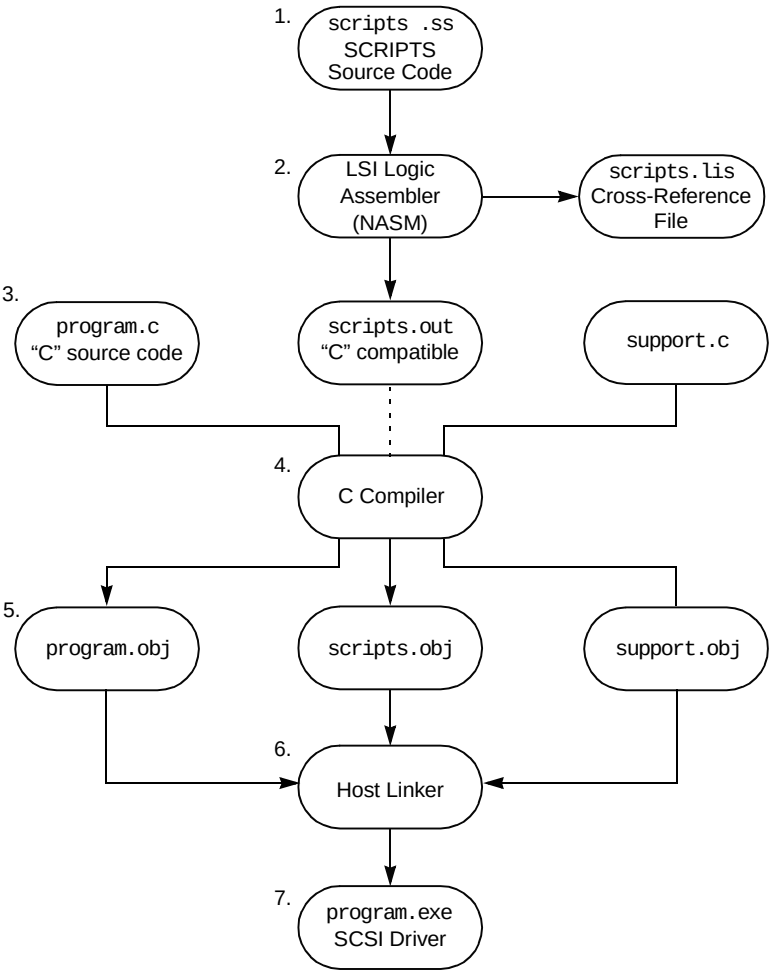
The SCSI SCRIPTS programs are created with any text editor that generates ASCII files. These text files must be transformed from their text form into the SCRIPTS processors instruction language before they can be executed by the SCRIPTS processor. This is accomplished by running the test file through NASM. NASM generates an output file (`.out`) that is

compatible with all standard “C” compilers, as well as a cross-reference list file (`.lis`) that includes the source instruction and the assembled output on the same line. The `.lis` file is useful for debugging code. All instructions and data are represented as hexadecimal numbers in C style array declarations. The `.out` file can be included in the “C” program and linked together with other system support object files to form the final executable code.

When the executable is run, areas of host memory are reserved for SCSI data transfer buffers and the SCRIPTS instructions. The instructions, which look like 32-bit integer arrays to the “C” program, are loaded into the appropriate area of memory by the “C” code. The driver program loads the address of the first instruction into the SCRIPTS processor to begin the SCRIPTS execution.

Figure 2.1 illustrates an overview of assembling the SCSI SCRIPTS.

Figure 2.1 Overview of Assembling SCSI SCRIPTS



1. Write SCSI SCRIPTS source code.
2. Assemble the source code using the LSI Logic Assembler (NASM).
3. Write "C" language source code and include assembled SCRIPTS code.
4. Compile all code using a "C" compiler.
5. The result is object (`.obj`) code.
6. Link all object modules together.
7. The result is an executable program.

2.4 Using SCSI SCRIPTS

The following section of the chapter describes various aspects of SCSI SCRIPTS.

2.4.1 SCRIPTS Data Sizes

Table 2.2 describes SCSI SCRIPTS data sizes.

Table 2.2 Data Sizes

Address	a 32-bit number
Value	a 32-bit number
Count	a 24-bit number
Data	an 8-bit number
ID	a 4-bit encoded SCSI ID

2.4.2 SCSI SCRIPTS Language Elements

Table 2.3 describes the SCSI SCRIPTS language elements.

Table 2.3 SCSI SCRIPTS Language Elements

Term	Definition
name	A name is a string of one or more consecutive characters. It may consist of letters, numbers, underscores, and dollar signs, but must begin with an alphabetic character. When used for labels, externals, and variables in the relative data area, names are passed on to the host development system and are subject to the host's syntactic restrictions. Names cannot be reserved words in the host language. For example, Turbo C, which is used as the host development system for NASM, does not allow names to begin with a digit or to contain a dollar sign ($). Therefore, the SCSI SCRIPTS writer for DOS and Turbo C should avoid names of this form.
label	A label is a name followed by a colon. Labels are symbolic addresses that can be used as transfer control destination points, such as jump or call destinations. Labels are case sensitive.
comment	Comments are used to notate the SCRIPTS. They are optional and are ignored by the compiler. Comments begin with a semicolon and continue to the end of a line.

2.4.3 SCSI SCRIPTS Expressions

There are two forms of SCSI SCRIPTS operators, arithmetic and bitwise, described in Table 2.4 and Table 2.5.

Table 2.4 Arithmetic Operators

Symbol	Meaning
+	addition
–	subtraction

Table 2.5 Bitwise Operators

Symbol	Meaning
&	Logical AND
|	Logical OR
XOR	Exclusive OR
SHL	Shift left
SHR	Shift right

The value of all expressions is 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 are used. For example, if an expression is used to represent a count, normally 24 bits, for a Move instruction, the evaluated value is truncated to 24 bits. You are notified if the expression has been truncated and if the value of the expression changes during truncation. The symbols for the bitwise operators are used only for register manipulations. Any other instruction using comparison must spell out AND or OR.

2.4.4 SCSI SCRIPTS Keywords

The SCSI SCRIPTS keywords have eight types: Declarative, Conditional, Logical, Flag Field, Qualifier, Action, SCSI Phase, and Register Name. Keywords are written in all capital letters for clarity, but are not case sensitive. Refer to Chapter 4, “Using the LSI Logic Assembler NASM™,” for detailed descriptions of individual keywords.

2.5 Big and Little Endian Byte Addressing

The guidelines in this section will help assure proper byte lane ordering in big or little endian designs. Please check the technical manual for each chip to determine whether your product supports big and/or little endian addressing. The later series of chips that have 64-bit addressing are all little endian.

Big endian addressing is used primarily in designs based on Motorola processors. The SCRIPTS processor treats D[31:24] as the lowest physical memory address. Little endian is used primarily in designs based on Intel processors and treats D[7:0] as the lowest physical memory address.

Table 2.6 describes big and little endian byte addressing.

Table 2.6 Big and Little Endian Byte Addressing

System Data Bus	[31:24]	[23:16]	[15:8]	[7:0]
Pins	[31:24]	[23:16]	[15:8]	[7:0]
Register	SCNTL3	SCNTL2	SCNTL1	SCNTL0
Little Endian Address	0x03	0x02	0x01	0x00
Big Endian Address	0x00	0x01	0x02	0x03

2.5.1 SCRIPTS Instruction Sequence

To ensure that SCSI SCRIPTS instructions are in the correct order, each SCRIPTS routine must be compiled in the target architecture. The “C” output file (.OUT) lists arrays of Dword values, which are stored in memory by the processor in the correct order for their subsequent execution. Execution of a little endian SCRIPTS instruction on a big endian machine requires reversal of the bytes before execution. The best way to guarantee correct byte ordering is to make sure the SCRIPTS are placed in memory with the opcode byte on the same byte lane as the Data Command (DCMD) register. A PROM cannot be moved from one environment to another without reordering bytes within each word.

2.5.2 Operating Register Access from Firmware

Developing code that works in either mode requires use of equates for the register names, with an endian switch specified at compile time that includes the appropriate set of address values. This change is only for byte access. If 32 bits are accessed, there is no address change from big to little endian.

2.5.3 Operating Register Access from SCRIPTS Routines

NASM uses logical names to access registers. Names do not change when the mode changes, nor does the binary code required to access a register.

2.5.4 User Data Byte Ordering

Data transfers between system memory and the SCSI bus always start at the beginning address and continue until the last byte is sent. No internal reordering 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, regardless of the big or little endian mode.

Chapter 3
The SCSI SCRIPTS Processor Instruction Set

This chapter describes the LSI Logic SCSI SCRIPTS processor instruction set and contains the following sections.

3.1 Overview of SCRIPTS Instructions

This section contains an overview of the instruction types supported by the SCRIPTS processor. Instruction types are groups of commands with similar functions. The commands for each instruction type, including all legal forms, are described in detail in Sections 3.2 and 3.3.

3.1.1 I/O Instructions

The I/O instruction type is selected when the two high order bits of the DCMD register are 0b01, with opcode bit values of 0b000–0b100. I/O instructions perform SCSI operations such as selection and reselection. Each function is a direct command to the SCRIPTS processor. The I/O operations, chosen with the opcode bits in the DCMD register, are described in Table 3.1.

Table 3.1 Opcode Bit Options

Opcode	Target	Initiator
0b000	RESELECT	SELECT, SELECT with ATN
0b001	DISCONNECT	WAIT for DISCONNECT
0b010	WAIT for SELECT	WAIT for RESELECT
0b011	SET	SET
0b100	CLEAR	CLEAR

3.1.2 Memory Move Instructions

The Memory Move Instruction type is selected when the two high order bits of the DCMD register are 0b11. Memory Moves allow data transfer from one 32-bit memory location to another. The source or the destination may be a chip register. A 24-bit byte counter allows large moves to occur with no intervention from the host processor. If both addresses are in system memory, the device functions as a high speed DMA controller, able to move data at sustained speeds up to 40 Mbytes/s without using the host processor or its cache memory. Data is moved from the source address into the chip's DMA FIFO and then out to the destination address. This instruction type does not allow indirect addressing, so the physical 32-bit address must be in the SCRIPTS instruction.

In chips supporting instruction prefetching, the NOFLUSH qualifier prevents flushing the prefetch buffer when the chip performs a Memory-to-Memory Move instruction.

3.1.3 Transfer Control Instructions

The Transfer Control instruction type is selected when the two high order bits of the DCMD register are 0b10. Transfer Controls perform SCRIPTS operations such as JUMP, CALL, RETURN, and INTERRUPT. These instructions allow comparisons of current phase values on the SCSI bus or the first byte of data on any asynchronous incoming bytes, and transfer control to another address depending on the results of the comparison test. These operations may conduct a test of the ALU carry

bit, and may enable interrupt on the fly, so that the interrupt instruction does not halt the SCRIPTS processor.

3.1.4 Read/Write Instructions

The Read/Write Instruction type is selected when the two high order bits of the DCMD register are 0b01, with the opcode bit values from 0b101–0b111. Read/Write instructions perform the following register operations, depending on the value of the operator bits in the Move Register instructions. Table 3.2 describes these instructions.

Table 3.2 Read/Write Instructions

Instruction Type	Definition
Move from SFBR	Moves the SCSI First Byte Received (SFBR) register (0x08) to a specified register address.
Move to SFBR	Moves a specified register value to the SFBR register.
Read/Modify/Write	Reads a specified register, modifies it, and writes the result back into the same register.

3.1.5 Block Move Instructions

The Block Move instruction type is selected when the two high order bits of the DCMD register are 0b00. Block Moves transfer data (user data or SCSI information) between user memory and the SCSI bus. Data comes from any memory address, so scatter/gather operations for user data are transparent to the chip and the external processor. A separate Block Move instruction is written for each piece of data being moved. This instruction allows indirect and table indirect addressing.

3.1.6 Load and Store Instructions

The Load/Store instruction type is selected when the three high order bits of the DCMD register are 0b111. Load and Store instructions are a more efficient way to move data directly between memory and an internal register than the Memory Move instruction. This is due to the fact that they utilize two Dwords instead of three and require one PCI bus ownership instead of two. Load and Store instructions move a maximum

of four bytes. The memory address may map to external memory space or to the SCRIPTS RAM.

Note: Load and Store instructions are not available to all LSI53C7XX/8XX/10XX family chips. Refer to your chip technical manual to determine if your specific device uses Load and Store.

3.2 Instruction Descriptions

The SCRIPTS instructions are shown in Table 3.3, grouped by instruction type. The individual instruction entries list the LSI53C7XX/8XX/10XX family members that support each instruction.

Table 3.3 SCRIPTS Instructions Set

Instruction Type	Commands
I/O	RESELECT, SELECT, SELECT WITH ATN, DISCONNECT, WAIT DISCONNECT, WAIT SELECT, WAIT RESELECT, SET, CLEAR
Memory Move	MOVE MEMORY
Transfer Control	JUMP, JUMP64, CALL, RETURN, INTERRUPT, INTFLY
Read/Write	MOVE REGISTER
Block Move	MOVE, MOVE64, CHMOV, CHMOV64
Load/Store	LOAD, STORE

The following sections in this chapter describe each command. The sections each have:

- SCRIPTS command example
- Description of the SCRIPTS clauses
- Register contents overview
- Register field and bit descriptions
- List of legal command forms

Each command description may also have additional command specific information.

3.2.1 CALL

```
CALL {REL(Address) | Address} [, {IF | WHEN}[NOT][ATN |
Phase] [AND | OR] [data[AND MASK data]]]
CALL {REL(Address) | Address} [, {IF | WHEN}[NOT][Carry]
```

Supported by All LSI Logic SCRIPTS Processors.

Definition SCSI Transfer Control, Call subroutine.

Operands This command has the following operands:

REL Indicates the use of relative addressing by setting the high order bit in the DMA Byte Counter (DBC) register.

Address Location to which execution is transferred if the subroutine is called. Stored in the second Dword of the instruction.

WHEN Forces the SCRIPTS engine to wait for a valid SCSI bus phase before continuing. A valid phase is indicated by assertion of the SREQ/ signal.

IF Causes the SCRIPTS processor to immediately check for a valid SCSI bus phase. IF should not be used when comparing for a phase as this could yield unpredictable results. The only exception is using a WHEN conditional just prior to the IF conditional for any given sequence of phase checks.

NOT Negates the comparison. It clears the True bit if present, otherwise the True bit is set.

Phase Specifies the Message, Command/Data, and Input/Output bit values that identify the SCSI phase in the instruction. The desired phase value is compared with the actual values of the SCSI phase lines before the SCRIPTS processor performs the instruction. Only valid for initiator mode and should not be used in the target mode.

ATN Indicates that a jump should take place based on an initiator SATN/ signal. Valid only for the target mode and should not be used in the initiator mode.

data Represents an 8-bit value that is stored in the data field of the instruction when this field is present. In addition, the Compare Data bit is set.

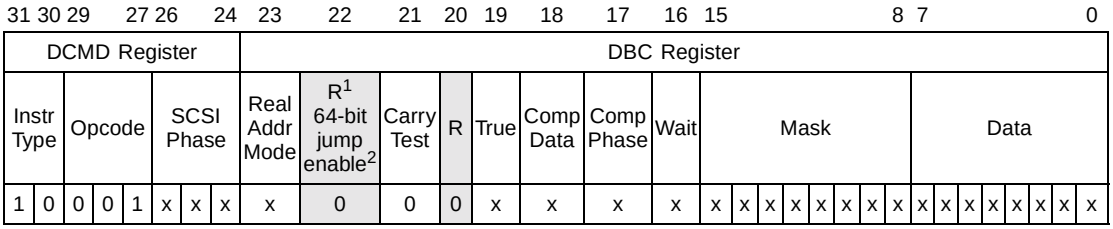
MASK Represents an 8-bit value that is stored in the mask field of the instruction when this field is present. Any bit that is set in the mask causes the corresponding bit in the data byte to be ignored at the time of the comparison.

CARRY Indicates that a jump should take place based on the value of the carry bit in the ALU. Carry comparisons cannot take place at the same time as data and phase comparisons.

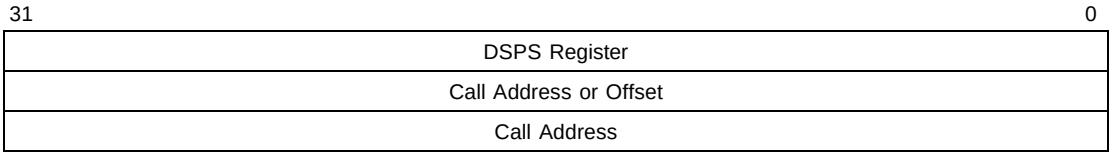
Example

```
CALL REL (Address), WHEN DATA_OUT
```

Figure 3.1 CALL Format



1. All chips except LSI53C10XX.
2. LSI53C10XX chips.



Field(s) This command has the following fields:

Opcode Transfer Control Instruction, Call subroutine.

SCSI Phase These bits reflect the actual values of the SCSI phase lines.

The values in Table 3.4 define the SCSI information transfer phase. The LSI53C10XX chips, with dual transition timing capabilities define two transfer phases, ST for single transition timing, and DT for dual transition timing.

Table 3.4 SCSI Phase Bit Values (CALL Format)[1]

Phase	Message	Command/Data	Input/Output
DATA_OUT[2] (ST_DATA_OUT)[3]	0	0	0
DATA_IN[2] (ST_DATA_IN)[3]	0	0	1
COMMAND	0	1	0
STATUS	0	1	1
RES4[4] (DT_DATA_OUT)[3]	1	0	0
RES5[4] (DT_DATA_IN)[3]	1	0	1
MESSAGE_OUT	1	1	0
MESSAGE_IN	1	1	1

1. 0 - False, negated; 1 - True, asserted. For these phases, SEL is negated and BSY is asserted.
2. All chips except LSI53C10XX.
3. LSI53C10XX chips.
4. RES4 and RES5 are reserved SCSI phases except in the LSI53C10XX chips.

Register Definition(s)

The information listed below describes the DBC and DSPS registers:

Relative Address Mode Relative Addressing Mode indicates that the 24-bit value in DSPS is to be used as an offset from DSP.

Carry Test When this bit is set, True/False comparisons may be made based on the ALU Carry bit.

True Transfer on TRUE/FALSE condition.
0 - Transfer if condition is FALSE
1 - Transfer if condition is TRUE

Compare Data Compare data byte to first byte of the received data.
0 - Do not compare data
1 - Perform comparison

Compare Phase Compare current SCSI phase to SCSI phase field or SATN/. This bit is set whenever the Phase operand is used.
0 - Do not compare phase
1 - Perform comparison

Wait	Wait for valid phase. This bit is set by the WHEN operand in the instruction, and cleared by the IF operand. 0 - Perform comparison immediately 1 - Wait for valid phase (SREQ/ asserted by target)
Mask	An 8-bit field that masks the value in SFBR before the comparison with the data field in the instruction takes place. As a result, any bits in the data byte that correspond to set bits in the mask field are ignored. If this field is not specified, a mask of 0x00 is used.
Data	An 8-bit field that is compared with the incoming data in SFBR after the mask operation of the mask byte takes place. Comparison indicates either an equal or not equal condition. If the Data field is not specified, the compare data bit is cleared and 0x00 is coded for both the mask and data bytes.
Call Address	A 32-bit address (or 24-bit offset, if relative addressing is used) where execution continues if the subroutine is called.

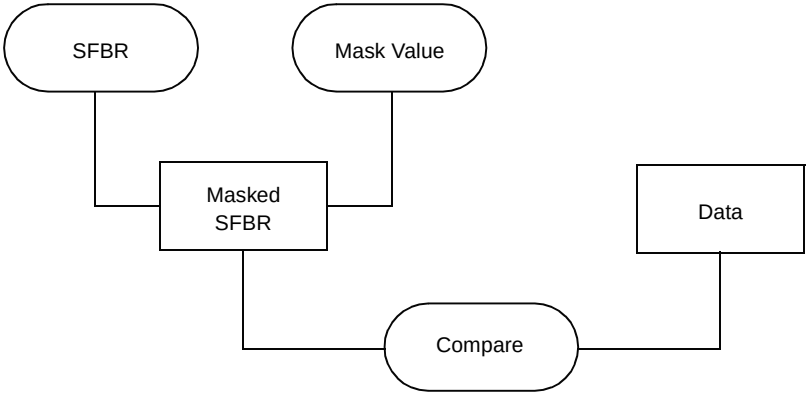
Description

The SCSI CALL instruction is a conditional subroutine call that fetches the next SCRIPTS instruction from memory at either the 32-bit call address or 24-bit offset. It is invoked if all conditions in the instruction or data are met. If the comparison is false, the SCRIPTS processor does not branch to the destination but instead fetches the next inline instruction and continues execution. If the subroutine is called, the next inline instruction address is stored in the chip's Temporary (TEMP) register, and is restored to the DMA SCRIPTS Pointer (DSP) register in response to a RETURN instruction following the CALL.

When the optional data field is used, it is compared to the first byte of the most recent asynchronous data, message, command, or status byte received. The user's SCSI SCRIPTS program can determine which routine to execute next based on actual data values received. Using a series of these compares, the algorithm can process complex sequences without intervention by the external processor.

When the optional MASK keyword and its associated value are specified, the SCRIPTS processor allows selective comparisons of bits within the data byte. This comparison is illustrated in Figure 3.2. During the comparison, any bits that are set in the mask data will cause the corresponding bit in the data byte to be ignored for the comparison.

Figure 3.2 Use of the Mask Keyword



Note: SCRIPTS does not directly support nested CALLs. If two CALL instructions are issued without any intervening RETURN instruction, the first return address in the chip's TEMP register is overwritten by the second CALL and lost. The REL keyword, which indicates relative addressing, is unrelated to the declarative keyword RELATIVE that establishes relative buffers.

Legal Forms

```
CALL address
CALL address, IF ATN
CALL address, IF Phase
CALL address, IF CARRY
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 CARRY
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 CARRY
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 CARRY
CALL address, WHEN NOT data
CALL address, WHEN NOT data AND MASK data
CALL address, WHEN NOT Phase OR data
CALL address, WHEN NOT Phase OR data AND MASK data
CALL REL(address)
CALL REL(address), IF ATN
CALL REL(address), IF Phase
CALL REL(address), IF CARRY
CALL REL(address), IF data
CALL REL(address), IF data AND MASK data
CALL REL(address), IF ATN AND data
CALL REL(address), IF ATN AND data AND MASK data
CALL REL(address), IF Phase AND data
CALL REL(address), IF Phase AND data AND MASK data
CALL REL(address), WHEN Phase
CALL REL(address), WHEN CARRY
CALL REL(address), WHEN data
CALL REL(address), WHEN data AND MASK data
CALL REL(address), WHEN Phase AND data
CALL REL(address), WHEN Phase AND data AND MASK data
CALL REL(address), IF NOT ATN
CALL REL(address), IF NOT Phase
CALL REL(address), IF NOT CARRY
CALL REL(address), IF NOT data
CALL REL(address), IF NOT data AND MASK data
CALL REL(address), IF NOT ATN OR data
CALL REL(address), IF NOT ATN OR data AND MASK data
CALL REL(address), IF NOT Phase OR data
CALL REL(address), IF NOT Phase OR data AND MASK data
CALL REL(address), WHEN NOT Phase
CALL REL(address), WHEN NOT CARRY
CALL REL(address), WHEN NOT data
CALL REL(address), WHEN NOT data AND MASK data
CALL REL(address), WHEN NOT Phase OR data
CALL REL(address), WHEN NOT Phase OR data AND MASK data
```

3.2.2 CHMOV

```
CHMOV {FROM | count,} [PTR] address,{WITH | WHEN} phase
```

Supported by LSI53C825A, LSI53C875, LSI53C876, LSI53C885, LSI53C895, LSI53C895A, LSI53C896, LSI53C1000, LSI53C1010, LSI53C1010R, LSI53C1000R.

Definition Wide SCSI Block Move.

Operands This command has the following operands:

FROM Indicates table indirect addressing mode.
Note: FROM and PTR must not be used in the same instruction.

count Number of bytes to transfer across the SCSI bus.

PTR Sets the indirect bit if present, it is cleared otherwise.
Note: PTR and FROM must not be used in the same instruction.

address The 32-bit starting address of the data in memory, unless PTR is present. If PTR is present, address represents the location of the starting address.

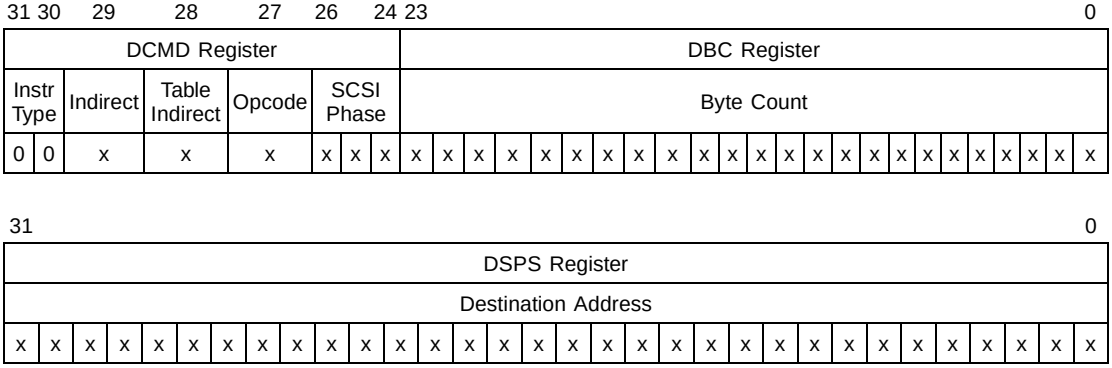
WITH/WHEN Sets device mode; WITH for target mode and WHEN for initiator mode.

Phase Specifies the Message, Command/Data, and Input/Output bit values that identify the SCSI phase in the instruction. The desired phase value is compared with the actual values of the SCSI phase lines before the SCRIPTS processor performs the instruction. This field is only valid for the initiator mode and should not be used in the target mode.

Example

```
CHMOV FROM dev_1 WITH Data_In
CHMOV 6, data_buf, WHEN Data_Out
```

Figure 3.3 CHMOV Format



Field(s) This command has the following fields:

Instruction Type Block Move.

Indirect Indirect Addressing Mode.
0 - Use destination field as an address
1 - Use destination field as an address to an address

Table Indirect Table Indirect Addressing Mode.
0 - Use Absolute addressing mode
1 - Use destination address as offset from the value of Data Structure Address (DSA) register

Opcode Defines whether the instruction will be executed as a Block Move or a Chained Block Move. This bit has different meanings, depending on whether the chip is operating in the target or initiator mode.

	Target	Initiator
MOVE	Opcode = 0	Opcode = 1
CHMOV	Opcode = 1	Opcode = 0

The values in Table 3.5 define the SCSI information transfer phase. The LSI53C10XX chips, with dual transition timing capabilities define two transfer phases, ST for single transition timing, and DT for dual transition timing.

Table 3.5 SCSI Phase Bit Values (CHMOV Format)[1]

Phase	Message	Command/Data	Input/Output
DATA_OUT[2] (ST_DATA_OUT)[3]	0	0	0
DATA_IN[2] (ST_DATA_IN)[3]	0	0	1
COMMAND	0	1	0
STATUS	0	1	1
R4[4] (DT_DATA_OUT)[3]	1	0	0

Table 3.5 SCSI Phase Bit Values (CHMOV Format)[1] (Cont.)

Phase	Message	Command/Data	Input/Output
R5[4] (DT_DATA_IN)[3]	1	0	1
MESSAGE_OUT	1	1	0
MESSAGE_IN	1	1	1

1. 0 - False, negated; 1 - True, asserted. For these phases, SEL is negated and BSY is asserted.
2. All chips except LSI53C10XX.
3. LSI53C10XX chips.
4. RES4 and RES5 are reserved SCSI phases except in the LSI53C10XX chips.

Register Definition(s)

The information listed below describes the DBC and DSPS registers:

SCSI Phase These bits reflect the actual values of the SCSI phase lines.

Byte Count A 24-bit number indicating the number of bytes to transfer.

Dest Addr Address to perform data transfer on, or offset from the DSA to fetch table indirect information.

Description

There are various forms of the Chained Block Move instruction. The “address” and “count” specify the address and byte count fields of the instruction. If the optional keyword “PTR” is present, the indirect bit is set. If PTR is present, the address specified in the instruction is the address of the pointer to the data in memory. “Phase” specifies the phase field of the instruction. WITH or WHEN specify the Block Move function codes. WITH signals the target role which sets the phase values, and WHEN is the initiator “test for phase” feature.

The SCRIPTS processor waits for a valid phase (initiator) or drives the phase lines (target). In the initiator role, it performs a comparison looking for a match between the phase specified in the SCRIPTS instruction and the actual value on the bus. If the phases do not match, an external interrupt occurs. A test prior to the Move instruction could be used to avoid this interrupt. If the phase does match, data is then transferred in or out according to the phase lines. When the count goes to zero, the SCRIPTS processor fetches the next sequential SCRIPTS instruction.

The Chained Move instruction transfers data to and from memory locations. Data may come from any data location, so scatter/gather operations are transparent to the chip and external processor.

When the SCRIPTS processor executes several CHMOV instructions and the ends are on an odd byte boundary, the chip temporarily stores the residual byte in the SCSI Output Data Latch (SODL) register (send operations) or SCSI Wide Residue Data (SWIDE) register (receive operations). The SCRIPTS processor takes the first byte from the subsequent CHMOV or MOVE instruction and lines it up with the residual byte in order to complete a wide transfer and maintain a continuous wide data flow on the SCSI bus.

For more information on Chained Block Move Instructions, please see the appropriate chip technical manual.

Legal Forms

```
CHMOV count, address, WITH phase
CHMOV count, address, WHEN phase
CHMOV count, PTR address, WITH phase
CHMOV count, PTR address, WHEN phase
CHMOV FROM address, WITH phase
CHMOV FROM address, WHEN phase
```

3.2.3 CLEAR

```
CLEAR {ACK | ATN | TARGET | CARRY} [and{ACK | ATN | TARGET
| CARRY} ... ]
```

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition Deasserts SCSI ACK or ATN, or clears internal flags.

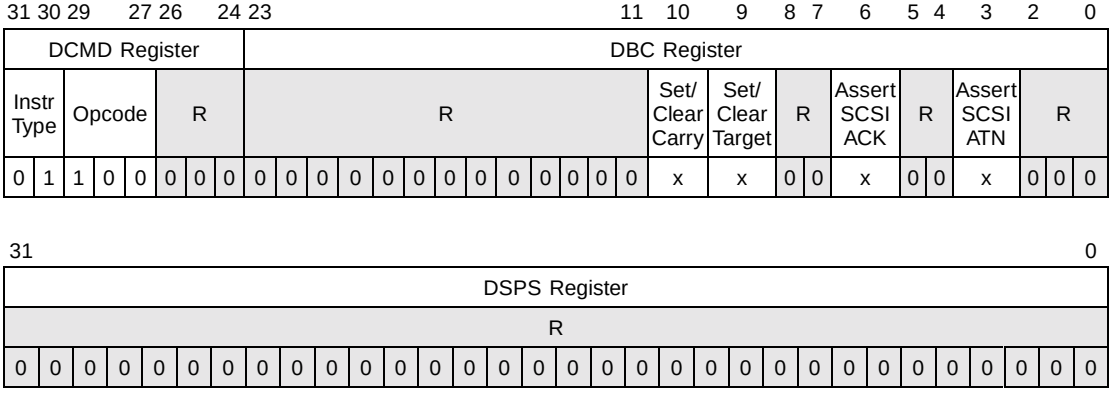
Operands This command has the following operands:

ACK	Clears the Assert SCSI ACK bit.
ATN	Clears the Assert SCSI ATN bit.
TARGET	Clears the Set Target role bit.
CARRY	Clears the CARRY bit in the ALU.

Example

```
CLEAR TARGET
CLEAR ACK and TARGET
```

Figure 3.4 CLEAR Format



Field(s) This command has the following fields:

Instruction Type I/O.

Opcode Clear instruction.

Set/Clear Carry
1 - clears the Carry bit in the ALU
0 - has no effect

Set/Clear Target Mode
1 - places the chip into initiator mode
0 - has no effect

Set/Clear SCSI ACK
1 - deasserts the SCSI acknowledge signal
0 - has no effect

Set/Clear SCSI ATN
1 - deasserts the SCSI attention signal
0 - has no effect

Description The chip deasserts the signals indicated in the instruction. Currently four bits are defined, clearing the SCSI SACK, target role, and SATN bits as well as the CARRY bit in the ALU. Bit 10 is for CARRY, bit 9 is for target, bit 6 is for Acknowledge, and bit 3 is for Attention.

Legal Forms

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

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

3.2.4 DISCONNECT

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition Perform disconnect.

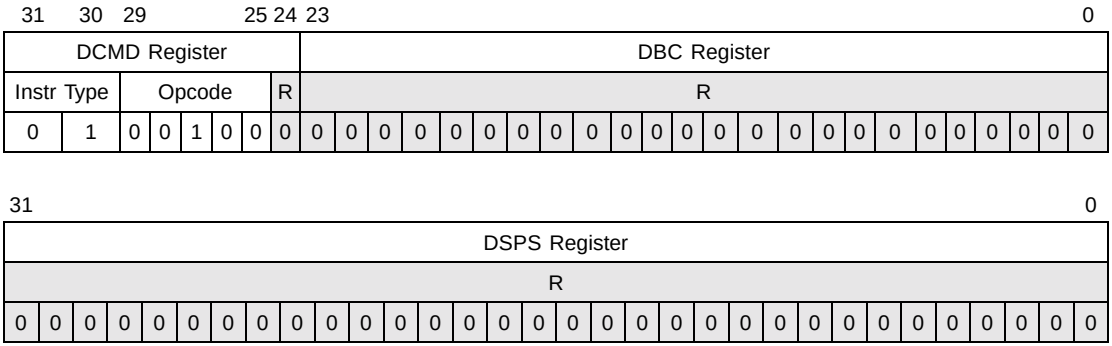
Operands This command has the following operands:

None.

Example

```
DISCONNECT
```

Figure 3.5 DISCONNECT Format



Field(s) This command has the following fields:

Instruction Type I/O.

Opcode Disconnect instruction.

Description The DISCONNECT instruction physically disconnects the chip from the bus when in the target mode.

Notes This instruction has no effect on the initiator when issued by a target. To disconnect from the SCSI bus, use the SET TARGET instruction before this instruction.

Legal Forms: `DISCONNECT`

3.2.5 INT

```
INT int_value [, {IF | WHEN}[NOT][ATN | Phase][AND | OR]
[data[AND MASK data]]]
INT int_value [, {IF | WHEN}[NOT] CARRY]
```

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition SCSI Transfer Control - Generate Interrupt and halt SCRIPTS operation.

Operands This command has the following operands:

Int_value A user defined 32-bit value available in the DMA SCRIPTS Pointer Save (DSPS) register at the time of the interrupt.

WHEN Forces the SCRIPTS engine to wait for a valid SCSI bus phase before continuing. A valid phase is indicated by assertion of the SREQ/ signal.

IF Causes the SCRIPTS processor to immediately check for a valid SCSI bus phase. IF should not be used when comparing for a phase as this could yield unpredictable results. The only exception is using a WHEN conditional just prior to the IF conditional for any given sequence of phase checks.

NOT Negates the comparison. Clears the True bit if present, otherwise the True bit is set.

Phase Specifies the Message, Command/Data, and Input/Output bit values that identify the SCSI phase in the instruction. The desired phase value is compared with the actual values of the SCSI phase lines before the SCRIPTS processor performs the instruction. This field is only valid for the initiator mode and should not be used in the target mode.

ATN Indicates that an interrupt should take place based on an initiator SATN/ signal. This field is valid only for the target mode and should not be used in the initiator mode.

data Represents an 8-bit value that is stored in the data field of the instruction. In addition the Compare Data bit is set.

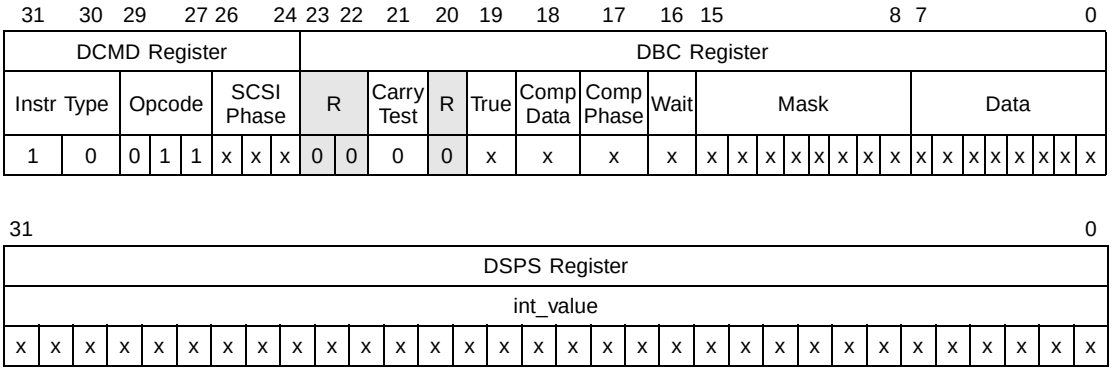
MASK Represents an 8-bit value that is stored in the mask field of the instruction. Any bit that is set in the mask causes the corresponding bit in the data byte to be ignored at the time of the comparison.

CARRY Indicates that an interrupt should take place based on the value of the carry bit in the ALU. Carry comparisons cannot take place at the same time as data and phase comparisons.

Example

```
INT 0x00000001, WHEN NOT COMMAND
INT 0x200010F7, IF 0xF8 AND MASK 0x07
```

Figure 3.6 INT Format



The values in Table 3.6 define the SCSI information transfer phase. The LSI53C10XX chips, with dual transition timing capabilities define two transfer phases, ST for single transition timing, and DT for dual transition timing.

Table 3.6 SCSI Phase Bit Values (INT Format) [1]

Phase	Message	Command/Data	Input/Output
DATA_OUT[2] (ST_DATA_OUT)[3]	0	0	0
DATA_IN[2] (ST_DATA_IN)[3]	0	0	1

Table 3.6 SCSI Phase Bit Values (INT Format) (Cont.) [1]

Phase	Message	Command/Data	Input/Output
COMMAND	0	1	0
STATUS	0	1	1
RES4[4] (DT_DATA_OUT)[3]	1	0	0
RES5[4] (DT_DATA_IN)[3]	1	0	1
MESSAGE_OUT	1	1	0
MESSAGE_IN	1	1	1

1. 0 - False, negated; 1 - True, asserted. For these phases, SEL is negated and BSY is asserted.
2. All chips except LSI53C10XX.
3. LSI53C10XX chips.
4. RES4 and RES5 are reserved SCSI phases except in the LSI53C10XX chips.

Field(s)

This command has the following fields:

Instruction Type Transfer Control.

Opcode Interrupt Instruction.

SCSI Phase These bits reflect the actual values of the SCSI phase lines.

Register Definition(s)

The information listed below describes the DBC and DSPS registers:

Carry Test When this bit is set, true/false comparisons are based on the ALU Carry bit. Carry comparisons cannot be made at the same time as data and phase comparisons.

True Transfer on TRUE/FALSE condition.
0 - Transfer if condition is FALSE
1 - Transfer if condition is TRUE

Compare Data Compare data byte to first byte of the received data.
0 - Do not compare data
1 - Perform comparison

Compare Phase Compare current SCSI phase to SCSI phase field or SATN/. This bit is set whenever the Phase operand is used.
0 - Do not compare phase
1 - Perform comparison

Wait Wait for valid phase. Set by the WHEN operand, cleared by the IF operand.
0 - Perform comparison immediately
1 - Wait for valid phase (SREQ/ asserted by target)

Mask An 8-bit field that masks the value in SFBR before the comparison with the data field in the instruction takes place. As a result of this operation, any bits that are set will cause the corresponding bit in the data byte to be ignored. If this field is not specified, a mask of 0x00 is used.

Data An 8-bit field that is compared with the incoming data after the mask operation of the mask byte takes place. Comparison indicates either an equal or not equal condition. If the Data field is not specified, the compare data bit is cleared and 0x00 is coded for both the mask and data bytes.

Int_Value A 32-bit user defined value that is available to the external processor to identify the cause of the interrupt. If the interrupt conditions are met, the int_value will be available in the DSPS register for the processor to use to determine the cause of the interrupt.

Description

The SCSI Interrupt instruction causes the chip to conditionally halt execution and post an interrupt request to the external processor. It is used if the SCSI phase, data, or attention condition compares true with the phase, data, or attention condition described in the instruction. The NOT qualifier determines a boolean true/false outcome for the comparison. If the comparison is false, the SCRIPTS processor does not post the interrupt but fetches the next instruction in line and continues execution.

When the optional data field is used, it is compared to the first byte of the SFBR. This contains the most recent byte of any kind of data that has been moved into the SFBR register. The user's SCSI SCRIPTS program determines which routine to execute next based on actual data values received. Using a series of these compares, the algorithm processes complex sequences without external processor intervention.

When the optional MASK keyword and its associated value are specified the SCRIPTS processor selectively compares bits within the data byte. Figure 3.2 illustrates this comparison. During the comparison, any bits set in the mask byte cause the corresponding bit in the data byte to be ignored for the comparison.

Legal Forms

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

3.2.6 INTFLY

```
INTFLY [int_value] [, {IF | WHEN}[NOT][ATN | Phase] [AND |
OR] [data[AND MASK data]]]
INTFLY [int_value] [, {IF | WHEN}[NOT] CARRY]
```

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition Generate Interrupts and continue SCRIPTS execution.

Operands This command has the following operands:

int_value A user defined 32-bit value that is written to the DSPS register at the time of the interrupt. However, since the processor continues to execute, the value is immediately overwritten with the next instruction fetch. Refer to the Note at the end of this section for more information.

WHEN Forces the SCRIPTS engine to wait for a valid SCSI bus phase before continuing. A valid phase is indicated by assertion of the SREQ/ signal.

IF Causes the SCRIPTS processor to immediately check for a valid SCSI bus phase. IF should not be used when comparing for a phase as this could yield unpredictable results. The only exception is using a WHEN conditional just prior to the IF conditional for any given sequence of phase checks.

NOT Negates the comparison. It clears the True bit if present, otherwise the True bit is set.

Phase Specifies the Message, Command/Data, and Input/Output bit values that identify the SCSI phase in the instruction. The desired phase value is compared with the actual values of the SCSI phase lines before the SCRIPTS processor performs the instruction. This field is only valid for the initiator mode and should not be used in the target mode.

ATN Indicates that an interrupt should take place based on the state of the initiator SATN/ signal. This field is valid only for the target mode and should not be used in the initiator mode.

data Represents an 8-bit value that is stored in the data field of the instruction. In addition the Compare Data bit is set.

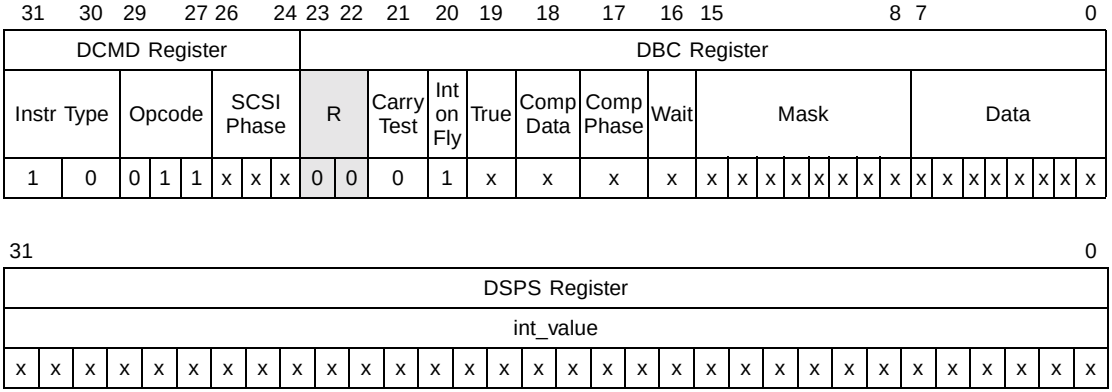
MASK Represents an 8-bit value that is stored in the mask field of the instruction. Any bit that is set in the mask causes the corresponding bit in the data byte to be ignored at the time of the comparison.

CARRY Indicates that a jump should take place based on the value of the carry bit in the ALU. Carry comparisons cannot be made in the same instruction as data or phase comparisons.

Example

```
INTFLY 0x00000001, WHEN NOT COMMAND
INTFLY 0x200010F7, IF 0xF8 AND MASK 0x07
```

Figure 3.7 INTFLY Format



Field(s) This command has the following fields:

Instruction Type Transfer Control.

Opcode Interrupt on the Fly instruction.

SCSI Phase These bits reflect the actual values of the SCSI phase lines.

The values in Table 3.7 define the SCSI information transfer phase. The LSI53C10XX chips, with dual transition timing capabilities define two transfer phases, ST for single transition timing, and DT for dual transition timing.

Table 3.7 SCSI Phase Bit Values (INTFLY Format)[1]

Phase	Message	Command/Data	Input/Output
DATA_OUT[2] (ST_DATA_OUT)[3]	0	0	0
DATA_IN[2] (ST_DATA_IN)[3]	0	0	1

Table 3.7 SCSI Phase Bit Values (INTFLY Format)[1] (Cont.)

Phase	Message	Command/Data	Input/Output
COMMAND	0	1	0
STATUS	0	1	1
RES4[4] (DT_DATA_OUT)[3]	1	0	0
RES5[4] (DT_DATA_IN)[3]	1	0	1
MESSAGE_OUT	1	1	0
MESSAGE_IN	1	1	1

1. 0 - False, negated; 1 - True, asserted. For these phases, SEL is negated and BSY is asserted.
2. All chips except LSI53C10XX.
3. LSI53C10XX chips.
4. RES4 and RES5 are reserved SCSI phases except in the LSI53C10XX chips.

Register Definition(s)

The information listed below describes the DBC and DSPS registers.

Carry Test When this bit is set, true/false comparisons may be made based on the ALU Carry bit. Carry comparisons cannot be made in the same instruction as data or phase comparisons.

Int on Fly When this bit is set, the Interrupt instruction will not halt the SCRIPTS processor.

True Transfer on TRUE/FALSE condition.
0 - Transfer if condition is FALSE
1 - Transfer if condition is TRUE

Compare Data Compare data byte to first byte of the received data.
0 - Do not compare data
1 - Perform comparison

Compare Phase Compare current SCSI phase to SCSI phase field or SATN. This bit is set whenever the Phase operand is used.
0 - Do not compare phase
1 - Perform comparison

Wait Wait for valid phase. This bit is set by the WHEN operand in the instruction, and cleared by the IF operand.
0 - Perform comparison immediately
1 - Wait for valid phase (SREQ/ asserted by target)

Mask An 8-bit field that is used to mask the value in SFBR before the comparison with the data field in the instruction takes place. As a result of this operation, any bits that are set will cause the corresponding bit in the data byte to be ignored. If this field is not specified, a mask of 0x00 is used.

Data An 8-bit field that is compared with the incoming data after the mask operation with the mask byte takes place. Comparison indicates either an equal or not equal condition. If the Data field is not specified, the compare data bit is cleared and 0x00 is coded for both the mask and data bytes.

Int_Value A 32-bit user defined value that identifies the cause of the interrupt. Even though the int_value is stored, since the processor continues to execute, it is immediately overwritten with the next instruction fetch. Refer to the Notes at the end of this section for more information.

Description

The SCSI Interrupt on-the-Fly instruction causes the chip to conditionally set the INTFLY bit in the Interrupt Status (ISTAT) register and post an interrupt request to the external processor. It is invoked if the SCSI phase, data, or attention condition compares true with the phase, data, or attention condition described in the instruction.

The NOT qualifier is used to indicate a boolean true/false desired outcome of the comparison. If the comparison is false, the SCRIPTS processor will not post the interrupt but will instead fetch the next instruction and continue SCRIPTS execution.

When the optional data field is used, it is compared to the first byte of the SFBR. This contains the most recent byte of any kind of data that has been moved into the SFBR register. The user's SCSI SCRIPTS program can determine which 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.

When the optional MASK keyword and its associated value are specified the SCRIPTS processor allows selective comparisons of bits within the data byte. This comparison is illustrated in Figure 3.2. During the comparison, any bits that are set in the mask field will cause the corresponding bit in the data byte to be ignored for the comparison.

Notes

Unlike the INT instruction, INTFLY does not allow a driver program to make an inquiry to the chip for the int_value. Even though the int_value is stored, since the processor continues to execute, it is immediately overwritten with the next instruction fetch. Users who want an accessible interrupt value can use the move memory instruction to store a user defined value to a known memory location before executing the INTFLY instruction.

Legal Forms

```
INTFLY
INTFLY, IF ATN
INTFLY, IF Phase
INTFLY, IF CARRY
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 CARRY
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 CARRY
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 CARRY
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 int_value
INTFLY int_value, IF ATN
INTFLY int_value, IF Phase
INTFLY int_value, IF CARRY
INTFLY int_value, IF data
INTFLY int_value, IF data AND MASK data
INTFLY int_value, IF ATN AND data
INTFLY int_value, IF ATN AND data AND MASK data
```

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

3.2.7 JUMP

```
JUMP {REL(Address) | Address} [,{IF | WHEN}[NOT][ATN |
Phase] AND | OR] [data[AND MASK data]]]
JUMP {[REL] (Address) | Address} [, {IF | WHEN}[NOT] CARRY]
```

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition SCSI Transfer Control - Jump.

Operands This command has the following operands:

REL Indicates the use of relative addressing.

Address Is the location to which execution will be transferred if the subroutine is called. If REL is used, Address is the offset from the current DSP value.

WHEN Forces the SCRIPTS engine to wait for a valid SCSI bus phase before continuing. A valid phase is indicated by assertion of the SREQ/ signal.

IF Causes the SCRIPTS processor to immediately check for a valid SCSI bus phase. IF should not be used when comparing for a phase as this could yield unpredictable results. The only exception is using a WHEN conditional just prior to the IF conditional for any given sequence of phase checks.

NOT Negates the comparison. It clears the True bit if present, otherwise the True bit is set.

Phase Specifies the Message, Command/Data, and Input/Output bit values that identify the SCSI phase in the instruction. The desired phase value is compared with the actual values of the SCSI phase lines before the SCRIPTS processor performs the instruction. This field is only valid for the initiator mode and should not be used in the target mode.

ATN Is used to indicate that a jump should take place based on the state of the initiator SATN/ signal. This field is valid only for target mode and should not be used in the initiator mode.

data Represents an 8-bit value that is stored in the data field of the instruction. In addition, this keyword indicates that the Compare Data bit is set.

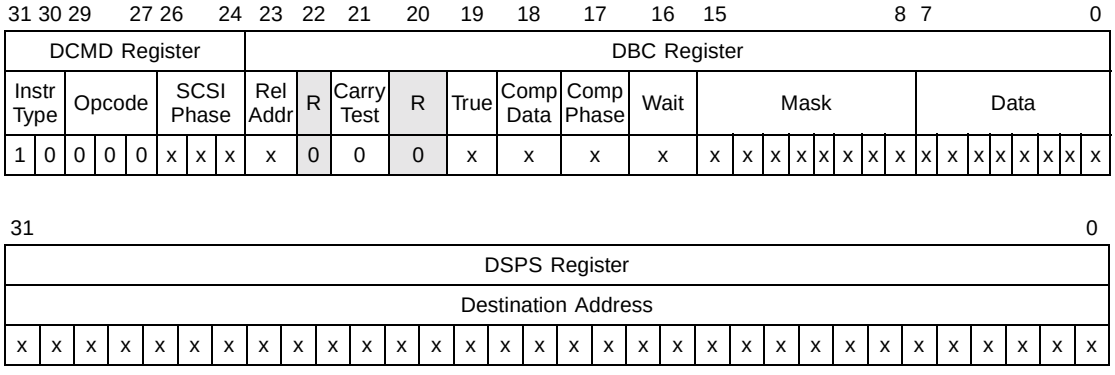
MASK Represents an 8-bit value that is stored in the mask field of the instruction. Any bit that is set in the mask causes the corresponding bit in the data byte to be ignored at the time of the comparison.

CARRY Indicates that a jump should take place based on the value of the carry bit in the ALU.

Example

```
JUMP Do_Next_Command WHEN COMMAND
JUMP Data_Check, IF DATA_IN AND 0x80 MASK 0x7F
```

Figure 3.8 JUMP Format



Field(s)

This command has the following fields:

Instruction Type Transfer Control.

Opcode Jump instruction.

SCSI Phase These bits reflect the actual values of the SCSI phase lines.

The values in Table 3.8 define the SCSI information transfer phase. The LSI53C10XX chips, with dual transition timing capabilities define two transfer phases, ST for single transition timing, and DT for dual transition timing.

Table 3.8 SCSI Phase Bit Values (JUMP Format)[1]

Phase	Message	Command/Data	Input/Output
DATA_OUT[2] (ST_DATA_OUT)[3]	0	0	0
DATA_IN[2] (ST_DATA_IN)[3]	0	0	1
COMMAND	0	1	0
STATUS	0	1	1
RES4[4] (DT_DATA_OUT)[3]	1	0	0
RES5[4] (DT_DATA_IN)[3]	1	0	1
MESSAGE_OUT	1	1	0
MESSAGE_IN	1	1	1

1. 0 - False, negated; 1 - True, asserted. For these phases, SEL is negated and BSY is asserted.
2. All chips except LSI53C10XX.
3. LSI53C10XX chips.
4. RES4 and RES5 are reserved SCSI phases except in the LSI53C10XX chips.

Register Definition(s)

The information listed below describes the DBC and DSPS registers:

Relative Address The Relative Addressing Mode indicates that the 24-bit address value in the instruction is to be used as an offset from the current DSP address (which is pointing to the next instruction, not the one currently executing).

Carry Test	When this bit is set, true/false comparisons are based on the ALU Carry bit. Comparisons to the state of the Carry flag may not be made in conjunction with other comparisons.
True	Transfer on TRUE/FALSE condition. 0 - Transfer if condition is FALSE 1 - Transfer if condition is TRUE
Compare Data	Compare data byte to first byte of the received data. 0 - Do not compare data 1 - Perform comparison
Compare Phase	Compare current SCSI phase to SCSI phase field or SATN/. This bit is set whenever the Phase operand is used. 0 - Do not compare phase 1 - Perform comparison
Wait	Wait for valid phase. This bit is set by the WHEN operand in the instruction, and cleared by the IF operand. 0 - Perform comparison immediately 1 - Wait for valid phase (SREQ/ asserted by target)
Mask	An 8-bit field that is used to mask the value in SFBR before the comparison with the data field in the instruction takes place. As a result of this operation, any bits that are set will cause the corresponding bit in the data byte to be ignored. If this field is not specified, a mask of 0x00 is used.
Data	An 8-bit field that is compared with the incoming data after the mask operation of the mask byte takes place. Comparison indicates either an equal or not equal condition. If the Data field is not specified, the Compare Data bit is cleared and 0x00 is coded for both the mask and data bytes.
Destination Address	A 32-bit address (or 24-bit offset) where execution will continue if the jump is executed.

Description

The SCSI Jump instruction is a conditional jump to the destination address, if the SCSI phase, data, or attention condition compares true with the phase, data, or attention condition described in the instruction. If the comparison is false, the SCRIPTS processor does not branch to the destination but instead fetches the next instruction and continues execution.

When the optional data field is used, it is compared to the SFBR. This contains the most recent byte of any type of data that has been moved into the SFBR register. The SCSI SCRIPTS program determines which routine to execute next based on received data values. Using a series of

these compares, the algorithm processes complex sequences with no intervention required by the external processor.

When the optional MASK keyword and its associated value are specified, the SCRIPTS processor allows selective comparisons of bits within the data byte. During the compare, any mask bits that are set will cause the corresponding bit in the data byte to be ignored for the comparison.

Notes

Jump instructions are used to control the flow of the SCRIPTS routines. They are used to avoid phase mismatch interrupts in situations where multiple phase sequences are possible.

The REL keyword, which indicates relative addressing, is unrelated to the declarative keyword RELATIVE that establishes relative buffers.

Legal Forms

```
JUMP address
JUMP address, IF ATN
JUMP address, IF Phase
JUMP address, IF CARRY
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 CARRY
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 CARRY
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 CARRY
JUMP address, WHEN NOT data
JUMP address, WHEN NOT data AND MASK data
JUMP address, WHEN NOT Phase OR data
JUMP address, WHEN NOT Phase OR data AND MASK data
JUMP REL(address)
JUMP REL(address), IF ATN
JUMP REL(address), IF Phase
```

```
JUMP REL(address), IF CARRY
JUMP REL(address), IF data
JUMP REL(address), IF data AND MASK data
JUMP REL(address), IF ATN AND data
JUMP REL(address), IF ATN AND data AND MASK data
JUMP REL(address), IF Phase AND data
JUMP REL(address), IF Phase AND data AND MASK data
JUMP REL(address), WHEN Phase
JUMP REL(address), WHEN CARRY
JUMP REL(address), WHEN data
JUMP REL(address), WHEN data AND MASK data
JUMP REL(address), WHEN Phase AND data
JUMP REL(address), WHEN Phase AND data AND MASK data
JUMP REL(address), IF NOT ATN
JUMP REL(address), IF NOT Phase
JUMP REL(address), IF NOT CARRY
JUMP REL(address), IF NOT data
JUMP REL(address), IF NOT data AND MASK data
JUMP REL(address), IF NOT ATN OR data
JUMP REL(address), IF NOT ATN OR data AND MASK data
JUMP REL(address), IF NOT Phase OR data
JUMP REL(address), IF NOT Phase OR data AND MASK data
JUMP REL(address), WHEN NOT Phase
JUMP REL(address), WHEN NOT CARRY
JUMP REL(address), WHEN NOT data
JUMP REL(address), WHEN NOT data AND MASK data
JUMP REL(address), WHEN NOT Phase OR data
JUMP REL(address), WHEN NOT Phase OR data AND MASK data
```

3.2.8 JUMP 64

This command is only available on LSI53C896 and newer chips.

```
JUMP64 {Address} [,{IF | WHEN}[NOT][ATN | Phase] AND | OR]
[data[AND MASK data]]]
JUMP64 {Address} [, {IF | WHEN}[NOT] CARRY]
```

Supported by LSI53C896 and later chips.

Definition SCSI Transfer Control - Jump.

Operands This command has the following operands:

Address Is the location to which execution will be transferred if the subroutine is called. If REL is used, Address is the offset from the current DSP value.

WHEN Forces the SCRIPTS engine to wait for a valid SCSI bus phase before continuing. A valid phase is indicated by assertion of the SREQ/ signal.

IF Causes the SCRIPTS processor to immediately check for a valid SCSI bus phase. IF should not be used when comparing for a phase as this could yield unpredictable results. The only exception is using a WHEN conditional just prior to the IF conditional for any given sequence of phase checks.

NOT Negates the comparison. It clears the True bit if present, otherwise the True bit is set.

Phase Specifies the Message, Command/Data, and Input/Output bit values that identify the SCSI phase in the instruction. The desired phase value is compared with the actual values of the SCSI phase lines before the SCRIPTS processor performs the instruction. This field is only valid for the initiator mode and should not be used in the target mode.

ATN Is used to indicate that a jump should take place based on the state of the initiator SATN/ signal. This field is valid only for target mode and should not be used in the initiator mode.

Data Represents an 8-bit value that is stored in the data field of the instruction. In addition, this keyword indicates that the Compare Data bit is set.

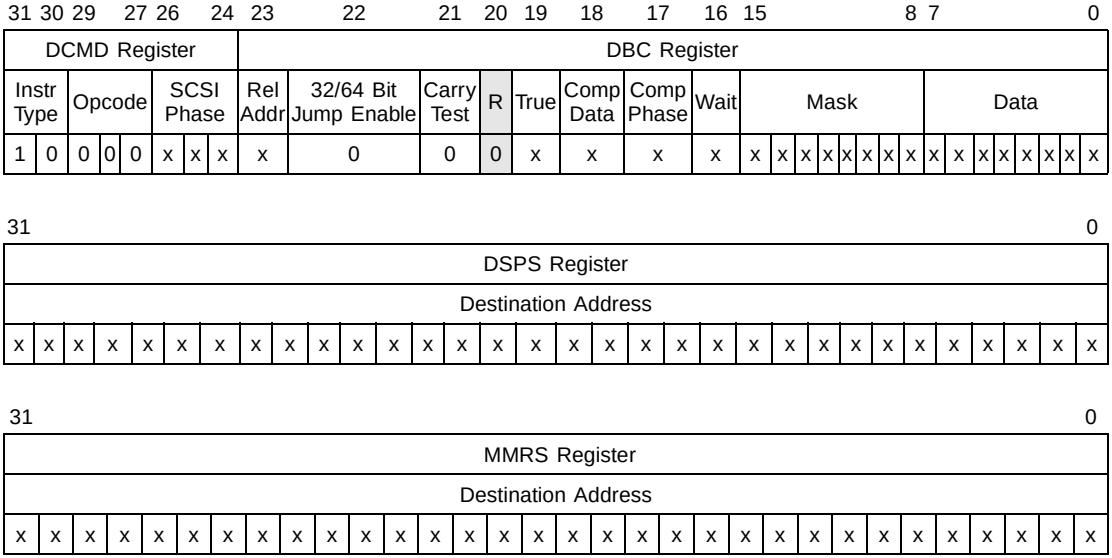
MASK Represents an 8-bit value that is stored in the mask field of the instruction. Any bit that is set in the mask causes the corresponding bit in the data byte to be ignored at the time of the comparison.

CARRY Indicates that a jump should take place based on the value of the carry bit in the ALU.

Example

```
JUMP Do_Next_Command WHEN COMMAND
JUMP Data_Check, IF DATA_IN AND 0x80 MASK 0x7F
```

Figure 3.9 JUMP 64 Format



Field(s) This command has the following fields:

Instruction Type Transfer Control.

Opcode Jump instruction.

SCSI Phase These bits reflect the actual values of the SCSI phase lines.

The values in Table 3.9 define the SCSI information transfer phase. The LSI53C10XX chips, with dual transition timing capabilities define two transfer phases, ST for single transition timing, and DT for dual transition timing.

Table 3.9 SCSI Phase Bit Values (JUMP 64 Format)[1]

Phase	Message	Command/Data	Input/Output
DATA_OUT[2] (ST_DATA_OUT)[3]	0	0	0
DATA_IN[2] (ST_DATA_IN)[3]	0	0	1
COMMAND	0	1	0
STATUS	0	1	1
RES4[4] (DT_DATA_OUT)[3]	1	0	0
RES5[4] (DT_DATA_IN)[3]	1	0	1
MESSAGE_OUT	1	1	0
MESSAGE_IN	1	1	1

1. 0 - False, negated; 1 - True, asserted. For these phases, SEL is negated and BSY is asserted.
2. All chips except LSI53C10XX.
3. LSI53C10XX chips.
4. RES4 and RES5 are reserved SCSI phases except in the LSI53C10XX chips.

Register Definition(s)

The information listed below describes the DBC, DSPS, and MMRS registers:

Relative Address The Relative Addressing Mode indicates that the 24-bit address value in the instruction is to be used as an offset from the current DSP address (which is pointing to the next instruction, not the one currently executing).

Carry Test When this bit is set, true/false comparisons are based on the ALU Carry bit. Comparisons to the state of the Carry flag may not be made in conjunction with other comparisons.

True Transfer on TRUE/FALSE condition.
0 - Transfer if condition is FALSE
1 - Transfer if condition is TRUE

Compare Data Compare data byte to first byte of the received data.
0 - Do not compare data
1 - Perform comparison

Compare Phase	Compare current SCSI phase to SCSI phase field or SATN/. This bit is set whenever the Phase operand is used. 0 - Do not compare phase 1 - Perform comparison
Wait	Wait for valid phase. This bit is set by the WHEN operand in the instruction, and cleared by the IF operand. 0 - Perform comparison immediately 1 - Wait for valid phase (SREQ/ asserted by target)
Mask	An 8-bit field that is used to mask the value in SFBR before the comparison with the data field in the instruction takes place. As a result of this operation, any bits that are set will cause the corresponding bit in the data byte to be ignored. If this field is not specified, a mask of 0x00 is used.
Data	An 8-bit field that is compared with the incoming data after the mask operation of the mask byte takes place. Comparison indicates either an equal or not equal condition. If the Data field is not specified, the Compare Data bit is cleared and 0x00 is coded for both the mask and data bytes.
Destination Address	A 32-bit address (or 24-bit offset) where execution will continue if the jump is executed.

Description

The SCSI Jump instruction is a conditional jump to the destination address, if the SCSI phase, data, or attention condition compares true with the phase, data, or attention condition described in the instruction. If the comparison is false, the SCRIPTS processor does not branch to the destination but instead fetches the next instruction and continues execution.

When the optional data field is used, it is compared to the SFBR. This contains the most recent byte of any type of data that has been moved into the SFBR register. The SCSI SCRIPTS program determines which routine to execute next based on received data values. Using a series of these compares, the algorithm processes complex sequences with no intervention required by the external processor.

When the optional MASK keyword and its associated value are specified, the SCRIPTS processor allows selective comparisons of bits within the data byte. During the compare, any mask bits that are set will cause the corresponding bit in the data byte to be ignored for the comparison.

Notes Jump instructions are used to control the flow of the SCRIPTS routines. They are used to avoid phase mismatch interrupts in situations where multiple phase sequences are possible.

The REL keyword, which indicates relative addressing, is unrelated to the declarative keyword RELATIVE that establishes relative buffers.

Legal Forms

```
JUMP64 address
JUMP64 address, IF ATN
JUMP64 address, IF Phase
JUMP64 address, IF CARRY
JUMP64 address, IF data
JUMP64 address, IF data AND MASK data
JUMP64 address, IF ATN AND data
JUMP64 address, IF ATN AND data AND MASK data
JUMP64 address, IF Phase AND data
JUMP64 address, IF Phase AND data AND MASK data
JUMP64 address, WHEN Phase
JUMP64 address, WHEN CARRY
JUMP64 address, WHEN data
JUMP64 address, WHEN data AND MASK data
JUMP64 address, WHEN Phase AND data
JUMP64 address, WHEN Phase AND data AND MASK data
JUMP64 address, IF NOT ATN
JUMP64 address, IF NOT Phase
JUMP64 address, IF NOT CARRY
JUMP64 address, IF NOT data
JUMP64 address, IF NOT data AND MASK data
JUMP64 address, IF NOT ATN OR data
JUMP64 address, IF NOT ATN OR data AND MASK data
JUMP64 address, IF NOT Phase OR data
JUMP64 address, IF NOT Phase OR data AND MASK data
JUMP64 address, WHEN NOT Phase
JUMP64 address, WHEN NOT CARRY
JUMP64 address, WHEN NOT data
JUMP64 address, WHEN NOT data AND MASK data
JUMP64 address, WHEN NOT Phase OR data
JUMP64 address, WHEN NOT Phase OR data AND MASK data
```

3.2.9 LOAD

```
LOAD register, byte_count, [DSAREL(]source_address[)]
```

Supported by LSI53C810A, LSI53C860, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, LSI53C895, LSI53C895A, LSI53C896, LSI53C1000, LSI53C1010, LSI53C1010R, LSI53C1000R.

Definition Load data from memory to an internal register of the chip.

Operands This command has the following operands:

Register Is one of the register names in the chip operating register set.

Byte Count Is the number of bytes [1:4] to be transferred from the source_address.

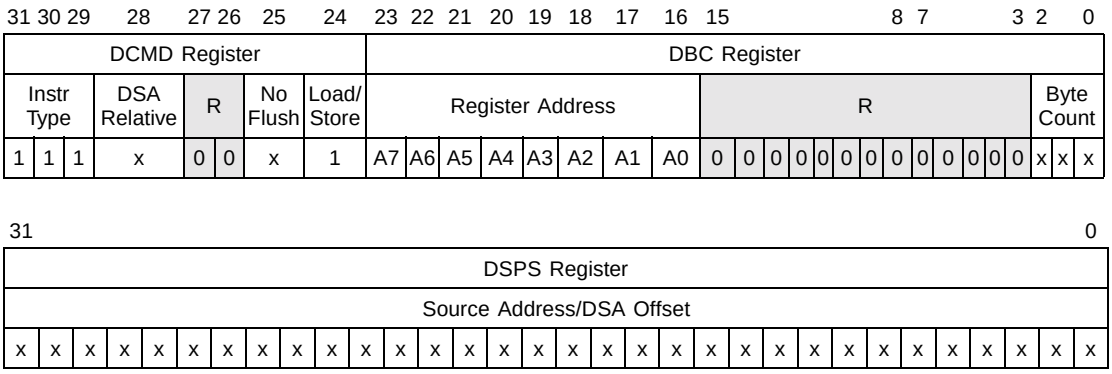
DSA Relative Indicates that the source_address is an offset and should be added to the DSA register to obtain the physical address (DSA relative).

Source Address Is the physical address or offset from the DSA to obtain the physical address of the data to be loaded into the register.

Example

```
LOAD SCRATCHA0, 4, data_buf
LOAD SCRATCHA3, 2, DSAREL (0x02)
```

Figure 3.10 LOAD Format



Field(s) This command has the following fields:

Instruction Type Load/Store.

DSA Relative Indicates source address location.
0 - DSPS contains actual address of data to load
1 - DSPS contains a 24-bit offset value that is added to the DSA to determine the source address.

No Flush Indicates a store instruction without flushing the prefetch unit. The Pre-fetch Enable bit in the DMA Control (DCNTL) register must be set.

Load/Store This field defines whether the instruction will be executed as a Load or a Store.
0 - Store instruction
1 - Load instruction

Register Address These bits select the register to load within the chip operating register set.

Byte Count Indicates the number of bytes to transfer. Valid values are 1, 2, 3, or 4.

Source Address Actual address (or offset from the DSA) of the data to load into the chip register.

Description

The Load instruction is more efficient than a Move Memory instruction when moving data from a memory location to an internal register of the chip. It is a two Dword instruction, compared to three Dwords for a Memory Move. This instruction may be used to move up to 4 bytes. The number of bytes being loaded is indicated by the low order bits in the first Dword of the instruction. The maximum number of bytes is defined by the Register Address field, as illustrated in Table 3.10.

Table 3.10 Register Address Field Definitions (LOAD Format)

DBC Bits [17:16] (Register Address bits A1:A0)	Number of Bytes to Load
00	1, 2, 3, or 4
01	1, 2, or 3
10	1 or 2
11	1

Notes

The register address and memory address must have the same byte alignment, and the byte count set so that it does not cross Dword boundaries. The memory address may not map back to the SCRIPTS processor operating registers, although it may map back to a location in the SCRIPTS RAM. If these conditions are violated, a PCI illegal read/write cycle will occur and the chip will issue an Interrupt (Illegal Instruction Detected) immediately following, because the intended operation did not happen.

Loads from SCRIPTS RAM cross the PCI bus, except for the LSI53C896/10XX chips. However, it is selectable for debug.

Legal Forms

```
LOAD register, byte_count, source_address
LOAD register, byte_count, DSAREL(source_address)
```

3.2.10 LOAD64

LOAD64 uses table indirect addressing only.

```
LOAD64 register, byte_count, [DSAREL(]source_address[)]
```

Supported by LSI53C896, LSI53C1000, LSI53C1010, LSI53C1010R, LSI53C1000R.

Definition Load 64-bit data from memory to an internal register of the chip.

Operands This command has the following operands:

Register Is one of the register names in the chip operating register set.

Byte Count Is the number of bytes [1:4] to be transferred from the source_address.

DSA Relative Indicates that the source_address is an offset and should be added to the DSA register to obtain the physical address (DSA relative).

Source Address Is the physical address or offset from the DSA to obtain the physical address of the data to be loaded into the register.

Example

```
LOAD64 SCRATCHA0, 4, data_buf
LOAD64 SCRATCHA3, 2, DSAREL (0x02)
```

Table 3.11 LOAD64 Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DCMD Register								DBC Register																							
Instr Type			DSA Relative	R		No Flush	Load/ Store	Register Address								R														Byte Count		
1	1	1	x	0	0	x	1	A7	A6	A5	A4	A3	A2	A1	A0	0	0	0	0	0	0	0	0	0	0	0	0	x	x	x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DSPS Register																															
Source Address/DSA Offset																															
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMRS Register																															
Destination Address																															
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Field(s) This command has the following fields:

Instruction Type Load/Store.

DSA Relative Indicates source address location.
0 - DSPS contains actual address of data to load.
1 - DSPS contains a 24-bit offset value that is added to the DSA to determine the source address.

No Flush Indicates a store instruction without flushing the prefetch unit. The Pre-fetch Enable bit in the DMA Control (DCNTL) register must be set.

Load/Store This field defines whether the instruction will be executed as a Load or a Store.
0 - Store instruction
1 - Load instruction

Register Address These bits select the register to load within the chip operating register set.

Byte Count Indicates the number of bytes to transfer. Valid values are 1, 2, 3, or 4.

Source Address Actual Address (or offset from the DSA) of the data to load into the chip register.

Description The LOAD64 instruction is more efficient than a Move Memory instruction when moving data from a memory location to an internal register of the chip. It is a two Dword instruction, compared to three Dwords for a Memory Move. This instruction may be used to move up to 4 bytes. The number of bytes being loaded is indicated by the low order bits in the first Dword of the instruction. The maximum number of bytes is defined by the Register Address field, as illustrated in Table 3.12.

Table 3.12 Register Address Field Definitions (LOAD64 Format)

DBC Bits [17:16] (Register Address bits [A1:A0])	Number of Bytes to Load
00	1, 2, 3, or 4
01	1, 2, or 3
10	1 or 2
11	1

Notes The register address and memory address must have the same byte alignment, and the byte count set so that it does not cross Dword boundaries. The memory address may not map back to the SCRIPTS processor operating registers, although it may map back to a location in the SCRIPTS RAM. If these conditions are violated, a PCI illegal read/write cycle will occur and the chip will issue an Interrupt (Illegal Instruction Detected) immediately following, because the intended operation did not happen.

Legal Forms

```
LOAD64 register, byte_count, source_address
LOAD64 register, byte_count, DSAREL(source_address)
```

3.2.11 MOVE

```
MOVE {FROM | count,} [PTR] address, {WITH | WHEN}phase
```

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition SCSI Block Move.

Operands This command has the following operands:

FROM Indicates the table indirect addressing mode.
Note: FROM and PTR must not be used in the same instruction.

count A 24-bit number indicating the number of bytes being transferred.

PTR Sets the indirect bit if present, it is cleared otherwise.
Note: Do not use PTR and FROM in the same instruction

address A 32-bit starting address of the data in memory.

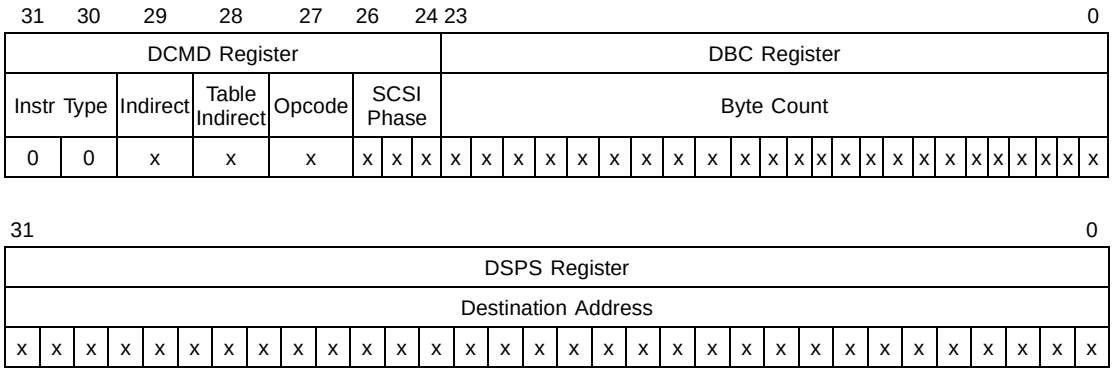
WITH/WHEN Sets the mode for the device; WITH for target mode and WHEN for initiator mode.

Phase Specifies the Message, Command/Data, and Input/Output bit values that identify the SCSI phase in the instruction. The desired phase value is compared with the actual values of the SCSI phase lines before the SCRIPTS processor performs the instruction. This field is only valid for the initiator mode and should not be used in the target mode.

Example

```
MOVE FROM dev_1, WITH MSG_IN
MOVE 6, cmd_buf, WHEN CMD
```

Figure 3.11 MOVE Format



Field(s) This command has the following fields:

Instruction Type Block Move.

Indirect Indirect Addressing Mode.
0 - Use destination field as an address
1 - Use destination field as a pointer to an address

Table Indirect Table Indirect Addressing Mode.
0 - Use Absolute addressing mode
1 - Use destination address as offset from the value of DSA register

Opcode This field defines whether the instruction executes as a Block Move or a Chained Block Move.

SCSI Phase These bits reflect the actual values of the SCSI phase lines.

	Target	Initiator
MOVE	Opcode = 0	Opcode = 1
CHMOV	Opcode = 1	Opcode = 0

The values in Table 3.13 define the SCSI information transfer phase. The LSI53C10XX chips, with dual transition timing capabilities define two transfer phases, ST for single transition timing, and DT for dual transition timing.

Table 3.13 SCSI Phase Bit Values (MOVE Format)[1]

Phase	Message	Command/Data	Input/Output
DATA_OUT[2] (ST_DATA_OUT)[3]	0	0	0
DATA_IN[2] (ST_DATA_IN)[3]	0	0	1
COMMAND	0	1	0
STATUS	0	1	1
RES4[4] (DT_DATA_OUT)[3]	1	0	0
RES5[4] (DT_DATA_IN)[3]	1	0	1

Table 3.13 SCSI Phase Bit Values (MOVE Format)[1]

Phase	Message	Command/Data	Input/Output
MESSAGE_OUT	1	1	0
MESSAGE_IN	1	1	1

1. 0 - False, negated; 1 - True, asserted. For these phases, SEL is negated and BSY is asserted.
2. All chips except LSI53C10XX.
3. LSI53C10XX chips.
4. RES4 and RES5 are reserved SCSI phases except in the LSI53C10XX chips.

Register Definition(s)

The information listed below describes the DBC and DSPS registers.

Byte Count A 24-bit number indicating the number of bytes to transfer.

Dest Addr Destination address for the transfer.

Description

There are various forms of the Block Move instruction. The “address” and “count” terms specify the address and byte count fields of the instruction. If the optional keyword “PTR” is present the Indirect bit is set. If the optional keyword FROM is present the Table Indirect bit is set (for more information on Table Indirect addressing, refer to Chapter 9). PTR and FROM may not be used in the same instruction. “Phase” specifies the phase field of the instruction. WITH or WHEN are used to specify the Block Move function codes. WITH is used to signal the target role which sets the phase values, and WHEN is the initiator “test for phase” feature.

The SCRIPTS processor waits for a valid phase (initiator) or drives the phase lines (target). In the initiator role, it performs a comparison looking for a match between the phase specified in the SCRIPT and the actual value on the bus. If the phases do not match, a phase mismatch interrupt occurs. If the phases match, data is transferred in or out according to the phase lines. After the last byte is transferred to its final destination, the SCRIPTS processor fetches the next SCRIPTS instruction. If the target changes phase in the middle of a block move, a phase mismatch interrupt will occur.

Notes

In the target mode, a MOVE instruction with a byte count of zero can be used during a Command phase. The SCRIPTS processor will determine the number of bytes to move from the command group code in the first byte of the command.

If the command code is vendor unique, the SCRIPTS processor uses the byte count from the instruction. If this byte count is zero, the chip issues an illegal instruction interrupt.

For LSI53C825A, LSI53C875, LSI53C876, LSI53C885, LSI53C895, LSI53C895A, LSI53C896, LSI53C1000, LSI53C1010, LSI53C1010R, LSI53C1000R only: If the SCSI group code is either Group 0, 1, 2, or 5 and if the Vendor Unique Enhancement bit 1 (VUE1) bit (SCNTL2 bit, 1) is cleared, the SCRIPTS processor overwrites the DBC register with the length of the CDB: 6, 10, or 12 bytes. If the Vendor Unique Enhancement 1 (VUE1) bit (SCNTL2, bit 1) is cleared and the SCSI group code is a vendor unique code, the chip receives the number of bytes in the count. If the VUE1 bit is set, the chip receives the number of bytes in the byte count regardless of the group code.

Legal Forms

```
MOVE count, address, WITH phase
MOVE count, address, WHEN phase
MOVE count, PTR address, WITH phase
MOVE count, PTR address, WHEN phase
MOVE FROM address, WITH phase
MOVE FROM address, WHEN phase
```

3.2.12 MOVE MEMORY

```
MOVE MEMORY[NO FLUSH]count, source_address,
destination_address
```

Supported by All LSI Logic SCSI SCRIPTS Processors; No Flush option is available on all SCRIPTS processors except for the LSI53C770.

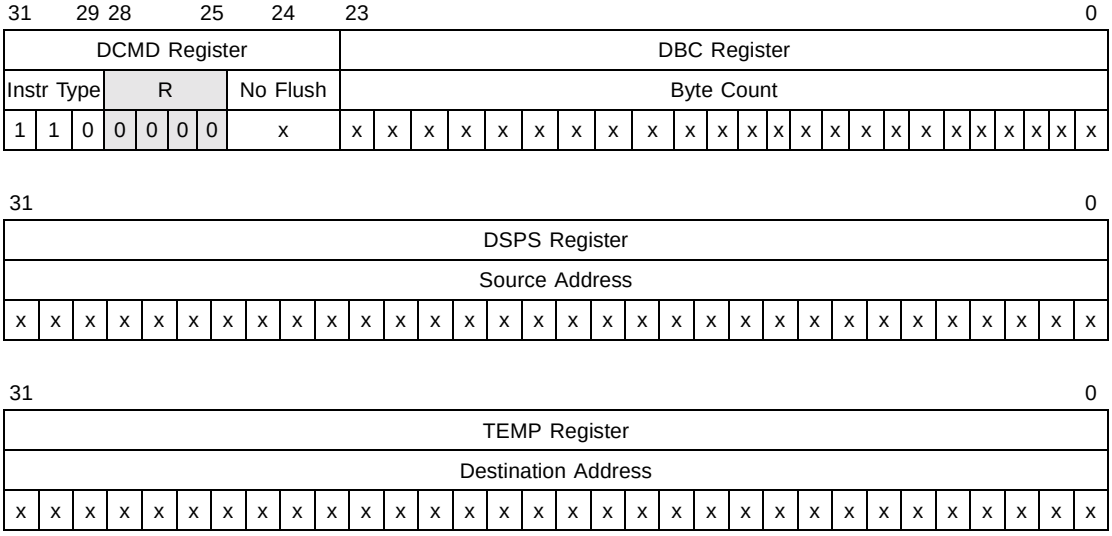
Definition Memory-to-Memory Move (DMA).

Operands This command has the following operands:

NOFLUSH	Allows the SCRIPTS processor to perform the Move Memory without flushing the prefetch buffer.
count	A 24-bit expression which indicates the number of bytes to transfer.
source_address	Absolute 32-bit starting address of the data in memory.
destination_address	Absolute 32-bit destination address of where to move the data.

Example `MOVE MEMORY 1024, Source_Buffer, Dest_Buffer`

Figure 3.12 MOVE MEMORY Format



Field(s) This command has the following fields:

Instruction Type Memory-to-Memory Move.

No Flush When this bit is set, the SCRIPTS processor performs the Move Memory without flushing the prefetch buffer. When this bit is cleared, the instruction automatically flushes the prefetch buffer. The No Flush option should be used if the source and destination are not within four instructions of the current Move Memory instruction.
This bit has no effect unless instruction Prefetching is enabled, by setting the Prefetch Enable bit in the DMA Control (DCNTL) register.

Register Definition(s) The information listed below describes the DBC, DSPS, and TEMP registers.

Byte Count A 24-bit number indicating the number of bytes to transfer.

Source Address	Absolute 32-bit starting address of the data in memory.
Destination Address	Absolute 32-bit destination address of where to move the data.

Description

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

If both addresses are in system memory, then the SCRIPTS processor functions as a high-speed DMA controller, able to move data at speeds up to 47 Mbytes/s without using the processor or its cache memory.

If just the destination address is in the system memory and the source is within the chip address space, then the instruction performs a register store to external memory.

If just the source address is in the system memory and the destination is within the chip address space, then the instruction performs a register load from external memory.

Notes

The Indirect Mode is not allowed for the Move Memory instruction.

If cache line bursting is not enabled, the source and destination addresses must be on the same byte boundary. If cache line bursting is enabled and the byte count is larger than 32, the lower four bits of the source and destination addresses must be identical. If these conditions are not met, an illegal instruction interrupt is generated.

If the chip is only I/O mapped, it cannot do memory-to-register or register-to-memory moves.

Legal Forms

```
MOVE MEMORY count, src_address, dest_address
```

3.2.13 MOVE REGISTER

```
MOVE {register | {data8} | register operator data8} TO
register [WITH CARRY]
```

Supported by

All LSI Logic SCSI SCRIPTS Processors; additional functionality supported by the LSI53C825A, LSI53C875, LSI53C876, LSI53C885,

LSI53C895, LSI53C895A, LSI53C896, LSI53C1000, LSI53C1010, LSI53C1010R, LSI53C1000R.

Definition

Register to Register Move.

Operands

This command has the following operands:

register One of the registers listed in the chip register set section in Chapter 6 of this manual. Either the register address or register name may be used in this instruction.

data8 Is an expression or value that evaluates to an 8-bit unsigned number. In all but the LSI53C770/810/860, SFBR may be substituted for data8 to add two register values. Bit 23 of the first Dword of the instruction indicates that the SFBR is to be used instead of a data8 value.

operator One of the following operators: '|' (OR), '&' (AND), SHL (Shift Left), SHR (Shift Right), XOR, '+' (Add), '–' (Subtract). The enhanced Move Register instruction does not support the SHL or SHR operators. See the appropriate product technical manual for detailed information on the supported operations.

WITH CARRY Adds in the current value of the CARRY bit from the ALU during a "+" or "–"' operation. It is not allowed for any other operations.

Example

```
MOVE 0xFF TO SFBR
MOVE SCNTL1 & 0x01 TO SCNTL1
```

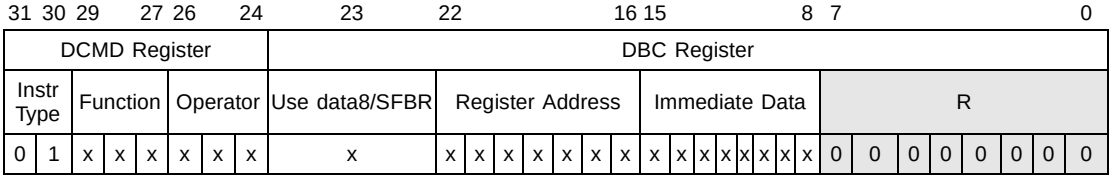
For LSI53C825A, LSI53C875, LSI53C876, LSI53C885 and LSI53C895 only:

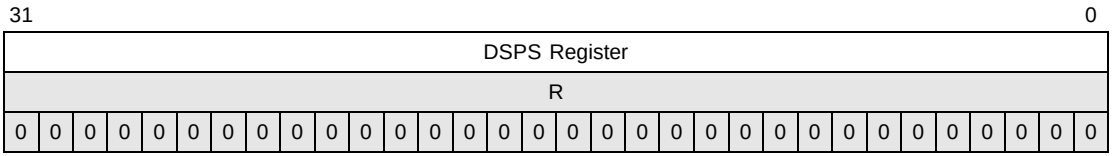
```
MOVE SCRATCHA + SFBR to SFBR
MOVE SCRATCHA XOR SFBR to SFBR
```

Subtraction (SFBR – SCRATCHA)

```
MOVE SCRATCHA XOR 0xFF to SCRATCHA
MOVE SCRATCHA + 1 to SCRATCHA
MOVE SCRATCHA + SFBR to SFBR
```

Figure 3.13 MOVE REGISTER Format





Field(s) This command has the following fields:

Instruction Type Read/Write.

Function The function bits select the desired register operation in either the target or initiator role.
101 - Move the SFBR register to the specified destination register
110 - Move the specified register to the SFBR register
111 - Read a specified register, modify it, and write the result back into the same register

Operator Specifies which logical or arithmetic operation will be performed.
000 - Move, no modification performed
001[1]- Shift source left one bit, store result in destination
010 - OR immediate data with source, store result in destination
011 - XOR immediate data with source, store result in destination
100 - AND immediate data with source, store result in destination
101[1]- Shift source right one bit, store result in destination
110 - ADD immediate data to source, store result in destination
111 - Add in immediate data plus Carry bit to source; store result in destination

1. Data is shifted through the Carry bit and the Carry bit is shifted into the data byte.

Register Definition(s)

The information listed below describes the DBC and DSPS registers.

Use data8/SFBR (not with the LSI53C770/ 810/860)	When this bit is set, SFBR will be used instead of the data8 value during a Read/Write instruction. This allows the user to add two register values.
Register Address	A 7-bit value that specifies which register to use as the source register for the instruction.
Immediate Data	An 8-bit value that will be used as the second operand in the logical and arithmetic functions. For the move function, the specified data is stored in the destination register.

Description

The Move Register instruction allows a register read-modify-write, or a move to/from a register from/to the SFBR register.

The SCRIPTS processor does not provide a true move from any source register to any destination register. To accomplish this, two register move instructions must be used. First move the source register to the SFBR register, then move the SFBR register to the desired destination register. The two register names in each line must be identical, or one must be SFBR. The two registers must be byte-aligned. If the 32-bit absolute addresses of the source and destination registers are known, then a register to register move can also be accomplished by using a Memory-to-Memory Move instruction. However, a SCRIPTS instruction written in this manner will be less portable to other machines than if the previous method is used.

Caution must be exercised when this instruction is used. Writing to certain registers could have adverse effects on the SCSI bus or chip operation. When a register is written or read, side effects may occur; the degree and possibility of these effects must be clearly understood. The LSI53C7XX/8XX/10XX family technical manuals contain detailed descriptions of individual register and bit operations.

The Add and Subtract operators can be used for loop counters in SCRIPTS programming. To subtract one value from another, first XOR the value to subtract (subtrahend) with 0XFF, and add 1 to the resulting value. This creates a 2's complement of the subtrahend. The two values can then be added to obtain the difference.

For LSI53C825A, LSI53C875, LSI53C876, LSI53C885, LSI53C895, LSI53C895A, LSI53C896, LSI53C1000, LSI53C1010, LSI53C1010R, LSI53C1000R only:
These chips allow use of the SFBR register for easier addition, subtraction, and comparison of two separate values within the chip. The instruction can perform the specified operation on the specified register and the SFBR, then store the result back to the specified register or the SFBR. The SFBR is used in place of the data8 value in the Read/Write operation. Subtraction cannot be used when the SFBR is used instead of a data8 value, because the SFBR value is not known at compile time.

Notes

The mathematical operation is performed by the chip during execution, not by the assembler when the SCRIPTS routine is being assembled.

Legal Forms

In the following, where the word register appears twice for an instruction, the register name must be the same name for both the source and destination, not two different register names.

```
Move register to register
Move data8 to REGISTER
Move REGISTER SHL REGISTER
Move REGISTER | data8 to REGISTER
Move REGISTER XOR data8 to REGISTER
Move REGISTER & data8 to REGISTER
Move REGISTER SHR REGISTER
Move REGISTER + data8 to REGISTER
Move REGISTER + data8 to REGISTER with Carry
Move REGISTER - data8 to REGISTER
Move data8 to SFBR
Move REGISTER to SFBR
Move REGISTER SHL SFBR
Move REGISTER | data8 to SFBR
Move REGISTER XOR data8 to SFBR
Move REGISTER & data8 to SFBR
Move REGISTER SHR SFBR
Move REGISTER + data8 to SFBR
Move REGISTER - data8 to SFBR
Move REGISTER + data8 to SFBR with Carry
Move SFBR SHL REGISTER
Move SFBR | data8 to REGISTER
Move SFBR XOR data8 to REGISTER
Move SFBR & data8 to REGISTER
Move SFBR SHR REGISTER
Move SFBR + data8 to REGISTER
Move SFBR - data8 to REGISTER
```

```
Move SFBR + data8 to REGISTER with Carry
```

Additional Forms for LSI53C825A/LSI53C875/LSI53C876/ LSI53C885/LSI53C895

```
Move SFBR to REGISTER
Move REGISTER | SFBR to REGISTER
Move REGISTER XOR SFBR to REGISTER
Move REGISTER & SFBR to REGISTER
Move REGISTER + SFBR to REGISTER
Move REGISTER + SFBR to REGISTER with Carry
Move REGISTER | SFBR to SFBR
Move REGISTER XOR SFBR to SFBR
Move REGISTER & SFBR to SFBR
Move REGISTER + SFBR to SFBR
Move REGISTER - SFBR to SFBR
Move REGISTER + SFBR to SFBR with Carry
Move SFBR to REGISTER
Move SFBR | SFBR to REGISTER
Move SFBR XOR SFBR to REGISTER
Move SFBR & SFBR to REGISTER
Move SFBR + SFBR to REGISTER
Move SFBR - SFBR to REGISTER
Move SFBR + SFBR to REGISTER with Carry
```

3.2.14 NOP

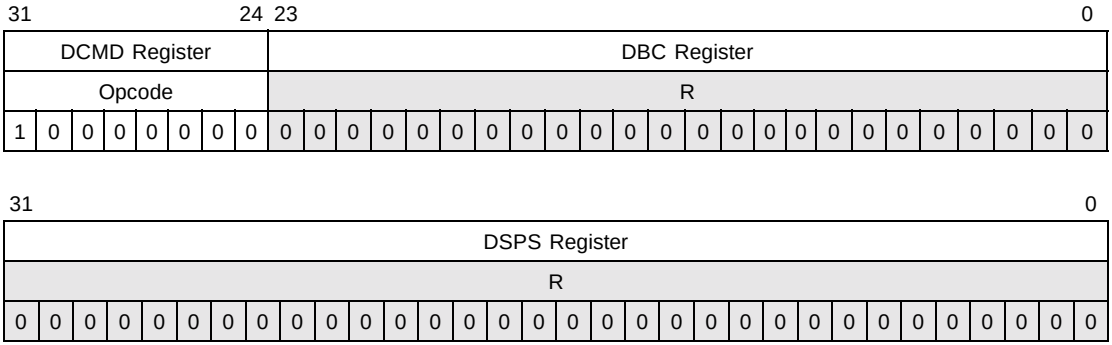
Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition No operation.

Operands This command has the following operands:

None.

Figure 3.14 NOP Format



Field(s) This command has the following field:

Opcode No Operation.

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

Legal Forms `NOP`

3.2.15 RESELECT

Section 9.4, “Synchronous Negotiation and Transfer,” has additional information about table indirect mode used during RESELECT.

```
RESELECT {FROM Address | ID}, {REL(Address) | Address}
```

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition Reselect the SCSI initiator device.

Operands This command has the following operands:

FROM Address Indicates the table indirect mode.

ID ID Number of the SCSI initiator being selected.

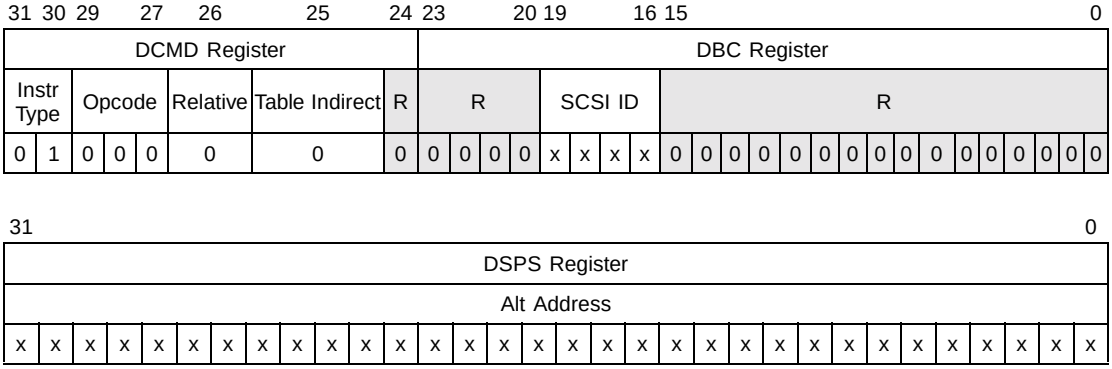
REL Indicates indirect addressing.

Address A 32-bit address that represents the address of the next instruction being fetched when the chip is selected or reselected.

Example

```
RESELECT host_1, rsel_addr
RESELECT FROM entry_2, REL rsel_addr
```

Figure 3.15 RESELECT Format



Field(s) This command has the following fields:

Instruction Type I/O.

Opcode Reselect instruction.

Relative Mode Indicates that the 24-bit address is an offset from the current program counter.

Table Indirect Mode The SCSI ID, synchronous, and wide parameters should be loaded offset from the Data Structure Address.

Register Definition(s) The information listed below describes the DBC and DSPS registers.

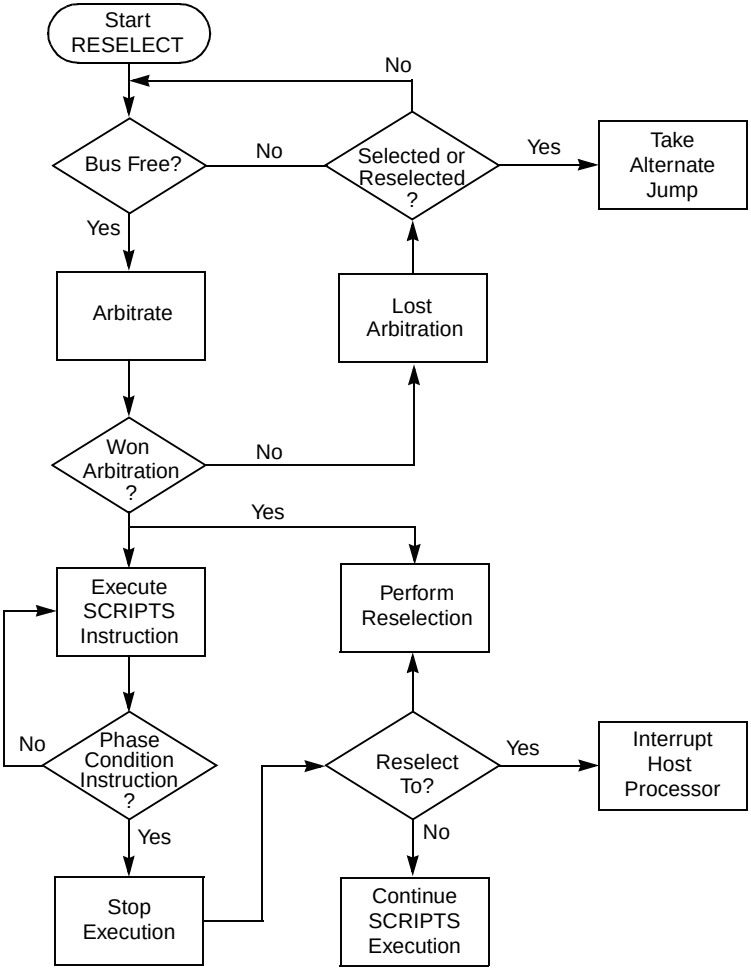
SCSI ID Identifies the SCSI initiator to be reselected. This 4-bit field specifics the encoded destination ID. This file is part of the address if the table indirect mode is used.

Alternate Address Specifies the memory address to fetch the next instruction if the SCRIPTS processor is selected or reselected.

Description

The chip waits for Bus Free, arbitrates for the SCSI bus, then performs a reselection. If the chip loses arbitration it will wait again for Bus Free and continue trying until it is successful, unless there is a bus initiated interrupt. Once arbitration is won, the SCRIPTS processor will continue to execute instructions until an interrupt or any instruction related to the SCSI bus is issued. If arbitration terminates because of a bus initiated selection or reselection, the chip will use the 32-bit jump address value to fetch the next instruction and begin execution at that address. When the instruction completes then the next sequential instruction is fetched and executed. The Reselection process is illustrated in Figure 3.16.

Figure 3.16 Reselection Instruction



Notes The REL keyword, which indicates relative addressing, is unrelated to the declarative keyword RELATIVE that establishes relative buffers.

Legal Forms

```
RESELECT scsi_id, address
RESELECT FROM table_entry, address
RESELECT scsi_id, REL(address)
RESELECT FROM table_entry, REL(address)
```

3.2.16 RETURN

```
RETURN [, {IF | WHEN}[NOT][ATN | Phase] [AND | OR] [data[,
AND MASK data]]]
RETURN [, {IF | WHEN}[NOT] CARRY]
```

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition SCSI Transfer Control - Return from a Subroutine.

Operands This command has the following operands:

WHEN Forces the SCRIPTS engine to wait for a valid SCSI bus phase before continuing. A valid phase is indicated by assertion of the SREQ/ signal.

IF Causes the SCRIPTS processor to immediately check for a valid SCSI bus phase. IF should not be used when comparing for a phase as this could yield unpredictable results. The only exception is using a WHEN conditional just prior to the IF conditional for any given sequence of phase checks.

NOT Negates the comparison. It clears the True bit if present, otherwise the True bit is set.

Phase Specifies the Message, Command/Data, and Input/Output bit values that identify the SCSI phase in the instruction. The desired phase value is compared with the actual values of the SCSI phase lines before the SCRIPTS processor performs the instruction. This field is only valid for the initiator mode and should not be used in the target mode.

ATN Indicates that a return should take place based on the state of the initiator SATN/ signal. This field is valid only for the target mode and should not be used in the initiator mode.

data Represents an 8-bit value that is stored in the data field of the instruction. In addition the Compare Data bit is set.

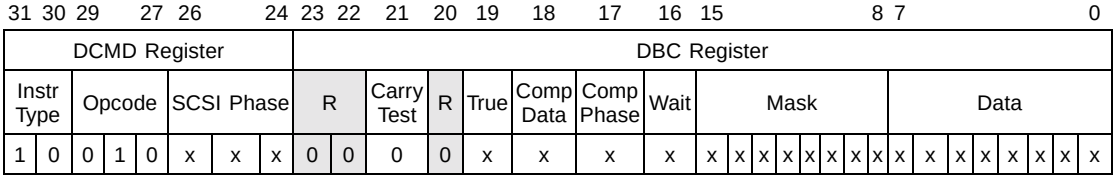
MASK Represents an 8-bit value that is stored in the mask field of the instruction. Any bit that is set in the mask causes the corresponding bit in the data byte to be ignored at the time of the comparison.

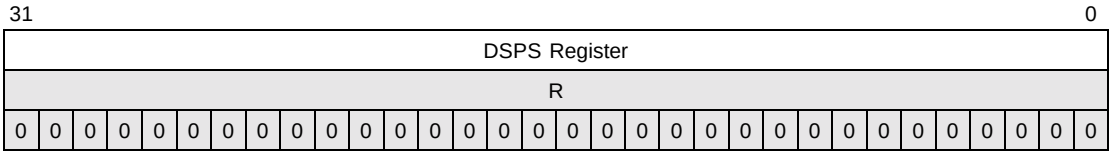
CARRY Indicates that a return should take place based on the value of the Carry bit in the ALU.

Example

```
RETURN
RETURN WHEN DATA_OUT
```

Figure 3.17 RETURN Format





Field(s) This command has the following fields:

Opcode Transfer Control, Return instruction.

SCSI Phase These bits reflect the actual values of the SCSI phase lines.

The values in Table 3.14 define the SCSI information transfer phase. The LSI53C10XX chips, with dual transition timing capabilities define two transfer phases, ST for single transition timing, and DT for dual transition timing.

Table 3.14 SCSI Phase Bit Values (RETURN Format)[1]

Phase	Message	Command/Data	Input/Output
DATA_OUT[2] (ST_DATA_OUT)[3]	0	0	0
DATA_IN[2] (ST_DATA_IN)[3]	0	0	1

Table 3.14 SCSI Phase Bit Values (RETURN Format)[1] (Cont.)

Phase	Message	Command/Data	Input/Output
COMMAND	0	1	0
STATUS	0	1	1
RES4[4] (DT_DATA_OUT)[3]	1	0	0
RES5[4] (DT_DATA_IN)[3]	1	0	1
MESSAGE_OUT	1	1	0
MESSAGE_IN	1	1	1

1. 0 - False, negated; 1 - True, asserted. For these phases, SEL is negated and BSY is asserted.
2. All chips except LSI53C10XX.
3. LSI53C10XX chips.
4. RES4 and RES5 are reserved SCSI phases except in the LSI53C10XX chips.

Register Definition(s)

The information listed below describes the DBC and DSPS registers.

Carry Test When this bit is set, true/false comparisons may be made based on the ALU Carry bit. The Carry test may not be combined with other types of comparisons.

True Transfer on TRUE/FALSE condition.
0 - Transfer if condition is FALSE
1 - Transfer if condition is TRUE

Compare Data Compare data byte to the SFBR register.
0 - Do not compare data
1 - Perform comparison

Compare Phase Compare current SCSI phase to SCSI phase field or SATN/. This bit is set whenever the Phase operand is used.
0 - Do not compare phase
1 - Perform comparison

Wait Wait for valid phase. This bit is set by the WHEN operand in the instruction, and cleared by the IF operand.
0 - Perform comparison immediately
1 - Wait for valid phase (SREQ/ asserted by target)

Mask An 8-bit field that masks the value in SFBR before the comparison with the data field in the instruction takes place. As a result of this operation, any bits that are set cause the corresponding bit in the data byte to be ignored. If this field is not specified, a mask of 0x00 is used.

Data An 8-bit field that is compared with the incoming data after the mask operation with the mask byte takes place. Comparison indicates either an equal or not equal condition. If the Data field is not specified, the compare data bit is cleared and 0x00 is coded for both the mask and data bytes.

Description

The SCSI RETURN instruction is a conditional return from a subroutine to the effective address, stored in the chip's TEMP register, if the SCSI phase, data, or attention condition compares true with the condition specified in the instruction.

When the optional data field is used, it is compared to the SFBR. This contains the most recent byte of any kind of data that has been moved into the SFBR register. The SCSI SCRIPTS program determines which routine to execute next based on actual data values received. Using a series of these comparisons, the algorithm processes complex sequences with no intervention required by the external processor.

When the optional MASK keyword and its associated value are specified the SCRIPTS processor allows selective comparisons of bits within the data byte. During the comparison, any bits that are set in the mask byte will cause the corresponding bit in the data byte to be ignored for the comparison.

Notes

If a RETURN instruction is executed without any previous CALL instruction, then there is no proper return address in the chip's TEMP register. This may cause the chip to generate an illegal opcode after the return.

Legal Forms

```
RETURN
RETURN, IF ATN
RETURN, IF Phase
RETURN, IF CARRY
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 CARRY
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 CARRY
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 CARRY
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
```

3.2.17 SELECT

Section 9.4, “Synchronous Negotiation and Transfer,” has additional information about table indirect mode used during SELECT.

```
SELECT [ATN] {FROM Address | ID}, {REL(Address) | Address}
```

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition Select SCSI target device.

Operands This command has the following operands:

FROM Address Indicates table indirect mode.

ID The ID Number of the SCSI target being selected.

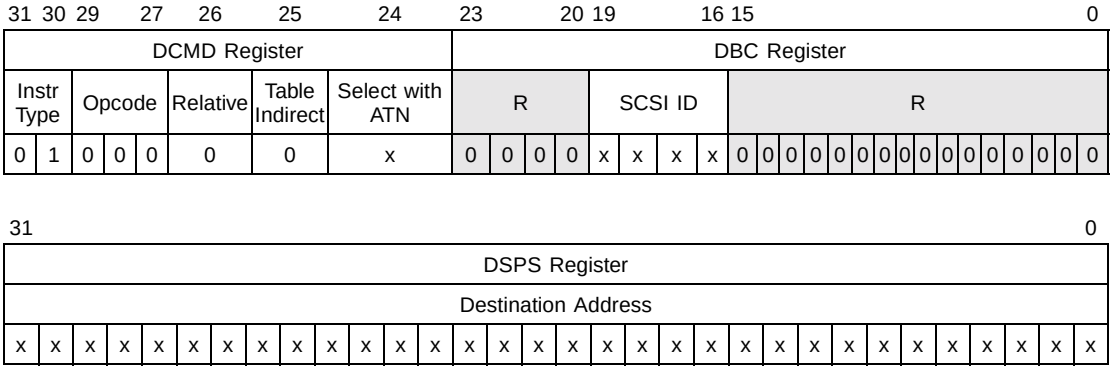
REL Indicates the use of relative addressing.

Address A 32-bit address (or 24-bit offset) that represents the address of the next instruction to fetch if the chip is selected or reselected by another device.

Example

```
SELECT host_1, sel_addr
SELECT FROM entry_2, sel_addr
```

Figure 3.18 SELECT Format



Field(s)

This command has the following fields:

Instruction Type I/O.

Opcode Select instruction.

Relative Mode Indicates that the 24-bit address is an offset from the current program counter.

Table Indirect Mode Indicates that the SCSI ID and synchronous and wide parameters should be loaded offset from the Data Structure Address.

Select with ATN Indicates whether or not the SCSI ATN/ signal should be asserted.

Register Definition(s)

The information listed below describes the DBC and DSPS registers.

SCSI ID Identifies the SCSI target to be selected. This 4-bit field specifies the encoded destination ID. This field is reserved if table indirect mode is used.

Destination Address Specifies the memory address to fetch the next instruction if the chip is selected or reselected during the selection.

Description

The chip waits for Bus Free, arbitrates for the SCSI bus, then performs a selection. If the chip loses arbitration it repeats the process until it is successful, unless there is a bus initiated interrupt. After winning

arbitration, the SCRIPTS processor continues to execute instructions until an interrupt or any instruction related to the SCSI bus is issued. If arbitration terminates because of a bus initiated selection or reselection, the chip uses the 32-bit jump address value to fetch the next instruction and begins execution at that address. When the instruction is completed then the next sequential instruction is fetched and executed.

Notes The REL keyword, which indicates relative addressing, is unrelated to the declarative keyword RELATIVE that establishes relative buffers.

Legal Forms

```
SELECT scsi_id, address
SELECT FROM table_entry, address
SELECT ATN scsi_id, address
SELECT ATN FROM table_entry, address
SELECT scsi_id, REL(address)
SELECT FROM table_entry, REL(address)
SELECT ATN scsi_id, REL(address)
SELECT ATN FROM table_entry, REL(address)
```

3.2.18 SET

```
SET {ACK|ATN|TARGET|CARRY}[and {ACK | ATN | TARGET | CARRY}
...]
```

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition Asserts SCSI ACK or ATN, or sets internal flags.

Operands This command has the following operands:

ACK Sets the Assert SCSI ACK bit.

ATN Sets the Assert SCSI ATN bit.

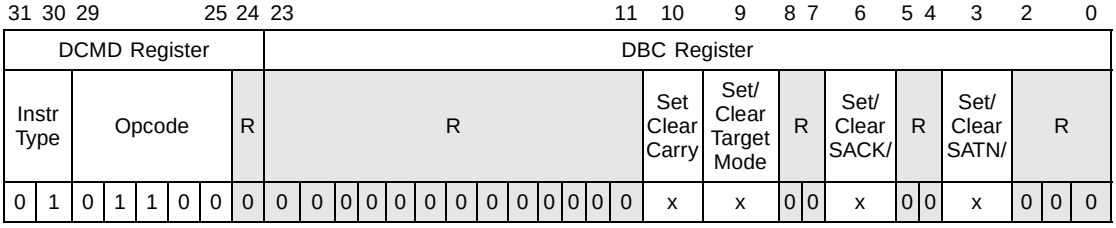
TARGET Sets the Set Target role bit.

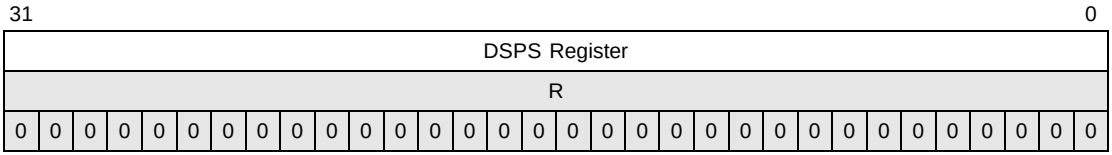
CARRY Sets the CARRY bit in the ALU.

Example

```
SET TARGET
SET ACK and TARGET
```

Figure 3.19 SET Format





Instruction Type I/O.

Opcode Set instruction.

Register Definition(s)

The information listed below describes the DBC and DSPS registers.

Set/Clear Carry
1 - sets the Carry bit in the ALU
0 - has no effect

Set/Clear Target Mode
1 - places the chip into target mode
0 - has no effect

Set/Clear SACK/
1 - asserts the SCSI acknowledge signal
0 - has no effect

Set/Clear SATN/
1 - asserts the SCSI attention
0 - has no effect

Description

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

Legal Forms

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

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

3.2.19 STORE

```
STORE [NOFLUSH] register, byte_count,
[DSAREL(]destination_address[)]
```

Supported by All except the LSI53C770 and LSI53C815.

Definition Store data from an internal chip register to memory.

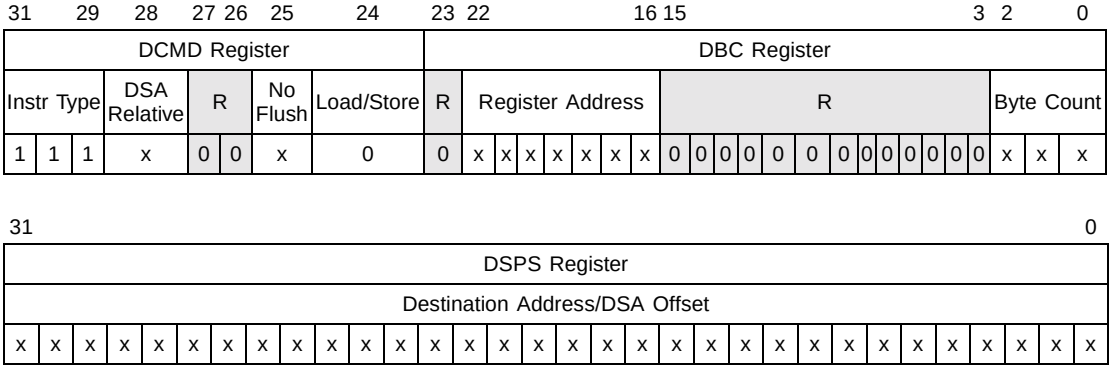
Operands This command has the following operands:

NOFLUSH	Indicates that the prefetch buffer should not be flushed when the instruction executes.
register	The register names in the chip operating register set.
byte_count	Number of bytes [1:4] to be transferred from the source_address.
DSAREL	Indicates that the source_address is an offset and should be added to the DSA register to obtain the physical address (DSA relative). Note: the FROM keyword can still be used to indicate DSA relative addressing, but it is being phased out in favor of DSAREL.
destination_address	Physical address or offset from the DSA to obtain the physical address of the destination.

Example

```
STORE SCRATCHA0, 4, data_buf
STORE SCRATCHA3, 2, DSAREL (0x02)
STORE NOFLUSH SCRATCHA0, 4, data_buf
```

Figure 3.20 STORE Format



Field(s) This command has the following fields:

Instruction Type Load/Store.

DSA Relative Indicates source address location.
0 - DSPS contains actual address of data to load
1 - DSPS contains a 24-bit offset value that is added to the DSA to determine the source address

No Flush When this bit is cleared, the prefetch buffer is flushed during the Store instruction. When set, the prefetch buffer is not flushed automatically on a Store instruction.

Load/Store This field defines whether the instruction will be executed as a Load or a Store.
0 - Store instruction
1 - Load instruction

Register Definition(s) The information listed below describes the DBC and DSPS registers.

Reg Addr These bits select the register to load within the chip operating register set.

Byte Count A 3-bit number indicating the number of bytes to transfer.

Destination Addr Actual address (or offset from the DSA) of the destination address.

Description The Store instruction is more efficient than the Move Memory instruction when moving data from an internal register of the chip to memory. It is a two Dword instruction. This instruction may be used to move up to

4 bytes. The number of bytes to store is indicated by the low order bits in the first Dword of the instruction, as illustrated in Table 3.15.

Table 3.15 Low Order Bit Options

DBC Bits [17:16] (Register Address bits A1-A0)	Number of Bytes to Store
00	1, 2, 3, or 4
01	1, 2, or 3
10	1 or 2
11	1

Notes The register address and memory address must have the same byte alignment and the byte count set so that it does not cross Dword boundaries. The memory address may not map back to the chip operating registers, although it may map back to a location in the SCRIPTS RAM. If these conditions are violated, a PCI illegal read/write cycle will occur and the chip will issue an Interrupt (Illegal Instruction Detected) immediately following, because the intended operation did not happen.

Legal Forms

```
STORE register, byte_count, destination_address
STORE register, byte_count, DSAREL (destination_address)
STORE NOFLUSH register, byte_count, destination_address
```

3.2.20 WAIT DISCONNECT

```
WAIT DISCONNECT
```

Supported by All LSI Logic SCSI SCRIPTS Processors.

Definition Wait for SCSI bus disconnect.

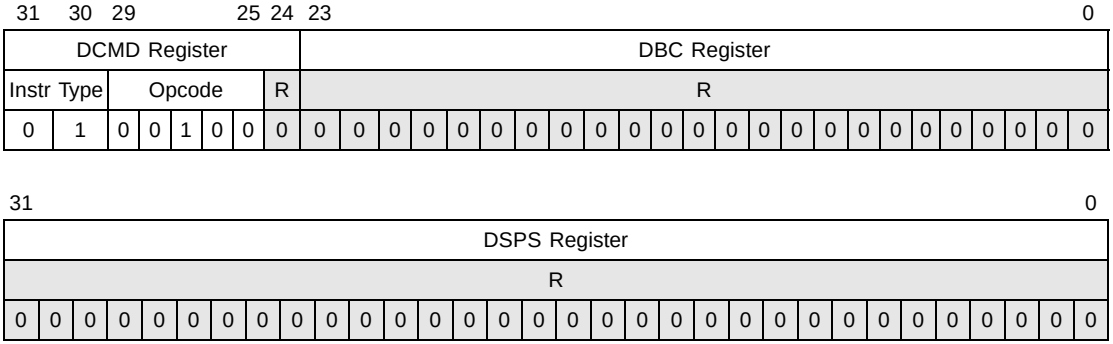
Operands This command has the following operands:

None.

Example

```
WAIT DISCONNECT
```

Figure 3.21 WAIT DISCONNECT Format



Field(s) This command has the following fields:

Instruction Type I/O.

Opcode Wait Disconnect.

Description The initiator waits for a disconnect from the SCSI bus. A legal disconnect is a loss of busy and select for the specified bus free time, following a DISCONNECT message or a COMMAND COMPLETE message. If the SCSI Disconnect Unexpected (SDU) bit (SCNTL2, bit 7) is cleared and a disconnect occurs, the next SCSI SCRIPTS instruction is executed. If the SDU bit is set and a disconnect occurs, an Unexpected Disconnect interrupt occurs.

Legal Forms

```
WAIT DISCONNECT
```

3.2.21 WAIT SELECT

```
WAIT SELECT {REL(Address) | Address}
```

Definition Wait for selection from initiator.

Operands This command has the following operands:

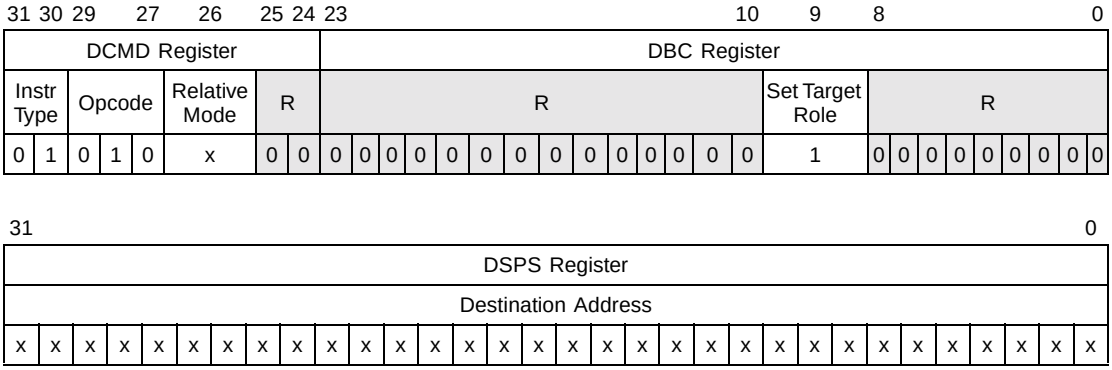
REL Indicates the use of relative addressing.

Address A 32-bit address (or 24-bit offset) of the next instruction to fetch if the chip is selected, or if the SIGP bit in the ISTAT register is set.

Example

```
WAIT SELECT alt_addr
WAIT SELECT REL(alt_addr)
```

Figure 3.22 WAIT SELECT Format



Field(s)

This command has the following fields:

Instruction Type I/O.

Opcode Wait Select instruction.

Relative Mode Indicates that the 24-bit address is an offset from the current program counter.

Register Definition(s)

The information listed below describes the DBC and DSPS registers.

Set Target Role 1 - places the chip into target mode
0 - places the chip into initiator mode

Destination Address Specifies the memory address to fetch the next instruction if the device is reselected during the selection attempt, or if the SIGP bit is set.

Description

The chip waits for a selection by another device on the SCSI bus. If the chip is already selected, then the next SCSI SCRIPTS is fetched and executed. When a bus initiated interrupt or reselect occurs, the chip changes to the initiator role, fetches the next instruction from the address pointed to by the 32-bit jump address, and continues execution. If the SIGP bit in the ISTAT register is set by the host processor, the chip will also fetch the instruction at the alternate address. The SCRIPTS

processor checks the SIGP bit before checking to see whether it has been reselected.

Legal Forms

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

3.2.22 WAIT RESELECT

```
WAIT RESELECT {REL(Address) | Address}
```

Definition Wait for reselection from target.

Supported by All LSI Logic SCSI SCRIPTS Processors.

Operands This command has the following operands:

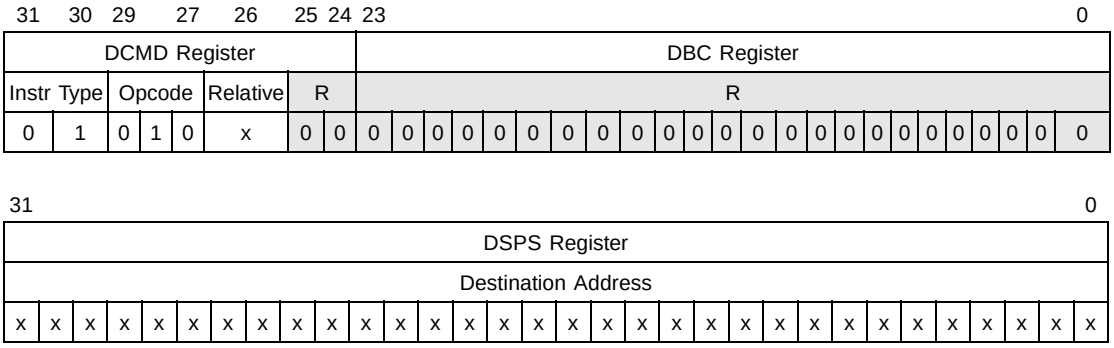
REL Indicates the use of relative addressing.

Address A 32-bit address (or 24-bit offset) of the next instruction to fetch if the chip is selected, or if the SIGP bit in the ISTAT register is set.

Example

```
WAIT RESELECT alt_addr
WAIT RESELECT REL(alt_addr)
```

Figure 3.23 WAIT RESELECT Format



Field(s) This command has the following fields:

Instruction Type I/O.

Opcode Wait Reselect.

Relative Mode Indicates that the 24-bit address is an offset from the current program counter.

Register Definition(s)

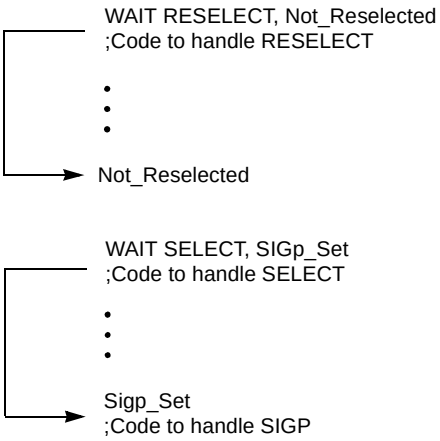
The information listed below describes the DBC and DSPS registers.

Destination Address Specifies the memory address of the next instruction to fetch if a reselection occurs or the SIGP bit is set by the host processor.

Description

The initiator waits to be reselected by a previously selected target device. If the chip is responding to a previous reselection, it fetches and executes the next instruction. If the chip has already responded to reselection, it immediately fetches the next instruction. If the operation completes as expected, the next instruction is fetched and executed by the SCRIPTS processor. However, if the chip is selected, then the alternate jump address should contain the address of a selection algorithm. Target instructions must include a WAIT in the address. That instruction's alternate address is the error recovery algorithm (for initiator role–reselect). The chip can determine exactly what happened and transfer control to the appropriate SCSI SCRIPTS algorithm. If the SIGP bit in the ISTAT register is set by the host processor, the chip will also fetch the instruction at the alternate address. This allows the driver program to schedule another I/O instead of waiting for the reselection to complete. This driver code activity is illustrated in Figure 3.24.

Figure 3.24 WAIT RESELECT and the SIGP Bit



Notes With the SCRIPTS processor 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. The SCRIPTS processor checks the SIGP bit before checking to see whether it has been reselected. SCSI SCRIPTS can be tuned for the various types of available target devices and executed with no external processor intervention.

Legal Forms

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

3.3 Instruction Examples

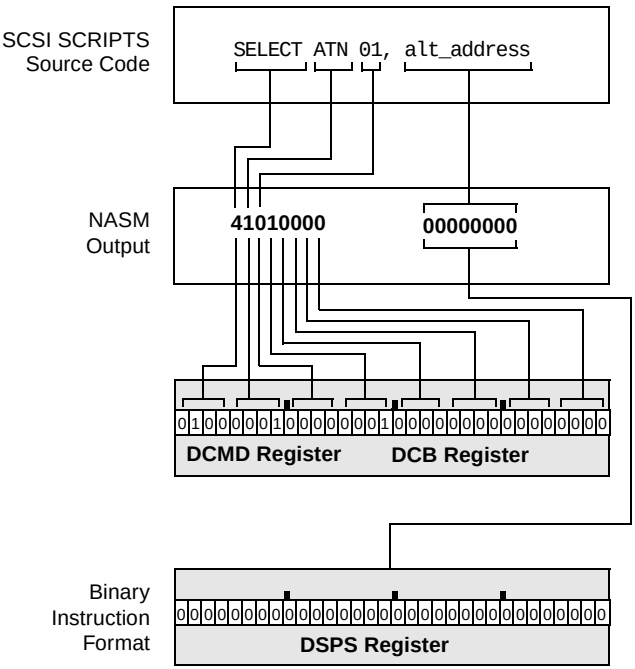
This section illustrates the operation of the five SCSI instruction types supported by the SCRIPTS processor. In each diagram, the SCSI SCRIPTS Source Code version shows how the operation would be expressed in the SCRIPTS language. This high-level textual format is translated by NASM into a hexadecimal format that is put inside a “C” language data declaration. After this intermediate form is compiled, the instruction exists in a binary form that can be loaded into host memory and fetched and executed by the SCRIPTS processor.

3.3.1 I/O Instruction Example

Figure 3.25 is an example of the processor when selecting the SCSI device with SCSI ID 01. The instruction is a Select With Attention, as indicated by the ATN keyword.

The SELECT instruction and ATN flag generates a value of 0x41 for the high order byte of the instruction, translating to a binary 01 for I/O Instruction type, 0b000 for the opcode, and a 1 in the ATN flag bit. The SCSI target identity (0b01) is encoded in the next byte. The rest of the bits are reserved and should remain cleared. The alternate address in the original SCRIPTS instruction is loaded into the DSPS register.

Figure 3.25 I/O Instruction Type



3.3.2 Memory Move Instruction Example

In this example, the processor moves eight bytes from the source address to the destination address relative to the source.

The MEMORY MOVE instruction generates an opcode of C0 for the high order byte of the instruction. The remaining bits of the DCMD register are reserved and must be set to zero. The DBC register contains a value of eight as directed by the translation of the command_length of 0x08. Figure 3.26 shows the original SCRIPTS language form of the instruction, the SCRIPTS compiler output, and the binary form of the first 32-bit word of the instruction.

Figure 3.26 Memory Move Instruction Part 1

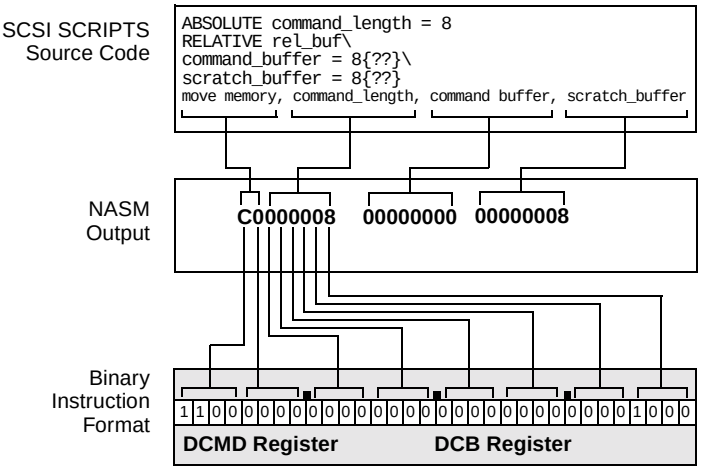
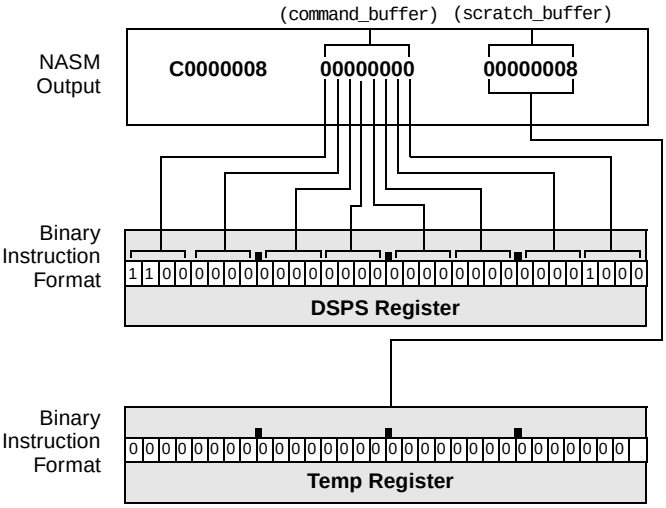


Figure 3.27 shows the Assembler output and the binary form of the second and third 32-bit words of the Memory Move instruction.

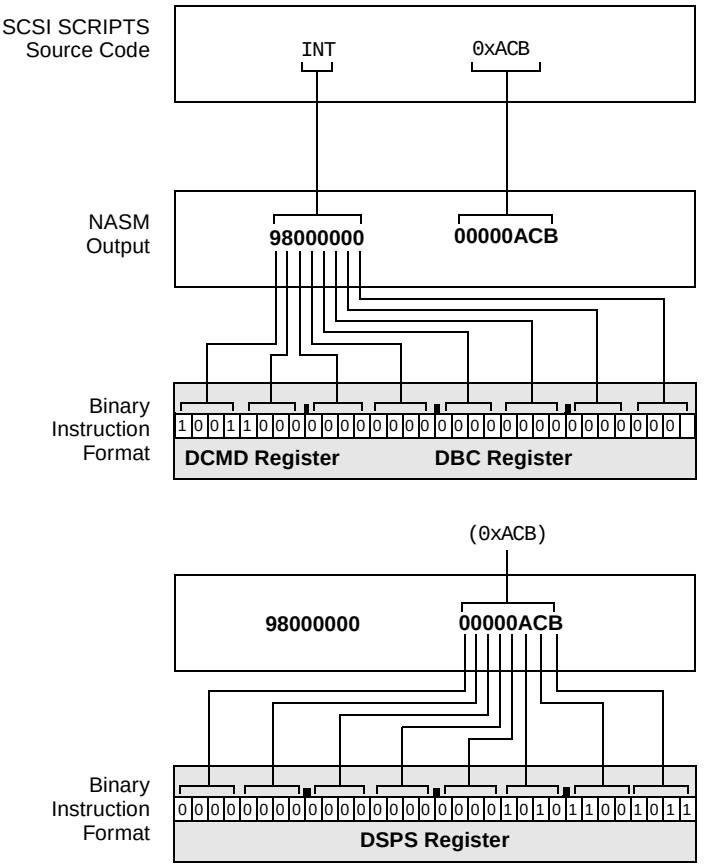
Figure 3.27 Memory Move Instruction Part 2



3.3.3 Transfer Control Instruction Example

In Figure 3.28, the processor performs an interrupt with a vector of 0xACB. The first version shows how the operation would be expressed in the SCRIPTS language. NASM translates the operation into the hexadecimal format shown. The hexadecimal format is then compiled producing the instruction in a binary form that can be loaded into host memory and put inside a “C” language data declaration. The INT instruction generates a hexadecimal value of 0x98 for the high order byte of the instruction, translating to 0b10 for Transfer Control, and 0b011 for the opcode for Interrupt.

Figure 3.28 Transfer Control Instruction

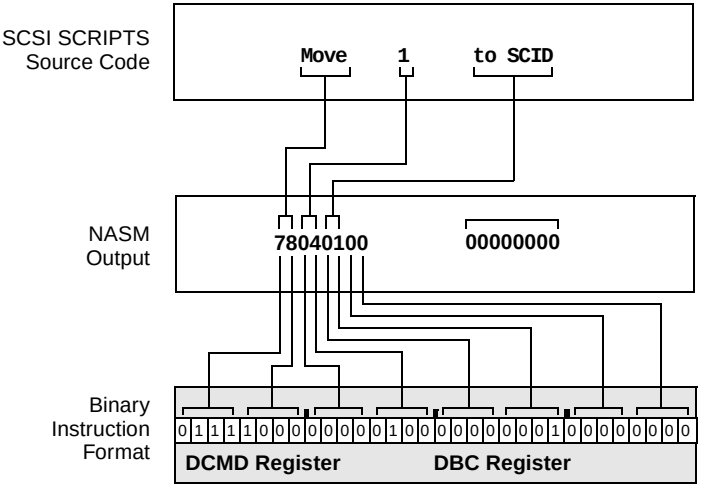


3.3.4 Read/Write Instruction Example

This example writes 0b01 into the SCSI Chip ID (SCID) register, as shown in Figure 3.29. This is illustrated by the translation of the hexadecimal compiler output into binary format.

The MOVE instruction is 78 in hexadecimal, translating into 0b01 for Read/Write; 0b111, the opcode for the Read/Modify/Write function; and 0b00 in the operator field to indicate that the instruction will operate on the immediate data and write to the destination register. The address of register SCID is 04 in hexadecimal, translating to a binary format for the Register Address bits of the DBC register.

Figure 3.29 Read/Write Instruction Example

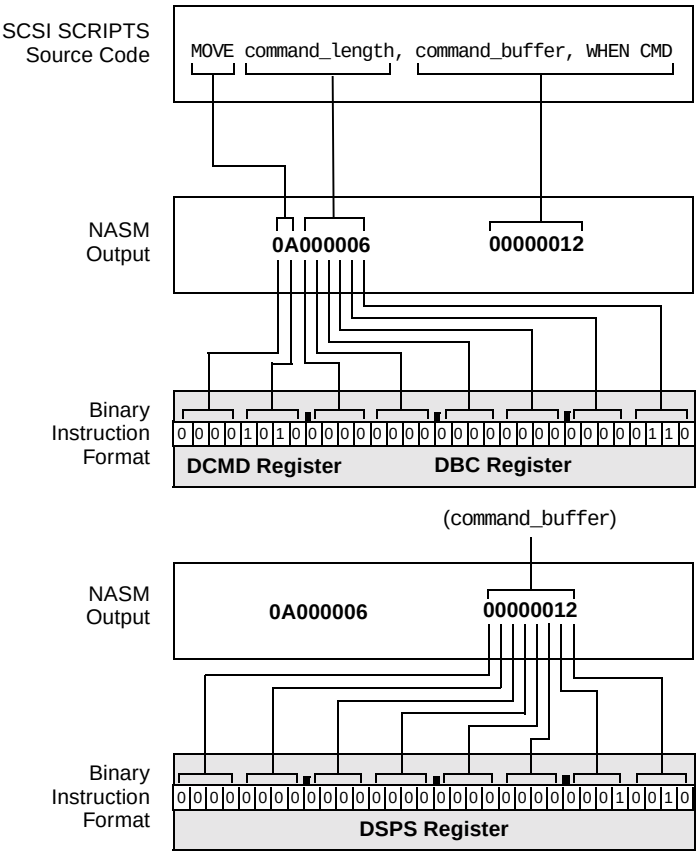


3.3.5 Block Move Instruction Example

In this example, shown in Figure 3.30, the processor waits for a valid phase (indicated by SREQ/ being asserted) and compares it to CMD phase. If the phase matches, the processor transfers the CDB from the address represented by the `command_buffer`. In the hexadecimal version of the first 32-bit word of the instruction, Move is represented by 0x0A, which translates into binary as an opcode of 00, indicating a Block Move instruction type. The 0b00 indicates that neither type of indirect addressing bits are on, 1 indicates that the processing is in the Initiator role, and 0b010 (Command) is the expected value of the SCSI phase lines. The command length is six bytes, indicated by 0x06. This length is loaded into the DBC register.

The bottom portion of the illustration shows the second 32-bit word of the instruction, defined by `command_buffer`. The Block Move instruction begins transferring data from this address. It is loaded into the DSPS register.

Figure 3.30 Block Move Instruction



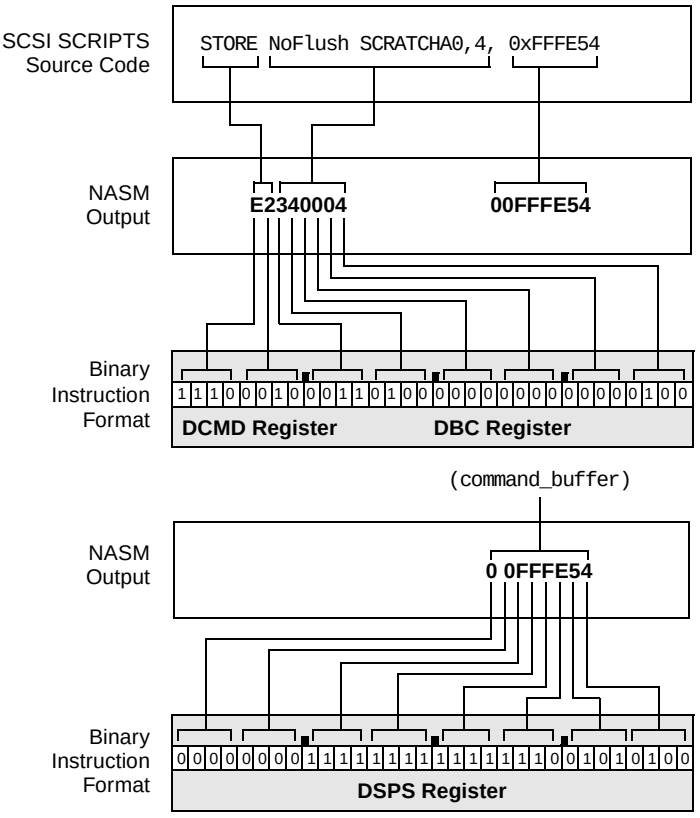
3.3.6 Load/Store Instruction Example

In this example, shown in Figure 3.31, the processor waits for a valid phase, indicated by assertion of SREQ/, and compares it to the CMD phase. If the phase matches, the processor then transfers the CDB from the address represented by the command_buffer. In the hexadecimal version of the first 32-bit word of the instruction, STORE with the No Flush option is represented by E2, which translates into binary as an opcode of 111, indicating a Load/Store instruction type. The 0 indicates that the DSPS value is the actual address to STORE from, and 0b0010 indicates that the prefetch buffer will not be flushed during the STORE, and that the SCRIPTS processor is performing a STORE rather than a

LOAD instruction. The data will be stored to the SCRATCHA register; one, two, three, or four bytes may be stored.

The bottom portion of the illustration shows the second 32-bit word of the instruction, defined by the command_buffer. The Block Move instruction begins transferring data from this address. It is loaded into the DSPS register.

Figure 3.31 Load/Store Instruction



Chapter 4
Using the LSI Logic Assembler NASM™

This chapter describes the LSI Logic Assembler (NASM) and contains the following sections:

4.1 Overview

The LSI Logic Assembler (NASM) is a DOS command line driven assembler that supports the LSI Logic SCSI SCRIPTS processor family. NASM creates a “C” header file from the SCSI SCRIPTS source file. It assembles SCSI SCRIPTS for inclusion into SCSI device driver software.

Inputs to the assembler are command line switches, as well as input and output file names. The assembler produces comprehensive error messages, cross referenced list files, and “C” include files. The source file may be created using any standard text editor that creates an ASCII file as output.

To assure portability, NASM does not provide support for directory paths. The resulting output file and the optional listing file will be placed in the directory where NASM is executed. Since the assembler is written in “C”, it can easily be ported to any non-DOS based development environment that offers a “C” compiler.

4.2 Using NASM

Before running the assembler, you must copy the assembler executable file directly into the directory from which the assembly will be performed. Entering `NASM` on the command line with no arguments produces a short description of all the valid switches. The NASM command line recognizes DOS wild card characters(“*”, “?”) in filenames. Usage:

```
NASM filename [options]
```

where:

filename	Name of the file you generated that is being assembled. Files should be specified in the standard DOS format:`[d:] [path] name.ext` The file name is the root file name of the `.ss` file unless otherwise indicated.
options	A series of options, listed in brief below, that modify the NASM output. The option is always preceded by a hyphen (-)
a [architecture]	Specifies SCSI architecture.
b	Generates binary cross reference values.
c	Changes from little endian to big endian. Not supported by all chips.
e [filename[.err]]	Saves error messages (filename optional).
l [filename[.lis]]	Generates cross reference (filename optional).
o [filename[out]]	Generates “C” source output (filename optional).
p [filename[.out]]	Generates partial “C” header (filename optional).
s [filename[.bin]]	Generates `.bin` format output (filename optional).
u	Excludes module termination record.

v	Verbose messages.
x	Lists patch offsets in cross reference listing.

4.3 Command Line Options

This section of the manual describes the NASM command line options.

4.3.1 Architecture

The -a option allows you to specify the LSI Logic chip for which you are generating code. The currently supported chips are listed in the table below, along with the corresponding number to enter to choose the architecture. An ARCH statement at the beginning of a SCRIPTS source file overrides any options typed on the command line. If the source file does not have an ARCH statement and no architecture is specified in the command line, NASM uses the default architecture, the LSI53C700, which is no longer supported.

Product Name	Command Line Entry
LSI53C770	-a 770
LSI53C810	-a 810
LSI53C810A	-a 810a
LSI53C825	-a 825
LSI53C815	-a 815
LSI53C825A (all package variations)	-a 825a
LSI53C875 (all package variations)	-a 875
LSI53C876	-a 876

Product Name	Command Line Entry
LSI53C885	-a 885
LSI53C895	-a 895
LSI53C895A	-a 895a
LSI53C896	-a 896
LSI53C1000/LSI53C1000R	-a 1000
LSI53C1010/LSI53C1010R	-a 1010

4.3.2 Binary Cross Reference Values

The `-b` option generates binary as well as hexadecimal opcodes in the listing file.

4.3.3 Error Listing File

The `-e` option generates an error message if errors occur during NASM assembly. If no file name is given, the `-e` option creates a file with the same root name as the source file, with a `.err` extension.

4.3.4 Listing File

The `-l` option creates an assembly listing (`.LIS`) file. When invoked, this option creates a file with the same root name as the source file and a `.LIS` extension, unless otherwise specified.

4.3.5 Output File

The `-O` option creates a “C” style output (`.OUT`) file. When invoked, this option creates a file with the same root name as the source file and a `.OUT` extension, unless otherwise specified.

4.3.6 Partial “C” Source

The `-p` option creates a partial “C” style output file with a `.out` extension, but no patch information is listed. Since it produces a subset of the same information as the `-o` option, it is mutually exclusive with the `-o` option, and should not be used at the same time. If the `-o` and `-p` options are both specified, the `-p` option always takes precedence. The portions of

the SCRIPTS out file that are eliminated when invoking the `-p` option are listed below. For additional information about the SCRIPTS output file, refer to Chapter 5, “The NASM Output File.”

`#define Ext_Count`	count of external variables
`char *External_Names[Ext_Count]`	array of external variable names
`#define E_buf_name.`	definition of the external buffer offset because it will always be zero
`#define Rel_Count`	count of relative buffers
`ULONG Rel_Patches [Rel_Count]`	array of relative patches
`#define R_buf_name`	define the relative buffer offsets
`#define Abs_Count`	which is a count of Absolute variables
`char *Absolute_Names[Abs_Count]`	which is an array of absolute names
`ULONG A_absolute_Used[]`	array of locations where absolute variables are used
Termination record	termination record is removed (as in the `-U` option)
`#define instruction 0x????????` is added	instruction count

4.3.7 .BIN Output

The `-s` option generates a file with a `.bin` extension.

4.3.8 Omit Termination Record

The `-u` option instructs the assembler to omit the INSTRUCTIONS and PATCHES information from the output file. It must be used when either the `-o` or `-p` options are used.

4.3.9 Verbose Messages

The `-v` option instructs the assembler to generate more comprehensive status messages.

4.3.10 Patch Offsets

The -x option produces an assembly level output file, including a list of patch addresses for each symbol. These addresses indicate where to patch each individual symbol value.

4.4 Example Assembler Command Lines

The following command lines are typical examples of how to use the various options.

```
NASM demoPCI.ss
```

This command line produces no output files, but allows a quick syntax check on the SCRIPTS instructions in the file named DEMOPCI.SS

```
NASM demoPCI.ss -a 875 -l -o -e errors.txt
```

This command line requests that NASM check the syntax and generate code for the LSI53C875 chip. It generates the listing, error log, and standard C header. Since no filenames were specified for the listing and C header files, they will take the name of the input file, but with .LIS and .OUT as the file extensions, respectively. The error log is sent to the file named ERRORS.TXT

4.5 How NASM Parses SCRIPTS Files

SCSI SCRIPTS programs contain a series of lines. Blank lines, lines containing only white space, and anything after a semicolon on a line are ignored.

The assembler is token oriented. It reads the source file and splits it up into tokens. White space and anything from a semicolon to the end of the line is not part of any token, and is ignored by the first pass of the assembler.

There are two types of tokens. Any string of consecutive letters, numbers, dollar signs, and underscores is a token. The second type of token consists of characters that are not part of other tokens. Anything that is not a letter, a digit, an underscore, or a dollar sign, will become a

token. For example, the string “xxx = 0x123; assign value to xxx” contains three tokens. “xxx” is a token, “=” is a token, and “0x123” is a token.

Numeric values may be specified in decimal, hexadecimal, octal, or binary format. Decimal numbers are specified by a string of digits that does not begin with a zero. Octal numbers are specified by a string of digits that begins with zero. Hex numbers are specified by a string consisting of “0x” or “0X” and the hex digits of the number. Both upper and lower case are allowed. A binary number is specified with “0b” or “0B”.

4.6 Assembler Declarative Keywords

To do its job efficiently, the assembler needs to recognize a set of commands that are different from the processor instructions. These commands, called declarative keywords, control the different aspects of code generation and are intended for the assembler’s use. In most cases, the declarative keywords will not produce executable code by themselves, but must be combined with processor instructions to generate assembled code.

The declarative keywords are grouped functionally in Table 4.1. They are listed alphabetically and defined in the remainder of this section.

Table 4.1 Keywords

Keyword	Function
Data Definition and Storage	
ABSOLUTE	Equates
RELATIVE, EXTERN	Storage Definition

Table 4.1 Keywords (Cont.)

Keyword	Function
TABLE	Table Addressing
Code Generation	
ARCH	Code Generation
Miscellaneous	
PROC	Module Definition
ENTRY	Code Entry Labels

4.6.1 ABSOLUTE

ABSOLUTE defines the symbol name by assigning it a numeric value. After you declare a name using ABSOLUTE, NASM substitutes this numeric value in each instruction where the name is used.

Syntax `ABSOLUTE name = expression`

Example
```
ABSOLUTE bytes = 2048; A sector is 2048 bytes
ABSOLUTE sectors = 4; A cluster has 4 sectors
ABSOLUTE cluster = bytes; cluster size
ABSOLUTE bytecnt = bytes; bytecnt is an indexing
variable
```

Description ABSOLUTE supplies a list of names, or labels, solely for the use of the assembler. NASM refers to this list when it is actually assembling the program.

4.6.2 ARCH

ARCH directs the assembler to generate instructions that are specific to a chip architecture.

Syntax `ARCH chip_number`

Fields chip_number

Example `ARCH 810A`
`ARCH 875`

Notes If used, this keyword should be placed before any executable statements so that the assembler knows which chip to generate code for. The chip architecture may also be specified on the assembler command line for the assembler using the `-A chip_number` option. ARCH takes precedence over the `-A` option in the NASM command line. The chip number entries should use the last three digits of the product number, as indicated in the example above.

4.6.3 ENTRY

ENTRY informs the driver program of the starting location of callable routines contained in a given SCRIPTS instruction. ENTRY allows the declaration of variables as entry points into the SCSI SCRIPTS instruction array. It defines the names and values of the variables, making them also available to the host development system.

Syntax `ENTRY label [, label ...]`

Example `ENTRY start, Data_Out_Entry`

Description The ENTRY keyword indicates which SCRIPTS entry points should be made visible to the driver code. Only those entry points named in the ENTRY keyword will generate information in the assembler output file.

Notes All entries must be used as a label somewhere in the SCRIPTS code, otherwise an error message will be reported.

4.6.4 EXTERN

EXTERN informs the assembler that a symbol should be resolved at link time. This keyword allows the declaration of variables that are defined external to the SCRIPTS program. EXTERN causes the assembler to keep an array of offsets into the SCRIPTS array that the driver can use to patch SCRIPTS instructions into the driver program.

Syntax

```
EXTERN label [, label ...] or
EXTERN label = data_specifier [, label =
data_specifier...]
a data specifier is:
{byte_val[, byte_val]} or count{byte_val | ??}
```

Fields A count is any valid constant with a value between 0 and 64 Kbytes.

Example

```
EXTERN buffer; a buffer in the driver
EXTERN buffer=1024{??}; same buffer, but now
; the debugger will have
; information about space
;  requirements
```

Description The first form of the EXTERN syntax is only provided for compatibility with older versions of the SCRIPTS compiler. The second form (with space requirements information for the debugger) should be used in all new programs. Declarative instructions never allocate memory, but give the debugger or driver code the information required to allocate the memory.

4.6.5 PASS

PASS allows the programmer to pass a “C” element unaltered to the SCRIPTS output file and on to the “C” compiler. Using this option avoids the need for run time patching of the addresses of SCRIPTS objects. PASS is typically used for two types of “C” elements; either an include statement or a literal string.

Syntax

```
PASS(element)
```

Examples

```
Include statement: PASS(include"SCRIPTS.h")
Literal string: Wait Reselect PASS(&alt_addr)
```

Description PASS tells NASM to pass everything between the left and right parentheses on to the output file, literally. Therefore, the passed statements can be read by the “C” compiler.

4.6.6 PROC

PROC builds output arrays with names other than the generic array name SCRIPT that NASM normally assigns to SCRIPTS opcode arrays. This is useful when more than one SCRIPTS file is used in a driver program. It also allows several output arrays to be created with specific code segments in each one. When SCRIPTS storage space is limited,

code can be divided into different sections where one section would fit in a limited space (such as SCRIPTS RAM) and the remaining code can be stored elsewhere.

Syntax `PROC label:`

Fields `label` is the name assigned to the SCRIPTS output array.

Examples `PROC Start:`

Description When a PROC keyword is used, the SCRIPTS output array in the `.out` file is given the name specified in `label`, overriding the default name SCRIPT. If additional PROC statements are used in the same SCRIPTS source file, NASM will create additional output arrays in the `.out` file with the name specified in `label` for each PROC statement.

4.6.7 RELATIVE

Use RELATIVE to begin the definition of a data structure named base`label` with offsets into the buffer specified by the labels. It allows the declaration of buffers to be positioned relative to one another. The expression used is the offset from the start of the relative data area where the buffer variable is located.

Syntax

```
RELATIVE label = expression [, label = expression...]
or
RELATIVE baselabel \
label = data_specifier [, label = data_specifier...]

a data specifier is:
{byte_val [, byte_val]}
or
count{byte_val |??}
```

Fields A byte_val is any valid constant with a value between 0 and 255. For example: 0x10 and 16 both represent a byte value of 16. Also, the special data value ‘??’ can be used to indicate that a byte should be reserved, but that it should not be initialized to a specific value. The SCRIPTS program does not allocate memory; this is done by “C” code in the SCRIPTS debugger or in the driver code. A count is any valid constant with a value between 0 and 64 Kbytes.

Example This example shows the typical use of the RELATIVE keyword. NASM syntax requires that no SCRIPTS statements span more than one line. However, in the case of RELATIVE, this would result in a very unreadable source code file. The following example demonstrates the use of the logical line continuation character '\'. When this character is used, the assembler appends the next line to the end of the current line.

```
RELATIVE data_buffer\
identify_msg_buf = 1{??}, \
synch_msgo_buf = {1,2,3,4,5},\
synch_msgi_buf = 5{??},\
cmd_buf = 12{??},\
W_cmd_buf = 12{??},\
stat_buf = 1{??},\
msg_in_buf = 1{??},\
disc_msg_in_buf = 2{??},\
read_cap_buf = {1,2,3,4,\
5,6,7,8},\
inquiry_buf = 36{??},\
request_sense_buf = 18{??},\
data_buf = 16384{??}
```

Description The RELATIVE keyword defines a template for a collection of data elements of the same or varying types, each of which can be accessed by a descriptive name, but no storage is allocated. It is up to the programmer to use the RELATIVE information that is placed in the output file to declare space in the driver program that RELATIVE maps to.

Notes The first form of the RELATIVE syntax example is only provided for compatibility with older versions of the SCRIPTS compiler. The second form (with baselabel definition) should be used for all new programs.
Since the SCRIPTS array will have only offsets from the base address of the buffer, the SCRIPTS elements containing references to relative buffers will need to be patched by the driver program after the buffer space is allocated.

4.6.8 TABLE

TABLE describes a data structure used with the table indirect addressing feature of the SCRIPTS processor. The starting location for the buffer is defined by the data structure address written to the DSA register. The expression specifies the offset into the buffer and is added to the starting address of the buffer (DSA register) to form the absolute address. This feature allows SCRIPTS to be programmed into a ROM.

Syntax

```
TABLE tablelabel \
label = data_specifier \
[, label = data_specifier...]
a data specifier is:
{byte_val [, byte_val]} or
count{byte_val | ??} or
ID{byte_val | ??}
```

Fields

A byte_val is any valid constant with a value between 0 and 255. For example, 0x10 and 16 both represent a byte value of 16. Also, the special data value '??' can be used to indicate that a byte should be reserved, but that it should not be initialized to a specific value. A count is any valid constant with a value between 0 and 64 Kbytes.

Example

This example shows the typical use of the TABLE keyword. NASM does not generate any output based on the TABLE keyword. This example is a template for a data structure that will be used in the driver program or in the SCRIPTS debugger. NASM assembler syntax requires SCRIPTS statements to span no more than one line. However, in the case of TABLE, this would result in a very unreadable source code file. The following example demonstrates the use of the logical line end character (\). When this character is used, NASM appends the next line to the end of the current line.

```
TABLE table_indirect\
stat_buf = {??},\ ;stat_buf = 1 byte
msg_in_buf= {??},\ ;msg_in_buf = 1 byte
data_buf = 512{??},\
R_data_buf = 512{??},\ ; read data buffer
W_data_buf = 512{0xaa},\ ; write data buffer
W_cmd_buf = {0x0A, 0x00, 0x00, 0x00, 0x01, 0x00}, \
R_cmd_buf= {0x08, 0x00, 0x00, 0x00, 0x01, 0x00}, \
dum_buf = 512{??},\
scsi_id = ID{??},\
select_id = ID{0x33, 0x00, 0x00, 0x00}
```

Description

Table indirect addressing allows a SCRIPTS program to be placed in ROM and still allows the driver program to dynamically specify different parameters for the BLOCK MOVE, SELECT, or RESELECT instructions.

Notes

The TABLE keyword defines the table entries, each of which can be accessed by a descriptive name, but no storage is allocated. It is up to the programmer to provide the data definition and allocation for the SCRIPTS table in the driver program and load the DSA prior to execution of SCRIPTS routines.

Currently, only one TABLE keyword per SCRIPTS routine is allowed. An error message will be generated if multiple TABLEs are used.

The ID parameter in the data specifier allows initialization of the table entries for use with the FROM keyword of the SELECT and RESELECT instructions on the LSI53C8XX family chips.

4.7 Conditional Keywords

Conditional keywords test for conditions such as an expected phase or data byte.

4.7.1 IF

The IF keyword indicates that a comparison is to be done immediately. Usage:

```
JUMP address, IF phase
```

4.7.2 WHEN

The WHEN keyword causes the chip, as an initiator, to wait for a phase to become valid. A valid phase is indicated by REQ/ being asserted on the SCSI bus. Since WHEN waits for the SCSI REQ/ signal when making a comparison, it may not work when comparing for conditions that are not related to the SCSI bus. Usage:

```
CALL address, WHEN data
```

4.8 Logical Keywords

Logical keywords are used in conjunction with conditional keywords to add detail or additional comparisons to the conditions being tested.

4.8.1 NOT

NOT negates (logically inverts) the conditions specified by the qualifiers that follow. For example, an instruction that reads `RETURN if NOT data` compares data to the contents of the SFBR register. If they are not identical, the operation executes. Usage:

```
JUMP address, if NOT data
```

4.8.2 AND

AND is used to compound the condition being tested. All conditions that are added with the AND keyword must be true for the operation to execute. Usage:

```
RETURN, WHEN data AND MASK DATA
```

4.8.3 OR

OR specifies a list of conditions, one of which must be true for the operation to execute. Usage:

```
CALL REL (address), IF NOT ATN OR data
```

4.9 Flag Fields

The Flag Fields keywords signify that a flag field bit has been set. The flag field bits are controlled with the SET and CLEAR instructions.

4.9.1 ACK

The target checks to see if the SCSI ACK/ signal is asserted. Usage:

```
CLEAR ACK
```

4.9.2 ATN

The target checks to see if the initiator has set the SCSI ATN/ signal. Usage:

```
JUMP address, IF NOT ATN
```

4.9.3 TARGET

By setting or clearing this bit, the SCRIPTS processor is placed in the target or the initiator role. This must be done before the chip can execute target or initiator specific operations, such as reselection. Usage:

```
SET TARGET
```

4.9.4 CARRY

This keyword checks the ALU Carry bit in the SCRIPTS processor to determine which SCRIPTS routine to execute next. CARRY is not valid if phase or data clauses are used in the same instruction. Register Move (arithmetic) operations also affect the CARRY flag. Usage:

```
JUMP address, IF CARRY
```

4.10 Qualifier Keywords

Qualifier keywords are used in conjunction with action keywords to add details about the instructions to be performed.

4.10.1 DSAREL

This keyword is only available in the LSI Logic devices that support Load and Store instructions. It is used in Load/Store instructions to indicate that the data to be loaded or stored is relative to the DSA register. This keyword replaces the RELATIVE keyword, although NASM still supports RELATIVE as well. Usage:

```
STORE NOFLUSH SCRATCHA0, 4 DSAREL (address)
```

4.10.2 FROM

This signifies use of table indirect addressing. It can be used with Block Move or Select operations. Usage:

```
MOVE FROM address, WITH phase
```

4.10.3 MASK

This keyword allows selective comparison of specified bits with the SFBR register. Any bits that are set in the mask byte eliminate the

corresponding bits in the SFBR register. Usage:
```
RETURN WHEN data AND MASK DATA
```

4.10.4 MEMORY

The MEMORY keyword is used in conjunction with an action keyword to signify a Memory-to-Memory Move instruction. Usage:
```
MOVE MEMORY 512, data_buf, data_buf1
```

4.10.5 PTR

PTR causes the Indirect bit to be set in a Block Move instruction. Usage:
```
MOVE count, PTR address, WITH phase
```

4.10.6 REG

This keyword allows access to a register by register number instead of register name. The register number must be in parenthesis. Usage:
```
MOVE REG(10) + 0x01 TO REG(10)
```

4.10.7 REL

This keyword indicates that relative addressing is used. Usage:
```
SELECT ID, REL(address)
```

4.10.8 TO

This keyword indicates the destination of a Register Move operation. Usage:
```
MOVE data TO register
```

4.10.9 WITH

The WITH keyword allows the target to drive the phase on the SCSI bus. This keyword is used for Target Move operations. Usage:
```
CHMOV count, address, WITH phase
```

4.10.10 NOFLUSH

This keyword is used in the LSI53C8XX family products that support instruction prefetching. It is used in conjunction with Move Memory and Store instructions that affect the prefetch buffer. Its purpose is to

preserve the contents of the prefetch buffer when one of these operations is performed. Usage:

```
STORE NOFLUSH SCRATCHA0, 4 DSAREL (address)
```

4.11 Other Keywords

This section of the manual describes NASM keywords.

4.11.1 Action Keywords

These words execute SCSI SCRIPTS instructions. They are described in detail in Chapter 3, “The SCSI SCRIPTS Processor Instruction Set.”

4.11.2 SCSI Phases

These words describe the phases of the SCSI bus. One of these keywords should be used in place of the word “phase” when it appears in programming examples in this guide. The SCSI phase keywords are CMD, COMMAND, DATA_IN, DATA_OUT, MSG_IN, MSG_OUT, STATUS, RES4, RES5 for all chips except the LSI53C10XX. The SCSI phase keywords for these chips are CMD, COMMAND, ST_DATA_IN, ST_DATA_OUT, MSG_IN, MSG_OUT, STATUS, DT_DATA_IN, DT_DATA_OUT. The phases are described in more detail in Chapter 3, “The SCSI SCRIPTS Processor Instruction Set.”

4.11.3 Register Names

All register names are reserved keywords.

Chapter 5
The NASM Output File

This chapter describes the output from NASM assembler and contains the following sections:

5.1 NASM Output Overview

The NASM assembler produces an output file containing all the necessary data structures and information needed by a programmer writing a driver program to load and run a SCSI SCRIPTS program. The assembler produces data structures compatible with ANSI “C”. The structures are included in a “C” program and compiled without any modifications.

Three command line parameters determine whether certain structures will be produced in the output file. The `-o` option allows NASM to generate all of the structures described in this chapter. The `-p` option allows generation of only some of the structures; please refer to the documentation for each section to see effects of the options. Finally, the `-u` option only affects the Termination Record which is detailed later in this chapter. The `-o` and `-p` options should not be used together. If they are used together in the command line, the `-p` option takes precedence. The `-u` option must be used in conjunction with either the `-o` or the `-p` option.

The example SCRIPTS program in Figure 5.1 demonstrates the various types of structures produced by the NASM assembler.

Figure 5.1 Sample SCRIPTS Program

```
ARCH 825
ABSOLUTE Got_Selected = 0xA5
ABSOLUTE Not_Msg_Out = 0x11
ABSOLUTE Select_ID = 2
ABSOLUTE Command_Complete = 0x01
EXTERN ex_buf1
EXTERN ex_buf2
RELATIVE rel_buffer \
        rel_buf1 = ??, \
        rel_buf2 = 6{??}, \
        rel_buf3 = ??
TABLE tbl_buffer \
        tbl_buf1 = ??, \
        tbl_buf2 = ??, \
        tbl_buf3 = ??
ENTRY Start
ENTRY Send_CMD
ENTRY Send_DATA
Start:
        SELECT ATN Select_ID, REL(Interrupt)
        INT Not_Msg_Out, WHEN NOT MSG_OUT
        MOVE 1, rel_buf1, WHEN MSG_OUT
Send_CMD:
        MOVE 6, rel_buf2, WHEN CMD
Send_DATA:
        MOVE FROM tbl_buf1, WHEN DATA_OUT
        MOVE FROM tbl_buf2, WHEN DATA_OUT
        MOVE FROM tbl_buf3, WHEN DATA_OUT
        MOVE 1, ex_buf1, WHEN STATUS
        MOVE 1, ex_buf1, WHEN MSG_IN
        MOVE SCNTL2 & 0x7F to SCNTL2
        CLEAR ACK
        WAIT DISCONNECT
        JUMP All_done
Interrupt:
        INT Got_Selected
All_done:
        INT Command_Complete
```

5.2 NASM Output File Examples

The code segments of the `.out` file discussed in this section correspond to the example SCRIPTS program in Figure 5.1.

5.2.1 SCRIPTS Array

The SCRIPTS array is an array of unsigned long values that is the actual contiguous machine code (opcodes) produced by the assembler. Each line of the array contains one instruction and one or two address fields, depending on the instruction. If a PROC directive is used in the source program, there may be more than one SCRIPTS array. For each PROC, a new array is declared with the name specified with the PROC directive.

For example, if the above code started with:

```
PROC SCSI_READ:
Start:
        SELECT ATN Select_ID, REL(Interrupt)
                        .
                        .
                        .
```

Then the SCRIPTS array starts:

```
typedef unsigned long ULONG;
ULONG SCSI_READ[] = {
   0x45020000L, 0x00000060L,
```

The default array name without the PROC statement is SCRIPT. The SCRIPTS array is not affected by NASM command line options.

Example of SCRIPTS array:

```
typedef unsigned long ULONG;
ULONG SCRIPT[] = {
   0x45020000L,0x00000060L,
   0x9E030000L,0x00000011L,
   0x0E000001L,0x00000000L,
   0x0A000006L,0x00000001L,
   0x18000000L,0x00000000L,
   0x18000000L,0x00000008L,
   0x18000000L,0x00000010L,
   0x0B000001L,0x00000000L,
   0x0F000001L,0x00000000L,
   0x7C027F00L,0x00000000L,
   0x60000040L,0x00000000L,
   0x48000000L,0x00000000L,
   0x80080000L,0x00000070L,
   0x98080000L,0x000000A5L,
   0x98080000L,0x00000001L
```

A PROC label generates separate arrays of SCRIPTS instructions for each PROC occurrence. An Entry specification generates a “C” language #define equal to the number of bytes between this entry and the beginning of the first code array. The #define offset is not relative to the array in which it appears, but is relative to the first code array created. In the example shown in Table 5.1, the first SCRIPTS instruction for INC_A is located 40 (hex) bytes after the location of MAIN[ ].

Table 5.1 Relationship Between Entry and PROC Statements and Output File

Source	Output File
	typedef unsigned long ULONG;
Entry MAIN	#define ENT_MAIN 0x00000000L
Entry CLEAR_A	#define ENT_CLEAR_A 0x00000018L
Entry INC_A	#define ENT_INC_A 0x00000040L
PROC MAIN:	ULONG MAIN[] = {
call CLEAR_A	0x88080000L, 0x00000018L,
call INC_A	0x88080000L, 0x00000040L,
call INC_A	0x88080000L, 0x00000040L,
	};
PROC CLEAR_A:	ULONG CLEAR_A[] = {
move SCRATCHA0 & 00 to SCRATCHA0	0x7C340000L, 0x00000000L,
move SCRATCHA1 & 00 to SCRATCHA1	0x7C350000L, 0x00000000L,
move SCRATCHA2 & 00 to SCRATCHA2	0x7C360000L, 0x00000000L,
move SCRATCHA3 & 00 to SCRATCHA3	0x7C370000L, 0x00000000L,
return;	0x90080000L, 0x00000000L
INC_A:	
move SCRATCHA0 + 1 to SCRATCHA0	0x7E340100L, 0x00000000L,
return, if NOT Carry;	0x90200000L, 0x00000000L,
move SCRATCHA1 + 1 to SCRATCHA1	0x7E350100L, 0x00000000L,
return, if NOT Carry;	0x90200000L, 0x00000000L,
move SCRATCHA2 + 1 to SCRATCHA2	0x7E360100L, 0x00000000L,
return, if NOT Carry;	0x90200000L, 0x00000000L,
move SCRATCHA3 + 1 to SCRATCHA3	0x7E370100L, 0x00000000L,
return;	0x90080000L, 0x00000000L
	};

5.2.2 External

The External section contains the external variable records, if any were declared. First, is the External Header Record which contains:

```
#define Ext_count count
```

Where count is defined as number of external variables. Second is a character array of all external names used:

```
char *External_Names[Ext_Count] = {
    “dsa_storage”,
    “in_offset”,
    “out_offset”
    };
```

Third is a list of External Contents Records:

```
#define E_name offset
```

Where name is the name of the variable and offset is defined as the byte offset from the beginning of the data area. It is always zero for externals.

Following this is an array of unsigned longs named by appending “_Used” to the variable name. This array is a list of Dword offsets from the beginning of the SCRIPTS array where the variable is used and should be patched.

```
#define E_name_Used offset
```

The last two sections (External Contents Record and Offset Array) of the External record are repeated for every External defined in the SCRIPT.

The output depends on which command line switches are selected. If the -o compiler option is used then all items mentioned above are included in the output file. If the -p (partial ‘C’ output) option is used then the External Header Record and Character Array are omitted from the output file. An example of the output generated using each compiler option is listed below.

Example using -o assembler option:

```
#define Ext_Count  2
char *External_Names[Ext_Count] = {
    “ex_buf2”,
    “ex_buf1”
};
#define E_ex_buf1 0x00000000L
ULONG E_ex_buf1_Used[] = {
    0x0000000FL,
```

```
    0x00000011L
};
```

Using -p assembler option:

```
ULONG E_ex_buf1_Used[] = {
    0x0000000FL,
    0x00000011L
};
```

5.2.3 Relative

The Relative section contains the relative buffer records, if any were declared. The first part is the Relative Header Record, which contains:

```
#define Rel_Count count
```

Where count is a total count of all the uses of all the Relative buffers in the SCRIPTS program. For example, in the SCRIPTS example above, rel_buf1 and rel_buf2 are each used once so Rel_Count is #defined to 2, indicating that there were two uses of Relative buffers in the SCRIPTS code.

The second part of the Relative record is the Relative Patch Array which contains:

```
ULONG Rel_Patches[Rel_Count] = {
    Rel_Offset1,
    Rel_Offset2,
    Rel_Offset3,
    .
    .
    Rel_Offsetn
};
```

Where Rel_Offsetn is an offset into the SCRIPTS array where a Relative buffer is used. This array, along with the Relative Header Record, can be used to patch all Relative buffers in a SCRIPTS program. Please see the subsection entitled “Patching,” on page 7-7 for more information.

The third part of the Relative record is the Relative Buffer Record, which contains:

```
#define R_name offset
```

Where name is the name of the Relative buffer, for example rel_buf1, and offset is the relative offset of this buffer from the beginning of the entire Relative buffer. For example, in the above SCRIPTS example rel_buf2 has an offset of 0x00000001L, indicating that it starts one byte from the beginning of the Relative buffer.

The final part of the Relative record is the offset array which lists the Dword offsets in the SCRIPTS array where each individual relative buffer is used. It is the same as the offset array used for External buffers, except that the array names are of the format R_name_Used where name is the name of the individual relative buffer.

```
#define R_name_Used offset
```

The last two sections of the Relative record, Relative Buffer Record and Offset Array, are repeated for every Relative defined in the SCRIPTS program.

Command line switches also effect Relative. Using the -o compiler option includes all items mentioned above in the output file. Using the -p, partial 'C' output option, omits the Relative Header Record and Relative Patch Array from the output file. An example of the output generated using each compiler option is listed below.

Example using -o assembler option:

```
#define Rel_Count 2
ULONG Rel_Patches[Rel_Count] = {
    0x00000007L,
    0x00000005L
};
#define R_rel_buf1 0x00000000L
ULONG R_rel_buf1_Used[] = {
    0x00000005L
};

#define R_rel_buf2 0x00000001L
ULONG R_rel_buf2_Used[] = {
    0x00000007L
};
```

Using -p assembler option:

```
#define R_rel_buf1 0x00000000L
ULONG R_rel_buf1_Used[] = {
    0x00000005L
```

```
};
#define R_rel_buf2 0x00000001L
ULONG R_rel_buf2_Used[] = {
    0x00000007L
};
```

5.2.4 Entry

The ENTRY section contains the entry records, if any were declared. An entry record is a `#define` of the entry name prefixed with `Ent_`, defined to be a byte offset into the SCRIPTS array.

Example:

Using `-o` or `-p` assembler option:

```
#define Ent_Send_CMD          0x00000018L
#define Ent_Send_DATA         0x00000020L
#define Ent_Start             0x00000000L
```

The labels defined as entries are the only ones available to the driver code. The “C” code examples in Figure 5.1 are examples of how the driver uses this information to start SCRIPTS routines at any location defined as an entry. The ENTRY section is not affected by NASM command line options.

5.2.5 Label Patches

The Label Patches section contains the label patch records. A label patch record is an array of locations that are referred to by an absolute Transfer Control instruction. These locations are the Dword offsets into the SCRIPTS array. The offsets patch in the physical addresses at run time. Please see the section on patching SCRIPTS in Chapter 7, “Integrating SCRIPTS Programs into “C” Language Drivers,” for more information on how to patch absolute jump instructions. The Label Patches section is not affected by NASM command line options.

Example:

Using `-o` or `-p` assembler option:

```
ULONG LABELPATCHES[] = {
    0x00000019L
                          };
```

5.2.6 Absolute

The Absolute section contains the Absolute records, if any were declared. First is the Absolute Header Record, which contains:

```
#define Abs_Count count
```

Where count is the number of Absolutes defined in the SCRIPTS program.

The second section is the Character Array of all Absolute names used, it contains:

```
char *Absolute_Names[Abs_Count] = {
    Abs_String1,
    Abs_String2,
    .
    .
    Abs_Stringn
};
```

Where Abs_Stringn is the name of the Absolute being defined.

Third is the Absolute Value Definition, which contains:

```
#define A_name value
```

Where name is the name of the Absolute and value is the value assigned to this Absolute in the SCRIPTS program.

The final part of the Absolute record is the Offset Array, which lists the offsets in the SCRIPTS array where each Absolute is used. It is the same as the offset array used for External buffers, except that the array names are of the format A_name_Used where name is the name of the Absolute.

The last two sections of the Absolute record, Absolute Value Definition and Offset Array, are repeated for every Absolute defined in the SCRIPTS program.

Command line switches also effect the output of absolute. Using the -o compiler option includes all items mentioned above in the output file. Using the -p option omits the Offset Array from the output file. An example of the output generated using each compiler option is listed below.

Example using -o assembler option:

```
#define Abs_Count 4
char *Absolute_Names[Abs_Count] = {
    “Command_Complete”,
    “Got_Selected”,
    “Not_Msg_Out”,
    “Select_ID”
};
#define A_Command_Complete 0x00000001L
ULONG A_Command_Complete_Used[] = {
    0x0000001DL
};
#define A_Select_ID 0x00000002L
ULONG A_Select_ID_Used[] = {
    0x00000000L
};

#define A_Not_Msg_Out 0x00000011L
ULONG A_Not_Msg_Out_Used[] = {
    0x00000003L
};
#define A_Got_Selected 0x000000A5L
ULONG A_Got_Selected_Used[] = {
    0x0000001BL
};
```

Using -p assembler option:

```
#define A_Command_Complete 0x00000001L
#define A_Select_ID 0x00000002L
#define A_Not_Msg_Out 0x00000011L
#define A_Got_Selected 0x000000A5L
```

5.2.7 Termination Record

The module termination record declares two variables, INSTRUCTIONS and PATCHES. INSTRUCTIONS is assigned the number of instructions found in the SCRIPTS program, and PATCHES is assigned the number of label patches. Using the -o compiler option includes all items mentioned above in the output file. Using the -p option omits the Patches variable from the output file. The -u option, exclude module termination record, omits both variables from the output file. An example of the output generated using each compiler option is listed below.

Example using -o assembler option:

```
ULONG INSTRUCTIONS = 0x0000000EL;
```

```
ULONG PATCHES     = 0x00000000L;
```

Using -p assembler option:

```
ULONG INSTRUCTIONS = 0x0000000EL;
```

Chapter 6
Using the Registers to Control Chip Operations

This chapter contains the following sections:

6.1 Overview

The SCRIPTS processor is initialized by setting and clearing bits in the operating registers. This chapter lists the various registers used by the LSI53C7XX/8XX/10XX family chips, grouped by function. The register descriptions provide an overview of the aspects of chip operation that are controlled in each register. The SCRIPTS processor also has a set of PCI Configuration registers, but they are not described in this document since they are initialized by the system, not by the SCSI driver program. Full definitions of these registers, as well as the individual bits in the operating registers, can be found in the chip technical manuals.

6.2 SCSI Registers

Table 6.1 lists the SCSI registers. The SCSI registers are used for the following functions:

- Performing SCSI operations by low level, register-oriented programming.
- Obtaining data for debugging, such as checking the signal status of the SBCL (SCSI Bus Control Lines) and SBDL (SCSI Bus Data Lines) registers to determine exactly what is on the SCSI bus at the time the registers are read.
- Obtaining SCSI interrupt status, which is contained in the SIST0 (SCSI Interrupt Status 0), and SIST1 (SCSI Interrupt Status 1) registers.
- Initialization of the SCSI interface, for example, parity generation and checking on the SCSI bus.
- Enabling or masking SCSI interrupts in the SIEN (SCSI Interrupt Enable) registers.

Table 6.1 SCSI Registers

Name	Definition	Functions
SWIDE[1]	SCSI Wide Residue Data	Contains a residual data byte that was never sent across the DMA bus after wide SCSI operation.
AIPCNTL(0, 1)[2]	Arbitration in Progress Control	These registers control and reflect the status of arbitration in process sequence, values, and errors.
ISTAT1[3]	Interrupt Status 1	Flushing the DMA FIFO; SCRIPTS engine operating; IRQ pin disable.
MBOX (0, 1)[3]	Mailbox	General purpose registers.
RESPID0	Response ID 0	Contains IDs the chip will respond to when it is selected or reselected.
RESPID1[1]	Response ID 1	Contains IDs the chip will respond to when it is selected or reselected.
SBCL	SCSI Bus Control Lines	Used to return SCSI control line status.
SBDL	SCSI Bus Data Lines	Contains SCSI data bus status.

Table 6.1 SCSI Registers (Cont.)

Name	Definition	Functions
SCID	SCSI Chip ID	Enable response to selection/reselection, set SCSI ID for chip.
SCNTL0	SCSI Control 0	Arbitration Mode bits; enable parity checking.
SCNTL1	SCSI Control 1	Add an extra clock cycle of setup to each SCSI data transfer; disable halt on parity error; Connected bit; parity bits; Immediate Arbitration bit.
SCNTL2	SCSI Control 2	Wide SCSI control bits, vendor unique enhancements; DIFFSENS mismatch indicator (LSI53C895 only).
SCNTL3	SCSI Control 3	Clock conversion factor bits, enable wide SCSI, enable Ultra SCSI or Ultra2 SCSI.
SCNTL4[2]	SCSI Control 4	This register is used during Table Indirect Select or Reselect SCRIPTS instructions.
SDID	SCSI Destination ID	Encoded destination SCSI ID.
SFBR	SCSI First Byte Received	Contains the first byte received in any asynchronous information transfer phase.
SIDL	SCSI Input Data Latch	Contains latched data from the SCSI bus.
SIEN0	SCSI Interrupt Enable 0	Interrupt mask bits for phase mismatch, SATN/, function complete, selection/reselection, gross error, unexpected disconnect, SCSI reset, parity error.
SIEN1	SCSI Interrupt Enable 1	Interrupt mask bits for selection/reselection time out, general purpose time-out, handshake-to-handshake time-out.
SLPAR	SCSI Longitudinal Parity	Performs a bytewise longitudinal parity check on all SCSI data.
SOCL	SCSI Output Control Latch	Testing SCSI control lines.
SODL	SCSI Output Data Latch	Data flows through this register when sending data in any mode.
SSID	SCSI Selector ID	The ID of the device that selected or reselected the chip.
SSTAT0	SCSI Status 0	SIDL, SODR, SODL least significant byte full; arbitration reporting bits; status of RST/ and SDP0/ signals.

Table 6.1 SCSI Registers (Cont.)

Name	Definition	Functions
SSTAT1	SCSI Status 1	FIFO flags; latched SCSI parity signal; latched SCSI phase status bits.
SSTAT2	SCSI Status 2	Reports SIDL, SODR, SODL most significant byte full; parity detection, disconnect detection.
STEST0	SCSI Test 0	These bits are used for low level operation and manufacturing testing, SCSI selected as ID.
STEST1	SCSI Test 1	Disable the external SCLK pin and use the PCI clock as the internal SCSI clock; enable the SCSI Clock doubler (LSI53C825A/875/876/885 only) or SCSI clock quadupler (LSI53C895/896/10XX only).
STEST2	SCSI Test 2	Clear synchronous offset; Enable Differential Mode; wide SCSI; extend SREQ/–SACK/ filtering; Low Level Mode enable.
STEST3	SCSI Test 3	Active negation enable; SCSI FIFO test read/write; Halt SCSI clock; Clear SCSI FIFO.
STEST4[4]	SCSI Test 4	Contains DIFFSENS pin values that indicate the type of SCSI device connected to the bus; frequency lock bit for clock quadrupler.
STIME0	SCSI Timer 0	Selects the handshake-to-handshake time-out period.
STIME1	SCSI Timer 1	Selects the general purpose time-out period.
SXFER	SCSI Transfer	Define synchronous transfer period and synchronous offset.

1. Wide SCSI products only.
2. LSI53C10XX only.
3. LSI53C895 and later.
4. LSI53C895A and later only.

6.3 DMA Registers

Table 6.2 lists the DMA registers. The DMA registers are used for the following functions:

- Setting up the host interface.

- Obtaining DMA interrupt status information contained in the DMA Status (DSTAT) register.
- Obtaining DMA FIFO information, such as the number of bytes it contains.
- Enabling or masking DMA interrupts with the DMA Interrupt Enable (DIEN) registers.

Table 6.2 DMA Registers

Name	Definition	Functions
DBC	DMA Byte Counter	Determines the number of bytes to be transferred in a Block Move instruction.
DCMD	DMA Command	Identifies the instruction that the chip will execute.
DCNTL	DMA Control	Enables the Single Step Mode; LSI53C700 compatibility bit; enables the PCI Cache Line Size register; enables instruction prefetching.
DFIFO	DMA FIFO	May be used to determine the number of bytes in the DMA FIFO when an interrupt occurs, when used in conjunction with DBC.
DIEN	DMA Interrupt Enable	Contains interrupt mask bits corresponding to master data parity error, bus fault, aborted, single step interrupt, SCRIPT interrupt instruction received, illegal instruction detected
DMODE	DMA Mode	Defines burst length; near or far memory access; enables PCI read line command; Manual Start Mode bit to prevent automatic execution of SCRIPTS.
DNAD	DMA Next Address	Contains the general purpose address pointer.
DSP	DMA SCRIPTS Pointer	Contains the address of the next SCRIPTS instruction to be fetched. Placing an address in this register starts SCRIPTS.
DSPS	DMA SCRIPTS Pointer Save	Contains the second Dword of a SCRIPTS instruction.
TEMP	Temporary Register	Stores pointer to the next SCRIPTS instruction to be executed when returning from a subroutine.

6.4 SCRIPTS Registers

The SCRIPTS registers hold the SCRIPTS instruction information which is fetched from host memory at run time by the SCRIPTS processor. MBOX registers are also used as SCRIPTS registers. The SCRIPTS

registers are listed in Table 6.3. They are described in Section 6.6, "Interrupt Registers."

Table 6.3 SCRIPTS Registers

Name	Definition	Functions
DBC	DMA Byte Counter	Determines the number of bytes to be transferred in a Block Move instruction.
DCMD	DMA Command	Identifies the instruction that the SCRIPTS processor will execute.
DNAD	DMA Next Address	Contains the general purpose address pointer.
DSA	Data Structure Address	Contains base address used for all table indirect calculations.
DSP	DMA SCRIPTS Pointer	Contains the address of the next SCRIPTS instruction to be fetched; placing an address in this register starts SCRIPTS.
DSPS	DMA SCRIPTS Pointer Save	Contains the second Dword of a SCRIPTS instruction.

6.5 64-Bit SCRIPTS Selector Registers

Table 6.4 lists the 64-bit Selector registers. The 64-bit Selector registers reflect/control various aspects of 64-bit operation.

Table 6.4 64-Bit Selector Registers

Name	Definition	Functions
CCNTL0[1]	Chip Control 0	Various JUMP control functions, Disable Auto-FIFO Clear, Disable Internal Load/Store (LSI53C89X only), Disable Internal SCRIPTS RAM Cycles (LSI53C10XX only), Disable Pipe Request
CCNTL1[1]	Chip Control 1	Disable DAC, 64-bit Table Indirect Indexing Mode, Enable 64-bit Table Indirect BMOV, Enable 64-bit Direct BMOV LSI53C89X only: High Impedance Mode LSI53C10XX only: Pull Enable, Pull Disable, Disable 64-bit Master Operation, Disable 64-bit Slave Cycles
CCNTL2[2]	Chip Control 2	Reserved

Table 6.4 64-Bit Selector Registers (Cont.)

Name	Definition	Functions
CCNTL3[2]	Chip Control 3	Skew Control, LVD Drive Strength Control.
MMRS[1]	Memory Move Read Selector	Supplies AD[63:32] during data read operations for Memory-to-Memory Move and absolute address LOAD operations.
MMWS[1]	Memory Move Write Selector	Supplies AD[63:32] during data write operations during Memory-to-Memory Moves and absolute address STORE operations.

1. LSI53C895 and later only.
2. LSI53C10XX only.

6.6 Interrupt Registers

Table 6.5 lists the Interrupt registers. Interrupt registers contain interrupt status information. The DSTAT contains the DMA interrupt status information. The SIST0 and SIST1 contain SCSI interrupt status bits. The remaining registers contain interrupt enable bits. The ISTAT register can be polled for interrupts. It is the only register that can be accessed while SCRIPTS is running. Refer to Chapter 9, "SCRIPTS Programming Topics," for more information on handling interrupts.

Table 6.5 Interrupt Registers

Name	Definition	Functions
CSO[1]	Current Inbound SCSI Offset	Indicates current SCSI offset.
DIEN	DMA Interrupt Enable	Contains interrupt mask bits corresponding to master data parity error, bus fault, aborted operation, single step interrupt, SCRIPTS interrupt instruction received, illegal instruction detected.
DSTAT	DMA Status	Reports sources of DMA interrupts: DMA FIFO empty, Master data parity error, bus fault, aborted, single step interrupt, SCRIPTS interrupt instruction received, illegal instruction detected.
ISTAT[2]	Interrupt Status	Interrupt polling; determines whether a SCSI or DMA interrupt has occurred; checks for stacked interrupts; aborts an operation; software reset; signal process bit; semaphore bit; interrupt on the fly bit; indicate SCSI interrupt pending (LSI53C885 only).

Table 6.5 Interrupt Registers (Cont.)

Name	Definition	Functions
ISTAT0[3]	Interrupt Status 0	Abort operation; software reset; semaphore bit; signal process bit; determines whether a SCSI or DMA interrupt is pending; SCSI connection.
ISTAT1[3]	Interrupt Status 1	Flushing the DMA FIFO; SCRIPTS engine operating; IRQ pin disable.
MBOX (0, 1)[3]	Mailbox	General purpose registers.
SIEN0	SCSI Interrupt Enable 0	Interrupt mask bits for phase mismatch, SATN/, function complete, selection/reselection, gross error, unexpected disconnect, SCSI reset, parity error.
SIEN1	SCSI Interrupt Enable 1	Interrupt mask bits for selection/reselection time-out, general purpose time-out, handshake-to-handshake time-out; wakeup (LSI53C885 only); SCSI bus mode change (LSI53C895/896/10XX only).
SIST0	SCSI Interrupt Status 0	Returns the status of the following interrupt conditions: phase mismatch (SATN/ active), function complete, selection/reselection, SCSI gross error, unexpected disconnect, SCSI RST/ received, parity error.
SIST1	SCSI Interrupt Status 1	Returns the status of the following interrupt conditions: selection/reselection time-out, general purpose timer expired, handshake-to-handshake timer expired; wakeup (LSI53C885 only).

1. LSI53C10XX only
2. Up to LSI53C895 only.
3. LSI53C895A and later only.

6.7 Phase Mismatch Registers

The Phase Mismatch registers contain information generated during BMOV instructions, particularly those executing during a phase mismatch. The Phase Mismatch registers are listed in Table 6.6. Unless otherwise noted, these registers are only on LSI53C895 and later chips.

Table 6.6 Phase Mismatch Registers

Name	Definition	Functions
CCNTL0	Chip Control 0	Various JUMP control functions, Disable Auto-FIFO Clear, Disable Internal Load/Store (LSI53C89X only), Disable Internal SCRIPTS RAM Cycles, Disable Pipe Request (LSI53C10XX only).
CSBC	Cumulative SCSI Byte Count	Cumulative byte count of data transferred across the SCSI bus during data phases.
ESA	Entry Storage Address	Contains BMOV instruction address information.
IA	Instruction Address	Contains the address of the BMOV instruction that was executing at the time of a phase mismatch.
PMJAD (1, 2)[1]	Phase Mismatch Jump Address	Contains the address that will be jumped to in the case of a phase mismatch. PMJAD is outbound, PMJAD2 is inbound.
RBC	Remaining Byte Count	Byte count that remains for the BMOV instruction that was executing at the time of a phase mismatch.
SBC	SCSI Byte Count	Number of bytes transferred to or from the SCSI bus during any given BMOV.
UA	Updated Address	Contains the updated data address of the BMOV that was executing at the time of a phase mismatch.

1. LSI53C10XX only.

6.8 Test and Miscellaneous Registers

The test registers are used to test the DMA and SCSI FIFOs and perform other miscellaneous functions. The test registers are listed in Table 6.7. Test registers can be used to decrement the byte count or increment the address count in the FIFOs.

Table 6.7 Test and Miscellaneous Registers

Name	Definition	Functions
ADDER	Adder Sum Output	Contains output of internal adder.
CCNTL0[1]	Chip Control 0	Various JUMP control functions, Disable Auto FIFO Clear, Disable Internal Load/Store (LSI53C89X only), Disable Internal SCRIPTS RAM Cycles (LSI53C10XX only), Disable Pipe Request
CCNTL1[1]	Chip Control 1	Disable DAC, 64-Bit Table Indirect Indexing Mode, Enable 64-Bit Table Indirect BMOV, Enable 64-Bit Direct BMOV LSI53C89X only: High Impedance Mode LSI53C10XX only: Pull Enable, Pull Disable, Disable 64-Bit Master Operation, Disable 64-Bit Slave Cycles.
CCNTL2[2]	Chip Control 2	Reserved.
CCNTL3[2]	Chip Control 3	Skew Control, LVD Drive Strength Control.
CTEST0	Chip Test 0	Used to enable power management modes in the LSI53C885.
CTEST1	Chip Test 1	DMA FIFO bits full or empty.
CTEST2	Chip Test 2	Data transfer direction; I/O or memory configuration; request/acknowledge status.
CTEST3	Chip Test 3	Revision level bits, flush/clear DMA FIFO.
CTEST4	Chip Test 4	Burst disable; master parity error enable; DMA FIFO byte control.
CTEST5	Chip Test 5	Clock address incrementor; clock byte counter; DMA direction; control of set or reset pulses.
CTEST6	Chip Test 6	Writes data to the DMA FIFO.

1. LSI53C895 and later only.
2. LSI53C10XX only.

6.9 General Purpose Registers

Table 6.8 describes SCRIPTS processor general purpose registers.

Table 6.8 General Purpose Registers

Name	Definition
CTEST0	Chip Test 0
DWT/SBR	DMA Watchdog Timer/Scratch Byte Register
GPCNTL	General Purpose Control
GPREG	General Purpose
MACNTL	Memory Access Control
SCRATCHA	General Purpose Scratchpad A
SCRATCHB	General Purpose Scratchpad B
SCRATCHC–J[1]	General Purpose Scratchpad C–J
SCRATCHC–R[2]	General Purpose Scratchpad C–R

1. LSI53C825A/875/876/885 only.
2. LSI53C895/895A/896/1000/1010/1010R/1000R only.

6.10 Register Initialization

The startup register values are determined by a “C” program, written by the software developer, that can be loaded automatically by the device driver. The appropriate startup values for the register bits depend on the design of the individual system. Therefore, a single startup algorithm will not support every application. The hardware default values for each bit are provided in the appropriate chip technical manuals.

Table 6.9 and Table 6.10 list the register bits you should consider when writing a startup program for a specific system. The startup program does not have to initialize all bits in the chip if the default values are acceptable. However, the bits in these lists affect features that should be enabled or disabled and other decisions that should be made when initializing the chip. For complete register and bit descriptions, refer to your chip technical manual. In addition, Chapter 2, “Programming with

SCRIPTS," contains a section on the bits and registers that affect parity checking and generation. All reserved bits should be left cleared by the startup program.

Table 6.9 LSI53C815/810A/860 Startup Bits

Register Address	Register Name	Bits	Remarks
0x00	SCNTL0	[7:6], 3, [1:0]	Bits [7:6]: Arbitration Mode Bit 3: Enable Parity Checking Bit 1: Assert SATN/ on Parity Error Bit 0: Target Mode. Bit 0 can be set either at initialization or during SCRIPTS operation. Set it at startup if the chip operates as a target only. If it switches between Target and Initiator Modes, use SCRIPTS to control this bit.
0x01	SCNTL1	7, 5	Bit 7: Extra Clock Cycle of Data Setup Bit 5: Disable Halt on Parity Error or SATN/ (for Target Mode only)
0x03	SCNTL3	7, [6:4], [2:0]	Bit 7: Ultra Enable (LSI53C860 only) Bits [6:4]: Synchronous Clock Conversion Factor Bits [2:0]: Clock Conversion Factor
0x04	SCID	6, 5, [2:0]	Bit 6: Enable Response to Reselection Bit 5: Enable Response to Selection Bits [2:0]: Encoded Chip SCSI ID
0x05	SXFER	[7:5], [3:0]	Since the default operation for SCSI is asynchronous transfers, these bits should probably not be set until synchronous parameters are established between the initiator and target. Bits [7:5]: Synchronous Transfer Period Bits [3:0]: Max SCSI Synchronous Offset
0x10–0x13	DSA	all	Must be initialized to use Table Indirect Mode.
0x1B	CTEST3	1, 0	Bit 1: Fetch Pin Mode Bit 0: Write and Invalidate Enable (LSI53C810A/860 only)
0x21	CTEST4	7, 3	Bit 7: Burst Disable Bit 3: Master Parity Error Enable
0x2C–0x2F	DSP	all	At the end of the initialization program, write the address of the first SCRIPTS instruction to this register to begin SCRIPTS execution.

Table 6.9 LSI53C815/810A/860 Startup Bits (Cont.)

Register Address	Register Name	Bits	Remarks
0x38	DMODE	[7:6], 5, 4, 3, 2	Bits [7:6]: Burst Length Bit 5: Source I/O-Memory Enable Bit 4: Destination I/O-Memory Enable Bit 3: Enable Read Line Bit 2: Enable Read Multiple (LSI53C810A/860 only)
0x39	DIEN	6, 5, 4, 3, 2, 0	Bit 6: Master Data Parity Error Bit 5: Bus Fault Bit 4: Aborted Bit 3: Single Step Interrupt Bit 2: SCRIPTS Interrupt Instruction Received Bit 0: Illegal Instruction Detected
0x3B	DCNTL	7, 5, 4, 3, 0	Bit 7: Cache Line Size Enable Bit 5: Prefetch Enable (LSI53C810A/860 only) Bit 4: Single Step Mode Bit 3: IRQ Mode Bit 0: LSI53C700 Compatibility
0x40	SIEN0	7, 6, 5, 4, 3, 2, 1, 0	Interrupt mask bits for: Bit 7: Phase Mismatch or SATN/ Bit 6: Function Complete Bit 5: Selected Bit 4: Reselected Bit 3: SCSI Gross Error Bit 2: Unexpected Disconnect Bit 1: SCSI Reset Condition Bit 0: SCSI Parity Error
0x41	SIEN1	2, 1, 0	Interrupt mask bits for: Bit 2: Selection or Reselection Time-out Bit 1: General Purpose Timer Expired Bit 0: Handshake-to-Handshake Timer Expired
0x46	MACNTL	3, 2, 1, 0	Initialize these when using the MAC_TESTOUT pin. These bits determine local or far access for the following operations: Bit 3: Data write Bit 2: Data read Bit 1: SCRIPTS pointers Bit 0: SCRIPTS fetches
0x48	STIME0	[7:4], [3:0]	Bits [7:4]: Handshake-to-Handshake Timer Period Bits [3:0]: Selection Time-out
0x49	STIME1	3–0	Bits [3:0]: General Purpose Timer Period
0x4A	RESPID	all	N/A

Table 6.9 LSI53C815/810A/860 Startup Bits (Cont.)

Register Address	Register Name	Bits	Remarks
0x4D	STEST1	7	Bit 7: SCLK
0x4E	STEST2	1	Bit 1: Extend SREQ/SACK Filtering
0x4F	STEST3	7	Bit 7: TolerANT Enable

Table 6.10 LSI53C825A/875/876/885/895/895A/896/10XX Startup Bits

Register Address	Register Name	Bits	Remarks
0x00	SCNTL0	[7:6], 3, 1, 0	Bits [7:6]: Arbitration Mode Bit 3: Enable Parity Checking Bit 1: Assert SATN/ on Parity Error Bit 0: Target Mode. Bit 0 can be set either at initialization or during SCRIPTS operation. Set it at startup if the chip operates as a target only. If it switches between Target and Initiator Modes, use SCRIPTS to control this bit.
0x01	SCNTL1	7, 5	Bit 7: Extra Clock Cycle of Data Setup Bit 5: Disable Halt on Parity Error or SATN/ (for Target Mode only)
0x03	SCNTL3	7, [6:4], [2:0]	Bit 7: Ultra Enable (LSI53C875/876/885/895 only) Bits [6:4]: Synchronous Clock Conversion Factor Bits [2:0]: Clock Conversion Factor
0x04	SCID	6, 5, 3, [2:0]	Bit 6: Enable Response to Reselection Bit 5: Enable Response to Selection Bit 3: Enable Wide SCSI Bits [2:0]: Encoded Chip SCSI ID
0x05	SXFER	7–5, 3–0	Since the default operation for SCSI is asynchronous transfers, these bits should not be set until synchronous parameters are established between the initiator and target. Bits 7–5: Synchronous Transfer Period Bits 3–0: Max SCSI Synchronous Offset
0x10–0x13	DSA	all	Must be initialized to use Table Indirect Mode.
0x1B	CTEST3	1, 0	Bit 1: Fetch Pin Mode Bit 0: Write and Invalidate Enable

Table 6.10 LSI53C825A/875/876/885/895/895A/896/10XX Startup Bits (Cont.)

Register Address	Register Name	Bits	Remarks
0x21	CTEST4	7, 3	Bit 7: Burst Disable Bit 3: Master Parity Error Enable
0x22	CTEST2	3	SCRATCHA/B operation (when SCRIPTS RAM is enabled).
0x18	CTEST0	[2:0]	Set the priority level for gaining access to the PCI bus (LSI53C885 only).
0x2C–0x2F	DSP	all	At the end of the initialization program, write the address of the first SCRIPTS instruction to this register to begin SCRIPTS execution.
0x38	DMODE	[7:6], 5, 4, 3, 2	Bits [7:6]: Burst Length Bit 5: Source I/O-Memory Enable Bit 4: Destination I/O-Memory Enable Bit 3: Enable Read Line Bit 2: Enable Read Multiple
0x39	DIEN	4, 3, 2, 0	Bit 4: Aborted Bit 3: Single Step Interrupt Bit 2: SCRIPTS Interrupt Instruction Received Bit 0: Illegal Instruction Detected
0x3B	DCNTL	7, 5, 4, 3, 0	Bit 7: Cache Line Size Enable Bit 5: Prefetch Enable Bit 4: Single Step Mode Bit 3: IRQ Mode Bit 0: LSI53C700 Compatibility
0x40	SIEN0	7, 6, 5, 4, 3, 2, 1, 0	Interrupt mask bits for: Bit 7: Phase Mismatch or SATN/ Bit 6: Function Complete Bit 5: Selected Bit 4: Reselected Bit 3: SCSI Gross Error Bit 2: Unexpected Disconnect Bit 1: SCSI Reset Condition Bit 0: SCSI Parity Error
0x41	SIEN1	4, 2, 1, 0	Interrupt mask bits for: Bit 4: SCSI Bus Mode Change (LSI53C895 only) Bit 2: Selection or Reselection Time-out Bit 1: General Purpose Timer Expired Bit 0: Handshake-to-Handshake Timer Expired

Table 6.10 LSI53C825A/875/876/885/895/895A/896/10XX Startup Bits (Cont.)

Register Address	Register Name	Bits	Remarks
0x46	MACNTL	3, 2, 1, 0	Initialize these when using the MAC_TESTOUT pin. These bits determine local or remote access for the following operations: Bit 3: Data write Bit 2: Data read Bit 1: SCRIPTS pointers Bit 0: SCRIPTS fetch
0x48	STIME0	[7:4], [3:0]	Bits [7:4]: Handshake-to-Handshake Timer Period Bits [3:0]: Selection Time-Out
0x49	STIME1	[3:0]	Bits [3:0]: General Purpose Timer Period
0x4A	RESPID0	all	N/A
0x4B	RESPID1	all	N/A
0x4D	STEST1	7, [3:2]	Bit 7: SCLK Bits [3:2]: SCSI Clock Doubler 1–0 (LSI53C875 only)
0x4E	STEST2	5, 1	Bit 5: SCSI Differential Mode Bit 1: Extend REQ/ACK Filtering
0x4F	STEST3	7	Bit 7: TolerANT Enable
0xBC	SCNTL4	7	Bit 7: Ultra3 Transfer Enable

Chapter 7
Integrating SCRIPTS Programs into “C” Language Drivers

This chapter demonstrates how assembled SCRIPTS programs are included in SCSI device drivers written in “C” language. This chapter contains the following sections:

7.1 Initializing the SCRIPTS Processor

The “C” code in Figure 7.1 is an example that shows how the SCRIPTS processor accesses the operating registers at initialization. The processor can be memory-mapped, I/O-mapped, or mapped using both methods. The example functions in this section access I/O mapped registers.

Figure 7.1 Accessing I/O Mapped Registers

```
/*****************************************************
 Function: IORead8
   Purpose: To read a byte from an io port
   Input: IO address of byte to be read
   Output: byte read from IO port
   Assumptions: That the IO port actually exists
   Restrictions: Although IO_Addr is defined as
                 a ULONG it must not exceed 16 bits
                 in length as this is the maximum
                 IO address the X86 architecture can produce
   Other functions called: inportb to read the io port
*****************************************************/
UBYTE IORead8(ULONG IO_Addr)
{
   return (inportb((UINT) IO_Addr));
```

```
}
/*****************************************************
 Function: IOWrite8
   Purpose: To write a byte out to an IO port
   Input: Value to be written and IO port address
   Output: None
   Assumptions: That the IO port actually exists
   Restrictions: Although IO_Addr is defined as a
                 ULONG it must not exceed 16 bits
                 in length as this is the maximum IO
                 address the X86 architecture can produce
   Other functions called: outportb to write to the io port
*****************************************************/
void IOWrite8(ULONG IO_Addr, UBYTE value)
{
   outportb((UINT) IO_Addr, value);
}
/*****************************************************
 Function: IORead32
   Purpose: To read a dword (32 bits) from an io port
   Input: IO address of dword to be read
   Output: dword read from io port
   Assumptions: That the IO port actually exists
   Restrictions: Although IO_Addr is defined as a
                 ULONG it must not exceed 16 bits in
                 length as this is the maximum IO
                 address the X86 architecture can produce
   Other functions called: none
*****************************************************/
ULONG IORead32(ULONG IO_Addr)
{
   ULONG result;
   asm
   {
          .386
          mov dx, [IO_Addr]
          in eax, dx
          mov [result], eax
   }
   return(result);
}
/*****************************************************
 Function: IOWrite32
   Purpose: To write a dword (32 bits) out to an IO port
   Input: Value to be written and IO port address
   Output: None
   Assumptions: That the IO port actually exists
   Restrictions: Although IO_Addr is defined as a
```

```
                ULONG it must not exceed 16 bits in
                length as this is the maximum IO
                address the X86 architecture can produce
    Other functions called: none
*************************************************/
void IOWrite32(ULONG IO_Addr, ULONG value)
{
    asm
    {
            .386
            mov dx, [IO_Addr]
            mov eax, [value]
            out dx, eax
    }
}
```

7.1.1 Reset

Figure 7.2 shows how to reset the SCRIPTS processor, by setting, then clearing, the Software Reset (SRST) bit in the ISTAT register. It executes a Read-Modify-Write for each register whose default value changes at reset.

Figure 7.2 Resetting the SCRIPTS Processor

```
* sets SRST(bit 6) */
IOWrite8(ISTAT, (IORead8(ISTAT) | 0x40));/
* clears SRST(bit 6) */
IOWrite8(ISTAT, (IORead8(ISTAT) & 0xBF))/
```

7.1.2 Table Indirect Operations

This section of the chapter provides an overview of table indirect operations. More information on Table Indirect operation and on creating a table is provided in Chapter 9, “SCRIPTS Programming Topics.”

7.1.2.1 Initializing a Table

Figure 7.3 is an example SCRIPTS table declaration. Although NASM does not actually generate any output based on the table declaration, it does place offsets into the SCRIPTS array based on the order of the buffers in the table declaration. The actual byte values and byte counts in the SCRIPTS instruction are not used at this stage because NASM does not generate any output from the table declaration.

Figure 7.3 SCRIPTS Table Declaration

```
TABLE dsa_table \
    sendmsg = ??, \
    rcvmsg = ??, \
    cmd_adr = ??, \
    device = ID{??}, \
    status_adr = ??, \
    ext_buf = ??, \
    sync_in = ??, \
data_adr = ??
```

7.1.2.2 Creating Table Indirect Entry Offsets

The “C” code example in Figure 7.4 sets up a table that can be used with the table indirect addressing mode. Each entry in the table is a pair of 32-bit values. These entries reference the same buffers as the SCRIPTS code examples above. For more illustration on the relationship between these pieces of code, refer to Section 7.1.2, “Table Indirect Operations.” For this SCRIPTS program to work correctly, the table must start on a Dword boundary and the offset labels must be in the same order as in the SCRIPTS table declaration.

Figure 7.4 Creating Table Indirect Entry Offsets

```
/* The following definition sets up a table that can be
used with the LSI53C8XX table indirect addressing mode.
Each entry in the table is a pair of 32 bit values. For
the SCRIPTS routine to work correctly the table MUST start
on a word boundary and the offset labels must be in the
same order in the SCRIPTS table declaration. */
enum offsets {
    SENDMSG = 0,
    RCVMSG,
    CMD_ADR,
    DEVICE,
    STATUS_ADR,
    EXT_BUF,
    SYNC_IN,
    DATA_ADR, /* DATA_ADR must be last buffer.
};
```

7.1.2.3 Defining the Table Structure

The code in Figure 7.5 defines a data structure with two fields, a count and an address, which correspond to one element in the DSA table. A type is then defined and a pointer to a variable of this type is also defined. This pointer and the enumerated offsets defined above are used to access specific elements of the table. This example defines the table structure, but no space has been allocated in memory.

Figure 7.5 Data Structure and Type Definition

```
struct_table {
    uquad   count;
    uquad   address;
};
typedef struct_table table:
```

7.1.2.4 Declaring a Pointer to the Table

The code example below defines declaring a pointer to the table.

```
extern table   *buffer_table;
```

7.1.2.5 Allocating Memory for the Table

The code in Figure 7.6 defines allocating memory for the table.

Figure 7.6 Data Structure and Type Definition

```
int init_table(void)
{
    UBYTE *buf_ptr;   /* temp ptr to ti tables */
    /* allocate space for table */
    buf_ptr =  (UBYTE far *) malloc((TABLE_SIZE

sizeof(ti_entry))+ 4);
    /* did we get the memory */
    if (buf_ptr == NULL) return(COMMANDFAILED);

    /* dword align the table buffer, ByteAlignBuffer does
         this */
    dsa_table = (ti_entry *) ByteAlignBuffer(buf_ptr, 0);

   /* This initializes the DSA register to point to the
buffer table that was allocated above*/
   IOWrite32(PCIDeviceIOBase+DSA, getPhysAddr(dsa_table));
    return(GOOD);
}
```

7.1.2.6 Using a Table

The example in Figure 7.7 creates two buffers, `identify_msg` and `test_unit_ready_cmd`. The byte counts and addresses for these buffers are then loaded into the `CMD_ADR` and `SENDMSG` elements of the `DSA_table` array. These examples define a message and a command buffer in the desired table and loads the bytes into the table. The enumerated types are used in the Test Unit Ready example to index into the table.

Figure 7.7 Creating Buffers

```
static ubyte identify_msg[] = {
    0xc0
/* 0xc0 = allow disconnect, 0x80 = no disconnect */
};
static ubyte test_unit_ready_cmd[] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
/* drive is the destination ID for the I/O*/
DSA_table[DEVICE].count=(ULONG)drive<< 16;
DSA_table[CMD_ADR].count = sizeof(test_unit_ready_cmd);
DSA_table[CMD_ADR].address=getPhyAddr(test_unit_ready_cmd;

DSA_table[SENDMSG].count = 1;
DSA_table[SENDMSG].address = getPhysAddr(identify_msg);

DSA_table[STATUS_ADR].count = 1;
DSA_table[STATUS_ADR].address = getPhysAddr(status);
```

7.2 Patching

Sometimes it is necessary for the “C” code to modify some elements of the SCRIPTS array after buffer allocation. This is called patching. Patching is required when relative transfer control instructions or table indirect addressing are not used. However, most applications will take advantage of these features, so patching is not often required. When patching is necessary, the general format of the patch in “C” is
`SCRIPT[patch_offset] = patch_value;`

When only part of the 32-bit value in the SCRIPTS array must be modified, a Read-Modify-Write can be used. The format for this type of operation is
`SCRIPT[patch_offset]|= patch_value;`
Any arithmetic or logical operator can be used in place of the logical OR (|) symbol to make the desired modification.

The `patch_offset` is an index into the SCRIPTS array where the patch must be made. This value is usually obtained from one of the sections of the NASM output file. Please see Chapter 5, “The NASM Output File,” for more information on the NASM output file and the patch offsets it contains.

The patch_value is usually either a buffer physical address or a byte count, but could be anything that modifies the part of the SCRIPTS program.

The remainder of this section contains patching techniques for various instructions and buffer types that require modification at run time. Please note that this chapter only describes the most common types of patches. Other types of patching can generally be used to modify any part of a SCRIPTS instruction by using the ENTRY point patching method described in this section.

7.2.1 EXTERN Buffers

This section of the chapter describes the procedure for setting up EXTERN buffers.

1. Create a buffer in 'C' statically or dynamically if necessary as shown in the example below.

   ```
   UCHAR msgin_buf[4];
   ```

2. Patch the SCRIPT wherever this buffer is used, with the patch array generated by NASM shown in the example below.

   ```
   SCRIPT[E_ex_buf1_Used[1]] = VirttoPhys(msgin_buf);
   ```

See Chapter 5, "The NASM Output File," for more information on the _Used patch array.

7.2.2 RELATIVE Buffers

RELATIVE buffers are essentially the same as External buffers. The SCRIPTS output file contains some additional information to aid in patching the SCRIPTS instructions. The individual relative buffer offset is encoded into the SCRIPTS instruction. There are two methods for establishing RELATIVE buffers.

7.2.2.1 Procedure 1

1. Create a buffer to hold all the individual relative buffers.

   ```
   UCHAR rel_buffer[8]
   ```

2. Patch the SCRIPTS array using the Patch array generated by NASM.

   ```
   SCRIPT[R_rel_buf2_Used[0]] += VirttoPhys(rel_buffer)
   ```

7.2.2.2 Procedure 2

1. Create a buffer to hold all the individual relative buffers.

   ```
   UCHAR rel_buffer[8]
   ```

2. Patch all buffers in one loop if the main Patch array is accessed and the Header record is used. The `-o` assembler option must be used for this procedure to work.

   ```
   for(i=0; i<Rel_Count; i++) {
   SCRIPT[Rel_Patches[i]] += VirttoPhys(rel_buffer);
   }
   ```

See Chapter 5, “The NASM Output File,” for more information on the structures created for patching relative buffers.

7.2.3 ABSOLUTE Values

ABSOLUTE values are patched exactly like EXTERN buffers. The `-o` compiler option must be used to patch Absolutes. See Section 5.2.6, “Absolute,” for more information on ABSOLUTE values.

7.2.4 Buffer Addresses

Buffer addresses are usually patched into Block Move, Memory to Memory, or Load/Store instructions. They are usually defined as EXTERNS, RELATIVES, or ABSOLUTES. The general format of this type of patch is:

```
SCRIPT[X_buffername_Used[n]] = VirttoPhys(c_buffer);
```

Where X is either E (Extern), R (Relative), or A (Absolute) depending on the type of buffer used.

n is the nth occurrence of this buffer in the SCRIPTS program.

c_buffer is a buffer/array defined in ‘C’.

See Chapter 5, “The NASM Output File,” for more information on the _Used array.

7.2.5 Byte Counts

Byte counts are usually patched into Block Move, Memory to Memory, or Load/Store instructions. Since the byte count is usually encoded in the

first Dword along with the opcode, be sure to OR in the byte count instead of doing a straight assignment. Byte counts to be patched are usually defined as EXTERNS, RELATIVES, or ABSOLUTES. The general format of this type of patch is:

```
SCRIPT[X_bytecount_Used[n]] |= c_byte_count;
```

Where X is either E, R, or A

n is the nth occurrence of this byte count in the SCRIPTS program.

c_byte_count is a variable/constant byte count value.

See Chapter 5, “The NASM Output File,” for more information on the _Used array.

7.2.6 Absolute JUMP/CALL Addresses

Use the LABELPATCHES array to patch in absolute JUMP or CALL addresses. The absolute offset from the beginning of the SCRIPTS instruction is encoded in the JUMP instruction at assembly. All that needs to be added is the base physical address of the SCRIPTS array. The general format of this type of patch is:

```
SCRIPT[LABELPATCHES[n]] += VirttoPhys(SCRIPT);
```

Where n is the nth jump instruction to be patched.

This can be automated using a loop and the PATCHES values.

See Chapter 5, “The NASM Output File,” for more information on the LABELPATCHES array.

7.2.7 Entry Locations

Entry offsets are byte offsets, not Dword offsets. Divide the Entry offset by 4 to get to a SCRIPTS instruction offset. This method can be used to modify any SCRIPTS instruction that normally does not need patching, but needs to be modified in a special circumstance. The general format of this type of patch is:

```
SCRIPT[Ent_entrylabel/4 + n] = value;
```

Where n is either 0, 1 or 2 depending on the particular Dword of the instruction that needs to be accessed.

If the first Dword of an instruction is being accessed, a Read-Modify-Write instruction may need to be done to maintain the opcode.

See Chapter 5, "The NASM Output File," for more information on the `Ent_offsets`.

7.2.8 Self-Modifying SCRIPTS Code

It is sometimes necessary to create self-modifying SCRIPTS code. When creating self-modifying SCRIPTS code it should be done in such a way that external patching is only necessary at initialization time.
Self-modifying code can be accomplished by using either a Memory-to-Memory Move instruction or a combination of LOAD and STORE instructions. The SCRIPTS example in Figure 7.8 shows a Memory-to-Memory Move modifying a Move Register instruction such that an offset can be added to a base address for jumping into a table.

Figure 7.8 Self-Modifying Code

```
ENTRY Patch_label1
ENTRY Patch_label2
EXTERN SCRATCHA1_addr
EXTERN SCRATCHB_addr
MOVE MEMORY 4, Patch_label2+4, SCRATCHB_addr
MOVE MEMORY 1, SCRATCHA1_addr, Patch_label1+1
Patch_label1:
MOVE SCRATCHB0 + 0 to SCRATCHB0
MOVE SCRATCHB1 + 0 to SCRATCHB1 WITH CARRY
MOVE SCRATCHB2 + 0 to SCRATCHB2 WITH CARRY
MOVE SCRATCHB3 + 0 to SCRATCHB3 WITH CARRY
MOVE MEMORY 4, SCRATCHB_addr, Patch_label2+4
Patch_label2:
JUMP REL(Jump_Table)
    .
    .
Jump_Table:
```

Patches to the SCRIPTS Instruction may be needed. Patch the Labels in the Memory-to-Memory Move instructions first:

```
for (i=0; i<PATCHES; i++) {
   SCRIPT[LABELPATCHES[i]] += VirttoPhys(SCRIPT);
}
```

Next patch Scratch register physical addresses:

```
SCRIPT[E_SCRATCHA1_addr_Used[0]] =
VirttoPhys(chip_reg[ScratchA]) + 1;
SCRIPT[E_SCRATCHB_addr_Used[0]] =
VirttoPhys(chip_reg[ScratchB]);
SCRIPT[E_SCRATCHB_addr_Used[1]] =
VirttoPhys(chip_reg[ScratchB]);
```

These are the only patches required. LOAD and STORE instructions could be used to replace the Memory-to-Memory Move instructions.

Note: SCRATCHA1 is used instead of SCRATCHA0 due to the alignment requirements of the Memory to Memory Move instruction.

7.3 Running a SCRIPTS Program

The SCRIPTS program is ready to run after all Command, Data, and Message buffers have been set up for the I/O. Writing the physical address of the program to the DSP register, starting at bit 3, initiates the SCRIPTS program. There are sections of sample code in Figures 7.9 and 7.10.

The entry points named in this example are all different points where SCRIPTS instructions could start.

```
static uquad start_offset[] = {
   Ent_init_siop3, Ent_start_up4, Ent_switch5
    };
```

This example starts the SCRIPTS program:

```
IOWrite32(PCIDeviceIOBase+DSP, getPhysAddr(script) +

start_offset[mode]);
```

In this example, `mode = 0` begins at `init_siop` label, `mode = 1` begins at `start_up`, and `mode = 2` begins at the `switch` label.

Figure 7.9 General.ss SCRIPTS Source File

```
; Single-threaded general purpose SCRIPTS routine

; Offset for counts and addresses in the table
TABLE dsa_table \
sendmsg = ??, \
```

Figure 7.9 General.ss SCRIPTS Source File (Cont.)

```
rcvmsg = ??, \
cmd_adr = ??, \
device = ID{??}, \
status_adr = ??, \
ext_buf = ??, \
sync_in = ??, \
data_adr = ??

; The SCRIPTS routine has finished initializing the SIOP.
Absolute done_init = 0x01

ABSOLUTE ok = 0x00
ABSOLUTE err1 = 0x0ff01
ABSOLUTE err2 = 0x0ff02
ABSOLUTE err3 = 0x0ff03
ABSOLUTE err4 = 0x0ff04
ABSOLUTE err5 = 0x0ff05
ABSOLUTE err6 = 0x0ff06
ABSOLUTE err7 = 0x0ff07
ABSOLUTE err8 = 0x0ff08
ABSOLUTE err9 = 0x0ff09

EXTERN dsa_storage, out_offset, in_offset

; SCSI I/O entry point. This address must be loaded into the
; SIOP before initiating a SCSI I/O.
   ENTRY init_siop
   ENTRY start_up
   ENTRY switch
   ENTRY datain
   ENTRY dataout

3  init_siop:
   INT done_init

4  start_up:

   SELECT ATN FROM device, REL(resel)

   ; Every phase comes back to here.
5  switch:
          JUMP REL(msgin), WHEN MSG_IN
          JUMP REL(msgout), IF MSG_OUT
          JUMP REL(command_phase), IF CMD
          JUMP REL(dataout), IF DATA_OUT
          JUMP REL(datain), IF DATA_IN
```

Figure 7.9 General.ss SCRIPTS Source File (Cont.)

```
            JUMP REL(end), IF STATUS

    INT err1

msgin:
    MOVE FROM rcvmsg, WHEN MSG_IN
    JUMP REL(ext_msg), IF 0x01
    JUMP REL(disc), IF 0x04
    CLEAR ACK
    JUMP REL(switch), IF 0x02 ; ignore save data pointers
    JUMP REL(switch), IF 0x07 ; ignore message reject)
    JUMP REL(switch), IF 0x03 ; ignore restore data pointers
    INT err2

ext_msg:
    CLEAR ACK
    MOVE FROM ext_buf, WHEN MSG_IN
    JUMP REL(sync_msg), IF 0x03

    INT err3

sync_msg:
    CLEAR ACK
    MOVE FROM sync_in, WHEN MSG_IN
    CLEAR ACK
    JUMP REL(switch)

disc:
    MOVE SCNTL2 & 0x7f to SCNTL2 ;expect disconnect
    CLEAR ACK
    WAIT DISCONNECT
    WAIT RESELECT REL(select_adr)

    INT err4, WHEN NOT MSG_IN
    MOVE FROM rcvmsg, WHEN MSG_IN
    CLEAR ACK
    INT err9
    JUMP REL(switch)

msgout:
    MOVE FROM sendmsg, WHEN MSG_OUT
    JUMP REL(switch)

command_phase:
    MOVE FROM cmd_adr, WHEN CMD
    JUMP REL(switch)
```

Figure 7.9 General.ss SCRIPTS Source File (Cont.)

```
; After every data transfer add 8 to data_adr. This allows
; scatter/gather operations when the list of addresses to
; read or write is appended to the end of the buffer_table.

1  dataout:
          MOVE FROM data_adr, WHEN DATA_OUT
          MOVE MEMORY 4, out_offset, scratch_adr
          CALL REL(addscratch)
          MOVE MEMORY 4, scratch_adr, out_offset
          JUMP REL(switch)
2  datain:
          MOVE FROM data_adr, WHEN DATA_IN
          MOVE MEMORY 4, in_offset, scratch_adr
          CALL REL(addscratch)
          MOVE MEMORY 4, scratch_adr, in_offset
          JUMP REL(switch)

addscratch:
   MOVE SCRATCHA0 + 8 to SCRATCHA0
   MOVE SCRATCHA0 to SFBR
   JUMP REL(ck_carry), IF 0x00
   RETURN

ck_carry:
   MOVE SCRATCHA1 + 1 to SCRATCHA1
   RETURN

end:
   MOVE FROM status_adr, WHEN STATUS
   INT err5, WHEN NOT MSG_IN
   MOVE FROM rcvmsg, WHEN MSG_IN
   MOVE SCNTL2 & 0x7f to SCNTL2 ;expect disconnect
   CLEAR ACK
   WAIT DISCONNECT
   INT ok

resel:
   INT err6

select_adr:
   INT err7
```

Figure 7.10 General.out NASM Output File

```
typedef unsigned long ULONG;

ULONG SCRIPT[] = {
    0x98080000L,0x00000001L,
    0x47000018L,0x000001E8L,
    0x878B0000L,0x00000030L,
    0x868A0000L,0x000000F0L,
    0x828A0000L,0x000000F8L,
    0x808A0000L,0x00000100L,
    0x818A0000L,0x00000128L,
    0x838A0000L,0x00000180L,
    0x98080000L,0x0000FF01L,
    0x1F000000L,0x00000008L,
    0x808C0001L,0x00000030L,
    0x808C0004L,0x00000068L,
    0x60000040L,0x00000000L,
    0x808C0002L,0xFFFFFFA0L,
    0x808C0007L,0xFFFFFF98L,
    0x808C0003L,0xFFFFFF90L,
    0x98080000L,0x0000FF02L,
    0x60000040L,0x00000000L,
    0x1F000000L,0x00000028L,
    0x808C0003L,0x00000008L,
    0x98080000L,0x0000FF03L,
    0x60000040L,0x00000000L,
    0x1F000000L,0x00000030L,
    0x60000040L,0x00000000L,
    0x80880000L,0xFFFFFF48L,
    0x7C027F00L,0x00000000L,
    0x60000040L,0x00000000L,
    0x48000000L,0x00000000L,
    0x54000000L,0x00000118L,
    0x9F030000L,0x0000FF04L,
    0x1F000000L,0x00000008L,
    0x60000040L,0x00000000L,
    0x98080000L,0x0000FF09L,
    0x80880000L,0xFFFFFF00L,
    0x1E000000L,0x00000000L,
    0x80880000L,0xFFFFFEF0L,
    0x1A000000L,0x00000010L,
    0x80880000L,0xFFFFFEE0L,
    0x18000000L,0x00000038L,
    0xC0000004L,0x00000000L,0x000DFE34L,
    0x88880000L,0x00000044L,
    0xC0000004L,0x000DFE34L,0x00000000L,
    0x80880000L,0xFFFFFEB0L,
```

Figure 7.10 General.out NASM Output File (Cont.)

```
    0x19000000L,0x00000038L,
    0xC0000004L,0x00000000L,0x000DFE34L,
    0x88880000L,0x00000014L,
    0xC0000004L,0x000DFE34L,0x00000000L,
    0x80880000L,0xFFFFFE80L,
    0x7E340800L,0x00000000L,
    0x72340000L,0x00000000L,
    0x808C0000L,0x00000008L,
    0x90080000L,0x00000000L,
    0x7E350100L,0x00000000L,
    0x90080000L,0x00000000L,
    0x1B000000L,0x00000020L,
    0x9F030000L,0x0000FF05L,
    0x1F000000L,0x00000008L,
    0x7C027F00L,0x00000000L,
    0x60000040L,0x00000000L,
    0x48000000L,0x00000000L,
    0x98080000L,0x00000000L,
    0x98080000L,0x0000FF06L,
    0x98080000L,0x0000FF07L

};
3  #define Ext_Count
char *External_Names[Ext_Count] = {
    "dsa_storage",
    "in_offset",
    "out_offset"
};

#define E_in_offset 0x00000000L
ULONG E_in_offset_Used[] = {
    0x0000005BL,
    0x00000061L
};

#define E_out_offset 0x00000000L
ULONG E_out_offset_Used[] = {
    0x0000004FL,
    0x00000055L
};

#define Abs_Count 11
char *Absolute_Names[Abs_Count] = {
    "done_init",
    "err2",
    "err1",
```

Figure 7.10 General.out NASM Output File (Cont.)

```
    "err3",
    "err4",
    "err5",
    "err6",
    "err7",
    "err9",
    "ok",
    "scratch_adr"
};

#define A_ok 0x00000000L
ULONG A_ok_Used[] = {
    0x0000007DL
};

#define A_done_init 0x00000001L
ULONG A_done_init_Used[] = {
    0x00000001L
};

#define A_err1 0x0000FF01L
ULONG A_err1_Used[] = {
    0x00000011L
};

#define A_err2 0x0000FF02L
ULONG A_err2_Used[] = {
    0x00000021L
};

#define A_err3 0x0000FF03L
ULONG A_err3_Used[] = {
    0x00000029L
};

#define A_err4 0x0000FF04L
ULONG A_err4_Used[] = {
    0x0000003BL
};

#define A_err5 0x0000FF05L
ULONG A_err5_Used[] = {
    0x00000073L
};

#define A_err6 0x0000FF06L
```

Figure 7.10 General.out NASM Output File (Cont.)

```
ULONG A_err6_Used[] = {
    0x0000007FL
};

#define A_err7 0x0000FF07L
ULONG A_err7_Used[] = {
    0x00000081L
};

#define A_err9 0x0000FF09L
ULONG A_err9_Used[] = {
    0x00000041L
};

#define A_scratch_adr 0x000DFE34L
ULONG A_scratch_adr_Used[] = {
    0x00000050L,
    0x00000054L,
    0x0000005CL,
    0x00000060L
};

2   #define Ent_datain 0x00000160L
1   #define Ent_dataout 0x00000130L
3   #define Ent_init_siop 0x00000000L
4   #define Ent_start_up 0x00000008L
5   #define Ent_switch 0x00000010L

ULONG INSTRUCTIONS = 0x0000003FL;
ULONG PATCHES = 0x00000000L;
```

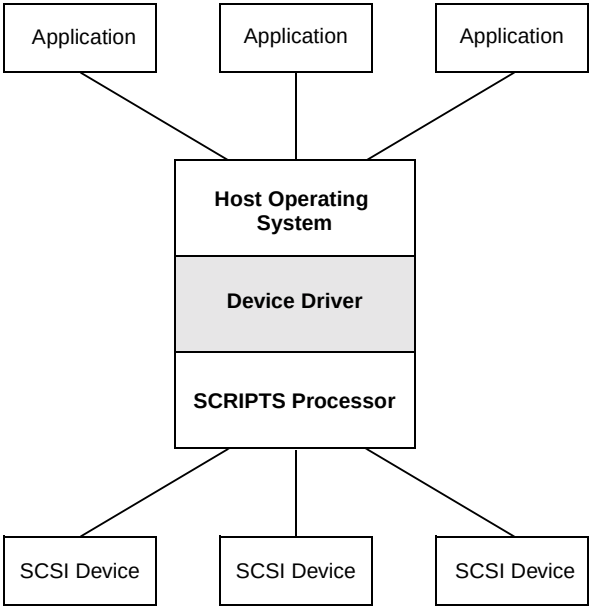
Chapter 8
Writing Device Drivers with SCRIPTS

This chapter describes writing SCSI device drivers with SCRIPTS and contains the following sections:

8.1 Device Driver Overview

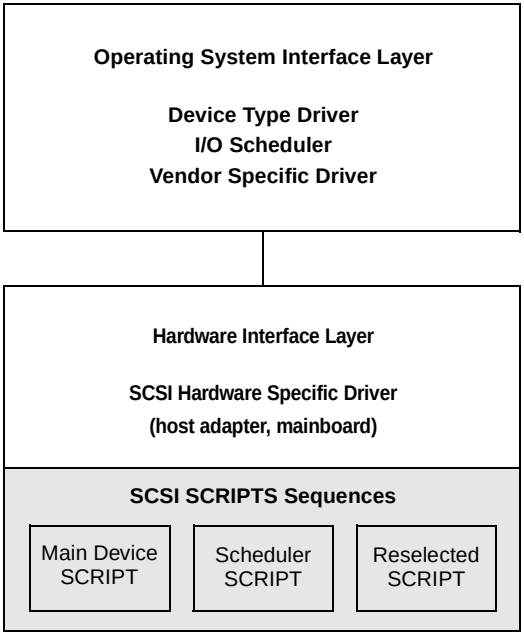
The architecture of a SCSI system can be viewed in layers, with each layer providing data to the layers immediately above and below. The device driver interfaces between the host operating system and the chip hardware and firmware. The device driver, host operating system, and all applications reside in the host computer's main memory. The LSI53C7XX/8XX/10XX family is a separate hardware component, but has direct access to host memory. Figure 8.1 shows the relationship of the device driver to other parts of the SCSI system.

Figure 8.1 The Role of the SCSI Device Drivers



The device driver itself contains two layers, illustrated in Figure 8.2. The top layer is the operating system interface. It accepts and interprets I/O requests from the host operating system. These requests may vary, depending on the type and vendor of the SCSI device. The formatted requests are passed to the hardware interface, or lower layer of the driver. The operating system interface must also schedule SCSI bus accesses when more than one device is active. It schedules the I/O requests and tracks the completed and outstanding I/Os based on status passed back from the hardware interface. The SCRIPTS program is compiled with the driver program and is loaded into host memory when the device driver program starts.

Figure 8.2 SCSI Device Driver Layers



The hardware interface layer:

- Interprets the operating system interface's formatted requests.
- Prepares the SCSI device by initializing the DMA, SCSI, and Interrupt registers and by loading the appropriate SCRIPTS into host memory.
- Reserves memory for any data buffers that will be used by the SCRIPTS program.
- Initializes data buffer addresses, byte counts, and SCSI IDs embedded in the SCRIPTS code.
- Starts the execution of the SCRIPTS routine by loading the DSP register (0x2C–0x2F) with the address of the first SCRIPTS instruction.
- Waits for an interrupt to signal that the I/O is complete.
- Passes I/O status information back to the operating system interface.

8.2 Command Block

When the operating system interface layer of the SCSI device driver receives an I/O request, it creates a data structure in host memory. This data structure contains the information required by the hardware interface for that specific request. This information generally includes:

- Length of the array
- SCSI ID for the target device
- Logical unit number (LUN)
- Length of the command block
- SCSI command containing the beginning block and the number of blocks to be transferred
- A place for the hardware interface to write its completion status. The operating system interface reads the completion status and uses it to update the scheduler information.

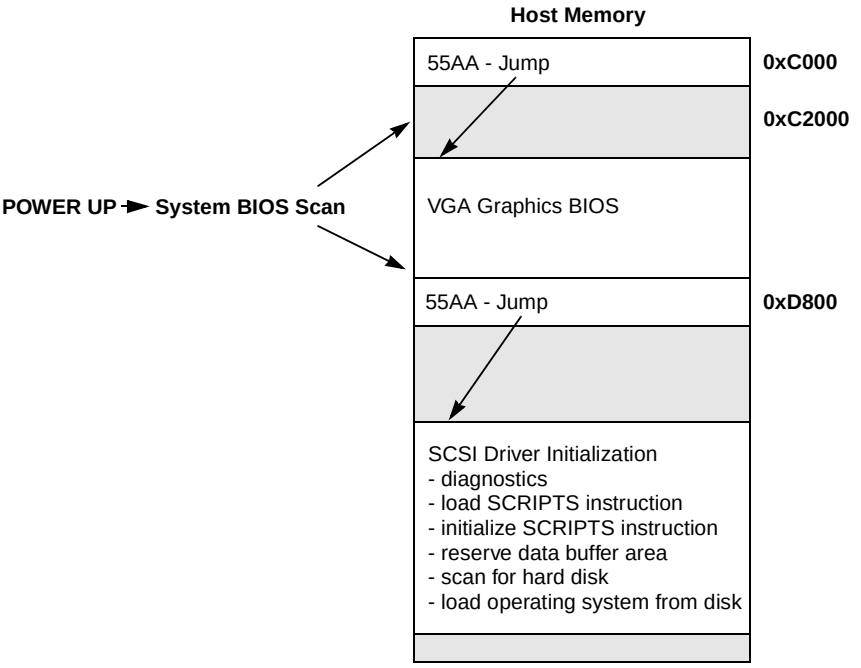
8.3 Power Up Example

The hardware interface initializes the chip whenever the system is powered up or reset. In the DOS example in Figure 8.3, the system BIOS scans host memory for a ROM, signified by a 55AA code. It reads the third byte of the ROM, which contains a jump address. The following SCRIPTS processor initialization information is located at that address:

- diagnostics to be run
- SCRIPTS to be loaded
- data buffer areas to be reserved

After performing these tasks, the hardware interface scans for the hard disk and after locating it, downloads the operating system. The operating system cannot be loaded from the disk until the SCSI driver is active. This power on sequence of activities is illustrated in Figure 8.3.

Figure 8.3 Power Up Examples



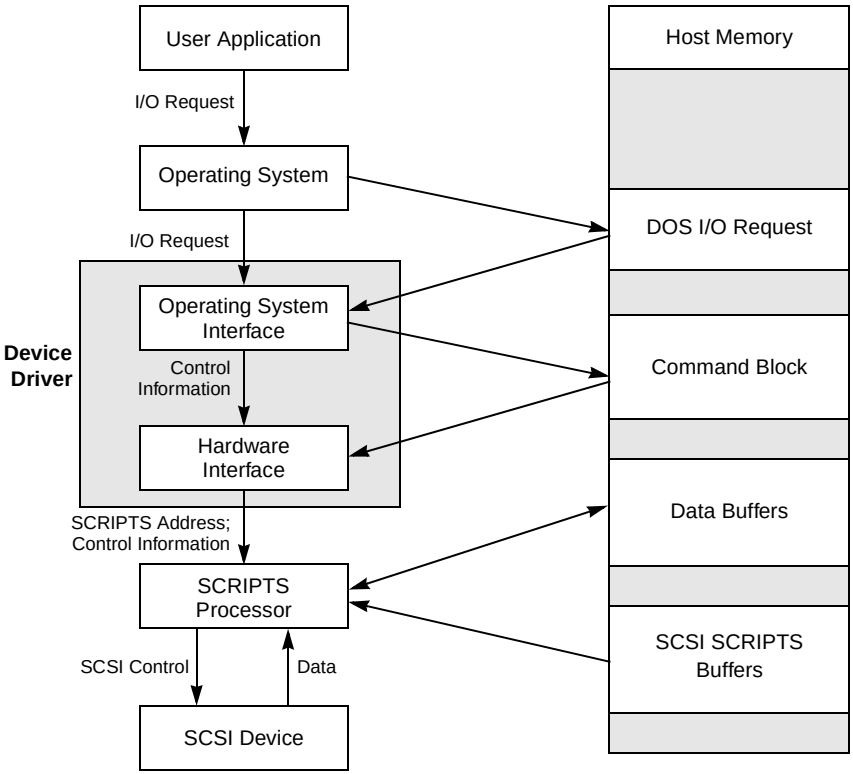
8.4 I/O Request Process

Figure 8.4 illustrates a typical SCSI I/O operation. The I/O begins when the user application makes a request to the host operating system to access data on a SCSI device. The request is passed to the SCSI device driver's operating system interface where it is interpreted, scheduled, and formatted for the hardware interface. The operating system interface creates a data structure in host memory, which it passes to the hardware interface layer for execution. The hardware interface uses the information in the command block to determine which SCRIPTS routine to run, as well as where to place the data in memory.

The hardware interface sets up the data areas for the command and data buffers. These buffers are initialized table areas and buffers that are needed for the SCRIPTS execution. It subsequently loads the SCRIPTS starting address into the DSP register of the chip. The SCRIPTS processor executes the subroutine, accessing the drive with the SCSI

device ID specified. When the I/O is complete, the hardware interface receives an interrupt and notifies the operating system interface. The operating system interface reads the completion status and uses it to update the scheduler information. For more information on the scheduler, refer to Chapter 10, “Multithreaded I/O.”

Figure 8.4 I/O Operation



8.5 How to Write a Device Driver with SCRIPTS

To develop an executable SCSI SCRIPTS program, you must first define the SCSI functions required. To do this, you identify which functions will be executed in SCRIPTS code and which ones must be contained in other parts of the driver code. After determining this, you design the specific algorithms for the functions that will be executed in the SCSI SCRIPTS portion of the SCSI driver. A SCSI SCRIPTS program contains

two areas: the definition area and the SCRIPTS area. The definition area contains variable and absolute values. These values may describe a variable location, variable byte count, or a fixed status byte value. The SCRIPTS area contains the SCSI instructions.

The SCRIPTS language writes instructions and assembles them to create the SCRIPTS output file. The assembler output is a “C” include file that includes relocation information required to load the SCRIPTS object module into main memory, if any relocation is required. It can be directly included in firmware written in the “C” language.

When the SCRIPTS starting address is loaded, the SCRIPTS absolute jump addresses must be resolved. You must patch in the correct buffer addresses, byte counts, destination ID, and so forth, if table indirect addressing is not used.

Writing a logical I/O driver for the LSI53C7XX/8XX/10XX family is easier than previous generation solutions. Because SCSI sequences are so simple to implement when written in SCSI SCRIPTS, you can rapidly prototype SCSI sequences for proof of concept and build on them to create more complete driver programs.

8.6 Table Indirect Addressing

Table indirect addressing simplifies SCRIPTS by separating addresses and device information from control information in Block Move and Select/Reselect instructions. One of the major advantages of table indirect addressing is that SCRIPTS directly loads operating system I/O data from the tables, which increases program efficiency and simplifies program structure. These tables eliminate the need for patching SCRIPTS at the beginning of an I/O. The table can begin on any Dword boundary and can cross system segment boundaries. There are three restrictions on the placement of tables in memory:

1. The I/O data structure must lie within 8 Mbytes above or below the base address.
2. An I/O table entry must have all 8 bytes contiguous in system memory.
3. The table must be a contiguous data structure of 8-byte entries.

Prior to the start of an I/O, load the DSA register with the base address of the table indirect data structure. The address must be on a Dword boundary. Adding the DSA to the 24-bit signed offset value from the opcode at the start of a table indirect instruction generates the address of the table entry. Both positive and negative offsets are allowed. With table indirect addressing, it is not necessary to initialize the SCSI ID, byte counts, clock dividers, synchronous parameters, or data buffers within the SCRIPTS instruction. Instead, only the table in memory needs to be updated.

To use table indirect addressing, you must set up tables in memory similar to the one shown in Figure 8.5. These tables contain device IDs, synchronous period information, byte counts, and data addresses. The data in the table entry is fetched into the appropriate instruction, depending on whether it is a Block Move or a Select/Reselect.

8.6.1 Block Move Instructions

When you select the table indirect mode by using the FROM operator in a SCRIPTS Block Move instruction, the 32-bit start address is treated as a 24-bit signed value. After the instruction is moved into the chip, the 24 bits are added to the DSA register to form a 32-bit physical address. The byte count (24 bits of count plus 8 bits of high-order zeros) and the data buffer address (32 bits) are fetched from this new address.

There are several programming implications of table indirect addressing. First, a standard SCSI data structure can be designed with values at predefined offsets. The Block Move instruction does not require the actual 32-bit address or 24-bit count to be within the instruction itself. At the start of an I/O and after the actual data structure is built, no further firmware intervention is required except loading the data table base address into the DSA register. Second, the SCRIPTS instructions may be placed in a PROM because no dynamic alteration is required at the start of an I/O. Finally, only one copy of the main SCSI SCRIPTS program is needed for all I/O operations, with a fast context switch used to change to another I/O. Only the data structure is unique to each I/O, and the SCRIPTS instructions are reusable, as shown in Table 8.1.

Table 8.1 Data Structure

Dword 0	Byte Lane 3	Byte Lane 2	Byte Lane 1	Byte Lane 0
	Byte Count			
Dword 1	Byte Lane 3	Byte Lane 2	Byte Lane 1	Byte Lane 0
	Address			

8.6.2 Select/Reselect Instructions

During a Select/Reselect and when FROM is used to indicate table indirect addressing, the 24-bit signed value in the DBC register is an offset relative to the value of the DSA register. The table indirect feature allows fetching the Synchronous Clock Conversion, Enable Wide SCSI, Clock Conversion Factor, SCSI Device ID, Synchronous Offset, and Synchronous Period bit values from an I/O data structure that is built at the start of an I/O. Thus, an I/O can begin with no requirement to write the values into the chip or into the actual SCRIPTS instruction in memory. In the I/O data structure, the user must have written the following 8-byte value, as shown in Table 8.2.

Table 8.2 I/O Data Structure

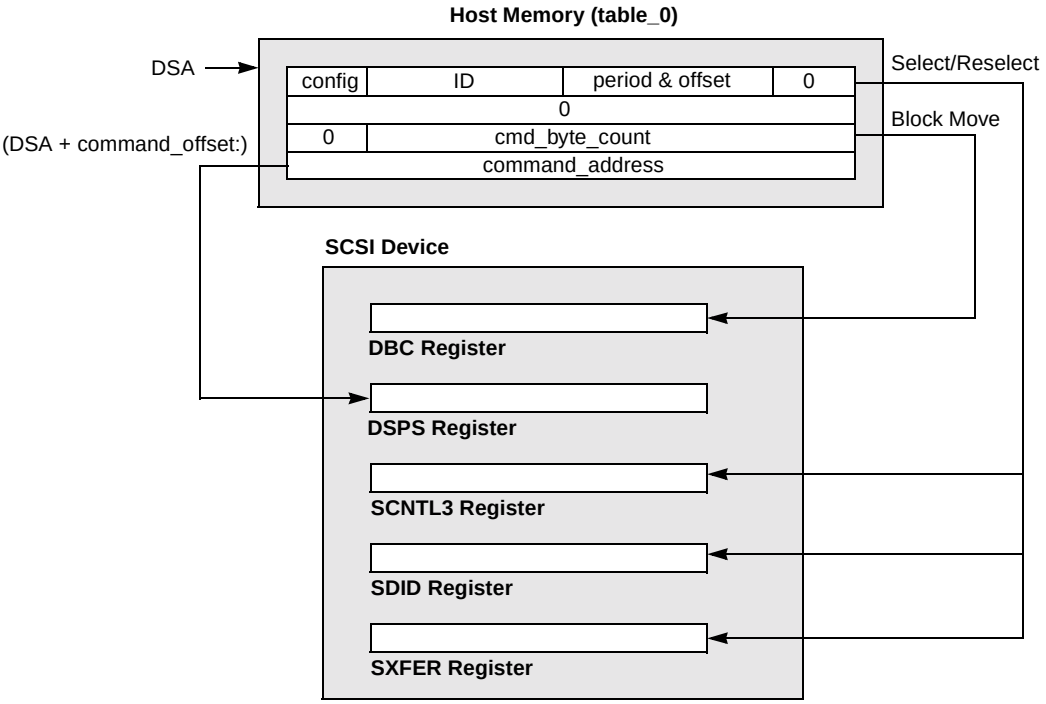
Dword 0	**Byte Lane 3**	**Byte Lane 2**	**Byte Lane 1**	**Byte Lane 0**
	Config (SCNTL3)	Device ID (SDID)	Period & Offset (SXFER)	R[1] (SCNTL4)[2]
Dword 1	**Byte Lane 3**	**Byte Lane 2**	**Byte Lane 1**	**Byte Lane 0**
	R	R	R	R

1. LSI53C896 and earlier.
2. LSI53C10XX.

The configuration information in byte lane 3 is mapped into the SCNTL3 register (0x03). This includes the Synchronous Clock Conversion Factor,

Enable Wide SCSI, Enable Ultra SCSI, and Clock Conversion Factor. The Encoded SCSI destination ID in byte lane 2 is mapped into the SDID register (0x06), and the period and offset information in byte lane 1 is mapped into the SXFER register (0x05). 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 DSA register. Figure 8.5 shows these relationships.

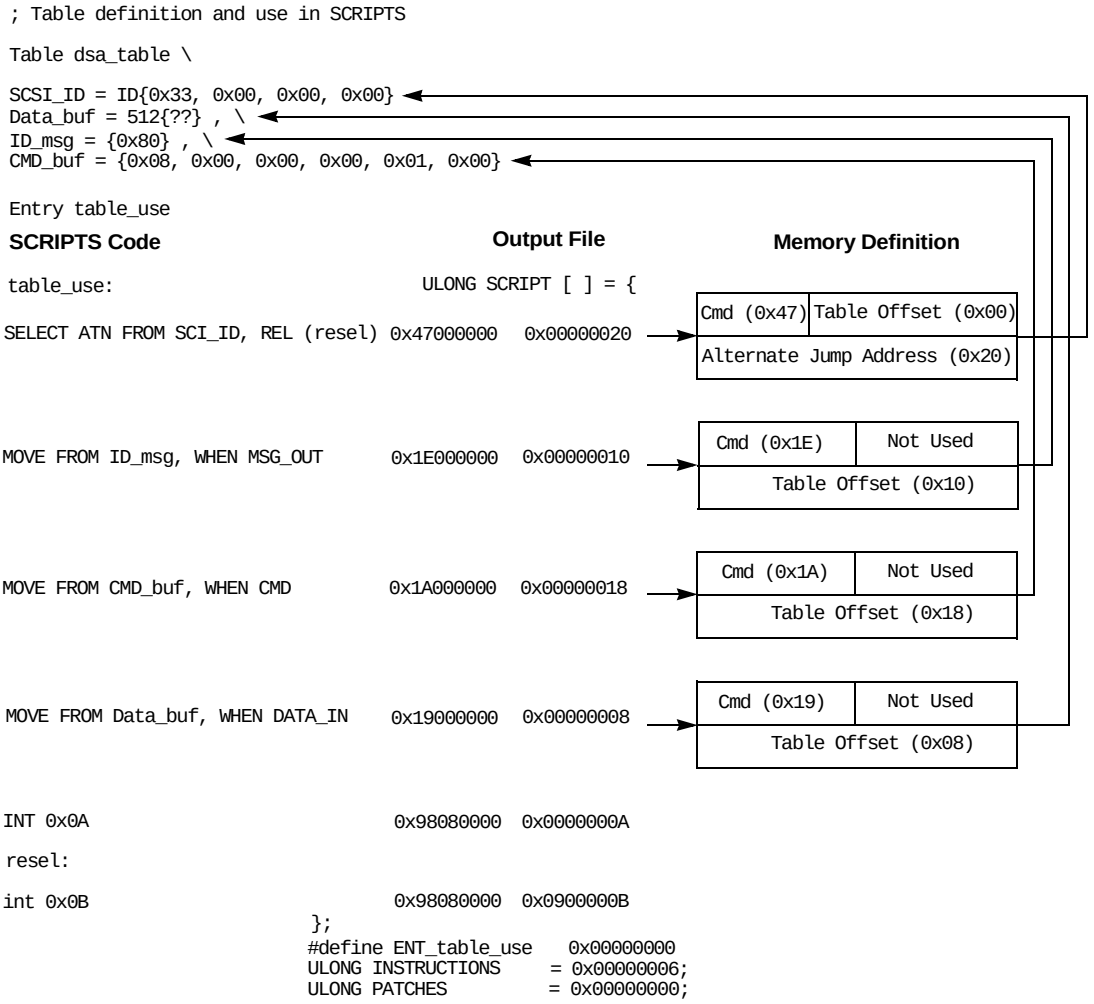
Figure 8.5 Table Indirect Addressing



8.6.3 Defining a Table

The first step in defining a table is to describe it in SCRIPTS code in terms of the order and size of table entries, or buffers. An example is shown in Figure 8.6.

Figure 8.6 Table Definitions



8.7 Relative Addressing

In the relative addressing mode, the 24-bit signed value in the DSPS register is the relative displacement from the current DMA SCRIPTS Pointer (DSP) register. Using this mode, the 32-bit physical address is formed at execution time, and there is no need to patch a SCRIPTS instruction at run time. Relative addressing can be used for jumps or calls and requires no initialization of jump and call addresses. This feature can

also be used with the alternate address field of Select, Reselect, Wait Select, and Wait Reselect instructions.

Note: Use the REL qualifier keyword in the SCRIPTS instructions to specify relative addressing. RELATIVE is a declarative keyword, used by the SCRIPTS assembler, to establish relative buffers. These relative buffers are not used in connection with relative addressing.

Chapter 9
SCRIPTS Programming Topics

This chapter presents general information for some of the programming tasks that are often performed by SCRIPTS programs. For the most up-to-date example code for many of these operations, please contact LSI Logic technical support.

This chapter contains the following sections:

9.1 Scatter/Gather Operations

You use scatter/gather to collect data that is scattered throughout memory and must be transferred across the SCSI bus together. Memory management units keep track of physical locations of user data that cannot be stored contiguously. During an I/O request for a SCSI device to fetch data, the memory management unit builds a gather table that provides the addresses of all of the desired data. There may be several entries, or pages, of data associated with a single transfer. Without scatter/gather each entry is treated as an individual transfer, requiring a processor interrupt and DMA setup.

With SCSI SCRIPTS, it is possible for you to set up multiple data buffer areas and then fill them rapidly without interrupting the host processor. This allows faster and more efficient scatter/gather operations. Block move data can come from any memory address, so scatter/gather operations for user data are transparent to the chip and the host processor. With the technique illustrated in Figure 9.1, a number of data buffers (pages, or gather table entries) are defined in advance and each is associated with a Block Move instruction. Any number of Block Moves can be hardcoded into the buffers. If the scatter/gather list requested has more entries than have been defined for the buffer, then an interrupt after the last entry in the series can inform the firmware it needs to set up the remaining scatter/gather entries after the first group is complete.

Figure 9.1 Scatter/Gather Operation

```
RW_Offset_patch_do:
;Relative offset will be changed so that we jump into the
;proper place in the scatter gather list
JUMP REL(Data_Out_xfer); Data_Out_xfer:
CHMOV FROM data_buf1, WHEN DATA_OUT CHMOV FROM data_buf2,
WHEN
; 16 moves to support Scatter Gather
DATA_OUT
CHMOV FROM data_buf3, WHEN DATA_OUT
CHMOV FROM data_buf4, WHEN DATA_OUT
CHMOV FROM data_buf5, WHEN DATA_OUT
CHMOV FROM data_buf6, WHEN DATA_OUT
CHMOV FROM data_buf7, WHEN DATA_OUT
CHMOV FROM data_buf8, WHEN DATA_OUT
CHMOV FROM data_buf9, WHEN DATA_OUT
CHMOV FROM data_buf10, WHEN DATA_OUT
CHMOV FROM data_buf11, WHEN DATA_OUT
CHMOV FROM data_buf12, WHEN DATA_OUT
CHMOV FROM data_buf13, WHEN DATA_OUT
CHMOV FROM data_buf14, WHEN DATA_OUT
CHMOV FROM data_buf15, WHEN DATA_OUT
CHMOV FROM data_buf16, WHEN DATA_OUT

; Check to see if we need more SG list entries
MOVE DWT & RW_NEED_MORE_SG_ENTRIES to SFBR
INT RW_Need_More_SG, if not 0
; If we are here then all the data was transferred
; so we set a flag to indicate that
MOVE SBR | RW_ALL_DATA_TRANSFERRED to DWT
JUMP REL(RW_Handle_Phase)
; *** Script move data ENTRY
```

```
RW_Offset_patch_di:
;Relative offset will be changed so that we jump into the
;proper place in the scatter gather list
JUMP REL(Data_In_xfer); Data_In_xfer:
CHMOV FROM rw_data_buf1, WHEN DATA_IN CHMOV FROM
rw_data_buf2, WHEN DATA_IN
; 16 moves to support Scatter Gather
CHMOV FROM rw_data_buf3, WHEN DATA_IN
CHMOV FROM rw_data_buf4, WHEN DATA_IN
CHMOV FROM rw_data_buf5, WHEN DATA_IN
CHMOV FROM rw_data_buf6, WHEN DATA_IN
CHMOV FROM rw_data_buf7, WHEN DATA_IN
CHMOV FROM rw_data_buf8, WHEN DATA_IN
CHMOV FROM rw_data_buf9, WHEN DATA_IN
CHMOV FROM rw_data_buf10, WHEN DATA_IN
CHMOV FROM rw_data_buf11, WHEN DATA_IN
CHMOV FROM rw_data_buf12, WHEN DATA_IN
CHMOV FROM rw_data_buf13, WHEN DATA_IN
CHMOV FROM rw_data_buf14, WHEN DATA_IN
CHMOV FROM rw_data_buf15, WHEN DATA_IN
CHMOV FROM rw_data_buf16, WHEN DATA_IN

; Check to see if we need more SG list entries
MOVE SBR & RW_NEED_MORE_SG_ENTRIES to SFBR
INT RW_Need_More_SG, if not 0
; If we are here then all the data was transferred
; so we set a flag to indicate that
MOVE SBR | RW_ALL_DATA_TRANSFERRED to DWT
```

The example in Figure 9.2 shows you an alternative method for doing scatter/gather operations using SCRIPTS. This mechanism uses a looping strategy to execute each scatter/gather entry. On each loop the DSA value is incremented by 8, effectively moving to the next scatter/gather entry in the scatter/gather list. Generally, when you use this strategy, the scatter/gather list is located at the end of the table indirect entries or is located separately from the other table indirect entries that handle (re)select, message, command and status phases. The DSA value is restored after the scatter/gather operations are complete or the target changes phase. This method of doing scatter/gather operations requires use of table indirect addressing.

Figure 9.2 Alternate Scatter/Gather Operation

```
Move_Data:
MOVE MEMORY 4, DSA_addr, ScratchB_addr  ; save DSA

address
JUMP REL(Data_In_Loop), WHEN DATA_IN
Data_Out_Loop:
MOVE FROM io_data_buf, WHEN DATA_OUT
MOVE DSA0 + 8 to DSA0 ; Update DSA for scatter gather
JUMP REL(Skip_Carry_Adds_DO), IF NOT CARRY; operations
MOVE DSA1 + 0 to DSA1 WITH CARRY
MOVE DSA2 + 0 to DSA2 WITH CARRY
MOVE DSA3 + 0 to DSA3 WITH CARRY
Skip_Carry_Adds_DO:
JUMP REL(Data_Out_Loop), WHEN DATA_OUT
MOVE MEMORY 4, ScratchB_addr, DSA_addr ; restore DSA
                                                                ;
address
JUMP REL(Get_Status), WHEN STATUS
JUMP REL(Handle_Message), WHEN MSG_IN
INT Unexpected_Phase
```

9.2 Loopback Mode

Loopback mode provides advanced diagnostic and testing capabilities. It allows the SCRIPTS processor to control and test all signals, regardless of mode, Initiator or Target, virtually by talking to itself. Loopback Mode also provides the ability to check the functionality of the part, ensuring proper SCRIPTS instruction fetches, checking bad parity procedures, and ensuring all data paths work properly. The SCRIPTS processor usually executes initiator instructions through the SCRIPTS program and the host CPU implements the Target Mode by asserting and polling the appropriate SCSI signals in the SOCL, SODL, SBCL, and SBDL registers. The Initiator Mode is accomplished using SCSI SCRIPTS and the Target Mode is implemented using “C” code to access the chip registers. The modes could be switched to test the Target Mode applications of the SCRIPTS processor.

To run the Loopback Mode correctly, the following registers must be initialized to the proper values.

- STEST2
  - Bits [4:3] should be set to turn on the Loopback Mode and set the SCSI pins to high impedance, so that signals are not asserted on to the SCSI bus. Bit 4 is reserved in the LSI53C10XX.
  - Bits [7:6] and 0 do not affect the loopback operation, but should remain cleared.
  - Bits 5 and [2:1] will not affect the running of the Loopback Mode. Bit 2 is reserved in the LSI53C10XX.
- DCNTL
  - Bit 4 should be set to turn on the Single Step Mode.

    This allows the target program to monitor when an initiator SCRIPTS instruction has completed.
  - Bits [3:2] should be cleared, and the remaining bit values will not affect the running of the Loopback Mode.
- DIEN
  - Bit 3 should be set to enable single step interrupts.

    This bit works in conjunction with the Single Step Mode bit to allow for monitoring of SCRIPTS instruction completion.
  - The remaining bit values in this register do not affect the running of the Loopback Mode.

9.2.1 Loopback Example – Selection

The example in Figure 9.3 demonstrates selection in SCSI Loopback Mode. It provides all the general code required to implement any of the various SCSI sequences in the Loopback Mode. This example assumes that the chip was initialized as described above. The initiator instructions are implemented using the SCRIPTS processor and SCRIPTS. The target instructions are implemented using the CPU and a “C” program.

When a SCRIPTS routine is executing, a waiting period is required to fetch the SCRIPTS instructions. This fetch time must be taken into account when writing the loopback code. To ensure proper operation, a delay should be inserted directly after SCRIPTS instructions have started executing. After the DSP register (0x2C–0x2F) is initialized with a SCRIPTS instruction address, the chip registers cannot be accessed

until the instruction has been fetched and begins executing. This delay time must include:

- Host arbitration
- SCRIPTS instruction fetch
- SCRIPTS instruction execution or internal bus moves

These delay times are system dependent due to host arbitration times, host bus width, and chip clock speed.

Figure 9.3 Loopback Mode

```
/*Load DSP with address of Select w/ATN instruction*/
/* SELECT ATN tar_id, REL(This_wont_occur) */
write_longreg (DSP,SCRIPTS_sel_inst);
/* Delay to allow instruction to be fetched by SIOP */
delay(1); /* 1 ms delay, varies with system*/
/* TARGET, wait for SEL to go high and BSY to go low */
while ((siop_reg[SBCL] & 0x30) != 0x10;
/*TARGET, check ID, but really don't care what it is */
printf("Initiator: Selecting target ID
%x\n",siop_reg[SBDL]);
/*TARGET, assert BSY*/
siop_reg[SOCL] = 0x20;
/*TARGET, wait for SEL to drop */
while ((siop_reg[SBCL] & 0x10) !=0);
```

In this section of code, the Initiator Select SCRIPTS routine is started by writing the address of the Select instruction to the DSP. A delay is inserted to ensure that the SIOP has time to fetch the instruction. Polling the SBCL register determines when SEL/ is active and selecting itself.

As shown in Figure 9.4, the variable `siop_reg` should be defined as a volatile pointer to the chip registers. This ensures that the registers are not shadowed internally by the CPU. Polling the SBDL register determines which SCSI ID bits are being driven. This is not a vital step in the loopback selection process, since the SCRIPTS processor is selecting itself. However, SBDL should be checked to make sure the correct bits are driven on the SCSI data bus during normal selection. The BSY/ bit is set in the SOCL register. This is a target operation performed by the CPU. Polling the SEL/ bit of the SBCL register determines when SEL/ is inactive. This indicates the initiator is properly responding to BSY/ being asserted by the target.

Figure 9.4 Target Operation

```
/*TARGET, check for ATN*/
if (siop_reg[SBCL] & 0x08) {
   /*TARGET, assert BSY, and MSG OUT*/
siop_reg[SOCL] = 0x26;
/*Self-Selection with ATN is now complete.*/
/* Wait for single step interrupt*/
while ((siop_reg[ISTAT] & 0x03) ==0);
/*Clear all interrupts*/
junk = siop_reg[SIST0];
junk = siop_reg[SIST1]
junk = siop_reg[DSTAT];
/*Start Script Block Move instruction*/
/*to send Identify Message to “Target” */
/*MOVE 1, identify_buf, WHEN MESSAGE OUT*/
siop_reg[DCNTL] |= 0x04;
/*Wait for SCRIPTS routine to finish using host bus*/
delay(1);
```

The program checks the SBCL register to determine if the selection is with or without SATN/. This effects the next phase asserted by the target. The desired phase is asserted by setting the MSG/, C_D/, and I/O bits in the SOCL register while maintaining BSY/. This would be MESSAGE OUT if SATN/ was sampled asserted or COMMAND if SATN/ was sampled deasserted in the SBCL register. At this point, selection with ATN/ is now complete. The SIP and DIP bits in the ISTAT register are polled for a single step interrupt and any others that may have occurred. Reading the SIST0, SIST1 and DSTAT registers clears these interrupts. The single step interrupt is cleared by reading the DSTAT register. Other interrupts may occur, depending on the particular settings in the SIEN and DIEN registers. You can safely clear all interrupts, as any pending interrupts would inhibit the execution of further SCRIPTS instructions. The example in Figure 9.4 uses a polled interrupt procedure. Hardware interrupts are handled in an interrupt service routine.

The Start DMA operation bit of the DCNTL register is set so that the Block Move SCRIPTS instruction begins execution. This Block Move instruction transfers the identify message associated with Selection with ATN/ to the target. A delay is inserted to ensure that the processor has time to fetch the instruction.

The next section of code, Figure 9.5, uses loopback mode to transfer bytes. Although this example can be used with the rest of the sample

code, it can also be used as a separate function. It can also be used for any generic data transfer between the initiator and the target, whenever the processor is executing a Block Move instruction.

Figure 9.5 Byte Transfer

```
/*TARGET, Get Message Byte */
/*TARGET, assert REQ, maintain all other SCSI signals*/
siop_reg[SOCL] |=0x80;
/*TARGET, wait for ACK*/
while ((siop_reg[SBCL] & 0x40) !=0)
msg_out_buf = siop_reg[SBDL]; /*read the data bus*/
siop_reg[SOCL] &=0x7f; /*deassert REQ*/
while ((siop_reg[SBCL] & 0x40) !=0) /* wait for ACK*/
/* verify message byte */
if (msg_out_buf !=identify_buf) {
loop_err = 1;
}
```

Assertion of the SREQ/ signal is the first step performed by this code. SREQ/ is asserted by keeping the phase bits the same and setting the SREQ/ bit in the SOCL register. This works for an initiator to target data transfer (DATA OUT, MESSAGE OUT, or COMMAND phase). To transfer from the target to the initiator (DATA IN, MESSAGE IN, or STATUS phase) place the data into the SODL register before asserting SREQ/. Because the processor clocks asynchronous data in on the rising edge of SACK/, data corruption results if this procedure is not followed. If SREQ/ is asserted, the processor immediately asserts ACK/ and clocks in the data in the SOCL register. If the data has not been placed into the SOCL register then incorrect data will be clocked in.

After asserting SREQ/, the initiator polls the SBCL register for SACK/ assertion. Subsequently, the target reads the SBDL register. It also deasserts SREQ/ using the SOCL register and polls the SBCL register for SACK/ deassertion of SACK/ by the initiator. The byte received by the target is verified with the byte sent by the initiator.

The code section in Figure 9.6 shows the final step of the selection procedure in the Loopback Mode. This selection procedure could be placed into a function, as could procedures that implement command, status, message in, and data transfer phases. Upon doing this, full SCSI sequences could be implemented in the Loopback Mode by various function calls in the proper order.

Figure 9.6 Loopback Mode Selection Procedure

```
else{ /*select without ATN*/
   printf("Initiator: Selecting without ATN.../n);
}
/*assert BSY and Command phase*/
siop_reg[SOCL} = 0x22;
/*wait for single step int.*/
while ((siop_reg[ISTAT] & 0x03) == 0);
/* clear all interrupts */
junk = siop_reg[SIST0];
junk = siop_reg[SIST1];
junk = siop_reg[DSTAT];
/*SELECTION COMPLETE*/
```

Selection without ATN/, requires only assertion of the next phase and waiting for a single step interrupt. The MSG/, C_D/, and I_O/ signals are set to the command phase using the SOCL register. BSY/ is also kept asserted. The SIP and DIP bits in the ISTAT register are polled for a single step interrupt and any other interrupts that may have occurred. These interrupts are cleared by reading the SIST1, SIST0, and DSTAT registers. The single step interrupt is cleared by reading the DSTAT register, but depending on the settings in the SIEN and DIEN registers, other interrupts may occur. You can safely clear all interrupts, as any pending interrupts would inhibit the execution of remaining SCRIPTS instructions. The example uses a polled interrupt procedure. If hardware interrupts are used then this would be handled in an interrupt service routine. After code execution, the chip is in a state to transfer command bytes using the generic byte transfer code given earlier.

9.3 Byte Recovery on Target Disconnect

There are three potential instances of disconnect. The first is during a Data Read, when a SCSI device may disconnect while it is seeking the data that was requested. This is very common, occurring when a SCSI disk drive performs a seek operation. Seeks often take many milliseconds and it is inefficient for the disk drive to stay active on the bus while transferring nothing. The second case may occur after a SCSI device completes a write operation and disconnects to empty its buffers before returning its status and command complete messages.

The third type of disconnect may occur at any time. It occurs when data is being written to a SCSI device and its internal buffers become full. The device disconnects before completing the data transfer to empty its buffers and avoid an overflow condition. When it does, the SCSI bus is in a different phase from that expected by the initiator, creating a phase mismatch. When this happens, the processor interrupts and the CPU must perform byte recovery. When this type of disconnect occurs, data may be in transition; it is important to determine how much data and its location. In addition, you must know where in the SCRIPTS program the transfer was interrupted so that it can be resumed at a later time. To save the state of the chip at the time of the disconnect, get the address of the current SCRIPTS instruction and calculate the number of bytes of active data remaining to be transferred. After saving the state of the chip, update the SCRIPTS program and flush or clear the FIFO.

9.3.1 Saving the Processor State

The first step in saving the state of the SCRIPTS processor is to write the address of the current SCRIPTS instruction from the DSP register to a special table indexed by SCSI ID. The instruction at that address can be restored later to resume processing. The DSP increments as the current instruction is fetched, so it always points to the next instruction. Therefore, the DSP decrements by 8 or 12, depending on whether the instruction was a regular SCRIPTS instruction or a Memory-to-Memory Move. This is determined by reading the DCMD register. Typically, the instruction is a Block Move. If table indirect addressing is used, it may only be necessary to update the table and not the SCRIPTS code.

Target disconnect may create a need to recover bytes in the chip's data paths. The location of the data is dependent on whether data is being moved into or out of the chip, and whether SCSI data is being transferred asynchronously or synchronously. Please consult the appropriate product technical manual for exact information on the default and extended (when supported) DMA FIFO sizes.

Saving the processor state for each type of SCSI transfer is described in the following sections.

9.3.1.1 Asynchronous SCSI Send

If the DMA FIFO size is set to the default size, check the DFIFO and DBC registers and calculate if there are bytes left in the DMA FIFO. To

make this calculation, subtract the seven least significant bits of the DBC register from the 7-bit value of the DFIFO register. AND the result with 0x7F for a byte count between zero and the FIFO size.

If the DMA FIFO size is set to the extended size, subtract the 10 least significant bits of the DBC register from the 10-bit value of the DMA FIFO Byte Offset Counter, which consists of bits [1:0] in the CTEST5 register and bits [7:0] of the DMA FIFO register. AND the result with 0x3FF for a byte count between zero and the extended FIFO size.

Read bit 5 in the SSTAT0 and SSTAT2 registers to determine if any bytes are left in the SODL register. If bit 5 is set in the SSTAT0 or SSTAT2 register then the least significant byte or the most significant byte in the SODL register is full, respectively. Checking this bit also reveals bytes left in the SODL register from a Chained Move operation with an odd byte count.

9.3.1.2 Synchronous SCSI Send

If the DMA FIFO size is set to the default size, look at the DFIFO and DBC registers and calculate if there are bytes left in the DMA FIFO. To make this calculation, subtract the seven least significant bits of the DBC register from the 7-bit value of the DFIFO register. AND the result with 0x7F for a byte count between zero and the FIFO size.

If the DMA FIFO size is set to the extended size, subtract the 10 least significant bits of the DBC register from the 10-bit value of the DMA FIFO Byte Offset Counter, which consists of bits [1:0] in the CTEST5 register and bits [7:0] of the DMA FIFO register. AND the result with 0x3FF for a byte count between zero and the FIFO size.

Read bit 5 in the SSTAT0 and SSTAT2 registers to determine if any bytes are left in the SODL register. If bit 5 is set in the SSTAT0 or SSTAT2 register then the least significant byte or the most significant byte in the SODL register is full, respectively. Checking this bit also reveals bytes left in the SODL register from a Chained Move operation with an odd byte count.

Read bit 6 in the SSTAT0 and SSTAT2 registers to determine if any bytes are left in the SODR register. If bit 6 is set in the SSTAT0 or SSTAT2 register then the least significant byte or the most significant byte in the SODR register is full, respectively.

9.3.1.3 Asynchronous SCSI Receive

If the DMA FIFO size is set to the default size, check the DFIFO and DBC registers and calculate if there are bytes left in the DMA FIFO. To make this calculation, subtract the seven least significant bits of the DBC register from the 7-bit value of the DFIFO register. AND the result with 0x7F for a byte count between zero and the FIFO size.

If the DMA FIFO size is set to the extended size, subtract the 10 least significant bits of the DBC register from the 10-bit value of the DMA FIFO Byte Offset Counter, which consists of bits [1:0] in the CTEST5 register and bits [7:0] of the DMA FIFO register. AND the result with 0x3FF for a byte count between zero and the FIFO size.

Read bit 7 in the SSTAT0 and SSTAT2 registers to determine if any bytes are left in the SIDL register. If bit 7 is set in the SSTAT0 or SSTAT2 register then the least significant byte or the most significant byte is full, respectively.

If any wide transfers have been performed using the Chained Move instruction, read the Wide SCSI Receive bit (SCNTL2, bit 0) to determine whether a byte is left in the SWIDE register.

9.3.1.4 Synchronous SCSI Receive

If the DMA FIFO size is set to the default size, subtract the seven least significant bits of the DBC register from the 7-bit value of the DFIFO register. AND the result with 0x7F for a byte count between zero and the FIFO size.

If the DMA FIFO size is set to the extended size, subtract the 10 least significant bits of the DBC register from the 10-bit value of the DMA FIFO Byte Offset Counter, which consists of bits [1:0] in the CTEST5 register and bits [7:0] of the DMA FIFO register. AND the result with 0x3FF for a byte count between zero and the FIFO size.

Read the SSTAT1 register, bit 4 of the SSTAT2 register for extended FIFO size and the binary representation of the number of valid bytes in the SCSI FIFO, to determine if any bytes are left in the SCSI FIFO.

If any wide transfers have been performed using the Chained Move instruction, read the Wide SCSI Receive bit (SCNTL2, bit 0) to determine whether a byte is left in the SWIDE register.

9.3.2 Updating the SCRIPTS Program

After calculating the number of bytes in transition, you update the SCRIPTS instruction so that the correct number of bytes are transferred when the target reselects. This is done by updating the byte count and address in the SCRIPTS program at wherever the current instruction was at the time of disconnect. The SCRIPT is stored in the host's main memory, so you can modify it at any time. You modify the binary version of the instruction in host memory unless table indirect addressing is used. If using Table Indirect Mode, you modify the byte count and address in the data structure instead of the binary version of the instruction.

9.3.3 Cleaning Up

Bytes that are already in transition must be processed. Depending on the direction of transfer and how you write the code, any data left in the chip must be flushed to memory (SCSI Receive only) or cleared and discarded. The Flush DMA FIFO bit in the CTEST3 register flushes the DMA FIFO data to memory. The Clear DMA FIFO bit in CTEST3 discards the data in the DMA FIFO. The Clear SCSI FIFO bit in STEST3 clears the data out of the Synchronous SCSI Receive FIFO and clears data in any other intermediate registers.

In a normal disconnect situation, when a Phase Mismatch interrupt occurs during a SCSI receive, no data should be left in the chip except in the SWIDE register.

Note: The Wide SCSI Send and Wide SCSI Receive bits are cleared by any nonwide send or receive action, such as moving message bytes. Examine these bit values first during byte recovery.

9.3.4 Example Byte Recovery Code

Byte recovery must be done when the SCRIPTS processor receives a phase mismatch interrupt either during the Data In or Data Out phase. Figure 9.7 is example code for moving data. Figures 9.8 and 9.9 are two example functions which handle these situations.

Figure 9.7 SCRIPTS Sequence to Move Data

```
Move_Data:
JUMP REL(RW_Offset_patch_di), WHEN DATA_IN
;During a write command, some devices disconnect after all the
;data has been sent and reselect with Status and msg_in. The
;following instructions prevents phase mismatch when this
;happens.
JUMP REL(RW_Handle_Phase) WHEN NOT DATA_OUT
; *** Script move data out ENTRY
RW_Offset_patch_do:
;Relative offset will be changed so that we jump
;into the proper place in the scatter gather list
JUMP REL(Data_Out_xfer); Data_Out_xfer:
CHMOV FROM data_buf1, WHEN DATA_OUT CHMOV FROM data_buf2, WHEN DATA_OUT
; 16 moves to support Scatter Gather
CHMOV FROM data_buf3, WHEN DATA_OUT
CHMOV FROM data_buf4, WHEN DATA_OUT
CHMOV FROM data_buf5, WHEN DATA_OUT
CHMOV FROM data_buf6, WHEN DATA_OUT
CHMOV FROM data_buf7, WHEN DATA_OUT
CHMOV FROM data_buf8, WHEN DATA_OUT
CHMOV FROM data_buf9, WHEN DATA_OUT
CHMOV FROM data_buf10, WHEN DATA_OUT
CHMOV FROM data_buf11, WHEN DATA_OUT
CHMOV FROM data_buf12, WHEN DATA_OUT
CHMOV FROM data_buf13, WHEN DATA_OUT
CHMOV FROM data_buf14, WHEN DATA_OUT
CHMOV FROM data_buf15, WHEN DATA_OUT
CHMOV FROM data_buf16, WHEN DATA_OUT

; Check to see if we need more SG list entries
;In older LSI53C8XX chips, SBR = DWT
MOVE SBR & RW_NEED_MORE_SG_ENTRIES to SFBR
INT RW_Need_More_SG, if not 0
; If we are here then all the data was transferred
; so we set a flag to indicate that
MOVE SBR | RW_ALL_DATA_TRANSFERRED to DWT
JUMP REL(RW_Handle_Phase)
; *** Script move data ENTRY
RW_Offset_patch_di:
;Relative offset will be changed so that we jump into the
;proper place in the scatter gather list
JUMP REL(Data_In_xfer); Data_In_xfer:
CHMOV FROM rw_data_buf1, WHEN DATA_IN CHMOV FROM rw_data_buf2, WHEN DATA_IN
; 16 moves to support Scatter Gather
CHMOV FROM rw_data_buf3, WHEN DATA_IN
CHMOV FROM rw_data_buf4, WHEN DATA_IN
CHMOV FROM rw_data_buf5, WHEN DATA_IN
CHMOV FROM rw_data_buf6, WHEN DATA_IN
CHMOV FROM rw_data_buf7, WHEN DATA_IN
CHMOV FROM rw_data_buf8, WHEN DATA_IN
CHMOV FROM rw_data_buf9, WHEN DATA_IN
```

```
CHMOV FROM rw_data_buf10, WHEN DATA_IN
CHMOV FROM rw_data_buf11, WHEN DATA_IN
CHMOV FROM rw_data_buf12, WHEN DATA_IN
CHMOV FROM rw_data_buf13, WHEN DATA_IN
CHMOV FROM rw_data_buf14, WHEN DATA_IN
CHMOV FROM rw_data_buf15, WHEN DATA_IN
CHMOV FROM rw_data_buf16, WHEN DATA_IN

; Check to see if we need more SG list entries
MOVE SBR & RW_NEED_MORE_SG_ENTRIES to SFBR
INT RW_Need_More_SG, if not 0
; If we are here then all the data was transferred
; so we set a flag to indicate that
MOVE SBR | RW_ALL_DATA_TRANSFERRED to DWT
JUMP REL(RW_Handle_Phase)
; *** Script move SWIDE byte ENTRY
RW_Move_swide_byte:
CHMOV 1, RW_Last_di_byte_buf, WHEN DATA_IN
INT RW_SWIDE_byte_moved
```

Figure 9.8 Example Function for Handling DATA IN Phase Mismatch Interrupts

```
/*******************************************************
 Function: HandleDataInPM
    Purpose : To handle clean up after a Phase Mismatch (PM)
                          during Data In phase
    Input:  The IO Base address of the SCSI chip
            A pointer to a variable which will indicate the
            Scatter Gather entry that was executing when the
            PM occurred, this is needed by the upper function
            if there was a byte in the SWIDE register.
    Output: Current_SG_Entry is filled in with the SG
                          entry that was being serviced.
    Assumptions:  That a phase mismatch has actually
                          occurred during data in.
    Restrictions: None
    Other functions called: IORead32 to read chip info
                                                    iowrite32 to start the script
    Global Variables Used:FirstDIMove_paddr is the
                                        physical address of the first Data In
                                        block move in the scatter/gather
                                        list. This is needed to get the
                                        location of the scatter/gather entry
                                        that was being serviced when the
                                        phase mismatch occurred.
                                  dsa_table is the table indirect table
                                        that is being used for this IO
                                  script is the actual script that was
                                        being executed when the phase
                                        mismatch occurred.
                            DATA_BUF1 is the offset into the Table
                                        Indirect entries for the first Data
                                        In table entry.
*******************************************************/
```

```
static void HandleDataInPM(ULONG PCIDeviceIOBase, INT\ *Current_SG_Entry)
{
    ULONG Current_DSP; /* Holds current DSP value */
    /* where am I in the SG list? */
    Current_DSP = IORead32(PCIDeviceIOBase+DSP) - 8;
    *Current_SG_Entry = (VINT) (Current_DSP -\ FirstDIMove_paddr) / 8;
    /* On Data In phase mismatch interrupts the part is automatically flushed so there
is no need to check for residual data in the part, except for data in the SWIDE byte*/
    /* now update the address and count */
    dsa_table[DATA_BUF1 + *Current_SG_Entry].address +=
           dsa_table[DATA_BUF1 + *Current_SG_Entry].count -
           (IORead32(PCIDeviceIOBase+DBC) & 0x00FFFFFFl);
    dsa_table[DATA_BUF1 + *Current_SG_Entry].count =
           IORead32(PCIDeviceIOBase+DBC) & 0x00FFFFFFl;
    /* update the jump offset into the SG list */
    script[(INT) (Ent_RW_Offset_patch_di/4) + 1] =
           (ULONG) *Current_SG_Entry * 8;
    /* move the byte in SWIDE if necessary */
    if (IORead8(PCIDeviceIOBase+SCNTL2) & 0x01)
    {
           /* patch move to get byte out of chip */
           script[(INT) E_RW_Last_di_byte_buf_Used[0]] =
                  buffer_table[DATA_BUF1 +
                  *Current_SG_Entry].address;
           /* start script to move byte */
           iowrite32(PCIDeviceIOBase+DSP,
                  getPhysAddr(rw_script) +
                  Ent_RW_Move_swide_byte);
    }
    else    /* nothing in swide so start the disconnect
                        /*script */
           iowrite32(PCIDeviceIOBase+DSP,
                  getPhysAddr(rw_script) + Ent_RW_Handle_Phase);
}
```

Figure 9.9 Example Function for Handling DATA OUT Phase Mismatch Interrupts

```
/*********************************************************
 Function: HandleDataOutPM
    Purpose: To handle clean up after a Phase Mismatch (PM) during Data Out phase
    Input: A pointer the pcidev_record.
    Output: None
    Assumptions: That a phase mismatch has actually
                          occurred during data out.
    Restrictions: None
    Other functions called:IORead32/8 to read chip info
                          RMWon to set bits in chip registers
                          iowrite32 to start the script
    Global Variables Used:FirstDOMove_paddr is the
                                          physical address of the first Data
                                          Out block move in the scatter/gather
                                          list. This is needed to get the
                                          location of the scatter/gather entry
```

```
                                    that was being serviced when the
                                    phase mismatch occurred.
                            dsa_table is the table indirect table that
                                    is being used for this IO
                            script is the actual script that was being
                                    executed when the phase mismatch
                                    occurred.
                            DATA_BUF1 is the offset into the Table
                                    Indirect entries for the first Data
                                    In table entry.
*****************************************************/
static void HandleDataOutPM(pcidev_record *PCIDevice)
{
    ULONG Current_DSP;/* holds current dsp value */
    INT Current_SG_Entry;/* Used to calc. Current SG entry */
    UINT DFIFO_val;       /* Holds chip DFIFO value */
    UINT Bytes_remaining;/* Used to account for other bytes in chip */
    /* where am I in the SG list? */
    Current_DSP = IORead32(PCIDeviceIOBase+DSP) - 8;
    Current_SG_Entry = (INT) (Current_DSP -
FirstDOMove_paddr) / 8;
    /* now update the address and count */
    buffer_table[DATA_BUF1 + Current_SG_Entry].address +=
      buffer_table[DATA_BUF1 + Current_SG_Entry].count -
            (IORead32(PCIDeviceIOBase+DBC) & 0x00FFFFFFl);
    buffer_table[DATA_BUF1 + Current_SG_Entry].count =
            IORead32(PCIDeviceIOBase+DBC) & 0x00FFFFFFl;
    /* Update count and address to reflect any data left in the chip */
    /* First check for data in the DMA FIFO */
    /* The variable DFIFO_val is a combination of bits
    /*1-0 of CTEST5 and bits 7-0 of the DFIFO register
    /*this will take care of both the extended FIFO devices
    /*and all others */
    DFIFO_val = ((IORead8(PCIDeviceIOBase+CTEST5) & 0x03) << 8) |
                                    (IORead8(PCIDeviceIOBase+DFIFO));
    if (IORead8(PCIDeviceIOBase+CTEST5) & 0x20)/* big fifo */
            Bytes_remaining = (DFIFO_val - (UINT)
                    IORead32(PCIDevice->base_addr2+DBC) & 0x3FF) &
                    0x3FF;
    else    /* default FIFO size*/
            Bytes_remaining = (DFIFO_val - (UINT)
                    IORead32(PCIDevice->base_addr2+DBC) & 0x7F) &
                    0x7F;
    /* now check the other regs that may contain data*/
    /* SODL LSB Full?*/
    if (IORead8(PCIDevice->base_addr2+SSTAT0) & 0x20)
            Bytes_remaining++;
    /* SODL MSB Full?*/
    if (IORead8(PCIDevice->base_addr2+SSTAT2) & 0x20
            ) Bytes_remaining++;
    /* SODR LSB Full?*/
    if (IORead8(PCIDevice->base_addr2+SSTAT0) & 0x40)
            Bytes_remaining++;
```

```
    /* SODR MSB Full?*/
    if (IORead8(PCIDevice->base_addr2+SSTAT2) & 0x40)
           Bytes_remaining++;
    /* Now update the TI entry */
    rw_buffer_table[RW_DATA_BUF1 +
    Current_SG_Entry].address -= Bytes_remaining;
    rw_buffer_table[RW_DATA_BUF1 +
    Current_SG_Entry].count += Bytes_remaining;
    /* update the jump offset into the SG list */
    rw_script[(INT) (Ent_RW_Offset_patch_do/4) + 1] =
    (ULONG) Current_SG_Entry * 8;
    /*clear the dma fifo to get any left over data out */
    RMWon(PCIDevice->base_addr2+CTEST3, 0x04);
    /* start the disconnect script */
    iowrite32(PCIDeviceIOBase, getPhysAddr(rw_script) +
    Ent_RW_Handle_Phase);
}
```

9.4 Synchronous Negotiation and Transfer

The SCRIPTS processor negotiates a set of parameters for each synchronous device on the SCSI bus. The parameters for each SCSI device are saved in memory and reloaded into the registers before communication resumes between the set of devices. A sample synchronous negotiation SCRIPTS program is supplied in Appendix B, "Multithreaded SCRIPTS Example." After the target receives acceptable synchronous parameters during the Message In phase, an interrupt returns control to the interrupt service routine. This programs the clock dividers and the synchronous parameters in the SCTNL3 and SXFER registers. These parameters are saved for this synchronous device.

When this device is selected again, you can use the SELECT FROM command to indicate table indirect addressing. If table indirect addressing is used, the SCNTL3, SDID, and SXFER registers are loaded from the table entry. If the device reselects the initiator, reload these parameters into the registers before the data transfer begins. One method for loading them is the table indirect Select instruction with the alternate address jump programmed to the next instruction. This instruction must be executed after determining the ITLQ nexus and loading the DSA to point to the proper I/O data structure. Example code for these steps is shown in Figure 9.10.

Figure 9.10 SELECT FROM Example Code

```
;ITLQ nexus complete and DSA loaded prior to performing
;this Select
SELECT FROM SCSI ID, REL(Next_Instr)
Next_Instr:
;begin I/O
```

The negotiated transfer information is stored in a table for use in later connections to a particular target. This information can be stored in the DSA table for use with table indirect Select and Reselect SCRIPTS instructions. The I/O command structure must have all four bytes contiguous in system memory, as shown below.

SCNTL3	SDID	SXFER	SCNTL4[1]

1. LSI53C10XX only.

9.5 Interrupt Handling

The SCRIPTS processor performs most functions independently of the host microprocessor. However, certain interrupt situations must be handled by the external microprocessor. This section explains all aspects of this type of interrupt.

9.5.1 Polling and Hardware Interrupts

There are two potential methods for informing the external microprocessor of an interrupt condition: polling or hardware interrupts. With polling the microprocessor continually loops and reads a register until it detects a set bit, indicating an interrupt. This method is faster, but it wastes CPU time that could be used for other system tasks. Hardware interrupts are the preferred method of detecting interrupts in most systems. In this case, the SCRIPTS processor asserts the Interrupt Request (IRQ/) line. Then an interrupt condition occurs, causing the microprocessor to execute an interrupt service routine. A hybrid approach can also be used that would use hardware interrupts for long waits, and polling for short waits.

9.5.2 Registers

The registers that are used for detecting or defining interrupts are the ISTAT, SIST0, SIST1, DSTAT, SIEN0, SIEN1, and DIEN.

9.5.2.1 ISTAT

Note: LSI53C896 and newer chips have two ISTAT registers. Refer to your chip technical manual for specific information regarding ISTAT. If your chip has two ISTAT registers, the instructions below refer to ISTAT0.

ISTAT registers are the only registers that can be accessed as slaves during SCRIPTS operation. Therefore, they are the registers polled when polled interrupts are used. It is also the first register that should be read when the IRQ/ pin has been asserted in response to a hardware interrupt. The INTF (Interrupt on the Fly) bit should be the first interrupt serviced. It must be written to one that is to be cleared. This interrupt must be cleared before servicing any other interrupts. If the SIP bit in the ISTAT register is set, then a SCSI type interrupt has occurred and the SIST0 and SIST1 registers should be read. If the DIP bit in the ISTAT register is set, then a DMA type interrupt has occurred and the DSTAT register should be read. SCSI type and DMA type interrupts may occur simultaneously, so in some cases both SIP and DIP may be set.

9.5.2.2 ISTAT1

Note: LSI53C896 and newer only.

This register contains two read-only bits, FLSH and SRUN. When set, these bits indicate whether the chip is flushing data from the DMA FIFO and if the SCRIPTS engine is currently fetching and executing SCRIPTS instructions, respectively. Writes do not affect the value of these bits. The other nonreserved bit in this register is SI, the synch interrupt disable bit. Setting this bit disables the INTA/ pin for Function A and the INTB/ pin for Function B. Clearing this bit enables normal operation of the INTA/ (or INTB/) pin. If the INTA/ (or INTB/) is already asserted and this bit is set, INT remains asserted until the interrupt is serviced. At this point the interrupt line is blocked for future interrupts until this bit is cleared. In addition, this bit may be read and written while SCRIPTS are executing.

9.5.2.3 SIST0 and SIST1

The SIST0 and SIST1 registers contain the SCSI type interrupt bits. Reading these registers determines which condition or conditions caused the SCSI type interrupt and clears that SCSI interrupt condition. If the chip is receiving data from the SCSI bus and a fatal interrupt condition occurs, the SCRIPTS processor attempts to send the contents of the DMA FIFO to memory before generating the interrupt. If the processor is sending data to the SCSI bus and a fatal SCSI interrupt condition occurs, data could be left in the DMA FIFO. Under these circumstances, check the DMA FIFO Empty (DFE) bit in DSTAT. If this bit is cleared, set the CLF (Clear DMA FIFO) and CSF (Clear SCSI FIFO) bits before continuing. The CLF bit is bit 2 in CTEST3. The FLF bit is bit 3 in CTEST3. The CSF bit is bit 1 in STEST3.

9.5.2.4 DSTAT

The DSTAT register contains the DMA type interrupt bits. Reading this register determines which condition or conditions caused the DMA type interrupt, and clears that DMA interrupt condition. Bit 7 in DSTAT, DFE (DMA FIFO Empty), is purely a status bit. This bit does not generate an interrupt under any circumstances and is not cleared when read. DMA interrupts do not flush either the DMA or SCSI FIFOs before generating the interrupt, so the DFE bit in the DSTAT register should be checked after any DMA interrupt. If the DFE bit is cleared, then the FIFOs must be cleared by setting the CLF (Clear DMA FIFO) and CSF (Clear SCSI FIFO) bits, or flushed by setting the FLF (Flush DMA FIFO) bit.

9.5.2.5 SIEN0 and SIEN1

The SIEN0 and SIEN1 registers are the interrupt enable registers for the SCSI interrupts in the SIST0 and SIST1 registers.

9.5.2.6 DIEN

The DIEN register is the interrupt enable register for DMA interrupts in DSTAT.

9.5.2.7 DCNTL (All chips except the LSI53C770, LSI53C810, LSI53C815, and LSI53C860)

When bit 1 in this register is set, the IRQ/ pin is not asserted when an interrupt condition occurs. The interrupt is not lost or ignored, but merely masked at the pin. Clearing this bit when an interrupt is pending immediately asserts the IRQ/ pin. As with any register other than ISTAT, this register cannot be accessed except by a SCRIPTS instruction during SCRIPTS execution.

9.5.3 Fatal vs. Nonfatal Interrupts

A fatal interrupt, as the name implies, always stops SCRIPTS execution. All nonfatal interrupts become fatal when they are enabled by setting the appropriate interrupt enable bit. All DMA interrupts are fatal. Interrupt masking is discussed in Section 9.5.4, “Masking.”

Some SCSI interrupts, as indicated by the SIP bit in the ISTAT register and one or more bits in the SIST0 or SIST1 register being set, are nonfatal. When the SCRIPTS processor is operating in the Initiator Mode, only the Function Complete (CMP), Selected (SEL), Reselected (RSL), General Purpose Timer Expired (GEN), and Handshake-to-Handshake Timer Expired (HTH) interrupts are nonfatal. When operating in the Target Mode, CMP, SEL, RSL, Target Mode: SATN/ active (M/A), GEN, and HTH are nonfatal. Refer to the description for the Disable Halt on a Parity Error or SATN/ active (Target Mode only) (DHP) bit in the SCNTL1 register to configure the chip’s behavior when the SATN/ interrupt is enabled during Target Mode operation. The Interrupt on the Fly interrupt is also nonfatal, since SCRIPTS can continue when it occurs.

Nonfatal interrupts allow continued SCRIPTS operation when an interrupt occurs that does not require service from the CPU. This prevents an interrupt when:

- Arbitration is complete (CMP set)
- The SCRIPTS processor has been selected or reselected (SEL or RSL set)
- The initiator has asserted SATN/ (Target Mode: SATN/ active)
- General Purpose or Handshake-to-Handshake timers expire

These interrupts are not needed for events that occur during high level SCRIPTS operation.

9.5.4 Masking

Masking an interrupt means disabling or ignoring that interrupt. Interrupts can be masked by clearing bits in the SIEN0 and SIEN1 (for SCSI interrupts) interrupt enable registers or the DIEN (for DMA interrupts) interrupt enable register. How the chip responds to masked interrupts depends on: whether polling or hardware interrupts are being used; whether the interrupt is fatal or nonfatal; and whether the chip is operating in Initiator or Target Mode.

If a nonfatal interrupt is masked and that condition occurs, SCRIPTS:

- Continues execution
- Sets the appropriate bit in the SIST0 or SIST1 register
- Does not set the SIP bit in the ISTAT
- Does not assert the IRQ/ pin

See Section 9.5.3, “Fatal vs. Nonfatal Interrupts,” for a list of the nonfatal interrupts.

If a fatal interrupt is masked and that condition occurs, then SCRIPTS stops execution, sets the appropriate bit in the DSTAT, SIST0, or SIST1 registers, and sets the SIP or DIP bits in the ISTAT. The IRQ/ pin is not asserted.

When the chip is initialized, you must enable all fatal interrupts if you are using hardware interrupts. If a fatal interrupt is disabled and that interrupt condition occurs, SCRIPTS halts. The system will not detect this unless it times out and checks the ISTAT after a certain period of inactivity.

If the ISTAT register is being polled, instead of using hardware interrupts, then masking a fatal interrupt has no impact. The SIP and DIP bits in the ISTAT inform the system of interrupts, not the IRQ/ pin.

Masking an interrupt after IRQ/ is asserted will not deassert IRQ/.

9.5.5 Stacked Interrupts

The SCRIPTS processor stacks interrupts if they occur in rapid succession. If the SIP or DIP bits in the ISTAT register are set (first level), then at least one pending interrupt exists. Any future interrupts are stacked in extra registers behind the SIST0, SIST1, and DSTAT registers (second level). When two interrupts have occurred and the two levels of the stack are full, any further interrupts set additional bits in the extra registers behind the SIST0, SIST1, and DSTAT registers. When the first level of interrupts is cleared, the subsequent interrupts move into the SIST0, SIST1, and DSTAT registers. After clearing the first interrupt by reading the appropriate register, the IRQ/ pin deasserts for a minimum of three CLKs, the stacked interrupt(s) move into the SIST0, SIST1, or DSTAT registers, and the IRQ/ pin reasserts.

A masked nonfatal interrupt does not set the SIP or DIP bits. Therefore, interrupt stacking does not occur. A masked, nonfatal interrupt still posts the interrupt in the SIST0 register, but does not assert the IRQ/ pin. Since no interrupt is generated, subsequent interrupts move right into the SIST0 or SIST1 register instead of being stacked behind another interrupt. On generation of another interrupt, the bit corresponding to the earlier masked nonfatal interrupt remains set.

Two simultaneous interrupts cause a similar situation. Since stacking does not occur until the SIP or DIP bits are set, a small timing window exists in which multiple interrupts can occur. Under these circumstances, the interrupts are not stacked. These could be multiple SCSI interrupts (SIP set), multiple DMA interrupts (DIP set), or multiple SCSI and multiple DMA interrupts (both SIP and DIP set).

As previously mentioned, DMA interrupts do not attempt to flush the FIFOs before generating the interrupt. You must set either the Clear DMA FIFO (CLF) and Clear SCSI FIFO (CSF) bits if a DMA interrupt occurs and the DMA FIFO Empty (DFE) bit is not set. Any subsequent SCSI interrupts are not posted until the DMA FIFO is cleared of data. These ‘locked out’ SCSI interrupts are posted as soon as the DMA FIFO is empty.

9.5.6 Halting in an Orderly Fashion

When an interrupt occurs, the SCRIPTS processor attempts to halt in an orderly fashion.

- If it is in the middle of an instruction fetch, the fetch will be completed, except in the case of a Bus Fault. Execution will not begin, but the DSP points to the next instruction since it is updated when the current instruction is fetched.
- If the DMA direction is a write to memory and a SCSI interrupt occurs, the SCRIPTS processor attempts to flush the DMA FIFO to memory before halting. Under any other circumstances only the current cycle will be completed before halting, so the DFE bit in the DSTAT register should be checked to see if any data remains in the DMA FIFO.
- SCSI SREQ/SACK handshakes that have begun will be completed before halting.
- The SCRIPTS processor attempts to clean up any outstanding synchronous offsets before halting.
- In the case of Transfer Control Instructions, once execution begins it will not halt until completion.
- If the instruction is a JUMP/CALL WHEN/IF <phase>, the DSP is updated to the transfer address before halting.
- All other instructions may halt before completion.

9.5.7 Sample Interrupt Service Routine

The following is a sample of an interrupt service routine. It can be repeated if polling is used, or should be called when the IRQ/ pin is asserted if hardware interrupts are used.

1. Read ISTAT (or ISTAT0 as appropriate if your system has a newer chip).
2. If the INTF bit is set, write it to a one to clear this status.
3. If only the SIP bit is set, read the SIST0 and SIST1 registers to clear the SCSI interrupt condition and get the SCSI interrupt status.

   The bits in the SIST0 and SIST1 registers define the interrupt(s) and determine what action is required to service them.
4. If only the DIP bit is set, read the DSTAT register to clear the interrupt condition and get the DMA interrupt status.

   The bits in the DSTAT register define the interrupt(s) and determine what action is required to service them.

5. If both the SIP and DIP bits are set, read the SIST0, SIST1, and DSTAT registers to clear the SCSI and DMA interrupt condition and get the interrupt status.

   If using 8-bit reads of the SIST0, SIST1, and DSTAT registers to clear interrupts, insert a 12 CLK delay between the consecutive reads to ensure that the interrupts clear properly. Both the SCSI and DMA interrupt conditions should be handled before leaving the ISR. It is recommended that the DMA interrupt be serviced before the SCSI interrupt, because a serious DMA interrupt condition could influence how the SCSI interrupt is acted upon.

When using polled interrupts, go back to Step 1 before leaving the interrupt service routine, in case any stacked interrupts moved in when the first interrupt was cleared. When using hardware interrupts, the IRQ/ pin will be asserted again if there are any stacked interrupts. This should cause the system to re-enter the interrupt service routine.

9.6 Migrating Existing Software to Ultra, Ultra2, and Ultra3 SCSI

Current SCSI technology extends the Fast SCSI-2 specification to allow synchronous transfer periods to be negotiated down as low as 50 ns (Ultra), 25 ns (Ultra2/3). Ultra3 SCSI supports dual transition clocking for an effective period of 12.5 ns. This allows a maximum transfer rate of 20 Mbytes/s on an 8-bit SCSI bus or 40 Mbytes/s on a wide SCSI bus for Ultra SCSI, and 40 Mbytes/s on an 8-bit bus, 80 Mbytes/s on a wide SCSI bus for Ultra2 SCSI and 160 Mbytes/s for Ultra3. Refer to Chapter 1, “Using the Programming Guide,” to determine which chips support Ultra/2/3 SCSI.

To achieve transfer rates reflecting current SCSI specifications, existing software programs must be updated to reflect changes in the following areas of the SCRIPTS processor. Additional minor changes may be needed to migrate existing software to support all the features in the new device:

- SCNTL3 register CCF bits – adjust the bit values to reflect the desired clock divider (not the LSI53C1010 or LSI53C1010R).

- SCNTL3 register SCF bits – adjust the bit values to reflect the SCLK frequency, doubled or quadrupled if applicable.
- SXFER register XFERP bits – adjust the bit values to reflect the desired divider values for the synchronous period.
- Adjust the Clock input as required for the SCSI processor being used.
- With the LSI53C860, add an external 80 MHz SCSI clock.
- With the LSI53C875, use an 80 MHz external SCSI clock or use an external 40 MHz clock and enable the SCSI clock doubler.
- With the LSI53C895, use an 80 MHz clock for Ultra SCSI or use an external 40 MHz clock with the clock quadrupler for Ultra2 SCSI.
- The LSI53C885 and LSI53C876 require a 40 MHz clock and use of the clock doubler.
- Ultra Enable bit, SCNTL3 register – set this bit to enable Ultra SCSI or Ultra2 SCSI transfers.
- SCNTL4 U3EN bit set to enable Ultra3 (LSI53C10XX only).

9.6.1 Clock Divider Bits

Two registers divide down the clock. The first is the SCNTL3 register. Except for the Ultra3 chips, the CCF bits determine the SCSI core speed used for asynchronous transfers and any other timings (such as selection time-out). These bits are set based on the input clock frequency and do not change. The SCF bits determine the timing for synchronous transfers and can be changed whenever the SCRIPTS processor connects to a different device on the SCSI bus.

The SCF bits in the SCNTL3 register, in conjunction with the XFERP bits in the SXFER register, determine the synchronous period. To get a transfer rate of 10 Mbytes/s with a 40 MHz clock, program the SCF bits to 0b001 for a divide by one factor and then program the XFERP bits for 0b000 for a divide by 4 factor. Forty MHz divided by 1 and then divided by 4 is 10 Mbytes/s. Other combinations of these two sets of bits select a variety of synchronous transfer rates. For more information on the supported bit combinations, see the clock divider bit descriptions in your chip technical manuals.

The LSI53C10XX has only a 40 MHz clock with no dividers.

9.6.2 Ultra Enable Bit

The Ultra Enable bit (also known as the Fast-20 Enable bit) adjusts the chip's timing to be compliant with the Fast-20 proposed standard. It should be set when the synchronous transfer period is less than 100 ns and cleared when it is greater than or equal to 100 ns.

9.6.3 Loading the New Register Values

Since the Ultra Enable bit and the clock dividers are in the SCNTL3 and SXFER registers, these registers can be automatically loaded during a selection or reselection by using Table Indirect Addressing. This allows the chips to transparently talk with any combination of Ultra, Ultra2, Ultra3, and Fast SCSI devices on the same SCSI bus.

9.6.4 Negotiating Synchronous Transfers

The easiest way to calculate the synchronous transfer period is by multiplying the clock period by the clock divider values. For example, a 40 MHz clock is a 25 ns period. (25 ns) x (1) x (4) = 100 ns, which is the Fast SCSI-2 synchronous transfer period.

If you use an 80 MHz clock (12.5 ns period) and are negotiating for Fast SCSI-2, rather than Ultra SCSI, program the SCF bits for SCLK/2 and the XFERP bits for 4, the resulting period is (12.5 ns) x (2) x (4) = 100 ns.

The SCSI-2 specification states that synchronous transfer rates must be a multiple of 4 ns. However, with an 80 MHz clock, the period must be a multiple of 12.5 ns. Ultra SCSI is defined to be a 20 megatransfers per second maximum, which would be a 50 ns period. Since 50 ns is not a multiple of 4, most SCSI devices cannot negotiate for this exact rate. Unless future revisions of the standard make a different recommendation, most devices will probably negotiate for a 48 ns period. The SCRIPTS processor cannot be programmed for a 48 ns period since it is not a multiple of 12.5 ns. Therefore driver programs should specify a 50 ns period and the chip should negotiate for a 48 ns period. This is acceptable because the SCSI-2 specification allows data to be transferred at a slower rate than what is negotiated for, but not faster.

To program the chip for a full Ultra SCSI transfer rate of 50 ns using the required 80 MHz clock, program the SCF bits for SCLK/1 and select an XFERP of 4. This comes out to (12.5 ns) x (1) x (4) = 50 ns.

9.6.5 Using the SCSI Clock Doubler

The LSI53C875, LSI53C876, and LSI53C885 can double the frequency of a 40–50 MHz SCSI clock, allowing the system to perform Ultra SCSI transfers in systems that do not have 80 MHz clock input. This option is user selectable with bit settings in the STEST1, STEST3, and SCNTL3 registers. At power on or reset, the doubler is disabled and powered down. Follow these steps to use the clock doubler:

1. Set the SCLK Doubler Enable bit (STEST1, bit 3).
2. Wait 20 µs.
3. Halt the SCSI clock by setting the Halt SCSI Clock bit (STEST3, bit 5).
4. Set the clock conversion factor using the SCF and CCF fields in the SCNTL3 register.
5. Set the SCLK Doubler Select bit (STEST1, bit 2).
6. Clear the Halt SCSI Clock bit.

9.6.6 Using the SCSI Clock Quadrupler

The LSI53C895/895A/10XX can quadruple the frequency of a 40 MHz SCSI clock, allowing the system to perform Ultra2 SCSI transfers. This option is user selectable with bit settings in the STEST1, STEST3, and SCNTL3 registers. At power on or reset, the quadrupler is disabled and powered down. Use the following steps to use the clock quadrupler:

1. Set the SCLK Quadrupler Enable bit (STEST1, bit 3).
2. Poll bit 5 of the STEST4 register.

   The LSI53C895 sets this bit as soon as it locks in the 160 MHz frequency. The frequency lockup takes approximately 100 microseconds.
3. Halt the SCSI clock by setting the Halt SCSI Clock bit (STEST3, bit 5).
4. Set the clock conversion factor using the SCF and CCF fields in the SCNTL3 register.
5. Set the SCLK Quadrupler Select bit (STEST1, bit 2).
6. Clear the Halt SCSI Clock bit.

9.7 Using the SCRIPTS RAM

Many of the chips supported by the SCRIPTS processor contain internal, general purpose RAM. Please refer to your chip technical manual for your chip's specifications. This RAM stores SCRIPTS instructions and I/O data structure information, but is not limited to this type of information. When the chip fetches SCRIPTS instructions or Table Indirect information from the internal RAM, the fetches remain internal to the chip and do not use the PCI bus. Other types of access to the RAM by chip with internal RAM use the PCI bus, as if they were external accesses. This section discusses loading SCRIPTS and Table Indirect information into the SCRIPTS RAM and other programming techniques for using internal RAM.

The RAM can be relocated anywhere in the 32-bit address (64-bit in newer chips) space by the PCI system BIOS. The RAM Base Address register, located in the chip's PCI configuration space, contains the internal RAM base address. This register is similar to the ROM Base Address register in the PCI Configuration register set. To simplify loading SCRIPTS instructions, the RAM base address appears in the SCRATCHB register when bit 3 of the CTEST2 register is set. The RAM is byte accessible from the PCI bus and is visible to any bus mastering device on the bus. Accesses made externally, that is by the CPU, follow the same timing sequence as a standard slave register access, except that the required target wait states drops from 5 to 3.

9.7.1 Loading SCRIPTS RAM

SCRIPTS instructions can be loaded into the internal RAM in one of two ways. You can simply copy the instructions into the RAM with the CPU. Alternatively, you can use a MOVE MEMORY instruction, which copies the SCRIPTS instructions from their initial location in host memory to the SCRIPTS RAM. This method is especially useful in the Intel processor real mode of operation because the SCRIPTS RAM is generally mapped by the PCI system BIOS outside the region where the processor can access it. The syntax of the move instruction is:

```
MOVE MEM Script_Inst_Bytes, SRC_Phys_Addr, \
Script_RAM_Phys_Addr
```

Script_Inst_Bytes is the number of instruction bytes to copy and SRC_Phys_Addr is the physical starting address of the static SCRIPTS array being copied into SCRIPTS RAM. Script_RAM_Phys_Addr is the physical base address of the SCRIPTS RAM, found in the SCRATCHB register. To create data structures such as table indirect tables, create a pointer to the location in SCRIPTS RAM that stores the data. An example is shown in Figure 9.11.

Figure 9.11 Storing Data Structures in SCRIPTS RAM

```
struct _table {/* Table indirect entry */
  uquad count;
  uquad address;
};
typedef struct table;
#define SCRAM_TABLE_OFFSET 0xC00; /* Locate table info at bottom 1K of SCRIPTS RAM*/
void main() {
table *buffer_table;  /* pointer to table indirect entries */
ulong SCRAM_Phys_Addr;
ulong Table_Phys_Addr;
    /* Get RAM physical address */
outpw(ChipIOBase+CTEST2,  0x08);/* Set bit 3 */
/* Get RAM Base in ScratchB */
SCRAM_Phys_Addr = (ulong) ((ulong) (inpw(ChipBaseIO+
SCRATCHB2) << 16) | inpw(ChipIOBase+SCRATCHB));  /* Read Reg*/
outpw(ChipIOBase+CTEST2, 0x00);/* Clear bit 3 */
/* Create pointer to RAM for Table */
Table_Phys_Addr = SCRAM_Phys_Addr + SCRAM_Table_Offset;
buffer_table = PhystoVirt(Table_Phys_Addr);
}
```

The routine “PhystoVirt” converts the physical address of the table location in the SCRIPT RAM to a virtual address that can be used as a pointer in “C”.

9.7.2 Programming Techniques when Using SCRIPTS RAM

SCRIPTS programs may be stored on the chip, outside the chip, or both. When the SCRIPTS code is located both internally and externally, the following techniques allow the internal SCRIPTS to successfully communicate with the external SCRIPTS and vice versa.

1. Create two source (.SS) files, one with the SCRIPTS programs that are to be located internally and the other with the SCRIPTS programs that are to be located externally.

2. Give the internal and external SCRIPTS programs unique array identifiers by using the PROC statement at the beginning of each so that both can be linked into a driver.

   The compiler generates the SCRIPTS arrays without the default SCRIPTS name.

3. Compile both source files with the `-p` option instead of the `-o` option.

   This prevents generation of data structures which share common names between the two files, causing a 'C' compile time conflict with both files being linked into a driver.

4. Use absolute jumps between the internal and external SCRIPTS routines and use EXTERNs as the destination address variable.

   This patches the proper jump address after the base addresses of both SCRIPTS programs have been established at run time.

5. Define any labels being jumped to from the opposite SCRIPTS program as entry points with the ENTRY declarative.

   This causes the compiler to provide the proper offset information in the compiled output file so that physical addresses can be resolved at run time.

6. Assign unique names to all labels, externs, and relative buffers in each SCRIPTS program to prevent 'C' compile time conflicts.

7. Use the REL modifier to process all jumps that move within the same SCRIPTS program.

8. Use RAMFIX to process the file that contains the internal SCRIPTS program to eliminate any other conflicts between the two files.

   The RAMFIX utility can be downloaded from the LSI Logic BBS.

Figures 9.12 through 9.15 are the internal and external `SCRIPTS.LIS` and `.OUT` files, and illustrate the interactions between the two. Certain parts of the program text appear in bold type to highlight the coding differences when both internal and external RAM are used for SCRIPTS program storage. The numbered notes at the end of each example program reference numbered items in the far left column of the program text.

Figure 9.12 External Script (SCRIPTS.LIS file)

```
    1 ARCH 825A
    2
    3 ABSOLUTE done=0xff
    4
1: 5 EXTERN Int_Start
    6 EXTERN Int_dataout
    7
2: 8 ENTRY Ext_Start
   9 ENTRY Ext_done
   10
3: 11 00000000:          PROC Ext_Script:
   12 00000000:              Ext_Start:
   13 00000000: 78344500 00000000   MOVE 0x45 to SCRATCHA0
   14 00000008: 78354600 00000000   MOVE 0x46 to SCRATCHA1
   15 00000010: 80880000 00000018   JUMP REL(Entry_point)
   16 00000018: 78364700 00000000   MOVE 0x47 to SCRATCHA2
   17 00000020: 78374800 00000000   MOVE 0x48 to SCRATCHA3
4: 18 00000028: 80080000 00000000   JUMP Int_Start
   19
   20 00000030:                    Entry_point:
   21 00000030: 6A360000 00000000   MOVE SFBR to SCRATCHA2
   22 00000038: 785C0000 00000000   MOVE 0x00 to SCRATCHB0
   23 00000040: 785D0100 00000000   MOVE 0x01 to SCRATCHB1
   24 00000048: 785F0200 00000000   MOVE 0x02 to SCRATCHB3
5: 25 00000050: 80080000 00000000   JUMP Int_dataout
   26
   27 00000058:                    Ext_done:
   28 00000058: 98080000 000000FF   INT done
   29
   30
```

1. Jump labels that are located in the internal SCRIPTS program are defined as EXTERNs to facilitate patching at driver run time.
2. Labels that will be jumped to from the internal SCRIPTS program are defined as ENTRYs to facilitate patching at driver run time.
3. The PROC directive is used to override the default SCRIPTS array name and replace it with Ext_Script.
4. This is a jump to a location in the internal SCRIPTS program and should be patched at driver init time.
5. This is a jump to a location in the internal SCRIPTS program and should be patched at driver init time.

Figure 9.13 External Script (SCRIPTS.OUT file)

```
typedef unsigned long ULONG;
1:ULONGExt_Script[] = {
    0x78344500L,0x00000000L,
    0x78354600L,0x00000000L,
    0x80880000L,0x00000018L,
    0x78364700L,0x00000000L,
    0x78374800L,0x00000000L,
    0x80080000L,0x00000000L,
    0x6A360000L,0x00000000L,
    0x785C0000L,0x00000000L,
    0x785D0100L,0x00000000L,
    0x785F0200L,0x00000000L,
    0x80080000L,0x00000000L,
    0x98080000L,0x000000FFL
};
2:ULONG E_Int_dataout_Used[] = {
    0x00000015L
};
ULONG E_Int_Start_Used[] = {
    0x0000000BL
};
#define A_done0x000000FFL
3:#define Ent_Ext_done         0x00000058L
#define Ent_Ext_Start        0x00000000L
```

1. The use of the PROC statement has forced the array to be named Ext_Script instead of SCRIPT so that a compile time conflict is avoided.
2. The offsets in these data structures indicate where the internal SCRIPTS jump address should be patched.
3. These are the offsets into the Ext_Script array of the entry points that are being jumped to from the internal SCRIPTS program. They are used to calculate the internal to external jump physical addresses to be patched into the internal SCRIPTS program.

Figure 9.14 Internal Script (SCRIPTS.LIS file)

```
1       ARCH 825A
2
3       ABSOLUTE scsi_id=0x00
4       ABSOLUTE resel=0x01
5
6       EXTERN identify_buf={0x80}
```

```
   7        EXTERN cmd_buf=6{??}
   8        EXTERN data_buf=512{??}
   9        EXTERN stat_buf=1{??}
   10       EXTERN msgin_buf=1{??}
   11
1:12        EXTERN Ext_Start
   14
2:15        ENTRY Int_Start
   16       ENTRY Int_dataout
   17
3:18 00000000: PROC Int_Script:
   19 00000000: Int_Start:
   20 00000000: 45000000 00000058   SELECT ATN scsi_id, REL(reselected)
   21 00000008:                    ident:
   22 00000008: 86830000 00000008   JUMP REL(send_cmd), WHEN NOT MSG_OUT
   23 00000010: 0E000001 00000000   MOVE 1, identify_buf, WHEN MSG_OUT
   24 00000018:                    send_cmd:
   25 00000018: 0A000006 00000000   MOVE 6, cmd_buf, WHEN CMD
4:26 00000020: 80080000 00000000   JUMP Ext_Start
   27 00000028:                    Int_dataout:
   28 00000028: 08000200 00000000   MOVE 512, data_buf, WHEN DATA_OUT
   29 00000030:                    stat:
   30 00000030: 0B000001 00000000   MOVE 1, stat_buf, WHEN STATUS
   31 00000038:                    msgin:
   32 00000038: 0F000001 00000000   MOVE 1, msgin_buf, WHEN MSG_IN
   33
   34 00000040:                    complete:
   35 00000040: 7C027F00 00000000   MOVE SCNTL2 & 0x7F to SCNTL2
   36 00000048: 60000040 00000000   CLEAR ACK
   37 00000050: 48000000 00000000   WAIT DISCONNECT
5:38 00000058: 80080000 00000000   JUMP Ext_done
   39
   40 00000060:                    reselected:
   41 00000060: 98080000 00000001   INT resel
  42
```

1. Jump labels that are located in the external SCRIPTS program are defined as EXTERNs to facilitate patching at driver run time.
2. Labels that will be jumped to from the external SCRIPTS program are defined as ENTRYs to facilitate patching at driver run time.
3. The PROC directive is used to override the default SCRIPTS array name and replace it with Int_Script.
4. This is a jump to a location in the external SCRIPTS program and should be patched at driver init time.
5. This is a jump to a location in the external SCRIPTS program and should be patched at driver init time.

Figure 9.15 Internal SCRIPTS Program (SCRIPTS.OUT file)

```
typedef unsigned long ULONG;
1:ULONGInt_Script[] = {
    0x45000000L,0x00000058L,
    0x86830000L,0x00000008L,
    0x0E000001L,0x00000000L,
    0x0A000006L,0x00000000L,
    0x80080000L,0x00000000L,
    0x08000200L,0x00000000L,
    0x0B000001L,0x00000000L,
    0x0F000001L,0x00000000L,
    0x7C027F00L,0x00000000L,
    0x60000040L,0x00000000L,
    0x48000000L,0x00000000L,
    0x80080000L,0x00000000L,
    0x98080000L,0x00000001L
};
ULONG E_cmd_buf_Used[] = {
    0x00000007L
};
ULONG E_data_buf_Used[] = {
    0x0000000BL
};
2:ULONG E_Ext_Start_Used[] = {
    0x00000009L
};
ULONG E_Ext_done_Used[] = {
    0x00000017L
};
ULONG E_identify_buf_Used[] = {
    0x00000005L
};
ULONG E_msgin_buf_Used[] = {
    0x0000000FL
};
ULONG E_stat_buf_Used[] = {
    0x0000000DL
};
#define A_scsi_id0x00000000L
#define A_resel0x00000001L
3:
#define Ent_Int_dataout      0x00000028L
#define Ent_Int_Start        0x00000000L
```

1. The use of the PROC statement has forced the array to be named Int_Script instead of SCRIPT so that a compile time conflict is avoided.

2. The offsets in these data structures indicate where the internal SCRIPTS jump address should be patched.
3. These are the offsets into the Int_Script array of the entry points that are being jumped to from the external SCRIPTS program. They are used to calculate the external to internal jump physical addresses to be patched into the external SCRIPTS program.

9.7.3 Patching Internal and External SCRIPTS Programs

The routine in Figure 9.16 patches the correct values into the above two SCRIPTS programs so that they can interact properly. The following assumptions are made in this routine:

- The Int_Script array was copied into the SCRIPTS RAM at the starting location of the RAM.
- The Ext_Script is already 32-bit aligned.
- The variable ChipIOBase contains the IO base address of the chips register set.
- VirttoPhys is a routine that will convert a virtual pointer to a physical address.

Figure 9.16 Patching Routine

```
void main() {
ulong Int_Script_Phys_Addr;
ulong Ext_Script_Phys_Addr;

/* Get RAM physical address, which is assumed to be */
/* the internal SCRIPTS physical address */
outpw(ChipIOBase+CTEST2, regval | 0x08);/* Set bit 3 to get RAM Base in ScratchB*/
Int_Script_Phys_Addr = (ulong) ((ulong) (inpw(ChipBaseIO+
SCRATCHB2) << 16) | inpw(ChipIOBase+SCRATCHB));  /* Read Reg*/
outpw(ChipIOBase+CTEST5, 0x00);/* Clear bit 3 */

Ext_Script_Phys_Addr = (ulong) VirttoPhys(Ext_Script);
/* Patch External Script entries */
Ext_Script[E_Int_dataout_Used[0]] = Int_Script_Phys_Addr +
Ent_Int_dataout;
Ext_Script[E_Int_Start_Used[0]] = Int_Script_Phys_Addr + Ent_Int_Start;
/* Patch Internal SCRIPTS entries */
/* The cmd_buf, data_buf, identify_buf, stat_buf */
/* and msgin_buf should also be done but they will not be */
/* shown in this example as they are not pertinent */
Int_Script[E_Ext_done_Used[0]] = Ext_Script_Phys_Addr +
Ent_Ext_done;
Int_Script[E_Ext_Start_Used[0]] = Ext_Script_Phys_Addr +
Ent_Ext_Start;
}
```

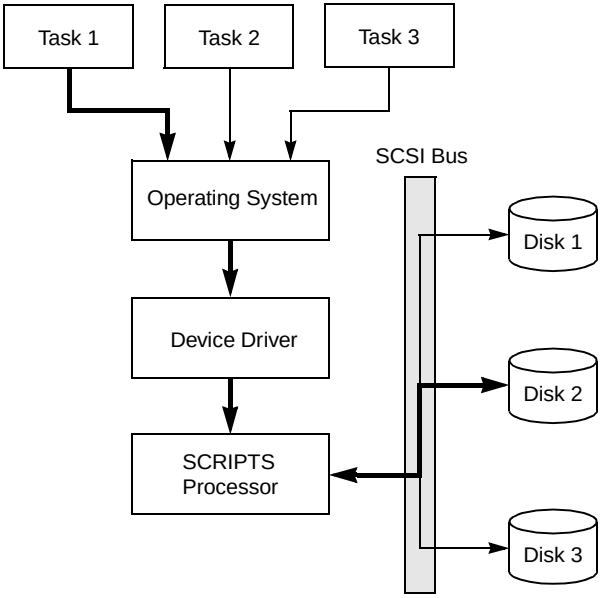
Chapter 10
Multithreaded I/O

This chapter describes multithreaded I/O and contains the following sections:

10.1 Overview

The SCRIPTS processor allows multithreaded I/O operations with minimal external processor intervention in systems that support multitasking. Multithreaded algorithms must be used any time more than one task is active in the system. Figure 10.1 shows a situation where multiple tasks are simultaneously accessing multiple devices. The path between Task 1 and Disk 2 is highlighted to show how information might be transferred. The device driver must schedule and control the I/O requests based on such considerations as what devices are available and the relative priorities of the requests.

Figure 10.1 Multithreaded System Operation

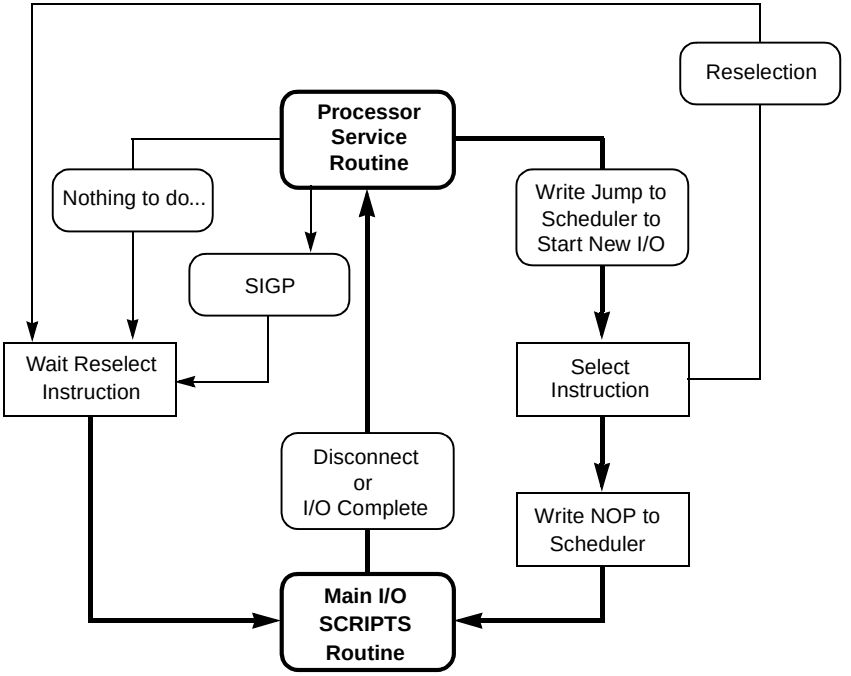


Multithreaded algorithms are similar to single threaded algorithms with disconnects, but a new element called the scheduler is added. The scheduler keeps track of SCSI bus operations when more than one task is active at a time. The SCRIPTS code must be stored in RAM to allow multithreaded operation because SCRIPTS and the CPU dynamically modify SCRIPTS. A multithreaded SCRIPTS algorithm contains three parts: the main SCRIPTS, the scheduler SCRIPTS, and the reselect SCRIPTS. These areas are described in detail after the overview of multithreaded I/O below. This example shows how to implement a scheduler in SCRIPTS. This is only one method of implementing a scheduler. You can choose to schedule I/Os in an upper layer, such as in the “C” driver code.

10.2 Multithreaded Operations Flow

Figure 10.2 shows the flow during multithreaded operation. The heavy lines in the figure represent the initial flow of information for a new operation. The lighter weight lines represent the flow as the chip finishes pending steps of a multithreaded operation.

Figure 10.2 Multithreaded SCRIPTS Operational Flow



To begin a multithreaded operation, your application determines that an I/O is needed and makes an I/O request of the operating system. The operating system then sets up and starts the appropriate device driver. The main driver program modifies the SCSI scheduler routine to call the appropriate I/O SCRIPTS instructions. At this point, normal processing continues as the SCRIPTS processor executes the instructions of the SCRIPTS routine.

When the CPU issues a request for service it writes a JUMP to the scheduler to start the I/O. The SCRIPTS processor selects the SCRIPTS needed to perform the requested action. That instruction writes a NOP to the scheduler to prevent restarting the same I/O. The number of entries (JUMPs) in the scheduler at any one time is the number of I/Os scheduled but not started. The chip then executes the SCRIPTS subroutine and interrupts at completion.

When the SCRIPTS processor has no more instructions to execute, it jumps to the scheduler SCRIPTS area. If no new I/Os are scheduled, the processor jumps to a WAIT RESELECT instruction. If a new I/O is

scheduled, the chip executes the JUMP instruction in the scheduler entry that corresponds to the SCSI ID of the target device to go to the main SCRIPTS area.

If the chip's operation halts until another peripheral device retrieves data, a Wait RESELECT SCRIPT is executed. When the chip is reselected by the target, it resumes execution of the main I/O routine while the chip waits to be reselected by the target device. The CPU may call the chip by setting the SIGP bit. The SCRIPTS processor schedules a new I/O and repeats the cycle described above.

10.3 SCRIPTS Areas

SCRIPTS code is subdivided into three functional areas: main, scheduler and reselect.

The main SCRIPTS area contains the SCRIPTS necessary for the standard operations associated with a SCSI command, such as transferring messages, commands, and data. The scheduler SCRIPTS area contains a three SCRIPTS entry for each job the CPU schedules. The scheduler is modified at run time. When the operating system interface receives an I/O request, it creates an area in host memory for the corresponding scheduler information, and then tracks each request it receives. New requests are classified as outstanding when they are processed and performed. Upon completion of the I/O request, the hardware interface returns a completed status to the operating system interface which updates the status of the request. The reselect SCRIPTS area is the portion of SCRIPTS code that is used after the target disconnects and the SCRIPTS processor is waiting to be reselected.

10.4 Multithreaded SCRIPTS Example

An example operation for the SCRIPTS processor is illustrated below. Steps 1 through 13 and Figures 10.3 through 10.9 make up the example. This example demonstrates multithreaded I/O where only one command is sent to each target at a time. To send more than one command to any target, you must use tagged command queueing. For more complex situations such as this, it may be preferable to use "C" code for scheduling I/Os. The SCRIPTS program must be modified to look at the

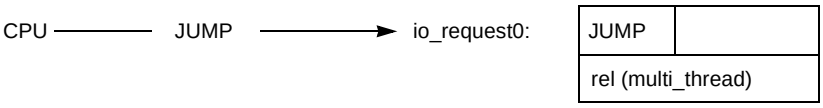
queue tag messages. There must also be a DSA table entry for each possible outstanding tagged command per target ID instead of just one per target ID as in this example. This program appears in Appendix B, “Multithreaded SCRIPTS Example.”

Any item in the code examples that is preceded by “PATCH_” must be patched by the driver. Patching only occurs when the driver is initially loaded. After initialization, all required addresses are in the SCRIPTS array. For more information on instruction patching, refer to Chapter 7, “Integrating SCRIPTS Programs into “C” Language Drivers.”

Note: Both dashed and solid lines are used in some of the program illustrations. The dashed lines indicate pointers and the solid lines indicate data movement in the direction indicated by the arrows.

1. The CPU writes a JUMP into the io_requestX scheduler slot as shown in Figure 10.3.

Figure 10.3 Multithreaded SCRIPTS Example Step 1



2. The CPU may need to set the SIGP bit to indicate that an I/O needs to be processed.

   If this happens the SCRIPTS processor JUMPs to the scheduler. The first instruction in the scheduler sets up the DSA to point to the correct table in the Figure 10.4 example.

Figure 10.4 Multithreaded SCRIPTS Example Step 2

```
;Scheduler SCRIPT code
scheduler:
entry0:
     MOVE MEMORY 4, PATCH_addr_of_table0_ptr, PATCH_chip_physaddr+DSA
```

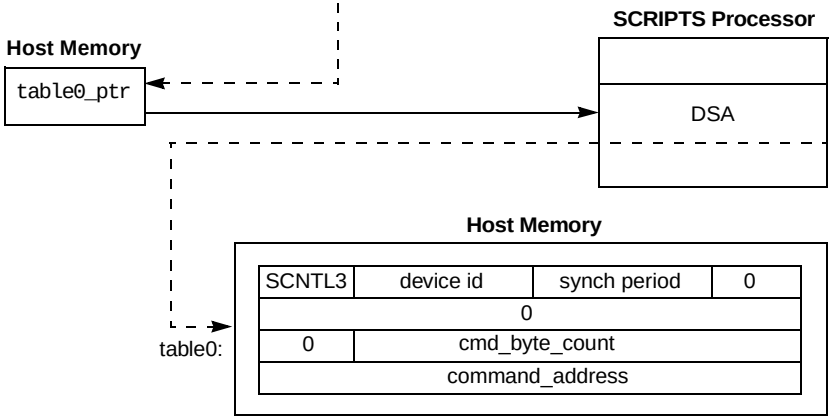
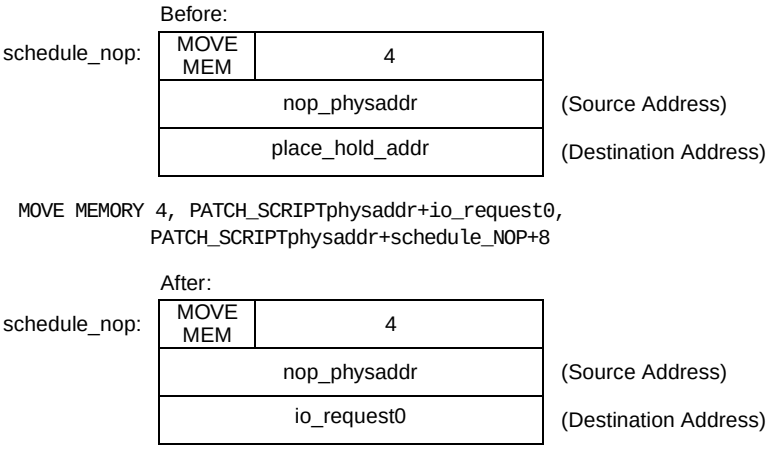


Table0 has the nexus information about any previously negotiated synchronous transfer period and offset. It also contains the SCSI ID of the target device. Clock divider information for the SCNTL3 register is also included in this table. The operating system builds the command and other buffer information into this table prior to starting this I/O.

3. The SCRIPTS instruction moves the address of the IO_requestX into the schedule_nop SCRIPTS destination address field.

   This allows the multithreaded SCRIPT instruction to write a NOP into the io_requestX location in the scheduler to indicate that the I/O has started. See Figure 10.5.

Figure 10.5 Multithreaded SCRIPTS Example Step 3



4. The scheduler jumps to the multithread SCRIPTS subroutine with:

```
io_request0:
JUMP rel (multi_thread)
```

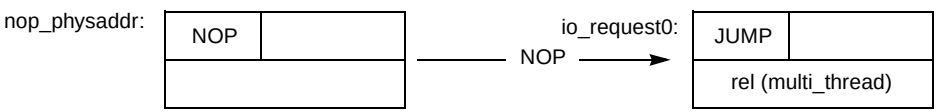
5. The main SCRIPTS routine executes a Select With Attention instruction to connect to the appropriate SCSI device:

```
;Main SCRIPT code
multi_thread:
        SELECT ATN FROM SCSI_id, REL (wait_for_reselect)
```

6. After the two devices are connected, the SCRIPTS instruction writes the NOP into the scheduler routine to avoid trying to start the I/O again.

   This is accomplished by using a Memory-to-Memory Move command. The source address is the address of a NOP SCRIPTS instruction. The destination address is the io_requestX location that was patched into place_hold_addr in the scheduler, as shown in Figure 10.6.

Figure 10.6 Multithreaded SCRIPTS Example Step 6



7. SCRIPTS continues, as in single threaded mode, until a disconnect occurs.

```
        JUMP REL (to_decisions), WHEN NOT MSG_OUT
id_msg_out:
        MOVE FROM identify_msg_buf, WHEN MSG_OUT
        .
        .
        ..
```

8. On disconnection, the initiator jumps to the wait_for_reselect SCRIPT.

   It waits for any device that had previously disconnected to reconnect. If a reselect occurs, the code continues to run. If the device gets selected or the processor issues a SIGP, the SCRIPTS continues at the alternate jump address. Setting the SIGP bit allows the processor to start a new I/O, instead of just waiting for a previous I/O to reconnect.

```
;Reselected SCRIPT code
wait_for_reselect:
        WAIT RESELECT REL (CPU_set_SIGP)
```

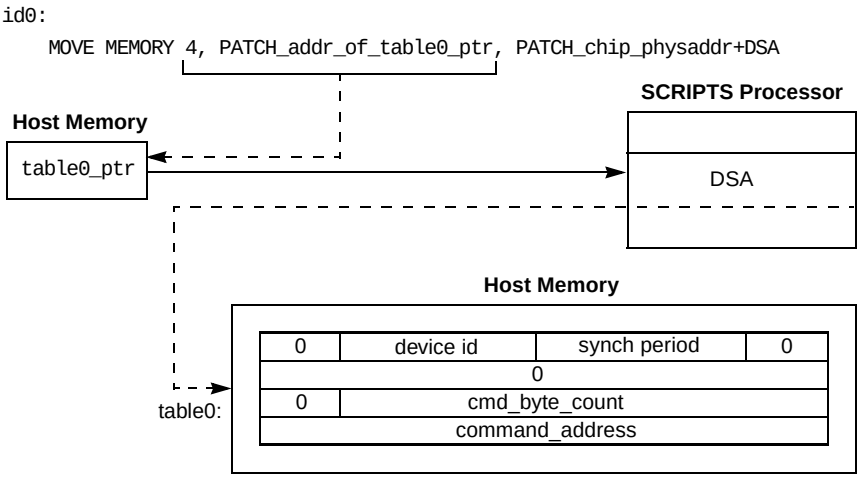
9. The SCRIPTS processor determines the SCSI ID if the reselected device after the initiator is reselected.

   The ID of the device that reselected the chip is in the SSID register.

```
SCSI_id_jump_table:
        MOVE SSID to SFBR
        JUMP REL (id_0), IF 0x00
        JUMP REL (id_1), IF 0x01
        JUMP REL (id_2), IF 0x02
        INT reselect_id_error
```

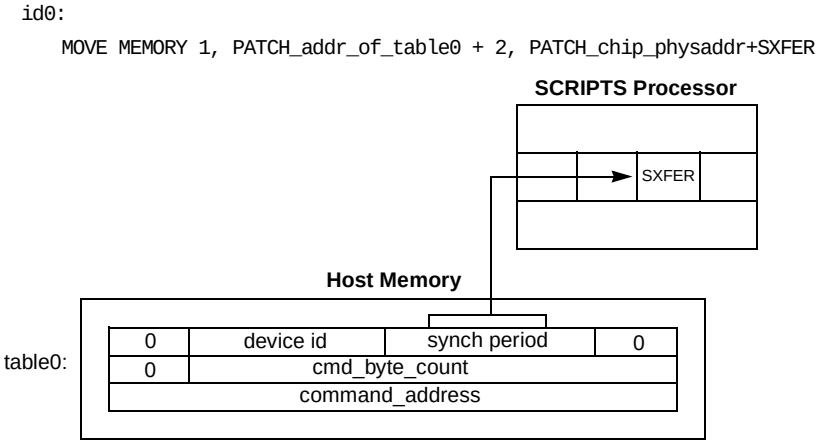
10. The DSA is written to the address of the correct table, depending on the SCSI ID that reselected the initiator. See Figure 10.7.

Figure 10.7 Multithreaded SCRIPTS Example Step 10



11. Synchronous data transfer parameters are restored to the SXFER register and the SCNTL3 register from the information stored in the table. See Figure 10.8.

Figure 10.8 Multithreaded SCRIPTS Example Step 11



12. Table indirect SCRIPTS receive the identify message after the DSA points to the correct table.

```
MOVE FROM identify_msg_buf, WHEN MSG_IN
CLEAR ACK
```

13. SCRIPTS continues with a normal I/O until I/O completion, as shown in Figure 10.9.

Figure 10.9 Multithreaded SCRIPTS Example Step 13

```
JUMP REL (to_decisions)
id_1:
MOVE MEMORY 4, PATCH_addr_of_table1_ptr,
PATCH_chip_physaddr+DSA
MOVE MEMORY 1, PATCH_addr_of_table 1+2,
PATCH_chip_physaddr+SXFER
MOVE FROM identify_msg_buf, WHEN MSG_IN
CLEAR ACK
JUMP REL (to_decisions)
id_2:
MOVE MEMORY 4, PATCH_addr_of_table 2_ptr,
PATCH_chip_physaddr+DSA
MOVE MEMORY 1, PATCH_addr_of_table 2+2,
PATCH_chip_physaddr+SXFER
MOVE FROM identify_msg_buf, WHEN MSG_IN
CLEAR ACK
JUMP REL (to_decisions)
```

10.5 Using the SIGP Bit to Abort an Instruction

The SIGP (Signal Process) bit in the ISTAT register passes a flag to a running SCRIPTS instruction. The SIGP signals that an I/O is ready for execution and has already been scheduled by the host processor. The only SCRIPTS instructions directly affected by this bit are Wait Select and Wait Reselect. Setting the SIGP bit immediately jumps the instruction to the alternate address. For more information on this bit, refer to your chip technical manual. The SCRIPTS code in Figure 10.10 is an example of how to use the SIGP bit when attempting to abort a Wait Reselect or Wait Select instruction, assuming that the device is in the initiator role.

Figure 10.10 Sample SIGP Code

```
;*********************************
reselect_entry:
 WAIT RESELECT alt_sig_p
;  if here, got reselected
handle_resel:
```

```
          *
          *
          *

;***********************************
selected_entry:
 WAIT SELECT alt_sig_p
;  if here, got selected
handle_sel:
          *
          *
          *


;***********************************
alt_sig_p:
;  We assume that the sig_p bit was set,
;  and a reselection needs to be performed.
;  If here because of a selection or
;  reselection or if a selection or
;  reselection occurred during the jump after
;  sig_p bit was set, the alternate address
;  'sel_resel' will be taken.
:  Setup relevant information for this IO.
 RESELECT FROM scsi_id, sel_resel
;  if here, sig_p was set and there was no
;  selection or reselection

 MOVE CTEST2 TO SFBR
;  clear sig_p bit

 MOVE FROM ident_msg, WITH MSG_IN
;  from this point a reselection is performed
;  as normal by moving through the SCSI phases
          *
          *
          *


;***********************************
sel_resel:
;  if here, we have been selected or reselected
;  and sig_p may or may not have been set.
 MOVE SIST0 & 0x20 TO SFBR
;  get selected bit

 JUMP sel, IF 0x20
;  if we got selected
```

```
 MOVE SIST0 & 0x10 TO SFBR
;  get reselected bit

 JUMP resel, IF 0x10
;  if we got reselected

 INT sel_resel_error
;  big error, should have been selected
;  or reselected

;***********************************
sel:
;  if here, selection occurred and sig_p may or
;  may not have been set.  But process selection
;  no matter what.
 JUMP handle_sel

;***********************************
resel:
;  if here, reselection occurred and sig_p may or
;  may not have been set.  But process reselection
;  no matter what.
 JUMP handle_resel:
```

10.6 I/O Completion

On I/O completion, the SCRIPTS processor informs the host system. You can program this operation in one of several ways:

- Write to an address to generate an external interrupt.

  This allows completely interrupt driven software.

  – Write to memory to signal the I/O driver.

  The driver polls the memory location, or, optionally, a general purpose output pin could be used to tell the processor the location contains information. For example, the `status_buf` or `msg_in_buf` would be polled for good status or command complete to signal that an I/O had completed.

```
MOVE 1, status_buf, WHEN STATUS
MOVE 1, msg_in_buf, WHEN MSG_IN
INT error_not_cmd_complete, IF NOT 0
CLEAR ACK
```

```
WAIT DISCONNECT
MOVE MEMORY 1, IO_DONE_BUF, DONE_YET_BUF
JUMP scheduler
```

– Execute a SCRIPTS INT instruction.

This is the simplest method. It causes the SCSI SCRIPTS to stop processing.

```
INT io_complete
```

– Execute a Memory-to-Memory Move to a predetermined location, then execute an INTFLY instruction to indicate to the processor to look at the predetermined location to verify which I/O has completed.

Chapter 11
Using the SCRIPTS Processor in Target Applications

This chapter describes using chips with the SCRIPTS processor in target applications and includes these topics:

11.1 SCSI and Target SCRIPTS Protocol

The LSI53C7XX/8XX/10XX family of chips run on target as well as host devices. Target operation is very similar to host operation, except that the SCRIPTS processor responds to SCSI commands from the host rather than initiating the commands. The basic structure of all target operations is:

- The SCRIPTS processor issues a Wait Select instruction.
- The SCSI bus goes into Message Out phase.
- The SCRIPTS processor performs a series of Block Moves corresponding to the next four SCSI bus phases. See Table 11.1.
- The SCRIPTS processor issues a Disconnect instruction to disconnect the target device from the bus.

Table 11.1 SCSI Protocol and Target SCRIPTS Instructions

Bus Phase	Definition	SCRIPTS Instruction
Bus Free	Indicates that the SCSI bus is available.	N/A
Arbitration	Allows the initiator to gain control of the SCSI bus.	N/A
Selection	During this phase, the target responds to the initiator's selection.	WAIT SELECT
Message Out	Target can receive messages from the initiator, such as queuing and error recovery information.	MOVE WITH MESSAGE OUT
Command	Target can receive commands in the form of a CDB to the target buffer.	MOVE
Data In/Out	Data In and Data Out phases send data to the initiator or to the target and are used dependent on the information transferred during the Command phase. This phase is optional. For example, a Test Unit Ready command does not require a data transfer.	MOVE
Status	Target sends status information to the initiator about the previously executed CDB.	MOVE
Message In	Target sends messages to the initiator. These messages can acknowledge or reject previously sent initiator messages. They also can provide other information like queuing, disconnect, or parity errors.	MOVE
Disconnect	Ends the target device's connection with the bus.	DISCONNECT
Bus Free	After successful completion of an I/O operation and a request for disconnect, the bus returns to the Bus Free state, indicating that it is now available.	DISCONNECT

11.2 Registers Used for Target Operation

Only a few of the operating register values are different for target operation when compared to initiator operation. Table 11.2 summarizes the register bit operations specific to target operation.

Table 11.2 Register Bits Used for Target Operation

Register Name	Bits	Description
RESPID1, RESPID0	all	Setting multiple bits in these registers allows the processor to respond to multiple SCSI IDs.
SCNTL0	0	Set this bit to make the chip a target device by default.
SCID	5	Set this bit to allow the processor to respond to bus initiated selection at the chip ID in the RESPID1–0 registers.
SCNTL1	5	When this bit is cleared, the processor halts the data transfer when a parity error is detected or when the SATN/ signal is asserted.

11.3 Using SCRIPTS for Target Operation

SCRIPTS instructions operate identically in target or initiator mode, except for certain forms that are valid in only one mode. These exceptions are all noted in the individual instruction descriptions in Chapter 3, "The SCSI SCRIPTS Processor Instruction Set." When the target device is moving data to the SCSI bus and is halted for any reason, the residual data in the FIFO must be cleared before resuming the transfer. It is most common to empty the FIFOs, send a Restore Pointers message and start the transfer again.

Most interrupts to target operation are expected. The floppy disk provided with this programming guide contains a sample interrupt service routine for a target device.

11.3.1 Sample Target Operation SCRIPTS Program

This section illustrates programming for target operation with a sample SCRIPTS program. This program is used for testing and development of LSI Logic SCSI products. The full text of the SCRIPTS source file and accompanying code for target operation can be downloaded from the LSI Logic website under OEM Development at this link:
`http://www.lsilogic.com/products/techsupp/index.html`
Figures 11.1 through 11.26 are sections of the code. Each figure has supporting text.

Figure 11.1 SCRIPTS Source Code–Comments

```
8xxtarg.ss   Revision 2.2 2/12/96
;
; This software was written by LSI Logic Inc. to
; develop and test new products. LSI Logic assumes
; no liability for its use. This software is released
; to the public domain to illustrate certain
; programming techniques for the LSI53C8xx chips in
; target mode.
;
```

ABSOLUTE declarations, as shown in Figure 11.2, are interrupts generated by the target. The SCRIPTS processor issues interrupts to notify the host of completed actions or to find out what action to take next.

Figure 11.2 SCRIPTS Source Code–ABSOLUTE Declarations

```
;ABSOLUTE DECLARATIONS
ABSOLUTE read_access_medium = 0x00
ABSOLUTE write_access_medium = 0x01
ABSOLUTE last_write_disconnect = 0x02
ABSOLUTE seek_command   = 0x03
ABSOLUTE set_up_synch_neg = 0x04
ABSOLUTE set_up_wide_neg = 0x05
ABSOLUTE non_handled_msg = 0x06
ABSOLUTE bad_extended_msg = 0x07
ABSOLUTE message_sent = 0x08
ABSOLUTE request_sense_command = 0x09
ABSOLUTE inquiry_command = 0x0a
ABSOLUTE read_capacity_command = 0x0b
ABSOLUTE start_stop_command = 0x0c
ABSOLUTE format_unit = 0x0d
ABSOLUTE send_diagnostic = 0x0e
ABSOLUTE command_aborted = 0x0f
ABSOLUTE illegal_cmd    = 0x10
ABSOLUTE got_SIGP = 0x11
ABSOLUTE done_with_copy = 0x12
ABSOLUTE got_selected   = 0x13
ABSOLUTE done_with_busy_command = 0x14
```

EXTERNs are variables used for Memory-to-Memory Move operations. For example, Figure 11.3 demonstrates moving SCRIPTS from program memory into RAM or moving data from one memory location to another.

Figure 11.3 SCRIPTS Source Code–EXTERN Variables

```
EXTERN count
EXTERN source_address
EXTERN destination_address
```

TABLE, shown in Figure 11.4, defines the table format and layout. Each entry in the table represents a two Dword entry in a data structure. Each entry contains a byte count and an address that points to a buffer for Block Move instructions. The buffer must be declared in the driver code.

Note: The declared values and sizes are only for the SCRIPTS debugger, NVPCI. The assembler does not use these and the information is not included in the “C” code. The buffers must be set up in the driver program.

Figure 11.4 SCRIPTS Source Code–TABLE

```
TABLE table_indirect \
   msg_out_buf = 1{??}, \
   cmd_buf = 12{??}, \
   synch_neg_msg_out = 2{??}, \
   wide_neg_msg_out = 1{??}, \
   neg_msg_in = {0x01, 0x03, 0x01, 0x19, 0x08}, \
   stat_buf = {0x02}, \
   identify_msg_in_buf = {0x80},\
   msg_in_buf = 1{??}, \
   data_buf = 512{??}, \
   save_pointers = {0x02}, \
   disconnect_msg = {0x02, 0x04}, \
   selector_id = ID{0x33, 0x07, 0x00, 0x00}, \
   sense_data_buf = {0x00,0x00,0x06,0x00, 0x00, 0x00, \
                  0x00, 0x0a, 0x00, 0x00, 0x00, 0x00, \
                  0x29, 0x00, 0x00, 0x00, 0x00, 0x00}, \
  inquiry_data_buf = {0x00,0x00,0x02,0x00, 0x1f,0x00, \
                  0x00, 0x10, 0x20, 0x20, 0x20, 0x20, \
                  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, \
                  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, \
                  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, \
                  0x20, 0x20, 0x20, 0x20, 0x20, 0x20}, \
   capacity_data_buf = {0x00, 0x80, 0x02, 0x00}

;*****************************************************
```

ENTRY declarations are starting points in the SCRIPTS program and are referred to in “C” code. See Chapter 5, “The NASM Output File,” for more information on output files and how they are assembled and used in the driver code and Figure 11.5 for example code.

Figure 11.5 SCRIPTS Source Code–ENTRY Declarations

```
; ENTRY declarations
ENTRY wait_select
ENTRY msg_out_phase
ENTRY tur
ENTRY stopped_busy_tur
ENTRY request_sense
ENTRY read_return
ENTRY read_reconnect
ENTRY write_return
ENTRY write_reconnect
ENTRY synch_wide_neg_return
ENTRY msg_in_phase
ENTRY inquiry
ENTRY read_capacity
ENTRY stopped_busy_wait_select
ENTRY copy_data
```

The `wait_select` label, Figure 11.6, is the generic starting point for target operations. The SCRIPTS processor waits at this point until selection. It jumps to Command phase if ATN is not set or performs one of the other commands described in the comments below. If the SIGP bit is set, it jumps to an alternate label.

Figure 11.6 SCRIPTS Source Code–wait_select Label

```
wait_select:
    wait select rel(SIGP_set)                  ;wait to be selected
    jump rel(command_phase), if not atn        ;SCSI-1 initiator support
    move from msg_out_buf, with msg_out        ;get message byte
    move sfbr to scratchb0                     ;save the identify message
    call rel(msg_out_phase), if atn            ;stay in message if atn still
                                               ;active
```

If the SCRIPTS processor is selected without ATN, it goes directly to the Command phase to support SCSI-1 initiators. The chip receives the CDB and performs various functions, described in the program comments of Figure 11.7, depending on the contents of the command.

Figure 11.7 SCRIPTS Source Code–CDB Functions

```
command_phase:
    move from cmd_buf, with cmd          ;get SCSI command
    move scntl1 & 0xdf to scntl1         ;turns on the halt on parity error or atn
    jump rel(read), if 0x08              ;jump to set up read (6-byte read)
    int write_access_medium, if 0x0a     ;interrupt to set up write (6-byte write)
    int seek_command, if 0x0b            ;interrupt to perform seek
    int seek_command, if 0x2b            ;interrupt to perform seek
    jump rel(read), if 0x28              ;jump to set up read (10-byte read)
    int write_access_medium, if 0x2a     ;interrupt to set up write (10-byte write)
    jump rel(tur), if 0x00               ;jump to test unit ready
    int request_sense_command,if 0x03    ;interrupt to set up request sense command
    int inquiry_command, if 0x12         ;interrupt to set up inquiry command
    int read_capacity_command,if 0x25    ;interrupt to set up read capacity command
    int start_stop_command, if 0x1b      ;interrupt to set up start/stop unit command

    jump rel(tur), if 0x2f               ;verify command, go to tur
    jump rel(reserve_unit), if 0x16      ;jump to reserve unit
    jump rel(release_unit), if 0x17      ;jump to release unit
    int send_diagnostic, if 0x1d         ;interrupt to set up send diagnostic
    int format_unit, if 0x04             ;interrupt to set up format unit
    int illegal_cmd                      ;interrupt on any other command
```

In Message Out phase, the initiator moves other types of messages, such as wide or synchronous negotiation, or NOPs. See Figure 11.8 for an example.

Figure 11.8 SCRIPTS Source Code–Message Out Phase

```
msg_out_phase:
    return, if not atn                   ;return if atn gone
    move from msg_out_buf, with msg_out  ;get message byte
    jump rel(extended_msg), if 0x01      ;jump if extended message
    jump rel(abort), if 0x06             ;jump if abort message
    jump rel(msg_out_phase), if 0x08     ;jump back if nop message
    int non_handled_msg                  ;interrupt if can't handle message
```

If the chip receives a byte in the message phase indicating an extended message, then it jumps to these commands as shown in Figure 11.9.

Figure 11.9 SCRIPTS Source Code–Extended Message

```
extended_msg:
    int bad_extended_msg, if not atn      ;if atn gone, extended message was bad
    move from msg_out_buf, with msg_out   ;get next message byte
    int bad_extended_msg, if not atn      ;if atn gone, extended message was bad
    move from msg_out_buf, with msg_out   ;this byte shows type of message
    jump rel(synch_neg), if 0x01          ;0x01 is a synchronous negotiation message
    jump rel(wide_neg), if 0x03           ;0x03 is a wide negotiation message
    int bad_extended_msg                  ;interrupt on any other type
```

In synchronous negotiation, Figure 11.10, the SCRIPTS processor moves synchronous period and offset data from the synchronous negotiation message and interrupts to set up the synchronous operation and return message.

Figure 11.10 SCRIPTS Source Code–Synchronous Negotiation

```
synch_neg:
    move from synch_neg_msg_out, with msg_out    ;move in the period and offset
    int set_up_synch_neg                         ;interrupt to set up synchronous and
                                                 ;message
```

If the extended message indicates wide negotiation, the SCRIPTS processor expects one more byte with SCSI bus width information, as shown in Figure 11.11. After receiving the byte, it interrupts to set up the answer.

Figure 11.11 SCRIPTS Source Code–Wide Negotiation

```
wide_neg:
    move from wide_neg_msg_out, with msg_out     ;move in the width
    int set_up_wide_neg                          ;interrupt to set up answer
```

After the interrupt service routine executes, the SCRIPTS processor sends its return negotiation message. See Figure 11.12 for an example.

Figure 11.12 SCRIPTS Source Code–Return Negotiation

```
synch_wide_neg_return:
    move from neg_msg_in, with msg_in     ;move out our answer to the negotiation
    jump rel(command_phase), if not atn   ;jump to command_phase if atn gone
    jump rel(msg_out_phase)               ;jump to msg_out_phse if atn still active
```

If the target device goes into the Message In phase, a resulting exception condition requires the target device to send some type of recovery message to the initiator, as shown in Figure 11.13. These commands indicate what the initiator must do next. The messages may include, but are not limited to: Message Reject or Restore Data Pointers.

Figure 11.13 SCRIPTS Source Code–Recovery Message

```
msg_in_phase:
    move from msg_in_buf, with msg_in     ;move a message to the initiator
    int message_sent                      ;interrupt to determine what to do next
```

Test Unit Ready, Figure 11.14, is the final sequence of commands for any I/O. The SCRIPTS processor sends a status message, disconnects from the SCSI bus, executes an interrupt on the fly, and goes back to the `wait_select` label to get ready for next command.

Figure 11.14 SCRIPTS Source Code–Test Unit Ready

```
tur: ; (Test Unit Ready)
    move from stat_buf, with status       ;send out status byte
    move from msg_in_buf, with msg_in     ;send out message byte
    move 0x20 to scntl1                   ;turns off the halt on parity error or atn
    disconnect                            ;disconnect from the SCSI bus
    intfly                                ;interrupt to signal end of process
    jump rel(wait_select)
```

The stopped_busy_tur, Figure 11.15, is the same as the Test Unit Ready label. However, the SCRIPTS processor has been selected while processing another command or has been issued a Stop command. If the command is one the target device does not have to accept when busy or stopped the device stops and sends back a busy status.

Figure 11.15 SCRIPTS Source Code–stopped_busy_tur Command

```
stopped_busy_tur:
    move from stat_buf, with status       ;send out status byte
    move from msg_in_buf, with msg_in     ;send out message byte
    move 0x20 to scntl1                   ;turns off the halt on parity error or atn
    disconnect                            ;disconnect from the SCSI bus
    move scratcha1 to sfbr                ;get the busy flag
    int done_with_busy_command, if 0x01   ;if busy, interrupt to continue
    intfly                                ;interrupt to signal end of process
    jump rel(stopped_busy_wait_select)
```

Request Sense, Figure 11.16, sends the sense, inquiry, or capacity data requested by the initiator. The SCRIPTS processor moves the data and checks to see which Test Unit Ready command to use next.

Figure 11.16 SCRIPTS Source Code–Request Sense

```
request_sense:
    move from sense_data_buf, with data_in      ;move the sense data from the buffer
    move scratcha2 to sfbr                      ;get the stopped/busy flag
    jump rel(tur) if 0x00                       ;go to the appropriate status and
                                                ;message phases
    jump rel(stopped_busy_tur)
inquiry:
    move from inquiry_data_buf, with data_in    ;move out inquiry data
    move scratcha2 to sfbr                      ;get the stopped/busy flag
    jump rel(tur) if 0x00                       ;go to the appropriate status and
                                                ;message phases
    jump rel(stopped_busy_tur)
read_capacity:
    move from capacity_data_buf, with data_in   ;move out read capacity data
    move scratcha2 to sfbr                      ;get the stopped/busy flag
    jump rel(tur) if 0x00                       ;go to the appropriate status and
                                                ;message phases
    jump rel(stopped_busy_tur)
```

The read label, Figure 11.17, is the starting point for all read commands. If disconnects are allowed, the chip jumps to the read_disconnect label. Read return is used after read information is set up in the data buffer. A series of commands determine if the transfer is finished. If finished, the SCRIPTS processor goes to Test Unit Ready or tries to disconnect again.

Figure 11.17 SCRIPTS Source Code–Read Label

```
read:
    move scratchb0 to sfbr                          ;get identify message
    jump rel(read_disconnect) if 0x40 and mask 0xbf;jump disconnect if
                                                    ;disconnects are allowed
    int read_access_medium                          ;interrupt to read data from
                                                    ; medium

read_return:
    move from data_buf, with data_in                ;move the data out from the buffer
    move scratchb1 to sfbr                          ;get the 'finished' flag
    jump rel(tur) if 0x00                           ;jump to status and message
                                                    ;if transfer done
    move scratchb0 to sfbr                          ;get identify message
    jump rel(read_disconnect) if 0x40 and mask 0xbf
                                                    ;jump to disconnect if disconnects
                                                    ;are allowed
    move from save_pointers, with msg_in            ;move out the save pointers
                                                    ;message
    call rel(msg_out_phase) if atn                  ;jump to message out if atn active
    int read_access_medium                          ;interrupt to access medium
```

The read_disconnect label, Figure 11.18 disconnects the device from the bus and sets the Semaphore bit, which tells the interrupt service routine it is disconnected.

Figure 11.18 SCRIPTS Source Code–read_disconnect Label

```
read_disconnect:
    move from disconnect_msg, with msg_in       ;move out the disconnect message
    call rel(msg_out_phase) if atn              ;jump to message out if atn active
    move 0x20 to scntl1                         ;turns off the halt on parity
                                                ;error or atn
    disconnect                                  ;disconnect from the bus
    move 0x10 to istat                          ;set the semaphore bit to say we
                                                ;are disconnected
    int read_access_medium                      ;interrupt to read data from medium
```

The read_reconnect label, Figure 11.19, performs reselection, moves the identify message from the message in buffer, and jumps to send the data.

Figure 11.19 SCRIPTS Source Code–read_reconnect Label

```
read_reconnect:
    reselect from selector_id, rel(alt_got_selected ;reselect the initiator
    move scntl1 & 0xdf to scntl1                    ;turns on the halt on parity
                                                    ;error or atn
    move from identify_msg_in_buf, with msg_in      ;move in identify message
    jump rel(read_return)                           ;jump to send data
```

On a write, the SCRIPTS processor interrupts immediately to set up counts for moving data, as shown in Figure 11.20. It takes data from the initiator, then begins the write. When the writing is complete, control jumps to the Test Unit Ready label.

Figure 11.20 SCRIPTS Source Code–Write

```
write_return:
    move from data_buf, with data_out     ;move the data into the buffer
    move scratchb0 to sfbr                ;get identify message
    jump rel(write_disconnect) if 0x40 and mask 0xbf
                                          ;jump to disconnect if disconnects allowed
    move scratchb1 to sfbr                ;get the 'finished' flag
    jump rel(tur) if 0x10                 ;jump to status and message if transfer done
    move from save_pointers, with msg_in  ;move out the save pointers message
    call rel(msg_out_phase) if atn        ;jump to message out if atn active
    int write_access_medium               ;interrupt to read data from medium
```

The write_disconnect label, Figure 11.21, does all the same things as the read_ disconnect. It sets the semaphore bit and issues one of two interrupts, depending on whether or not this is the last write of the transfer.

Figure 11.21 SCRIPTS Source Code–write_disconnect Label

```
write_disconnect:
    move from disconnect_msg, with msg_in ;move out the disconnect message
    call rel(msg_out_phase) if atn        ;jump to message out if atn active
    move 0x20 to scntl1                   ;turns off the halt on parity error or atn
    disconnect                            ;disconnect from the bus
    move 0x10 to istat                    ;set the semaphore bit to say we are
                                          ;disconnected
    move scratchb1 to sfbr ;get the 'finished' flag
    int last_write_disconnect if 0x10     ;special interrupt after last data phase
    int write_access_medium               ;interrupt to read data from medium
```

The write_reconnect label, Figure 11.22, operates the same as read_reconnect.

Figure 11.22 SCRIPTS Source Code–write_reconnect Label

```
write_reconnect:
    reselect from selector_id, rel(alt_got_selected)  ;reselect the initiator
    move scntl1 & 0xdf to scntl1                      ;turns on the halt on parity
                                                      ;error or atn
    move from identify_msg_in_buf, with msg_in        ;move in identify message
    move scratchb1 to sfbr                            ;get the 'finished' flag
    jump rel(tur) if 0x00                             ;jump to status and message
                                                      ;if transfer complete
    jump rel(write_return)                            ;jump to get data
```

The reserve_unit label, Figure 11.23, sets a reservation flag, gets the ID of the initiator that sent the command, jumps to Test Unit Ready, and completes the command.

Figure 11.23 SCRIPTS Source Code–reserve_unit Label

```
reserve_unit:
    move 0x01 to scratchb2              ;set 'reserved' in reservation flag
    move ssid & 0x7f to sfbr            ;get the ID of who reserved us
    move sfbr to scratchb3              ;move ID into storage buffer
    jump rel(tur)                       ;go to status and message
```

The release_unit command, Figure 11.24, clears the reserved flag and goes to Test Unit Ready.

Figure 11.24 SCRIPTS Source Code–release_unit Command

```
release_unit:
    move 0x00 to scratchb2        ;set 'not reserved' in reservation flag
    jump rel(tur)                 ;go to status and message
```

The abort label turns off the halt on parity or ATN bit, and disconnects from the bus, as shown in Figure 11.25. The chip executes this command when it receives an Abort message for the command in process. The interrupt service routine then cleans up the job.

Figure 11.25 SCRIPTS Source Code–abort Label

```
abort:
    move 0x20 to scntl1          ;turns off the halt on parity error or atn
    disconnect                   ;go to bus free
    int command_aborted          ;int to notify driver that command was aborted
```

The SCRIPTS processor only performs the stopped_busy_wait_select, Figure 11.26, if it is selected while stopped or busy working on another command. The Request Sense, Test Unit Ready, Inquiry, and Read Capacity commands are valid while the target device is busy or stopped and the chip must respond to them. If the chip is stopped, it only responds to one of these commands or to a Start command.

Figure 11.26 SCRIPTS Source Code–stopped_busy_wait_select Command

```
stopped_busy_wait_select:
    wait select rel(SIGP_set)            ;wait to be selected
    move from msg_out_buf, with msg_out  ;get message byte
    call rel(msg_out_phase), if atn      ;stay in message if atn still active
    move from cmd_buf, with cmd          ;get SCSI command
    move scntl1 & 0xdf to scntl1         ;turns on the halt on parity error or atn

    jump rel(stopped_busy_tur), if 0x00  ;jump to test unit ready
    int request_sense_command, if 0x03   ;interrupt to set up request sense command
    int inquiry_command, if 0x12         ;interrupt to set up inquiry command
    int read_capacity_command, if 0x25   ;interrupt to set up read capacity command

    move sfbr to scratcha3               ;save the first byte of the command
    move scratcha1 to sfbr               ;get the busy flag
    jump rel(stopped_busy_tur), if 0x01  ;if busy, go right to status and message

    move scratcha3 to sfbr               ;restore the first byte of the command
    int start_stop_command, if 0x1b      ;interrupt to set up start/stop unit command

    jump rel(stopped_busy_tur)           ;go to status and message for any other
                                         ;command

alt_got_selected:
    int got_selected                     ;interrupt because got selected during
                                         ;reselect attempt

SIGP_set:
    int got_SIGP                         ;taking interrupt because got SIGP

copy_data:
    move memory count, source_address, destination_address
                                         ;memory move to write SCRIPTS RAM and to
                                         ;transfer data to and from upper memory
    int done_with_copy                   ;signal completion of memory move
```

11.4 Synchronous Negotiation by a Target Device

For target operation, negotiating occurs when a synchronous negotiation message is received from the initiator. After receiving this message, a SCRIPTS Interrupt instruction is executed to determine the necessary response. After establishing the synchronous parameters for a particular initiator, they should be saved in a table for later reconnects to the same device. If reselecting an initiator, the RESELECT FROM command can be used to indicate table indirect addressing. Subsequently, the SXFER, SCNTL3, and SDID register values are loaded from the table entry. When selected by an initiator that has previously negotiated for synchronous transfers, these registers are reloaded from memory before the target goes to the data transfer phase.

Chapter 12
Debugging the SCRIPTS Processor

This chapter describes debugging the SCRIPTS processor and includes these topics:

12.1 Chip Debugging Guidelines

The list below has common problems and solutions you can use as part of a debugging routine.

- Check the register initialization routine.

  Several registers should be checked in this step. The most important registers to verify are listed in Chapter 6, “Using the Registers to Control Chip Operations.”

  Save and print out the data values in all SCRIPTS processor registers at the time the problem occurs.

  Record the value of the ISTAT register first, since further register accesses may trigger interrupts that were not caused by the initial problem. If there is not an interrupt, abort the SCRIPTS operation by writing to the ABRT bit in the ISTAT register. This will cause a DMA abort interrupt. Reset this bit before reading the DSTAT register to prevent further interrupts from being generated. Clear the interrupt(s) following the method suggested in Chapter 6, “Using the Registers to Control Chip Operations.”

  Check the registers listed in Table 12.1 after clearing the interrupts.

  If there is no indication of what is causing the problem, it might be helpful to examine the remaining registers.

- Use the DSP, DSPS, DCMD, and DBC registers to determine where SCRIPTS execution was stopped.

  The `.LIS` file generated by NASM using the `-l` option can be very helpful in this step. Compare the listings to the debugging register values to determine what might be causing the problem.

- Examine the logic analyzer traces of both the host bus and the SCSI bus to verify that SCRIPTS fetches are occurring correctly.

  This may also be helpful when comparing data transferred between the two interfaces.

- Perform timing verification using a logic analyzer.

  Signal quality issues and clock problems may require the use of an oscilloscope.

- The CPU is accessing registers other than ISTAT while SCRIPTS are running.

  ISTAT is the only register that can be accessed during SCRIPTS operation.

- The RESPID register(s) are not initialized.

  This would keep the chip from responding to any selection/reselection. Make sure these registers are initialized correctly.

- Verify signal connectivity. (Make sure that the chip pins are all connected to board traces.)

- Verify power and ground connection to the chip.

- Verify that decoupling capacitors are connected as recommended in the chip technical manual to avoid noise problems.

- Make sure that the Enable Response to Selection/Reselection bits are set correctly.

If you still have problems, take the information collected, along with your code, and contact your LSI Logic field engineer.

12.2 Register Used for Debugging

The SCRIPTS registers and the SCSI registers contain information that may be helpful in debugging the chip. Table 12.1 shows the information contained in the registers.

Table 12.1 Registers Useful for Debugging SCRIPTS Processor

Information	Register	Remarks
Information regarding the most recent interrupt	ISTAT	Check this register first, since its contents may be affected by reading or writing other registers.
Current SCRIPTS instruction	DCMD and DBC (first 32-bits); DNAD or DSPS (second 32-bits)	The DCMD and DBC always contain the opcode of the most recently executed SCRIPTS instruction. Use the cross reference file created from the SCRIPTS source by NASM to interpret the contents. The DSPS or DNAD contains the second 32-bit field of the SCRIPTS instruction fetched.
Next SCRIPTS Instruction address	DSP	Contains the address of the next instruction to be fetched. This is analogous to the program counter of a microprocessor. Instruction addresses are on 8-byte boundaries (except Memory Move, which is on a 12-byte boundary) and so the value in the DSP should be eight past the address of the current instruction.
SCSI Bus Control Lines	SBCL	Contains the current state of the SCSI control lines.
SCSI Bus Data Lines	SBDL	Contains the current status of the SCSI data lines.
Last SCSI Phase serviced	SOCL	Contains the phase to match (initiator) or the phase driven (target) from the last SCRIPTS instruction executed.
Last SCSI data byte sent	SODL	Contains the last byte transferred to the SCSI bus.
Last SCSI data byte received	SIDL	Contains the last byte transferred in from the SCSI bus.
First byte received from Block Move instruction executed	SFBR	Contains the first byte of a block move transferred in from the SCSI bus. It also contains SCSI identities after a reselection, if using the LSI53C700 compatibility mode and if the IDs are in the 0–7 range.
SCSI ID	SCID	Contains the chip's SCSI ID.

Table 12.1 Registers Useful for Debugging SCRIPTS Processor (Cont.)

Information	Register	Remarks
Destination SCSI ID	SDID	Contains the identity of the target for the last select or reselect instruction executed.
Response ID	RESPID0, RESPID1 (wide SCSI devices only)	Contains the IDs that the chip responds to on the SCSI bus. The chip can respond to multiple IDs, so more than one bit can be set in these registers.

Chapter 13
New SCRIPTS Processor Features

This chapter describes features found in the LSI Logic 64-bit chips with the SCRIPTS processor. Features described include:

Note: The chips covered in this section are the LSI53C895A, LSI53C896, and the LSI53C10XX. Refer to Table 1.1 for an overview of their specifications.

13.1 Improved FIFO Flushing

During data in phase mismatches, the SCRIPTS processor flushes at the programmed burst size until the available burst size is less than the programmed burst size. When the flush completes, an interrupt is generated. If the available burst size is less than the programmed value, it flushes as it normally would, one Dword per PCI cycle. Enhanced flushing is enabled and disabled in parallel with phase mismatch handling but it can also be disabled independently.

13.2 Larger FIFO

FIFO size varies with the specific chip. Refer to Table 1.2 for specifics. Max burst size is now 64 levels in the LSI53C896/10XX due to the increased FIFO width.

13.3 New ISTAT Registers

ISTAT for these chips is now 32-bits wide. ISTAT1 is in byte lane 1. For more details on ISTAT registers, refer to the appropriate chip technical manual. Table 13.1 provides an overview of the registers in ISTAT1.

Table 13.1 ISTAT1 Register

Bits [7:3]	Reserved
Bit 2	Flushing in progress
Bit 1	SCRIPTS running
Bit 0	Synchronous IRQ disable

Byte lane 2 and 3 are general purpose mailbox registers (MBOX1 and MBOX2) that communicate with the SCRIPTS engine. All 8 bits of a mailbox should be either writes or reads, not a combination.

13.4 New Scratch Registers

Eight new 32-bit Scratch registers have been added to the chips, for a total of 18. The new registers are SCRATCHK through SCRATCHR.

13.5 New Load/Store Feature

The SCRIPTS processor no longer uses the PCI bus for Load/Store instructions when moving data between the chip registers and the SCRIPTS RAM. This feature can be disabled by setting bit 1 of CCNTL0 (Offset 0x56).

13.6 Phase Mismatch Handling

Phase Mismatch Handling eliminates the Phase Mismatch Interrupt. The default setting for this feature is OFF. Bit 7 of CCNTL0 (offset 0x56) enables the feature. Phase Mismatch Handling has the following features.

- Performs all necessary byte count/pointer calculations then jumps to a SCRIPTS phase mismatch handler.
- Supports two jump vectors with programmable jump control.
- Supports jump enable/disable during nondata phases.
- Supports Loadable Cumulative SCSI Byte Count to maintain total bytes transferred for a given I/O.

  Note: Overhead to jump is approximately 16 PCI clocks, not including time to flush.

13.6.1 Control Bits

This section describes the control bits used for phase mismatch handling. All bits are located in CCNTL0 (0x56).

- Bit 7: ENPMJ, Enable Phase Mismatch Jump (default = 0)
- Bit 6: PMJCTL, Phase Mismatch Jump Control (default = 0)

  This bit controls which decision mechanism is used when jumping on phase mismatches. When PMJCTL is clear, PMJAD1 is used when WSR is clear, PMJAD2 when WSR is set. When PMJCTL is set, PMJAD1 is used during data out phases, PMJAD2 used during data in phases.
- Bit 5: ENNDJ, Enable Nondata Jump (default = 0)

  When this bit is clear a Phase Mismatch interrupt is generated on nondata phase mismatches, such as Status, Msg In/Out, and Command. When set, jumps are taken during nondata phases.
- Bit 4: DISFC, Disable Auto FIFO Clear (default = 0)

  This bit disables automatic FIFO clearing on data out phase mismatches and disables enhanced flushing.

13.6.2 Registers

This is a list of the registers that are involved with Phase Mismatch Handling.

- Phase Mismatch Jump Address one, PMJAD1 (0xC0–0xC3) R/W

  This register contains the address the SCRIPTS engine jumps to on phase mismatch if WSR is clear or during a data out phase.

- Phase Mismatch Jump Address two, PMJAD2 (0xC4–0xC7) R/W

  This register contains the address to which the SCRIPTS engine jumps on phase mismatch if WSR is set or during a data in phase.

- Remaining Byte Count, RBC (0xC8–0xCB) R/W

  This register contains the remaining byte count for the block move that was executing when the phase mismatch occurred. The upper byte also contains an opcode for a direct or indirect block move or the upper byte of the table entry for table indirect block moves.

- Updated Address, UA (0xCC–0xCF) R/W

  This register contains the updated source/destination data address for the block move that was executing when the phase mismatch occurred. If there is a byte in SWIDE, then this register points to the address where the byte should be stored. The address must be incremented manually.

- Entry Storage Address, ESA (0xD0–0xD3) R/W

  For direct/indirect block moves, this register contains the address of the block move instruction that was executing when the phase mismatch occurred. For table indirect block moves this register contains the address of the table entry being used when the phase mismatch occurred.

- Instruction Address, IA (0xD4–0xD7) R/W

  This register always contains the address of the block move that was executing when the phase mismatch occurred.

- SCSI Byte Count/SBC (0xD8–0xDA) Read only

  This register counts bytes transferred to/from the SCSI bus during any given block move. Resets to zero at the start of each block move. Will be off by one in the case of an odd byte count wide transfer or

when during a wide send and there is a chained byte from a previous transfer.

- Cumulative SCSI Byte Count, CSBC (0xDC–0xDF) R/W

  This loadable register counts bytes transferred across the SCSI bus independent of the block move executing. Only counts bytes during data phase transfers.

The SWIDE byte must be flushed manually. Phase Mismatch that occurs when WSS is set condition is handled. PMJAD1 and PMJAD2 are fully static. RBC, UA, ESA and IA only change when a phase mismatch occurs. SBC and CSBC change from block move to block move. You can get stuck in a tight loop in SCRIPTS if you are not careful.

13.6.3 SCRIPTS Example

The following example is a Direct/Table indirect BMOV example, with PMJCTL = 0

```
HandlePhaseMismatchNoWSR:
   CALL REL(Save_cumulative_byte_count)

   CALL REL(Get_msg_bytes_and_ensure_save_pointers)

; Update the Direct/Table Indirect Scatter Gather entry
Update_SG_entry:
   ; Modify Mem to Mem move to update Table indirect entry
   STORE ESA0, 4, Mem2Mem_to_be_patched + 8

; Move the new byte count and address to the entry
Mem2Mem_to_be_patched:
   MOVE MEMORY 8, RBC_addr, 0

   JUMP REL(Do_code_architecture_specific_update)

HandlePhaseMismatchWSR:
   CALL REL(Save_cumulative_byte_count)

   ; If here there is a byte in SWIDE to be moved
   ; Patch the BMOV that will flush SWIDE
   STORE UA0, 4, SWIDE_patch+4

SWIDE_patch:
   ; Using a BMOV here is optimal due to the fact
   ; that there are no alignment restrictions as
```

```
    ; there are in mem2mem moves or stores
          CHMOV 1, 0, WHEN DATA_IN

; Now increment the data address
    MOVE UA + 1 to UA
    MOVE UA + 0 to UA WITH CARRY
    MOVE UA + 0 to UA WITH CARRY
    MOVE UA + 0 to UA WITH CARRY

    ; Jump back up to update the Scatter Gather entry
    JUMP REL(Update_SG_entry)
```

13.7 64-Bit SCRIPTS Addressing

Three extended addressing modes are available.

- Full 64-bit data addressing for direct block moves (bit enabled).
- 64-bit indexed data addressing mode for table indirect block moves (bit enabled).
- 40-bit data addressing mode for table indirect block moves (bit enabled).

Six selectors provide the upper 32-bits of a 64-bit address. If a selector is zero, a single address cycle is issued. If the selector is nonzero then a dual address cycle is issued. Five of the selectors are fully static and the remaining one is semidynamic. For table index mode, the 16 Scratch registers are also available.

Note: Crossing 4-Gbyte boundaries is not supported.

13.7.1 Control Bits

Control bits for 64-bit addressing are located in CCNTL1 register (0x57).

- Bit 2: 64TIMOD, 64-bit Table Indirect Index mode (default = 0)

  Clear: D[28:24] of first Dword of table entry is used as an index to select one of 22 selectors (ScratchC–R, MMRS, MMWS, SFS, DRS, DBMS, or SBMS).

  Set: D[31:24] of first Dword of table entry is used as AD[39:32] to form a 40-bit address.

- Bit 1: EN64TIBMV, Enable 64-bit Table Indirect BMOV (default = 0)

  Enables table indirect block moves to use the upper byte of the first Dword of the table entry for 64-bit addressing. Use of this byte is determined by the setting of 64TIMOD.

- Bit 0: EN64DBMV, Enable 64-bit direct BMOV (default = 0)

  Enables a 64-bit version of a direct block move. When set, all direct block moves are three Dword instructions.

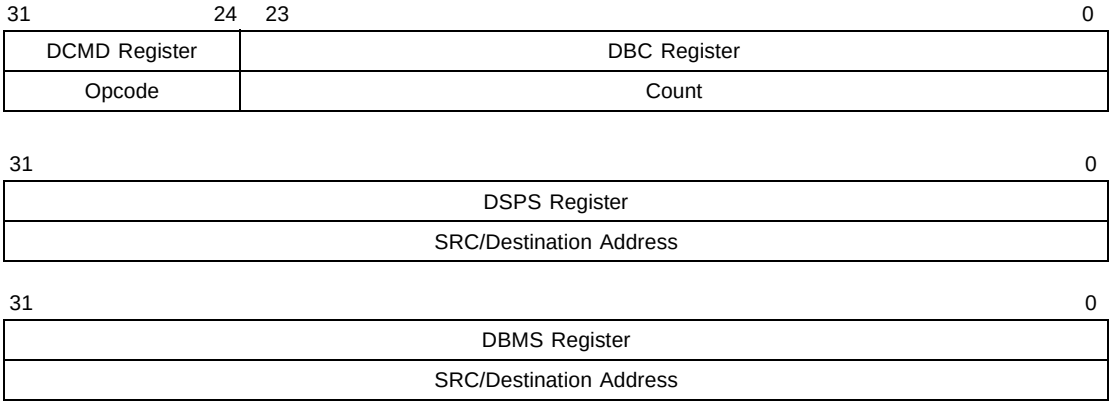
13.7.2 Block Move

By default, BMOV data transfers use the SBMS register. By setting the appropriate control bits, direct BMOVs and table indirect BMOVs can dynamically change the upper 32 (or 8) address bits. Indirect BMOVs always use SBMS.

13.7.3 Direct Block Move

Direct block moves are enabled by setting EN64DBMV. These moves become three Dword instructions where the third Dword is loaded into DBMS.

Figure 13.1 64-Bit Direct Block Move Format



13.7.4 Mode 0 Table Indirect Block Move

Mode 0 is set by setting EN64TIBMV and clearing 64TIMOD, both in the CCNTL1 register. D[28:24] of first Dword of table entry is used as an index to choose one of 22 selectors.

- ScratchC–R
- MMRS
- MMWS
- SFS
- DRS
- SBMS
- DBMS

The chosen selector is moved into DNAD64 (0xB8–0xBB) to form a 64-bit address.

Table 13.2 has the Index Mode 0 table entry format.

13.7.5 Mode 1 Table Indirect Block Move

Table 13.2 Index Mapping

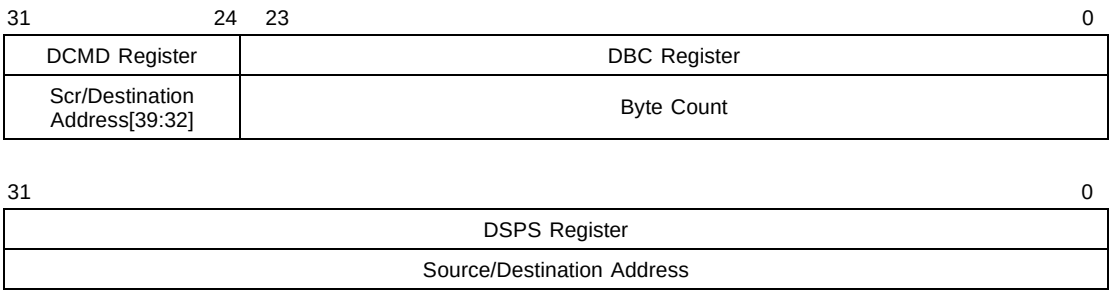
Index	Selector Used
0x00	ScratchC
0x01	ScratchD
0x02	ScratchE
0x03	ScratchF
0x04	ScratchG
0x05	ScratchH
0x06	ScratchI
0x07	ScratchJ
0x08	ScratchK
0x09	ScratchL
0x0A	ScratchM
0x0B	ScratchN
0x0C	ScratchO
0x0D	ScratchP

Table 13.2 Index Mapping (Cont.)

Index	Selector Used
0x0E	ScratchQ
0x0F	ScratchR
0x10	MMRS
0x11	MMWS
0x12	SFS
0x13	DRS
0x4	SBMS
0x15	DBMX
0x16–0x1E	Illegal, results in UD interrupt

Mode 1 is set by writing EN64TIBMV and 64TIMOD. D[31:24] of the first Dword of the table entry is used to create a 40-bit address by copying directly to DNAD64. Refer to Figure 13.2.

Figure 13.2 Index Mode 1 Table Entry Format



13.7.6 Table Indirect Block Move Summary

Table 13.3 summarizes the address locations for a table indirect move.

Table 13.3 Table Indirect BMOV Upper 32-Bit Address Locations

EN64TIBMOV	64TIMOD	Upper 32-bit Address Source
0	0	SBMS
0	1	SBMS
1	0	ScratchC–R, MMWS, MMRS, SPS, DRS, DBMS
1	1	First table entry Dword bits 31–24 (40-bit addressing)

13.7.7 LSI53C1010/LSI53C1010R

This chip supports Ultra3 SCSI, which is enabled in SCNTL4. It uses DT timing. It also has two new SCSI phases.

- DT_DATA_OUT
- DT_DATA_IN

The chip also supports Byte Recovery and shadowed registers SIST0 and SIST1. Specifics for these registers are listed below.

- SIST0

  Pad Request with no CRC Request Following

  Force CRC

  Switch from DT to ST timings during a transfer

  Phase Change with no final CRC Request

  Multiple CRC Requests with the same offset

- SIST1

  Residual Data in SCSI FIFO

  Phase Change with outstanding Offset

  Offset Overflow

  Offset Underflow

  Data Overflow

  Data Underflow

Appendix A
NASM Error Messages

Table A.1 NASM Error Messages

Error	Description
24-bit value expected	The value specified is not within the range of a 24-bit unsigned integer. The value must be between 0 and 4 Mbytes. Something other than a value was found.
8-bit value expected	The value specified is not within the range of an 8-bit unsigned integer. The value must be between 0 and 255. Something other than a value was found.
ACK, ATN, TARGET or CARRY expected string	String was found instead of ACK, ATN, TARGET or CARRY.
AND or OR expected string	String was found instead of AND or OR.
ATN specified multiple times	The ATN field may only be specified once per instruction.
Cannot compare CARRY and Data	The command is requesting that both a comparison of the SFBR register to the specified data and a test of the carry bit take place, but only one test is allowed.
Cannot compare PHASE and Data	The command is requesting that both a comparison of the SCSI bus phase and a comparison of the SFBR register to the specified data take place, but only one test is allowed.
Cannot specify PHASE when using ATN	The use of PHASE and ATN are mutually exclusive.
Cannot use MASK without compare Data	Valid Data must be present when using the MASK field.
Cannot use Pass for count address	The PASS feature cannot be used in a count field. The current format of the output file does not support this.
Carry operations not available on LSI53C700 architectures	The Carry feature is only available on the LSI53C710 or higher architectures.
CARRY specified multiple times	CARRY may only be specified once per instruction.

Table A.1 NASM Error Messages (Cont.)

Error	Description
CHMOV LSI53C720, LSI53C770, LSI53C82X, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The CHMOV instruction is only available on the chips that support wide SCSI.
Comma expected string	String was found instead of a comma.
CTEST7 LSI53C700 and LSI53C710 architectures only	The CTEST7 register is only available on the LSI53C700/710 architectures.
CTEST8 LSI53C700 and LSI53C710 architectures only	The CTEST8 register is only available on the LSI53C700/710 architectures.
Data list expected string	String was found instead of a list of initialized data.
Data specified multiple times	The Data field may only be specified once for a given instruction.
Data specifier expected string	String was found instead of a Data specifier. A Data specifier is used to specify the size of a data area and to initialize that data area.
Declaration expected string	String was found when a declaration was expected. A declaration is an assignment of a variable to some value or data specifier.
Divide or mod by zero	
DSAREL: LSI53C810A, LSI53C825A, LSI53C860, LSI53C875, and LSI53C895 architectures only	The DSAREL keyword is only supported by the chips that support Load and Store instructions.
Entry identifier expected name	Name was found instead of an identifier. An identifier is a symbol that has not previously been declared.
Expression must evaluate to a constant string	A label, relative, external, or an undeclared identifier, string, does not evaluate to a known value. The value must be known at assembly time.
Expression or External expected	
GPCNTL LSI53C720, LSI53C770, and LSI53C8XX only	The GPCNTL register is available only on the LSI53C720 and higher architectures.
GPREG LSI53C720, LSI53C770, and LSI53C8XX architectures only	The GPREG register is only available on the LSI53C720 and higher architectures.

Table A.1 NASM Error Messages (Cont.)

Error	Description
ID specifier only valid for table entries	
IF or WHEN expected string	String was found instead of one of IF or WHEN.
INTFLY: LSI53C720, LSI53C770, and LSI53C8XX architecture only	The INTFLY instruction is only available on the LSI53C720 and higher architectures.
Invalid Address string	String was found instead of a valid address. A valid address is an expression, external, relative, table, or an absolute.
Invalid assignment	
Invalid character/s	
Invalid constant type	
Invalid destination address string	String was found instead of a valid destination address. A valid destination address is an expression, external, relative, table or an absolute.
Invalid register operator string	String was found instead of a valid operator. Valid operators are '+', '–', '|', '&'.
Invalid register value	Value must be in the range 0x0–0x3F for the LSI53C700/710 and 0x0–0x5C for the LSI53C720.
Invalid SCSI id	Value must have only one bit set (bits [0:7]) for the LSI53C700/710/810. Must be in the range of [0:15] for the LSI53C720/820/825.
Invalid syntax string	String was found and not expected causing an unknown syntax error.
Invalid test condition string	String was found instead of a valid test condition. The valid test conditions are CARRY, a PHASE, an 8-bit value, or a MASK.
LCRC LSI53C710 architectures only	The LCRC register is only available on the LSI53C710 architecture.
Left parenthesis expected string	String was found instead of a left parentheses.
LOAD: LSI53C810A, LSI53C825A, LSI53C860, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures	The LOAD instruction is only supported by the LSI53C810A and higher architectures.
LOAD: Count must not exceed 4 bytes	Four bytes is the maximum byte count to LOAD.

Table A.1 NASM Error Messages (Cont.)

Error	Description
Logical end of line '\' expected string	A logical line separator is needed before continuing the directive on a new line.
MACNTL LSI53C720, LSI53C770, and LSI53C8XX architectures only	The MACNTL register is available only on LSI53C720 and higher architectures.
MASK specified multiple times	MASK may only be specified once per instruction.
Memory Move operations not available on LSI53C700 architectures	The Memory Move instruction is only available on LSI53C710 and higher architectures.
Memory Move Noflush only available on LSI53C810A, LSI53C825A, LSI53C860, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures	The No Flush option is only available in the LSI53C810A and higher architectures.
Old EXTERNAL directive, use new EXTERNAL directive string	When the Debug switch is on, the operand string must be declared with the new EXTERNAL directive syntax. The new syntax informs the debugger of the size of the external variable.
Old RELATIVE directive, use new RELATIVE directive string	When the Debug switch is on, the operand string must be declared with the new RELATIVE directive syntax. The new syntax informs the debugger of the size of the relative data area.
One register must be SFBR or both the same	The Register Move instruction requires either the source or destination register be the SFBR register, or that both the source and destination be the same register.
Only use CARRY with Addition or Subtraction	The CARRY bit can only be checked when either an addition or subtraction operation is used.
Operand must be a TABLE entry string	When the Debug switch is on, the operand where the string resides must be of type TABLE entry. This is used for table indirect addressing and to inform the debugger about the size of the table.
Parenthesis must match when PASS is used as an argument	When a PASS variable is used as an argument the parentheses must match.
PHASE expected string	String was found instead of a PHASE.

Table A.1 NASM Error Messages (Cont.)

Error	Description
PHASE specified multiple times	
Redeclaration of Label string	The string has previously been declared as a label or some other type of identifier other than an ENTRY.
Redeclaration of TABLE identifier	The string has previously been declared as a TABLE name or some other type of identifier. Only one TABLE declaration per source file is allowed.
Register or Data24 value expected string	String was found instead of a register or a 24-bit value.
Register right of operand must be SFBR	In a Move to SFBR operation, SFBR must be to the right of the operand.
Relative addressing not available on LSI53C700 architecture	Relative addressing is not supported by the LSI53C700 architecture.
RESPID LSI53C81X architecture only	The RESPID register is only one byte in the LSI53C810.
RESPID0 LSI53C720, LSI53C770, LSI53C82X, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The RESPID0 register is only available in devices that support Wide SCSI.
RESPID1 LSI53C720, LSI53C770, LSI53C82X, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The RESPID1 register is only available in devices that support Wide SCSI.
Right parenthesis expected string	String was found instead of a right parentheses.
SBDL LSI53C700, LSI53C710, and LSI53C81X architectures only	The SBDL register is only one byte in the LSI53C700, LSI53C710, and LSI53C81X architectures.
SBDL0 LSI53C720, LSI53C770, LSI53C82X, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SBDL register is two bytes in the devices that support Wide SCSI.
SBDL1 LSI53C720, LSI53C770, LSI53C82X, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SBDL register is two bytes in the devices that support Wide SCSI.

Table A.1 NASM Error Messages (Cont.)

Error	Description
SCNTL2 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SCNTL2 register is only available on the LSI53C720 and higher architectures.
SCNTL3 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SCNTL3 register is only available on the LSI53C720 and higher architectures.
Scratch0 LSI53C710 architectures only	The SCRATCH0 register is only available on the LSI53C710 architecture.
Scratch1 LSI53C710 architectures only	The SCRATCH1 register is only available on the LSI53C710 architecture.
Scratch2 LSI53C710 architectures only	The SCRATCH2 register is only available on the LSI53C710 architecture.
Scratch3 LSI53C710 architectures only	The SCRATCH3 register is only available on the LSI53C710 architecture.
Scratcha0 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SCRATCHA0 register is only available on the LSI53C720 and higher architectures.
Scratcha1 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SCRATCHA1 register is only available on the LSI53C720 and higher architectures.
Scratcha2 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SCRATCHA2 register is only available on the LSI53C720 and higher architectures.
Scratcha3 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SCRATCHA3 register is only available on the LSI53C720 and higher architectures.
Scratchb0 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SCRATCHB0 register is only available on the LSI53C720 and higher architectures.
Scratchb1 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SCRATCHB1 register is only available on the LSI53C720 and higher architectures.
Scratchb2 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SCRATCHB2 register is only available on the LSI53C720 and higher architectures.

Table A.1 NASM Error Messages (Cont.)

Error	Description
Scratchb3 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SCRATCHB3 register is only available on the LSI53C720 and higher architectures.
Scratchc0 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchc1 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchc2 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchc3 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchd0 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchd1 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchd2 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchd3 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratche0 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.

Table A.1 NASM Error Messages (Cont.)

Error	Description
Scratche1 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratche2 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratche3 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchf0 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchf1 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchf2 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchf3 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchg0 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchg1 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.

Table A.1 NASM Error Messages (Cont.)

Error	Description
Scratchg2 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchg3 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchh0 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchh1 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchh2 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchh3 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchi0 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchi1 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchi2 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.

Table A.1 NASM Error Messages (Cont.)

Error	Description
Scratchi3 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchj0 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchj1 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchj2 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
Scratchj3 LSI53C770, LSI53C825A, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SCRATCHC–J registers are only available on the LSI53C770, LSI53C825A, LSI53C875, and LSI53C895 architectures.
SELID0: LSI53C720, LSI53C770, and LSI53C8XX architectures only	
SELID1: LSI53C720, LSI53C770, and LSI53C8XX architectures only	
Separator expected ',' or '\\'	A comma or a logical line separator is needed to delimit declarations.
SIDL LSI53C700, LSI53C710, and LSI53C81X architectures only	The SIDL register is only one byte on the LSI53C700, LSI53C710, and LSI53C81X chips.
SIDL0 LSI53C720, LSI53C770, LSI53C82X, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SIDL register is two bytes on the chips that support Wide SCSI.
SIDL1 LSI53C720, LSI53C770, LSI53C82X, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SIDL register is two bytes on the chips that support Wide SCSI.
SHL LSI53C720, LSI53C770, and LSI538XX architectures only	The shift left instruction is only supported on LSI53C720 and higher architectures.

Table A.1 NASM Error Messages (Cont.)

Error	Description
SHR LSI53C720, LSI53C770, and LSI53C8XX architectures only	The shift right instruction is only supported on LSI53C720 and higher architectures.
SIEN LSI53C700 and LSI53C710 architectures only	The SIEN register is only available on the LSI53C700/710 architectures.
SIEN0 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SIEN0 register is only available on the LSI53C720 and higher architectures.
SIEN1 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SIEN1 register is only available on the LSI53C720 and higher architectures.
SIST0 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SIST0 register is only available on the LSI53C720 and higher architectures.
SIST1 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SIST1 register is only available on the LSI53C720 and higher architectures.
SLPAR LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SLPAR register is only available on the LSI53C720 and higher architectures.
SODL LSI53C700, LSI53C710, and LSI53C81X architectures only	The SODL register is one byte only on the LSI53C700, LSI53C710, and LSI53C81X chips.
SODL0 LSI53C720, LSI53C770, LSI53C82X, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SODL register is two bytes on the chips that support Wide SCSI.
SODL1 LSI53C720, LSI53C770, LSI53C82X, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SODL register is two bytes on the chips that support Wide SCSI.
SSID LSI53C720, LSI53C770, and LSI53C8XX architectures only	The SSID register is only available on the LSI53C720 and higher architectures.
STEST0 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The STEST0 register is only available on the LSI53C720 and higher architectures.

Table A.1 NASM Error Messages (Cont.)

Error	Description
STEST1 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The STEST1 register is only available on the LSI53C720 and higher architectures.
STEST2 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The STEST2 register is only available on the LSI53C720 and higher architectures.
STEST3 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The STEST3 register is only available on the LSI53C720 and higher architectures.
STEST4 LSI53C895 architecture only	The STEST4 register is only available on the LSI53C895.
STIME0 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The STIME0 register is only available on the LSI53C720 and higher architectures.
STIME1 LSI53C720, LSI53C770, and LSI53C8XX architectures only	The STIME1 register is only available on the LSI53C720 and higher architectures.
STORE: LSI53C810A, LSI53C825A, LSI53C860, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures	The STORE instruction is only supported by the LSI53C810A and higher architectures.
STORE: Count must not exceed 4 bytes	Four bytes is the maximum byte count to STORE.
SWIDE LSI53C720, LSI53C82X, LSI53C875, LSI53C876, LSI53C885, and LSI53C895 architectures only	The SWIDE register is only available on the LSI Logic SCSI processors that support wide SCSI.
TABLE directive not available on LSI53C700 architecture	Table indirect operations are not supported by the LSI53C700.
Table indirect operations not available on LSI53C700 architecture	Table indirect addressing is not supported by the LSI53C700 architecture.
Table name expected string	The directive TABLE was found without a table name declaration.
Unexpected EOF	End of file was found when not expected.

Table A.1 NASM Error Messages (Cont.)

Error	Description
Unresolved Label or Identifier string	String was used but never declared as a label, external, relative, absolute or table.
WITH or WHEN expected	
XOR LSI53C720, LSI53C810, and LSI53C825 only	XOR operations are only supported on LSI53C720 and higher architectures

Table A.2 Fatal Errors

Error	Description
Fatal Error allocating input file buffer(s)	
Fatal File Not found	The file named filename was not found in the path specified.
Fatal Memory allocation error	Not enough dynamic memory available to complete assembly of the file. Try dividing up file or freeing memory.
Fatal No source file specified	A source file to assemble must be specified on the command line. Try specifying source files first before options.
Fatal Opening file	The filename specified cannot be opened for some unknown reason.
Fatal read permission denied for file	The filename specified cannot be opened with read access.

Table A.3 Warnings

Error	Description
ACK specified multiple times	The ACK bit can only be specified once per instruction.
ATN specified multiple times	The ATN bit can only be specified once per instruction.
Cannot extract pass information correctly	The pass variable is poorly formatted and may not have been correctly interpreted.
CARRY specified multiple times	The CARRY bit can only be specified once per instruction.
Initializer value truncated to byte value	Initialization of data by byte offset only.
Debug record contains old format EXTERNAL statement, data size unknown	Use the new style EXTERNAL directive where data specifiers are used.
Debug record contains old format RELATIVE statement, Data size unknown	Use the new style RELATIVE directive where data specifiers are used.
Initializer value truncated to byte	
Possible truncation of constant value	The value of the constant may have been truncated. This is caused by the ASCII conversion of the value.
Relative offset value truncated	
Source and .bin file have the same name	The binary file has the same name as the source. The binary file will be renamed or not created.
Source and Error file have the same name	The error file has the same name as the source. The error file will be renamed or not created.
Source and listing file have the same name	The listing file has the same name as the source. The listing file will be renamed or not created.
Source and Object file have the same name	The object file and source file have the same name. The object file will be renamed or not created.
Source and Out file have the same name	The output file and source file have the same name. The output file will be renamed or not created.
TARGET specified multiple times	The TARGET bit can only be specified once per instruction.

Appendix B
Multithreaded SCRIPTS Example

```
;************************************************************
********
; 53C810 MULTI THREAD EXAMPLE
;************************************************************
********; ABSOLUTE declarations

ABSOLUTE SCSI_id       = 0
ABSOLUTE MATCH_SCSI_ID = 0x81

; Messages
ABSOLUTE  CMD_COMPLETE_ =     0x00
ABSOLUTE  EXTEND_MSG_=        0x01
ABSOLUTE  SAVE_DATAPTR_=      0x02
ABSOLUTE  DISCONNECT_=        0x04
ABSOLUTE  MSG_REJECT_=        0x07

; Interrupt codes
ABSOLUTE  error_not_cmd_phase =      0x01
ABSOLUTE  error_not_data_in_phase =  0x02
ABSOLUTE  error_not_data_out_phase = 0x03
ABSOLUTE  error_not_msg_in_phase =   0x04
ABSOLUTE  error_not_msg_out_phase =  0x05
ABSOLUTE  error_not_status_phase =   0x06
ABSOLUTE  error_unexpected_phase =   0x07
ABSOLUTE  error_jump_not_taken =     0x10
ABSOLUTE  error_not_cmd_complete =   0x20
ABSOLUTE  error_not_extended_msg =   0x21
ABSOLUTE  io_complete =              0x0A
ABSOLUTE  setup_SXFER =              0x888
ABSOLUTE  reselect_id_error =        0x999
ABSOLUTE  select_error =             0xfff
;************************************************************
********
; TABLE declarations for Table Indirect offsets in bytes
Table Table_Indirect                                        \
SCSI_ID=ID{0x00,0x00,0x00,0x00}, \
identify _msg_buf = {0xc0},                                  \
```

```
synch_msgi_buf  = 5{??},                                        \
cmd_buf = 12{??},                                                \
status_buf = 1{??},                                              \
msg_in_buf = 1{??},                                              \
data_buf = 512{??}

;************************************************************
********
; ENTRY declarations
ENTRY multi_thread
ENTRY to_decisions
ENTRY id_msg_out
ENTRY msg_in_phase
 ENTRY cmd_phase
ENTRY data_in_phase
ENTRY data_out_phase
ENTRY status_phase
ENTRY disconnected
ENTRY entry0
ENTRY entry1
ENTRY entry2
ENTRY io_request0
ENTRY io_request1
ENTRY io_request2
ENTRY schedule_NOP

; Scheduler SCRIPT code
scheduler:

entry0:
   ;Initialize DSA register with table base address for
using table
   ;indirect addressing
   MOVE MEMORY 4, PATCH_addr_of_table0_ptr,
PATCH_chip_physaddr+DSA
   ;Initilize address for changing jump to nop after
starting new I/O
   ;(after SELECT instruction in main SCRIPT code)
   MOVE MEMORY 4,
PATCH_SCRIPTphysaddr+io_request0,PATCH_SCRIPTphysaddr+schedu
le_NOP+8
io_request0:
   JUMP REL(multi_thread)

entry1:
   MOVE MEMORY 4, PATCH_addr_of_table1_ptr,
PATCH_chip_physaddr+DSA
   MOVE MEMORY 4,
```

```
PATCH_SCRIPTphysaddr+io_request1,PATCH_SCRIPTphysaddr+schedu
le_NOP+8
io_request1:
    JUMP REL(multi_thread)

entry2:
    MOVE MEMORY 4, PATCH_addr_of_table2_ptr,
PATCH_chip_physaddr+DSA
    MOVE MEMORY 4,
PATCH_SCRIPTphysaddr+io_request2,PATCH_SCRIPTphysaddr+schedu
le_NOP+8
io_request2:
    JUMP REL(multi_thread)

    JUMP REL(wait_for_reselect)


;*********************************************************
********
; main SCRIPT code
multi_thread:
    SELECT ATN FROM SCSI_id, REL(wait_for_reselect)

;Change jump to nop in scheduler after starting new I/O
;the destination address is initialized from scheduler
SCRIPT
schedule_NOP:
    MOVE MEMORY 4, PATCH_nop_physaddr, PATCH_place_hold_addr

    JUMP REL(to_decisions), WHEN NOT MSG_OUT

id_msg_out:
    MOVE FROM identify_msg_buf, WHEN MSG_OUT
    JUMP REL(to_decisions), WHEN NOT CMD

cmd_phase:
    CLEAR ATN
    MOVE FROM cmd_buf, WHEN CMD
    JUMP REL(to_decisions), WHEN NOT DATA_IN

data_in_phase:
    MOVE FROM data_buf, WHEN DATA_IN
    JUMP REL(status_phase), WHEN STATUS
    JUMP REL(to_decisions)

data_out_phase:
    MOVE FROM data_buf, WHEN DATA_OUT
    JUMP REL(to_decisions), WHEN NOT STATUS
```

```
status_phase:
    MOVE FROM status_buf, WHEN STATUS
    JUMP REL(to_decisions), WHEN NOT MSG_IN
msg_in_phase:
    MOVE FROM msg_in_buf, WHEN MSG_IN
    JUMP REL(disconnected), IF DISCONNECT_
    JUMP REL(msg_in_phase), WHEN SAVE_DATAPTR_ ;compare data,
wait for phase
    INT error_not_cmd_complete, IF NOT 0x00
    CLEAR ACK
    MOVE SCNTL2 & 0x7F TO SCNTL2
    WAIT DISCONNECT
    INT io_complete
disconnected:
    MOVE SCNTL2 & 0x7F TO SCNTL2
    WAIT DISCONNECT
    JUMP REL(wait_for_reselect)


to_decisions:
    JUMP REL(msg_in_phase),         WHEN MSG_IN
    JUMP REL(cmd_phase),            IF CMD
    JUMP REL(data_in_phase),        IF DATA_IN
    JUMP REL(data_out_phase),       IF DATA_OUT
    JUMP REL(status_phase),         IF STATUS
    INT error_unexpected_phase


;Reselect SCRIPT code
wait_for_reselect:
    WAIT RESELECT REL(CPU_set_SIGP)

SCSI_id_jump_table:
    MOVE SSID TO SFBR
    JUMP REL(id_0), IF 0x00 ;
    JUMP REL(id_1), IF 0x01
    JUMP REL(id_2), IF 0x02
    INT reselect_id_error

id_0:
    MOVE MEMORY 4, PATCH_addr_of_table0_ptr,
PATCH_chip_physaddr+DSA
    ;initialize SXFER for synchronous transfers from table
    MOVE MEMORY
1,PATCH_addr_of_table0+2,PATCH_chip_physaddr+SXFER
    MOVE MEMORY
1,PATCH_addr_of_table0,PATCH_chip_physaddr+SCNTL3
```

```
    ; This will set up the clock dividers as defined in the
SCNTL3 register
    MOVE FROM identify_msg_buf, WHEN MSG_IN
    CLEAR ACK
    JUMP REL(to_decisions)

id_1:
    MOVE MEMORY 4, PATCH_addr_of_table1_ptr,
PATCH_chip_physaddr+DSA
    ;initialize SXFER for synchronous transfers from table
    MOVE MEMORY 1,
PATCH_addr_of_table1+2,PATCH_chip_physaddr+SXFER
    MOVE MEMORY
1,PATCH_addr_of_table1,PATCH_chip_physaddr+SCNTL3
    ; This will set up the clock dividers as defined in the
SCNTL3 register
    MOVE FROM identify_msg_buf, WHEN MSG_IN
    CLEAR ACK
    JUMP REL(to_decisions)

id_2:
    MOVE MEMORY 4, PATCH_addr_of_table2_ptr,
PATCH_chip_physaddr+DSA
    ;initialize SXFER for synchronous transfers from table
    MOVE MEMORY
1,PATCH_addr_of_table2+2,PATCH_chip_physaddr+SXFER
    MOVE MEMORY
1,PATCH_addr_of_table2,PATCH_chip_physaddr+SCNTL3
    ; This will set up the clock dividers as defined in the
SCNTL3 register
    MOVE FROM identify_msg_buf, WHEN MSG_IN
    CLEAR ACK
    JUMP REL(to_decisions)


CPU_set_SIGP:
    JUMP scheduler
```

Appendix C
Glossary of Terms and Abbreviations

160/m	An industry initiative extension of the Ultra160 SCSI specification that requires support of Double Transition (DT) Clocking, Domain Validation, and Cyclic Redundancy Check (CRC).
Active Termination	The electrical connection required at each end of the SCSI bus, composed of active voltage regulation and a set of termination resistors. Ultra, Ultra2, and Ultra160 SCSI require active termination.
Address	A specific location in memory, designated either numerically or by a symbolic name.
AIP	Asynchronous Information Protection provides error checking for asynchronous, nondata phases of the SCSI bus.
Asynchronous Data Transfer	One of the ways data is transferred over the SCSI bus. It is slower than synchronous data transfer.
BIOS	Basic Input/Output System. Software that provides basic read/write capability. Usually kept as firmware (ROM based). The system BIOS on the mainboard of a computer is used to boot and control the system. The SCSI BIOS on your host adapter acts as an extension of the system BIOS.
Bit	A binary digit. The smallest unit of information a computer uses. The value of a bit (0 or 1) represents a two-way choice, such as on or off, true or false, and so on.
Bus	A collection of unbroken signal lines across which information is transmitted from one part of a computer system to another. Connections to the bus are made using taps on the lines.

Bus Mastering A high-performance way to transfer data. The host adapter controls the transfer of data directly to and from system memory without interrupting the computer's microprocessor. This is the fastest way for multitasking operating systems to transfer data.

Byte A unit of information consisting of eight bits.

CISPR A special international committee on radio interference (Committee, International and Special, for Protection in Radio).

Configuration Refers to the way a computer is set up; the combined hardware components (computer, monitor, keyboard, and peripheral devices) that make up a computer system; or the software settings that allow the hardware components to communicate with each other.

CRC Cyclic Redundancy Check is an error detection code used in Ultra160 SCSI. Four bytes are transferred with the data to increase the reliability of data transfers. CRC is used on the Double Transition (DT) Data In and DT Data Out phases.

CPU Central Processing Unit. The "brain" of the computer that performs the actual computations. The term Microprocessor Unit (MPU) is also used.

DMA Bus Master A feature that allows a peripheral to control the flow of data to and from system memory by blocks, as opposed to PIO (Programmed I/O) where the processor is in control and the flow is by byte.

Device Driver A program that allows a microprocessor (through the operating system) to direct the operation of a peripheral device.

Differential SCSI A hardware configuration for connecting SCSI devices. It uses a pair of lines for each signal transfer (as opposed to Single-Ended SCSI which references each SCSI signal to a common ground).

Domain Validation Domain Validation is a software procedure in which a host queries a device to determine its ability to communicate at the negotiated Ultra160 data rate.

Double Transition (DT) Clocking In Double Transition Clocking data is sampled on both the asserting and deasserting edge of the REQ/ACK signal. DT clocking may only be implemented on an LVD SCSI bus.

Dword	A doubleword is a group of four consecutive bytes or characters that are stored, addressed, transmitted, and operated on as a unit. The lower two address bits of the least significant byte must equal zero in order to be Dword aligned.
EEPROM	Electronically Erasable Programmable Read Only Memory. A memory chip typically used to store configuration information. See NVRAM.
EISA	Extended Industry Standard Architecture. An extension of the 16-bit ISA bus standard. It allows devices to perform 32-bit data transfers.
External SCSI Device	A SCSI device installed outside the computer cabinet. These devices are connected in a continuous chain using specific types of shielded cables.
Fast-20	The SCSI Trade Association (STA) supports the use of “Ultra SCSI” over the term “Fast-20”. Please see Ultra SCSI.
Fast-40	The SCSI Trade Association (STA) supports the use of “Ultra2 SCSI” over the term “Fast-40”. Please see Ultra2 SCSI.
Fast SCSI	A standard for SCSI data transfers. It allows a transfer rate of up to 10 Mbytes/s over an 8-bit SCSI bus and up to 20 Mbytes/s over a 16-bit SCSI bus.
FCC	Federal Communications Commission.
File	A named collection of information stored on a disk.
Firmware	Software that is permanently stored in ROM. Therefore, it can be accessed during boot time.
Hard Disk	A disk made of metal and permanently sealed into a drive cartridge. A hard disk can store very large amounts of information.
Host	The computer system in which a SCSI host adapter is installed. It uses the SCSI host adapter to transfer information to and from devices attached to the SCSI bus.
Host Adapter	A circuit board or integrated circuit that provides a SCSI bus connection to the computer system.
Internal SCSI Device	A SCSI device installed inside the computer cabinet. These devices are connected in a continuous chain using an unshielded ribbon cable.

IRQ	Interrupt Request Channel. A path through which a device can get the immediate attention of the computer's CPU. The PCI bus assigns an IRQ path for each SCSI host adapter.
ISA	Industry Standard Architecture. A type of computer bus used in most PCs. It allows devices to send and receive data up to 16 bits at a time.
Kbyte	Kilobyte. A measure of computer storage equal to 1024 bytes.
Local Bus	A way to connect peripherals directly to computer memory. It bypasses the slower ISA and EISA buses. PCI is a local bus standard.
Logical Unit	A subdivision, either logical or physical, of a SCSI device (actually the place for the device on the SCSI bus). Most devices have only one logical unit, but up to eight are allowed for each of the eight possible devices on a SCSI bus.
LUN	Logical Unit Number. An identifier, zero to seven, for a logical unit.
LVD Link	Low Voltage Differential Link allows greater Ultra2 SCSI device connectability and longer SCSI cables. LVD Link lowers the amplitude of noise reflections and allows higher transmission frequencies.
Mainboard	A large circuit board that holds RAM, ROM, the microprocessor, custom integrated circuits, and other components that make a computer work. It also has expansion slots for host adapters and other expansion boards.
Main Memory	The part of a computer's memory which is directly accessible by the CPU (usually synonymous with RAM).
Mbyte	Megabyte. A measure of computer storage equal to 1024 kilobytes.
Motherboard	See Mainboard. In some countries, the term Motherboard is not appropriate.
Multitasking	The executing of more than one command at the same time. This allows programs to operate in parallel.
Multithreading	The simultaneous accessing of data by more than one SCSI device. This increases the data throughput.
NVRAM	NonVolatile Random Access Memory. Actually an EEPROM (Electronically Erasable Read Only Memory chip) used to store configuration information. See EEPROM.

Operating System — A program that organizes the internal activities of the computer and its peripheral devices. An operating system performs basic tasks such as moving data to and from devices, and managing information in memory. It also provides the user interface.

Parity Checking — A way to verify the accuracy of data transmitted over the SCSI bus. The parity bit in the transfer is used to make the sum of all the 1 bits either odd or even (for odd or even parity). If the sum is not correct, the information may be retransmitted or an error message may appear.

Passive Termination — The electrical connection required at each end of the SCSI bus, composed of a set of resistors. It improves the integrity of bus signals.

PCI — Peripheral Component Interconnect. A local bus specification that allows connection of peripherals directly to computer memory. It bypasses the slower ISA and EISA buses.

Peripheral Devices — A piece of hardware (such as a video monitor, disk drive, printer, or CD-ROM) used with a computer and under the computer's control. SCSI peripherals are controlled through a SCSI host adapter.

Pin-1 Orientation — The alignment of pin 1 on a SCSI cable connector and the pin-1 position on the SCSI connector into which it is inserted. External SCSI cables are always keyed to insure proper alignment, but internal SCSI ribbon cables sometimes are not keyed.

PIO — Programmed Input/Output. A way the CPU can transfer data to and from memory using the computer's I/O ports. PIO is usually faster than DMA, but requires CPU time.

Port Address — Also Port Number. The address through which commands are sent to a host adapter board. This address is assigned by the PCI bus.

Port Number — See Port Address.

Queue Tags — A way to keep track of multiple commands that allow for increased throughput on the SCSI bus.

RAM Random Access Memory. The computer's primary working memory in which program instructions and data are stored and are accessible to the CPU. Information can be written to and read from RAM. The contents of RAM are lost when the computer is turned off.

RISC Core LSI Logic SCSI chips contain a RISC (Reduced Instruction Set Computer) processor, programmed through microcode SCRIPTS.

ROM Read Only Memory. Memory from which information can be read but not changed. The contents of ROM are not erased when the computer is turned off.

SCAM SCSI Configured AutoMatically. A method to automatically allocate SCSI IDs using software when SCAM compliant SCSI devices are attached.

SCSI Small Computer System Interface. A specification for a high-performance peripheral bus and command set. The original standard is referred to as SCSI-1.

SCSI-2 The SCSI specification which adds features to the original SCSI standard.

SCSI-3 The current SCSI specification which adds features to the SCSI-2 standard.

SCSI Bus A host adapter and one or more SCSI peripherals connected by cables in a linear chain configuration. The host adapter may exist anywhere on the chain, allowing connection of both internal and external SCSI devices. A system may have more than one SCSI bus by using multiple host adapters.

SCSI Device Any device that conforms to the SCSI standard and is attached to the SCSI bus by a SCSI cable. This includes SCSI host adapters and SCSI peripherals.

SCSI ID A way to uniquely identify each SCSI device on the SCSI bus. Each SCSI bus has eight available SCSI IDs numbered 0 through 7 (or 0 through 15 for Wide SCSI). The host adapter usually gets the highest ID (7 or 15) giving it priority to control the bus.

SCSI SCRIPTS A SCSI programming language that works with the SCRIPTS processor that is embedded on the LSI53C7XX, LSI53C8XX, or LSI53C10XX device. These SCRIPTS reside in in host computer system memory.

SCRIPTS Processor	The SCRIPTS processor allows users to fine tune SCSI operations with regard to unique vendor commands or new SCSI specifications. The SCRIPTS processor fetches SCRIPTS instructions from system memory to control operation of the LSI53C7XX, LSI53C8XX, or LSI53C10XX device.
Single-Ended SCSI	A hardware specification for connecting SCSI devices. It references each SCSI signal to a common ground. This is the most common method (as opposed to differential SCSI which uses a separate ground for each signal).
STA	SCSI Trade Association. A group of companies that cooperate to promote SCSI parallel interface technology as a viable mainstream I/O interconnect for commercial computing.
Synchronous Data Transfer	One of the ways data is transferred over the SCSI bus. Transfers are clocked with fixed frequency pulses. This is faster than asynchronous data transfer. Synchronous data transfers are negotiated between the SCSI host adapter and each SCSI device.
System BIOS	Controls the low-level POST (Power-On Self-Test), and basic operation of the CPU and computer system.
TolerANT Technology	A technology developed and used by LSI Logic to improve data integrity, data transfer rates, and noise immunity through the use of active negation and input signal filtering.
Ultra SCSI	A standard for SCSI data transfers. It allows a transfer rate of up to 20 Mbytes/s over an 8-bit SCSI bus and up to 40 Mbytes/s over a 16-bit SCSI bus. SCSI Trade Association (STA) supports using the term “Ultra SCSI” over the older term “Fast-20”.
Ultra2 SCSI	A standard for SCSI data transfers. It allows a transfer rate of up to 40 Mbytes/s over an 8-bit SCSI bus, and up to 80 Mbytes/s over a 16-bit SCSI bus. SCSI Trade Association (STA) supports using the term “Ultra2 SCSI” over the term “Fast-40”.
Ultra160 SCSI	A standard for SCSI data transfers. It allows a transfer rate of up to 160 Mbytes/s over a 16-bit SCSI bus.
VCCI	Voluntary Control Council for Interference.
VDE	Verband Deucher Elektroniker (Association of German Electrical Engineers).

Virtual Memory	Space on a hard disk that can be used as if it were RAM.
Wide SCSI	A SCSI-2 feature allowing 16-bit or 32-bit transfers on the SCSI bus. This dramatically increases the transfer rate over the standard 8-bit SCSI bus.
Wide Ultra SCSI	The SCSI Trade Association (STA) term for SCSI bus width 16-bits, SCSI bus speed maximum data rate 40 Mbytes/s.
Wide Ultra2 SCSI	The SCSI Trade Association (STA) term for SCSI bus width 16-bits, SCSI bus speed maximum data rate 80 Mbytes/s.
Word	A two byte (or 16-bit) unit of information.

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Customer Feedback

We would appreciate your feedback on this document. Please copy the following page, add your comments, and fax it to us at the number shown.

If appropriate, please also fax copies of any marked-up pages from this document.

Important: Please include your name, phone number, fax number, and company address so that we may contact you directly for clarification or additional information.

Thank you for your help in improving the quality of our documents.

Reader's Comments

Fax your comments to: LSI Logic Corporation
Technical Publications
M/S E-198
Fax: 408.433.4333

Please tell us how you rate this document: *SCSI SCRIPTS™ Processors Programming Guide.* Place a check mark in the appropriate blank for each category.

	Excellent	Good	Average	Fair	Poor
Completeness of information	____	____	____	____	____
Clarity of information	____	____	____	____	____
Ease of finding information	____	____	____	____	____
Technical content	____	____	____	____	____
Usefulness of examples and illustrations	____	____	____	____	____
Overall manual	____	____	____	____	____

What could we do to improve this document?

_____

_____

_____

_____

If you found errors in this document, please specify the error and page number. If appropriate, please fax a marked-up copy of the page(s).

_____

_____

_____

Please complete the information below so that we may contact you directly for clarification or additional information.

Name _____ Date _____

Telephone _____ Fax _____

Title _____

Department _____ Mail Stop _____

Company Name _____

Street _____

City, State, Zip _____

U.S. Distributors by State

A. E. Avnet Electronics
http://www.hh.avnet.com
B. M. Bell Microproducts, Inc. (for HAB's)
http://www.bellmicro.com
I. E. Insight Electronics
http://www.insight-electronics.com
W. E. Wyle Electronics
http://www.wyle.com

Alabama
Daphne
I. E. Tel: 334.626.6190
Huntsville
A. E. Tel: 256.837.8700
B. M. Tel: 256.705.3559
I. E. Tel: 256.830.1222
W. E. Tel: 800.964.9953

Alaska
A. E. Tel: 800.332.8638

Arizona
Phoenix
A. E. Tel: 480.736.7000
B. M. Tel: 602.267.9551
W. E. Tel: 800.528.4040
Tempe
I. E. Tel: 480.829.1800
Tucson
A. E. Tel: 520.742.0515

Arkansas
W. E. Tel: 972.235.9953

California
Agoura Hills
B. M. Tel: 818.865.0266
Granite Bay
B. M. Tel: 916.523.7047
Irvine
A. E. Tel: 949.789.4100
B. M. Tel: 949.470.2900
I. E. Tel: 949.727.3291
W. E. Tel: 800.626.9953
Los Angeles
A. E. Tel: 818.594.0404
W. E. Tel: 800.288.9953
Sacramento
A. E. Tel: 916.632.4500
W. E. Tel: 800.627.9953
San Diego
A. E. Tel: 858.385.7500
B. M. Tel: 858.597.3010
I. E. Tel: 800.677.6011
W. E. Tel: 800.829.9953
San Jose
A. E. Tel: 408.435.3500
B. M. Tel: 408.436.0881
I. E. Tel: 408.952.7000
Santa Clara
W. E. Tel: 800.866.9953
Woodland Hills
A. E. Tel: 818.594.0404
Westlake Village
I. E. Tel: 818.707.2101

Colorado
Denver
A. E. Tel: 303.790.1662
B. M. Tel: 303.846.3065
W. E. Tel: 800.933.9953
Englewood
I. E. Tel: 303.649.1800
Idaho Springs
B. M. Tel: 303.567.0703

Connecticut
Cheshire
A. E. Tel: 203.271.5700
I. E. Tel: 203.272.5843
Wallingford
W. E. Tel: 800.605.9953

Delaware
North/South
A. E. Tel: 800.526.4812
Tel: 800.638.5988
B. M. Tel: 302.328.8968
W. E. Tel: 856.439.9110

Florida
Altamonte Springs
B. M. Tel: 407.682.1199
I. E. Tel: 407.834.6310
Boca Raton
I. E. Tel: 561.997.2540
Bonita Springs
B. M. Tel: 941.498.6011
Clearwater
I. E. Tel: 727.524.8850
Fort Lauderdale
A. E. Tel: 954.484.5482
W. E. Tel: 800.568.9953
Miami
B. M. Tel: 305.477.6406
Orlando
A. E. Tel: 407.657.3300
W. E. Tel: 407.740.7450
Tampa
W. E. Tel: 800.395.9953
St. Petersburg
A. E. Tel: 727.507.5000

Georgia
Atlanta
A. E. Tel: 770.623.4400
B. M. Tel: 770.980.4922
W. E. Tel: 800.876.9953
Duluth
I. E. Tel: 678.584.0812

Hawaii
A. E. Tel: 800.851.2282

Idaho
A. E. Tel: 801.365.3800
W. E. Tel: 801.974.9953

Illinois
North/South
A. E. Tel: 847.797.7300
Tel: 314.291.5350
Chicago
B. M. Tel: 847.413.8530
W. E. Tel: 800.853.9953
Schaumburg
I. E. Tel: 847.885.9700

Indiana
Fort Wayne
I. E. Tel: 219.436.4250
W. E. Tel: 888.358.9953
Indianapolis
A. E. Tel: 317.575.3500

Iowa
W. E. Tel: 612.853.2280
Cedar Rapids
A. E. Tel: 319.393.0033

Kansas
W. E. Tel: 303.457.9953
Kansas City
A. E. Tel: 913.663.7900
Lenexa
I. E. Tel: 913.492.0408

Kentucky
W. E. Tel: 937.436.9953
Central/Northern/ Western
A. E. Tel: 800.984.9503
Tel: 800.767.0329
Tel: 800.829.0146

Louisiana
W. E. Tel: 713.854.9953
North/South
A. E. Tel: 800.231.0253
Tel: 800.231.5775

Maine
A. E. Tel: 800.272.9255
W. E. Tel: 781.271.9953

Maryland
Baltimore
A. E. Tel: 410.720.3400
W. E. Tel: 800.863.9953
Columbia
B. M. Tel: 800.673.7461
I. E. Tel: 410.381.3131

Massachusetts
Boston
A. E. Tel: 978.532.9808
W. E. Tel: 800.444.9953
Burlington
I. E. Tel: 781.270.9400
Marlborough
B. M. Tel: 800.673.7459
Woburn
B. M. Tel: 800.552.4305

Michigan
Brighton
I. E. Tel: 810.229.7710
Detroit
A. E. Tel: 734.416.5800
W. E. Tel: 888.318.9953
Clarkston
B. M. Tel: 877.922.9363

Minnesota
Champlin
B. M. Tel: 800.557.2566
Eden Prairie
B. M. Tel: 800.255.1469
Minneapolis
A. E. Tel: 612.346.3000
W. E. Tel: 800.860.9953
St. Louis Park
I. E. Tel: 612.525.9999

Mississippi
A. E. Tel: 800.633.2918
W. E. Tel: 256.830.1119

Missouri
W. E. Tel: 630.620.0969
St. Louis
A. E. Tel: 314.291.5350
I. E. Tel: 314.872.2182

Montana
A. E. Tel: 800.526.1741
W. E. Tel: 801.974.9953

Nebraska
A. E. Tel: 800.332.4375
W. E. Tel: 303.457.9953

Nevada
Las Vegas
A. E. Tel: 800.528.8471
W. E. Tel: 702.765.7117

New Hampshire
A. E. Tel: 800.272.9255
W. E. Tel: 781.271.9953

New Jersey
North/South
A. E. Tel: 201.515.1641
Tel: 609.222.6400
Mt. Laurel
I. E. Tel: 856.222.9566
Pine Brook
B. M. Tel: 973.244.9668
W. E. Tel: 800.862.9953
Parsippany
I. E. Tel: 973.299.4425
Wayne
W. E. Tel: 973.237.9010

New Mexico
W. E. Tel: 480.804.7000
Albuquerque
A. E. Tel: 505.293.5119

U.S. Distributors by State (Continued)

New York
Hauppauge
I. E. Tel: 516.761.0960
Long Island
A. E. Tel: 516.434.7400
W. E. Tel: 800.861.9953
Rochester
A. E. Tel: 716.475.9130
I. E. Tel: 716.242.7790
W. E. Tel: 800.319.9953
Smithtown
B. M. Tel: 800.543.2008
Syracuse
A. E. Tel: 315.449.4927

North Carolina
Raleigh
A. E. Tel: 919.859.9159
I. E. Tel: 919.873.9922
W. E. Tel: 800.560.9953

North Dakota
A. E. Tel: 800.829.0116
W. E. Tel: 612.853.2280

Ohio
Cleveland
A. E. Tel: 216.498.1100
W. E. Tel: 800.763.9953
Dayton
A. E. Tel: 614.888.3313
I. E. Tel: 937.253.7501
W. E. Tel: 800.575.9953
Strongsville
B. M. Tel: 440.238.0404
Valley View
I. E. Tel: 216.520.4333

Oklahoma
W. E. Tel: 972.235.9953
Tulsa
A. E. Tel: 918.459.6000
I. E. Tel: 918.665.4664

Oregon
Beaverton
B. M. Tel: 503.524.1075
I. E. Tel: 503.644.3300
Portland
A. E. Tel: 503.526.6200
W. E. Tel: 800.879.9953

Pennsylvania
Mercer
I. E. Tel: 412.662.2707
Philadelphia
A. E. Tel: 800.526.4812
B. M. Tel: 877.351.2355
W. E. Tel: 800.871.9953
Pittsburgh
A. E. Tel: 412.281.4150
W. E. Tel: 440.248.9996

Rhode Island
A. E. 800.272.9255
W. E. Tel: 781.271.9953

South Carolina
A. E. Tel: 919.872.0712
W. E. Tel: 919.469.1502

South Dakota
A. E. Tel: 800.829.0116
W. E. Tel: 612.853.2280

Tennessee
W. E. Tel: 256.830.1119
East/West
A. E. Tel: 800.241.8182
Tel: 800.633.2918

Texas
Arlington
B. M. Tel: 817.417.5993
Austin
A. E. Tel: 512.219.3700
B. M. Tel: 512.258.0725
I. E. Tel: 512.719.3090
W. E. Tel: 800.365.9953
Dallas
A. E. Tel: 214.553.4300
B. M. Tel: 972.783.4191
W. E. Tel: 800.955.9953
El Paso
A. E. Tel: 800.526.9238
Houston
A. E. Tel: 713.781.6100
B. M. Tel: 713.917.0663
W. E. Tel: 800.888.9953
Richardson
I. E. Tel: 972.783.0800
Rio Grande Valley
A. E. Tel: 210.412.2047
Stafford
I. E. Tel: 281.277.8200

Utah
Centerville
B. M. Tel: 801.295.3900
Murray
I. E. Tel: 801.288.9001
Salt Lake City
A. E. Tel: 801.365.3800
W. E. Tel: 800.477.9953

Vermont
A. E. Tel: 800.272.9255
W. E. Tel: 716.334.5970

Virginia
A. E. Tel: 800.638.5988
W. E. Tel: 301.604.8488
Haymarket
B. M. Tel: 703.754.3399
Springfield
B. M. Tel: 703.644.9045

Washington
Kirkland
I. E. Tel: 425.820.8100
Maple Valley
B. M. Tel: 206.223.0080
Seattle
A. E. Tel: 425.882.7000
W. E. Tel: 800.248.9953

West Virginia
A. E. Tel: 800.638.5988

Wisconsin
Milwaukee
A. E. Tel: 414.513.1500
W. E. Tel: 800.867.9953
Wauwatosa
I. E. Tel: 414.258.5338

Wyoming
A. E. Tel: 800.332.9326
W. E. Tel: 801.974.9953

Direct Sales Representatives by State (Components and Boards)

E. A. Earle Associates
E. L. Electrodyne - UT
GRP Group 2000
I. S. Infinity Sales, Inc.
ION ION Associates, Inc.
R. A. Rathsburg Associates, Inc.
SGY Synergy Associates, Inc.

Arizona
Tempe
E. A. Tel: 480.921.3305

California
Calabasas
I. S. Tel: 818.880.6480
Irvine
I. S. Tel: 714.833.0300
San Diego
E. A. Tel: 619.278.5441

Illinois
Elmhurst
R. A. Tel: 630.516.8400

Indiana
Cicero
R. A. Tel: 317.984.8608
Ligonier
R. A. Tel: 219.894.3184
Plainfield
R. A. Tel: 317.838.0360

Massachusetts
Burlington
SGY Tel: 781.238.0870

Michigan
Byron Center
R. A. Tel: 616.554.1460
Good Rich
R. A. Tel: 810.636.6060
Novi
R. A. Tel: 810.615.4000

North Carolina
Cary
GRP Tel: 919.481.1530

Ohio
Columbus
R. A. Tel: 614.457.2242
Dayton
R. A. Tel: 513.291.4001
Independence
R. A. Tel: 216.447.8825

Pennsylvania
Somerset
R. A. Tel: 814.445.6976

Texas
Austin
ION Tel: 512.794.9006
Arlington
ION Tel: 817.695.8000
Houston
ION Tel: 281.376.2000

Utah
Salt Lake City
E. L. Tel: 801.264.8050

Wisconsin
Muskego
R. A. Tel: 414.679.8250
Saukville
R. A. Tel: 414.268.1152

Sales Offices and Design Resource Centers

LSI Logic Corporation
Corporate Headquarters
1551 McCarthy Blvd
Milpitas CA 95035
Tel: 408.433.8000
Fax: 408.433.8989

NORTH AMERICA

California
Irvine
18301 Von Karman Ave
Suite 900
Irvine, CA 92612
♦ Tel: 949.809.4600
Fax: 949.809.4444

Pleasanton Design Center
5050 Hopyard Road, 3rd Floor
Suite 300
Pleasanton, CA 94588
Tel: 925.730.8800
Fax: 925.730.8700

San Diego
7585 Ronson Road
Suite 100
San Diego, CA 92111
Tel: 858.467.6981
Fax: 858.496.0548

Silicon Valley
1551 McCarthy Blvd
Sales Office
M/S C-500
Milpitas, CA 95035
♦ Tel: 408.433.8000
Fax: 408.954.3353
Design Center
M/S C-410
Tel: 408.433.8000
Fax: 408.433.7695

Wireless Design Center
11452 El Camino Real
Suite 210
San Diego, CA 92130
Tel: 858.350.5560
Fax: 858.350.0171

Colorado
Boulder
4940 Pearl East Circle
Suite 201
Boulder, CO 80301
♦ Tel: 303.447.3800
Fax: 303.541.0641

Colorado Springs
4420 Arrowswest Drive
Colorado Springs, CO 80907
Tel: 719.533.7000
Fax: 719.533.7020

Fort Collins
2001 Danfield Court
Fort Collins, CO 80525
Tel: 970.223.5100
Fax: 970.206.5549

Florida
Boca Raton
2255 Glades Road
Suite 324A
Boca Raton, FL 33431
Tel: 561.989.3236
Fax: 561.989.3237

Georgia
Alpharetta
2475 North Winds Parkway
Suite 200
Alpharetta, GA 30004
Tel: 770.753.6146
Fax: 770.753.6147

Illinois
Oakbrook Terrace
Two Mid American Plaza
Suite 800
Oakbrook Terrace, IL 60181
Tel: 630.954.2234
Fax: 630.954.2235

Kentucky
Bowling Green
1262 Chestnut Street
Bowling Green, KY 42101
Tel: 270.793.0010
Fax: 270.793.0040

Maryland
Bethesda
6903 Rockledge Drive
Suite 230
Bethesda, MD 20817
Tel: 301.897.5800
Fax: 301.897.8389

Massachusetts
Waltham
200 West Street
Waltham, MA 02451
♦ Tel: 781.890.0180
Fax: 781.890.6158

Burlington - Mint Technology
77 South Bedford Street
Burlington, MA 01803
Tel: 781.685.3800
Fax: 781.685.3801

Minnesota
Minneapolis
8300 Norman Center Drive
Suite 730
Minneapolis, MN 55437
♦ Tel: 612.921.8300
Fax: 612.921.8399

New Jersey
Red Bank
125 Half Mile Road
Suite 200
Red Bank, NJ 07701
Tel: 732.933.2656
Fax: 732.933.2643

Cherry Hill - Mint Technology
215 Longstone Drive
Cherry Hill, NJ 08003
Tel: 856.489.5530
Fax: 856.489.5531

New York
Fairport
550 Willowbrook Office Park
Fairport, NY 14450
Tel: 716.218.0020
Fax: 716.218.9010

North Carolina
Raleigh
Phase II
4601 Six Forks Road
Suite 528
Raleigh, NC 27609
Tel: 919.785.4520
Fax: 919.783.8909

Oregon
Beaverton
15455 NW Greenbrier Parkway
Suite 235
Beaverton, OR 97006
Tel: 503.645.0589
Fax: 503.645.6612

Texas
Austin
9020 Capital of TX Highway North
Building 1
Suite 150
Austin, TX 78759
Tel: 512.388.7294
Fax: 512.388.4171

Plano
500 North Central Expressway
Suite 440
Plano, TX 75074
♦ Tel: 972.244.5000
Fax: 972.244.5001

Houston
20405 State Highway 249
Suite 450
Houston, TX 77070
Tel: 281.379.7800
Fax: 281.379.7818

Canada
Ontario
Ottawa
260 Hearst Way
Suite 400
Kanata, ON K2L 3H1
♦ Tel: 613.592.1263
Fax: 613.592.3253

INTERNATIONAL

France
Paris
LSI Logic S.A.
Immeuble Europa
53 bis Avenue de l'Europe
B.P. 139
78148 Velizy-Villacoublay
Cedex, Paris
♦ Tel: 33.1.34.63.13.13
Fax: 33.1.34.63.13.19

Germany
Munich
LSI Logic GmbH
Orleansstrasse 4
81669 Munich
♦ Tel: 49.89.4.58.33.0
Fax: 49.89.4.58.33.108

Stuttgart
Mittlerer Pfad 4
D-70499 Stuttgart
♦ Tel: 49.711.13.96.90
Fax: 49.711.86.61.428

Italy
Milan
LSI Logic S.P.A.
Centro Direzionale Colleoni
Palazzo Orione Ingresso 1
20041 Agrate Brianza, Milano
♦ Tel: 39.039.687371
Fax: 39.039.6057867

Japan
Tokyo
LSI Logic K.K.
Rivage-Shinagawa Bldg. 14F
4-1-8 Kounan
Minato-ku, Tokyo 108-0075
♦ Tel: 81.3.5463.7821
Fax: 81.3.5463.7820

Osaka
Crystal Tower 14F
1-2-27 Shiromi
Chuo-ku, Osaka 540-6014
♦ Tel: 81.6.947.5281
Fax: 81.6.947.5287

Sales Offices and Design Resource Centers (Continued)

Korea
Seoul
LSI Logic Corporation of Korea Ltd
10th Fl., Haesung 1 Bldg.
942, Daechi-dong,
Kangnam-ku, Seoul, 135-283
Tel: 82.2.528.3400
Fax: 82.2.528.2250

The Netherlands
Eindhoven
LSI Logic Europe Ltd
World Trade Center Eindhoven
Building 'Rijder'
Bogert 26
5612 LZ Eindhoven
Tel: 31.40.265.3580
Fax: 31.40.296.2109

Singapore
Singapore
LSI Logic Pte Ltd
7 Temasek Boulevard
#28-02 Suntec Tower One
Singapore 038987
Tel: 65.334.9061
Fax: 65.334.4749

Sweden
Stockholm
LSI Logic AB
Finlandsgatan 14
164 74 Kista
♦ Tel: 46.8.444.15.00
Fax: 46.8.750.66.47

Taiwan
Taipei
LSI Logic Asia, Inc.
Taiwan Branch
10/F 156 Min Sheng E. Road
Section 3
Taipei, Taiwan R.O.C.
Tel: 886.2.2718.7828
Fax: 886.2.2718.8869

United Kingdom
Bracknell
LSI Logic Europe Ltd
Greenwood House
London Road
Bracknell, Berkshire RG12 2UB
♦ Tel: 44.1344.426544
Fax: 44.1344.481039

♦ Sales Offices with Design Resource Centers

International Distributors

Australia
New South Wales
Reptechnic Pty Ltd
3/36 Bydown Street
Neutral Bay, NSW 2089
♦ Tel: 612.9953.9844
Fax: 612.9953.9683

Belgium
Acal nv/sa
Lozenberg 4
1932 Zaventem
Tel: 32.2.7205983
Fax: 32.2.7251014

China
Beijing
LSI Logic International Services Inc.
Beijing Representative Office
Room 708
Canway Building
66 Nan Li Shi Lu
Xicheng District
Beijing 100045, China
Tel: 86.10.6804.2534 to 38
Fax: 86.10.6804.2521

France
Rungis Cedex
Azzurri Technology France
22 Rue Saarinen
Sillic 274
94578 Rungis Cedex
Tel: 33.1.41806310
Fax: 33.1.41730340

Germany
Haar
EBV Elektronik
Hans-Pinsel Str. 4
D-85540 Haar
Tel: 49.89.4600980
Fax: 49.89.46009840

Munich
Avnet Emg GmbH
Stahlgruberring 12
81829 Munich
Tel: 49.89.45110102
Fax: 49.89.42.27.75

Wuennenberg-Haaren
Peacock AG
Graf-Zepplin-Str 14
D-33181 Wuennenberg-Haaren
Tel: 49.2957.79.1692
Fax: 49.2957.79.9341

Hong Kong
Hong Kong
AVT Industrial Ltd
Unit 608 Tower 1
Cheung Sha Wan Plaza
833 Cheung Sha Wan Road
Kowloon, Hong Kong
Tel: 852.2428.0008
Fax: 852.2401.2105

Serial System (HK) Ltd
2301 Nanyang Plaza
57 Hung To Road, Kwun Tong
Kowloon, Hong Kong
Tel: 852.2995.7538
Fax: 852.2950.0386

India
Bangalore
Spike Technologies India Private Ltd
951, Vijayalakshmi Complex,
2nd Floor, 24th Main,
J P Nagar II Phase,
Bangalore, India 560078
♦ Tel: 91.80.664.5530
Fax: 91.80.664.9748

Israel
Tel Aviv
Eastronics Ltd
11 Rozanis Street
P.O. Box 39300
Tel Aviv 61392
Tel: 972.3.6458777
Fax: 972.3.6458666

Japan
Tokyo
Daito Electron
Sogo Kojimachi No.3 Bldg
1-6 Kojimachi
Chiyoda-ku, Tokyo 102-8730
Tel: 81.3.3264.0326
Fax: 81.3.3261.3984

Global Electronics Corporation
Nichibei Time24 Bldg. 35 Tansu-cho
Shinjuku-ku, Tokyo 162-0833
Tel: 81.3.3260.1411
Fax: 81.3.3260.7100
Technical Center
Tel: 81.471.43.8200

Marubeni Solutions
1-26-20 Higashi
Shibuya-ku, Tokyo 150-0001
Tel: 81.3.5778.8662
Fax: 81.3.5778.8669

Shinki Electronics
Myuru Daikanyama 3F
3-7-3 Ebisu Minami
Shibuya-ku, Tokyo 150-0022
Tel: 81.3.3760.3110
Fax: 81.3.3760.3101

Yokohama-City
Innotech
2-15-10 Shin Yokohama
Kohoku-ku
Yokohama-City, 222-8580
Tel: 81.45.474.9037
Fax: 81.45.474.9065

Macnica Corporation
Hakusan High-Tech Park
1-22-2 Hadusan, Midori-Ku,
Yokohama-City, 226-8505
Tel: 81.45.939.6140
Fax: 81.45.939.6141

The Netherlands
Eindhoven
Acal Nederland b.v.
Beatrix de Rijkweg 8
5657 EG Eindhoven
Tel: 31.40.2.502602
Fax: 31.40.2.510255

Switzerland
Brugg
LSI Logic Sulzer AG
Mattenstrasse 6a
CH 2555 Brugg
Tel: 41.32.3743232
Fax: 41.32.3743233

Taiwan
Taipei
Avnet-Mercuries Corporation, Ltd
14F, No. 145,
Sec. 2, Chien Kuo N. Road
Taipei, Taiwan, R.O.C.
Tel: 886.2.2516.7303
Fax: 886.2.2505.7391

Lumax International Corporation, Ltd
7th Fl., 52, Sec. 3
Nan-Kang Road
Taipei, Taiwan, R.O.C.
Tel: 886.2.2788.3656
Fax: 886.2.2788.3568

Prospect Technology Corporation, Ltd
4Fl., No. 34, Chu Luen Street
Taipei, Taiwan, R.O.C.
Tel: 886.2.2721.9533
Fax: 886.2.2773.3756

Wintech Microeletronics Co., Ltd
7F., No. 34, Sec. 3, Pateh Road
Taipei, Taiwan, R.O.C.
Tel: 886.2.2579.5858
Fax: 886.2.2570.3123

United Kingdom
Maidenhead
Azzurri Technology Ltd
16 Grove Park Business Estate
Waltham Road
White Waltham
Maidenhead, Berkshire SL6 3LW
Tel: 44.1628.826826
Fax: 44.1628.829730

Milton Keynes
Ingram Micro (UK) Ltd
Garamonde Drive
Wymbush
Milton Keynes
Buckinghamshire MK8 8DF
Tel: 44.1908.260422

Swindon
EBV Elektronik
12 Interface Business Park
Bincknoll Lane
Wootton Bassett,
Swindon, Wiltshire SN4 8SY
Tel: 44.1793.849933
Fax: 44.1793.859555

♦ Sales Offices with Design Resource Centers