Single Board Computers
SCSI Software User's Manual

(SBCSCSI/D1)

Notice

While reasonable efforts have been made to assure the accuracy of this document, Motorola, Inc. assumes no liability resulting from any omissions in this document, or from the use of the information obtained therein. Motorola reserves the right to revise this document and to make changes from time to time in the content hereof without obligation of Motorola to notify any person of such revision or changes.

It is possible that this publication may contain reference to, or information about Motorola products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that Motorola intends to announce such Motorola products, programming, or services in your country.

Restricted Rights Legend

If the documentation contained herein is supplied, directly or indirectly, to the U.S. Government, the following notice shall apply unless otherwise agreed to in writing by Motorola, Inc.

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Motorola, Inc.
Computer Group
2900 South Diablo Way
Tempe, Arizona 85282

Preface

The Single Board Computers SCSI Software User's Manual describes the SCSI Software, a building block for SCSI services. It is intended for developers who have a working knowledge of SCSI. In the context of this manual, SCSI Software describes the Firmware used to control the NCR53C710 SCSI I/O Processor (SIOP) used on the Motorola Single Board Computers containing that chip. It does not include operating system specific device drivers.

Throughout this manual, the term Single Board Computer (SBC) refers to any of the MVME162/162LX/166/167/187/197 series of CPU boards.

A working knowledge of the SCSI-2 Specification is assumed. To use this manual, you should be familiar with the publications listed in the *Related Documentation* section in Chapter 1.

The SCSI Software described in this manual is written to be independent of any particular operating system. The SCSI Software is used by Motorola-supplied SYSTEM V/68, SYSTEM V/88, VMEexec, and the onboard ROM or FLASH Debuggers (MVME162Bug, etc.). It can be adapted to work with nearly all software running on these boards. Only the interface routines described in Appendix C must be provided external to the SCSI Software.

Note **This manual replaces the *MVME167/MVME187 Single Board Computers SCSI Software User's Manual, MVME187FW/D1*, which is obsolete.**

Printed in the United States of America

November 1993

Contents

List of Figures

List of Tables

INTRODUCTION 1

General Information

This chapter explains what SCSI is and what the SBC SCSI Software does to support SCSI-related hardware on selected Single Board Computers (SBCs). It also explains the meanings conveyed by the variety of fonts and special text symbols found throughout this manual. Most importantly, it assists in finding desirable information on these pages and elsewhere.

Notes

This user's manual documents the SBC SCSI Software Release 1.1. It does not necessarily apply to the release superceded, which was known as the MVME167/187 SCSI Software R10V1.

Throughout this manual, the term Single Board Computer (SBC) refers to any of the MVME162/162LX/166/167/187/197 series of CPU boards.

Organization of This Manual

Here are some short descriptions of the remaining chapters in this manual:

- Chapter 2 provides an overall perspective of the core of the SBC SCSI Software, which is referred to as the SCSI I/O Processor (SIOP) Firmware (or just Firmware).
- Chapter 3 details the programming interface used to invoke the Firmware.
- Chapter 4 explains the special tools (NCR Build Tools) used to build the Firmware from source and how it must be prepared for its run-time environment.
- Chapter 5 covers additional special topics such as the debug logging facility and certain run-time considerations.
- Appendix A maps out the directory structure of the SBC SCSI Software release media.
- Appendix B specifies the data structures used to communicate with the SIOP Firmware via its programming interface.
- Appendix C describes the external routines which must be provided, with C-language interfaces, in order to compile and use the Firmware.
- Appendix D catalogs the Firmware's run-time returned error conditions.
- Appendix E documents a demo (test) program which illustrates the use of the SBC SCSI Software and can be used to verify its functionality.
- The Glossary explains terminology often used when discussing the SBC SCSI Software.

Conventions

The conventions used in this document are:

bold	for user input that you type just as it appears; also used for commands, options and arguments to commands, and names of programs, directories, and files.
italic	for names of *variables* to which you assign values; also used for comments in screen displays and examples.
`fixed font`	for system output (e.g., screen displays, reports), examples, and system prompts.
`|`	to separate two or more items and indicate that a choice is to be made; only one of the items separated by this symbol should be selected.
[ ]	to enclose an item that is optional.
{ }	to enclose an optional symbol.
…	to repeat the previous argument.
0x or	
$	specifies a hexadecimal character. Unless otherwise specified, all address references are in hexadecimal throughout this manual.
<CR>	the single key you press that performs the return function.

Related Documentation

The publications are applicable to the SBCs and may provide additional helpful information pertinent to the use of the SCSI Software. If not shipped with this product, they may be purchased by contacting your local Motorola sales office. Non-Motorola documents may be obtained from the sources listed.

Document Title	Motorola Part Number
SBC SCSI Software Release 1.1 Software Release Guide	SBCSCSI/S1
MVME162 Embedded Controller Programmer's Reference Guide	MVME162PG
MVME162LX Embedded Controller Programmer's Reference Guide	MVME162LXPG
MVME166/MVME167/MVME187 Single Board Computers Programmer's Reference Guide	MVME187PG
MVME197LE, MVME197DP, and MVME197SP Single Board Computers Programmer's Reference Guide	MVME197PG
MVME162Bug Debugging Package User's Manual	MVME162BUG
MVME167Bug Debugging Package User's Manual	MVME167BUG
Debugging Package for Motorola 68K CISC CPUs User's Manual	68KBUG
MVME187Bug Debugging Package User's Manual	MVME187BUG
Debugging Package for Motorola 88K RISC CPUs User's Manual	88KBUG
MVME197Bug Debugging Package User's Manual	MVME197BUG
MVME162 Embedded Controller User's Manual	MVME162
MVME162LX Embedded Controller User's Manual	MVME162LX
MVME166 Single Board Computer User's Manual	MVME166
MVME167 Single Board Computer User's Manual	MVME167
MVME187 RISC Single Board Computer User's Manual	MVME187
MVME197LE Single Board Computer User's Manual	MVME197LE

Note **Although not shown in the above list, each Motorola Computer Group manual publication number is suffixed with characters which represent the revision level of the document, such as "/D2" (the second revision of a manual); a supplement bears the same number as a manual but has a suffix such as "/D2A1" (the first supplement to the manual).**

The following publications are available from the sources indicated.

ANSI Small Computer System Interface-2 (SCSI-2), Draft Document X3.131-198X, Revision 10c; Global Engineering Documents, P.O. Box 19539, Irvine, CA 92714.

NCR 53C710 SCSI I/O Processor Data Manual, document #NCR53C710DM; NCR Corporation, Microelectronics Products Division, Colorado Springs, CO.

NCR 53C710 SCSI I/O Processor Programmer's Guide, document #NCR53C710PG; NCR Corporation, Microelectronics Products Division, Colorado Springs, CO.

Definition of SCSI

According to the SCSI-2 specification,

"SCSI is a local I/O bus that can be operated over a wide range of data rates. The primary objective of the interface is to provide host computers with device independence within a class of devices. Thus, different disk drives, tape drives, printers, optical media drives, and other devices can be added to the host computers without requiring modifications to generic system hardware or software. Provision is made for the addition of special features and functions through the use of vendor unique fields and codes. Reserved fields and codes are provided for future standardization."

General Description of the SCSI Software

The SCSI Software, or Firmware, for the Single Board Computers (SBCs) provides comprehensive access to the SCSI bus. It consists of Motorola Processor (MPU) code and NCR SCSI SCRIPTS code (SCRIPTS) used to control the NCR 53C710 SCSI I/O Processor (SIOP). The SIOP provides the actual physical connection to the SCSI bus.

The SCRIPTS are instructions executed by the SIOP which control the specific operations and functionality of the SIOP. The MPU code formats command control information in a manner compatible to the SCRIPTS. It also handles situations where the SIOP or SCRIPTS cannot perform the required function.

From an application viewpoint, the SCSI Software can be interconnected with the application in order to provide access to the SCSI bus. SIOP programming is not required in order to create the interconnection. The SCSI Software allows access to the SCSI bus in conformance with SCSI bus protocols and can be linked into the final application to create the connection to the SCSI bus. Typically, an application specific device driver is used to interface application code with the SCSI Software.

SIOP Firmware

The SIOP Firmware provides these unique features and facilities:

- ❑ Handles all aspects of SCSI protocol conformance.
- ❑ No firmware intelligence is imposed on the command requests issued to peripherals.
- ❑ Internally enqueues all command requests, in a linked list, until they can be dispatched to the SCSI bus; therefore, an unlimited number of command requests can be issued to the Firmware before status for any of them is received by the user.
- ❑ Executes in either polled mode or interrupt mode. Polled mode operates without interrupts and returns from a command after the command completes.
- ❑ Provides Target level access (*AVAILABLE ONLY IN A FUTURE RELEASE*).
- ❑ The SIOP executes independently of the Motorola Processor thus reducing the MPU overhead associated with SCSI accesses.

OVERVIEW 2

Introduction

The following chapter is an introduction to the workings of the SIOP Firmware which is used to control the SCSI port on the SBCs. It is recommended that the reader have a working knowledge of SCSI.

A Basic View of the SIOP Firmware

The Firmware consists of the Motorola Microprocessor (MPU) code and the NCR SCSI SCRIPTS code (SCRIPTS) which work together to control the NCR 53C710 SCSI I/O Processor (SIOP).

The MPU code provides the user interface to the SCSI bus. In addition, it translates user command requests into a format executable by the SCRIPTS, manages SIOP operating parameters, handles SIOP interrupt conditions, manages internal command request queues, and returns status to the user. The MPU code initiates user command requests to the SCSI bus by invoking the SCRIPTS.

All of the MPU code is written in "C". The interfaces to the Firmware shown in Chapter 3 are based on the "C" syntax. In addition, "C" syntax is used throughout this manual for data structures, examples, etc. For the convenience of assembly language users, Tables 3-2 and 3-3 show the assembly language interface.

The SCRIPTS, which are instructions executed by the SIOP, control the functional flow of the SIOP (i.e., transition of SCSI data and control lines). The SCRIPTS manage the physical thread of each SCSI command request by using the processed information provided by the MPU code to drive the SIOP.

The SIOP handles the hardware interface to the SCSI bus. The SIOP operates in a manner which conforms to the physical requirements of the SCSI specification.

2

Firmware Interface

The SIOP Firmware has several externally accessible routines, or entry points, which may be called by the user to initiate Firmware action. The following is a brief summary of these routines.

siop_init() This routine is for Firmware initialization. The user calls this entry point to provide the Firmware with memory resources and basic SIOP operating parameters and to allow the Firmware to perform required initialization.

siop_cmd() This routine is for Firmware command requests. The user calls this entry point to send command structures (*siop_struc*) to the Firmware. These commands can be used to either configure the Firmware or initiate SCSI bus activity. This is the primary entry point for SCSI bus accesses.

siop_int() This routine is for SIOP interrupt handling. The user calls this entry point to allow the Firmware to process interrupts generated by the SIOP. This entry point must be called at or above the interrupt level of the SIOP to protect critical code regions.

sdt_tinit() This routine is for Firmware debug logging initialization. The user calls this entry point to enable the debug logger and provide it with memory resources.

sdt_alloc() This routine is for debug logging. If debug logging has been enabled, this entry point is called to get the next block of memory to be used for debug logging.

sfw_getrev() The sfw_getrev() entry point provides a release ID string that identifies the Firmware. The calling application provides a pointer to MPU-writable memory and the number of bytes available there.

Division of Functional Responsibilities

The Firmware and the user have definite and separate responsibilities when communicating with SCSI devices. Basically, the Firmware manages the SCSI bus protocol while the user manages the device specifics. (See Figure 2-1.) The following sections provide details of these responsibilities.

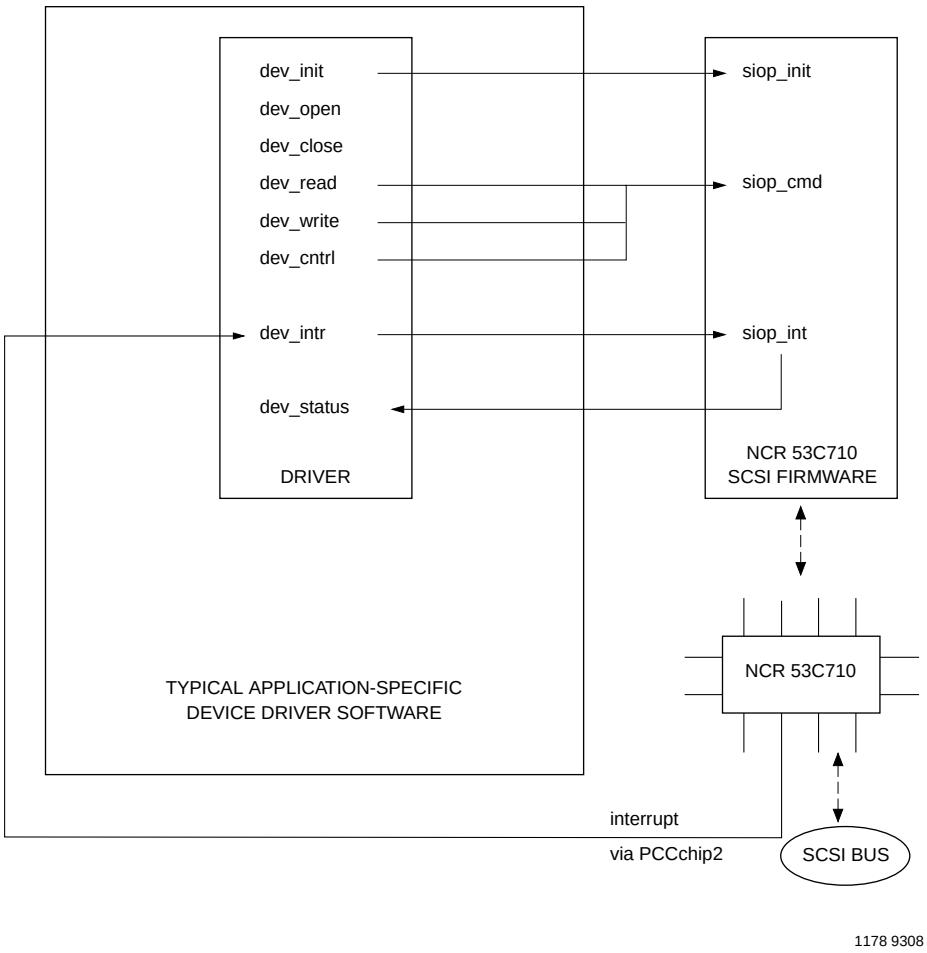

Figure 2-1. Firmware/User Interaction Block Diagram

Primary Functions of the Firmware

The Firmware is designed to manage the SCSI bus protocol when interfacing to a device on the SCSI bus. The Firmware operates independently of any specific knowledge of devices on the SCSI bus.

The Firmware handles the following aspects of SCSI protocol.

- **Interpret Received Messages**

  The Firmware automatically receives and processes the messages received from target devices.

- **Send Required/Response Messages**

  The Firmware generates and sends any required messages, such as **identify**, during the course of command processing. The Firmware also generates and sends messages in response to received messages (e.g., **synchronous data transfer request** message exchange).

- **Phase Transition Handling**

  The Firmware automatically handles all phase transitions that occur during the normal course of interfacing to a SCSI device.

  For example, the Firmware transitions through the following bus phases to execute a single SCSI **read** command to a disk drive: ARBITRATION, SELECTION (WITH ATTENTION), MESSAGE (**identify**), COMMAND, DATA-IN, STATUS, MESSAGE (**command complete**), and BUS FREE. In addition to this minimal set of phase transitions, the device may disconnect and reconnect several times during the DATA-IN phase. The Firmware handles each of these phase changes without direct intervention from the user.

- **Multi-Threaded Command Management**

  The Firmware manages multiple threading of commands to the SCSI bus. SCSI protocol allows for all devices attached to the bus to have simultaneously outstanding commands. The logical and physical thread management used to facilitate this multiple threading is handled by the Firmware and is transparent to the user.

- **Device Queue Management**

  The Firmware provides a command queue management scheme that allows the user to send an unlimited number of command requests to a device before receiving status for any of them.

- **Error Recovery**

  The Firmware supports only minimal error recovery. When an error condition is detected, the Firmware attempts to get the physically threaded device off the SCSI bus so the bus is free to send command requests to other SCSI devices or subsequent commands to the same device.

Primary Functions Required of the User

The following duties are required of a user when interfacing to the Firmware.

- **Device Configuration Management**

  The Firmware does not contain device specific knowledge such as device configuration parameters (i.e., block size). It is the responsibility of the user to maintain and manage this device specific knowledge.

- **Resource Allocation**

  The Firmware does not contain any static data areas; therefore, the user must provide all of the memory resources for the Firmware. The Firmware maintains independence from the specific operating system environment interfacing to it, with respect to memory mapping and cache management, by requiring the user to allocate memory resources.

- **Status Interpretation**

  The user is responsible for interpretation of all status returned by the Firmware. The Firmware is finished with a command and has released all resources and knowledge concerning the particular command request when final status has been posted for a specific command structure (*siop_struc*). At this point it is left to the user to determine what subsequent action is required to handle the returned status. Refer to Appendix D in this document for a detailed discussion of returned status codes.

- **User Supplied Routines**

  The user is responsible for supplying routines which are invoked by the Firmware. Two of these routines, **splhi()** and **splx()**, need to be globally defined so the Firmware can raise and the lower the interrupt mask when it is necessary to protect a section of code. A *return status routine*, which is specific to the user, is called by the Firmware to notify the user of the completion of command requests. A pointer to this return status routine **must** be installed in every command structure (*siop_struc*) processed by the Firmware. Refer to Appendix C.

2

Functional Overview

The following sections describe the functional attributes of the Firmware. A simplistic description of the flow of a command request through the Firmware is given. The SIOP interrupts are explained in some detail as are the Firmware responses to various SCSI messages which may be received.

Command Flow

The basic flow of the Firmware is followed as a command request is executed. This flow should give the reader an understanding of the interdependence between the MPU code, SCRIPTS code, and SIOP in executing a user request.

The first user access to the Firmware is through a call to **siop_init()**. The parameters passed to this initialization call set the operation mode of the SIOP and Firmware. Interrupt level, snoop mode, clock speed, and SCSI address are all parameters which are used to program the hardware. Different software paths are taken if the interrupt level is 0 (polled mode) as opposed to non-zero (interrupt mode). The Firmware needs to be initialized only once, but may be initialized many times as long as there are no outstanding command requests to the Firmware when **siop_init()** is called.

After the Firmware has been initialized, the user initiates a command to an SCSI device by passing a command structure (*siop_struc*) to the command entry point, **siop_cmd()**. Some of the fields in the *siop_struc* which the user must initialize for all command requests are command control (*cmd_ctrl*), device SCSI address and LUN (*addr_ilvl* and *lun*), command descriptor block (*cdb*), and return status routine pointer (*status_ptr*). Once called, the Firmware MPU code performs minimal management operations to enqueue the command structure (*siop_struc*) for execution by the SCRIPTS. Additionally, the user may not alter any field within the *siop_struc* until the control of the structure is returned to the user through the invocation of the return status routine.

From this point, the functional flow of the Firmware differs depending on whether the interrupt level is set for polled mode or interrupt mode. The following sections outline the flow for the separate modes.

Interrupt Mode

After the command structure (*siop_struc*) is enqueued for execution, the Firmware returns control of the MPU back to the user. All subsequent Firmware MPU code processing of the user's command request is initiated through the Firmware interrupt handler entry point, **siop_int()**. The user calls this entry point when an interrupt from the SIOP is detected.

The first interrupt is generated by the SCRIPTS to notify the MPU code that the SIOP is not busy with any SCSI bus activity. The MPU code dequeues the next command structure (*siop_struc*) which is available for execution and initializes some SCRIPTS control structures. Next, the MPU code invokes the appropriate SCRIPTS entry point for the command request.

Control of the MPU is returned to the user after the SCRIPTS begin execution. All SCSI bus activity is handled without MPU interruption except extended messages (i.e., synchronous data transfer negotiations), disconnects (to save the state of the physically threaded command request and, possibly, initiate another command request to the SCSI bus), reselects (to restore the state of the physically threaded command request), and error conditions. All of these situations are detected by the SCRIPTS/SIOP and a corresponding interrupt generated so the MPU code can resolve the issue.

The command request is finished when the device sends a **command complete** message and then transitions to the BUS FREE phase. When this sequence of events occurs, the SCRIPTS generate another interrupt to notify the MPU code that the command is done. The MPU code updates some fields in the command structure (*siop_struc*), updates Firmware queues, and then calls the user's return status routine. At this point, the *siop_struc* is back in the control of the user. The user may immediately call the Firmware through **siop_cmd()** to send a new command request; however, it is recommended that the status in the *status.allstat* field of the returned *siop_struc* be checked first to determine if any immediate recovery actions are needed (i.e., send a SCSI **request sense** command to the device) which might preempt the anticipated command request.

Upon return from the user's return status routine, the MPU code enqueues the next available command request for the device which just completed a request. Finally, the Firmware initiates another command request to the SCSI bus if a command structure (*siop_struc*) is available for execution.

Polled Mode

After the command structure (*siop_struc*) is enqueued, the Firmware calls **siop_int()** where the MPU code waits for the command request to complete. The MPU code resolves intermediate interrupt conditions for the command request until the command complete interrupt is generated. These intermediate interrupt conditions are the same as in interrupt mode and consist of, but are not limited to, extended messages (i.e., synchronous data transfer negotiations), disconnects (to save the state of the physically threaded command request), reselects (to restore the state of the physically threaded command request), and error conditions.

After the command request completes, the MPU code updates some fields in the command structure (*siop_struc*), updates Firmware queues, and then calls the user's return status routine. At this point, the *siop_struc* is back in the control of the user. The user may immediately call the Firmware through **siop_cmd()** to send a new command request; however, it is recommended that the user wait until the Firmware returns control of the MPU via the **siop_cmd()** return before sending any more command requests. If the user calls **siop_cmd()** each time the return status routine is invoked then the stack eventually overflows. This happens because the Firmware will not have returned (unstacked) from any of the calls to **siop_cmd()**.

Upon return from the user's return status routine, control of the MPU is returned to the user. The user should now check the status in the *status.allstat* field of the command structure (*siop_struc*) to determine if any recovery actions are needed.

Interrupt Mechanism

Interrupts from the SIOP are generated in response to various hardware conditions or are programmed interrupts generated by the SCRIPTS INT instruction. The SIOP halts SCRIPTS execution whenever an interrupt occurs. The following is the list of SIOP interrupts.

SCSI Bus Reset

This interrupt is generated when the SIOP detects a SCSI bus reset. The Firmware terminates all outstanding commands and returns status for each.

Phase Mismatch

This interrupt is generated when a target changes SCSI phases before the SIOP data counter register has exhausted its count. This interrupt occurs when an intermediate disconnect is pending or a data underrun has occurred.

Selection Timeout

This interrupt is generated when a device at a selected SCSI address fails to respond within 250 milliseconds after the start of the SELECTION phase.

Unexpected Disconnect

This interrupt is generated when a target device unexpectedly transitions to the BUS FREE phase. The Firmware determines if the disconnect resulted from an intentional action initiated by the Firmware (i.e., **device reset** message).

SCSI Gross Error

This interrupt is generated when the SIOP detects an illegal condition in the SIOP bus control logic (e.g., an overflow of the SCSI FIFO) or and illegal condition on the SCSI bus (e.g., a phase change with an outstanding synchronous offset). The Firmware hangs if this interrupt occurs.

INT Instruction

This is the SCRIPTS programmed interrupt invoked by the INT instruction. Programmed interrupts cause the MPU code to handle situations which the SCRIPTS or SIOP cannot.

Illegal Instruction

This interrupt is generated when the SIOP attempts to execute an illegal SCRIPTS instruction. Several different situations can produce an illegal SCRIPTS instruction. The Firmware determines the specific reason for the illegal instruction. The Firmware gets the current target off the bus before terminating the command and returning status to the user. These are some of the reasons for an illegal instruction.

1. The NCR SCRIPTS compiler generated the wrong opcode for a SCRIPTS instruction forcing the SIOP to execute an illegal opcode.
2. The memory where the SCRIPTS reside has been corrupted. This results in the SIOP executing an illegal opcode.
3. The SIOP attempts to execute a SCRIPTS instruction which is non-longword (four-byte) aligned. All SCRIPTS must be aligned to byte boundaries that are integer multiples of 4.
4. The SIOP attempts to transfer information but has been supplied with a transfer count of zero. This could occur if the user built an *siop_struc* to execute a SCSI **read** but initialized the *data_count* field to zero.

Bus Fault

This interrupt is generated whenever the SIOP receives a bus error in response to a local bus access. Action by the Firmware is dependent upon the current phase of the SCSI bus.

Single Step — This interrupt is generated only in a special diagnostic mode enabled by recompiling the Firmware source code. If enabled, the SIOP generates an interrupt after the successful execution of each SCRIPTS instruction.

Message Handling

This section deals with the messages that may be received by the Firmware and the associated Firmware response.

CMD Complete

A SCRIPTS instruction clears the SCSI ACK signal to complete the MSG-IN phase and then waits for the BUS FREE phase. A command complete INT instruction interrupt is then generated.

Save Data Pointers

A SCRIPTS instruction clears the SCSI ACK signal to complete the MSG-IN phase and then waits for an expected **disconnect** message. If a MSG-IN phase does not follow current message phase then a protocol violation INT instruction interrupt is generated.

Restore Data Pointers

A SCRIPTS instruction clears the SCSI ACK signal to complete the MSG-IN phase and then the SCRIPTS transition to the next phase dictated by the target.

Disconnect — A SCRIPTS instruction clears the SCSI ACK signal to complete the MSG-IN phase and then waits for the BUS FREE phase. A disconnect INT instruction interrupt is then generated.

Message Reject

A message reject INT instruction interrupt is generated. This holds the bus in the MSG-IN phase while the Firmware determines which message was rejected by the target. When the MPU code resolves the rejected message issue, it restarts the SCRIPTS at an instruction which clears the SCSI ACK signal to complete the MSG-IN phase.

Linked CMD Complete

A linked command complete INT instruction interrupt is generated. This holds the bus in the MSG-IN phase while the Firmware sets up data structures for the next linked command. The MPU code restarts the SCRIPTS at an instruction which clears the SCSI ACK signal to complete the MSG-IN phase and then waits for the COMMAND phase.

Extended Message

A SCRIPTS instruction clears the SCSI ACK signal to accept the message byte and then the rest of the message bytes are received. ACK is not negated for the last message byte. Instead, an extended message INT instruction interrupt is generated. The MPU code copies the entire message into the MSG-IN buffer and then restarts the SCRIPTS at an instruction which clears the SCSI ACK signal to complete the MSG-IN phase.

If the extended message was a **synchronous data transfer request**, then the Firmware determines the next course of action. If the received message was a response to a **synchronous data transfer request** message, the Firmware logs the acceptable rate and offset. If the message was unsolicited, then the Firmware builds a response message with a rate and offset which is mutually acceptable, and then restarts the SCRIPTS at an instruction which asserts ATN before it clears ACK. The assertion of ATN tells the target that the SIOP has a message (out) ready to transmit in response to the last message (in).

FIRMWARE INTERFACE 3

Introduction

This chapter defines the interface to the SIOP Firmware. The entry points, input parameters, and return parameters are also described in this chapter.

For ease of documentation, the following typedefs are used within this section:

```
typedef unsigned char UCHAR;   /* 8 bit unsigned integer */ typedef
unsigned short USHORT; /* 16 bit unsigned integer */ typedef unsigned
int UINT;      /* 32 bit unsigned integer */
```

Tables 3-1, 3-2, and 3-3 contain a summary of the entry points and parameters for the SIOP Firmware.

Table 3-1. C Call Interface

Name	Input Parameters		Output Parameters
siop_init	init_vals	siopdatap	status
siop_cmd	siop_struc	siopdatap	void (status via status return routine)
siop_int	siopdatap		void
sdt_tinit	sdt_tvalue	siopdatap	void
sdt_alloc	siopdatap		&trace_entry
sfw_getrev	string_buffer	max_size	bytes available

Table 3-2. 68K Assembler Interface

Name	Input Registers		Output Registers
siop_init	init_vals 4 (A7)	siopdatap 8 (A7)	status D0
siop_cmd	siop_struc (A7)	siopdatap 8 (A7)	status via status return routine
siop_int	siopdatap 4 (A7)		
sdt_tinit	sdt_tvalue 4 (A7)	siopdatap 8 (A7)	
sdt_alloc	siopdatap 4 (A7)		&trace_entry A0
sfw_getrev	string_buffer 4 (A7)	max_size 8(A7)	bytes available D0

Table 3-3. 88K Assembler Interface

Name	Input Registers		Output Registers
siop_init	init_vals r2	siopdatap r3	status r2
siop_cmd	siop_struc r2	siopdatap r3	status via status return routine
siop_int	siopdatap r2		
sdt_tinit	sdt_tvalue r2	siopdatap r3	
sdt_alloc	siopdatap r2		&trace_entry r2
sfw_getrev	string_buffer r2	max_size r3	bytes available r2

siop_init()

NAME

siop_init —Initialize the SIOP Firmware

SYNOPSIS

```
#include "scsi.h"          /* SCSI specific definitions */
#include "ncr.h"           /* Firmware structure definitions */
#include "ncr710.h"        /* hardware/firmware specific definitions */
#include "scsi_err.h"      /* error definitions */

UINT siop_init(initvals, siopdatap)
INIT_TYPE_1 *initvals;     /* pointer to a structure which contains init
                              values (refer to Appendix B) */
struct siopdata *siopdatap;/* SIOP Firmware global data pointer */
```

DESCRIPTION

The initialization routine initializes the SIOP chip, Firmware structures, and Firmware flags. It must be called before the siop_cmd and siop_int entry points. Input to this routine is a pointer to an initialization structure and a pointer to a global data work area.

The `INIT_TYPE_1` initialization structure is defined in the appendix on data structures (Appendix B). The `siopdata` area is allocated by your Application, but its definition is private to the Firmware (see below).

This routine returns status to the caller after initialization is complete.

RETURN VALUE

Status of the Firmware is returned in the least significant byte (LSB) of the returned value.

ERROR CONDITIONS

A successful call to **siop_init()** must be executed before any other access to the Firmware can be attempted. All non-zero return codes are fatal and require the problem to be remedied. The return codes are listed along with an explanation of their meaning and recommended remedy.

SI_GOOD (0x00)
No errors.

SI_BADCLKPAR (0x0A)
A clock speed parameter was supplied that cannot be interpreted.
If `siop_init()` is passed an `INITPARS/INIT_TYPE_0` structure, this code may also indicate that the MPU clock (`clk_speed`) parameter contains ASCII values outside the range '0' to '9'.

SI_BADPARAM (0x21)
Bad parameter supplied via entry point. Returned when `siop_init()` does not recognize the signature of the `initvals` structure. Verify that it is a valid structure for the firmware revision in use.

SI_BADPATCH (0x1E)
The Firmware failed while patching the run-time SCRIPTS code.

SI_CLK2FAST (0x09)
For `INIT_TYPE_1`, the `sclk_speed` initialization parameter is faster than 75MHz, the highest speed currently supported. For `INIT_TYPE_0/INITVALS`, the MPU `clk_speed` is above 38MHz.

SI_NOSCSIBUS (0x1F)
The SCSI bus was found to be in an illegal state. This may result from the SIOP being connected to a SCSI bus which is not correctly powered or terminated.

Refer also to **incl/scsi_err.h** for defined error values.

NOTES

- The Firmware needs to be initialized only once but may be initialized many times as long as there are no outstanding command requests to the Firmware when **siop_init()** is called. If a user calls the initialization routine while commands are outstanding to the Firmware, then unpredictable results will occur. However, a user may call this module any number of times before calling the command entry module to reinitialize the firmware.
- The siop data area (*siopdata*), pointed to by *siopdatap*, is for use by the Firmware only and cannot be modified at any time by the caller.The value of `siopdatap` passed to **siop_init()** establishes the value of `siopdatap` passed on all subsequent calls to the SIOP other than **siop_init()**. Only subsequent calls to **siop_init()** where a new value of `siopdatap` is passed can change the current value. Successive calls to **siop_init()** with the same `siopdatap` has no effect on the current value of `siopdatap`. The user is responsible for clearing the *siopdata* area before calling **siop_init()** and must ensure that the *siopdata* area is cache coherent.
- If initialization is called after a call to the command entry point (**siop_cmd()**), then all synchronous data transfer information and tagged command queuing information will be lost. Also refer to the section in Chapter 5 on *Use of the Firmware After Use by the SBC ROM Debugger* for more information.
- The user may change the SIOP interrupt level and SIOP SCSI address after initialization through a *CONFIG* command to **siop_cmd()**.
- The siop data area, pointed to by `siopdatap`, is for use by the firmware only and cannot be modified at any time by the caller.
- The size of the siop data area pointed to by siopdatap must be of at least `sizeof(struct siopdata)` bytes, and must be aligned to a four-byte boundary.
- When using the Firmware Debug Log, the user must call the debug trace initialization routine **sdt_tinit()** prior to initializing the firmware.

siop_cmd()

NAME

siop_cmd —SIOP Firmware command call

SYNOPSIS

```
#include "scsi.h"            /* SCSI specific definitions */
#include "ncr.h"             /* firmware structure definitions */
#include "ncr710.h"          /* hardware/firmware specific definitions */
#include "scsi_err.h"        /* error definitions */

void siop_cmd(siop_struc, siopdatap)
struct siop_struc *siop_struc;/* SIOP Firmware command structure
                         pointer (refer to Appendix B) */
struct siopdata *siopdatap;   /* SIOP Firmware global data pointer */
```

DESCRIPTION

The function of the command routine is to receive a command structure from a user and execute the requested operation.

RETURN VALUE

none

ERROR CONDITIONS

Refer to `incl/scsi_err.h` or Appendix D (*Returned Errors*) for defined error values.

Error codes, associated with a command, are posted asynchronously through the user supplied return status routine found in the command structure.

NOTES

Once a call is made to this routine, the user cannot modify any field within the command structure associated with the call until the user supplied return status routine is invoked by the Firmware. The user is responsible for making sure the memory used for the command structure is cache coherent.

The siop data area, pointed to by *siopdatap*, is for use by the Firmware only and cannot be modified at any time by the caller.

siop_int()

NAME

siop_int —SIOP interrupt handler

SYNOPSIS

```
#include "scsi.h"          /* SCSI specific definitions */
#include "ncr.h"           /* firmware structure definitions */
#include "ncr710.h"        /* hardware/firmware specific definitions */
#include "scsi_err.h"      /* error definitions */

void siop_int(siopdatap)
struct siopdata *siopdatap;/* SIOP Firmware global data pointer */
```

DESCRIPTION

The interrupt handler routine is called by the code which is invoked when the SIOP interrupt occurs. This routine contains the functions necessary to execute in the interrupt mode and to recover from intermediate SIOP and Firmware problems. This is the routine in which the Firmware will idle when in the polled mode. For initiator and target mode commands, this routine is responsible for calling the status return routine supplied by the user.

RETURN VALUE

none

ERROR CONDITIONS

none

NOTES

Results are indeterminate if an SIOP interrupt did **not** occur and this routine was called by the user.

The siop data area, pointed to by *siopdatap*, is for use by the Firmware only and cannot be modified at any time by the caller.

This entry point **must** be called at or above the interrupt level of the SIOP.

sdt_tinit()

NAME

sdt_tinit —Initialize the SCSI debug trace log

SYNOPSIS

```
#include "scsi.h"             /* SCSI specific definitions */
#include "scsi_dbg.h"         /* SCSI debug defines and macros */
#include "ncr.h"              /* firmware structure definitions */

void sdt_tinit(sdtptr, siopdatap)
struct sdt_tvalue *sdtptr;    /* pointer to a structure which contains init
                         values (refer to Appendix B) */
struct siopdata *siopdatap;   /* SIOP Firmware global data pointer */
```

DESCRIPTION

The function of this routine is to initialize debug tracing. Debug tracing is useful when problems in the Firmware are encountered and the source of the problem cannot be detected in any other manner. Trace entries are logged in the memory range pointed to by the boundary addresses in the *sdt_tvalue* structure.

One of the elements in the *sdt_tvalue* structure is a flag to enable or disable debug tracing. If the *sdt_tvalue* structure is located in NVRAM, then enabling debug tracing is done by simply setting the flag to the proper value prior to calling this routine. Debug tracing is not normally enabled because the associated overhead slows the performance of the Firmware. If debug tracing is enabled, it is recommended to call this routine prior to the call to the Firmware initialization routine (**siop_init()**).

RETURN VALUE

none

ERROR CONDITIONS

none

NOTES

This routine may be called any number of times with succeeding calls resetting the debug log back to the beginning. Calling this routine is allowed prior to calling the SIOP Firmware initialization (**siop_init()**) routine.

Performance of the SIOP Firmware is drastically altered when debug tracing is enabled. For each SIOP command, many debug log entries can be made thus significantly altering the time it takes to execute the command.

Each time the user changes the location of the *siopdata* structure (changes the value of *siopdatap*) for a call to **siop_init()**, a new call to **sdt_tinit** with the new *siopdata* structure must be made to re-enable debug logging.

For more detailed information concerning use of the debug log, refer to Chapter 5, *Special Topics*.

sdt_alloc()

NAME

sdt_alloc —Allocate a SCSI debug trace entry

SYNOPSIS

```
#include "scsi.h"             /* SCSI specific definitions */
#include "scsi_dbg.h"         /* SCSI debug defines and macros */
#include "ncr.h"              /* firmware structure definitions */

struct trace_entry *sdt_alloc(siopdatap)
struct siopdata *siopdatap;   /* SIOP Firmware global data pointer */
```

DESCRIPTION

The function of this routine is to return a pointer to a debug trace entry. The returned pointer is the next sequential entry allocated from the block of memory which was assigned for debug logging with the call to **sdt_tinit**. The entry is used to hold information that pertains to the code being traced. Debug log entry wraparound is possible because the debug log is a circular buffer. The size of the entry is given by `sizeof(struct trace_entry)`.

RETURN VALUE

Pointer to the trace entry.

ERROR CONDITIONS

none

NOTES

Refer to Chapter 5, *Special Topics,* for more information.

sfw_getrev()

NAME

sfw_getrev — Return Firmware Revision String

SYNOPSIS

```
#include "scsi.h"          /* SCSI specific definitions */
#include "ncr.h"           /* firmware structure definitions */
#include "ncr710.h"        /* hardware/firmware specific definitions */
#include "scsi_err.h"      /* error definitions */

UINT sfw_getrev(string_buffer, max_size);

UCHAR   *string_buffer;    /* revision string is placed here */
UINT     max_size;         /* number of bytes available in string_buffer */
```

DESCRIPTION

The sfw_getrev() entry point provides a release ID string that identifies the Firmware. The calling application provides a pointer to MPU-writable memory and the number of bytes available there.

Up to `max_size` bytes are written into the `string_buffer`. sfw_getrev() always returns the number of bytes *available*. If the number of bytes available is not greater than `max_size`, then the return value is just the same as the number of bytes *written*. A return value of greater than `max_size` means that the given number of bytes are available, but that only `max_size` will have been written into the buffer.

RETURN VALUE

The number of bytes of Firmware identification that are available.

ERROR CONDITIONS

none

NOTES

A `max_size` of 32 bytes should be adequate for all available data. If you must be sure, however, allocate space for `string_buffer` after determining the space available. The following shows how one might accomplish this:

```
int     avail_size;
char   *string_buffer;

avail_size = sfw_getrev(NULL, 0);
string_buffer = malloc(avail_size + 1);

if (string_buffer != NULL) {
    (void) sfw_getrev(string_buffer, avail_size);
    string_buffer[avail_size] = '\0';
}
```

In the preceding example, note that if a null-terminated string is desired, it is up to the Application to provide enough space and insert the terminating '\0'.

NCR BUILD TOOLS 4

Introduction

The NCR build tools are a set of utilities provided to compile NCR SCSI SCRIPTS source modules. The utilities provided are the SCRIPTS compiler and the SCRIPTS preprocessor. The compiler, as written by the NCR Corporation, is used to compile general purpose SCRIPTS source files into executable NCR 53C710 machine instructions. The preprocessor, written specifically for use in the Firmware environment, provides a mechanism whereby SCRIPTS data references can be changed from the values assigned at link time to new values defined dynamically at run time as required by the Firmware.

SCSI SCRIPTS Data Reference Relocation

Because the `siopdata` data area required by the SIOP Firmware is dynamically allocated, the references to this data area by SCRIPTS must then be changed from the link time values through a patching process. This patching is done at run-time by the **siop_init()** routine prior to SCRIPTS execution.

The SCRIPTS preprocessor utilities **n710p68k** and **n710p80k** are provided to generate a relocation table used by **siop_init()** to perform this patch. This table contains entries pointing to each script array contained in the script source file, followed by offset values that point to locations within each script array containing data references that must be patched.

During the Firmware initialization, the **siop_init()** function walks this relocation table and patches all locations pointed to by this table to adjust for the base address of the dynamically allocated `siopdata`.

Execution of the SCRIPTS preprocessor is required only when the SCRIPTS source code is recompiled. However, the patching of the SCRIPTS is performed on each call to **siop_init()**. Multiple calls to **siop_init()** perform this relocation independently of any relocations performed on previous calls to **siop_init()**.

Example Usage of the NCR Build Utilities

What follows is an extract from a Makefile showing an example **.n.o** make rule for compiling a SCRIPTS module. This example is specific to a SYSTEM V/68 host. For a SYSTEM V/88 host, `N710P` should be set to use **n710p80k** and `N710C` should likewise be set to **n710c80k**.


```
TMPDIR = ./tmp_hdrs
INCDIR = ../incl

CPP    = cc -E
N710C  = n710c68k
N710P  = n710p68k

PFLAGS = -Um68k -UsysV68 -Um88k -Uunix -I$(INCDIR)

EXTHDRS=    $(INCDIR)/ncr.h \
            $(INCDIR)/ncr710.h \
            $(INCDIR)/scsi.h \
            $(INCDIR)/scsi_err.h \
            $(INCDIR)/sfw_cnfg.h

.n.o:
1       rm -rf $(TMPDIR); mkdir $(TMPDIR)
2       @for i in $(EXTHDRS); \
3       do \
4            j=‘basename $$i‘; \
5            grep ’^#.*[^\]$$’ $$i >$(TMPDIR)/$$j; \
6       done
7       $(CPP) $(PFLAGS) $*.n | \
8       sed -e ’s/ */ /g’ -e ’/^# *ident/d’ -e ’/^# *pragma/d’ \
9            -e ’/^# *[0-9][0-9]*/d’ -e ’s/\[ /\[/g’ \
10           -e ’s/ \]/\]/g’ > $*.i
11      $(N710P) $*.i
12      $(N710C) $*.i -u -o $*.j
13      sed “/typedef.*ULONG/s/long/int/g” $*.j > $*.c
14      $(CC) -c $*.c
15      rm -rf  $(TMPDIR) $*.i $*.j $*.c
```

Note **With the following explanations, refer to the numbered columns to the left of the ".n.o" make rule shown above.**

Line(s)	Function
1	Create a directory, `TMPDIR`, into which filtered headers will be placed.
2-6	Strip the headers down to only the single-line preprocessor directives.
7	Apply the C preprocessor to perform macro substitution on the SCRIPTS module. Pipe the output to **sed** for more preprocessing.
8-10	Use **sed** to remove blank lines, unknown compiler directives, and other **cpp** ejecta which are meaningless to the build utilities (or worse).
11	Invoke the Motorola-supplied SCRIPTS preprocessor.
12	Compile using NCR's SCRIPTS compiler.
13	Convert the `ULONG` typedef from an `unsigned long` to an `unsigned int`.
14	Turn the *file*.c output of the SCRIPTS compiler into an object module.
15	Remove all intermediate files and the directory of massaged header files.

n710p68k (n710p80k)

NAME

n710p68k (**n710p80k**) —Preprocessor for NCR SCSI SCRIPTS files

SYNOPSIS


```
extern  unsigned relocation[];
extern  unsigned script_ptr[];
```

n710p68k *file*

DESCRIPTION

The **n710p68k (n710p80k)** is the NCR SCSI SCRIPTS preprocessor resident on an MC**68*xxx*** (MC**8**810**0**) host that preprocesses a single SCRIPTS *file* prior to compilation by the SCRIPTS compiler. Execution of this utility is required for any SCRIPTS file compiled for the Firmware environment to enable all SCRIPTS data references to be made relocatable. This utility takes as input either a *file*.**n** SCRIPTS source file or a *file*.**i** output from **cpp** and generates as output *file*.**i** containing global declarations for a relocation integer array `relocation[]` and SCRIPTS pointer integer array `script_ptr[]`. Source modules referencing these global arrays **MUST** make the following declaration using the **EXACT** same names and types:

```
extern unsigned relocation[];
extern unsigned script_ptr[];
```

The `relocation[]` array contains unsigned integer pairs consisting of a token and associated integer value:

Token	Token Value	Token Use	Associated Integer Value
SCRIPT_BASE	0x98080000	Script pointer token	Pointer to SCRIPTS
MEM_SIZE	0x0F000001	Script byte size entry token	Size in bytes of SCRIPTS
DATA_OFFSET	0x88080000	Data offset token	Offset location from SCRIPTS
CODE_OFFSET	0x80080000	Code offset token	Offset location from SCRIPTS
TABLE_END	0x60000040	End of relocation table token	N/A

The `script_ptr[]` array contains pointers to the starting location of each SCRIPTS declared with a `PROC` **n710c68k (n710c80k)** compiler directive. Array indexes used to access `script_ptr[]` array elements are global integer variables generated by the preprocessor and placed in the output file *file*.**i**. The variable name of an array index is formed from the name of the SCRIPTS array, with the suffix "_idx" appended. The steps in the example below illustrate how to reference a SCRIPTS pointer for the SCRIPTS array selected.

1. Declare the external `script_ptr[]` array:

   Example: `extern unsigned script_ptr[];`

2. Declare the external array index names:

   Example: `extern unsigned selected_idx;`

3. Reference the `script_ptr[]` entry for selected:

   Example: `script_addr = script_ptr[selected_idx];`

RETURN VALUE

If **n710p68k (n710p80k)** is successful, then 0 is returned. If **n710p68k (n710p80k)** is not successful, then a nonzero value is returned and an error message is printed to *stderr*.

NOTES

All variables referenced by the SCRIPTS must be contained in a single global data structure. Furthermore, all references within SCRIPTS instructions to these variables must be made with the following syntax:

```
PASS(NCROF(variable))
```

where the length of the string name for *variable* cannot exceed 25 characters. The C macro `NCROF` is defined as:

```
#define NCROF(variable) (UINT) &((struct siopdata *) 0)->variable
```

Here `siopdata` is the name of the global data structure containing all relocatable data references. An example SCRIPTS source level instruction referencing a relocatable variable is:

```
move memory 1, PASS(NCROF(nopmsg)), PASS(NCROF(q_taginflg))
```

If the C preprocessor **cpp** is run prior to the execution of **n710p68k (n710p80k)**, then it is necessary to prevent the `NCROF` macro from being expanded until after **n710p68k (n710p80k)** has executed.

All SCRIPTS instructions must be contained within a single source module.

The identifiers `rel_d`, `rel_c`, and `NCROF` are reserved and may not be used in the SCRIPTS source code. The SCRIPTS `PROC` labels `relocation` and `first_datap` are reserved.

SEE ALSO

n710c68k (n710c80k)

n710c68k (n710c80k)

NAME

n710c68k (**n710c80k**) —Compiler for NCR SCSI SCRIPTS files

SYNOPSIS

n710c68k *file [options] [outfile]*

DESCRIPTION

The **n710c68k (n710c80k)** is the NCR SCSI SCRIPTS compiler for the NCR 53C710 SCSI I/O Processor resident on an MC**68*xxx*** (MC**8810**0) host that translates SCRIPTS contained in *file* to C language integer arrays of NCR 53C710 opcodes and operands. The compiled *outfile* may then be compiled by a C language compiler to produce an *outfile*.**o** object file. This **n710c68k (n710c80k)** compiler does not support directory paths in the specification of either the *file* or *outfile*, requiring *file* and *outfile* to reside in the current directory.

The following options apply:

-e *errorfile* This option generates a file where all the error information is stored. If the **-e** option is used without specifying errorfile name, then the errorfile name defaults to *file*.**err**.

-l *listfile* This option determines if a listfile is generated and if so what the name of the filename is. If the **-l** option is given without specifying a filename, then the filename defaults to *file*.**lis**. For every instruction, the listfile lists an offset from the beginning of the SCRIPTS, the longword instruction, the longword address, and the corresponding ASCII source instruction. Labels appear on a line by themselves as they are encountered in the SCRIPTS. Also produced is a list of absolute or relative variables, and their location in the SCRIPTS. This is followed by a list of labels and label locations that appear in the SCRIPTS. The location is an offset from the beginning of the SCRIPTS.
The final list gives the label patches. Label patches are offsets into the SCRIPTS where a label is referenced. They are called patches because the absolute address of the labels must be patched into the SCRIPTS at runtime.

-o *outfile* This option determines if a C-compilable data file is generated and if so what the name of the file is. If the **-o** is given without specifying a filename, the output filename defaults to a *file*.**out**.

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

-v This option prints all relevant information about the compilation process to the screen for the user to view.

-z *debugfile* This option generates a file that is necessary if the SCRIPTS debugger is to be used. If the debugger is used, this is the file that is loaded to begin the debug process. If the **-z** option is given without specifying a debugfile name, then the debugfile name defaults to *file*.**sod**.

RETURN VALUE

If **n710c68k (n710c80k)** is successful then 0 is returned. If **n710c68k (n710c80k)** is not successful, then a nonzero value is returned.

NOTES

The `PASS()` directive allows the string contained within the parentheses to be ignored by the SCRIPTS compiler and passed directly to the output file. For example, the statement `PASS(#include header.h)` results in the line `#include header.h` to be placed in the *outfile*. The string is limited to 32 characters in length.

The **n710c68k (n710c80k)** resolves all data references as direct references, which are then resolved at link time. To generate SCRIPTS that contain data references that can be relocated at run time, it is necessary to invoke the NCR preprocessor **n710p68k (n710p80k)** to generate a run-time relocation table.

The compiler declares all compiled SCRIPTS as C array declarations of type `ULONG`. The compiler typedefs `ULONG` and "`unsigned long`" in the *outfile*.

All entries past a `PROC` directive up to the next `PROC` directive (or the end of the source file) are included as elements of the C array produced for that `PROC` directive.

SEE ALSO

n710p68k (n710p80k)

SPECIAL TOPICS 5

Introduction

This chapter covers topics which most users will not use in the normal course of SCSI operation. It is provided as a guide for those who wish to exercise the full functionality of the Firmware.

Firmware Debug Logging

This section describes the operation of the Firmware debug trace mechanism. Debug tracing is useful when problems are encountered while accessing the SCSI bus through the Firmware. The debug trace can reveal the source of many functional discrepancies.

Debug Logging Interface

sdt_tinit() This routine is for debug logging initialization. The user calls this entry point to enable the debug logger and provide it with memory resources.

sdt_alloc() This routine is for debug logging. If debug logging has been enabled, this entry point is called to get the next block of memory to be used for debug logging.

Functional Overview

Implementing a debug trace is composed of three levels of operation. These levels are the user level setup, code level setup, and trace display.

The user level setup is the first step and involves allocating a block of memory for the debug log and initializing an `sdt_tvalue` structure (refer to Appendix B). The `begin` and `end` fields of the structure are initialized to the beginning and ending address of the allocated memory block. A value of 0x44 (ASCII 'D') is installed in the `flag` field to enable debug logging.

The code level setup is the next step. Debug logging becomes enabled when a call to **sdt_tinit()** passes in the `sdt_tvalue` and `siopdata` structures. **sdt_tinit()** checks the flag field in `sdt_tvalue` for a value of 0x44 (ASCII 'D'). If the flag is set to this enable value then the routine will set an enable flag in the `siopdata` structure. If the flag is not set to this enable value then the routine will clear the enable flag in the `siopdata` structure.

After logging is enabled, a user calls **sdt_alloc()** to get a pointer to the next entry to be used for debug logging. A user makes this call from within a section of code which is suspected to introduce problems or when critical information is available to be saved in the log.

The final step in debug logging is retrieval and analysis of the logged information. This is done after a problem has occurred and the user is trying to determine the source of that problem.

Debug Trace Memory Structure

The following figure is a memory diagram of the debug trace log and is intended to show the order of trace entry allocation.

On the first **sdt_alloc()** call, the `current` pointer is equal to the `start` location `current` pointer towards the low end of memory, one frame at a time, until the *special first entry* is reached. When the *special first entry* is reached, a wraparound condition exists and the `current` pointer will again be set to `start`.

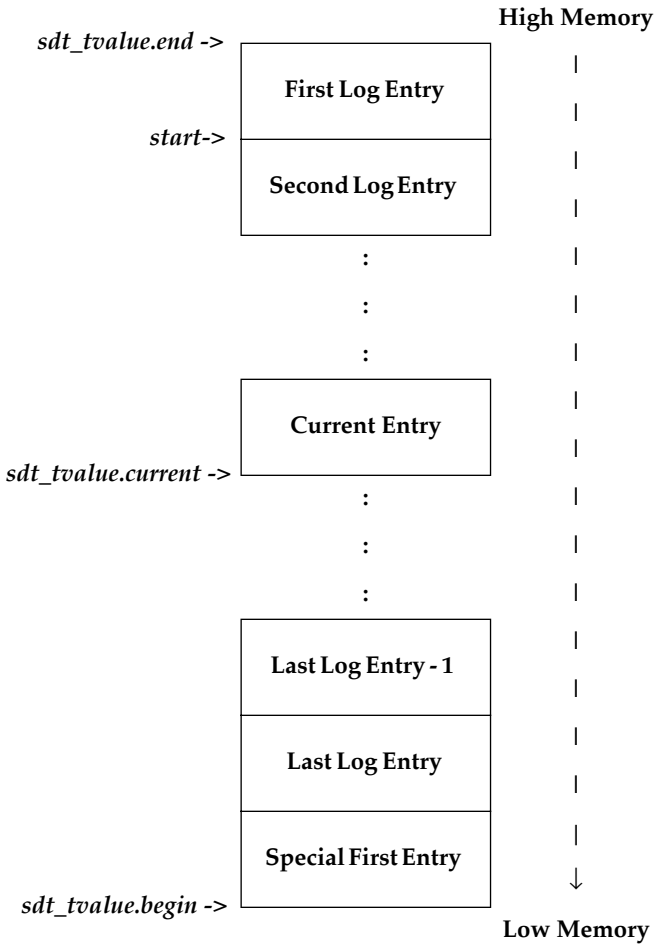

Figure 5-1. Debug Trace Memory Structure

Example

This section shows, through example, how to enable debug logging, how to record debug information in a trace entry, and how to display the log. A ROM debugger will be used to initialize the sdt_tvalue structure and display the log. The information which will be displayed is from currently embedded log entry points in the Firmware.

Examples shown here use the MVME167 ROM debugger; however, these same commands can be used with the other ROM debuggers.

5

User Level Setup

This section demonstrates a method for enabling debug logging by the Firmware. The ROM debugger calls **sdt_tinit()** before its first access to a local SCSI device after a board reset. Before this access is made, the sdt_tvalue structure must be located and initialized. Use the ROM debugger to find and initialize this structure which is located in NVRAM.

1. Find the address of the string **'PRVT'** using the **BS** command.

```
167-Bug>bs fffc0000 fffc2000 'PRVT'
Effective address: FFFC0000
Effective address: FFFC2000
FFFC16F8
167-Bug>
```

   String address = **0xFFFC16F8**

2. The pointer to sdt_tvalue resides at **0xFFFC16F8 + 4**. Display this pointer value using the **MD** command.

```
167-Bug>md fffc16f8+4
FFFC16FC  FFFC 1758 0000 0000  0014 0032 5350 4346  ...X.......2SPCF
167-Bug>
```

   sdt_tvalue structure address = **0xFFFC1758**

3. Initialize sdt_tvalue to the following values using the **MM** command.

   Set the debug_flag to **0x00440000** at location **0xFFFC1758**.
   Set the begin address to **0x00400000** at location **0xFFFC175C**.
   Set the end address to **0x00800000** at location **0xFFFC1760**.

```
167-Bug>mm fffc1758;l
FFFC1758 00000000? 00440000
FFFC175C 00000000? 00400000
FFFC1760 00000000? 00800000
FFFC1764 00092691? .
167-Bug>
```

4. Clear the debug trace memory area using the **BF** command.

```
167-Bug>bf 400000 800000 0
Effective address: 00400000
Effective address: 007FFFFF
167-Bug>
```

5. sdt_tvalue is initialized to enable debug logging but debug logging is not yet enabled. To enable debug logging and to log some debug trace entries, execute an **IOP** command to a local SCSI device.

Code Level Setup

Embedded in the debugger code is a decision to call **sdt_tinit()** before the first access to the local SCSI bus after a board reset. Therefore, the debug logging functionality will have been enabled during the first part of the **IOP** command because of the call to **sdt_tinit()**. Later in the command, a call to **siop_cmd()** will have been made to request the Firmware to access a local SCSI device. Trace entries will have been logged to the debug log during execution of the Firmware. The following excerpt from the Firmware is where the MPU code is invoked in response to a command complete interrupt generated by the SCRIPTS.

```
UINT sdt_flag = siopdatap->sdt_flag; /* debug logging flag */

/*
 * debug log, if enabled
 */
if (sdt_flag == DEBUG_ENABLE)
{
    register struct trace_entry *log; /* pointer to debug log entry */

    /*
     * raise interrupt mask
     * get trace memory block
     * log block as Firmware entry
     * log eyecatcher (source of interrupt)
     * log some SIOP register values
     * log current command structure
     * log command structure (siop_struc)
     * restore interrupt mask
     */
    s = splhi();
    log = (struct trace_entry *)sdt_alloc(siopdatap);
    log->eyepicker = SDT_NCR;
    log->eyecatcher = SDT_COMP;

    save_siop(log, siopdatap);
    NCR_LOG(log->data)->cns = *cnsptr;
    NCR_LOG(log->data)->cmd = *(cnsptr->curcmd);
    splx(s);
}
```

The following procedure was adhered to in the above code.

1. If the global value `sdt_flag` located in `siopdata` is set to an ASCII 'D' (0x44) then continue with the trace operation.
2. Raise the interrupt mask using the function **splhi()** (refer to Appendix C). Save the return value so it can be restored upon completion of the trace.
3. Call the function **sdt_alloc()** to get a pointer to a debug log entry.
4. Set the `eyepicker` to a unique four byte ASCII string. This field is used to identify who (i.e., SIOP, driver, etc.) is writing to the trace log.
5. Set the `eyecatcher` to a unique four byte ASCII string. This field is used to identify the log point.
6. Enter all of the items of interest into the debug log.
7. Restore the interrupt level back to the original value using the function **splx()** (refer to Appendix C).

Debug Trace Display

This section demonstrates a method for displaying the trace entries logged by the Firmware.

1. When the **IOP** command has completed, the debugger prompt will be displayed. Display the special first trace entry which is always logged at the `begin` address of the `sdt_tvalue` structure. The value used for `begin` in this example is taken from the enable procedure above and has a value of **0x400000**.

```
167-Bug>md 400000:10
00400000  0044 0000 0040 0000  007F FFF0 007F D070  .D...@.........p
00400010  3638 4B3D 0000 0098  3838 4B3D 0000 004C  68K=....88K=...L
167-Bug>
```

The first 32 bytes of the first trace entry are for log management. The following definitions are referenced to the offset from the beginning address of the allocated debug log block as specified in `sdt_tvalue`.

0x00 - Flag — This is the enable/disable flag. This value will always be XX44XXXX.

0x04 - Begin — This is the beginning address of the debug log as provided in `sdt_tvalue`.

0x08 - End — This is the end of the debug log as determined by **sdt_tinit()**. The values provided in `sdt_tvalue` may not have indicated a block which was sized to X number of entries. There could

have been extra bytes in which a trace entry would not fit. **sdt_alloc()** uses this field to determine the bounds of the log when a wraparound situation occurs.

0x0C - Current — This is a pointer to the last trace entry which was logged.

0x10 - 68K — This is an eyecatcher to inform the user that the next field (68Ksize) is the number to specify when displaying a trace entry with the 68K ROM debugger **MD** command.

0x14 - 68Ksize — This is the number to specify when displaying a trace entry with the 68K ROM debugger **MD** command.

0x18 - 88K — This is an eyecatcher to inform the user that the next field (88Ksize) is the number to specify when displaying a trace entry with the 88K ROM debugger **MD** command.

0x1C - 88Ksize — This is the number to specify when displaying a trace entry with the 88K ROM debugger **MD** command.

2. Extract the following values from the `first_entry`.

   The *current* trace entry pointer = **0x7FD070**.
   The trace entry frame *size* = **0x98**.

3. Display trace entries from the *current* trace frame to sequentially older entries in memory.

```
167-Bug>md 7fd070:98
007FD070  4E43 5220 434F 4D50  0000 0000 0000 0000  NCR COMP........
007FD080  0000 0000 AE01 20D4  0000 8080 0000 0000  ...... .........
007FD090  0F00 0000 FF00 0000  0000 F000 1200 0000  ................
007FD0A0  FFF4 7036 0100 0000  9808 0000 0000 6DFC  ..p6..........m.
007FD0B0  0000 6DFC FF83 33CC  0008 020E 2100 3DE0  ..m...3.....!.=.
007FD0C0  0083 A1C8 0000 0200  0000 E000 0000 000A  ................
007FD0D0  0000 1098 0000 0001  0000 10E7 0000 0006  ................
007FD0E0  0000 10C0 0000 0001  0000 611C 0001 0000  ..........a.....
007FD0F0  0000 1084 0008 000E  0000 0000 0000 0000  ................
007FD100  0000 0000 0000 60CC  0000 0000 8004 000E  ......`.........
007FD110  0000 0000 0000 0000  0000 000A 2800 0000  ............(...
007FD120  0000 0000 0100 0000  0000 0007 0000 10B1  ................
007FD130  0103 0100 0000 0000  0000 0000 0000 0006  ................
007FD140  0000 10C0 C001 0301  1900 0000 0000 0000  ................
007FD150  0000 0000 0000 E000  0000 0000 0000 0000  ................
007FD160  0000 0000 FF84 E120  0000 0000 0000 0200  ....... ........
007FD170  0000 0000 0000 0000  0000 0000 0000 0000  ................
007FD180  0000 1084 0000 0000  0000 0001 0000 6F64  ..............od
007FD190  0000 0000 0000 0000  0000 0000 0000 0000  ................
```

```
167-Bug>
007FD1A0  4E43 5220 5245 534C  8100 0000 0000 0000  NCR RESL........
007FD1B0  0000 0000 AE01 30D4  4700 8080 6780 8080  ......0.G...g...
007FD1C0  0F00 0000 FF00 0000  0000 F000 1200 0000  ................
007FD1D0  FFF4 7021 0100 0800  9808 0000 0000 67EC  ..p!..........g.
007FD1E0  0000 67EC FF83 310A  000A 400E 2100 3DE0  ..g...1...@.!.=.
007FD1F0  0083 98F6 0000 0200  0000 E000 0000 000A  ................
007FD200  0000 1098 0000 0001  0000 10E7 0000 0006  ................
007FD210  0000 10C0 0000 0001  0000 611C 0001 0000  ..........a.....
007FD220  0000 1084 0000 000E  0000 0000 0000 0000  ................
007FD230  0000 0000 0000 60CC  0000 0000 8004 000E  ......`.........
007FD240  0000 0000 0000 0000  0000 000A 2800 0000  ............(...
007FD250  0000 0000 0100 0000  0000 0007 0000 10B1  ................
007FD260  0103 0100 0000 0000  0000 0000 0000 0006  ................
007FD270  0000 10C0 C001 0301  1900 0000 0000 0000  ................
007FD280  0000 0200 0000 E000  0000 0000 0000 0000  ................
007FD290  0000 0000 FF84 E120  0000 0000 0000 0000  ....... ........
007FD2A0  0000 0000 0000 0000  0000 0000 0000 0000  ................
007FD2B0  0000 1084 0000 0000  0000 0001 0000 6F64  ..............od
007FD2C0  0000 0000 0000 0000  0000 0000 0000 0000  ................

167-Bug>
007FD2D0  4E43 5220 4449 5343  0000 0000 0000 0000  NCR DISC........
007FD2E0  0000 0000 AE01 20D4  0000 8080 0000 0404  ...... .........
007FD2F0  0F00 0000 FF00 0000  0000 F000 1200 0000  ................
007FD300  FFF4 7036 BA00 0000  9808 0000 0000 6E68  ..p6..........nh
007FD310  0000 6E68 FF83 2FDE  000A 400E 2100 3DE0  ..nh../...@.!.=.
007FD320  0083 9E46 0000 0200  0000 E000 0000 000A  ...F............
007FD330  0000 1098 0000 0001  0000 10E7 0000 0006  ................
007FD340  0000 10C0 0000 0001  0000 611C 0001 0000  ..........a.....
007FD350  0000 1084 000A 000E  0000 0000 0000 0000  ................
007FD360  0000 0000 0000 60CC  0000 0000 8004 000E  ......`.........
007FD370  0000 0000 0000 0000  0000 000A 2800 0000  ............(...
007FD380  0000 0000 0100 0000  0000 0007 0000 10B1  ................
007FD390  0103 0100 0000 0000  0000 0000 0000 0006  ................
007FD3A0  0000 10C0 C001 0301  1900 0000 0000 0000  ................
007FD3B0  0000 0200 0000 E000  0000 0000 0000 0000  ................
007FD3C0  0000 0000 FF84 E120  0000 0000 0000 0000  ....... ........
007FD3D0  0000 0000 0000 0000  0000 0000 0000 0000  ................
007FD3E0  0000 1084 0000 0000  0000 0001 0000 6F64  ..............od
007FD3F0  0000 0000 0000 0000  0000 0000 0000 0000  ................
167-Bug>
```

Firmware Debug Log Map

The following section explains the fields within a Firmware trace entry. The generic trace entry, as defined in the header file *scsi_dbg.h,* is:

```
struct trace_entry {
        UINT eyepicker;
        UINT eyecatcher;
        UCHAR data[LARGE_AMOUNT];
};
```

All Firmware trace entries will contain an *eyepicker* entry of **NCR** that identifies this as a Firmware trace entry (as opposed to a driver trace frame) and an *eyecatcher* label that identifies the unique log point within the Firmware.

The trace *data* area of each trace entry is mapped according to the following structure defined in **incl/ncr710db.h**:

```
/*
 * NCR trace items.
 *
 * The size of this structure must NEVER get larger than the size of
 * the trace_entry.data array[LARGE_AMOUNT] in scsi_dbg.h.
 */

struct ncr_debug {
    UINT inst1;            /* 1st 32 bits of next SIOP instruction */
    UINT inst2;            /* 2nd 32 bits of next SIOP instruction */
    UINT inst3;            /* 3rd 32 bits of next SIOP instruction */
    NCR_SIOP siop;         /* software model of the chip           */
    SIOP_CNS cns;          /* copy of the current nexus structure  */
    SIOPCMD cmd;           /* copy of the siop_struc               */
};
```

The mapping table below shows the relative placement of all structures and structure elements for a typical screen display. For a given Firmware log entry, some combination of these elements will be valid.

Table 5-1. Firmware Display Frame Map

Byte	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	**eyepicker**				**eyecatcher**				*inst1*				*inst2*			
0x10	*inst3*				*sien*	*sdid*	*scntl 1*	*scntl 0*	*socl*	*sodl*	*sxfer*	*scid*	*sbcl*	*sbdl*	*sidl*	*sfbr*
0x20	*sstat2*	*sstat1*	*sstat0*	*dstat*	*dsa*				*ctest3*	*ctest2*	*ctest1*	*ctest0*	*ctest7*	*ctest6*	*ctest5*	*ctest4*
0x30	*temp*				*lcrc*	*ctest 8*	*istat*	*dfifo*	*dcmd*				*dnad*			
0x40	*dsp*				*dsps*				*scratch3*	*scratch2*	*scratch1*	*scratch0*	*dcntl*	*dwt*	*dien*	*dmode*
0x50	*adder*				**datacnt**				**dataptr**				**cdblnth**			
0x60	**cdbptr**				**statlnth**				**statptr**				**msgoutlnth**			
0x70	**msgoutptr**				**msginlnth**				**msginptr**				**siop_id**			
0x80	**curcmd**				**cmd_cntrl**				**seqphase[0-7]**							
0x90	**sgptr**				**start**				*user_defined*				*cmd_ctrl*			
0xA0	*addr_ilvl*				*lun*				*cdb_lgth*				*cdb[0-3]*			
0xB0	*cdb[4-0xB]*								*msg_in_lgth*				*msg_in_ptr*			
0xC0	*msg_in[0-0xB]*												*msg_out_lgth*			
0xD0	*msg_out_pt*				*msg_out[0-0xB]*											
0xE0	*data_count*				*data_ptr*				*link_ptr*				*scsi_phase[0-3]*			
0xF0	*scsi_phase[4-7]*				*status_ptr*				*status*				*term_count*			
0x100	*err_addr*				*qtag*				*rdyq*				*waitq*			
0x110	*top*				*seq*				*flow*				*script*			
0x120	*not used*															

The mapping table below shows which fields are valid for the given Firmware debug logging entry point. The entry points are listed according to the *eyecatcher* field.

Table 5-2. Firmware Display Data Map Key

Frame Byte Range	Variable Group	Valid at Eyecatcher Entries	Structure Name	Header File
0x08 through 0x13	NCR 53C710 opcodes	STEP	*instX*	ncr710db.h
0x14 through 0x53	SIOP H.W. Registers	**ALL EXCEPT** INIT, CMD	*ncr_siop*	ncr710.h
0x54 through 0x97	Current Nexus Structure	**ALL EXCEPT** CMD, QEKO, BRST, STEP, INIT, XSTO, SIID	*siop_cns*	ncr.h
0x98 through 0x11F	Command Structure	**ALL EXCEPT** QEKO, BRST, INIT, STEP, XSTO, SIID, BERR, SGE	*siop_struc*	ncr.h

Firmware Debug Log Entry Descriptions

The following descriptions are listed according to the associated *eyecatcher*.

BERR

structures logged- *ncr_siop*, *siop_cns*

description- The SIOP experienced a bus error while accessing the local bus. The Firmware will attempt to recover from this condition.

BRST

structures logged- *ncr_siop*

description- The SIOP received a SCSI bus reset. All outstanding command requests to the firmware are returned.

COMP

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The SCRIPTS received a **command complete** message for the current physically threaded command.

DISC

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The SCRIPTS received a **disconnect** message for the current physically threaded command. The state of the SCSI command was saved. If another command request was on the ready queue, it was enqueued for SCRIPTS command execution, otherwise, the SCRIPTS were placed in an idle loop.

IDOV

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The SCRIPTS determined that more data bytes were being transferred across the SCSI bus than what the user had indicated in the *siop_struc*. An error is logged in the *siop_struc*. The SCRIPTS will be restarted at an entry point which will either receive bytes into a bit bucket (DATA-IN) or write an error pattern to the device (DATA-OUT) until the target transitions out of the DATA phase.

INIT

structures logged- none

description- This log entry only documents that the SIOP initialization routine **siop_init()** was called.

INT

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- An invalid reselection occurred. A device for which no `siop_struc` is available, reselected the SIOP.

KICK

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The Firmware SCSI command startup routine was entered to process an `siop_struc`. A command structure was found and dequeued from the ready queue and then the command was dispatched to the SCRIPTS for execution.

LCMP

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The SCRIPTS received a **linked command complete** message for the current physically threaded command. The Firmware enqueues the next linked `siop_struc` for SCRIPTS command execution.

MREJ

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The SCRIPTS received an **message reject** message in response to a message sent to the target for the current physically threaded command. The Firmware determines which message was rejected and acts upon that determination.

PMM

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The SIOP detected that the SCSI bus changed phases before the SIOP counter register exhausted its count for the previous phase. This situation will occur when a disconnect is pending or a data underrun has happened.

PVER

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The SCRIPTS detected a SCSI protocol violation. An error is logged in the `siop_struc` and the SCRIPTS are restarted at a recovery point.

QEKO

structures logged- *ncr_siop*

description- The Firmware SCSI command startup routine was entered to process an `siop_struc` but no command structure was found on the ready queue.

RESL

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- A valid reselection occurred. A device for which a `siop_struc` is available, reselected the SIOP. The state of the command is restored and the SCRIPTS are invoked to continue execution of the command request.

SGE

structures logged- *ncr_siop*, *siop_cns*

description- The SIOP has experienced a SCSI hardware error. No recovery action is instituted.

SIID

structures logged- *ncr_siop*

description- The SIOP tried to execute an illegal instruction. SCRIPTS execution was resumed at a recovery point.

STEP

structures logged- *inst1*, *inst2*, *inst3*, *ncr_siop*

description- Single step mode has been enabled in the SIOP. Each instruction executed by the SIOP will be logged.

STO

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The SIOP selection time out timer has expired. The Firmware returns status to the user.

UDC

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The SIOP determined that an unexpected disconnect occurred. The physically threaded target transitioned to the `BUS FREE` phase. The Firmware returns status to the user.

XMSG

structures logged- *ncr_siop*, *siop_cns*, *siop_struc*

description- The SCRIPTS received an **extended** message for the current physically threaded command. The Firmware copies the received message into the `MESSAGE-IN` buffer and acts upon the message if it was a **synchronous data transfer request** or **wide data transfer request** message.

XSTO

structures logged- *ncr_siop*

description- The Firmware determined that the *external* selection time out timer has expired. This condition is similar to an **STO**. The difference is that the internal SIOP time out timer failed to work and so the SIOP had to be reset and reinitialized.

Use of the Firmware After Use by the SBC ROM Debugger

Special considerations are required when using the Firmware to talk to an SCSI device after the SBC ROM Debugger (e.g., 167Bug or 187Bug) is used to talk to that same device. ROM Debugger commands such as **IOP** (I/O Physical) and **BO** (Boot) set and expect certain configuration modes on the device. The ROM Debugger makes every attempt to leave the device in a state in which it can later be used by other software such as an operating system.

Caution must be used when returning control to the ROM Debugger from the operating system. For example, the operating system might set a device in the synchronous data transfer mode. If control is returned to the ROM Debugger, the ROM Debugger may not be able to access the device (using the asynchronous mode). The solution for this case is to RESET the SCSI device on the SCSI bus to return the device to its default state. The ROM Debugger provides a **RESET** command and **ENV** parameters to RESET the SCSI bus. The user must be aware of these restrictions, and other ROM Debugger to operating system handoff considerations, when using the Firmware.

Cache Coherency

The user is responsible for making sure all of the memory used by the Firmware is cache coherent. This includes the Firmware data area (`siopdata`), the command structure (`siop_struct`), and all buffers contained within the command structure. Generally, the proper snoop mode setting (refer to Table B-5) handles cache coherency for all onboard memory on Motorola's 68040- and 88110-based SBCs. On the MVME187, which is based on the 88100, cache coherency can be accomplished by allocating the Firmware memory in cache-inhibited areas.

Local Bus Usage by the NCR 53C710

Because the NCR 53C710 is a local bus master, it consumes local bus cycles. The NCR 53C710 uses the local bus to fetch instructions and to transfer data.

The percentage of the local bus bandwidth consumed by SCSI data transfers is determined by the SCSI data transfer rate (SDTR) and the NCR 53C710 local bus transfer rate (LBDTR). The bandwidth percentage is calculated using the equation:

$$BWP = (SDTR / LBDTR) * 100$$

The SCSI data transfer rate depends on the SCSI device which is connected to the SBC and how that device is used. In a typical system, if we assume a disk transfer is composed of an average seek (16 milliseconds), followed by an average rotational latency (8 milliseconds), followed by an 8KB data transfer, the average transfer rate would be about 300KB/sec.

The NCR 53C710 transfers SCSI data using the burst protocol defined by the MC68040 bus. The NCR 53C710 transfers a maximum of four longwords (16 bytes) during each bus tenure. At 25 MHz, the NCR 53C710 local bus transfer rate is 57MB/sec for writes and 44MB/sec for reads with parity off and 40MB/sec for reads with parity on. The typical percentage of local bus bandwidth consumed by SCSI data transfers is summarized in the following table:

Table 5-3. Typical NCR 53C710 Local Bus Usage for SCSI Data Transfers

SCSI Data Rate (MB/sec)	Write	Read (With Parity Off)	Read (With Parity On)
0.6	1.1%	1.5%	1.4%
5.0	9.0%	12%	13%

Target Mode

(THIS INFORMATION WILL BE AVAILABLE ONLY IN A FUTURE RELEASE.)

DIRECTORY STRUCTURE

A

Miscellaneous files used for the SCSI Software for the compiler and preprocessor binaries are mentioned in Chapter 4. These files are located under **bin**. Other unsupported files are located under **src** and **lib**. See Figure A-1.

The SCSI Software is mostly organized under a single directory, **sbc_scsi**. Under this directory are two main subdirectories for the include files and the SIOP Firmware. See Figure A-2.

These two subdirectories are:

incl/ Contains the SCSI Software-specific include files.

ncr710/ Contains the source files for the SIOP Firmware.

The other subdirectory under **sbc_scsi** is **sdl**, which itself has various subdirectories with several files. See Figure A-3.

All of this directory structure is depicted on the following pages. Note that not all of the files are supported. Many are for example purposes only and are released as unsupported software. Refer to the *SBC SCSI Software Release 1.1 Software Release Guide*, as listed in the *Related Documentation* section of Chapter 1, for more information.

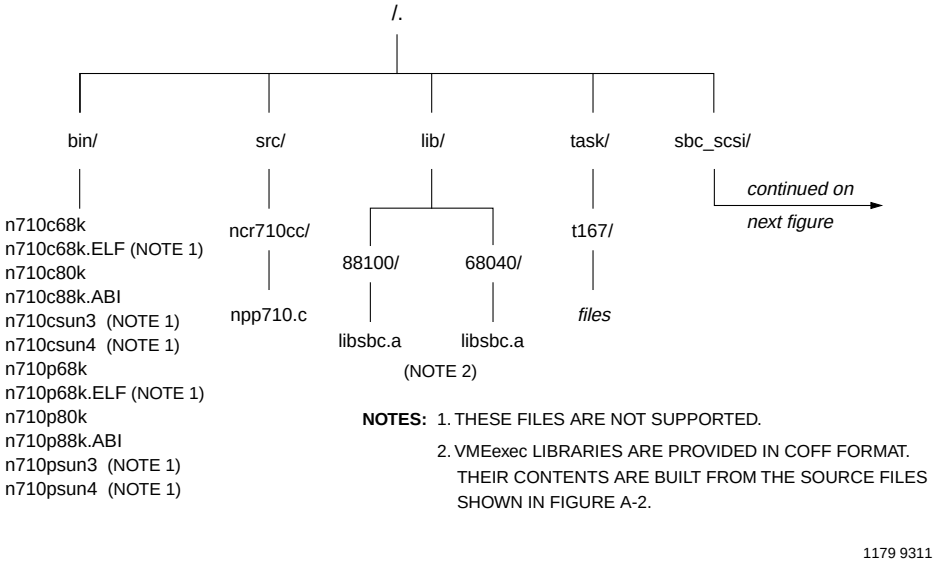

Figure A-1. Directory Structure: bin, src, and lib Files

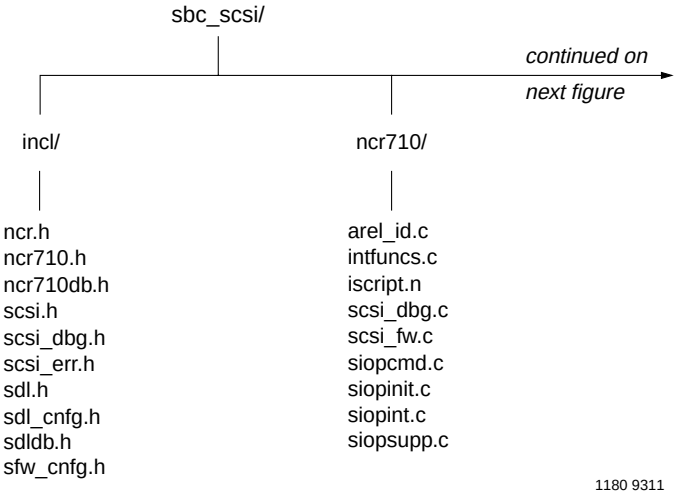

Figure A-2. Directory Structure: Include Files and SIOP Firmware

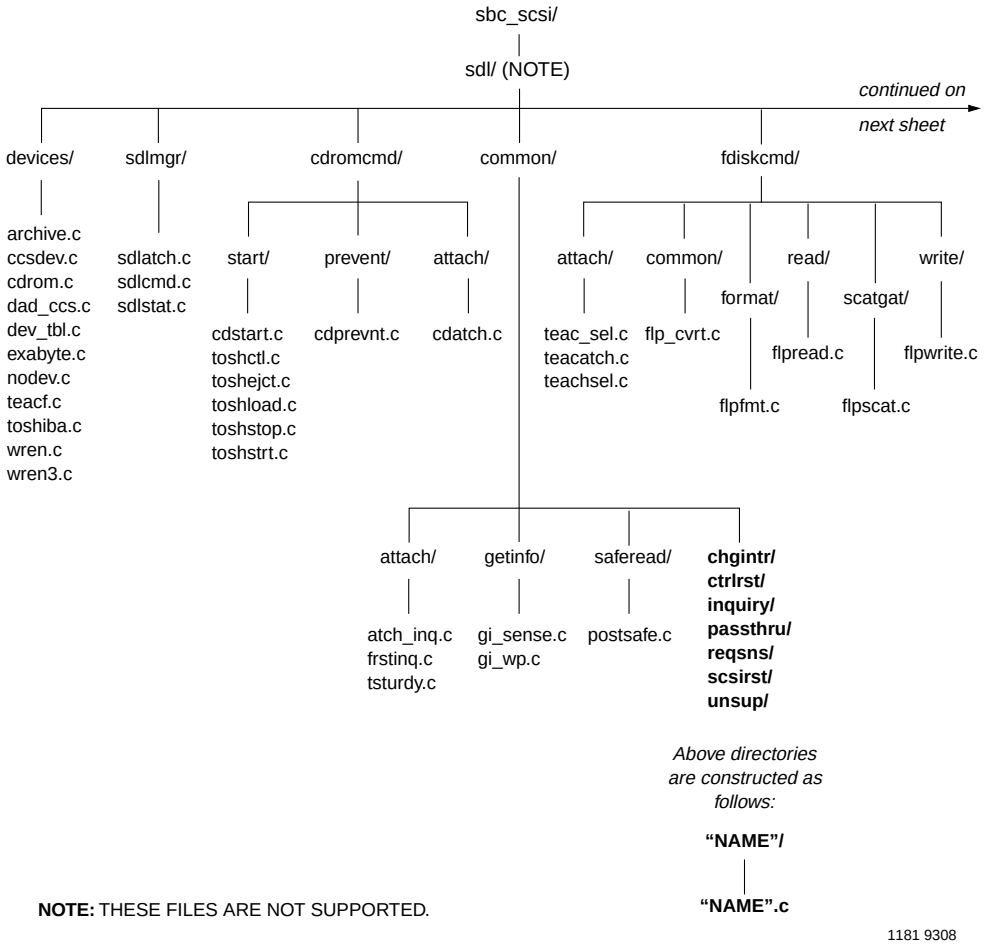

Figure A-3. Directory Structure: sdl Files (sheet 1 of 2)

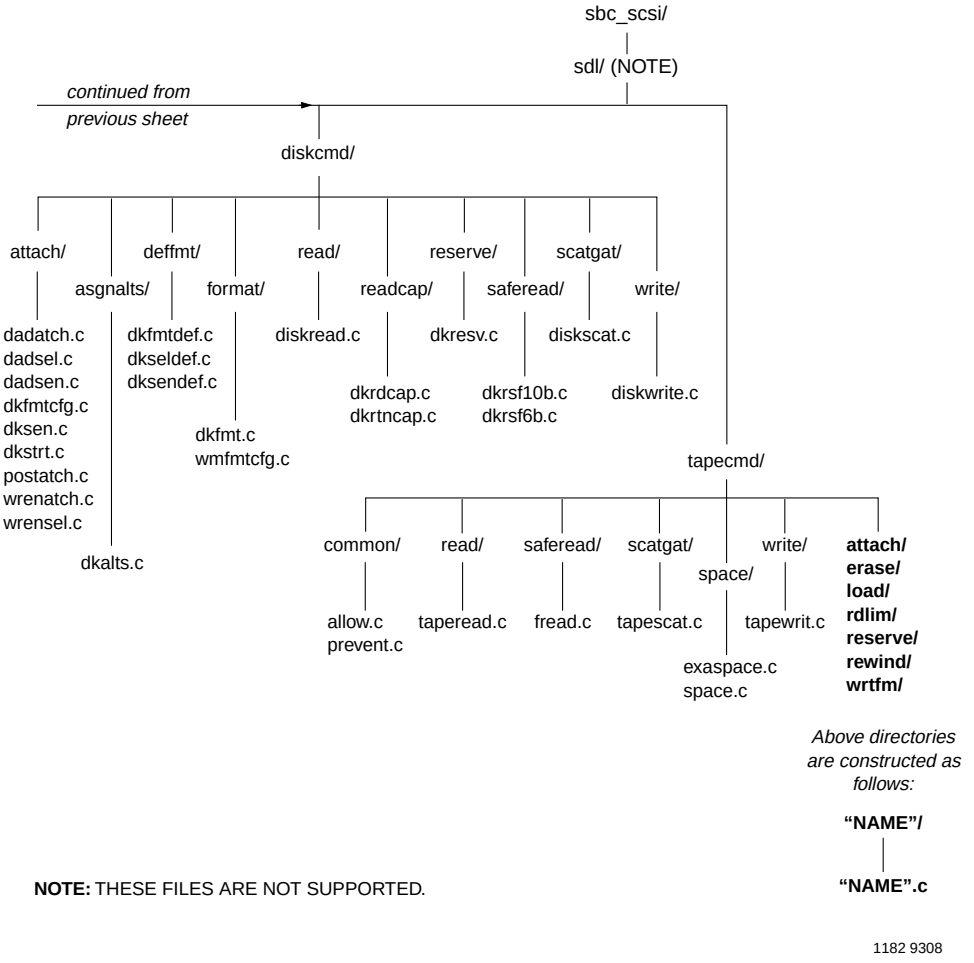

Figure A-3. Directory Structure: sdl Files (sheet 2 of 2)

SIOP FIRMWARE STRUCTURES B

Introduction

This appendix lists all of the structures which are external to or defined by the SIOP Firmware and which must be provided by the code using the services of the Firmware.

siop_struc (Command Structure)

The command structure contains all the information necessary to manage a single call to the SIOP Firmware. This structure is initialized by the user and allocated to the Firmware when the user makes a command call to the Firmware. This structure consists of control information, command information, and various pointers. Each field is defined for the command mode in which it is used. The INTATR bit, the TARGET bit, and the CONFIG bit in the Command Control field are mutually exclusive. The Firmware can operate in only one mode per command. If more than one of these bits are set in a single command structure then the results are unpredictable. Once this structure is allocated to the Firmware the user cannot modify any field within it until the structure is released at status time. **The command structure must be aligned on a four-byte boundary (longword for the CISC MVME162 /166/167; word for the RISC MVME187/197).**

Cache coherency for the command structure is the responsibility of the user. Generally, on all supported SBCs except the MVME187, the proper snoop mode setting (refer to Table B-5) handles the cache coherency. On the MVME187, it is suggested that the command structure be allocated in cache-inhibited memory.

Table B-1. Command Structure

Byte Offset (hex)	Parameter
00	User ID
04	Command Control
08	Device Address or SIOP Interrupt Level
0C	LUN
10	CDB Length or Queue Depth
14	CDB (bytes 0-3)
18	CDB (bytes 4-7)
1C	CDB (bytes 8-B)
20	Message-In Length
24	Message-In Buffer Pointer
28	Message-In Bytes (0-3)
2C	Message-In Bytes (4-7)
30	Message-In Bytes (8-B)
34	Message-Out Length
38	Message-Out Buffer Pointer
3C	Message-Out Bytes (0-3)
40	Message-Out Bytes (4-7)
44	Message-Out Bytes (8-B)
48	Data Count
4C	Data Pointer or Scatter/Gather List Pointer
50	Link Pointer
54	SCSI Phase Sequence (bytes 0-3)
58	SCSI Phase Sequence (bytes 4-7)
5C	Status Return Function Pointer
60	Status
64	Termination Transfer Byte Count
68	Error Address
6C	SCSI Queue Tag
70-84	Work Area

User ID

The user ID field is for user command logging. This field is not examined by the Firmware nor is it altered by the Firmware.

Command Control

The command control field designates the mode in which the Firmware operates. The three mode bits, INTATR, TARGET, and CONFIG, are mutually exclusive. The Firmware can operate in only one mode per command request. In addition to mode control, this field also contains command control information pertaining to the request contained within the command structure. The bits within this field are defined below. If a bit is defined for more than one mode then all modes for which that bit is valid are defined. Bits that are reserved (rsrvd) should be cleared for future compatibility.

Table B-2. Command Control Bit Definitions

BIT	31	30	29	28	27	26	25	24
NAME	INTATR	TARGET	CONFIG	rsrvd	rsrvd	rsrvd	rsrvd	rsrvd

BIT	23	22	21	20	19	18	17	16
NAME	rsrvd	rsrvd	rsrvd	rsrvd	rsrvd	PAR	FIRST	DEVRST

BIT	15	14	13	12	11	10	9	8
NAME	MIBUF	MOBUF	NO_ATN	rsrvd	rsrvd	SIOPADD	SIOPINT	SCSIRST

BIT	7	6	5	4	3	2	1	0
NAME	TAG_Q	LINK	rsrvd	S/G	D_PH	R/$\overline{W}$	ASYNC	SYNC

Bit 31 -- INTATR

If set, this bit designates the information in the structure to contain control information in regards to an initiator mode request. Other bits in the command control field are examined and used as indicated by their initiator definitions.

Bit 30 -- TARGET

If the INTATR bit is not set and the TARGET bit is set then the Firmware operates in target mode. This bit designates the information in the structure to contain control information in regards to a target mode request as indicated by either the T_EXEC or T_WAIT bit in this field. The T_EXEC and T_WAIT bits are mutually exclusive. Other bits in the command control field are examined and used as indicated by their target definitions.

Bit 29 -- CONFIG

The configuration control field has the lowest priority of the three mode control bits which the Firmware examines. If the INTATR bit and the TARGET bit are cleared and the CONFIG bit is set then the Firmware operates in the configuration mode. This bit designates the information in the structure to contain control information in regards to a configuration request. The other config defined bits are mutually exclusive. Only one configuration mode request may be executed per command entry call. If this is the only configuration mode bit set in the command control field then no other parameters within the command structure are examined and the command request is handled as a NOP with GOOD status returned.

Bit 18 -- PAR

initiator mode

If set, parity is checked during data transfers. If a parity error is detected a parity error status is returned but no indication of the byte(s) in error is returned to the user.

Bit 17 -- FIRST

initiator mode

Setting this bit overrides the normal FIFO ordering of the device queue. The Firmware places this *siop_struc* at the head of the device queue to ensure it is the next command sent to the specified device.

This bit is important in situations where the user wishes to send an uninterrupted sequence of command requests to a specific device. An example of this would be a large **read**, where the number of blocks required by the user exceeds the block capacity of a 10-byte CDB. The single read must be broken into a sequence of separate but contiguous commands to the device. The first **read** command sent to the Firmware would not set the FIRST bit set; however, all subsequent command requests in this read sequence would have this bit set.

Another use of this bit is when sending a **request sense** command in response to a CHECK status received for some command to a device. The **request sense** command must be sent before any other commands to prevent the CHECK condition from being cleared before the user can find out the reason for the CHECK status.

Bit 16 -- DEVRST

initiator mode

If set, the command structure allocated to the Firmware contains a *bus device reset* message in the first byte of the message out field. To issue the *bus device reset* message to a SCSI device the Firmware selects with ATN asserted and when the target responds to selection and transitions to the message out phase, the Firmware responds to that phase by sending the message in the message-out field.

The user must initialize the command structure for this sequence. The user must install the SCSI address, LUN, message-out length, message-out pointer, and message byte. (The message-out pointer field can point to the message-out field of the structure if that is where the user installed the *bus device reset* message.) Additionally, the user must clear the NO_ATN bit, set the MOBUF bit, and set the DEVRST bit in the command control field. The DEVRST bit is used for error checking and is not tested except for unexpected disconnects.

In the case of an unexpected disconnect, if the DEVRST bit is set, the Firmware resolves the device reset condition. For a device reset condition all pending commands to the target LUN are returned. Internal synchronous data rate information is set to asynchronous. Internal tagged command queuing information is set to non-support.

Bit 15 -- MIBUF

initiator mode

If set, defines the message-in buffer pointer field to contain a pointer to a buffer where messages from the target are to be saved. If clear, the messages are saved in the 12-byte message-in field and the message-in buffer pointer field is initialized by the Firmware. It is recommended to NOT set the MIBUF in the command control field for normal operation; this feature of the Firmware is intended to be used for very long extended messages.

Bit 14 -- MOBUF

initiator mode

If set, defines the message-out buffer pointer field to contain a pointer to a user initialized buffer where messages to the target are located. The user also must initialize the message-out length field to the byte count of the message transfer. The user is responsible for ALL messages to be sent to the target. The first byte in the buffer must be an *identify* message. The external message buffer cannot contain *synchronous data transfer request* or *wide data transfer request* messages. Synchronous data transfer negotiations and wide data transfer negotiations must be handled by the Firmware. Additionally, if tagged command queuing is enabled then this bit should not be set because if the target disconnects and then reselects, the Firmware has no way of rethreading for this nexus. If clear, the Firmware initializes the message-out buffer pointer field and the message-out length field. Also the Firmware generates all necessary message bytes such as *identify* (if the NO_ATN bit is cleared), *queue tag* (if tagged queuing has been enabled in the Firmware), *synchronous data transfer request* (if the SYNC or ASYNC bit is set), and *wide data transfer request* (if the WIDE16 bit is set). It is recommended to NOT set the MOBUF in the command control field for normal operation; this feature of the Firmware is intended to be used for very long extended messages.

Bit 13 -- NO_ATN

initiator mode

If set, Firmware selects the target device **without** ATN asserted. This bit is set if the user does not wish for the target device to disconnect during this command. This bit cannot be used with SCSI linked commands or with devices that have tagged command queuing enabled. For SCSI bus efficiency, it is recommended to NOT set this bit.

Bit 10 -- SIOPADD

config mode

If set, the Firmware sets the SCSI address at which the SIOP resides. The user designates the address in the device address field of the command structure.

Bit 9 -- SIOPINT

config mode

If set, the Firmware installs the interrupt level at which the SIOP Firmware interrupts the MVME167/MVME187 MPU. The user designates the interrupt level in the SIOP interrupt level field of the command structure.

Bit 8 -- SCSIRST

config mode

If set, the Firmware causes the SCSI bus to be reset and terminated status is returned for all pending commands. For all LUNs, internal synchronous data rate information is set to asynchronous and internal tagged command queuing information is set to non-support.

Bit 7 -- TAG_Q

config mode

If set, the user is informing the initiator mode Firmware that the drive defined by the device address field and the LUN field has been enabled or disabled for SCSI tagged command queuing. If enabled the user specifies with a non-zero value the maximum number of commands the target LUN can queue in the queue depth field of the command structure. The Firmware handles the message exchanges that are required for tagged command queuing. The *simple queue tag* message is the tagged queuing message the Firmware uses. If the FIRST bit is set then the Firmware uses the *head of queue tag* message. The initiator mode Firmware keeps track of the tags and makes status available to the user as soon as commands are completed by the target. The user must insure that the target LUN is in the tagged command queuing mode before sending this configuration request to the Firmware.

If tagged command queuing has been enabled and the user disables tagged command queuing then the user must notify the Firmware. This notification is done by placing a value of zero (0) in the queue depth field for a TAG_Q configuration mode request. However, all outstanding tagged commands must be completed before disabling tagged command queuing.

Bit 6 -- LINK

initiator mode

If set, command in the CDB has the link bit set. The link field in the command structure must point to the next command structure to be used. This bit must be set for linked commands so the Firmware does not flag a linked command operation as a protocol violation.

Bit 4 -- S/G

initiator mode

If set, scatter/gather is used for data transfers. The user must supply a scatter/gather list. The pointer to the scatter/gather list is in the Data Pointer field of the command structure. The scatter/gather list must be aligned to a four-byte boundary.

Bit 3 -- D_PH

initiator mode

If set, the command contained in the CDB causes a data phase. For example, *test unit ready* does not have a data phase associated with it; this bit should be cleared. A *read* command has a data phase associated with it; this bit should be set. A *format unit* command may or may not have a data phase associated with it. The bit should be set if a defect list is to be included in the command. If no defect list or defect list header is to be sent then this bit should be cleared.

Bit 2 -- R/$\overline{W}$

initiator mode

If set, data direction is in to the initiator (read). If clear, data direction is out from the initiator (write). This bit is ignored if the D_PH bit is not set.

Bit 1 -- ASYNC

initiator mode

If set, the Firmware initiates negotiations for asynchronous data transfers. Normally, this bit is only set to place the SCSI device in a state that is compatible with the state of the Firmware after a call to the initialization module. If a synchronous data transfer rate has not been established then the user should clear this bit to avoid unnecessary negotiations.

Bit 0 -- SYNC

initiator mode

If set, the Firmware initiates negotiations for synchronous data transfers. The SIOP initiates negotiation for the maximum offset it supports and the fastest period possible. If a synchronous data transfer rate has been established then the user should clear this bit to avoid unnecessary negotiations.

ASYNC Bit	SYNC Bit	Firmware Action
0	0	does not initiate negotiations
0	1	initiates negotiations for synchronous data transfers
1	0	initiates negotiations for asynchronous data transfers
1	1	initiates negotiations for synchronous data transfers

Device Address or SIOP Interrupt Level

initiator mode

The user initializes the device address field with the SCSI address of the device to be accessed. For example, a command for SCSI address 4, LUN 2 would need a 0x00000004 placed in this field.

config mode

TAG_Q bit set-
The user initializes the device address field with the SCSI address of the target device that supports tagged command queuing. For example, SCSI address 6, LUN 1 would need a 0x00000006 placed in this field.

SCSIADD bit set-
The user initializes this field with the SCSI address that the MVME167/MVME187 SCSI port (SIOP) is to occupy. For example, if the user wants the SIOP to occupy address 4 then 0x00000004 would be placed in this field. This field is not checked for erroneous values. Instead, it is interpreted as a modulo 8 value; therefore, a hex value of 0xf9 is read as 0x01.

SCSIINT bit set-
The user initializes this field to the level (1 through 7) at which the SIOP is to interrupt the MPU. For polled mode (Firmware does not relinquish processor until command is complete) this field is cleared (0x00). This field is not checked for erroneous values. Instead, it is interpreted as a modulo 8 value; therefore, a hex value of 0x146578 is read as 0x00.

LUN

initiator mode

The user initializes the LUN field with the Logical Unit Number (LUN) of the target device to be accessed. For example, a command for SCSI address 4, LUN 2 would need a 0x00000002 placed in this field so the Firmware can generate the correct *identify* message needed for selection.

config mode

TAG_Q bit set-
The user initializes the LUN field with the logical unit number of the target device that supports tagged command queuing. For example, SCSI address 0, LUN 1 would need a 0x00000001 placed in this field.

CDB Length or Queue Depth

initiator mode

The CDB length field contains the length of the CDB, in bytes, that the user wants to be transferred during the command phase. CDB lengths defined by the SCSI specification are 6, 10, and 12 bytes.

config mode

TAG_Q bit set-
The user initializes this field with the maximum number of SCSI commands the target LUN can queue at a time. A value up to 0xff means the initiator mode Firmware dispatches up to that many commands to the target device without waiting for SCSI status to be returned for any of the outstanding commands. Any value over 0xff is not defined by the current SCSI specification and a Firmware error is generated. The Firmware interprets a value of zero (0) to mean untagged queuing and the Firmware control fields are set to reflect non-support of tagged command queuing. Untagged queuing is implicitly supported by the Firmware; therefore, a value of zero (0) should only be used to disable tagged command queuing after it has been enabled.

CDB

initiator mode

The CDB field contains the CDB that is to be transferred during the command phase on the SCSI bus for this command structure.

Message-In Length

This field may be modified by the Firmware.

initiator mode

The value in this field is used for buffer over-run control. The value in this field is decremented as the returned message bytes are placed in the Message-In buffer. If the MIBUF bit is not set then this value is initialized to 12 and the Message-In Pointer field is initialized to point to the Message-In Bytes buffer within this command structure. The Firmware receives all messages from the target. The only messages placed in the buffer (returned to the user), however, are SCSI extended messages.

Message-In Buffer Pointer

This field may be modified by the Firmware.

initiator mode

If the MIBUF bit is set, the message-in buffer pointer field contains the starting address where a message from the target is to be stored. It is recommended to not set the MIBUF in the command control field for normal operation; this feature of the Firmware is intended to be used for very long extended messages. If the MIBUF bit is cleared, this field is initialized by the Firmware. This field is updated by the Firmware as (extended message) message bytes are received.

Message-In Bytes (0-B)

This field may be modified by the Firmware.

initiator mode

The message-in bytes are where (extended message) message bytes from a target are stored if the MIBUF bit in the command control field is cleared.

Message-Out Length

This field may be modified by the Firmware.

initiator mode

If the MOBUF bit is set, this field contains the length of the message, in bytes, the initiator sends during a message-out phase. Usually, the user does not need to supply any messages to send to the target.

Message-Out Buffer Pointer

This field may be modified by the Firmware.

initiator mode

If the MOBUF bit is set, the message-out pointer field contains the address where a message to the target is placed. Normally, the user does not need to supply any messages to send to the target. It is recommended to not set the MOBUF bit in the command control field for normal operation; this feature of the Firmware is intended to be used for very long extended messages. If the MOBUF bit is cleared, this field is initialized by the Firmware.

Message-Out Bytes (0-B)

This field may be modified by the Firmware.

initiator mode

If the MOBUF bit cleared, the message-out bytes are where messages to a target are placed.

Data Count

This field may be modified by the Firmware.

initiator mode

The data count field contains the number of bytes to be transferred during a data phase on the SCSI bus. If no data phase is to be executed then this field must be set to zero (0x00000000). **This value must not exceed 24 bits (0xFFFFFF or 16,777,215).** If the S/G bit is set in the command control field then this field is ignored.

Data Pointer or Scatter/Gather List Pointer

This field may be modified by the Firmware.

initiator mode

The data pointer field contains a pointer to a buffer where data is transferred to/from. If the S/G bit is set in the command control field then this field contains a pointer to a scatter/gather list. **The scatter/gather list must be aligned on a four-byte boundary (longword boundary for the CISC MVME167; word boundary for the RISC MVME187).**

Link Pointer

initiator mode

The link pointer field is initialized by the user for linked commands. This field points to the command structure which contains the linked CDB. The LINK bit in the command control field in this structure must be set and the link bit in the CDB must be set.

Status Return Function Pointer

initiator mode

This field contains the starting address of a function that processes a returned command structure. The Firmware calls this routine as a "C" function and passes the address of the finished command structure to it.

```
void (* ret_stat)();     /* define ret_stat as a pointer to a function */


ret_stat = siop_struc->ret_stat;     /* init ret_stat with address of rtn */
ret_stat(&siop_struc);     /* call routine with command structure as input */
```

Status

initiator mode

The status field contains status concerning the target device, Firmware, and SCSI bus. The least significant byte in this field contains the status received from the target during the status phase of the SCSI command. The second least significant byte in this field contains an error code generated by the Firmware if a Firmware detected error occurred. This field is initially cleared by the Firmware when the command structure is first allocated to the Firmware.

config mode

The least significant byte in this field is not used for a config mode request. The second least significant byte in this field contains an error code generated by the Firmware if a Firmware detected error occurred. This field is initially cleared by the Firmware when the command structure is first allocated to the Firmware.

Termination Transfer Byte Count

initiator mode

The termination transfer count field contains the number of bytes successfully transferred during a data phase on the SCSI bus. The user should compare this value with the expected count to verify that the correct number of bytes were transferred. This field is initially cleared by the Firmware when the command structure is first allocated to the Firmware.

Error Address

initiator mode

This is the address where an SIOP bus fault or illegal SIOP instruction error was encountered.

SCSI Queue Tag

initiator mode

This field is used by the Firmware to track queued commands to peripherals that support SCSI tagged command queuing. Upon command completion this field contains the queue tag used by the Firmware.

Work Area

initiator mode

The work area fields are used by the Firmware to save, change, or amend device and command dependent parameters and statuses.

Scatter/Gather List

The scatter/gather list is initialized by the user and allocated to the Firmware with the command structure. The Firmware may modify the scatter/gather list in the case of a disconnect. Below is an example of a scatter/gather list with two entries. For this example list, the user would place 0x00004000 in the Scatter/Gather List Pointer (Data Pointer) field of the command structure and set the S/G bit in the command control field. **A scatter/gather list must be aligned on a four-byte boundary (longword boundary for the CISC MVME167; word boundary for the RISC MVME187).**

Table B-3. Example Scatter/Gather List

Address	Parameter
0x4000	24 bit Byte Count
0x4004	Buffer Pointer
0x4008	24 bit Byte Count
0x400C	Buffer Pointer
0x4010	Logical End (32 bit value = 0)
0x4014	don't care

Byte Count

Byte count is the number of bytes to transfer for this entry of the scatter/gather list. The value in this field must not exceed 0xFFFFFF (16,777,215).

Buffer Pointer

This is a pointer to the area where the data is to be transferred.

Logical End

The last entry in a scatter/gather list is the logical end of list. This entry must be all zeroes.

siop_init (Firmware Initialization Structure)

This is the data structure passed to the `siop_init()` entry point.

```
typedef struct {

  UINT  signature    /* MASK_INIT_1 | SCSI address of the SIOP */    0x00
  UINT  intr_lev;    /* level at which SIOP interrupt should occur */0x04
  UINT  sclk_speed;  /* frequency of the SIOP clock (in Hz) */       0x08
  UINT  snoopmode;   /* snoop mode of processor */                   0x0c
  RELC *reloctbl;    /* pointer to scripts relocation table */       0x10
  UINT *script_ptr;  /* pointer to script pointer array */           0x14

} INIT_TYPE_1;
```

signature

The first 32-bit element represents a *signature* for the entire structure. This allows firmware data structure definitions to be changed without breaking existing source or binaries. In the case of the structure used for initialization, the SCSI host address is encoded in the LSB of the signature.

The signature for an `INIT_TYPE_1` structure is a bitwise-or of the preprocessor define `MASK_INIT_1` ('IT1\0') and the SCSI address of the SIOP.

intr_lev

The user initializes this field to the level (1-7) at which the SIOP is to interrupt the MPU. For polled mode, in which the firmware does not relinquish the processor until the command is complete, this field should be cleared. This field is not checked for erroneous values. It is used directly to program the PCC.

sclk_speed

This is the speed (in *Hz*) of the SCSI core clock feeding the SIOP. In the *NCR 53C710 SIOP Data Manual*, this clock is referred to as SCLK and it is used to derive important timing information. The SCLK rate being fed into the SIOP varies depending upon the system into which the SIOP has been designed.

The following table lists the SCLK rate on Motorola VMEmodules:

Table B-4. SIOP Clock Rates for VMEmodules

VMEmodule	MPU Clock	SIOP Clock	
	MHz	MHz	Hz
167	25	50	50000000 (0x02FAF080)
187			
167	33	66	66666666 (0x03F940AA)
187			
197	50	50	50000000 (0x02FAF080)

snoopmode

The SIOP has snoop control output lines which are compatible with the snoop control inputs of the MC68040 and MC88110. The user initializes this field to the snoop control mode in which the SIOP is to operate. Recommended settings for the SBCs are indicated with a "*" in the table below.

Table B-5. Snoop Control Modes

Bit 1	Bit 0	Requested Snoop Operation		MVME162/ 166/167/197	MVME187
		Read Access	Write Access		
0	0	Inhibit Snooping	Inhibit Snooping		*
0	1	Supply Dirty Data and Leave Dirty	Sink Byte/Word/Long Data	*	
1	0	Supply Dirty Data and Mark Invalid	Invalidate Data		
1	1	Reserved (Snoop Inhibited)	Reserved (Snoop Inhibited)		

Note **In the table above, * = recommended setting.**

reloctbl

The user initializes this field to the globally defined value of the relocation table (`extern RELC relocation[]`) which is created by the preprocessor to the NCR SCRIPTS compiler.

script_ptr

The user initializes this field to the globally defined value of the SCSI Script pointer array (`extern unsigned script_ptr[]`) which is created by the preprocessor to the NCR SCRIPTS compiler.

Initialization Structure (depreciated version)

The INITPARS structure is archaic. It is defined here in order to support compatibility with existing code. Please switch to using its replacement, the INIT_TYPE_1 structure.

```
typedef struct {

 UINT  scsi_addr;    /* SCSI address of the SIOP */                    0x00
 UINT  intr_lev;     /* Level at which SIOP interrupt will occur */    0x04
 UINT  clk_speed;    /* ASCII representation of MPU clock (x10KHz) */  0x08
 UINT  snoopmode;    /* Snoop mode of processor */                     0x0c
 RELC *reloctbl;     /* Pointer to scripts relocation table */         0x10
 UINT *script_ptr;   /* Pointer to script pointer array */             0x14

} INIT_TYPE_0, INITPARS;
```

SCSI Address

The user initializes this field with the SCSI address that the MVME167 or MVME187 SCSI port (SIOP) is to occupy. For example, if the user wants the SIOP to occupy address 4 then 0x00000004 would be placed in this field.

In the case of this structure, the signature is defined as 0. This definition, combined with the assumption that existing source and binaries followed recommended practice when initializing the scsi_addr parameter, allows the initialization code to know what it is looking at without breaking existing applications.

Interrupt Level

Same definition as in the INIT_TYPE_1 structure.

Clock Speed

The user initializes this field to an ASCII value which is equal to the clock speed of the MPU as expressed as a multiple of 10KHz. For example, if the clock speed of the MPU was 25MHz then this field should be initialized to '2500' (0x32353030).

Snoop Mode

Same definition as in the INIT_TYPE_1 structure.

Pointer to Relocation Table

Same definition as in the INIT_TYPE_1 structure.

Pointer to Script Pointer Array

Same definition as in the `INIT_TYPE_1` structure.

sdt_tinit (Debug Logging Initialization Structure)

Some of the initialization values which are used by the debug logging initialization routine, **sdt_tinit**, are contained in a structure. A pointer to this structure is passed into the initialization routine. The **fill** bytes in the structure are unused at the current time and it is recommended to clear these bytes.

Table B-6. Debug Logging Initialization Values Structure

Byte offset (hex)	Parameter			
00	fill	Flag	fill	fill
04	Beginning Address			
08	Ending Address			

Flag

This field is examined for a value of 0x44 (ASCII 'D') to determine if debug logging is to be enabled. If this field is not 0x44 then an internal debug logging flag is set to disabled and debug logging initialization is terminated.

Beginning Address

If debug logging is enabled, then this is the beginning address of a block of memory that has been allocated for debug logging. If this address is greater than the Ending Address, then debug logging is disabled. If the size of the block of memory is less than the size of one debug log entry, then debug logging is disabled.

Ending Address

If debug logging is enabled, then this is the ending address of a block of memory that has been allocated for debug logging. If this address is smaller than the Beginning Address, then debug logging is disabled. If the size of the block of memory is less than the size of one debug log entry, then debug logging is disabled.

EXTERNAL ROUTINES C

Introduction

The SCSI Software requires services to be provided external to the SCSI Software itself. These services must be provided by the specific application.

There are two global services which must be provided:

splhi A routine used to mask interrupts from the NCR 53C710 SCSI chip. This may be a null routine if the specific application of the SCSI Software uses the software in polled mode (i.e., at interrupt level 0).

splx A companion routine to **splhi**, used to restore interrupts to the level existing before the corresponding **splhi**. This may be a null routine if the specific application of the SCSI Software uses the software in polled mode (i.e., at interrupt level 0).

Also, a command-specific service must be provided:

ret_stat A routine called by the Firmware to notify the user that final status is available for a command request. A pointer to this routine is supplied in each command structure (*siop_struc*) issued to the Firmware.

Firmware for the MVME197 also requires routines to enforce serialized access to I/O registers:

serialize_memory_access
deserialize_memory_access

When compiling to run on that board (refer to the file **incl/sfw_cnfg.h**), the user-supplied routines set/clear the SRM (serialize memory) bit in the 88110 PSR (processor status register). If one compiled object may be executed on both an 88100 and an 88110, then those routines should verify that the processor is an 88110.

All of the above routines are described further on the following pages.

C

splhi

NAME

splhi —Mask interrupts

SYNOPSIS

```
UINT splhi()
```

DESCRIPTION

The **splhi** routine masks interrupts from the NCR 53C710 SCSI chip. This routine is normally used in conjunction with **splx**.

Example:

```
level = splhi();  /* mask interrupts */

/* code that cannot be interrupted goes here */

splx(level);  /* go to previous priority level */
```

RETURN VALUE

splhi is always successful and returns the previous interrupt priority level. This value is hardware specific and it is not guaranteed to be in the range 0 through 7.

ERROR CONDITIONS

none

splx

NAME

splx —Restore Interrupt Priority Level

SYNOPSIS

```
UINT splx(level)
UINT level;              /* previous Interrupt Priority Level */
```

DESCRIPTION

The **splx** routine restores the interrupt priority level following a call to **splhi**. This function is normally used in conjunction with **splhi**. *level* is hardware specific and is not guaranteed to be in the range 0 through 7.

Example:

```
level = splhi();  /* mask interrupts */

/* code that cannot be interrupted goes here */

splx(level);  /* go to previous priority level */
```

RETURN VALUE

none

ERROR CONDITIONS

none

ret_stat

NAME

ret_stat —Notify User of Command Completion

SYNOPSIS

```
void ret_stat(cmd)
struct siop_struc *cmd;         /* completed command structure */
```

DESCRIPTION

The **ret_stat** routine is user-supplied code which executes when a command request to the Firmware is completed. A pointer to this routine **must** be installed in the *status_ptr* field of each *siop_struc* used by the Firmware. This routine should check the *status* field of the returned *siop_struc* to determine if any errors occurred.

The firmware calls this routine as a "C" function and passes the address of the finished command structure to it. For example:

```
void (* ret_stat)();     /* define ret_stat */

ret_stat = siop_struc->ret_stat; /* init ret_stat */

ret_stat(&siop_struc);     /* call routine */
```

RETURN VALUE

none

ERROR CONDITIONS

none

NOTES

The user **cannot** lower the interrupt mask while in this routine.

The Firmware has not completed command clean up when this routine is invoked; therefore, the user is required to exit this routine as a normal function call to return the processor to the Firmware.

Since the Firmware calls this routine by address reference, this routine can be any name. A different **ret_stat** routine may be used for each command request.

The user may call **siop_cmd()** while in this routine.

(de)serialize_memory_access

NAME

serialize_memory_access — Enforce serialized access to I/O memory

deserialize_memory_access — Allow out-of-order access to I/O memory

SYNOPSIS

```
void serialize_memory_access()
void deserialize_memory_access()
```

DESCRIPTION

Many processors execute and complete load and store instructions in the order in which they are encountered. For instance, if a program has a store instruction followed by a load instruction, the processor will execute the store instruction, and then the load instruction. Some processors, for efficiency reasons, overlap instruction executions and do not necessarily finish instructions in the same order as they begin them. Consider a possible execution of the previous example on such a processor: the store will be issued, then the load will be issued, but the load instruction may complete before the store instruction does.

Normally, this is not a problem because the processor will produce results in memory that are the same as if all loads and stores were executed in program order. Note, however, that this is because the memory system functions correctly regardless of the order of loads and stores. That is, a store to address A won't change the results of a load from address B, for distinct addresses A and B. However, some hardware devices and their drivers require that the loads and stores be visible to the device in a specific order. In such a case, a processor must perform the instructions in a serialized fashion to guarantee correct behavior.

When compiling to run on the MVME197 (refer to the file **incl/sfw_cnfg.h**), the user-supplied routines set/clear the `SRM` (serialize memory) bit in the 88110 `PSR` (processor status register). If one compiled object may be executed on both an 88100 and an 88110, then those routines should verify that the processor is an 88110.

The SCSI Firmware will call **serialize_memory_access** expecting that upon return, all accesses to I/O registers (i.e., the 53C710, VMEchip2, and the PCCchip2) will complete in order. It will match every call to enable serialization with a subsequent call to **deserialize_memory_access** before returning control to the Application.

RETURN VALUE

none

ERROR CONDITIONS

none

NOTES

No calls may be made to the SCSI Firmware from within these routines.

Your compilation tools or run-time environment may already supply routines similar to these. Or, they may provide C-language interfaces to modify processor registers, eliminating the need for you to code these routines in assembly language.

Status Field

Command status returned in the status field of *siop_struc* is subdivided into the following structure as extracted from the header file *ncr.h*.

```
/*
 *    command structure STATUS field structure definition
 */
typedef struct siop_stat {
   UCHAR sense_key;        /* not used (MSB) */
   UCHAR sdl_key;          /* not used */
   UCHAR siop_key;         /* non SCSI status from SIOP */
   UCHAR status_key;       /* SCSI status byte from SIOP (LSB) */
} SIOP_STAT;
```

All bytes of the status field contained in *siop_struc* are cleared by the Firmware for initiator and configuration mode commands upon *siop_cmd()* entry. For target mode commands, only the fields *sense_key*, *sdl_key*, and *siop_key* are cleared upon *siop_cmd()* entry so that the Firmware target mode application can return a SCSI bus status to the initiator in the *status_key* field.

The status field is not valid until the Firmware calls the return status routine supplied by the user in the *siop_struc*. Only the *status_key* and *siop_key* fields are set by the Firmware. The remaining fields, *sense_key* and *sdl_key*, are not accessed by the Firmware but are reserved for use by the user.

status_key Error Codes

The first field, *status_key*, is returned only for initiator mode commands. This is the status returned by the target device during the SCSI bus status phase. Returned values for this field and recommended responses follow.

SS_GOOD (0x00)

This is the SCSI status of GOOD. This status indicates that the target has successfully completed the command. No corrective action is necessary.

SS_CHECK (0x02)

This is the SCSI status of CHECK CONDITION. This status indicates that the target has preserved sense data for the initiator. The user should send a SCSI **request sense** command to the target to get the sense data from the target.

SS_CM_GOOD (0x04)

This is the SCSI status of CONDITION MET. This status is returned whenever certain SCSI operations, such as **search data**, are satisfied. This should be interpreted as a good status by the user with no corrective action necessary.

SS_BUSY (0x08)

This is the SCSI status of BUSY. This status indicates that the target is busy. This status is returned whenever a target is unable to accept a command from an otherwise acceptable initiator. The user may resend this command to the Firmware. The user should track the number of times this command has been resent and error out if a specified retry count limit has been exceeded.

SS_I_GOOD (0x10)

This is the SCSI status of INTERMEDIATE. This status, or INTERMEDIATE-CONDITION MET, is returned for successfully completed commands in a series of SCSI linked commands (except the last command).

When the Firmware returns the final status on a series of SCSI linked commands, the user should walk the linked list of *siop_struc* structures and verify that the *status_key* of each (except the last) is either SI_GOOD or SS_I_CM_GOOD. Each of these should be interpreted as good status by the user with no corrective action necessary.

SS_I_CM_GOOD (0x14)

This is the SCSI status of INTERMEDIATE-CONDITION MET. This status is the combination of the CONDITION MET and INTERMEDIATE statuses.

When the Firmware returns the final status on a series of SCSI linked commands, the user should walk the linked list of *siop_struc* structures and verify that the *status_key* of each (except the last) is either SI_GOOD or SS_I_CM_GOOD. Each of these should be interpreted as good status by the user with no corrective action necessary.

SS_RSVCON (0x18)

This is the SCSI status of RESERVATION CONFLICT. This status is returned whenever an initiator attempts to access a logical unit that is reserved exclusively for another SCSI initiator.

This error indicates that in a multiple initiator system another initiator has locked a target device. The simplest recovery attempt is to retry this command at a later time in hopes that the device has since become released. Recovery beyond this measure would require communication with the other initiator through some user defined protocol.

SS_CMDTERM (0x22)

This is the SCSI status of COMMAND TERMINATED. This status is returned whenever the target terminates the current I/O process after receiving a **terminate I/O process** message. This is the expected status posted to the user in response to initiating this message.

SS_QFULL (0x28)

This is the SCSI status of QUEUE FULL. This status is implemented if tagged queuing is implemented. This status is returned when a **simple queue tag**, **ordered queue tag**, or **head of queue tag** message is received and the command queue on the target device is full.

This status indicates that the Firmware is configured for a Q_DEPTH greater than that supported by the target device. The user may simply wait until outstanding commands to the Firmware are returned before resending this command; however, a more robust solution would be to reconfigure the target device Q_DEPTH to a smaller value using the Firmware *CONFIG* command.

siop_key Error Codes

For the field *siop_key*, the following error codes are listed along with an explanation of their meaning and recommended response. Some of these errors are also returned for an *siop_init()* call, in which case, they are located in the LSB of the returned value.

SI_GOOD (0x00)

Current command has completed successfully. No corrective action is necessary.

SI_NOP (0x01)

An *siop_struc* was sent with no command mode bit (i.e., *INTATR*, *TARG*, or *CONFIG*) set in the *cmd_ctrl* word. The user may have intentionally issued this command, in which case the returned status indicates that the desired no operation action has occurred. Otherwise, the user has issued an *siop_struc* in error and must set one of the command mode bits.

SI_SCSIRST (0x02)

The Firmware has either detected or issued a SCSI bus reset. All commands currently outstanding to the Firmware at the time of a SCSI bus reset are terminated with this status. All commands which are terminated in a *SI_SCSIRST* may be reissued by the user. **The Firmware status information for all target devices is set to asynchronous data transfers and tagged command queuing support is disabled.**

SI_DEVRST (0x03)

This status is returned only for target mode commands outstanding to the Firmware. It indicates that a SCSI **device reset** message has been received by the Firmware in target mode. The target application task should reset itself in accordance to its configured reset mode and ready itself to receive more target commands.

SI_ABRT (0x04)

This status is returned only for target mode commands outstanding to the Firmware. It indicates that a SCSI **abort** message has been received by the Firmware in target mode for the specified logical unit. The target task should abort the current I/O and all queued I/O.

SI_ABRTTAG (0x05)

This status is returned only for target mode commands outstanding to the Firmware. It indicates that a SCSI **abort tag** message has been received by the Firmware in target mode for the specified logical unit. If the target task supports tagged queuing, it should abort only the current I/O.

SI_CLEARQ (0x06)

This status is returned only for target mode commands outstanding to the Firmware. It indicates that a SCSI **clear queue** message has been received by the Firmware in target mode for the specified logical unit. If the target task supports tagged queuing, it should abort the current I/O and all queued I/O.

SI_DATAOV (0x07)

This status is received for initiator mode commands to indicate a data overflow to the target in a DATA-OUT phase or to the SIOP in DATA-IN phase. A data overflow is defined to occur if the user supplied byte count is exhausted yet the target device remains in the DATA-IN or DATA-OUT phase. For the DATA-IN phase the SIOP reads the remaining data from the target device into a bit bucket before terminating the command. For the DATA-OUT phase the SIOP writes the data byte 0x0E to the target device for all requested bytes until the target terminates the DATA-IN phase.

A possible source of this error at the *siop_cmd()* level may be that the byte count specified for the Firmware is less than the byte count determined by the target.

SI_DATAUR (0x08)

This status is received for initiator mode commands to indicate a data underrun to the target in a DATA-OUT phase or to the SIOP in DATA-IN phase. A data underrun is defined as a condition where the target terminates the current data transfer yet the user supplied byte count has not been exhausted.

A possible source of this error at the *siop_cmd()* level may be that the byte count specified for the Firmware is greater than the byte count determined by the target. Additionally, this error is returned if the user expects to go to a data phase and the target returns a CHECK or BUSY status without going to the data phase.

SI_CLK2FAST (0x09)

This error is returned during Firmware initialization. The initialization parameter, *initval.clk_speed*, contains an ASCII numeric value greater than 7500 (**0x37353030**) which represents a clock speed greater than 75 MHz.

SI_BADCLKPAR (0x0A)

This error is returned during Firmware initialization. The initialization parameter, *initval.clk_speed*, contains a character that is not a valid ASCII equivalent of the decimal values 0 through 9 (**0x30** through **0x39**). The user must provide only decimal values in ASCII format for the *initval.clk_speed* parameter.

SI_BADQDEPTH (0x0B)

This status is returned if the Firmware *CONFIG* command attempts to set the queue depth to a value greater than 255 (0xFF).

The user must change the *cdb_lgth* of the *siop_struc* to a queue depth value 255 (0xFF) and reissue the Firmware *CONFIG* command to install this new queue depth value.

SI_SELTO (0x0C)

This status is returned when the Firmware attempts to select a specified target address and a target does not respond to the selection within 250 ms. Normally, this status indicates that there is no target device at the specified address.

SI_RESELTO (0x0D)

This status is returned when the Firmware, as a target, attempts to reselect an initiator and the initiator does not respond to the reselection within 250 ms. The user may wish to attempt reselection again up to some maximum retry limit to see if the initiator responds.

SI_BERR (0x0E)

This status if returned if a bus error occurs when the SIOP is attempting to access the local bus while it is physically threaded to a SCSI device during a SCSI bus DATA PHASE. The Firmware gets the target device off the SCSI bus by dumping the remaining data to a bit bucket for DATA-IN phase or writing an error data pattern (0x0E) for DATA-OUT phase. The *err_addr* field of the *siop_struc* points to the fault address.

SI_BERRCMD (0x0F)

This status if returned if a bus error occurs when the SIOP is attempting to access the local bus while it is physically threaded to a SCSI device in any phase other than a SCSI bus DATA PHASE (i.e., BERR while sending a CDB out to a target). The Firmware gets the target device off the SCSI bus before returning this error code. The *err_addr* field of the *siop_struc* points to the fault address.

SI_ILGLINST (0x10)

This interrupt is generated when the SIOP attempts to execute an illegal SCRIPTS instruction. Several different situations can produce an illegal SCRIPTS instruction; however, each is fatal and no attempt to retry the command should be made. The Firmware gets the current target off the SCSI bus before terminating the command and returning status to the user. The *err_addr* field of the *siop_struc* points to the address where the illegal instruction is located.

Some of the sources of an illegal instruction are:

1. The NCR SCRIPTS compiler generated the wrong opcode for a SCRIPTS instruction forcing the SIOP to execute an illegal opcode.
2. The memory where the SCRIPTS reside has been overwritten or otherwise corrupted resulting in the SIOP executing an illegal opcode.
3. The SIOP attempts to execute a SCRIPTS instruction that is non-longword (four-byte) aligned. All SCRIPTS must be aligned to byte boundaries that are integer multiples of 4.

SI_UDC (0x11)

This error code is returned when a target unexpectedly transitions to the bus free state. This condition may be caused by a problem with the target or on the SCSI bus itself.

The user should retry this command at least once in an attempt to recover from some random or transient problem with the target.

SI_UPC (0x12)

This error is returned when the Firmware detected an unexpected phase change in a target device to which it was physically threaded. This condition may be caused by a problem on the device or with the SCSI bus itself. The Firmware gets the target off the SCSI bus before returning this error code.

The user should retry this command at least once in an attempt to recover from some random or transient problem with the target.

SI_BUSHUNG (0x13)

This status is returned when the SIOP has waited more than 250 ms for some SCSI activity from another SCSI device to which it is physically threaded. A SCSI bus reset is the only recovery action which reliably restores the SCSI bus to a usable state.

Protocol Violation Errors (SI_PVE01 - SI_PVE0A)

The following class of errors are termed protocol violation errors because each indicates a violation of the SCSI bus protocol. Generally, these violations are illegal phase transitions from a given phase. The Firmware gets the device off the bus in each instance of a protocol violation before returning the *siop_struc* to the user. Each instance of the protocol violations is a follows:

SI_PVE01 (0x14)

This protocol violation error is returned when a physically threaded target device transitions to a data phase (either DATA-IN or DATA-OUT) when the D_PH bit of the *cmd_ctrl* word in the *siop_struc* was not set.

SI_PVE02 (0x15)

This protocol violation error is returned when a physically threaded target device transitioned to a specific data phase (either DATA-IN or DATA-OUT) when the R_W bit of the *cmd_ctrl* word in the *siop_struc* indicated the data was to be moved in the other direction.

SI_PVE03 (0x16)

This protocol violation error is returned when a physically threaded target device transitions to the incorrect phase following selection. In the case of select without ATN the COMMAND phase did not follow the SELECTION phase. In the case of select with ATN the MSG-OUT (identify message) did not follow SELECTION.

SI_PVE04 (0x17)

This protocol violation error is returned when a physically threaded target device does not transition to one of the following phases from the MSG-OUT phase: CMD, STATUS, DATA-IN, DATA-OUT, MSG-IN, MSG-OUT. Transitions to any other phase result in a protocol violation.

SI_PVE05 (0x18)

This protocol violation error is returned when a physically threaded target device transitions to an incorrect phase following either the DATA-IN or DATA-OUT phase. A phase other than MSG-IN, STATUS, or DATA results in a protocol violation.

SI_PVE06 (0x19)

This protocol violation error is returned when a physically threaded target device transitions to an incorrect phase following the COMMAND phase. The only valid phases allowed following the COMMAND phase are: STATUS, DATA, and MSG-IN. If the invalid phase is the COMMAND phase, the problem could be that the user set the value of the *cdb_lgth* in the *siop_struc* to a value less than the actual CDB length.

SI_PVE07 (0x1A)

This protocol violation error is returned when a physically threaded target device transitions to any phase other than MSG-IN following the STATUS phase.

SI_PVE09 (0x1C)

This protocol violation error is returned when a physically threaded target device follows a **save data pointers** message with a transition to a phase other than MSG-IN for the subsequent **disconnect** message.

SI_PVE0A (0x1D)

This protocol violation error is returned when a physically threaded target device transitions to any phase other than MSG-IN after reselection. The target must transition to the MSG-IN phase in order to transmit the required **identify** message.

SI_BADPATCH (0x1E)

This error is returned during Firmware initialization. The SCRIPTS program which patches the run-time SCRIPTS received an interrupt other than a SCRIPTS INT instruction interrupt. Possible error interrupt sources are bus fault, abort and illegal instruction. The invalid interrupt source read from the SIOP DSTAT register is not returned to the user.

SI_NOSCSIBUS (0x1F)

During Firmware initialization, the SCSI bus is monitored and was found to be in an illegal state. This illegal SCSI bus state could result from the SIOP being connected to a SCSI bus which is not correctly powered or terminated.

SI_BADPARAM (0x21)

Bad parameter supplied via entry point. Returned when `siop_init()` does not recognize the signature of the `initvals` structure. Verify that it is a valid structure for the firmware revision in use.

t167 TEST PROGRAM

E

Introduction

This appendix describes **t167**, a test program that can be used with the SBC SCSI Software. The test program is primarily used to send commands to the SIOP Firmware. The test program can also be used to send commands to the SBC SCSI Driver Library (SDL). The SDL is not described in any User's Manual. It is given out as unsupported source on the *Single Board Computers SCSI Software* Release Tape. Refer to the *SBC SCSI Software Release 1.1 Software Release Guide* (SBCSCSI/S1) for additional information on this release tape.

t167 and the SDL are for example purposes only. They are not maintained as supported products.

Overview

t167 is a menu-driven program that is used to interface to the SBC SCSI Software (SIOP Firmware and the SDL). Its architecture supports both Tools and Tests for the SIOP Firmware and the SDL. The Tools menu items allow commands to be sent to the SIOP Firmware or to the SDL. These commands are those defined by the SIOP Firmware and the SDL. The Tests menu items are not implemented, but they do show up in the menus.

t167 is designed to run in a stand-alone environment or as a task under VMEexec 2.0 or later. In the stand-alone mode, **t167** does not require any operating system. It uses only the services of the SBC ROM Debuggers for I/O to the console port. In this mode, **t167** takes over complete control of the SBC. When **t167** is used in the VMEexec environment, it runs as any other VMEexec task. A README file is provided along with the source to **t167**. This file explains how to run **t167** in each of these modes.

The remaining sections in this appendix provide detailed descriptions of the submenus and submenu selections which comprise **t167** from a user's perspective. Figure E-1 on the following page is a graphical representation of the submenus and functions which make up **t167**. Not all of the menu items are implemented.

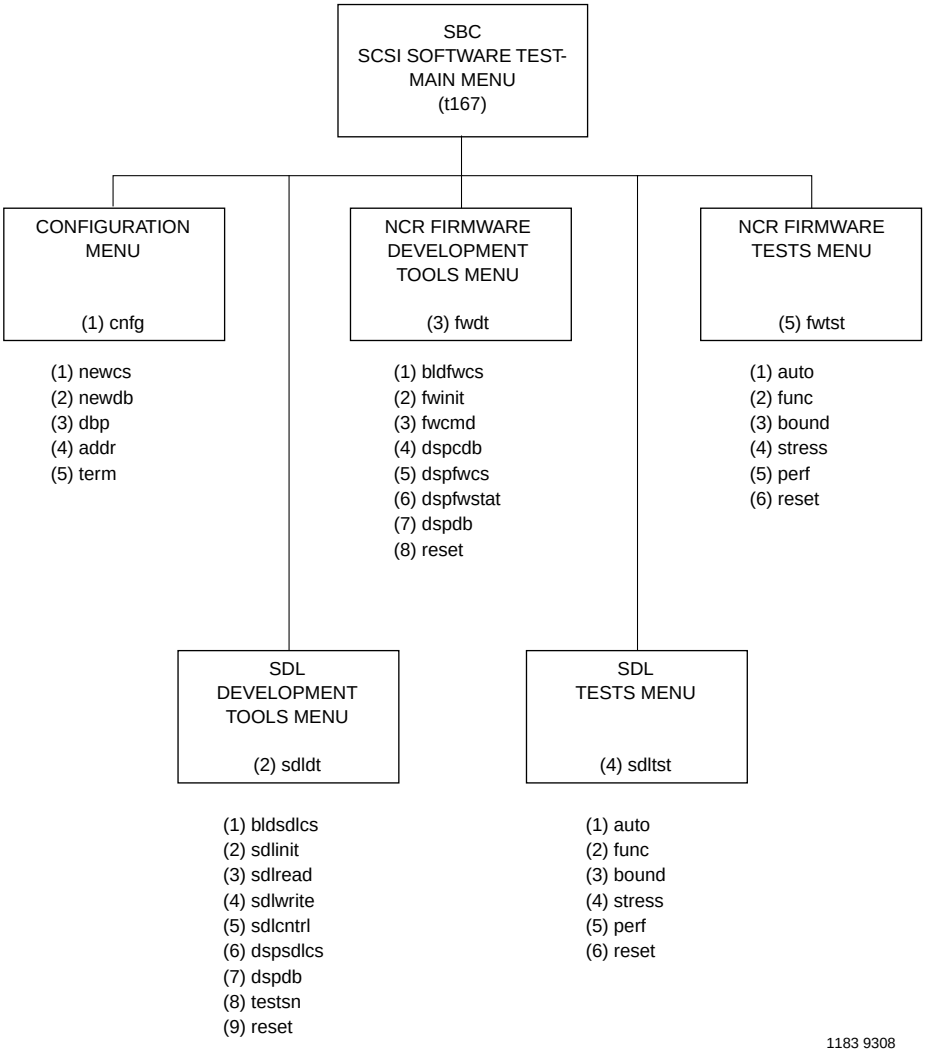

Figure E-1. t167 Submenus and Functions

Menu Item Descriptions

Main Menu

t167 contains five major menu items which can be selected. In addition to these, there are four items (*help*, *status*, *quit*, and *exit*) which are special selections found in the main menu and in all submenus:

Item	Name	Description
1	cnfg	t167 Configuration
2	sdldt	SCSI Driver Library Development Tools
3	fwdt	NCR Firmware Development Tools
4	sdltst	SCSI Driver Library Tests
5	fwtst	NCR Firmware Tests
h	help	Display this menu
s	status	Status of previous command
q	quit	Exit from t167
x	exit	Exit from t167

Note **All command functions described in the succeeding sections prompt a user for any required parameters. In some cases, however, a user may provide parameters as command line arguments when the function is selected. The descriptions of functions which accept command line arguments include a usage example. If command line arguments are provided, the function runs to completion without further interaction unless an error occurs.**

Each of these menu items is discussed briefly below. Following this, each is then described in detail.

t167 Configuration

The selections within this submenu permit a user to alter certain parameters used throughout **t167**. Although the design of **t167** is configured to useful default parameters at run-time, it may sometimes be useful to change one or more of the configurable values, particularly when doing development work.

SCSI Driver Library Development Tools

This submenu provides access to a number of functions which are useful when working with the SDL. From this menu, a user may build and issue SDL commands. Once a command has been executed by the SDL, other menu selections permit the user to inspect the results returned by the SDL.

NCR Firmware Development Tools

This submenu provides access to the SIOP Firmware. From this menu, a user may build and issue SIOP commands. Once a command has been executed by the SIOP Firmware, other menu selections permit the user to inspect the results returned by the SIOP Firmware.

SCSI Driver Library Tests

This menu item does not perform any useful action.

NCR Firmware Tests

This menu item does not perform any useful action.

help

This selection simply causes the current menu to be redisplayed.

status

The command line for the previous command is displayed, along with the status returned when that command was completed.

quit

From a submenu, this selection returns control to the next higher level. From the main menu, this selection causes **t167** to terminate.

exit

From any menu, this selection causes **t167** to terminate. This item and *quit* in the main menu are redundant; both are included to maintain consistency with all other menus.

t167 Configuration Menu

The selections within this submenu permit a user to alter certain parameters used throughout **t167**. Although the design of **t167** is configured to useful default parameters at run-time, it may sometimes be useful to change one or more of the configurable values, particularly when doing development work.

Item	Name	Description
1	newcs	Allocate New Control Structure Set
2	newdb	Allocate New Data Buffer
3	dbp	Display / Alter Data Buffer Parameters
4	addr	SDL and NCR Firmware Addresses
5	term	Select Terminal Type
h	help	Display this menu
s	status	Status of previous command
q	quit	Return to previous menu
x	exit	Exit from t167

Allocate New Control Structure Set

Several structures are used to pass parameters to the SDL and SIOP Firmware. One set of structures is provided when **t167** runs. Additional control structure sets may be allocated by selecting this item. The maximum number of sets, currently 16, is defined by **MAX_CS** in the file *t167.h*.

Allocate New Data Buffer

One buffer area for data transfer operations is provided by **t167**. Selection of this item permits a user to allocate additional data buffers. Data buffers are referenced by sequentially assigned numbers. When a new buffer is allocated, the user is prompted for the length of the buffer in bytes. The user may also specify an absolute starting address; however, this results in system faults when **t167** is running under a host operating system, and can result in any number of errors during stand-alone operation unless extreme care is taken. Once allocated, buffers cannot be deallocated. The maximum number of data buffers, currently 16, is defined by **MAX_DB** in the file *t167.h*.

Display/Alter Data Buffer Parameters

The base address and length of each allocated data buffer are displayed, as shown in the example below. The base address and length cannot be changed (even though the menu item says "alter"). Example display:

```
Buffer          First Address       End Address          Length
  0               0x4080f4            4084f3       1024, (0x400)
  1               0x4180f8            41ac07       11024, (0x2b10)
```

SDL and NCR Firmware Addresses

The entry points into the SDL and SIOP Firmware are maintained as variables which are initialized at run time to the addresses of the SDL and SIOP Firmware modules linked with **t167**. This menu item displays the current addresses.

Select Terminal Type

Certain data displays are capable of overwriting displayed values rather than scrolling. Support for this option requires knowledge specific to the terminal in use. One terminal type is defined at this time.

dumb	This is the default terminal type. No special terminal control codes are used, and the display is always scrolled once the bottom line has been reached.

SCSI Driver Library Development Tools Menu

This submenu provides access to a number of functions which are useful when working with the SDL. From this menu, a user may build and issue SDL commands. Once a command has been executed by the SDL, other menu selections permit the user to inspect the results returned by the SDL.

Item	Name	Description
1	bldsdlcs	Build sdl_cmd structure
2	sdlinit	Issue sdl_init command
3	sdlread	Issue sdl_read command
4	sdlwrite	Issue sdl_write command
5	sdlcntrl	Issue sdl_cntrl command
6	dspsdlcs	Display sdl_cmd structure
7	dspdb	Display data buffer contents
8	testsn	Display/set test serial number
9	reset	Reset SCSI bus
h	help	Display this menu
s	status	Status of previous command
q	quit	Return to previous menu
x	exit	Exit from t167

Note **Certain commands require data space, in the form of a buffer and/or configuration control structure. If needed, each command queries the user to determine which buffer or control structure is to be used. This permits a sequence of commands to be executed repeatedly without the need to reconfigure before each.**

Commands which query for specific data values list acceptable values in square brackets and existing values in ellipses. Existing values may be selected by responding with a carriage return. If an unacceptable response is detected, the prompt is reissued. If a '.' is entered at a parameter prompt, any new information is retained and all other data left intact. An 'x' at a parameter prompt results in termination of **t167**.

The display commands can produce large amounts of output data. Listing control is built into these commands to limit display pages to 24 lines or less. As with the parameter prompts, a '.' or 'x' entered at a paging prompt causes the command or **t167** to terminate.

Build sdl_cmd Structure

This command is used to completely configure a **t167** control structure prior to the execution of an SDL command. Any allocated control structure may be configured. The user is asked to provide information regarding the type of command to be built.

Refer to Tables E-1 and E-2 for a list of the SDL commands. In general, an sdl_init followed by an ATTACH command must be the first commands sent to the SDL. As shown in the tables, some commands are allowed prior to the ATTACH.

If the command is ATTACH, a list of supported devices is displayed. Upon selection of a supported device, all defined parameters for that device are copied from a statically-configured area to the selected control structure. Information regarding device-specific configuration is discussed in the section *Adding SDL Tools Support For New Devices* later in this appendix.

Issue sdl_init Command

The SDL is entered through the **sdl_init** entry point. Unless specified on the command line, the user is prompted for scsi_addr, intr_lvl, clk_speed, and snoopmode values. Return status is displayed upon completion of the **sdl_init** command. Chapter 3 describes these parameters in the **siop_init** call (they are the same as for the **sdl_init** call).

Command format: **sdl_init** <scsi_addr> <intr_lvl> <clk_speed> <snoopmode>

Issue sdl_read Command

The SDL is entered through the **sdl_read** entry point. Unless specified on the command line, the user is prompted for an allocated control structure number. Status is displayed upon completion of the **sdl_read** command.

Command format: **sdlread** [-c]
sdlread <control_structure>

If the 'c' option is selected, the current (last specified) control structure is used. A control structure may not be specified if this option is selected.

Issue sdl_write Command

The SDL is entered through the **sdl_write** entry point. Unless specified on the command line, the user is prompted for an allocated control structure number. Status is displayed upon completion of the **sdl_write** command.

Command format: **sdlwrite** [-c]
sdlwrite <control_structure>

If the 'c' option is selected, the current (last specified) control structure is used. A control structure may not be specified if this option is selected.

Issue sdl_cntrl Command

The SDL is entered through the **sdl_cntrl** entry point. Unless specified on the command line, the user is prompted for an allocated control structure number. Status is displayed upon completion of the **sdl_cntrl** command.

Refer to Tables E-1 and E-2 for a list of the SDL commands. In general, an ATTACH command must be the first command sent to the SDL (after the sdl_init is performed). As shown in the tables, some commands are allowed prior to the ATTACH.

Command format: **sdlcntrl** [-c]
sdlcntrl <control_structure>

If the 'c' option is selected, the current (last specified) control structure is used. A control structure may not be specified if this option is selected.

Display sdl_cmd Structure

The entire contents of a selected **sdl_cmd** structure are displayed. Individual structure elements are labeled according with the names assigned in their respective templates. Arrays are displayed in the same manner as data buffers, an example of which can be found below.

dspsdlcs displays meaningful information only if the **sdl_cmd** structure has previously been initialized.

Display Data Buffer Contents

The contents of the data buffer pointed to by the current SDL control structure are displayed, in the format shown below. The first and last addresses of the selected buffer are also displayed.

Data buffer contents display example:

```
Offset                  Hexadecimal Value                    ASCII Equivalent
db[0]:    First address = 0x1fbdca0  Last address = 0x1fbe09f
 60    60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F    `abcdefghijklmno
 70    70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F    pqrstuvwxyz{|}~.
```

Command format: **dspdb** [-c] [-b*buffer*] <offset> <length>

offset is number of bytes from beginning of buffer at which buffer display is to begin.

length is total number of bytes to display. This value is always limited internally to at most the number of bytes from *offset* to the end of the selected buffer.

Options:

-b*buffer* Display contents of *buffer* (default: current data buffer)

-c Display buffer contents continuously. Unless this option is selected, the data buffer contents is displayed in 256-byte blocks.

Display/Set Test Serial Number

This menu item does not perform any useful action.

Reset SCSI Bus

This menu item does not perform any useful action.

Table E-1. SDL Direct Access Commands

cmd	hex code	*arg*	*argp*	notes
ATTACH	0x06	Driver status routine	Pointer to work area	Uses SCSI-2 mode pages
RD_SCAT_GAT	0x07	Start block	Pointer to S/G list	S/G list longword aligned
WR_SCAT_GAT	0x08	Start block	Pointer to S/G list	S/G list longword aligned
SAFE_READ	0x09	Size of buffer/ Number of blocks	Pointer to buffer	Boot device read
INQIRY (NOTE)	0x0A	Size in bytes of inquiry data	Pointer to buffer	Device dependent format
PASS_THRU (NOTE)	0x0B	Address of status routine	Pointer to buffer	arg=0, FILTERED mode
SCSI_RST (NOTE)	0x0C	Not used	Not used	Resets all devices on SCSI Bus
CNTL_RST (NOTE)	0x0D	Not used	Not used	Resets specified SCSI controller
RESRVE	0x0E	0 = Release, 1 = Reserve	Not used	State = released after 1st ATTACH
CHG_INTR (NOTE)	0x0F	Interrupt level	Not used	Change SIOP interrupt level
GET_INFO	0x10	Size of argp data area	Pointer to data area	Get peripheral information
FORMAT	0x20	Defect list length	Pointer to defect list	Device Defined Defect List
ASG_ALTS	0x21	Defect list length	Pointer to defect list	Device Defined Defect List
DEF_FORMAT	0x22	Size of argp data area	Pointer to data area	Restores all pages to default
RD_CAPACITY	0x23	Returned block size	Returned block capacity	Read capacity

NOTE: Command may be issued prior to the first attach.

Table E-2. SDL Supported Sequential Access Commands

cmd	hex code	*arg*	*argp*	notes
ATTACH	0x06	Driver status routine	Pointer to work area	Uses SCSI-2 mode pages
RD_SCAT_GAT	0x07	Start block	Pointer to S/G list	S/G list longword aligned
WR_SCAT_GAT	0x08	Start block	Pointer to S/G list	S/G list longword aligned
SAFE_READ	0x09	Size of buffer/ Number of blocks	Pointer to buffer	Boot device read
INQIRY (NOTE)	0x0A	Size in bytes of inquiry data	Pointer to buffer	Device dependent format
PASS_THRU (NOTE)	0x0B	Address of status routine	Pointer to buffer	arg=0, FILTERED mode
SCSI_RST (NOTE)	0x0C	Not used	Not used	Resets all devices on SCSI Bus
CNTL_RST (NOTE)	0x0D	Not used	Not used	Resets specified SCSI controller
RESRVE	0x0E	0 = Release, 1 = Reserve	Not used	State = released after 1st ATTACH
CHG_INTR (NOTE)	0x0F	Interrupt level	Not used	Change SIOP interrupt level
GET_INFO	0x10	Size of argp data area	Pointer to data area	Get peripheral information
SPCE	0x20	Space Control:# of Units (+,-) to Space	Units = blocks or filemarks	
WR_FMKS	0x21	Number of Filemarks to write	Not used	Maximum filemarks = 128
REWND	0x22	Not used	Not used	Rewinds to BOT
ERASE	0x23	Erase mode control	Not used	Short and long erase
LOAD_UNLOAD	0x24	Space mode control	Not used	Position to BOT, Turn LED on/off
RD_BLK_LIM	0x25	Return Max Limit	Return Min Limit	Read Block Limits

NOTE: Command may be issued prior to the first attach.

NCR Firmware Development Tools Menu

This submenu provides access to a number of functions which are useful when working with the SIOP Firmware. From this menu, a user may build and issue commands to the SIOP Firmware. Once a command has been executed, other menu selections permit the user to inspect the results returned by the SIOP Firmware.

Item	Name	Description
1	bldfwcs	Build F/W Control Structure
2	fwinit	Issue siop_init Command
3	fwcmd	Issue siop_cmd Command
4	dspcdb	Display command descriptor block
5	dspfwcs	Display F/W Control Structure
6	dspfwstat	Display F/W Status
7	dspdb	Display Data Buffer Contents
8	reset	Reset SCSI bus
h	help	Display this menu
s	status	Status of previous command
q	quit	Return to previous menu
x	exit	Exit from t167

Note **Certain commands require data space, in the form of a buffer and/or configuration control structure. If needed, each command queries the user to determine which buffer or control structure is to be used. This permits a sequence of commands to be executed repeatedly without the need to reconfigure before each.**

Commands which query for specific data values list acceptable values in square brackets and existing values in ellipses. Existing values may be selected by responding with a carriage return. If an unacceptable response is detected, the prompt is reissued. If a '.' is entered at a parameter prompt, any new information is retained and all other data left intact. An 'x' at a parameter prompt results in termination of **t167**.

The display commands can produce large amounts of output data. Listing control is built into these commands to limit display pages to 24 lines or less. As with the parameter prompts, a '.' or 'x' entered at a paging prompt causes the command or **t167** to terminate.

Build F/W Control Structure

Interactive function which prompts user for values necessary to create an SIOP Firmware command. Default values are provided where applicable, and all values are retained from one Firmware command invocation to the next.

Issue siop_init Command

The SIOP Firmware is entered through the **siop_init** entry point. Unless specified on the command line, the user is prompted for scsi_addr, intr_lvl, clk_speed, and snoopmode values. Return status is displayed upon completion of the siop_init command. Chapter 3 describes these parameters in the **siop_init** call.

Command format: **siop_init** <scsi_addr> <intr_lvl> <clk_speed> <snoopmode>

Issue siop_cmd Command

Control is passed to the SIOP Firmware, with the address of the current SIOP Firmware control structure as an argument. Final status is displayed upon completion of the command.

Display Command Descriptor Block

The command descriptor block contained in the current SIOP Firmware control structure is displayed.

Display F/W Firmware Control Structure

The current SIOP Firmware control structure is displayed, along with descriptors which identify each field.

Display F/W Status

The Firmware status value from the current SIOP Firmware control structure is displayed.

Display Data Buffer Contents

The contents of the data buffer pointed to by the current SIOP Firmware control structure are displayed, in the format shown below. The first and last addresses of the selected buffer are also displayed.

Data buffer contents display example:

```
Offset                  Hexadecimal Value                  ASCII Equivalent
db[0]:   First address = 0x1fbdca0  Last address = 0x1fbe09f
  60    60 61 62 63 64 65 66 67  68 69 6A 6B 6C 6D 6E 6F `abcdefghijklmno
  70    70 71 72 73 74 75 76 77  78 79 7A 7B 7C 7D 7E 7F pqrstuvwxyz{|}~.
```

Command format: **dspdb** [-c] [-b*buffer*] <offset> <length>

offset is number of bytes from beginning of buffer at which buffer display is to begin.

length is total number of bytes to display. This value is always limited internally to at most the number of bytes from *offset* to the end of the selected buffer.

Options:

-b*buffer* Display contents of *buffer* (default: current data buffer)

-c Display buffer contents continuously. Unless this option is selected, the data buffer contents is displayed in 256-byte blocks.

Reset SCSI Bus

This menu item does not perform any useful action.

Menu Expansion

New menus are added by creating a uniquely-named **menu** structure and adding a **menu_item** structure with the address of the new menu to a parent menu. The total number of menu selections within a menu is **MAX_ITEMS**. If fewer selections are contained within a menu, the final **menu_item** structure should be **MENU_END**. Each menu should also contain a *help* selection, which contains a pointer to the pseudo-function **MENU_HELP**. **MAX_ITEMS**, **MENU_HELP**, and **MENU_QUIT** are defined in the file *menus.h*. All menu declarations are in the file *menus.c*.

For convenience, the structure *example* in the file *menus.c* has been created as a foundation for new menus.

Adding SDL Tools Support for New Devices

t167 currently provides support for a number of common devices. Some provisions have been made for expanding the list of known devices. Peripherals of the same types as those already supported: direct-access, fixed-media devices; direct-access, removable-media devices; and sequential-access devices may be supported without adding program code, although several modules have to be compiled and the program re-linked. Entirely different peripheral types, such as CD-ROM devices, require more extensive changes.

Shown below are the steps required to add support for a new device of a known type.

1. Choose a unique filename of eight or fewer characters to contain the data for the new device. This filename is referenced as *file*.
2. Create "*file*.c" containing appropriate initialization constants. For convenience, use an existing device file as a template. An example of a direct-access, fixed-media device file is *wrenIV.c* (CDC WREN IV hard disk). A direct-access, removable-media device file is *fc1dsdd5.c* (double sided, double density floppy); and a sequential-access device file is *archive.c* (Archive 2150 tape). The following table list the files which contain the static initialization structures.

Table E-3. Template Files

Device Type	File	Array
direct-access, fixed-media devices, i.e., conventional hard-disk drives	"t167daf.c"	sdp_daf[ ]
direct-access, removable-media devices, i.e., floppy-disk drives	"t167dar.c"	sdp_dar[ ]
sequential-access devices, i.e., streaming tape drives	"t167sad.c"	sdp_sad[ ]

3. Edit "*file*.c", altering all parameters required to support the new device as required. Refer to the SCSI-2, SCSI Driver Library, and manufacturer's specifications for initialization parameters.
4. If a "Makefile" is being used, add "*file*.c" and "*file*.o" to the SOURCES and OBJECTS lists, as well as the compiler invocation instructions which build "*file*.o".
5. From the above table, locate the appropriate array for the device type being added, in the file "globals.c". Add the name of the new device to the array. If more than MAX_DEVICES (defined in "t167.h") entries appear in the array declaration, the compiler generates an error message. Either the value of MAX_DEVICES must then be changed, or support for an existing device deleted.

Example Use of t167

This section describes the typical use of **t167**. Once it is loaded and executing, these step-by-step examples can be followed to gain an understanding of the overall operation.

Shown below is the main menu presented by **t167**:

```
MVME167/187 SCSI Test and Development Utility

 Item   Name              Description
 ----  ------        --------------------
   1   cnfg          t167 Configuration
   2   sdldt         SCSI Driver Library Development Tools
   3   fwdt          NCR Firmware Development Tools
   4   sdltst        SCSI Driver Library Tests
   5   fwtst         NCR Firmware Tests
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Exit from t167
   x   exit          Exit from t167

t167 -->
```

The use of **t167** with the SDL is presented first, followed by NCR Firmware examples.

Use of t167 with the SDL

When issuing **t167** commands through the SDL interface, the following procedure **MUST** be executed prior to issuing any other commands to the SDL through **t167**.

1. From the main **t167** menu above, select menu item **2** to display the *sdldt* screen:

```
SCSI Driver Library Development Tools

 Item   Name              Description
 ----  ------        --------------------
   1   bldsdlcs      Build sdl_cmd structure
   2   sdlinit       Issue sdl_init command
   3   sdlread       Issue sdl_read command
   4   sdlwrite      Issue sdl_write command
   5   sdlcntrl      Issue sdl_cntrl command
   6   dspsdlcs      Display sdl_cmd structure
   7   dspdb         Display data buffer contents
   8   testsn        Display/set test serial number
   9   reset         Reset SCSI bus
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Return to previous menu
   x   exit          Exit from t167

SDL Tools -->
```

2. The user **MUST** select menu item **2** (sdlinit) prior to issuing any other commands to the SDL. Answer with a **<CR>** to select the defaults as shown below:

```
SDL Tools --> 2
   SCSI address (7):
Interrupt level (0):
Clock frequency ('2500'):
     Snoop-mode (0):
sdl_init status: 0x0
SDL Tools -->
```

3. The user **MUST** next select menu item **1** (bldsdlcs) to configure for an ATTACH command to the device prior to issuing any other commands to the SDL. Answer as shown in the example below. The example assumes a Wren V device.

```
SDL Tools --> 1
Control structure [0] (0):
Structure type (control or read/write) [c, r] (c):
Device type (direct or sequential access)  [d, s] (d):
Medium type (fixed or removable)  [f, r] (f):
32-bit SCSI device number (0, 0x0):
Command (0x6, ATTACH):
Supported devices:

     1. Seagate Wren IV
     2. Seagate Wren V

Device (1): 2
SDL Tools -->
```

4. To verify the command parameters prior to issuing the ATTACH command, select menu item **6** as shown below:

```
SDL Tools --> 6
Control structure [0] (0):

Contents of sdl_cmd structure:
  sdl_cmd.io structure type ......... sdl_cntrl

Contents of sdl_cntrl structure:
  UINT dev ........................... 0x0
  UINT cmd ........................... 0x6, ATTACH
  UINT arg ........................... 0x1FD1AA8
  UINT argp .......................... 0x1FBCE50

Device type .......................  direct-access, fixed-media
Descriptor ........................  Seagate Wren V

<CR> for more, 'q' or '.' to quit, 'x' to exit: .
SDL Tools -->
```

5. Now, issue the ATTACH command by selecting menu item 5.

```
SDL Tools --> 5
Control structure [0] (0):
sdl_cntrl status: 0x0
SDL Tools -->
```

A status of 0 indicates a good ATTACH.

Add a Second Data Buffer to the t167 Configuration

In order to issue an SDL INQIRY command to the device, **t167** must have a second data buffer allocated of length less than the 0xff byte maximum allowed by the SCSI-2 when issuing an INQUIRY to a device. This procedure demonstrates the creating of such a buffer.

1. From the current menu, enter a **q** command to go back one menu level to the **t167** entry menu.

```
SCSI Driver Library Development Tools
 Item   Name              Description
 ----  ------        --------------------
   1   bldsdlcs      Build sdl_cmd structure
   2   sdlinit       Issue sdl_init command
   3   sdlread       Issue sdl_read command
   4   sdlwrite      Issue sdl_write command
   5   sdlcntrl      Issue sdl_cntrl command
   6   dspsdlcs      Display sdl_cmd structure
   7   dspdb         Display data buffer contents
   8   testsn        Display/set test serial number
   9   reset         Reset SCSI bus
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Return to previous menu
   x   exit          Exit from t167

SDL Tools --> q
```

2. At the top menu, select item **1** to configure **t167**.

```
SCSI Test and Development Utility
 Item   Name              Description
 ----  ------        --------------------
   1   cnfg          t167 Configuration
   2   sdldt         SCSI Driver Library Development Tools
   3   fwdt          NCR Firmware Development Tools
   4   sdltst        SCSI Driver Library Tests
   5   fwtst         NCR Firmware Tests
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Exit from t167
   x   exit          Exit from t167

t167 --> 1
```

3. At the **t167** configuration menu, select item **2**, to create a new data buffer.

```
t167 Configuration

 Item   Name              Description
 ----  ------        --------------------
   1   newcs         Allocate new control structure set
   2   newdb         Allocate new data buffer
   3   dbp           Display/alter data buffer parameters
   4   addr          SDL and NCR firmware addresses
   5   term          Select terminal type
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Return to previous menu
   x   exit          Exit from t167

t167 Config --> 2
```

4. Specify a length 0x40 bytes for this new buffer for use when issuing a SCSI INQUIRY command to the device.

```
t167 Config --> 2
Allocating buffer 2
Length, in bytes (1024): 0x40
Use next available starting address? [n, y] (y): y
```

5. Issue a **q** command to return to the main **t167** menu, and then select menu item **2** to return back to the SDL development menu.

```
t167 --> q

SCSI Test and Development Utility

 Item   Name              Description
 ----  ------        --------------------
   1   cnfg          t167 Configuration
   2   sdldt         SCSI Driver Library Development Tools
   3   fwdt          NCR Firmware Development Tools
   4   sdltst        SCSI Driver Library Tests
   5   fwtst         NCR Firmware Tests
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Exit from t167
   x   exit          Exit from t167

t167 -->2
```

Issue an INQIRY Command

1. To issue an SDL INQIRY command to the device, build a new control command through menu selection **1**. Here, input INQIRY instead of ATTACH. *(Note that entry for the SDL command can either be the ASCII name or the command code, both obtained from the tables presented earlier.)*

```
SDL Tools --> 1
Control structure [0] (0):
Structure type (control or read/write) [c, r] (c):
Device type (direct or sequential access)  [d, s] (d):
Medium type (fixed or removable)  [f, r] (f):
32-bit SCSI device number (0, 0x0):
Command (0x6, ATTACH): INQIRY
Data buffer [0 - 1] (0): 1
SDL Tools -->
```

2. Verify the command structure, select menu item **6**, and then select menu item **5** to issue the command to the SDL.

```
SDL Tools --> 6
Control structure [0] (0):

Contents of sdl_cmd structure:
  sdl_cmd.io structure type ......... sdl_cntrl

Contents of sdl_cntrl structure:
  UINT dev ........................... 0x0
  UINT cmd ........................... 0xA, INQIRY
  UINT arg ........................... 0x40
  UINT argp .......................... 0x1FBDCA0

<CR> for more, 'q' or '.' to quit, 'x' to exit: .

SDL Tools --> 5
Control structure [0] (0):
sdl_cntrl status: 0x800
SDL Tools -->
```

(Note the return status of 0x800. This is data underrun status and is acceptable because the data count we sent to the SDL/NCR Firmware exceeded the actual amount of data transferred.)

3. Select menu item **7** to display the data buffer holding the INQIRY data:

```
SCSI Driver Library Development Tools

 Item   Name              Description
 ----  ------        --------------------
   1   bldsdlcs      Build sdl_cmd structure
   2   sdlinit       Issue sdl_init command
   3   sdlread       Issue sdl_read command
   4   sdlwrite      Issue sdl_write command
   5   sdlcntrl      Issue sdl_cntrl command
   6   dspsdlcs      Display sdl_cmd structure
   7   dspdb         Display data buffer contents
   8   testsn        Display/set test serial number
   9   reset         Reset SCSI bus
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Return to previous menu
   x   exit          Exit from t167

SDL Tools --> 7
Data buffer [0 - 1] (1):
Offset into buffer (0):
Number of bytes to display (64, 0x40):

db[0]:  First address = 0x1fbdca0  Last address = 0x1fbe09f

 0  00 00 01 01  1F 00 00 00  48 50 20 20  20 20 20 20  ........HP
10  39 37 35 34  38 53 20 20  20 20 20 20  20 20 20 20  97548S
20  38 39 31 36  00 00 00 00  00 00 00 00  00 00 00 00  8916............
30  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
SDL Tools -->
```

Issue a Format Command

The format command **FORMAT** allows a user to format a device. If the user is unsure of the current format of the device, it is advisable to issue this command. Generally, this command can take a while to complete.

Caution **The Format command erases any data on a disk.**

1. Select menu item **1** to set up the command structure for a format command.

```
SDL Tools --> 1
Control structure [0] (0):
Structure type (control or read/write) [c, r] (c):
Device type (direct or sequential access)  [d, s] (d):
Medium type (fixed or removable)  [f, r] (f):
32-bit SCSI device number (0, 0x0):
Command (0xA, INQIRY): FORMAT
FmtData bit (clear or set) [c, s] (clear): c
CmpLst bit (clear or set) [c, s] (clear): c
Defect list formats:

     0. Block format
     4. Bytes from index format
     5. Physical sector format
     6. Vendor specific format (not supported)

Defect list format (0):
SDL Tools -->
```

2. Select menu item **6** to verify the command structure, then issue the command to the device by selecting menu item **5**.

```
SDL Tools --> 6
Control structure [0] (0):

Contents of sdl_cmd structure:
  sdl_cmd.io structure type ......... sdl_cntrl

Contents of sdl_cntrl structure:
  UINT dev ........................... 0x0
  UINT cmd ........................... 0x20, FORMAT
  UINT arg ........................... 0x0
  UINT argp .......................... 0x1FB9CA0

<CR> for more, 'q' or '.' to quit, 'x' to exit: .

SDL Tools --> 5
Control structure [0] (0):
sdl_cntrl status: 0x0
SDL Tools -->
```

(Allow enough time for the FORMAT to complete.)

Issue Reads and Writes to a Disk Device

The **t167** tools do not provide a way to directly modify data buffers, but instead rely upon the MVME167/187 ROM Debugger for this function. This exercise demonstrates this procedure as well as demonstrates how to read and write to a device using the **t167**/SDL tools.

1. Use the **q** command to go from the current menu to the main **t167** menu and select menu item **1**, t167 cnfg.
2. In the cnfg menu, select item **3** to display the current data buffer parameters. Use this function to get the address and size of data buffer 0.

```
t167 Configuration

 Item   Name             Description
 ----  ------        --------------------
   1   newcs         Allocate new control structure set
   2   newdb         Allocate new data buffer
   3   dbp           Display/alter data buffer parameters
   4   addr          SDL and NCR firmware addresses
   5   term          Select terminal type
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Return to previous menu
   x   exit          Exit from t167

t167 Config --> 3
Statistics for allocated data buffers:

  Buffer     First Address       End Address         Length

     0          0x1FBDCA0         0x1FBE09F     1024, (0x400)
     1          0x1FB9CA0         0x1FB9CDF     64, (0x40)

t167 Config -->
```

3. Hit the ABORT switch on the front panel of the target CPU and use the memory modify (**MM**) command to fill in the contents of data buffer 0.

```
t167 -->
Exception: Interrupt Level 7 (ABORT Switch)
Vector Number =1, Address =008
SXIP =0001A1E4 SNIP =0001A1E2 TPSR =800003F0
IP   =0001A1E0 CR02 =800003F0 CR07 =0005D000 R00  =00000000
R01  =DEADDEAD R02  =00000000 R03  =00000000 R04  =00010000
R05  =01F76000 R06  =00000000 R07  =00000000 R08  =00000000
R09  =00000000 R10  =00000000 R11  =00000000 R12  =00000000
R13  =00000000 R14  =00000000 R15  =00000000 R16  =00000000
R17  =00000000 R18  =00000000 R19  =00000000 R20  =00000000
R21  =00000000 R22  =00000000 R23  =00000000 R24  =00000000
R25  =00000000 R26  =00000000 R27  =00000000 R28  =00000000
R29  =00000000 R30  =00000000 R31  =01FFEDA0
0001A1E0 C0000000  BR          $0001A1E0
187-Bug>md 1fbdca0:10
01FBDCA0  00000000 00000000 00000000 00000000  ................
01FBDCB0  00000000 00000000 00000000 00000000  ................
01FBDCC0  00000000 00000000 00000000 00000000  ................
01FBDCD0  00000000 00000000 00000000 00000000  ................
187-Bug>bf 1fbdca0 1fbdca0+400 'DADA'
```

4. Issue the go (**G**) command to the MVME167/187 ROM Debugger and hit an **h** to repaint the **t167** screen. Go back to the SDL screen, and initialize the command structure for a write.

```
SDL Tools --> 1
Control structure [0] (0):
Structure type (control or read/write) [c, r] (c): r
32-bit SCSI device number (0, 0x0):
Data buffer [0, 1] (0):
Logical block count (33372840, 0x1FD3AA8): 2
Logical block number (33287760, 0x1FBEE50): 0
SDL Tools -->
```

5. Select menu item **6** to verify the command structure.

```
SDL Tools --> 6
Control structure [0] (0):

Contents of sdl_cmd structure:
  sdl_cmd.io structure type ......... sdl_rw

Contents of sdl_rw structure:
  UINT dev ........................... 0x0
  UCHAR *bufp ........................ 0x1FBFCA0
  UINT cnt ........................... 2, 0x2
  UINT block ......................... 0, 0x0

<CR> for more, 'q' or '.' to quit, 'x' to exit: .
SDL Tools -->
```

6. Select menu item **4** to send the WRITE command to the SDL.

```
SCSI Driver Library Development Tools

 Item   Name             Description
 ----  ------        --------------------
   1   bldsdlcs      Build sdl_cmd structure
   2   sdlinit       Issue sdl_init command
   3   sdlread       Issue sdl_read command
   4   sdlwrite      Issue sdl_write command
   5   sdlcntrl      Issue sdl_cntrl command
   6   dspsdlcs      Display sdl_cmd structure
   7   dspdb         Display data buffer contents
   8   testsn        Display/set test serial number
   9   reset         Reset SCSI bus
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Return to previous menu
   x   exit          Exit from t167

SDL Tools --> 4
Control structure [0] (0):
sdl_write status: 0x0
SDL Tools -->
```

7. Press the ABORT switch on the front panel and use the MVME167/187 ROM Debugger block fill (**BF**) command to re-write the data buffer to zeros.

```
SDL Tools -->
Exception: Interrupt Level 7 (ABORT Switch)
Vector Number =1, Address =008
SXIP =0001A1E2 SNIP =0001A1E4 TPSR =800003F0
IP   =0001A1E0 CR02 =800003F0 CR07 =0005D000 R00  =00000000
R01  =DEADDEAD R02  =00000000 R03  =00000000 R04  =00010000
R05  =01F76000 R06  =00000000 R07  =00000000 R08  =00000000
R09  =00000000 R10  =00000000 R11  =00000000 R12  =00000000
R13  =00000000 R14  =00000000 R15  =00000000 R16  =00000000
R17  =00000000 R18  =00000000 R19  =00000000 R20  =00000000
R21  =00000000 R22  =00000000 R23  =00000000 R24  =00000000
R25  =00000000 R26  =00000000 R27  =00000000 R28  =00000000
R29  =00000000 R30  =00000000 R31  =01FFEDA0
0001A1E0 C0000000  BR          $0001A1E0
187-Bug>md 1fbdca0:10
01FBDCA0  44414441 44414441 44414441 44414441  DADADADADADADADA
01FBDCB0  44414441 44414441 44414441 44414441  DADADADADADADADA
01FBDCC0  44414441 44414441 44414441 44414441  DADADADADADADADA
01FBDCD0  44414441 44414441 44414441 44414441  DADADADADADADADA
187-Bug>bf 1fbdca0 1fbdca0+400 0
Effective address: 01FBDCA0
Effective address: 01FBE09F
187-Bug>md 1fbdca0:10
01FBDCA0  00000000 00000000 00000000 00000000  ................
01FBDCB0  00000000 00000000 00000000 00000000  ................
01FBDCC0  00000000 00000000 00000000 00000000  ................
01FBDCD0  00000000 00000000 00000000 00000000  ................
187-Bug>
```

8. Issue the go (**G**) command to the MVME167/187 ROM Debugger and hit an **h** to repaint the **t167** screen. Select menu item **3** to send the read command to the SDL.

```
SDL Tools --> 3
Control structure [0] (0):
sdl_read status: 0x0
```

9. Select menu item **7** to display the data buffer contents.

```
SDL Tools --> 7
Data buffer [0, 1] (0):
Offset into buffer (0):
Number of bytes to display (1024, 0x400): 0x60
db[0]:  First address = 0x1fbdca0  Last address = 0x1fbe09f
 0  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
10  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
20  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
30  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
40  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
50  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
SDL Tools -->
```

Use of t167 with the NCR Firmware

Use of **t167** with the NCR Firmware is described in this section. This is the lowest level of interface to the local SCSI bus, and allows the user to build direct SCSI commands (CDBs) to devices on the SCSI bus.

Initialize the NCR Firmware Interface

The following procedure **MUST** be executed prior to issuing any other commands to the Firmware.

1. From the main **t167** menu, select menu item **3** to display the Firmware development screen:

```
MVME167/187 SCSI Test and Development Utility
 Item   Name              Description
 ----  ------        --------------------
   1   cnfg          t167 Configuration
   2   sdldt         SCSI Driver Library Development Tools
   3   fwdt          NCR Firmware Development Tools
   4   sdltst        SCSI Driver Library Tests
   5   fwtst         NCR Firmware Tests
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Exit from t167
   x   exit          Exit from t167

t167 --> 3
```

2. Select menu item **2** to initialize the Firmware interface. This **MUST** be done even if the initialization was done previously from the SDL interface.

```
NCR Firmware Development Tools
 Item   Name              Description
 ----  ------        --------------------
   1   bldfwcs       Build F/W Control Structure
   2   fwinit        Issue siop_init Command
   3   fwcmd         Issue siop_cmd Command
   4   dspcdb        Display command descriptor block
   5   dspfwcs       Display F/W Control Structure
   6   dspfwstat     Display F/W Status
   7   dspdb         Display Data Buffer Contents
   8   reset         Reset SCSI bus
   h   help          Display this menu
   s   status        Status of previous command
   q   quit          Return to previous menu
   x   exit          Exit from t167

F/W Tools --> 2
   SCSI address (7):
Interrupt level (0):
Clock frequency ('2500'):
     Snoop-mode (0):
siop_init status: 0x0
F/W Tools -->
```

Send SCSI INQUIRY to the Device

Building commands at the Firmware interface level requires the construction of an *siop_struc* command structure (described in Appendix B).

The initialization of the Firmware interface in the previous step resulted in a reset of the NCR Firmware internal SCSI data transfer mode configuration for all devices to the **ASYNC** mode. If any SDL commands have been used previously, then the actual SCSI drive may be set up for **SYNC**. To re-align the Firmware and devices transfer modes, the first command to the device that results in a data phase must first force a re-negotiation of the data transfer mode. In this example, we set bit 0 of the command control word to force **SYNC** data transfers. If the re-negotiation is not done, the drive hangs when commands are sent to it.

1. Select menu item **1** to build a Firmware control structure. Note that most of the parameters can be left at their default values.

```
F/W Tools --> 1
Control structure [0] (0):
  UINT  user_defined ............... (0):
  UINT cmd_ctrl .................... (0): 0x8000000D
  UINT addr_ilvl ................... (0): 0
  UINT lun ......................... (0): 0
  UINT cdb_lgth .................... [0 - 12] (0): 6
    UCHAR cdb[0] ................... (0): 0x12
    UCHAR cdb[1] ................... (0):
    UCHAR cdb[2] ................... (0):
    UCHAR cdb[3] ................... (0):
    UCHAR cdb[4] ................... (0): 0x40
    UCHAR cdb[5] ................... (0):
  UINT msg_in_lgth ................. (12):
  UCHAR *msg_in_ptr ................ (0x1FBCCDC):
    UCHAR msg_in[0] ................ (0):
    UCHAR msg_in[1] ................ (0):
    UCHAR msg_in[2] ................ (0):
    UCHAR msg_in[3] ................ (0):
    UCHAR msg_in[4] ................ (0):
    UCHAR msg_in[5] ................ (0):
    UCHAR msg_in[6] ................ (0):
    UCHAR msg_in[7] ................ (0):
    UCHAR msg_in[8] ................ (0):
    UCHAR msg_in[9] ................ (0):
    UCHAR msg_in[10] ............... (0):
    UCHAR msg_in[11] ............... (0):
  UINT msg_out_lgth ................ (12):
  UCHAR *msg_out_ptr ............... (0x1FBCCF0):
    UCHAR msg_out[0] ............... (0):
    UCHAR msg_out[1] ............... (0):
    UCHAR msg_out[2] ............... (0):
    UCHAR msg_out[3] ............... (0):
    UCHAR msg_out[4] ............... (0):
    UCHAR msg_out[5] ............... (0):
```

```
    UCHAR msg_out[6] ............... (0):
    UCHAR msg_out[7] ............... (0):
    UCHAR msg_out[8] ............... (0):
    UCHAR msg_out[9] ............... (0):
    UCHAR msg_out[10] ............... (0):
    UCHAR msg_out[11] ............... (0):
  UINT data_count ................. (1024):
  UCHAR *data_ptr ................. (0x1FBDCA0):
  struct siop_struc *link_ptr ..... (0):
    UCHAR scsi_phase[0] ............. (0):
    UCHAR scsi_phase[1] ............. (0):
    UCHAR scsi_phase[2] ............. (0):
    UCHAR scsi_phase[3] ............. (0):
    UCHAR scsi_phase[4] ............. (0):
    UCHAR scsi_phase[5] ............. (0):
    UCHAR scsi_phase[6] ............. (0):
    UCHAR scsi_phase[7] ............. (0):
  void (*status_ptr)() ............ (0x1FD1A70):
F/W Tools -->
```

2. Select menu item **5** to display a portion of the command structure.

```
F/W Tools --> 5
Control structure [0] (0):

Address of siop_struc structure: 0x1FBCCB4
Contents of siop_struc structure:
  UINT  user_defined ............... 0x0
  UINT cmd_ctrl .................... 0x8000000D
  UINT addr_ilvl ................... 0, 0x0
  UINT lun ......................... 0, 0x0
  UINT cdb_lgth .................... 6, 0x6
  UCHAR cdb[MAX_CDB]:
0  12 00 00 00  40 00 00 00  00 00 00 00                ....@.......
  UINT msg_in_lgth ................. 12, 0xC
  UCHAR *msg_in_ptr ................ 0x1FBCCDC
  UCHAR msg_in[MAX_MSG_IN]:
0  00 00 00 00  00 00 00 00  00 00 00 00                ............
  UINT msg_out_lgth ................ 12, 0xC
  UCHAR *msg_out_ptr ............... 0x1FBCCF0
  UCHAR msg_out[MAX_MSG_OUT]:
0  00 00 00 00  00 00 00 00  00 00 00 00                ............
  UINT data_count .................. 1024, 0x400
  UCHAR *data_ptr .................. 0x1FBDCA0
  struct siop_struc *link_ptr ..... 0x0

<CR> for more, 'q' or '.' to quit, 'x' to exit: .
F/W Tools -->
```

3. Send the command to the Firmware by selecting menu item **3**.

```
F/W Tools --> 3
Control structure [0] (0):
siop_cmd status: 0x800
Return status: 2048 (0x800)
F/W Tools -->
```

Note the returned status of 0x800 (Data Underrun). This status is received because we gave the Firmware a buffer of size 1024 bytes but only requested 0x40 bytes from the device. This is an expected and acceptable status.

4. Select menu item **7** to display the data buffer to see the inquiry data.

```
F/W Tools --> 7
Data buffer [0, 1] (0):
Offset into buffer (0):
Number of bytes to display (1024, 0x400): 0x40


db[0]:  First address = 0x1fbdca0  Last address = 0x1fbe09f


 0  00 00 01 01  1F 00 00 00  48 50 20 20  20 20 20 20  ........HP
10  39 37 35 34  38 53 20 20  20 20 20 20  20 20 20 20  97548S
20  38 39 31 36  44 41 44 41  44 41 44 41  44 41 44 41  8916DADADADADADA
30  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
F/W Tools -->
```

Note the fill pattern of **DADA** starting at offset 0x24 (assuming the previous SDL steps were followed to block fill (**BF**) the buffer). This is because we requested 0x40 bytes in the CDB, but the device only returned 0x00 through 0x23 bytes, the amount of inquiry data specified for this device.

Mode Sense Parameters

This exercise shows how to get the current device's mode page parameters. This example senses the *current* page 3 parameters; however, *default, changeable*, and *saved* parameters could be fetched instead as well as other mode pages.

1. Select menu item **1** to build a Firmware control structure. Note that most of the parameters can be left at their default values. Also note that the data_ptr parameter is reset back to the start of data buffer 0. Once the last parameter that requires a change has been changed, the **"."** command can be issued to return the command prompt.


```
F/W Tools --> 1
Control structure [0] (0):
  UINT  user_defined ............... (0):
  UINT cmd_ctrl .................... (-2147483635): 0x8000000D
  UINT addr_ilvl ................... (0):
  UINT lun ......................... (0):
  UINT cdb_lgth .................... [0 - 12] (6):
    UCHAR cdb[0] ................... (18): 0x1a
    UCHAR cdb[1] ................... (0): 0
    UCHAR cdb[2] ................... (0): 3
    UCHAR cdb[3] ................... (0): 0
    UCHAR cdb[4] ................... (64): 0xff
    UCHAR cdb[5] ................... (0): 0
  UINT msg_in_lgth ................. (7):
  UCHAR *msg_in_ptr ................ (0x1FBCCE1):
    UCHAR msg_in[0] ................ (1):
    UCHAR msg_in[1] ................ (3):
    UCHAR msg_in[2] ................ (1):
    UCHAR msg_in[3] ................ (63):
    UCHAR msg_in[4] ................ (8):
    UCHAR msg_in[5] ................ (0):
    UCHAR msg_in[6] ................ (0):
  UINT msg_out_lgth ................ (6):
  UCHAR *msg_out_ptr ............... (0x1FBCCF0):
    UCHAR msg_out[0] ............... (192):
    UCHAR msg_out[1] ............... (1):
    UCHAR msg_out[2] ............... (3):
    UCHAR msg_out[3] ............... (1):
    UCHAR msg_out[4] ............... (25):
    UCHAR msg_out[5] ............... (8):
  UINT data_count .................. (988):
  UCHAR *data_ptr .................. (0x1FBDCC4): 0x1fbdca0
  struct siop_struc *link_ptr ..... (0): .
F/W Tools -->
```

2. Select menu item **6** to display the command structure.

```
F/W Tools --> 5
Control structure [0] (0):

Address of siop_struc structure: 0x1FBCCB4
Contents of siop_struc structure:
  UINT  user_defined ............... 0x0
  UINT cmd_ctrl .................... 0x8000000D
  UINT addr_ilvl ................... 0, 0x0
  UINT lun ......................... 0, 0x0
  UINT cdb_lgth .................... 6, 0x6
  UCHAR cdb[MAX_CDB]:
0  1A 00 03 00  FF 00 00 00  00 00 00 00                ............
  UINT msg_in_lgth ................. 7, 0x7
  UCHAR *msg_in_ptr ................ 0x1FBCCE1
  UCHAR msg_in[MAX_MSG_IN]:
0  01 03 01 3F  08 00 00 00  00 00 00 00                ...?........
  UINT msg_out_lgth ................ 6, 0x6
  UCHAR *msg_out_ptr ............... 0x1FBCCF0
  UCHAR msg_out[MAX_MSG_OUT]:
0  C0 01 03 01  19 08 00 00  00 00 00 00                ............
  UINT data_count .................. 988, 0x3DC
  UCHAR *data_ptr .................. 0x1FBDCA0
  struct siop_struc *link_ptr ....... 0x0

<CR> for more, 'q' or '.' to quit, 'x' to exit: .
F/W Tools -->
```

3. Select menu item **3** to send the command to the Firmware.

```
F/W Tools --> 3
Control structure [0] (0):
siop_cmd status: 0x800
Return status: 2048 (0x800)
F/W Tools -->
```

4. Select menu item **7** to display the data buffer and examine the mode sense parameters.

```
F/W Tools --> 7
Data buffer [0, 1] (0):
Offset into buffer (0):
Number of bytes to display (1024, 0x400): 0x60

db[0]:  First address = 0x1fbdca0  Last address = 0x1fbe09f

 0  23 00 00 08  00 00 00 00  00 00 02 00  03 16 00 01  #...............
10  00 01 00 00  00 8C 00 38  02 00 00 01  00 0C 00 12  .......8........
20  40 00 00 00  44 41 44 41  44 41 44 41  44 41 44 41  @...DADADADADADA
30  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
40  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
50  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
F/W Tools -->
```

Issue a Read Command to the Device

1. Hit the ABORT switch on the front panel of the target CPU and use the memory modify (**MM**) command to fill in the contents of data buffer 0.

```
F/W Tools -->
Exception: Interrupt Level 7 (ABORT Switch)
Vector Number =1, Address =008
SXIP =0001A1E4 SNIP =0001A1E2 TPSR =800003F0
IP   =0001A1E0 CR02 =800003F0 CR07 =0005D000 R00  =00000000
R01  =DEADDEAD R02  =00000000 R03  =00000000 R04  =00010000
R05  =01F76000 R06  =00000000 R07  =00000000 R08  =00000000
R09  =00000000 R10  =00000000 R11  =00000000 R12  =00000000
R13  =00000000 R14  =00000000 R15  =00000000 R16  =00000000
R17  =00000000 R18  =00000000 R19  =00000000 R20  =00000000
R21  =00000000 R22  =00000000 R23  =00000000 R24  =00000000
R25  =00000000 R26  =00000000 R27  =00000000 R28  =00000000
R29  =00000000 R30  =00000000 R31  =01FFEDA0
0001A1E0 C0000000  BR          $0001A1E0
187-Bug>bf 1fbdca0 1fbdca0+400 0
Effective address: 01FBDCA0
Effective address: 01FBE09F
187-Bug>md 1fbdca0:10
01FBDCA0  00000000 00000000 00000000 00000000  ................
01FBDCB0  00000000 00000000 00000000 00000000  ................
01FBDCC0  00000000 00000000 00000000 00000000  ................
01FBDCD0  00000000 00000000 00000000 00000000  ................
187-Bug>
```

2. Issue the go (**G**) command to the MVME167/187 ROM Debugger and hit an **h** to repaint the **t167** screen.

3. Build the command structure for a 10-byte read.

```
F/W Tools --> 1
Control structure [0] (0):
  UINT  user_defined ..............  (0):
  UINT cmd_ctrl ...................  (-2147483635): 0x8000000D
  UINT addr_ilvl ..................  (0):
  UINT lun ........................  (0):
  UINT cdb_lgth ...................  [0 - 12] (6): 0xa
    UCHAR cdb[0] ..................  (26): 0x28
    UCHAR cdb[1] ..................  (0):
    UCHAR cdb[2] ..................  (3): 0
    UCHAR cdb[3] ..................  (0):
    UCHAR cdb[4] ..................  (255): 0
    UCHAR cdb[5] ..................  (0):
    UCHAR cdb[6] ..................  (0):
    UCHAR cdb[7] ..................  (0):
    UCHAR cdb[8] ..................  (0): 2
    UCHAR cdb[9] ..................  (0):
  UINT msg_in_lgth ................  (7):
  UCHAR *msg_in_ptr ...............  (0x1FBCCE1):
    UCHAR msg_in[0] ...............  (1):
    UCHAR msg_in[1] ...............  (3):
    UCHAR msg_in[2] ...............  (1):
    UCHAR msg_in[3] ...............  (63):
    UCHAR msg_in[4] ...............  (8):
    UCHAR msg_in[5] ...............  (0):
    UCHAR msg_in[6] ...............  (0):
  UINT msg_out_lgth ...............  (6):
  UCHAR *msg_out_ptr ..............  (0x1FBCCF0):
    UCHAR msg_out[0] ..............  (192):
    UCHAR msg_out[1] ..............  (1):
    UCHAR msg_out[2] ..............  (3):
    UCHAR msg_out[3] ..............  (1):
    UCHAR msg_out[4] ..............  (25):
    UCHAR msg_out[5] ..............  (8):
  UINT data_count .................  (952): 0x400
  UCHAR *data_ptr .................  (0x1FBDCC4): 0x1fbdca0
  struct siop_struc *link_ptr .....  (0): .
F/W Tools -->
```

4. Select menu item **5** to display the command structure.

```
F/W Tools --> 5
Control structure [0] (0):

Address of siop_struc structure: 0x1FBCCB4
Contents of siop_struc structure:
  UINT  user_defined ................ 0x0
  UINT cmd_ctrl ..................... 0x8000000D
  UINT addr_ilvl .................... 0, 0x0
  UINT lun .......................... 0, 0x0
  UINT cdb_lgth ..................... 10, 0xA
  UCHAR cdb[MAX_CDB]:
0  28 00 00 00  00 00 00 00  02 00 00 00                (...........
  UINT msg_in_lgth .................. 7, 0x7
  UCHAR *msg_in_ptr ................. 0x1FBCCE1
  UCHAR msg_in[MAX_MSG_IN]:
0  01 03 01 3F  08 00 00 00  00 00 00 00                ...?........
  UINT msg_out_lgth ................. 6, 0x6
  UCHAR *msg_out_ptr ................ 0x1FBCCF0
  UCHAR msg_out[MAX_MSG_OUT]:
0  C0 01 03 01  19 08 00 00  00 00 00 00                ............
  UINT data_count ................... 1024, 0x400
  UCHAR *data_ptr ................... 0x1FBDCA0
  struct siop_struc *link_ptr ....... 0x0

<CR> for more, 'q' or '.' to quit, 'x' to exit: .
```

5. Issue the Firmware command to do the read.

```
F/W Tools --> 3
Control structure [0] (0):
siop_cmd status: 0x0
F/W Tools -->
```

6. Select menu item **7** to display the data buffer contents.

```
F/W Tools --> 7
Data buffer [0, 1] (0):
Offset into buffer (0):
Number of bytes to display (1024, 0x400): 0x60
 0  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
10  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
20  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
30  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
40  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
50  44 41 44 41  44 41 44 41  44 41 44 41  44 41 44 41  DADADADADADADADA
F/W Tools -->
```

Glossary

Term	Definition
CDB (NOTE)	Command Descriptor Block - A defined SCSI structure used to communicate commands from an initiator to a target.
DISconnect (NOTE)	The function that occurs when an SCSI target releases control of the SCSI bus, allowing it to go to the BUS FREE phase.
Initiator (NOTE)	An SCSI device (usually a host system) that requests an I/O process to be performed by another SCSI device (a target).
Logical unit (NOTE)	A physical or virtual peripheral device addressable through a target.
LUN (NOTE)	Logical Unit Number.
MPU	Microprocessor Unit (MC68040 on the MVME162, 166, and 167; MC88100 on the MVME187;MC88110 on the MVME197).
Nexus (NOTE)	An SCSI bus relationship that begins with the establishment of an initial connection and ends with the completion of the I/O process. The relationship may be restricted to specify a single logical unit or target routine by the successful transfer of an "*identify*" message. The relationship may be further restricted by the successful transfer of a queue tag message.
NVRAM	The non-volatile RAM contained in the MK48T08 chip on the Single Board computer (SBC). Also called Battery Backed-up RAM (BBRAM).
Peripheral device	A physical peripheral device that can be attached to an SCSI device, which connects to the SCSI bus. The peripheral device and the SCSI device (peripheral controller) may be physically packaged together. Often there is a one-to-one mapping between peripheral devices and logical units, but this is not required. Examples of peripheral devices are: magnetic disks, printers, optical disks, and magnetic tapes.
Reselect (NOTE)	The SCSI function that occurs when a target selects an initiator to continue an operation after a disconnect.
SCRIPTS	NCR SCSI SCRIPTS code.

Term	Definition
SCSI (NOTE)	Small Computer System Interface - An ANSI-defined intelligent I/O bus.
SCSI-2 (NOTE)	An ANSI-defined intelligent I/O bus with broader functional definitions and tighter protocol definitions than SCSI-1 (SCSI).
SCSI address (NOTE)	The hexadecimal representation of the unique address (0 through 7) assigned to an SCSI device.
SCSI device (NOTE)	A host adapter or a target controller that can be attached to the SCSI bus.
SCSI ID (NOTE)	The bit-significant representation of the SCSI address referring to one of the data lines (DB0 - DB7).
SCSI status	During the SCSI STATUS phase, one byte of information sent from the target to the initiator upon completion of each SCSI command.
SDL	SCSI Driver Library. (It is released as unsupported software to be used for example purposes only.)
Select (NOTE)	The SCSI function that occurs when an initiator selects a target to perform an I/O request.
SIOP	NCR 53C710 SCSI I/O Processor.
t167	An example test program which runs on the MVME167 or other SBCs. (It is released as unsupported software to be used for example purposes only.)
Tagged command queuing (NOTE)	An SCSI command control flow implementation that allows the target to receive multiple command requests from the same initiator before returning status for any of the command requests.
Target (NOTE)	An SCSI device that performs an operation requested by an initiator.
Thread	A physical or logical connection between a target and an initiator.

Note **These terms are derived from the SCSI specification. Refer to it for the authoritative definitions.**

Index

T

U

W

X

INDEX