SCRAMNet® Network

Windows NT® DLL Reference Guide

Document No. C-T-ML-NTDLL###-A-0-A7

FOREWORD

Revised: February 16, 2000

Systran Corporation
4126 Linden Avenue
Dayton, OH 45432-3068 USA
(800) 252-5601

TABLE OF CONTENTS

APPENDICES

TABLES

This page intentionally left blank

1. INTRODUCTION

1.1 How To Use This Manual

1.1.1 Purpose

The SCRAMNet Windows NT Dynamic Link Library (DLL) software is a collection of basic functions designed to assist Windows NT application programmers with application development. This set of high-level routines provides all needed SCRAMNet hardware access.

1.1.2 Scope

This document:

- Defines a set of software routines to assist in configuration and application program development for the SCRAMNet general-purpose node USING Windows NT DLL.
- Describes the basic operations of the SCRAMNet Network Interface DLL routines for all of the supported host platforms using the SCRAMNet Network hardware.
- Explains how to use the SCRAMNet DLL routines, assists in the general node setup; interrupt initialization, managing data-flow control, and accessing of general network-board information.

This manual is intended for system and software developers who want to call the exported DLL routines in applications programming. The reader must understand the routine's capability and have an understanding of the SCRAMNet product to use the routines provided with SCRAMNet effectively.

1.1.3 Style Conventions

- Called functions are italicized and are followed by a set of open and closed parentheses: *OpenConnect()*
- File names and function parameters are bolded; for example, **config.c**
- Directory names, and path names are italicized; for example, *c:\winnt\system32*.
- Hexadecimal values in normal text are written with the word hex italicized and following the value with a font size one smaller than the context: FB001040 *hex*
- Code and monitor screen displays of input and output are boxed and indented on a separate line.

```
/* EXAMPLE */
new_ptr = (unsigned long huge*)old_ptr;
```

- Large samples of code are Courier font, at least one size less than context, and are usually on a separate page.

1.2 Related Documentation

SCRAMNet *Network (bus/platform) Hardware Interface Reference Manual (D-T-MR-XXXXXXXX)* - This manual describes the features, function, installation, and operation of each Systran network board.

SCRAMNet *Network Software Installation Manual for the (bus/platform) Using (Operating System) (C-T-MI-XXXXXXXX)* - This manual provides information concerning automatic and manual installation options for each product/platform and for a variety of operating systems.

SCRAMNet *Network Utilities Manual (C-T-MU-UTIL)* - This manual describes the use of Systran hardware, diagnostic software, SCRAMNet+ EEPROM initialization software, and the SCRAMNet Monitor software.

SCRAMNet *Network Windows Utilities Manual (C-T-MU-WNUTIL)* - This manual describes the memory monitor programs for Windows: **scrmon** and **winmon**.

SCRAMNet *Network Programmer's Reference Guide (C-T-ML-PROGREF)* – this manual describes the basic operations of the SCRAMNet Network Interface Library routines for all of the supported host platforms using the SCRAMNet Network hardware.

SCRAMNet *Network Windows NT Utilities Manual (C-T-MU-NTUTIL)* - This manual describes the basic features of the SCRAMNet Windows NT utilities: network monitor, interrupt testing, diagnostics, and EEPROM initialization.

SCRAMNet *Network Windows DLL Reference Guide* (C-T-ML-WINDLL) - This manual defines a set of user routines that are exported by the Windows Dynamic Link Library (DLL) to aid in application development.

☞ **NOTE**: "XXXXXXXX" in the document number is the product ID. (e.g., VME6U, PCI, PMC, EISA, etc.)

1.3 Quality Assurance

Systran Corporate policy is to provide our customers with the highest quality products and services. In addition to the physical product, the company provides documentation, sales and marketing support, hardware and software technical support, and timely product delivery. Our quality commitment begins with product concept, and continues after receipt of the purchased product.

Systran's Quality System conforms to the ISO 9001 international standard for quality systems. ISO 9001 is the model for quality assurance in design, development, production, installation and servicing. The ISO 9001 standard addresses all 20 clauses of the ISO quality system and is the most comprehensive of the conformance standards.

Our Quality System addresses the following basic objectives:

- Achieve, maintain and continually improve the quality of our products through established design, test, and production procedures.
- Improve the quality of our operations to meet the needs of our customers, suppliers, and other stakeholders.
- Provide our employees with the tools and overall work environment to fulfill, maintain, and improve product and service quality.

- Ensure our customer and other stakeholders that only the highest quality product or service will be delivered.

The British Standards Institution (BSI), the world's largest and most respected standardization authority, assessed Systran's Quality System. BSI's Quality Assurance division certified we meet or exceed all applicable international standards, and issued Certificate of Registration, number FM 31468, on May 16, 1995. The scope of Systran's registration is: "Design, manufacture and service of high technology hardware and software computer communications products." The registration is maintained under BSI QA's bi-annual quality audit program.

Customer feedback is integral to our quality and reliability program. We encourage customers to contact us with questions, suggestions, or comments regarding any of our products or services. We guarantee professional and quick responses to your questions, comments, or problems.

1.4 Technical Support

Technical documentation is provided with all of our products. This documentation describes the technology, its performance characteristics, and includes some typical applications. It also includes comprehensive support information, designed to answer any technical questions that might arise concerning the use of this product. We also publish and distribute technical briefs and application notes that cover a wide assortment of topics. Although we try to tailor the applications to real scenarios, not all possible circumstances are covered.

Although we have attempted to make this document comprehensive, you may have specific problems or issues this document does not satisfactorily cover. Our goal is to offer a combination of products and services that provide complete, easy-to-use solutions for your application.

If you have any technical or non-technical questions or comments (including software), contact us. Hours of operation are from 8:00 a.m. to 5:00 p.m. Eastern Standard/Daylight Time.

- Phone: **(937) 252-5601** or **(800) 252-5601**
- E-mail: **support@systran.com**
- Fax: **(937) 252-1349**

1.5 Ordering Process

To learn more about Systran products or to place an order, please use the following contact information. Hours of operation are from 8:00 a.m. to 5:00 p.m. Eastern Standard/Daylight Time.

- Phone: **(937) 252-5601** or **(800) 252-5601**
- E-mail: i**nfo@systran.com**
- World Wide Web address: **www.systran.com**

This page intentionally left blank

2. PRODUCT OVERVIEW

2.1 Overview

Using high-level routines to perform all hardware access to the SCRAMNet board means that a single application can work with different types of SCRAMNet hardware by simply using a different hardware-dependent DLL for each different type of SCRAMNet board. For example, develop an application with a SCRAMNet ISA board, and then run the same application on another system with a PCI board using the PCI version of the SCRAMNet DLL.

2.2 Organization

Any user-written, high-level language program following the steps necessary to call DLL functions can access these routines. The various functions are divided into four categories: Configuration, Data Flow, Memory Access and General.

2.2.1 Configuration Routines

get_scr_node_id()	READ the current network node id
get_scr_phy_csr_addr()	Get physical base address of the SCRAMNet CSR registers
get_scr_phy_mem_addr()	Get physical base address of the SCRAMNet network memory
get_scr_time_out()	READ current network time out value
get_scr_user_mem_size()	READ current memory size for application to use
scr_acr_read()	READ ACR Location
scr_acr_write()	WRITE ACR Location
scr_brd_select()	Select a SCRAMNet board for control (Multiple board support only)
scr_board_status()	Create message windows with current SCRAMNet configuration data displayed in it
scr_mem_mm()	Maps the SCRAMNet physical memory
scr_probe_mm()	Validates SCRAMNet memory mapping information
scr_reg_mm()	Maps the SCRAMNet Control Status Registers
scr_reset_mm()	Reset SCRAMNet Node
sp_cfg_read()	READ config file
sp_plus_find()	Find Board Type
sp_scram_init()	READ config and map registers and memory
sw_get_int()	Get the current driver interrupt number
sw_int_connect()	Initialize interrupt Operations
sw_int_disconnect()	Terminate interrupt Operations
sw_mem_addr()	Return physical address of SCRAMNet memory
sw_net_to()	Return network time-out (as stored in registry)
sw_reg_addr()	Return physical address of CSRs
sw_set_size()	Set mapped memory size to use upon driver initialization
sw_set_int()	Set the interrupt number to use upon driver installation
sw_user_size()	Return user-specified memory size (as stored in registry)

2.2.2 Data Flow Control Routines

GetScrTransactionType()	Obtain method currently being used to alter the byte order
scr_bswp_mm()	Byte Swapper control switch
scr_dfltr_mm()	Data Filtering control switch
scr_lnk_mm()	SCRAMNet node linker switch
scr_wml_mm()	Host WRITE Disable switch
SetScrTransactionType()	Set byte-swapping method to be used
sp_gtm_mm()	Return current transaction mode (PCI/PMC Only)
sp_stm_mm()	Set transaction mode (PCI/PMC Only)

2.2.3 Interrupt Routines

scr_acr_mm()	Auxiliary Control RAM control switch
scr_int_mm()	Interrupts Mode Select

2.2.4 Memory Access Routines

get_base_mem()	Obtain SCRAMNet memory pointer
ReadSCRByte()	READ a byte from memory
ReadSCRLong()	READ a longword from memory
ReadSCRWord()	READ a word from memory
WriteSCRByte()	WRITE a byte to memory
WriteSCRLong()	WRITE a longword to memory
WriteSCRWord()	WRITE a word to memory

2.2.5 DMA Routines

scr_dma_read()	DMA READ from SCRAMNet memory (PCI only)
scr_dma_write()	DMA WRITE into SCRAMNet memory (PCI only)

2.2.6 General Routines

scr_csr_read()	READ CSR values
scr_csr_write()	WRITE values to CSRs
scr_error_mm()	Checks for and explains errors defined by CSR1
scr_fifo_mm()	Error Register Status Dump/FIFO Reset
scr_fswin_mm()	Check for fiber optic switch
scr_id_mm()	Node Identification information
scr_load_mm()	Configuration File Loader
scr_mclr_mm()	Memory or ACR clear control
scr_read_int_fifo()	READ interrupt FIFO CSRs
scr_rw_mm()	Control Status Register control
scr_save_mm()	Configuration File Saver
scr_smem_mm()	Memory or ACR modifier
sp_bist_rd()	READ BIST data
sp_mem_size()	Get memory size of board
sp_msg_life()	Set Message Life
sp_net_to()	Set Network Timeout
sp_protocol()	Set Network Protocol
sp_rx_id()	Set Receiver ID
sp_set_cntr()	Set General Purpose Counter
sp_set_sm_addr()	Set Physical Memory Address
sp_set_vp()	Set Virtual Page
sp_txrx_id()	Set Transmitter and Receiver ID

2.3 Related Documentation

SCRAMNet *Network (bus/platform) Hardware Interface Reference Manual (D-T-MR-XXXXXXXX)* - This manual describes the features, function, installation, and operation of each Systran network board.

SCRAMNet *Network Software Installation Manual for the (bus/platform) Using (Operating System) (C-T-MI-XXXXXXXX)* - This manual provides information concerning automatic and manual installation options for each product/platform and for a variety of operating systems.

SCRAMNet Network Programmer's Reference Guide (C-T-MR-PROGREF) - This manual describes the basic operations of the SCRAMNet Network Interface Library routines for all of the supported host platforms using the SCRAMNet Network hardware. This manual is a guide to understanding the library routines and how they relate to the SCRAMNet Network control/status registers.

SCRAMNet *Network Utilities Manual (C-T-MU-UTIL)* - This manual describes the use of Systran hardware, diagnostic software, SCRAMNet+ EEPROM initialization software, and the SCRAMNet Monitor software.

SCRAMNet *Network Windows Utilities Manual (C-T-MU-WNUTIL)* - This manual describes the memory monitor programs for Windows: **scrmon** and **winmon**.

SCRAMNet *Network Windows NT Utilities Manual (C-T-MU-NTUTIL)* - This manual describes the basic features of the SCRAMNet Windows NT utilities: network monitor, interrupt probe, diagnostics, and EEPROM initialization.

SCRAMNet *Network Windows DLL Reference Guide* (C-T-ML-WINDLL) - This manual defines a set of user routines that are exported by the Windows Dynamic Link Library (DLL) to aid in application development.

☞ **NOTE**: "XXXXXXXX" in the document number is the product ID. (e.g., VME6U, PCI, PMC, EISA, etc.)

This page intentionally left blank

3. ACCESSING SCRAMNET HARDWARE

3.1 Mapping SCRAMNet to the Host System

The first step in programming the SCRAMNet board is to map the hardware into the host system as a physical device. This process is specific to each host platform environment. The SCRAMNet Network Interface Library has routines that perform this physical mapping for each supported host environment. The SCRAMNet DLL calls will appear the same on all platforms, with respect to the user's calling application. In reality, the code that performs the physical mapping is unique to the host platform, but the application program interface is the same on all the platforms.

3.1.1 Unique Physical Interfaces

The SCRAMNet node has three physical interfaces unique to the host system: memory, registers, and hardware interrupts. Each one of these interfaces must be recognized by the host system to operate SCRAMNet properly. There are several SCRAMNet Network Interface Library routines that perform these mapping functions. The SCRAMNet Network Interface Library contains the *sp_scram_init()* routine which provides all of the SCRAMNet mapping functions in one call.

3.1.2 Mapping SCRAMNet Hardware Registers

Mapping the SCRAMNet hardware registers is the first and most crucial operation to be performed. Access to register space varies across systems. After the SCRAMNet register space is mapped, by calling the *sp_scram_init()* routine, access to SCRAMNet registers is provided by a pair of functions that allow you to READ and WRITE these registers. READs and WRITEs to the registers should always be performed by calling the *scr_csr_read()* and *scr_csr_write()* functions.

3.1.3 Mapping SCRAMNet Memory

Once the registers are successfully mapped, the SCRAMNet memory must be mapped into the host system address space. The SCRAMNet on-board memory is not considered to be system memory by any means. The memory is only accessed as a physical memory device with memory mapped into an address space that does not coincide with the system memory address space. Mapping SCRAMNet memory is accomplished by calling the *sp_scram_init()* function.

Upon successful completion of this function, a global pointer within the SCRAMNet DLL is initialized to the address of the first location of SCRAMNet memory. The value of this pointer can be obtained by the application program by calling the *get_base_mem()* function.

3.1.4 Mapping Multiple SCRAMNet devices

The SCRAMNet Windows NT device driver NTPCPC2, version A4 or later, supports multiple SCRAMNet devices. This permits running multiple nodes from a single user application. All SCRAMNet API function calls default to board zero. Board numbers are assigned in the order of scanning the PCI bus. To gain access to other boards on the same system, the API function *scr_brd_select(i)* switches control to board *i*, passed as the function argument. After calling this function, all SCRAMNet API calls are performed on board *i*. If board *i* is not found on the system, *scr_brd_select(i)* returns a '-1' and control remains with the previously selected board.

3.2 Accessing SCRAMNet Registers from the Host System

The SCRAMNet hardware operation is defined and controlled by the on-board register set. SCRAMNet data is transferred between nodes via on-board memory. All of the SCRAMNet data transfer options are available to the application programmer by enabling and/or disabling specific bits in the control/status registers (CSR). The host platform environment, consisting of the bus being used (e.g., VME, EISA, SBus), the operating system (e.g., Windows, UNIX, DOS, VxWorks), and the compiler, define how access is provided to the SCRAMNet register space. The routines *scr_csr_read()* and *scr_csr_write()* provide access to these registers through software.

These routines are coded to provide the proper interface required on each system supported. For example, on many PCI platforms it is necessary to map the SCRAMNet registers to one of the available PCI address spaces in which the system returns a pointer to a virtual address that is mapped directly to the physical location of the SCRAMNet registers. On a DOS EISA platform the registers are mapped to the I/O bus, which is provided solely for hardware device register access. The physical slot in which the board is inserted defines the beginning I/O address on this system. On a system such as this, register access is performed by a special IN and OUT machine instruction just for the I/O bus.

3.2.1 Physical Address Space Requirements

The SCRAMNet register set requires 64 bytes of physical address space on all of the various host systems, except PCI systems. In order to prevent prefetching from affecting the registers, the PCI version of the SCRAMNet Network distributes its registers along 4 kilobytes of address space. This address space used cannot be overlapped with any other type of device being mapped into the same area.

On most platforms there are 32 bytes of actual SCRAMNet registers, which are defined as 16-bit-wide registers. SCRAMNet registers can be read and written as 8-, 16- or 32-bit values. The library routines READ (*scr_csr_read()*) and WRITE (*scr_csr_write()*) the registers as 16-bit values.

3.2.2 CSR Definition

The definition and function of almost all of these registers is the same on all platforms. There may be subtle differences in these definitions on different platforms, but these differences only apply to the platform specific issues, such as the interrupt-vector registers. The specific definitions of each bit in each register are described in detail in the SCRAMNet *Network (bus) Hardware Interface Reference Manual* for the host platform.

3.3 Accessing SCRAMNet Memory from the Host System

Once the registers and memory are successfully mapped into the host system environment, it is possible to access the on-board SCRAMNet memory space. If configured correctly (through on-board registers, and the host system physical memory mapping requirements), the SCRAMNet on-board memory will be accessible via READs and WRITEs from the host. If inserted into the ring, any WRITEs to this memory will reflect the same data at the same location to each node in the ring. If Virtual Paging is used, the location will be offset. For example, a 2 MB board can page to 0, 2, 4, or 6 MB address.

3.3.1 Array Style Access

After successfully mapping the SCRAMNet memory into the host system address space, the calling application can initialize a pointer to SCRAMNet memory by calling the *get_base_mem()* function. This function will return a pointer to an unsigned long (32-bit) integer. This pointer can then be cast to the desired data size (8-, 16- or 32-bit) to be used for accessing the SCRAMNet memory. Reading or writing the memory is equivalent to reading or writing an element in an array. For example, to WRITE and then READ a data value in the first longword of SCRAMNet memory, would look something like this in 'C':

```
/* EXAMPLE */
unsigned long int * Lmem_ptr;

Lmem_ptr = get_base_mem();
/* to write */
Lmem_ptr[0] = 0x12345678;
/* to read */
scram_value = Lmem_ptr[0];
```

After this statement executes, the value of '12345678' *hex* will be present at offset 0 (or the Virtual Paging offset) on every on-line node in the network. Since the size of SCRAMNet memory is finite, it is very important to pay close attention to the limits of the indices so they do not extend beyond SCRAMNet memory. Different platforms will react differently to this situation. A PCI or ISA based PC-Clone may give a General Protection Fault or no indication at all, depending on the operating system environment. The limit of the index used is dependent on, first, the physical-memory size of the SCRAMNet hardware, and second, the data size of the pointer being used.

```
/* EXAMPLE */
unsigned long int * Lmem_ptr;
unsigned short int * Smem_ptr;
unsigned char * Bmem_ptr;
/* SCRAMNet H/W size */
/* 128 KB = 0x20000 */
Bmem_ptr[0x20000] = 0x5a;          /* ERROR */
Bmem_ptr[0x1FFFF] = 0x5a;          /* OK */
Smem_ptr[0x1FFFF] = 0x1234;        /* ERROR */
Smem_ptr[0xFFFE] = 0x1234;         /* OK */
Lmem_ptr[0x8000] = 0x12345678;     /* ERROR */
Lmem_ptr[0x7FFC] = 0x12345678;     /* OK */
```

When using array style indexing with 16-bit compilers, compile the code using the huge-memory model if addressing more than 64 KB of SCRAMNet memory. If it is necessary to compile the application using a different memory model, choose the memory access functions that work through the DLL (which compiles in the huge-memory model) for all direct-memory access. When mixing memory models between the application and the DLL, make sure that the SCRAMNet memory pointer(s) are typecast to a "huge" pointer type before they are accessed.

```
/* EXAMPLE */
new_ptr = (unsigned long huge*)old_ptr;
```

3.3.2 Function Oriented Memory Access

After successfully mapping the SCRAMNet memory into the host system address space, the calling application can access any SCRAMNet memory location by calling one of the following functions: *ReadSCRLong(), ReadSCRWord(), ReadSCRByte(), WriteSCRLong(), WriteSCRWord(), or WriteSCRByte().*

For example, to WRITE and then READ a data value in the first longword of SCRAMNet memory, would look something like this in ‘C’:

```
/* EXAMPLE */
unsigned long int lValue;

LValue = 0x12345678;
WriteSCRLong(0,lValue);
ReadSCRLong(0,&lValue);
```

After this statement executes, the value of ‘12345678’ *hex* will be present at offset 0 (or the Virtual Paging offset) on every on-line node in the network. And as with array access, ensure the memory index used does not exceed the memory size.

3.4 Accessing the SCRAMNet Auxiliary Control Ram (ACR)

The SCRAMNet Auxiliary Control Ram (ACR) can be described as a “Shadow RAM” over the normal SCRAMNet shared memory. SCRAMNet memory is organized as 32-bit longwords. At each SCRAMNet longword location there is one (8-bit) byte of ACR. The other three bytes of the same longword are non-existent memory. The ACR and normal shared memory can never be accessed at the same time. The key to accessing the ACR is CSR0[4]. When this bit is set, ACR access is enabled, and normal shared memory is inaccessible from the host system, but is still updated by network traffic. The ACR byte at each SCRAMNet longword location is usually located in the least significant byte location. The other three bytes in the same longword cannot be modified. This is one method of determining the byte location of the ACR byte if there is any question about its location within each longword.

Using the ACR is only necessary if configuring the SCRAMNet node for interrupts, external triggers, or High Performance mode (HIPRO). It is recommended that ACR setup be performed during initialization of the node at startup. Before enabling the ACR, clear CSR2 to zero for SCRAMNet Classic, to disable byte swapping. There is one ACR byte for every longword in SCRAMNet shared memory. Setting the bits in each ACR location will define actions for that shared-memory longword location only. After initializing the desired ACR bytes, clear CSR0[4] returning memory access back to normal SCRAMNet shared memory. When this is done, continue the normal node initialization for network operation.

EXAMPLE:

A given configuration calls for nodes 2 and 3 to generate hardware interrupts whenever node 1 performs a WRITE to the first SCRAMNet longword location. Node 1 enables the ACR and sets bit 1 of the ACR byte (Transmit Interrupt Enable) for SCRAMNet longword 0. After clearing ACR-enable on node 1, enable ACR-enable on nodes 2 and 3. Each of these nodes set Receive Interrupt Enable ACR[0] for SCRAMNet longword 0.

After clearing ACR enable on nodes 2 and 3, and initializing both nodes for host interrupt enable, a WRITE to longword 0 on node 1 generates a network interrupt message. When this message arrives at nodes 2 and 3, longword 0 will be updated, each node will add the address of the interrupting longword location into the Receiver (Interrupt) FIFO and generate a hardware interrupt to the host platform of each node.

If a node 4 was also on the ring, and the Receive Interrupt Enable ACR bit was not set on this node, there would only be an update to the data in longword 0, and no hardware interrupt or FIFO entry.

3.5 Host Platform Byte-Ordering Considerations

(SCRAMNet-LX and SCRAMNet+ only) By default, the SCRAMNet hardware follows the byte-ordering format found on almost all VME based processors, which is considered to be "Big-Endian" byte order. This means that all VME SCRAMNet nodes are Big-Endian byte order and do not have any byte swapping facility on-board. SCRAMNet nodes based on backplanes that may have "Little-Endian" byte ordering (i.e., PCs), will have some facility on-board to allow switching between Big-Endian and Little-Endian.

With respect to SCRAMNet network operation, if a ring is configured with only Little-Endian nodes, it will not be necessary to configure any of the nodes to perform byte-swapping. The data format will be consistent around the ring.

NOTE: If a node utilizing the Big-Endian data format, by default, is introduced into the ring, then it will be necessary to configure all of the Little-Endian nodes to perform byte-swapping from Little-Endian to Big-Endian format.

3.6 Windows NT Programming

CAUTION: It is critical to remember certain requirements when programming in the Windows NT environment, so please read these paragraphs! Source files WILL NOT compile and execute if these directions are not followed.

First, unlike DOS or Unix implementations, the Windows NT SCRAMNet API is implemented in a dynamic link library (DLL), which must be included in the project when compiling source code. The setup program copies a file called **ntpciscr.lib** into the SCRAMNet directory (selected during setup). Include this file in the project build /make file (compiler-dependent). Receipt of errors such as "external routine not found" during the link phase of your compilation, indicates the library file has not been properly included.

Second, much of the original API code was ported directly to Windows NT, but the prototypes for DOS/Unix are different than those for Windows NT. The following terms MUST be defined globally to ensure the proper prototypes and #include directives are compiled:

_WINDOWS
_WINNT

Defining them in the main program source file will most likely NOT be sufficient. Edit your project options/parameters (compiler-dependent) and #define these terms globally. Receipt of errors such as "redefined prototype" or "xyz not defined", indicates these terms have not been correctly defined.

Third, to implement the SCRAMNet API faithfully, all references to multiple SCRAMNet cards in a single computer have been removed. Therefore, if existing drivers and code is based on the Windows 3.11/Windows 95 SCRAMNet specifications, source code must be modified to remove the additional parameter referring to the card number.

Save **ntpciscr.dll** in a standard DLL location, such as *%windir\system32* (typically *c:\winnt\system32* for Windows NT 4.0 or *c:\winnt35\system32* for Windows NT 3.51) or the current directory for the executable file.

The **pnpscr.dll** and **pnpscr.lib** are the newest versions of the SCRAMNet library and DLL files, and are compiled with the **_std** call (**ntpciscr.dll** is compiled with the **_cdecl)**. The **ntpciscr.lib** and **ntpciscr.dll** are older versions of the library and DLL files, and are retained to maintain backward compatibility with previous versions of the Windows NT software.

Although the Windows NT software package was specifically designed for use with C/C++ compilers, it can also be used with Visual Basic. To use a Visual Basic compiler with the SCRAMNet API you must use **pnpscr.dll**. This software package was tested with Microsoft Visual Basic 6.0.8169.

4. INTERRUPTS

4.1 Overview

When discussing SCRAMNet interrupts, it is important to be aware of the difference between hardware interrupts to the host platform and SCRAMNet Network interrupt messages. For the purpose of this discussion, the data transferred around a SCRAMNet network will be referred to as a “message”. Each SCRAMNet node generates messages around the ring that are considered either “data” or “interrupt” messages. The type of message generated is under user-application control. Configuring a node to transfer interrupt messages does not necessarily mean that each interrupt message will generate a hardware interrupt to the host platform. This is optionally controlled by the user’s application. All of the different methods of generating SCRAMNet interrupts, described in section 4.3, may optionally be configured to generate a hardware interrupt to the host platform.

4.2 SCRAMNet Hardware Interrupts

A hardware interrupt is an asynchronous signal, normally generated by a peripheral on the occurrence of an external or internal event, which forces the processor to suspend its normal sequence of operations. The processor control is then transferred to a special routine in memory that services the acknowledged interrupt. In windows programming this routine is typically incorporated into a device driver.

In an environment where more than one source of an interrupt is possible, device drivers must be implemented to deal with every different kind of service required. This is typically accomplished with an interrupt-priority scheme. Interrupts that are to be processed upon receipt would need to be assigned a higher priority than those that can be processed later. For example, in a READ-WRITE operation, a WRITE operation would require a higher priority than a READ to ensure up-to-date READ requests.

4.3 SCRAMNet Network Interrupt Messages

Each SCRAMNet network board on the network ring can be uniquely configured to receive and/or transmit interrupts. SCRAMNet interrupts are generated in two ways:

- A data WRITE to a shared-memory location on the network
- SCRAMNet network errors detected on the local node.

Data-WRITE interrupts on the network can be either “selected” or “forced”. These terms are described in sections 4.4 and 4.5.

Each SCRAMNet host can be “armed” to process interrupts. If the host is to process interrupts, set Interrupts Armed CSR1[14]. Then, upon receipt of either a selected or forced interrupt, CSR5 will contain the most significant seven bits of the 23-bit interrupt address, and the remaining 16 bits are placed in CSR4. Each word read from CSR4 and CSR5 will contain the SCRAMNet memory address of the data received from the network interrupt; a byte address from the beginning of SCRAMNet memory.

Every interrupt address received, including those received while another interrupt is being processed, is placed on the Interrupt FIFO stack.

The device driver uses this information to process the interrupt (the process defined for each interrupt type is user-dependent). Upon completion, the host processor READs the CSRs to get the next interrupt memory address until the Interrupt FIFO Not Empty CSR5[15] is zero, indicating that all the interrupts received thus far have been serviced. To re-enable interrupts, Interrupts Armed CSR1[14] must be set by writing any value to CSR1. Sample Interrupt Service Routine algorithms can be found in the SCRAMNet *Hardware Reference Manual* for the targeted host platform.

☞ **NOTE**: The Interrupt FIFO can hold up to 1,024 interrupt addresses. This means the count of unprocessed interrupts following 1024 is not maintained, however the related data is not lost. After 1,024, the data related to an unprocessed interrupt is written to memory and the 1,024th Interrupt Address is overwritten in the Interrupt FIFO.

Handling hardware interrupts from the SCRAMNet Node is specific to each host platform supporting SCRAMNet. The SCRAMNet CSRs are defined the same across all backplanes except for the following registers: CSR6, CSR7, CSR10, CSR11, CSR14, CSR15 and CSR16. These registers are used for host-platform specific functions such as hardware interrupts.

EXAMPLES

On a VME SCRAMNet node, CSR6 is loaded with a VME Interrupt Vector for handling network interrupts. CSR7 is loaded with a VME Interrupt Vector for handling error interrupts. On an SBus SCRAMNet node CSR6 is not used. CSR7 and CSR15 are used for all of the SBus hardware interrupt configuration values.

4.4 Selected Interrupts

The selected interrupt technique allows the options to either receive and/or transmit interrupts at a particular SCRAMNet shared memory location. This is accomplished with the proper configuration of the ACR[0] and ACR[1], respectively.

Table 4-1 ACR Functions

Bit	Function
0	Receive Interrupt Enable (RIE)
1	Transmit Interrupt Enable (TIE)
2	External Trigger 1 (Host READ/WRITE)
3	External Trigger 2 (Network WRITE)
4	HIPRO Location Enable
5	Reserved
6	Reserved
7	Reserved

Setting Receive Interrupt Enable ACR[0] generates an interrupt to the host for network interrupt data received at a particular shared-memory location. Setting Transmit-Interrupt Enable ACR**[**1] generates an interrupt to the network for a host WRITE to a particular shared-memory location. To assert the two previous interrupt mode bits, you would have to WRITE to the ACR.

Use the following steps to WRITE to the ACR:

- WRITE a ‘0000’ *hex* to CSR2 to clear the register
- WRITE a ‘0010’ *hex* into CSR0 to enable the ACR (CSR0[4])
- Depending on the required interrupt cycle, either WRITE a ‘01’ *hex* for receive or ‘02’ *hex* for a transmit.

Clearing CSR2 is required to disable various features while accessing the ACR. It is important to follow the sequence, that is, clear CSR2 first, and then enable the ACR-enable bit in CSR0.

Once the Auxiliary Control RAM is enabled, the least significant byte of every shared-memory location responds as an ACR byte. Therefore, every shared-memory location has an ACR associated with it to transmit and/or receive interrupts. Clear ACR-Enable CSR0[4] after it is configured, so that every memory location returns to its normal setup. This does not affect any ACR interrupt locations.

Setting Override Receive Interrupt Enable CSR0[6] and Override Transmit Interrupt Enable CSR0[9], overrides any configuration set through the ACR regardless of the status of the ACR Interrupt bits. Setting these two bits forces every shared-memory location to behave as though the ACR bit is set, whether or not it is actually set. Also, with data filtering enabled, interrupts are still generated at memory locations where data values have not actually changed.

☞ **NOTE**: The host processor has to be armed in order to generate hardware interrupts; otherwise no interrupt sequences are possible. Set Interrupts Armed CSR1[14], set Host Interrupt Enable (HIE) CSR0[3] and Interrupt Memory Mask Match (IMME) CSR0[5]. This is equivalent to writing ‘0028’ *hex* into CSR0. If Interrupts Armed is zero, then the host will receive no interrupts. To return to an active status, re-enable interrupts by rewriting a ‘0028’ *hex* into CSR0.

With selected interrupts, specific shared-memory locations can be set to transmit and/or receive interrupts. If Interrupts Armed CSR1[14] set to ‘1’, a selected interrupt sequence is generated by transmitting data to any interrupt memory location.

Therefore, any processor on the network with a specific memory location set to receive interrupts, will receive an interrupt message from the network if data was transmitted to that specific memory location by any other processor on the same network.

While processing an interrupt, Interrupts Armed CSR1[14] goes low. However, receipt of a network interrupt while the previous interrupt is being processed would result in the shared-memory location being updated (data WRITE), and the interrupt addresses added to the Interrupt FIFO stack.

4.4.1 External Triggers

The selected-interrupt technique also provides two external trigger mechanisms through the ACR. ACR[2] controls trigger 1 (ET1) and a hard pulse is generated at the end of host READ/WRITE cycle. ACR[3] controls trigger 2 (ET2) which generates a hard pulse at the end of a network WRITE cycle. The enabling procedure is similar to enabling the interrupt bits, except that to trigger ET1, WRITE a ‘04’ *hex* in the ACR and a ‘08’ *hex* to trigger ET2.

(SCRAMNet Classic only) Three other external triggers are constantly available and do not require any ACR designations. Trigger 3 generates a hard pulse when a network message is received with the Control Slot bit set. Trigger 4 generates a hard pulse when a network message is received with the interrupt bit set. Trigger 5 generates a hard pulse when the host sends a message to the network with the interrupt bit set.

4.5 Forced Interrupts

The forced-interrupt technique operates in the same manner as the selected-interrupt technique, except that the forced-interrupt technique automatically sets up all of the SCRAMNet shared-memory locations to either receive and/or transmit interrupts. With this technique, there is no particular address “selection” option for the user. External triggers are disabled when using the forced technique.

4.6 Interrupts on Errors

Interrupts can also be generated by network errors. This is accomplished by enabling Interrupts on Errors CSR0[7] by writing a ‘0080’ *hex* into CSR0. Remember the host has to be armed to process interrupts. This means enabling interrupts by writing a ‘0028’ *hex* into CSR0 that sets Host Interrupt Enable and Interrupt On Memory Mask Match Enable. The various network errors are signaled in CSR1 and a mask for these errors can be defined in CSR9. See the SCRAMNet *Network (bus) Hardware Interface Reference* for details on these error bits.

5. INTERFACE ROUTINES

5.1 Description

Routines residing in the SCRAMNet Interface Library can be accessed from any user-written program. All the subroutines provided in the library are classified into four general categories. Similar routines are grouped together. The purpose of this section is to acquaint the user with the SCRAMNet Interface Library routines and the logic behind the SCRAMNet Network Board.

5.2 Organization and Format

The following types of information are used to describe each routine:

Table 5-1 Interface Library Routine Information

NAME:	The routine name appears at the top of the first page.
SYNOPSIS:	The routine prototype and external variables changed by the subroutine, and the necessary include files.
SPECIFICATION:	If present, relates the information to particular computer systems.
DESCRIPTION:	Provides information about specific actions taken by the routine.
ARGUMENTS:	Describes the parameters passed to the function.
RETURNS:	Describes the values the function returns.
ERROR:	If present, contains information on the success or failure of the routine during its execution; with appropriate error messages.
NOTE:	Contains any additional information that is required of the routine, for example: all the additional data types required and their location.
EXAMPLE	Highlights the function only, the rest has been minimized. Each example may use supporting functions which are assumed to work correctly.

Table 5-2 Interface Routine Directory

Name	Type	Page	Function
get_base_mem()	Memory Access	5-49	Obtain SCRAMNet memory pointer
get_scr_node_id()	Configuration	5-4	Get SCRAMNet node ID.
get_scr_phy_csr_addr()	Configuration	5-5	Get physical base address of SCRAMNet CSR registers.
get_scr_phy_mem_addr()	Configuration	5-6	Get physical base address of SCRAMNet network memory.
get_scr_time_out()	Configuration	5-7	Get current network time-out value.
get_scr_user_mem_size()	Configuration	5-8	Get current memory size of board.
GetScrTransactionType()	Data Flow	5-36	Detect byte swapping method used.
ReadSCRByte()	Memory Access	5-50	READ a byte from memory.
ReadSCRLong()	Memory Access	5-51	READ a longword from memory.
ReadSCRWord()	Memory Access	5-52	READ a word from memory.
scr_acr_mm()	Interrupt	5-46	Auxiliary Control RAM control switch
scr_acr_read()	Configuration	5-9	READ ACR location
scr_acr_write()	Configuration	5-10	WRITE ACR location
scr_brd_select()	Configuration	5-11	Change SCRAMNet Board (Multiple Board Support Only)
scr_board_status()	Configuration	5-12	Create message window with current SCRAMNet config data.
scr_csr_read()	General	5-58	READ CSR values
scr_csr_write()	General	5-59	WRITE values to CSRs
scr_dma_read()	DMA	5-56	SCRAMNet+ DMA READ from SCRAMNet memory into user memory buffer
scr_dma_write()	DMA	5-57	SCRAMNet+ DMA WRITE from user memory buffer into SCRAMNet memory
scr_dfltr_mm()	Data Flow	5-38	Data Filtering control switch
scr_error_mm()	General	5-60	Checks for and explains errors defined by CSR1
scr_fifo_mm()	General	5-62	Error Register status dump/FIFO reset
scr_fswin_mm()	General	5-64	Check for fiber optic switch
scr_id_mm()	General	5-66	Node Identification information
scr_int_mm()	Interrupt	5-47	Interrupts mode select
scr_lnk_mm()	Data Flow	5-40	SCRAMNet node linker switch
scr_load_mm()	General	5-68	Configuration file loader
scr_mclr_mm()	General	5-70	Memory or ACR clear control
scr_mem_mm()	Configuration	5-13	Maps the SCRAMNet physical memory
scr_probe_mm()	Configuration	5-15	Validates SCRAMNet memory mapping information
scr_read_int_fifo()	General	5-72	READ interrupt FIFO CSRs
scr_reg_mm()	Configuration	5-17	Maps the SCRAMNet control status registers
scr_reset_mm()	Configuration	5-19	Reset SCRAMNet node
scr_rw_mm()	General	5-74	Control Status register control
scr_save_mm()	General	5-76	Configuration file saver

Name	Type	Page	Function
scr_smem_mm()	General	5-78	Memory or ACR modifier
scr_wml_mm()	Data Flow	5-42	Host Write disable switch
SetScrTransactionType()	Data Flow	5-37	Set byte swapping method to be used.
sp_bist_rd()	General	5-80	READ BIST data.
sp_cfg_read()	Configuration	5-21	READ Config File
sp_gtm_mm()	Data Flow	5-44	Return current transaction mode (PCI/PMC Only)
sp_mem_size()	General	5-81	Get memory size of board
sp_msg_life()	General	5-82	Set message life
sp_net_to()	General	5-83	Set network time-out
sp_plus_find()	Configuration	5-22	Detect board type
sp_protocol()	General	5-84	Set network protocol
sp_rx_id()	General	5-85	Set receiver ID
sp_scram_init()	Configuration	5-23	READ config and map registers and memory
sp_set_cntr()	General	5-86	Set General Purpose Counter/Timer
sp_set_sm_addr()	General	5-87	Set physical memory address
sp_set_vp()	General	5-88	Set virtual page
sp_stm_mm()	Data Flow	5-45	Set transaction mode (PCI/PMC Only)
sp_txrx_id()	General	5-89	Set transmitter and receiver ID
sw_cfg_fill()	Configuration	5-24	Fills SCRAM_CFG data structure
sw_get_int()	Configuration	5-26	Get the current driver interrupt number.
sw_int_connect()	Configuration	5-27	Terminate interrupt operations.
sw_int_disconnect()	Configuration	5-28	Initialize interrupt operations.
sw_mem_addr()	Configuration	5-29	Get physical address of SCRAMNet memory.
sw_net_to()	Configuration	5-30	Get network time-out (as stored in registry).
sw_reg_addr()	Configuration	5-31	Get physical address of CSRs.
sw_set_size()	Configuration	5-32	Set mapped memory size to use upon driver initialization.
sw_set_int()	Configuration	5-33	Set the interrupt number to use upon driver installation.
sw_user_size()	Configuration	5-34	Get user-specified memory size (as stored in registry).
WriteSCRByte()	Memory Access	5-53	WRITE a byte to memory.
WriteSCRLong()	Memory Access	5-54	WRITE a longword to memory.
WriteSCRWord()	Memory Access	5-55	WRITE a word to memory.

5.3 Configuration Routines

5.3.1 get_scr_node_id() - Get SCRAMNet Node ID

SYNOPSIS

```
#include <scr.h>
BYTE get_scr_node_id( void )
```

DESCRIPTION

This routine returns the node id assigned to the SCRAMNet card.

ARGUMENTS

None.

RETURNS

Returns a BYTE value set equal to the node id. Valid node id's are in the range 0 - 255.

ERROR

get_scr_node_id() will fail if:

- *sp_scram_init()* has not been called to initialize the card.

5.3.2 get_scr_phy_csr_addr() - Get SCRAMNet CSR Address

SYNOPSIS

```
#include <scr.h>
DWORD get_scr_phy_csr_addr( void )
```

DESCRIPTION

This routine returns the physical-memory address at which the SCRAMNet Control/Status Registers (CSR) are located.

ARGUMENTS

None.

RETURNS

DWORD value set equal to the base address at which the SCRAMNet CSRs are located.

ERROR

get_scr_phy_csr_addr() will fail if:

- *sp_scram_init()* has not been called to initialize the card.

5.3.3 get_scr_phy_mem_addr() - Get SCRAMNet Memory Address

SYNOPSIS

```
#include <scr.h>
DWORD get_scr_phy_mem_addr( void )
```

DESCRIPTION

This routine returns the physical-memory address at which the SCRAMNet memory is located.

ARGUMENTS

None.

RETURNS

Returns a DWORD value set equal to the base address at which the SCRAMNet memory is located.

ERROR

get_scr_phy_mem_addr() will fail if:

- *sp_scram_init()* has not been called to initialize the card.

5.3.4 get_scr_time_out() - Get Network Time-out

SYNOPSIS

```
#include <scr.h>
BYTE get_scr_time_out( void )
```

DESCRIPTION

This routine returns the SCRAMNet Network node time-out value.

ARGUMENTS

None.

RETURNS

Returns a BYTE value set equal to the time-out value. Valid values are in the range 0 - 255.

ERROR

get_scr_time_out() will fail if:

- *sp_scram_init()* has not been called to initialize the card.

5.3.5 get_scr_user_mem_size() - Get Application Memory Size

SYNOPSIS

```
#include <scr.h>
DWORD get_scr_user_mem_size( void )
```

DESCRIPTION

This routine returns the memory size to be used by the application. This value is stored in the **Windows Registry** and is provided as a means of allowing an application to use only a portion of the available SCRAMNet memory.

ARGUMENTS

None.

RETURNS

Returns the number of bytes of SCRAMNet memory to be used**.** The **Windows Registry** describes memory sizes in kilobytes but this routine returns values in bytes**.** The value returned cannot be larger than the total amount of physical SCRAMNet memory in the system.

ERROR

get_scr_user_mem_size() will fail if:

- *sp_scram_init()* has not been called to initialize the card.

5.3.6 scr_acr_read() - READ ACR location

SYNOPSIS

```
#include <scrplus.h>
unsigned char scr_acr_read( unsigned long mem_loc );
```

DESCRIPTION

The *scr_acr_read* function will READ the Auxiliary Control RAM (ACR) and return the value. This function interfaces with the hardware dependent portion of the libraries to determine exactly how to access these memory locations. The number of ACR locations depends upon the SCRAMNet memory size. There is one byte of ACR associated with every longword (32 bits) of SCRAMNet memory. Definitions of the bit functions for the ACR are provided in the header files as well as in the hardware documentation.

This routine will not function until SCRAMNet memory and CSRs have been mapped.

ARGUMENTS

The **mem_loc** parameter is the number of the memory location that the associated ACR is to be read from, as in the following example:

unsigned char acr_val;
acr_val = scr_acr_read(0);

RETURNS

Returns the value read from the specified ACR.

ERROR

scr_acr_read will fail if:

- CSRs have not yet been mapped.
- SCRAMNet memory has not been mapped.
- An invalid memory location was passed by **mem_loc** parameter.

5.3.7 scr_acr_write() - WRITE ACR location

SYNOPSIS

```
#include <scrplus.h>
void scr_acr_write( unsigned long mem_loc, unsigned char
acr_val );
```

DESCRIPTION

The *scr_acr_write* routine will WRITE the Auxiliary Control RAM (ACR) specified by **mem_loc** with the value given. It interfaces with the hardware dependent portion of the libraries to determine exactly how to access the ACR locations. The number of accessible ACR locations depends upon the SCRAMNet memory size. There is one byte of ACR associated with every longword (32 bits) of SCRAMNet memory. Definitions of the bit functions for the ACR are provided in the header files as well as in the hardware documentation.

This routine will not function until SCRAMNet CSRs have been mapped.

ARGUMENTS

The **mem_loc** parameter is the number of the memory location where the associated ACR will be written, as in the following example:

unsigned char acr_val = ACR_RIE | ACR_TIE;
scr_acr_write(0, acr_val);

ACR bits are defined in the header files.

RETURNS

None

ERROR

scr_acr_write will fail if:

- CSRs have not yet been mapped.
- SCRAMNet memory has not been mapped.
- An invalid memory location was passed by **mem_loc** parameter.

5.3.8 scr_brd_select() - Change SCRAMNet Board (Multiple Board Support Only)

SYNOPSIS

```
#include <scrplus.h>
int scr_brd_select( int brd_num);
```

DESCRIPTION

scr_brd_select() selects a single SCRAMNet board for user control in multiple board systems. After calling this function with the board number passed as the only parameter, all other SCRAMNet library routines may then be called to access the given board.

This routine is used only on systems that support multiple SCRAMNet boards.

ARGUMENTS

The board number parameter must lie within the range of 0 through N-1, where N is the number of board configured on the system.

RETURNS

Returns the board number selected if successful or ‘-1’ if an error has occurred.

ERROR

scr_brd_select() will fail if:

- The board number specified is out of range.

5.3.9 scr_board_status() - Display Board Status

SYNOPSIS

```
void scr_board_status( HWND )
```

DESCRIPTION

Displays a common dialog box filled with information concerning the current SCRAMNet board configuration.

The following current information is given:

- Node transmit ID
- CSR base address
- Memory base address
- Memory mapped in bytes
- Memory size in bytes
- Network time-out
- External power
- Mechanical switch loopback status

ARGUMENTS

A valid (parent) window handle. This may be any valid handle, such as that from your main Windows process or from any dialog box you open.

RETURNS

None.

ERROR

None.

EXAMPLE

The following code segment displays the status dialog box. Due to the complexity of Windows NT code, only a small portion of the code is presented. However, full examples can be seen in the **windiags.c** file.

```
#include <scrplus.h>
LRESULT WndProc( HWND hWnd, . . .) {
    . . .
    sp_scram_init(); // map CSRs and memory
    . . .
    scr_board_status(hWnd); // display status dialog box
```

5.3.10 scr_mem_mm() - Map Memory

SYNOPSIS

```
#include <scrplus.h>
unsigned int   scr_mem_mm( int arg );
```

DESCRIPTION

scr_mem_mm() performs the host system specific operations to map the SCRAMNet physical base memory address into system address space, thus providing pointer access to SCRAMNet on-board memory. The global pointer (SCR_LONG_PTR scr_vmem_ptrs) is the pointer to the base address of SCRAMNet memory returned by the host system. The actual pointer value may be obtained by calling the *get_base_mem()* function.

ARGUMENTS

Valid arguments are **MAP** and **UNMAP**. Not all systems require that physical mapping be unmapped before terminating the application but it is a safe practice to unmap the SCRAMNet memory before exiting.

RETURNS

Returns a ‘0’ if successful, or a ‘-1’ if an error has occurred.

ERROR

scr_mem_mm() will fail if:

- Memory segment size is greater than a system imposed maximum.
- Shared memory identifier does not exist.
- Shared memory identifier is created but the system imposed limit on the maximum number of allowed memory system identifiers is exceeded.
- User is not a superuser and memory device file requires superuser permission.
- Number of shared memory segments attached to the calling process exceeded the system-imposed limit.
- Available data space is not large enough to accommodate the shared memory segment.
- Illegal physical memory address.
- Invalid shared memory identifier.

EXAMPLE: SCR_MEM_MM IN FILE SCR_MEM_EX.C

This program uses a pointer to SCRAMNet memory to READ and WRITE values. In this case, the SCRAMNet memory pointer is cast to an (unsigned char *) to allow character I/O. The pointer can be cast to any integral or floating point type. The *puts()* function displays the null terminated string at offset 100 in SCRAMNet memory. SCRAMNet memory is released before program termination.

```
#include <stdio.h>
#include <stdlib.h>
#include <scrplus.h>
main( )
{
SCR_BYTE_PTR mem_ptr;
   /* map SCRAMNet memory and set the global pointer to the base */
if( scr_mem_mm( MAP ) != 0 )   {
   printf("Could not map SCRAMNet RAM");
   exit( 0 );   }

   /* cast to a character pointer for writing to SCRAMNet RAM
             on a character boundary */
mem_ptr = (SCR_BYTE_PTR) get_base_mem();

   /* read string into SCRAMNet shared memory at address 0 */
printf("\nEnter string to store in SCRAMNet: ");
gets( mem_ptr );
printf("String stored at offset 0\n\n");
   /* display string at offset 100 in SCRAMNet memory */
printf("Found this at offset 100: ");
strcpy( &mem_ptr[100], "This is offset 100\0" );
puts( &mem_ptr[100] );

   /* unmap SCRAMNet memory before termination */
if( scr_mem_mm( UNMAP ) != 0 )   {
   printf("Could not unmap SCRAMNet RAM");
   exit( 0 );   }
}
```

5.3.11 scr_probe_mm() - Probe

SYNOPSIS

```
#include <scrplus.h>
int  scr_probe_mm( void *addr,  unsigned short int flag );
```

DESCRIPTION

scr_probe_mm() probes the specified address **addr** for the validity of that address location. The existence of the specified location is determined by attempting a READ operation at that address. The memory location in question could be those that have been mapped using the *scr_mem_mm()* and *scr_reg_mm()* routines in this library. Always call this routine to ensure proper mapping was performed when calling the map routines.

ARGUMENTS

Valid arguments for the **flag** field are either 1, 2 or 4 based on the READ access required. This access refers to the number of bytes.

RETURNS

Returns a ‘0’ if a valid address location is specified by the **addr** argument, or a ‘-1’ if the address is invalid.

ERROR

scr_probe_mm() will fail if:

- The **flag** argument is anything other than a 1, 2 or 4.

EXAMPLE SCR_PROBE_MM IN FILE SCR_PROBE_EX.C

This program maps the memory using *scr_mem_mm()* and then attempts to access the newly mapped memory. It is recommended that *scr_probe_mm()* be used during startup to test the initialization of SCRAMNet. This program is limited in usefulness because it does not perform SCRAMNet configuration. The CSRs must be mapped via the *scr_reg_mm()* function before configuration can be accomplished. The CSR registers must also be mapped if they are to be probed.

```
#include <stdio.h>
#include <stdlib.h>
#include <scrplus.h>
#define TWO 2
main( )
{
SCR_SHORT_PTR  mem_ptr;

   /* map SCRAMNet memory and set the global pointer to the base */

scr_mem_mm( MAP );

   /* cast to an unsigned short int pointer for writing to SCRAMNet
     memory on a 16-bit boundary or TWO byte boundary */

mem_ptr = (SCR_SHORT_PTR) get_base_mem();

   /* try to access the base of SCRAMNet memory */

if( scr_probe_mm( mem_ptr, TWO ) == -1 )   {
  printf(“Unable to access mapped memory. Please check
scr_mem_mm()\n”);
  printf(“to see if memory was mapped correctly\n\n”);
  exit( 0 );   }
else
   printf(“\nMemory mapped and accessed successfully\n”);

   /* unmap SCRAMNet memory before termination */

scr_mem_mm( UNMAP );
}
```

5.3.12 scr_reg_mm() - Map Registers

SYNOPSIS

```
#include <scrplus.h>
int scr_reg_mm( int arg );
```

DESCRIPTION

scr_reg_mm() sets the global address of SCRAMNet Control Status Registers (CSRs). Access to the individual CSRs is provided via *scr_csr_read()* and *scr_csr_write()* routines. This function must be called before CSRs can be read or written.

ARGUMENTS

Valid arguments to *scr_reg_mm()* are **MAP** and **UNMAP**. It is good programming practice to unmap the CSRs once the application process is completed.

RETURNS

Returns a ‘0’ if successful, or a ‘-1’ if an error has occurred.

ERROR

scr_reg_mm() will fail if:

- Memory segment size is greater than a system imposed maximum.
- Shared memory identifier does not exist.
- Shared memory identifier is created but the system imposed limit on the maximum number of allowed memory system identifiers is exceeded.
- User is not a superuser or the root.
- Number of shared memory segments attached to the calling process exceeded the system-imposed limit.
- Available data space is not large enough to accommodate the shared memory segment.
- Illegal physical memory address.
- Invalid shared memory identifier.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <scrplus.h>

main( )
{

    /* map SCRAMNet CSRs and set the global pointer to the base */
if( scr_reg_mm( MAP ) != 0 )   {
   printf(“Could not map SCRAMNet CSRs”);
   exit( 0 );   }

    /* set CSR0 to network operation w/data filtering */
scr_csr_write(SCR_CSR0, 0x8c03);

    /* display contents of error register CSR1 */
printf(“\nThe contents of CSR1 = %x\n”, scr_csr_read(SCR_CSR1) );

    /* unmap SCRAMNet CSRs before termination */
if( scr_reg_mm( UNMAP ) != 0 )   {
   printf(“Could not unmap SCRAMNet CSRs”);
   exit( 0 );   }

}
```

5.3.13 scr_reset_mm() - Reset FIFOs

SYNOPSIS

```
#include <scrplus.h>
void scr_reset_mm( void );
```

DESCRIPTION

scr_reset_mm() duplicates the hardware reset function. This routine initializes the SCRAMNet Network Node to a zero condition such that there is always a standard starting point when configuring the board from a cold start or a new application.

scr_reset_mm() accomplishes this task by resetting the controlling mechanism of the SCRAMNet Node, via CSR0 and CSR2.

This reset includes resetting the FIFOs:

CSR0[12] - Transmit/Receive FIFO

CSR0[13] - Interrupt FIFO

CSR0[14] - Shared Memory FIFO

and clearing the data rerouter and other control bits in CSR2.

ARGUMENTS

None.

RETURNS

Void.

ERROR

scr_reset_mm() will fail if:

- Control status registers were not mapped.

EXAMPLE SCR_RESET_MM IN FILE SCR_RESET_EX.C

This program maps the CSRs so that *scr_reset_mm()* will have them available, calls *scr_reset_mm()*, and resets the board. CSRs are unmapped before program termination.

```
#include <scrplus.h>

main( )
{
  /* map  SCRAMNet registers, set the global pointer to the base */
scr_reg_mm( MAP );

  /* reset  SCRAMNet */
scr_reset_mm( );

  /* unmap  SCRAMNet registers before termination */
scr_reg_mm( UNMAP );
}
```

5.3.14 sp_cfg_read() - READ SCRAMNet Configuration File

SYNOPSIS

```
#include <scrplus.h>
int sp_cfg_read ( int board_number )
```

DESCRIPTION

This routine is not used for PCI NT. See **sw_cfg_fill()** on page 5-24.

ARGUMENTS

Not used.

RETURNS

Returns a ‘0’.

ERROR

Not used.

5.3.15 sp_plus_find() - Find Board Type

SYNOPSIS

```
#include <scrplus.h>
int sp_plus_find ( void )
```

DESCRIPTION

The *sp_plus_find* routine will determine whether the SCRAMNet is a Classic or LX/+ board. CSR3 is used to make this determination. Therefore, SCRAMNet registers must be mapped before this routine is called.

ARGUMENTS

None.

RETURNS

Returns a ‘0’ if the board is a Classic, or a ‘1’ if the board is an LX/+ board.

ERROR

sp_plus_find() will fail if:

- The SCRAMNet registers are not mapped.

5.3.16 sp_scram_init() - Initialize SCRAMNet Mapping

SYNOPSIS

```
#include <scr.h>
BOOL sp_scram_init( void )
```

DESCRIPTION

This routine sets up the board for access by reading in values from the **Windows Registry**, maps the SCRAMNet CSR registers and memory and sets the values for the Node Id and time-out value based on the values in the **Windows Registry**.

ARGUMENTS

None.

RETURNS

0	(FALSE) if successful

If unsuccessful (TRUE), returns one of the following;

-4	Could not open driver
-3	Could not map memory
-2	Could not map CSR's
-1	Could not read registry

ERROR

sp_scram_init() will fail if:

- The system is unable to determine the system address for accessing the **SCRAMNet** memory or Control/Status Registers (CSR).
- The system is unable to properly complete the mapping or other setup needed to access the **SCRAMNet** memory and CSR.

5.3.17 sw_cfg_fill() - Fills Values Defined in SCRAM_CFG Data Structure

SYNOPSIS

```
#include <scrplus.h>
int sw_cfg_read( SCRAM_CFG *Scramnet_config )
```

DESCRIPTION

When programming in a Microsoft Windows environment, it is important to remember the fundamental differences between Windows and other environments such as DOS and UNIX. One difference is that global data generated for a Dynamic Link Library is not shared with any application that uses the Dynamic Link Library. All of the configuration information pertinent to the SCRAMNet board is stored in the Dynamic Link Library's global memory, which is unavailable to the application program. In a DOS or UNIX environment, simply including the SCRAMNet header files would generate this variable and fill in the values at run time. This is not true for Windows.

However, the application program will most likely require certain configuration parameters to run correctly. One such parameter, SCR_MEM_LEN (defined in **scrhwd.h**), conveniently allows the application to determine the size of (mapped) SCRAMNet memory.

The application is responsible for copying the configuration information from the dynamic link library to its memory space, and it uses *sw_cfg_read()* to do so. When the application calls *sw_cfg_read()* (with a pointer to the SCRAMNet_*config* variable), the dynamic link library fills in the structure, and the configuration data members (such as SCR_MEM_LEN) are then, and only then, valid.

NOTE: If the application never uses the configuration data members (defined in **scrhwd.h**), this function need not be called at all.

ARGUEMENTS

A pointer to the memory reserved for the SCRAMNet_*config* variable. The variable MUST be named SCRAMNet_*config* (the compiler preprocessor definitions in **scrhwd.h** require this for compatibility). Also, the memory must have been previously allocated for this variable or the function will fail.

RETURNS

Returns a '0' if successful or '-1' if there was a memory error during the data copy.

ERROR

sw_cfg_fill() will fail if memory was not reserved for the configuration structure.

EXAMPLE

This program initializes the SCRAMNet card then fills the configuration variable.

```
#include <process.h> // exit()
#include <scrplus.h> // includes scrhwd.h

SCRAM_CFG Scramnet_config; // NOTE: memory allocated from global
pool!

int main ( void ) {

sp_scram_init();

if ( sw_cfg_fill(&Scramnet_config) < 0 ) {
printf("Error filling configuration data structure.\n");
exit(-1);
} /* if */

printf("Mapped SCRAMNet memory size (in bytes) is:
%d\n",SCR_MEM_LEN);

return(0);
} /* main */
```

5.3.18 sw_get_int() - Get NT Interrupt Number

SYNOPSIS

```
BYTE sw_get_int( void )
```

DESCRIPTION

This routine gets the current interrupt number being used by the Windows NT SCRAMNet driver.

ARGUMENTS

None.

RETURNS

BYTE value (unsigned char) representing current interrupt number.

ERROR

None.

EXAMPLE

This program first maps SCRAMNet memory, then it determines the current interrupt.

```
#include <scrplus.h>

int main( void ) {

    sp_scram_init(); // map CSRs and memory

printf(“Current SCRAMNet interrupt %d\n”,sw_get_int());
    return(0);
} /* main */
```

5.3.19 sw_int_connect() - Initialize NT Interrupts

SYNOPSIS

```
BOOL sw_int_connect( void )
```

DESCRIPTION

This routine initializes interrupts in the Windows NT environment. This call creates an event object which is passed to the SCRAMNet device driver. When the interrupt fires, the object is signaled.

The defined value `SCR_INTERRUPT_OBJECT` is used along with the board number as the interrupt object. When running a single-board system, just use board number 0.

ARGUMENTS

None.

RETURNS

TRUE if interrupts were successfully connected, FALSE if not.

ERROR

None.

EXAMPLE

This program first maps SCRAMNet memory, then it connects interrupts. If the interrupts connected successfully, it creates an event object and waits for an interrupt.

```
#include <scr.h>
#include <scrplus.h>

int main( void )
{
char intString[30]
sp_scram_init(); // map CSRs and memory

if ( sw_int_connect() )
{
   /* Create interrupt event object. */
   sprintf(intString, “%s%d”, SCR_INTERRUPT_OBJECT, 0);
   hIntEvent = CreateEvent(NULL,FALSE,FALSE,intString);

   /* Wait forever, or until SCRAMNet interrupts. */
   if (WAIT_OBJECT_0 == WaitForSingleObject(hIntEvent,INFINITE))
   {
         printf(“SCRAMNet interrupted!\n”);
         sw_int_disconnect(); // disconnect interrupts
         CloseHandle(hIntEvent); // close interrupt object handle
   } /* if */
   return(0);
}} /* main */
```

5.3.20 sw_int_disconnect() - Terminate NT Interrupts

SYNOPSIS

```
BOOL sw_int_disconnect( void )
```

DESCRIPTION

This routine terminates interrupt operations in the Windows NT environment.

This function does not close the handle created by using `SCR_INTERRUPT_OBJECT.` That must be done using the interrupt termination code.

ARGUMENTS

None.

RETURNS

TRUE if interrupts were successfully disconnected, FALSE if not.

ERROR

None.

EXAMPLE

This program first maps SCRAMNet memory, then it connects interrupts. If the interrupts connected successfully, it creates an event object and waits for an interrupt. When the interrupt fires, interrupt processing is terminated.

```
#include <scrplus.h>

int main( void )
{
    char intString[30]
    sp_scram_init(); // map CSRs and memory
    if ( sw_int_connect() )
    {
    /* Create interrupt event object. */
    sprintf(intString, “%s%d”, SCR_INTERRUPT_OBJECT, 0);
    hIntEvent = CreateEvent(NULL,FALSE,FALSE,intString);

    /* Wait forever, or until SCRAMNet interrupts. */
    if (WAIT_OBJECT_0 == WaitForSingleObject(hIntEvent,INFINITE))
    {
    printf(“SCRAMNet interrupted!\n”);
    sw_int_disconnect(); // disconnect interrupts
    CloseHandle(hIntEvent); // close interrupt object handle
    } /* if */
    return(0);
}} /* main */
```

5.3.21 sw_mem_addr() - READ SCRAMNet Memory Address

SYNOPSIS

```
DWORD sw_mem_addr( void )
```

DESCRIPTION

This routine returns the physical card address of SCRAMNet memory.

This information may be used by display routines or for loop determination. The value returned, however, may NOT be used to access SCRAMNet memory (use *get_base_mem()* for that).

ARGUMENTS

None.

RETURNS

The DWORD physical address at which the Base of SCRAMNet memory is located.

ERROR

sw_mem_addr() will return a NULL pointer value if it cannot access the configuration registers on the card.

EXAMPLE

This program first maps SCRAMNet memory, then it determines the physical address of the memory.

```
#include <scrplus.h>

int main( void ) {

    sp_scram_init(); // map CSRs and memory

printf(“Physical address of SCRAMNet memory is: \
0x%08x”,sw_mem_addr());

    return(0);
} /* main */
```

5.3.22 sw_net_to() - READ Network Time-out

SYNOPSIS

```
int sw_net_to( void )
```

DESCRIPTION

This routine returns the network time-out (as stored in Windows NT Registry). This value was written to the SCRAMNet card when the driver began execution.

ARGUMENTS

None.

RETURNS

The integer value representing the network time-out, as stored in the Registry (use WinEPI to modify this value in the Registry, or use *sp_net_to()* to change the current time-out value for the SCRAMNet card).

ERROR

None

EXAMPLE

This program first maps SCRAMNet memory, then it reports the time-out value.

```
#include <scrplus.h>

int main( void ) {

    sp_scram_init(); // map CSRs and memory

printf(“Stored (Registry) time-out value is %d”,sw_net_to());

    return(0);
} /* main */
```

5.3.23 sw_reg_addr() - READ CSR Address

SYNOPSIS

```
DWORD sw_reg_addr( void )
```

DESCRIPTION

This routine returns the physical card address of the CSRs.

This information may be used by display routines or for loop determination. The value returned, however, may NOT be used to access the CSRs (use *scr_csr_read()* and *scr_csr_write()* for that).

ARGUMENTS

None.

RETURNS

The DWORD physical address of the CSR registers.

ERROR

sw_reg_addr() will return a NULL pointer value if it cannot access the configuration registers on the card.

EXAMPLE

This program first maps the CSRs, then it determines the physical address of the CSRs.

```
#include <scrplus.h>

int main( void ) {

    sp_scram_init(); // map CSRs and memory

    printf(“Physical address of CSRs is: 0x%08x”,sw_reg_addr());

    return(0);
} /* main */
```

5.3.24 sw_set_size() - Set NT Memory Size

SYNOPSIS

```
BOOL sw_set_size( DWORD )
```

DESCRIPTION

This routine sets the desired amount of memory to map in the Windows NT Registry upon driver initialization (any amount up to the physical memory limitations of the SCRAMNet card). Previously, this was done by editing the **scrcfg.dat** file. However, Windows NT uses the Registry making modification of this value more difficult without this routine.

The memory setting assigned here does not take effect until the SCRAMNet driver begins execution. The easiest way to be sure the proper settings are used is to shut down Windows NT (reboot) and re-execute the SCRAMNet code.

ARGUMENTS

A DWORD value indicating the number of bytes of memory to map.

RETURNS

TRUE if setting was stored in the Registry, FALSE if not.

ERROR

None

EXAMPLE

This program first maps SCRAMNet memory, then it sets the desired mapping range to 256KB.

```
#include <scrplus.h>

int main( void ) {

    sp_scram_init(); // map CSRs and memory

if ( sw_set_size(256*1024) ) { // convert to _BYTES_
printf(“SCRAMNet mapped memory modified (please reboot)!\n”);
} /* if */
else {
printf(“Error setting SCRAMNet mapped memory!\n”);
} /* else */
    return(0);
} /* main */
```

5.3.25 sw_set_int() - Set NT Interrupt Number

SYNOPSIS

```
BOOL sw_set_int( BYTE )
```

DESCRIPTION

This routine provides a way to set the interrupt number that the Windows NT SCRAMNet driver will use when it initializes. A correct value here is required to assure proper interrupt operations.

When the interrupt number has been determined, use *sw_set_int()* to modify the Registry, modify the Registry manually, or use WinEPI or WinDiags (under the “Settings…” menu command) to modify the Registry.

ARGUMENTS

A BYTE value (unsigned char) consisting of the following:

- 3	- 10
- 4	- 11
- 5	- 12
- 7	- 13
- 9	- 14

RETURNS

TRUE if setting was stored in the Registry, FALSE if not.

ERROR

None

EXAMPLE

This program first maps SCRAMNet memory, and then it sets the desired interrupt to 10.

```
#include <scrplus.h>

int main( void ) {

    sp_scram_init(); // map CSRs and memory

if ( sw_set_int(10) ) {
printf(“SCRAMNet interrupt modified (please reboot)!\n”);
} /* if */
else {
printf(“Error setting SCRAMNet interrupt!\n”);
} /* else */
    return(0);
} /* main */
```

5.3.26 sw_user_size() - READ NT Memory Size

SYNOPSIS

```
DWORD sw_user_size( void )
```

DESCRIPTION

Determines the user-specified memory size value as stored in the Windows NT Registry. This value is the amount of memory the SCRAMNet driver will map, which may include all or some of the physical SCRAMNet memory. Values greater than the physical amount of memory reported by the SCRAMNet card will result in mapping the entire SCRAMNet memory space. A value of zero is undefined and will cause failure.

To change this value, see *sw_set_size()*. (Page 5-32)

ARGUMENTS

None.

RETURNS

The DWORD value representing the user-specified memory (in bytes).

ERROR

None

EXAMPLE

This program first maps SCRAMNet memory, then it reports the user-specified memory value.

```
#include <scrplus.h>

int main( void ) {

    sp_scram_init(); // map CSRs and memory

printf("Stored (Registry) user-specified memory size is \
%ld",sw_user_size());

    return(0);
} /* main */
```

5.4 Data Flow Control Routines

The Data Flow Control routines are designed to aid the development of a program that communicates between Big-Endian and Little-Endian based machines. Since SCRAMNet VME cards do not provide swapping functions and are run primarily on Big-Endian systems, the PCI, EISA, ISA and other PC-based SCRAMNet cards do provide byte swapping functions to help convert data from Little Endian to Big Endian format.

The SCRAMNet DLL provides a function to place the SCRAMNet Network interface card in one of three modes—byte, longword or word. In byte mode all of the bytes are in reverse order. In longword mode there is no byte swapping. In word mode the upper two bytes and the lower two bytes of every longword are reversed.

To use the same offsets for both Big- and Little-Endian systems, the SCRAMNet card can be instructed to alter the byte order of the values being read. This alteration allows a Little-Endian system reading a byte at offset zero to see the same value as a Big-Endian system.

EXAMPLE

Suppose that the longword value ‘AABBCCDD *hex’* is stored in SCRAMNet memory at offset zero. Bits 31 through 24 (AA *hex*) would be written at byte offset 0; bits 23 through 16 (BB *hex*) at byte offset 1; bits 15 through 8 (CC *hex*) at byte offset 2; and bits 7 through 0 (DD *hex*) at byte offset 3. On a Little-Endian system, these bytes would be stored and accessed in reverse order so that a READ to offset zero would return ‘DD *hex*’ while the same READ on a Big-Endian system would return ‘AA *hex*’.

5.4.1 GetScrTransactionType() - Get Byte Swapping Mode

SYNOPSIS

```
#include <scr.h>
BYTE GetScrTransactionType ( void )
```

DESCRIPTION

This routine accesses the SCRAMNet byte-swapping controls to determine the current byte-swapping mode, and returns a constant integer value indicating the current byte-swapping method in use. (See Table 5-3 for more information on the differences between Long_mode, Word_mode, and Byte_mode).

ARGUMENTS

None

RETURNS

Returns the constant value Long_mode if the card is in longword swapping mode.
Returns the constant value Word_mode if the card is in word swapping mode.
Returns the constant value Byte_mode if the card is in byte swapping mode.

ERROR

GetScrTransactionType() will fail if:

- There is an error in interacting with the **SCRAMNet** byte-swapping controls.

5.4.2 SetScrTransactionType() - Alter Byte-Access Order

SYNOPSIS

```
#include <scr.h>
BOOL SetScrTransactionType ( BYTE Mode )
```

DESCRIPTION

This routine alters the order in which bytes are accessed from SCRAMNet memory. The header file includes constants defined as Long_mode, Word_mode and Byte_mode. When this function is called with one of these values it alters the byte order accordingly.

ARGUMENTS

The **mode** parameter is used to determine which byte-swapping mode the card should be placed in. This argument is set using one of the following constants Long_mode, Word_mode or Byte_mode. When the function is called with one of these values the byte order is altered according to Table 5-3.

Table 5-3 Byte Swapping Table

Parameter	Swap mode	Data bit translation			
		M(31:24)	**M(23:16)**	**M(15:8)**	**M(7:0)**
Long_mode	no swap, 32-bit	R(31:24)	R(23:26)	R(15:8)	R(7:0)
Word_mode	16-bit	R(15:8)	R(7:0)	R(31:24)	R(23:16)
Byte_mode	8-bit	R(7:0)	R(15:8)	R(23:16)	R(31:24)

RETURNS

TRUE if operation successful, FALSE if there was an error in configuring the SCRAMNet card to work in the appropriate swapping mode.

ERROR

SetScrTransactionType() will fail if:

- There is an error in configuring the device to work in the specified mode.

5.4.3 scr_dfltr_mm() - Set Data Filter

SYNOPSIS

```
#include <scrplus.h>
void scr_dfltr_mm( int arg)
```

DESCRIPTION

scr_dfltr_mm() sets the SCRAMNet node to one of three data filtering modes—turn off filter mode, filter above the first 4 KB memory locations, or filter entire SCRAMNet memory. One of these three modes initializes the host appropriately to filter network data WRITE cycles.

The Data Filter control bits in CSR0 include:

CSR0[10] - Enable TX Data Filter
CSR0[11] - Enable Lower 4 KB Data Filter

CSR0[10]	CSR0[11]	Data Filter State
0	0	OFF
0	1	OFF
1	0	HI - Filter After First 4 KB
1	1	ALL - Filter All Memory

ARGUMENTS

Valid arguments to *scr_dfltr_mm()* are **FLT_OFF**, **FLT_HGH** or **FLT_ALL**. **FLT_OFF** disables the data filter altogether. **FLT_HGH** monitors and filters above the first 4 KB of SCRAMNet memory. **FLT_ALL** initializes the host to monitor and filter the entire SCRAMNet memory.

RETURNS

Void

ERROR

scr_dfltr_mm() will fail if:

- Control status registers were not mapped.

EXAMPLE SCR_DFLTR_MM IN FILE SCR_DFLTR_EX.C

This program first maps the CSRs, then it demonstrates the setting of the data filter to its various modes.

```
#include <scrplus.h>

main( )
{
  /* map SCRAMNet registers and set the global pointer to the base */
scr_reg_mm( MAP );

  /* reset  SCRAMNet NOTE: no filtering now */
scr_reset_mm( );

  /* enable data filtering above the first 4k only */
scr_dfltr_mm( FLT_HGH );

  /* enable data filtering for all memory */
scr_dfltr_mm( FLT_ALL );

  /* disable data filtering */
scr_dfltr_mm( FLT_OFF );

  /* unmap  SCRAMNet registers before termination */
scr_reg_mm( UNMAP );
}
```

5.4.4 scr_lnk_mm() - Link to Network

SYNOPSIS

```
#include <scrplus.h>
void  scr_lnk_mm( int cmd );
```

DESCRIPTION

scr_lnk_mm() activates the node and then inserts it into the network ring. Inserting the node involves setting it to transmit and receive data as well as enabling the Insert Mode bit. If the fiber optic bypass switch is being used, then the Disable Fiber Optic Loopback bit must be set.

The Insert node control bits in CSR0 include:

CSR0[0] - Receiver Enable
CSR0[1] - Transmitter Enable
CSR0[15] - Insert Node

The Insert node control bits in CSR 2 include:

CSR2[7] - Disable Fiber Optic Loopback

ARGUMENTS

Valid arguments to *scr_lnk_mm()* are **SCR_LNK** and **RST_LNK**.

SCR_LNK initializes the SCRAMNet Network node and includes this specific node into the network ring. **RST_LNK** deactivates the node by extracting it off the network and resetting the control bits.

RETURNS

Void

ERROR

scr_lnk_mm() will fail if:

- Control/status registers were not mapped.

EXAMPLE SCR_LNK_MM IN FILE SCR_LNK_EX.C

This program first maps the CSRs, then it demonstrates the insertion of a node into the network ring. In this mode, the network updates SCRAMNet RAM and this node updates on other nodes by writing to SCRAMNet RAM. If the node is not inserted into the ring, the SCRAMNet card functions simply as a memory board.

```
#include <scrplus.h>

main( )
{
  /* map  SCRAMNet registers and set the global pointer to the
     base */
scr_reg_mm( MAP );

  /* reset  SCRAMNet NOTE: Node not inserted into network now*/
scr_reset_mm( );

  /* makes node an active network participant */
scr_lnk_mm( SCR_LNK );

  /* removes node from network participation */
scr_lnk_mm( RST_LNK );

  /* unmap  SCRAMNet registers before termination */
scr_reg_mm( UNMAP );
}
```

5.4.5 scr_wml_mm() - Write-Me-Last

SYNOPSIS

```
#include <scrplus.h>
void scr_wml_mm( int arg );
```

DESCRIPTION

scr_wml_mm() can set the update mode for SCRAMNet RAM. In normal mode, the WRITE attempt to SCRAMNet RAM results in the data going directly to RAM and simultaneously being mirrored to the network. The alternative is “Write-Me-Last” mode. In this mode, the WRITE attempt to SCRAMNet memory is sent directly to the network. SCRAMNet RAM remains unchanged until the message returns from the network. Upon return, the WRITE data is placed in the SCRAMNet RAM.

scr_wml_mm() can be used as a diagnostic procedure in determining whether the data received by the host system is the same as the data that was sent out. This can also be used in conjunction with the self-interrupt mode.

The No-Write-to-Host control bits in CSR2 include:

CSR2[8] - Disable Host to SM WRITE
CSR2[9] - Enable Write Own Slot

ARGUMENTS

Valid arguments to *scr_wml_mm()* are **NTW_WRT_HST** and **RST_NWH**. **NTW_WRT_HST** enables the diagnostic mode, whereas **RST_NWH** enables Host WRITEs to shared memory and disables writing its own network slot to memory.

RETURNS

Void

ERROR

scr_wml_mm() will fail if:

- Control status registers were not mapped.

☞ **NOTE**: Earlier versions of the SCRAMNet library supported this routine with the name *scr_nwh_mm()*. This new routine *scr_wml_mm()* provides the same functionality as *scr_nwh_mm()* did.

EXAMPLE SCR_WML_MM IN FILE SCR_WML_EX.C

This program illustrates how to set the SCRAMNet RAM update mode for either Write-Me-Last, or for direct WRITE to SCRAMNet RAM, which is simultaneously mirrored to the network. The Write-Me-Last mode forces any WRITE to SCRAMNet RAM onto the network first. After the message returns from the network, the data is written to the originating node. This mode ensures that all other nodes receive the data before the originating node.

To use the Write-Me-Last mode, WRITE ‘0x8003’ to CSR0. This entry ensures that the node is inserted into the ring and is transmitting and receiving. Calling *scr_lnk()* accomplishes this task. If the node is not active on the ring, the WRITE to memory that is being redirected to the network will never be transmitted, the data will never be received, and memory will not be updated. Instead, the FIFO buffer will eventually overflow and data will be lost permanently. If the condition is observed before the transmit FIFO overflows, set CSR0 to ‘0x8003’, and the buffer will flush itself onto the network. Once the data is received, SCRAMNet RAM will be updated.

WARNING: The transmit FIFO can hold only 1024 WRITE attempts.

```
#include <scrplus.h>

main( )
{
 /* map SCRAMNet registers and set the global pointer to the base
*/
scr_reg_mm( MAP );

 /* reset  SCRAMNet NOTE: Node now WRITEs to shared memory only.
    This is the same mode as scr_wml_mm( RST_NWH ) */
scr_reset_mm( );

 /* makes node an active network participant */
scr_lnk_mm( SCR_LNK );

 /* now we change it so that the WRITEs to shared memory go
    directly to the network. Shared memory will not be updated
    until the data returns from the network */
scr_wml_mm( NTW_WRT_HST );

 /* now we WRITE directly to SCRAMNet RAM which is mirrored to the
    network */
scr_wml_mm( RST_NWH );

 /* removes node from network participation */
scr_lnk_mm( RST_LNK );

 /* unmap  SCRAMNet registers before termination */
scr_reg_mm( UNMAP );
}
```

5.4.6 sp_gtm_mm() - Return current transaction mode (PCI/PMC Only)

SYNOPSIS

```
BYTE sp_gtm_mm( void )
```

DESCRIPTION

Provides a method for determining the current transaction mode for SCRAMNet memory access (Longword, Word, or Byte).

ARGUMENTS

None.

RETURNS

The BYTE value (unsigned char) representing the current transaction mode (0 for Longword, 1 for Word, and 2 for Byte).

ERROR

None.

EXAMPLE

This program first maps SCRAMNet memory, then it determines the current transaction mode.

```
#include <scrplus.h>

int main( void ) {

    sp_scram_init(); // map CSRs and memory

printf("Current SCRAMNet memory transaction mode is: \
%d\n",sp_gtm_mm());

    return(0);
} /* main */
```

5.4.7 sp_stm_mm() - Set transaction mode (PCI/PMC Only)

SYNOPSIS

```
BOOL sp_stm_mm( BYTE )
```

DESCRIPTION

Provides a method for setting the transaction mode for SCRAMNet memory access (Longword, Word, or Byte).

ARGUMENTS

BYTE value (unsigned char) consisting of:

- “Long_mode” (0)
- “Word_mode” (1)
- “Byte_mode” (2)

See Table 5-3 for more information.

RETURNS

TRUE if the transaction mode was set, FALSE if not.

ERROR

None.

EXAMPLE

This program first maps SCRAMNet memory, then sets the transaction mode to “Word_mode”.

```
#include <scrplus.h>

int main( void ) {

   sp_scram_init(); // map CSRs and memory

   sp_stm_mm(Word_mode);

   return(0);
} /* main */
```

5.5 Interrupt Routines

5.5.1 scr_acr_mm() - ACR Enable

SYNOPSIS

```
#include <scrplus.h>
void     scr_acr_mm ( int arg );
```

DESCRIPTION

scr_acr_mm() enables/disables the Auxiliary Control RAM (ACR). The Auxiliary Control RAM provides a method of interrupt control when a particular shared memory location is to be accessed. This function permits access to the ACR. Once in ACR access mode, the ACR is swapped in place of the SCRAMNet Network Node memory. Access to the ACR is via the same pointer used to access SCRAMNet main RAM. WRITEs using this pointer are directed to the ACR, as are READs when in ACR mode. Once in ACR access mode, configure ACR and use this command again with an appropriate argument to disable ACR access mode and regain access to SCRAMNet main RAM.

The Auxiliary Control RAM control bits in CSR0 include:
CSR0[4] - Auxiliary Control RAM enable

ARGUMENTS

Valid arguments to *scr_acr_mm()* are **MAP** and **UNMAP**. **MAP** enables the ACR mode for modification. This action must be followed by **UNMAP**, which disables the ACR and returns to normal operation.

RETURNS

Void

ERROR

scr_acr_mm() will fail if:

- Control status registers were not mapped.

EXAMPLE SCR_ACR_MM

See *scr_int_mm* (Page 5-47).

5.5.2 scr_int_mm() - Set Interrupt Mode

SYNOPSIS

```
#include <scrplus.h>
void     scr_int_mm ( int cmd, int arg );
```

DESCRIPTION

scr_int_mm() configures the host SCRAMNet node for interrupt operations. This routine assumes that the Auxiliary Control RAM (ACR) has been configured.

scr_int_mm() sets the desired type of interrupt scheme specified by the user. SCRAMNet defines three levels of **cmd** mode; that is, three levels of interrupt schemes.

1 - Receive Interrupts
2 - Transmit Interrupts
3 - Interrupt on Errors

These modes can either be set or reset based on the argument fields.

ARGUMENTS

Valid **cmd** arguments to *scr_int_mm()* are **TX_INT**, **RX_INT** and **INT_ERR**. Valid **arg** arguments to *scr_int_mm()* include **CLR_INT** to clear the selected mode, **RST_INT** to reset the mode, and **SET_INT** to set the selected **cmd** interrupt mode.

RETURNS

Void.

ERROR

scr_int_mm() will fail if:

- Control/status registers were not mapped.

EXAMPLE SCR_ACR_MM & SCR_INT_MM IN FILE SCR_ACR_INT_EX.C

This program demonstrates setting Interrupts by enabling and disabling the ACR with *scr_acr_mm()*. While in ACR access mode, address-interrupt properties can be modified by changing the corresponding ACR address. Once ACR is configured, enable processing of interrupts with *scr_int_mm()*.

```
#include <scrplus.h>

main( )
{
 SCR_LONG_PTR  mem_ptr;
 unsigned short int csr_tmp=0;

  /* map  SCRAMNet memory and registers */

scr_reg_mm(MAP);
scr_mem_mm(MAP);

  /* cast the memory  and ACR pointer to access main RAM and ACR */
mem_ptr = get_base_mem();

  /* set memory to ACR access mode */
scr_acr_mm(MAP);

  /* first four bytes Transmit Interrupt Enable (TIE) */
mem_ptr[0] = 0x01;

  /* the next four bytes both TIE and RIE */
mem_ptr[1] = 0x03;

  /* re-enable main  SCRAMNet RAM disabling ACR RAM */
scr_acr_mm( UNMAP );

  /* set SCRAMNet to recognize the need to transmit interrupts */
scr_int_mm(TX_INT,SET_INT);

  /* set SCRAMNet to recognize and enable the processing of incoming
     interrupts */
scr_int_mm(RX_INT,SET_INT);

  /* unmap  SCRAMNet memory and registers before termination */
scr_mem_mm(UNMAP);
scr_reg_mm(UNMAP);
}
```

5.6 Memory Access Routines

5.6.1 get_base_mem() - Get Memory Pointer

SYNOPSIS

```
#include <scrplus.h>
SCR_LONG_PTR get_base_mem ( void );
```

DESCRIPTION

This routine will return a pointer to the base address of SCRAMNet memory. The routine *scr_mem_mm()* must be called before this function will return a valid value.

ARGUMENTS

None.

RETURNS

Returns the pointer to SCRAMNet memory.

ERROR

get_base_mem will fail if:

- SCRAMNet memory has not yet been mapped.

5.6.2 ReadSCRByte() - READ Byte of SCRAMNet Memory

SYNOPSIS

```
#include <scr.h>
BYTE ReadSCRByte( DWORD offset, BYTEPTR nValue )
```

DESCRIPTION

This routine reads a single BYTE (8-bit unsigned char) value from SCRAMNet memory at the offset specified.

ARGUMENTS

The **offset** parameter is a DWORD value specifying the number of BYTE values from the start of SCRAMNet memory to the desired value.

The **nValue** parameter is a pointer to a BYTE in which the value read from memory will be copied.

RETURNS

Returns the BYTE read from memory.

ERROR

ReadSCRByte() will fail if:

- **SCRAMNet** memory has not been mapped yet.

5.6.3 ReadSCRLong() - READ Longword of SCRAMNet Memory

SYNOPSIS

```
#include <scr.h>
DWORD ReadSCRLong( DWORD offset, DWORDPTR lnValue )
```

DESCRIPTION

This routine reads a single DWORD (32-bit unsigned long integer) value from SCRAMNet memory at the specified offset.

ARGUMENTS

The **offset** parameter is a DWORD value specifying the number of DWORD values from start the of SCRAMNet memory to the desired value.

The **lnValue** parameter is a pointer to a DWORD were the value read from memory will be copied.

RETURNS

Returns the DWORD read from memory.

ERROR

ReadSCRLong() will fail if:

- **SCRAMNet** memory has not been mapped yet.

5.6.4 ReadSCRWord() - READ Word of SCRAMNet Memory

SYNOPSIS

```
#include <scr.h>
WORD ReadSCRWord( DWORD offset, WORDPTR nValue )
```

DESCRIPTION

This routine reads a single WORD (16-bit unsigned short integer) value from SCRAMNet memory at the offset specified.

ARGUMENTS

The **offset** parameter is a DWORD value specifying the number of WORD values from the start of SCRAMNet memory to the desired value.

The **nValue** parameter is a pointer to a WORD were the value read from memory will be copied.

RETURNS

Returns the WORD read from memory.

ERROR

ReadSCRWord() will fail if:

- **SCRAMNet** memory has not been mapped yet.

5.6.5 WriteSCRByte() - WRITE Byte to SCRAMNet Memory

SYNOPSIS

```
#include <scr.h>
BYTE WriteSCRByte ( DWORD offset, BYTE nValue )
```

DESCRIPTION

This routine writes a single BYTE (8-bit unsigned char) value to SCRAMNet memory at the offset specified.

ARGUMENTS

The **offset** parameter is a DWORD value specifying the number of BYTE values from the start of SCRAMNet memory to the desired location.

The **nValue** parameter is a the BYTE that is to be written to SCRAMNet memory.

RETURNS

Returns the BYTE written to memory.

ERROR

WriteSCRByte() will fail if:

- **SCRAMNet** memory has not been mapped yet.

5.6.6 WriteSCRLong() - WRITE Longword to SCRAMNet Memory

SYNOPSIS

```
#include <scr.h>
DWORD WriteSCRLong( DWORD offset, DWORD lnValue )
```

DESCRIPTION

This routine writes a single DWORD (32-bit unsigned long integer) value to SCRAMNet memory at the offset specified.

ARGUMENTS

The **offset** parameter is a DWORD value specifying the number of DWORD values from the start of SCRAMNet memory to the desired location.

The **lnValue** parameter is the DWORD that is to be written to SCRAMNet memory.

RETURNS

Returns the DWORD written to memory.

ERROR

WriteSCRLong() will fail if:

- **SCRAMNet** memory has not been mapped yet.

5.6.7 WriteSCRWord() - WRITE Word to SCRAMNet Memory

SYNOPSIS

```
#include <scr.h>
WORD WriteSCRWord( DWORD offset, WORD nValue )
```

DESCRIPTION

This routine writes a single WORD (16-bit unsigned short integer) value to SCRAMNet memory at the offset specified.

ARGUMENTS

The **offset** parameter is a DWORD value specifying the number of WORD values from the start of SCRAMNet memory to the desired location.

The **nValue** parameter is the WORD to be written to SCRAMNet memory.

RETURNS

Returns the WORD written to memory.

ERROR

WriteSCRWord() will fail if:

- **SCRAMNet** memory has not been mapped yet.

5.7 DMA Routines

5.7.1 Overview

The SCRAMNet PCI card provides DMA capability. A user application can access SCRAMNet DMA capability through the two routines *scr_dma_read()* and *scr_dma_write()*. These functions are only available in the NTPCPC2S software package for NT 4.0.

These DMA routines generate hardware interrupts during the DMA transaction.

5.7.2 scr_dma_read() - DMA READ from SCRAMNet Memory (PCI Only)

SYNOPSIS

```
#include <scrhwd.h>
int scr_dma_read( PVOID user_addr, ULONG scr_offset, ULONG
num_bytes )
```

DESCRIPTION

This routine uses the SCRAMNet+ PCI DMA capability to READ from SCRAMNet memory into a user memory buffer.

ARGUMENTS

PVOID user_addr - This is the starting address of the user memory buffer that is the destination for the data READ from SCRAMNet memory.
ULONG scr_offset - This is the offset address into SCRAMNet memory to be read from. Since this is a local address, SCRAMNet memory starts at ‘0’.
ULONG num_bytes - This is the number of longwords to READ from SCRAMNet memory into the destination memory buffer.

RETURNS

Returns a ‘0’ if successful, or ‘-1’ if an error has occurred.

ERROR

scr_dma_read() will fail if:

- DMA transfer size too large. The maximum transfer size will depend on the system page size and continuous nature of the user memory buffer. A 512 K or smaller DMA transfer should always be possible.
- The DMA transaction timed out after one second.

5.7.3 scr_dma_write() - DMA WRITE into SCRAMNet Memory (PCI Only)

SYNOPSIS

```
#include <scrhwd.h>
int scr_dma_write( PVOID user_addr, ULONG scr_offset, ULONG
num_bytes )
```

DESCRIPTION

This routine uses the SCRAMNet+ PCI DMA capability to WRITE from a user memory buffer into SCRAMNet memory.

ARGUMENTS

PVOID user_addr -	This is the starting address of the user memory buffer that is the source of the data to be written into SCRAMNet memory.
ULONG scr_offset -	This is the offset address into SCRAMNet memory to be written to. Since this is a local address, SCRAMNet memory starts at '0'.
ULONG num_bytes -	This is the number of longwords to WRITE from the memory buffer into SCRAMNet memory.

RETURNS

Returns a '0' if successful, or '-1' if an error has occurred.

ERROR

scr_dma_write() will fail if:

– DMA transfer size too large. The maximum transfer size will depend on the system page size and continuous nature of the user memory buffer. A 512 K or smaller DMA transfer should always be possible.

– The DMA transaction timed out after one second.

5.8 General Routines

5.8.1 scr_csr_read() - READ Registers

SYNOPSIS

```
#include <scrplus.h>
unsigned short scr_csr_read( unsigned int csr_number );
```

DESCRIPTION

The *scr_csr_read()* function will READ the Control/Status register (CSR) and return the value read. This function interfaces with the hardware dependent portion of the libraries to determine exactly how to access these registers. The number of accessible CSRs depends upon the SCRAMNet model (Classic or LX/+). Definitions of the bit functions for each CSR are provided in the header files as well as in the hardware documentation.

This routine will not function until SCRAMNet CSRs have been mapped.

ARGUMENTS

The **csr_number** parameter is the number of the CSR to be read, as in the following example:

unsigned short csr_val;
csr_val = scr_csr_read(SCR_CSR0);

CSR numbers are defined in the header files. The definitions are actually the number of the CSR itself.

RETURNS

Returns a value read from the specified CSR.

ERROR

scr_csr_read will fail if:

- CSRs have not yet been mapped.
- An invalid CSR number was passed by **csr_number** parameter.

5.8.2 scr_csr_write() - WRITE Registers

SYNOPSIS

```
#include <scrplus.h>
void scr_csr_write( unsigned int csr_number, unsigned short
value );
```

DESCRIPTION

The *scr_csr_write()* routine will WRITE the Control/Status register (CSR) specified by **csr_number** with the value given. This routine interfaces with the hardware dependent portion of the libraries to determine exactly how to access these registers. The number of accessible CSRs depends upon the SCRAMNet model (Classic or LX/+). Definitions of the bit functions for each CSR are provided in the header files as well as in the hardware documentation.

This routine will not function until SCRAMNet CSRs have been mapped.

ARGUMENTS

The **csr_number** parameter is the number of the CSR to be written, as in the following example:

```
unsigned short csr_val = 8003 hex;
scr_csr_write( SCR_CSR0, csr_val );
```

CSR numbers are defined in the header files. The definitions are actually the number of the CSR itself.

RETURNS

None

ERROR

scr_csr_write will fail if:

- CSRs have not yet been mapped.
- An invalid CSR number was passed by **csr_number** parameter.

5.8.3 scr_error_mm() - Network Error Display

SYNOPSIS

```
#include <scrplus.h>
void  scr_error_mm ( FILE *ofd,unsigned short int tmp_csr
);
```

DESCRIPTION

scr_error_mm() receives the contents of CSR1 and displays an appropriate error message to explain the error. The output message is displayed to the screen by default and may be sent to a file pointer to **ofd**. If **ofd** is equal to a NULL, output is only displayed on the screen. Otherwise, output is sent to the file pointed to by **ofd** after being displayed on the screen.

ARGUMENTS

Valid **cmd** arguments to *scr_error_mm()* are **(FILE *)ofd** and **(unsigned short int)tmp_csr**. Valid values of **ofd** must either be a valid file pointer or NULL if output is not to be sent to a file. The argument **tmp_csr** must contain the contents of CSR1.

RETURNS

Void.

ERROR

scr_error_mm() will fail if:

- Control status registers were not mapped.
- If the **ofd** file pointer is not properly declared as a file pointer or equal to NULL.

EXAMPLE SCR_ERROR_MM IN FILE SCR_ERROR_EX.C

This program maps CSRs and then uses *scr_error_mm()* to check CSR1 for errors. If any errors are found, *scr_error_mm()* will display the appropriate error message. SCRAMNet registers are released before program termination. Please note that the file pointer parameter for *scr_error_mm()* is a NULL. In this case, output is displayed only on the screen.

```
#include <stdio.h>
#include <stdlib.h>
#include <scrplus.h>

   /* the pointer that will be set by the scr_reg_mm() */
extern unsigned short int * scr_reg_ptr;
extern unsigned long int * scr_reg_mm();

main( )
{
struct scr_device *dp;

   /* map  SCRAMNet CSRs and set the global pointer to the base */
if( scr_reg_mm( MAP ) != 0 )   {
   printf(“Could not map  SCRAMNet CSRs”);
   exit( 0 );   }

   /* display message of error register CSR1 */
scr_error_mm( NULL, scr_csr_read(SCR_CSR1));

   /* unmap  SCRAMNet CSRs before termination */
if( scr_reg_mm( UNMAP ) != 0 )   {
   printf(“Could not unmap  SCRAMNet CSRs”);
   exit( 0 );   }

}
```

5.8.4 scr_fifo_mm() - Reset FIFOs/READ Status

SYNOPSIS

```
#include <scrplus.h>
void    scr_fifo_mm( int cmd, struct rd_fifo *ptr );
```

DESCRIPTION

scr_fifo_mm() allows the user to reset the shared memory, interrupt and transmit/receive FIFO or READ the status register CSR1. See the **scr.h** file in your software package for a prototype of the structure **rd_fifo**.

The reset FIFO control bits in CSR0 include:

CSR0[12] - Transmit/Receive FIFO
CSR0[13] - Interrupt FIFO
CSR0[14] - Shared Memory FIFO

In case of reading the Status register the bits include:

CSR1[0] - SM FIFO Full
CSR1[1] - SM FIFO Not Empty
CSR1[2] - SM FIFO Half Full
CSR1[3] - T/R FIFO Full
CSR1[4] - Interrupt FIFO Full
CSR1[5] - SM FIFO Skew
CSR1[6] - Carrier Detect
CSR1[7] - Bad Byte
CSR1[8] - Receiver Overflow

ARGUMENTS

Valid **cmd** arguments to *scr_fifo_mm()* are **RST_FIFO** to reset one of three FIFO's or **RD_FIFO** status via CSR1.

RETURNS

Void.

ERROR

scr_fifo_mm() will fail if:

- Control status registers were not mapped.

EXAMPLE SCR_FIFO_MM IN FILE SCR_FIFO_EX.C

This program READs the status of the FIFO buffers and checks the shared memory FIFO full flag as an example. If the structure member is equal to one, the condition exists. The next line resets all of the FIFOs. The structure **rd_fifo** is found in **scr.h** and explains what conditions are available other than the one in this example.

```
#include <stdio.h>
#include <scrplus.h>

extern void scr_fifo_mm( );

main( )
{
struct rd_fifo ptr;

   /* map  SCRAMNet registers */
scr_reg_mm( MAP );

   /* READs the status of the FIFOs and places status in structure
         ptr */
scr_fifo_mm( RD_FIFO, &ptr );

   /* As an example, check to see if shared memory FIFO is half
full */
if( ptr.smfhf == 1 )   {
  printf(“Shared Memory FIFO is half FULL”);   }

   /* all FIFOs are reset here structure in ptr is unchanged
         during reset operation. */
scr_fifo_mm( RST_FIFO, &ptr );

   /* unmap  SCRAMNet registers before termination */
scr_reg_mm( UNMAP );

}
```

5.8.5 scr_fswin_mm() - Check for Fiber Optic Switch

SYNOPSIS

```
#include <scrplus.h>
int scr_fswin_mm( void );
```

DESCRIPTION

scr_fswin_mm() checks the state of the “Fiber Optic Bypass Not Connected” bit in CSR1. The status of this bit is reflected in the return value of the function call.

ARGUMENTS

None.

RETURNS

Returns a value of ‘0’ if the switch is not present, or a ‘1’ if the switch is present.

ERROR

scr_fswin_mm() will fail if:

- Control/status registers were not mapped.

EXAMPLE SCR_FSWIN_MM

This program maps the CSRs so that *scr_fswin_mm()* will have them available. CSRs are unmapped before program termination.

```
#include <scrplus.h>

main( )
{
   /* map  SCRAMNet registers, set the global pointer to the base */
scr_reg_mm( MAP );

   /* Check SCRAMNet By-Pass Switch */
if (scr_fswin_mm( ) == 1)
  printf("\n Fiber Optic By-Pass switch present.");
else
  printf("\n Fiber Optic By-Pass switch NOT present.");

  /* unmap  SCRAMNet registers before termination */
scr_reg_mm( UNMAP );
}
```

5.8.6 scr_id_mm() - READ Node Identification

SYNOPSIS

```
#include <scrplus.h>
void   scr_id_mm( char *id, char *cnt );
```

DESCRIPTION

scr_id_mm() returns the node ID of the node on the network ring, and the total number of nodes in the network ring. The node ID and the total node count ranges from 0 - 255.

The number of nodes on the network ring is valid only when the host node has successfully transmitted at least one message around the ring.

This is general information and may be used by the user as host system identification.

ARGUMENTS

Two character pointers are passed, **id** and **cnt**. The pointer **id** will contain the node ID number of the node executing the command. The pointer **cnt** will contain the number of nodes in the current ring.

RETURNS

A value is not returned; but this routine sets passed pointer variables to correct values.

ERROR

scr_id_mm() will fail if:

- Control status registers were not mapped.

EXAMPLE SCR_ID_MM IN FILE SCR_ID_EX.C

This program first sends a message around the network by setting a variable. Since the data filters are not active immediately after a reset, this will generate a network update message. Next the *scr_id_mm()* routine is used to find, then display, the node ID and total node count.

```
#include <stdio.h>
#include <scrplus.h>

  /* the pointers set by scr_reg_mm() and scr_mem_mm()*/
extern unsigned short int * scr_reg_ptr;
extern unsigned long int * scr_vmem_ptrs;

extern long int scr_mem_mm( ), scr_reg_mm( );
extern void scr_id_mm( );
main( )
{
unsigned char cnt, id;
short int *mem_ptr;
struct scr_device *dp;

  /* map  SCRAMNet memory and registers */
scr_mem_mm( MAP );
scr_reg_mm( MAP );

  /* reset  SCRAMNet board */
scr_reset_mm( );

  /* initialize memory pointer and send one message */
mem_ptr = get_base_mem( );

  /* insert node into the network */
scr_csr_write(SCR_CSR0,0x8003);

  /* transmit message to get node number count */
*mem_ptr = 0;

  /* get the node id and node count */
scr_id_mm( &id, &cnt );

  /* Display the node id and total node count */
printf("\nThe id of this node is %u or in hex %x\n\
The number of active nodes in the network is %u\n", id, id, cnt );

  /* unmap  SCRAMNet memory and registers before termination */
scr_mem_mm( UNMAP );
scr_reg_mm( UNMAP );
}
```

5.8.7 scr_load_mm() - READ File to Load ACR / RAM

SYNOPSIS

```
#include <scrplus.h>
void  scr_load_mm ( char *strptr, int  cmd );
```

DESCRIPTION

scr_load_mm() READs the disk file specified by the character string at pointer **strptr**. On successful file access, the contents of the file are loaded into the SCRAMNet shared memory or the Auxiliary Control RAM according to **cmd**.

This routine permits easy reconfiguration of the SCRAMNet Host Node to different modes from a selection of predetermined files and is complementary to *scr_save_mm().*

This routine also allows reloading of the entire SCRAMNet memory, or the entire SCRAMNet interrupt scheme as defined in the ACR, or both.

ARGUMENTS

Valid **cmd** arguments to *scr_load_mm()* are **MEM** or **ACR**, depending on what it is the user wishes to reload. **Strptr** is a null-terminated string that contains the file name.

RETURNS

Void.

ERROR

scr_load_mm() will fail if:

- Control status registers were not mapped.
- Shared memory was not mapped.
- File could not be opened.
- File READ error occurred.
- Illegal memory mapping was attempted.

EXAMPLE SCR_LOAD_MM IN FILE SCR_LOAD_EX.C

This program first gets the name of the file to be read, then uses *scr_load_mm()* to open the file and read the contents of the file into SCRAMNet RAM.

```
#include <stdio.h>
#include <scrplus.h>

extern void scr_load_mm( );

main( )
{
char str[20];

  /* map  SCRAMNet memory and registers */
scr_mem_mm( MAP );
scr_reg_mm( MAP );

  /* reset  SCRAMNet board */
scr_reset_mm( );

  /* read the file name, then the file contents into  SCRAMNet
     RAM */
printf(“\nPlease enter the filename to read into  SCRAMNet RAM:
“);
gets( str );
scr_load_mm( str, MEM );

  /* unmap  SCRAMNet memory and registers before termination */
scr_mem_mm( UNMAP );
scr_reg_mm( UNMAP );
}
```

5.8.8 scr_mclr_mm() - Clear ACR / RAM

SYNOPSIS

```
#include <scrplus.h>
void  scr_mclr_mm( int arg );
```

DESCRIPTION

scr_mclr_mm() zeroes all of shared memory segment mapped for SCRAMNet. This routine also initializes the SCRAMNet Network Node to a clear memory condition such that there is always a standard starting point for new applications.

scr_mclr_mm() supports the option to either clear contents of the entire SCRAMNet memory or to clear or zero out the contents of the entire SCRAMNet Auxiliary Control RAM (ACR).

ARGUMENTS

Valid **cmd** arguments to *scr_mclr_mm()* are **ACR** or **MEM** depending on what needs to be zeroed out.

RETURNS

Void.

ERROR

scr_mclr_mm() will fail if:

- Control/status registers were not mapped.

EXAMPLE SCR_MCLR_MM IN FILE SCR_MCLR_EX.C

This program first maps CSRs and then clears the ACR and SCRAMNet RAM. The board is reset before and after clearing to make sure there is nothing in FIFO's when finished.

```
#include <scrplus.h>

extern void scr_mclr_mm( );

main( )
{

   /* Mapping  SCRAMNet RAM and registers */
scr_mem_mm( MAP );
scr_reg_mm( MAP );

   /* reset  SCRAMNet board */
scr_reset_mm( );

   /* clear ACR and  SCRAMNet RAM */
scr_mclr_mm( ACR );
scr_mclr_mm( MEM );

   /* reset  SCRAMNet board */
scr_reset_mm( );

   /* unmap  SCRAMNet registers before termination */
scr_reg_mm( UNMAP );
scr_reg_mm( UNMAP );
}
```

5.8.9 scr_read_int_fifo() - READ Interrupt FIFO CSRs

SYNOPSIS

```
#include <scrplus.h>
int scr_read_int_fifo( unsigned long int *fifo_value );
```

DESCRIPTION

scr_read_int_fifo() READs the first value in the interrupt FIFO. This value is found by reading CSR4 and CSR5. The value read is written to the **fifo_value** parameter. The return status will indicate whether the FIFO was empty or not. This tells whether the value read from CSR4 and CSR5 is valid or not.

ARGUMENTS

Valid argument to *scr_read_int_fifo()* is a pointer to an **unsigned long int** in which this routine will WRITE the value found in the interrupt FIFO.

RETURNS

Returns a value indicating whether or not the interrupt FIFO was empty when read.

ERROR

scr_read_int_fifo() will fail if:

- Control status registers were not mapped.

EXAMPLE SCR_READ_INT_FIFO

This program maps the CSRs so that *scr_read_int_fifo()* will have them available, then calls *scr_read_int_fifo()*. CSRs are unmapped before program termination.

```
#include <scrplus.h>

main( )
{
  unsigned long fifo_entry;

   /* map  SCRAMNet registers, set the global pointer to the base */
scr_reg_mm( MAP );

   /* read  SCRAMNet interrupt FIFO */
if  (scr_read_int_fifo(&fifo_entry ) == TRUE)
   printf("\n Current Interrupt FIFO entry: %lx",fifo_entry);
else
   printf("\n Interrupt FIFO is empty");

   /* unmap  SCRAMNet registers before termination */
scr_reg_mm( UNMAP );
}
```

5.8.10 scr_rw_mm() - READ / WRITE / Modify Registers

SYNOPSIS

```
#include <scrplus.h>
extern void
scr_rw_mm( operation )
struct rw_scr *operation;
```

DESCRIPTION

scr_rw_mm() READs, WRITEs, or modifies one of the SCRAMNet Control Status Registers (CSRs) based on the operation structure defined by *struct rw_scr* below.

ARGUMENTS

Structure of type *rw_scr*. The *rw_scr* structure is defined as:

struct rw_scr {

char rw_flag;
char csr_reg;
short int csr_arg; };

Valid *rw_flag* options are:

REG_RD - Register READ
REG_WRT - Register Write
REG_MDFY - Register Modify

Valid *csr_reg* options are:

SCR_CSR0 - SCRAMNet CSR 0
SCR_CSR1 - SCRAMNet CSR 1
SCR_CSR2 - SCRAMNet CSR 2
SCR_CSR3 - SCRAMNet CSR 3
SCR_CSR4 - SCRAMNet CSR 4
SCR_CSR5 - SCRAMNet CSR 5
SCR_CSR6 - SCRAMNet CSR 6

Valid *csr_arg* options are:

short int value to be written to appropriate CSR if *rw_flag* == *REG_WRT*.
Otherwise, *rw_flag* == *REG_MDFY*, appropriate CSR is OR'ed with *csr_arg*.

RETURNS

If READ operation, *csr_arg* will contain the appropriate register value.

ERROR

scr_rw_mm() will fail if:

- Control/status registers were not mapped.
- Illegal register number.
- WRITE or modify attempted on a READ-only register.

EXAMPLE SCR_RW_MM IN FILE SCR_RW_EX.C

This program maps the CSRs then sets CSR0 to ‘0x8003’ to begin basic network participation. Then CSR1 is read and checked to see if Transmit FIFO is full. Registers are then unmapped.

```
#include <stdio.h>
#include <scrplus.h>

extern void scr_rw_mm( );

main( )
{
struct rw_scr temp;

   /* Mapping  SCRAMNet registers */
scr_reg_mm( MAP );

   /* set CSR0 to 0x8003 to enable network participation */
temp.rw_flag = REG_WRT;
temp.csr_reg = SCR_CSR0;
temp.csr_arg = 0x8003;
scr_rw_mm( &temp );

   /* check CSR1 to see if Transmit FIFO is full, bit 1 = 1 */
temp.rw_flag = REG_RD;
temp.csr_reg = SCR_CSR1;
scr_rw_mm( &temp );

if( (temp.csr_arg & 0x01) == 0x01 )
     printf(“Transmit FIFO is FULL\n”);

else
     printf(“Transmit FIFO is not FULL\n”);

   /* unmap  SCRAMNet registers before termination */
scr_reg_mm( UNMAP );
}
```

5.8.11 scr_save_mm() - Save ACR or RAM Contents to File

SYNOPSIS

```
#include <scrplus.h>
extern void
scr_save_mm( strptr, cmd )
char *strptr;
int cmd;
```

DESCRIPTION

scr_save_mm() WRITEs to the disk file specified by the character string at pointer **strptr**. On successful file access, the contents of SCRAMNet shared memory or the contents of the Auxiliary Control RAM are written into the file.

This routine permits easy reconfiguration of the SCRAMNet Host Node to different modes from a selection of predetermined files and is complementary to *scr_load_mm()*.

This routine also allows saving of the entire SCRAMNet memory, or the entire SCRAMNet interrupt scheme as defined in the ACR, or both.

ARGUMENTS

Valid **cmd** arguments to *scr_save_mm()* are **MEM** or **ACR**, depending on what needs to be reloaded.

RETURNS

Void.

ERROR

scr_save_mm() will fail if:

- Control/status registers were not mapped.
- Shared memory was not mapped.
- File could not be opened.

EXAMPLE SCR_SAVE_MM IN FILE SCR_SAVE_EX.C

This program first gets the filename to hold the contents of SCRAMNet RAM, then uses *scr_save_mm( )* to open the file and WRITE the contents of SCRAMNet RAM into that file.

```
#include <stdio.h>
#include <scrplus.h>

extern scr_save_mm( );

main( )
{
char str[20];

  /* map  SCRAMNet memory and registers */
scr_mem_mm( MAP );
scr_reg_mm( MAP );

  /* reset  SCRAMNet board */
scr_reset_mm( );

  /* get the file name to save in, and save contents of  SCRAMNet
     RAM */
printf("\nPlease enter the filename to save  SCRAMNet RAM in: ");
gets( str );
scr_save_mm( str, MEM );

   /* unmap  SCRAMNet memory and registers before termination */
scr_mem_mm( UNMAP );
scr_reg_mm( UNMAP );
}
```

5.8.12 scr_smem_mm() - Set ACR / RAM to Pattern

SYNOPSIS

```
#include <scrplus.h>
extern void
scr_smem_mm( arg, value )
int arg;
unsigned long value;
```

DESCRIPTION

scr_smem_mm() will set SCRAMNet memory or the SCRAMNet Auxiliary Control RAM (ACR) to the value supplied by the **value** argument field. The value field is set throughout SCRAMNet memory or ACR.

This feature is convenient when it is necessary to set entire SCRAMNet memory or the ACR to one particular value. For instance, in the case of the ACR, this function is useful when it is necessary to set all the memory to enable-interrupt on receipt of a new data.

ARGUMENTS

Valid arguments to *scr_smem_mm()* are **MEM** or **ACR** depending on whether the user wishes to set a value to entire SCRAMNet memory or the ACR.

RETURNS

Void.

ERROR

scr_smem_mm() will fail if:

- Control/status registers were not mapped.
- Memory was not mapped.
- TRANSMIT FIFO FULL error condition will cause this routine to “hang” waiting for the FIFO to empty itself. There must be some loopback to ensure that the FIFO has a path to empty itself.

EXAMPLE SCR_SMEM_MM IN FILE SCR_SMEM_EX.C

This program first maps memory and registers. Then memory and ACR are both set to certain values using *scr_smem_mm( )*. Note that the Transmit FIFO requires a path to empty itself. There must be a loopback path of some type, either a wire loopback or a fiber optic loopback. If no carrier, then default is wire. Then the node is inserted and will use the appropriate loopback if installed. If the Transmit FIFO fills, the library routine *scr_smem_mm()* will “hang” until it is no longer full. Finally, RAM and CSRs are unmapped.

```
#include <stdio.h>
#include <stdlib.h>
#include <scrplus.h>

  /* the pointers set by scr_mem_mm() and scr_reg_mm() */
extern unsigned short int * scr_reg_ptr;
extern void scr_smem_mm( );
extern long int scr_mem_mm( ), scr_reg_mm( );

main( )
{
struct scr_device *dp;

  /* map  SCRAMNet memory and registers */
scr_mem_mm( MAP );
scr_reg_mm( MAP );

  /* if no carrier, operate in wire loopback */
if(scr_csr_read(SCR_CSR1) & 0x0040 )
scr_csr_write(SCR_CSR2,0x0080);
scr_csr_write(SCR_CSR0,0x8003;

  /* clear  SCRAMNet RAM */
scr_smem_mm( MEM, 0 );

  /* set interrupts to transmit for all of  SCRAMNet RAM */
scr_smem_mm( ACR, 0x1 );

  /* if no carrier,  disable wire loopback */
if(scr_csr_read(SCR_CSR1) & 0x0040 )  scr_csr_write(SCR_CSR2,0);

  /* unmap  SCRAMNet memory and registers before termination */
scr_mem_mm( UNMAP );
scr_reg_mm( UNMAP );
}
```

5.8.13 sp_bist_rd() - READ BIST Data

SYNOPSIS

```
#include <scrplus.h>
int sp_bist_rd ( int* bitstream )
```

DESCRIPTION

This routine will READ the SCRAMNet LX ASIC built-in self-test data from CSR9 and stores it in the **bitstream** parameter. Any operations, including incoming network traffic, will disturb this number.

This routine does not work on a SCRAMNet Classic board.

ARGUMENTS

On return the **bitstream** argument will contain the built-in self-test data. The **bitstream** argument must be allocated enough space to hold 23 bytes of data.

RETURNS

Returns a ‘0’ if successful, or a ‘-1’ if an error has occurred.

ERROR

sp_bist_rd() will fail if:

- The SCRAMNet registers are not mapped.
- This routine is called using a SCRAMNet Classic Board.

5.8.14 sp_mem_size() - Get Hardware Memory Size

SYNOPSIS

```
#include <scrplus.h>
unsigned long sp_mem_size ( void )
```

DESCRIPTION

This routine will READ the memory size code from CSR8 and return the memory size in bytes.

This routine does not work on a SCRAMNet Classic board.

ARGUMENTS

None.

RETURNS

Returns the SCRAMNet memory size in bytes if successful, or a ‘0’ if an error has occurred.

ERROR

sp_mem_size() will fail if:

- The SCRAMNet registers are not mapped.
- This routine is called using a SCRAMNet Classic Board.

5.8.15 sp_msg_life() - Pre-Age Network Messages

SYNOPSIS

```
#include <scrplus.h>
int sp_msg_life ( unsigned hops )
```

DESCRIPTION

The *sp_msg_life* routine allows network messages to be pre-aged. Normally, a node sends messages to the network with an age of 0, then each node visited increments this age field by 1. When the age field reaches a value of 255, the message is removed from the network. The maximum number of nodes visited can be specified by passing this number to the routine. This routine should only be used when operating the network with BURST (or BURST+) mode protocols because other protocols expect to receive their own message. Note that a message will also be removed from the network if it is received by a node with a receiver ID that matches the ID field of the message (set with the value of the Transmit ID of the sender).

This routine does not work on a SCRAMNet Classic board.

ARGUMENTS

The **hops** argument should be set to the value of the number of node-hops the message will take before being removed from the network. For example, if a node wants to send the message only to its immediate neighbor, **hops** would be set to ‘1’.

RETURNS

Returns the previous value passed as **hops** if successful, or a ‘-1’ if an error has occurred.

ERROR

sp_plus_find() will fail if:

- The SCRAMNet registers are not mapped.
- A value of ‘0’ or a value > ‘255’ is passed for **hops**.
- This routine is called using a SCRAMNet Classic Board.

5.8.16 sp_net_to() - Set Network Time-out Value

SYNOPSIS

```
#include <scrplus.h>

int sp_net_to ( unsigned short time-out )
```

DESCRIPTION

This routine will set the Network Time-out value for the SCRAMNet LX node by writing the passed value to CSR5. The recommended formula for calculating network time-out value is:

#NODES + (TOTAL CABLE LENGTH OF RING IN METERS / 50) + 1

This routine does not work on a SCRAMNet Classic board.

ARGUMENTS

time-out - should be passed as the value to set as the network time-out value.

RETURNS

Returns a ‘0’ if successful, or a ‘-1’ if an error has occurred.

ERROR

sp_net_to() will fail if:

- The SCRAMNet registers are not mapped.
- This routine is called using a SCRAMNet Classic Board.

5.8.17 sp_protocol() - Set Network Protocol

SYNOPSIS

```
#include <scrplus.h>
int sp_protocol ( unsigned ProtocolMode )
```

DESCRIPTION

This routine will set the protocol mode for sending network messages. See the *Software Reference Manual* for specific information about which protocols are available on a particular host system.

ARGUMENTS

ProtocolMode - should be passed as the constant for the desired protocol. These constants are found in **scrplus.h** and are defined as follows:

BURST	- use the Burst Network Protocol
BURST_PLUS	- use the Burst Plus Network Protocol
PLATINUM	- use the Platinum Network Protocol
PLATINUM_PLUS	- use the Platinum Plus Network Protocol
GOLD	- use the Gold Network Protocol

SCRAMNet Classic boards do not support PLATINUM Network Protocol mode or any of the Plus protocols. SCRAMNet LX/+ does not support GOLD Network Protocol mode.

RETURNS

Returns the previous Network Protocol constant if successful, or a ‘-1’ if an error has occurred.

ERROR

sp_protocol() will fail if:

- The SCRAMNet registers are not mapped.
- An invalid value is passed as **ProtocolMode**.

5.8.18 sp_rx_id() - Set Receive ID

SYNOPSIS

```
#include <scrplus.h>
int sp_rx_id ( unsigned char NewID )
```

DESCRIPTION

This routine will set the Receiver ID of the node. The node will not remove its own messages if its Transmit ID is different that its Receiver ID. Each node will remove those messages with an ID that matches the Receiver ID of the receiving node.

This routine will only work on SCRAMNet LX/+ boards and is accomplished by writing to CSR3.

ARGUMENTS

The Receiver ID is set to the value passed as **NewID**. The valid range for this value is 0-255.

RETURNS

Returns the previous Receiver ID if successful, or a '-1' if an error has occurred.

ERROR

sp_rx_id() will fail if:

- The SCRAMNet registers are not mapped.
- This routine is called using a SCRAMNet Classic board.

5.8.19 sp_set_cntr() - Set General Purpose Counter Mode

SYNOPSIS

```
#include <scrplus.h>
int sp_set_cntr ( int mode )
```

DESCRIPTION

The *sp_set_cntr* routine allows the general-purpose counter mode to be selected. Reading CSR13 retrieves the value of the counter. There are several modes that can be selected as specified by the **mode** argument. The routine selects these modes by setting bits in CSR8 and CSR9. This routine will clear the value of the counter by writing a ‘0’ to CSR13.

The counter/timer functionality is not supported for SCRAMNet Classic boards.

ARGUMENTS

The **mode** argument determines the timer/counter mode in effect. The constants used to specify the **mode** parameter are defined in **scrplus.h**, and their effect is as follows:

CNTR_ERRORS	- Count Errors. The counter will be incremented every time a **SCRAMNet** error occurs.
CNTR_TRIG	- Count Triggers. The counter will be incremented every time a Trigger 1 or Trigger 2 occurs.
CNTR_TRANSIT	- Transmit Time. When this mode is set the counter will begin counting when the next message is transmitted and stop counting when any messages generated by this node is received.
CNTR_NET_EVENTS	- Network Events. This mode will count incoming network messages.
CNTR_FREERUN	- Free Run @ 26.66 ns. This mode will increment the counter using an internal 37.5 MHz clock. Counter will roll over every 1.748 ms.
CNTR_FREERUN_T2	- Free Run @ 1.706 µs with Trigger 2 clear. This mode will increment the counter using 585.9 KHz clock. Counter will roll over every 111.8 ms. The assertion of Trigger 2 will clear the counter.

RETURNS

Returns a ‘0’ if successful, or a ‘-1’ if the **mode** value is invalid.

ERROR

sp_set_cntr() will fail if:

- The SCRAMNet registers are not mapped.
- This routine is called using a SCRAMNet Classic Board.

5.8.20 sp_set_sm_addr() - Set Physical Memory Address

SYNOPSIS

```
#include <scrplus.h>
int sp_set_sm_addr ( unsigned long addr )
```

DESCRIPTION

This routine will set the SCRAMNet physical address in CSRs 10 and 11 and set the Shared Memory Access Enable bit in CSR10. After this routine is called with a valid address, the SCRAMNet shared memory is available for access. The serial EEPROM can be programmed to pre-load the correct address upon power-up. See the *Hardware Reference Manual* for more details on the SCRAMNet Memory Address.

This routine does not apply to PCI, EISA, and ISA boards.

This routine does not work on the SCRAMNet Classic board.

ARGUMENTS

The **addr** argument is the physical address of SCRAMNet memory. The address must be “open” on the system address bus (i.e. no conflicts with the other memory). The value of the address must be on an even memory size boundary with the amount of shared memory contained on the SCRAMNet device (i.e. a 128 KB SCRAMNet product can only be mapped on a multiple of a 0x0002 0000 boundary).

RETURNS

Returns a ‘0’ if successful, or a ‘-1’ if an error has occurred.

ERROR

sp_set_sm_addr() will fail if:

- The SCRAMNet registers are not mapped.
- This routine is called using a SCRAMNet Classic board.

5.8.21 sp_set_vp() - Set Virtual Page Number

SYNOPSIS

```
#include <scrplus.h>
int sp_set_vp ( int request, unsigned pagenumber )
```

DESCRIPTION

This routine can be used to set, clear, or READ the virtual page number.

ARGUMENTS

The **request** argument can be passed as the following:

VP_SET	- This tells the routine to enable virtual paging and to set the virtual page to the value of the **pagenumber** argument.
VP_READ	- READs and returns the virtual page if virtual paging was enabled.
VP_RESET	- This will disable virtual paging and clears the virtual page number.

The value of the **pagenumber** argument is only relevant if the **request** argument is *VP_SET* in which case it is used to set the virtual page number. The valid range for **pagenumber** is dependent on the on-board memory size as follows:

valid virtual page range = (0..(8 M/on-board memory size) - 1).

On-board Memory	Virtual Page Range
4 K	(0..2047)
128 K	(0..63)
512 K	(0..15)
1 M	(0..7)
2 M	(0..3)
8 M	(0)

RETURNS

Returns one of the following values for a successful call, depending on the **request** argument:

VP_SET	- returns the value of the virtual page register after setting it.
VP_READ	- returns the value of the virtual page register.
VP_RESET	- returns the value of the virtual page register after clearing it (should be 0).

Returns a '-1' if an error has occurred.

ERROR

sp_set_vp() will fail if:

- The SCRAMNet registers are not mapped.
- This routine is called using a SCRAMNet Classic board.
- The **request** argument is invalid.
- The **request** argument is *VP_SET* and the **pagenumber** argument is invalid.

5.8.22 sp_txrx_id() - Set Transmit / Receive Node ID

SYNOPSIS

```
#include <scrplus.h>
int sp_txrx_id ( unsigned char NewID )
```

DESCRIPTION

This routine will set both the Transmit and Receiver ID of the node. This will result in the node removing any of its own messages from the network when received. If the Receiver ID is set to be different from the Transmit ID (through *sp_rx_id()*) then the node will not remove its own messages, but only those messages with an ID that matches the Receiver ID of the receiving node.

This routine will only work on SCRAMNet LX/+ boards and is accomplished by writing to CSR3.

ARGUMENTS

The Transmit and Receive ID is set to the value passed as **NewID**. The valid range for this value is 0-255.

RETURNS

Returns the previous ID if successful, or a '-1' if an error has occurred.

ERROR

sp_txrx_id() will fail if:

- The SCRAMNet registers are not mapped.
- This routine is called using a SCRAMNet Classic Board.

This page intentionally left blank

GLOSSARY

byte	8 bits.
CSR	Control/Status Register.
FIFO	First-In, First-Out data buffer.
Kbytes	1024 bytes.
LAN	Local Area Network.
longword	32-bit or 4-byte word.
LSB	Least Significant Byte or Bit.
LSHLW	Least Significant Half of a Longword.
LSP	Least Significant Portion.
MSB	Most Significant Byte or Bit.
MSHLW	Most Significant Half of a Longword.
MSP	Most Significant Portion.
NETWORK	A means of serial communications among otherwise unrelated processors.
NODE	A SCRAMNet board which is the point of interface between the host processor and the network.
NODE ID	Network identification number of a given SCRAMNet node (0-255).
ns	Nanosecond.
OS	Operating System.
I/O PAGE	Block of address space which contains the CSRs for the processor and interface devices.
SCRAMNet	Shared Common Random Access Memory Network.
Platinum	Multiple messages with error correction.
Plus	Message size up to 1024 bytes.
word	16-bit or 2-byte word.

This page intentionally left blank