Working Draft American National Standard

Project 04-284r1 (T10/1740-D)

Revision 0
17 January 2005

Information technology - Serial Attached SCSI Driver Interface (SDI)

T10 Technical Editor: Robert C Elliott
Hewlett-Packard Corporation
MC 140801
PO Box 692000
Houston, TX 77269-2000
USA

Telephone: 281-518-5037
Email: elliott@hp.com

Reference number
ISO/IEC 14776-xxx:200x
ANSI INCITS *-200x**

Printed 12:43 AM Monday 17 January 2005

Points of contact

International Committee for Information Technology Standards (INCITS) T10 Technical Committee

T10 Chair
John B. Lohmeyer
LSI Logic
4420 Arrows West Drive
Colorado Springs, CO 80907-3444
USA

Telephone: (719) 533-7560
Email: lohmeyer@t10.org

T10 Vice-Chair
George O. Penokie
IBM Corporation
MS: 2C6
3605 Highway 52 N
Rochester, MN 55901
USA

Telephone: (507) 253-5208
Email: gop@us.ibm.com

T10 Web Site: http://www.t10.org

T10 E-mail reflector:
Server: majordomo@t10.org
To subscribe send e-mail with ‘subscribe’ in message body
To unsubscribe send e-mail with ‘unsubscribe’ in message body

INCITS Secretariat
Suite 200
1250 Eye Street, NW
Washington, DC 20005
USA

Telephone: 202-737-8888
Web site: http://www.incits.org
Email: incits@itic.org

Information Technology Industry Council
Web site: http://www.itic.org

Document Distribution
INCITS Online Store
managed by Techstreet
1327 Jones Drive
Ann Arbor, MI 48105
USA

Web site: http://www.techstreet.com/incits.html
Telephone: (734) 302-7801 or (800) 699-9277

Global Engineering Documents, an IHS Company
15 Inverness Way East
Englewood, CO 80112-5704
USA

Web site: http://global.ihs.com
Telephone: (303) 397-7956 or (303) 792-2181 or (800) 854-7179

American National Standard
for Information Technology

Serial Attached SCSI Driver Interface (SDI)

Secretariat
Information Technology Industry Council

Approved mm.dd.yy

American National Standards Institute, Inc.

ABSTRACT

This standard specifies an interface for Serial Attached SCSI (SAS) and Serial ATA (SATA) host bus adapter (HBA) drivers to allow management and diagnostic programs to query and control the HBA and request that it send SSP (Serial SCSI Protocol), STP (Serial ATA Tunneling Protocol), SMP (Serial Management Protocol), and Serial ATA frames.

American National Standard

Approval of an American National Standard requires verification by ANSI that the requirements for due process, consensus, and other criteria for approval have been met by the standards developer. Consensus is established when, in the judgment of the ANSI Board of Standards Review, substantial agreement has been reached by directly and materially affected interests. Substantial agreement means much more than a simple majority, but not necessarily unanimity. Consensus requires that all views and objections be considered, and that effort be made towards their resolution.

The use of American National Standards is completely voluntary; their existence does not in any respect preclude anyone, whether he has approved the standards or not, from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standards.

The American National Standards Institute does not develop standards and will in no circumstances give interpretation on any American National Standard. Moreover, no person shall have the right or authority to issue an interpretation of an American National Standard in the name of the American National Standards Institute. Requests for interpretations should be addressed to the secretariat or sponsor whose name appears on the title page of this standard.

CAUTION NOTICE: This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken periodically to reaffirm, revise, or withdraw this standard. Purchasers of American National Standards may receive current information on all standards by calling or writing the American National Standards Institute.

CAUTION: The developers of this standard have requested that holders of patents that may be required for the implementation of the standard, disclose such patents to the publisher. However, neither the developers nor the publisher have undertaken a patent search in order to identify which, if any, patents may apply to this standard. As of the date of publication of this standard, following calls for the identification of patents that may be required for the implementation of the standard, no such claims have been made. No further patent search is conducted by the developer or the publisher in respect to any standard it processes. No representation is made or implied that licenses are not required to avoid infringement in the use of this standard.

Published by
American National Standards Institute
11 W. 42nd Street, New York, New York 10036

Printed in the United States of America

Revision Information

R.1 Revision 04-284r1 (17 January 2005)

First revision in FrameMaker based on:

a) 04-245r1 Project proposal for Serial Attached SCSI Driver Interface (SDI) (Steve Fairchild and Rob Elliott, HP)
b) 04-284r0 Common Storage Management Interface (Steve Fairchild, HP)

Lots of editors notes identify areas for discussion by T10.

Contents

Page

1 Scope 1

2 Normative References 2
2.1 Normative references overview 2
2.2 Approved references 2
2.3 References under development 3
2.4 Other references 3

3 Definitions, symbols, abbreviations, keywords, and conventions 4
3.1 Definitions 4
3.2 Symbols and abbreviations 5
3.3 Keywords 5
3.4 Conventions 6

4 General 8
4.1 General overview 8
4.2 Microsoft® Windows® 8
4.2.1 Platform requirements 8
4.2.2 Function invocation 9
4.2.3 Input 9
4.2.4 Output 10
4.2.5 Structure Definitions 10
4.2.6 Security 11
4.3 Linux® 12
4.3.1 Function invocation 12
4.3.2 Input 12
4.3.3 Output 12
4.3.4 Structure Definitions 13
4.3.5 Security 13
4.4 Novell® NetWare® 13
4.4.1 Platform requriements 13
4.4.2 Function invocation 13
4.4.3 HACB Usage 13
4.4.4 Input 14
4.4.5 Output 14
4.4.6 Structure Definitions 14
4.4.7 Security 15

5 Return codes 16
5.1 Return codes 16

6 SDI functions 19
6.1 SDI functions overview 19
6.2 CC_SDI_GET_DRIVER_INFO 21
6.2.1 Behavior 21
6.2.2 Input 21
6.2.3 Output 21
6.2.4 Structure Definitions 22
6.3 CC_SDI_GET_CNTLR_CONFIG 22
6.3.1 Behavior 22
6.3.2 Input 22
6.3.3 Output 23
6.3.4 Structure Definitions 24
6.4 CC_SDI_GET_CNTLR_STATUS 25

6.4.1 Behavior 25
6.4.2 Input 25
6.4.3 Output 26
6.4.4 Structure Definitions 26
6.5 CC_SDI_FIRMWARE_DOWNLOAD 26
6.5.1 Behavior 26
6.5.2 Input 27
6.5.3 Output 27
6.5.4 Structure Definitions 28
6.6 CC_SDI_GET_RAID_INFO 28
6.6.1 Behavior 28
6.6.2 Input 28
6.6.3 Output 28
6.6.4 Structure Definitions 28
6.7 CC_SDI_GET_RAID_CONFIG 29
6.7.1 Behavior 29
6.7.2 Input 29
6.7.3 Output 29
6.7.4 Structure Definitions 31
6.8 CC_SDI_GET_PHY_INFO 32
6.8.1 Behavior 32
6.8.2 Input 32
6.8.3 Output 32
6.8.4 Structure Definitions 35
6.9 CC_SDI_SET_PHY_INFO 36
6.9.1 Behavior 36
6.9.2 Input 36
6.9.3 Output 37
6.9.4 Structure Definitions 37
6.10 CC_SDI_GET_LINK_ERRORS 37
6.10.1 Behavior 37
6.10.2 Input 37
6.10.3 Output 38
6.10.4 Structure Definitions 38
6.11 CC_SDI_SMP_PASSTHROUGH 38
6.11.1 Behavior 38
6.11.2 Security 38
6.11.3 Input 38
6.11.4 Output 39
6.11.5 Structure Definitions 40
6.12 CC_SDI_SSP_PASSTHROUGH 41
6.12.1 Behavior 41
6.12.2 Security 42
6.12.3 Input 42
6.12.4 Output 44
6.12.5 Structure Definitions 45
6.13 CC_SDI_STP_PASSTHROUGH 45
6.13.1 Behavior 45
6.13.2 Security 46
6.13.3 Input 46
6.13.4 Output 47
6.13.5 Structure Definitions 48
6.14 CC_SDI_GET_SATA_SIGNATURE 49
6.14.1 Behavior 49
6.14.2 Input 49
6.14.3 Output 49
6.14.4 Structure Definitions 49

6.15 CC_SDI_GET_SCSI_ADDRESS 49
6.15.1 Behavior 49
6.15.2 Input 50
6.15.3 Structure Definitions 50
6.16 CC_SDI_GET_DEVICE_ADDRESS 50
6.16.1 Behavior 50
6.16.2 Input 51
6.16.3 Output 51
6.16.4 Structure Definitions 51
6.17 CC_SDI_TASK_MANAGEMENT 51
6.17.1 Behavior 51
6.17.2 Security 52
6.17.3 Input 52
6.17.4 Output 53
6.17.5 Structure Definitions 53
6.18 CC_SDI_PHY_CONTROL 54
6.18.1 Behavior 54
6.18.2 Security 54
6.18.3 Spinup behavior model 54
6.18.4 Phy signal control behavior model 55
6.18.5 Input 55
6.18.6 Output 58
6.18.7 Structure Definitions 59
6.19 CC_SDI_GET_CONNECTOR_INFO 60
6.19.1 Behavior 60
6.19.2 Input 60
6.19.3 Output 60
6.19.4 Structure Definitions 61

Annex A Header file 62
A.1 Header file 62

Tables

Page

1 Standards bodies 2
2 ISO and American numbering conventions 7
3 Parameter naming convention 7
4 SDI security levels 8
5 Windows to SDI data type mapping 10
6 Windows registry encoding of SDI security levels 12
7 ReturnCode field 16
8 SDI functions 20
9 SCSI commands allowed without full security access 42
10 ATA commands allowed without full security access 46

Figures

Page

Foreword (This foreword is not part of this standard)

Requests for interpretation, suggestions for improvement and addenda, or defect reports are welcome. They should be sent to the INCITS Secretariat, International Committee for Information Technology Standards, Information Technology Institute, 1250 Eye Street, NW, Suite 200, Washington, DC 20005-3922.

This standard was processed and approved for submittal to ANSI by the International Committee for Information Technology Standards (INCITS). Committee approval of the standard does not necessarily imply that all committee members voted for approval. At the time it approved this standard, INCITS had the following members:

Karen Higginbottom, Chair

David Michael, Vice-Chair

INCITS Technical Committee T10 on Lower Level Interfaces, which developed and reviewed this standard, had the following members:

John B. Lohmeyer, Chair

George O. Penokie, Vice-Chair

Ralph O. Weber, Secretary

Introduction

The standard is organized as follows:

Clause 1 (Scope) describes the relationship of this standard to the SCSI family of standards.
Clause 2 (Normative References) provides references to other standards and documents.
Clause 3 (Definitions, symbols, abbreviations, keywords, and conventions) defines terms and conventions used throughout this standard.
Clause 4 (General) provides a general overview.
Clause 5 (Return codes) defines the SDI return codes.
Clause 6 (SDI functions) defines the SDI functions.

American National Standard
for Information Technology -

Serial Attached SCSI Driver Interface (SDI)

1 Scope

This standard specifies an interface for Serial Attached SCSI (SAS) and Serial ATA (SATA) host bus adapter (HBA) drivers to allow management and diagnostic programs to query and control the HBA and request that it send SSP (Serial SCSI Protocol), STP (Serial ATA Tunneling Protocol), SMP (Serial Management Protocol), and Serial ATA frames.

2 Normative References

2.1 Normative references overview

The following standards contain provisions that, by reference in the text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below.

Copies of the following documents may be obtained from ANSI:

a) approved ANSI standards;
b) approved and draft international and regional standards (e.g., ISO, IEC, CEN/CENELEC, ITU-T); and
c) approved and draft foreign standards (e.g., BSI, JIS, and DIN).

For further information, contact ANSI Customer Service Department at 212-642-4900 (phone), 212-302-1286 (fax) or via the World Wide Web at http://www.ansi.org.

Additional availability contact information is provided below as needed.

Table 1 lists standards bodies and their web sites.

Table 1 — Standards bodies

Abbreviation	Standards body	Web site
ANSI	American National Standards Institute	http://www.ansi.org
BSI	British Standards Institution	http://www.bsi-global.com
CEN	European Committee for Standardization	http://www.cenorm.be
CENELEC	European Committee for Electrotechnical Standardization	http://www.cenelec.org
DIN	German Institute for Standardization	http://www.din.de
IEC	International Engineering Consortium	http://www.iec.ch
IEEE	Institute of Electrical and Electronics Engineers	http://www.ieee.org
IETF	Internet Engineering Task Force	http://www.ietf.org
INCITS	International Committee for Information Technology Standards	http://www.incits.org
ISO	International Standards Organization	http://www.iso.ch
ITI	Information Technology Industry Council	http://www.itic.org
ITU-T	International Telecommunications Union Telecommunications Standardization Sector	http://www.itu.int
JIS	Japanese Industrial Standards Committee	http://www.jisc.org
T10	INCITS T10 Committee - SCSI storage interfaces	http://www.t10.org
T11	INCITS T11 Committee - Fibre Channel interfaces	http://www.t11.org
T13	INCITS T13 Committee - ATA storage interface	http://www.t13.org

2.2 Approved references

At the time of publication, the following referenced standards were approved:

ISO/IEC 9899:1999, *Programming Languages - C*
ISO/IEC 9899:1999 Cor. 1:2001, *Technical Corrigendum 1*

2.3 References under development

At the time of publication, the following referenced standards were still under development. For information on the current status of the documents, or regarding availability, contact the relevant standards body as indicated.

ISO/IEC 14776-413, *SCSI Architecture Model - 3 (SAM-3) standard* (T10/1561-D)
ISO/IEC 14776-453, *SCSI Primary Commands - 3 (SPC-3) standard* (T10/1416-D)
ISO/IEC 14776-372, *SCSI Enclosure Services - 2 (SES-2) standard* (T10/1559-D)

NOTE 1 - For more information on the current status of the document, contact the INCITS Secretariat at 202-737-8888 (telephone), 202-638-4922 (fax) or via Email at incits@itic.org. To obtain copies of this document, contact Global Engineering at 15 Inverness Way East Englewood, CO 80112-5704 at 800-854-7179 (telephone), 303-792-2181 (telephone), or 303-792-2192 (fax).

2.4 Other references

Information on the Microsoft® Windows® operating system is available on http://www.microsoft.com. Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries. Specific references:

Microsoft Windows Server™ 2003 Driver Development Kit (DDK). See http://www.microsoft.com/whdc/devtools/ddk/default.mspx.
Platform Software Development Kit (SDK). See ???

Information on the Linux® operating system is available on http://www.kernel.org. Linux is a registered trademark of Linus Torvalds. Specific references:

Linux 2.4 SCSI subsystem HOWTO. Revision 2.1, by Douglas Gilbert. See http://sg.torque.net/scsi/SCSI-2.4-HOWTO.

Information on the Novell® NetWare® operating system is available on http://www.novell.com. Novell and NetWare are registered trademarks of Novell, Inc. in the United States and/or other countries. Specific references:

Novell NetWare Developers Kit. See http://developer.novell.com/ndk/doc.htm)
Novell NetWare Peripheral Architecture (NWPA). See http://developer.novell.com/ndk/doc/storarch.

Editor's Note 1: TM = Ctrl-Q *, (R) = Ctrl-Q (

3 Definitions, symbols, abbreviations, keywords, and conventions

3.1 Definitions

3.1.1 application client: An object that is the source of SCSI commands. See SAM-3.

3.1.2 byte: A sequence of eight contiguous bits considered as a unit.

3.1.3 command: A request describing a unit of work to be performed by a device server. See SAM-3.

3.1.4 command descriptor block (CDB): The structure used to communicate commands from an application client to a device server. See SPC-3.

3.1.5 data-in buffer: The buffer identified by the application client to receive data from the device server during the processing of a command. See SAM-3.

3.1.6 data-out buffer: The buffer identified by the application client to supply data that is sent from the application client to the device server during the processing of a command. See SAM-3.

3.1.7 device server: An object within a logical unit that processes SCSI tasks according to the rules of task management. See SAM-3.

3.1.8 device type: The type of device (or device model) implemented by the device server as indicated by the PERIPHERAL DEVICE TYPE field of the standard INQUIRY data. See SPC-3.

3.1.9 direct-access block device: A device that is capable of containing data stored in blocks that each have a unique logical block address.

3.1.10 domain: An I/O system consisting of a set of SCSI devices that interact with one another by means of a service delivery subsystem. See SAM-3.

3.1.11 field: A group of one or more contiguous bits, a part of a larger structure such as a CDB (see 3.1.4) or sense data (see SPC-3).

3.1.12 hard reset: A condition resulting from the events defined by SAM-3 in which the SCSI device performs the hard reset operations described in SAM-3, SPC-3, SES-2 (if applicable), and this standard.

3.1.13 I_T nexus loss: A condition resulting from the events defined by SAM-3 in which the SCSI device performs the I_T nexus loss operations described in SAM-3, SPC-3, SES-2 (if applicable), and this standard.

3.1.14 logical unit (LU): An externally addressable entity within a target that implements a SCSI device model and contains a device server. A detailed definition of a logical unit may be found in SAM-3.

3.1.15 logical unit number (LUN): An encoded 64-bit identifier for a logical unit. A detailed definition of a logical unit number may be found in SAM-3.

3.1.16 logical unit reset: A condition resulting from the events defined by SAM-3 in which the logical unit performs the logical unit reset operations described in SAM-3, SPC-3, SES-2 (if applicable), and this standard.

3.1.17 power cycle: Power being removed followed by power being applied to a SCSI device.

3.1.18 power on: A condition resulting from the events defined by SAM-3 in which the SCSI device performs the power on operations described in SAM-3, SPC-3, SES-2 (if applicable), and this standard.

3.1.19 sense data: Data describing an error or exceptional condition that a device server delivers to an application client in association with CHECK CONDITION status. See SPC-3.

3.1.20 status: One byte of response information sent from a device server to an application client upon completion of each command. See SAM-3.

3.2 Symbols and abbreviations

See table 1 for abbreviations of standards bodies (e.g., ISO). Additional symbols and abbreviations used in this standard include:

Abbreviation	Meaning
CDB	command descriptor block (see 3.1.4)
FCP	Fibre Channel Protocol (revision not relevant)
FCP-3	Fibre Channel Protocol - 3 standard
I/O	input/output
iSCSI	Internet SCSI standard
LSB	least significant bit
LU	logical unit (see 3.1.14)
LUN	logical unit number (see 3.1.15)
MSB	most significant bit
SAM-3	SCSI Architecture Model - 3 standard
SAS	Serial Attached SCSI (revision not relevant)
SAS-1.1	Serial Attached SCSI - 1.1 standard
SCSI	Small Computer System Interface family of standards
SCC-2	SCSI-3 Controller Commands - 2 standard
SES-2	SCSI Enclosure Services - 2 standard
SPC-3	SCSI Primary Commands - 3 standard

3.3 Keywords

3.3.1 can: A keyword used for statements of possibility and capability indicating a condition that is required to be handled (equivalent “it is possible to”).

3.3.2 cannot: A keyword used for statements of possibility and capability indicating a condition that is not required to be handled (equivalent “it is not possible to”).

NOTE 2 - “May” signifies permission expressed by this standard, whereas “can” refers the ability of a device compliant with this standard to handle events outside of control of this standard.

3.3.3 expected: A keyword used to describe the behavior of the hardware or software in the design models assumed by this standard. Other hardware and software design models may also be implemented.

3.3.4 ignored: A keyword used to describe an unused bit, byte, word, field or code value. The contents or value of an ignored bit, byte, word, field or code value shall not be examined by the receiving SCSI device and may be set to any value by the transmitting SCSI device.

3.3.5 invalid: A keyword used to describe an illegal or unsupported bit, byte, word, field or code value. Receipt of an invalid bit, byte, word, field or code value shall be reported as an error.

3.3.6 mandatory: A keyword indicating an item that is required to be implemented as defined in this standard.

3.3.7 may: A keyword that indicates flexibility of choice with no implied preference; equivalent to “may or may not” and equivalent to the phrase “it is permitted.”

3.3.8 may not: Keywords that indicate flexibility of choice with no implied preference; equivalent to “may or may not” and equivalent to the phrase “it is permitted.”

3.3.9 need not: Keywords indicating a feature that is not required to be implemented; equivalent to “is not required that.”

3.3.10 obsolete: A keyword indicating that an item was defined in prior SCSI standards but has been removed from this standard.

3.3.11 optional: A keyword that describes features that are not required to be implemented by this standard. However, if any optional feature defined by this standard is implemented, then it shall be implemented as defined by this standard.

3.3.12 reserved: A keyword referring to bits, bytes, words, fields and code values that are set aside for future standardization. A reserved bit, byte, word or field shall be set to zero, or in accordance with a future extension to this standard. Recipients are not required to check reserved bits, bytes, words or fields for zero values. Receipt of reserved code values in defined fields shall be reported as an error.

3.3.13 restricted: A keyword referring to bits, bytes, words, and fields that are set aside for use in other SCSI standards. A restricted bit, byte, word, or field shall be treated as a reserved bit, byte, word or field for the purposes of the requirements defined in this standard.

3.3.14 shall: A keyword indicating a mandatory requirement. Designers are required to implement all such mandatory requirements to ensure interoperability with other products that conform to this standard.

3.3.15 should: A keyword indicating flexibility of choice with a strongly preferred alternative; equivalent to the phrase “it is strongly recommended.”

3.3.16 vendor-specific: Something (e.g., a bit, field, or code value) that is not defined by this standard and may be used differently in various implementations.

3.4 Conventions

Certain words and terms used in this standard have a specific meaning beyond the normal English meaning. These words and terms are defined either in this clause or in the text where they first appear.

Names of commands are in all uppercase (e.g., INQUIRY or IDENTIFY DEVICE).

Names of fields and state variables are in small uppercase (e.g. NAME). When a field or state variable name contains acronyms, uppercase letters may be used for readability. Normal case is used when the contents of a field or state variable are being discussed. Fields or state variables containing only one bit are usually referred to as the NAME bit instead of the NAME field.

Normal case is used for words having the normal English meaning.

The American convention of numbering is used (i.e., the thousands and higher multiples are separated by a comma and a period is used as the decimal point). Table 2 shows a comparison of the ISO and American numbering conventions.

Table 2 — ISO and American numbering conventions

ISO	American
0,6	0.6
3,141 592 65	3.14159265
1 000	1,000
1 323 462,95	1,323,462.95

Numbers that are not immediately followed by lower-case b or h are decimal values.

Numbers immediately followed by lower-case b (e.g., 0101b) are binary values. Underscores may be included in binary values to increase readability or delineate field boundaries (e.g., 0101_1010b).

A sequence of numbers or upper case letters ‘A’ through ‘F’ immediately followed by lower-case h (e.g., FA23h) are hexadecimal values. Underscores may be included in hexadecimal values to increase readability or delineate field boundaries (e.g., FD8C_FA23h).

The prefix ‘0x’ followed by a sequence of numbers or upper case letters ‘A’ through ‘F’ (e.g., 0xFA23) is a hexadecimal value. Underscores may be included in hexadecimal values to increase readability or delineate field boundaries (e.g., 0xFD8C_FA23).

Lists sequenced by letters (e.g., a) red, b) blue, c) green) show no ordering relationship between the listed items. Numbered lists (e.g., 1) red, 2) blue, 3) green) show an ordering between the listed items.

If a conflict arises between text, tables or figures, the order of precedence to resolve the conflicts is text, then tables, and finally figures. Not all tables or figures are fully described in the text. Tables show data format and values.

Notes do not constitute any requirements for implementers.

Table 3 shows the convention for parameter names in data structures.

Table 3 — Parameter naming convention

Prefix	Description
b	char or unsigned char (8 bits)
us	unsigned short (16 bits)
u	unsigned long (32 bits)
sz	ASCII string terminated with a NULL character (0x00)

Editor’s Note 2: should Hungarian notation be used or not? If so, what prefixes? b usually means boolean, not byte; by can be used for byte and c for char. l is for long, not just u. Linux kernel coding style guidelines recommend against Hungarian - if the .h file is destined there, it may be best to avoid it.

4 General

4.1 General overview

This document is intended to define a Serial Attached SCSI Driver Interface (SDI) composed of a set of function codes, definitions, data structures and return codes that a Windows, Linux or Netware driver should implement to provide a standard mechanism for accessing the physical components within a Serial Attached SCSI or Serial ATA domain.

The SDI function codes and submission mechanism is dependent on the OS platform, but is often based on device I/O controls (i.e., IOCTLs). The definitions, data structures, return codes and functions are independent of the OS platform.

The SDI data structure that defines the SDI function uses a platform-specific header structure. To allow a common C language header file to define SDI, the header structure is named IOCTL_HEADER. While the name is the same across OS platforms, the actual content of the IOCTL_HEADER data structure is unique to the OS platform. The application needs to be aware of the OS platform in order to properly access the elements of the IOCTL_HEADER structure - some of the field names are common (e.g., *ReturnCode*, *Length*, and *Timeout*) while others are platform-specific.

All drivers should make an accessible device node available for the controller, even if no physical devices are registered with the SCSI subsystem.

SDI security levels are defined in table 4.

Table 4 — SDI security levels

SDI security level	Description
None	Access to all SDI functions is prohibited.
Restricted	Access to some SDI functions (e.g., reads) is allowed; access to others (e.g., writes) is prohibited.
Limited	Access to some SDI functions (e.g., reads and firmware downloads) is allowed; access to others (e.g., writes) is prohibited.
Full	Access to all SDI functions allowed

Editor's Note 3: these names are sometimes confusing - Full could be interpreted as "the user must have full access rights" while none could be viewed as "no special permission necessary"

4.2 Microsoft® Windows®

4.2.1 Platform requirements

The driver may be a SCSIPort or StorPort based miniport driver.

SCSIPort drivers in Windows Server 2003 and later operating systems shall set HKLM\System\CurrentControlSet\Services\<ServiceName>\Parameters\Device\CreateInitiatorLU to 1, so the port driver accepts requests even if no device is connected to the controller. either. SCSIPort drivers in older operating systems shall create a pseudo-LUN to provide access. StorPort drives need not do either.

Editor's Note 4: investigate above some more

4.2.2 Function invocation

For Windows, SDI is defined as a set of function codes that are submitted using the **DeviceIoControl** function (defined in <winbase.h>; included from <windows.h>)(see the Windows Platform SDK):

```
BOOL DeviceIoControl {
      HANDLE hDevice;
      DWORD dwIoControlCode;
      LPVOID lpInBuffer;
      DWORD nInBufferSize;
      LPVOID lpOutBuffer;
      DWORD nOutBufferSize;
      LPDWORD lpBytesReturned;
      LPOVERLAPPED lpOverlapped;
);
```

with the following parameters:

a) *hDevice:* specifies the handle of a device managed by the device driver;
b) *dwIoControlCode*: set to IOCTL_SCSI_MINIPORT (defined in <ntddscsi.h>);
c) *lpInBuffer*: points to an input buffer containing an IOCTL_HEADER (i.e., SRB_IO_CONTROL) data structure that contains the specific SDI function code being requested and any necessary input data;
d) *nInBufferSize*: specifies the size of the input buffer data structure in bytes;
e) *lpOutBuffer*: points to an output buffer to receive an IOCTL_HEADER (i.e., SRB_IO_CONTROL) data structure;
f) *nOutBufferSize*: specifies the size of the output buffer in bytes;
g) *lpBytesReturned*: points to a variable to receive the size of the output buffer; and
h) *lpOverlapped*: points to an OVERLAPPED structure if the device was opened with the FILE_FLAG_OVERLAPPED flag.

This request is forwarded to the miniport driver (conceptually with HwStorBuildIo and HwStorStartIo for StorPort (see <storport.h>) or just HwScsiStartIo for SCSIPort (see <scsiport.h>)) as a SCSI_REQUEST_BLOCK (defined in <srb.h>) with the *Function* field set to SRB_FUNCTION_IO_CONTROL. Only the *SrbFlags*, *TimeOutValue*, *DataBuffer*, and *DataTransferLength* fields are used.

If a SDI function code is not supported, the **DeviceIoControl** function shall return a 1 indicating function success and set the *IoctlHeader.ReturnCode* field to SDI_STATUS_BAD_CNTL_CODE.

If the SDI buffer provided is too small, then the **DeviceIoControl** function shall return a 1 indicating success and set the *IoctlHeader.ReturnCode* to SDI_STATUS_INVALID_PARAMETER.

Editor's Note 5: should boolean values be referenced with defines TRUE and FALSE rather than 0 and 1?

4.2.3 Input

The **DeviceIoControl** function with the IOCTL_SCSI_MINIPORT control code accepts an IOCTL_HEADER (i.e., SRB_IO_CONTROL) data structure containing the following fields:

a) *HeaderLength* (platform-specific): Specifies the length of the IOCTL_HEADER data structure (i.e., sizeof (IOCTL_HEADER));
b) *Signature* (platform-specific): Specifies a namespace signature, dependent on the SDI function code used. See Security and Enabling Features;
c) *Timeout* (platform-independent): Specifies the time in seconds to wait before the SDI function is considered to have failed. See Timeouts;
d) *ControlCode* (platform-specific): Specifies which SDI function to execute. Control codes are defined in 6.1;

Editor's Note 6: plan to rename ControlCode to FunctionCode everywhere possible to avoid confusion with the Windows DeviceIoControl function dwIoControlCode argument. In Linux, the value is passed in a request argument.

e) *ReturnCode* (platform-independent): This field shall be set to 0;
f) *Length* (platform-independent): Specifies the length in bytes of the SDI data structure buffer that immediately follows the IOCTL_HEADER data structure. This field should be set to at least (sizeof(SDI_xxx_BUFFER) – sizeof(IOCTL_HEADER)) where xxx_xxx is associated with the SDI function name. A larger buffer may be supplied.

4.2.4 Output

The **DeviceIoControl** function with the IOCTL_SCSI_MINIPORT control code shall return an IOCTL_HEADER (i.e., SRB_IO_CONTROL) data structure with the following fields:

a) *HeaderLength*: Same as input;
b) *Signature*: Same as input;
c) *Timeout*: Same as input;
d) *ControlCode*: Same as input;
e) *ReturnCode*: indicates the resulting status of the SDI function. Return codes are defined in 5.1; and
f) *Length*: Same as input.

4.2.5 Structure Definitions

Editor's Note 7: move structure definitions ahead of the place where they are first referenced. For example, the _SRB_IO_CONTROL structure would move ahead of the Input and Output sections.

For Windows, the SRB_IO_CONTROL data structure is used as the IOCTL_HEADER data structure. The SRB_IO_CONTROL uses the standard Windows data types for its members. Table 5 shows how the Windows data types in the SRB_IO_CONTROL correspond to SDI data types.

Table 5 — Windows to SDI data type mapping

Windows data type	SDI data type
UCHAR	__u8
CHAR	__i8
USHORT	__u16
ULONG	__u32

Editor's Note 8: (not Windows specific): the is at least one 64 bit data structure defined (BaseMemoryAddress in SDI_CNTLR_CONFIG), but __u64 is not used for them. That is not ideal for 64-bit and/or big-endian processors. That structure forces the four least significant bytes of the 8-byte address to be in the first/lowest 4 bytes of storage (bytes 0-3); this is not the way a big-endian processor would normally store a 64-bit memory address (where the most significant byte should be at byte 0).

The following data structures are used (defined in Windows <ntddscsi.h>):

```
typedef struct _SRB_IO_CONTROL {
      ULONG HeaderLength;
```

```
        UCHAR Signature[8];
        ULONG Timeout;
        ULONG ControlCode;
        ULONG ReturnCode;
        ULONG Length;
} SRB_IO_CONTROL, *PSRB_IO_CONTROL;
```

4.2.6 Security

Since the DeviceIoControl function IOCTL_SCSI_MINIPORT control code is not protected, the driver shall use the **DriverParameters** registry value (for SCSIPort) or **DriverParameter** registry value of the miniport driver registry definition (see the Windows DDK) to identify which SDI functions are allowed.

Editor's Note 9: Windows protects IOCTL_SCSI_PASSTHROUGH by requiring the application to have both read and write access to the device. IOCTL_SCSI_MINIPORT has the same protection - in ntddscsi.h the ControlCode is defined similarly:

#define IOCTL_SCSI_PASSTHROUGH CTL_CODE(IOCTL_SCSI_BASE, 0x0401, METHOD_BUFFERED, FILE_READ_ACCESS | FILE_WRITE_ACCESS);

#define IOCTL_SCSI__MINIPORT CTL_CODE(IOCTL_SCSI_BASE, 0x0402, METHOD_BUFFERED, FILE_READ_ACCESS | FILE_WRITE_ACCESS);

Although that could imply that SDI does not need additional protection (since the main SCSI passthrough provides no more protection), some miniports apply additional checks on the commands being used. SDI follows that approach.

Editor's Note 10: Instead, could the miniport check if the user has Administrator privileges? That would provide a similar level of security across OSes and avoid the need for this complicated scheme for Windows.

The SDI security registry value shall be delineated from existing **DriverParameters** registry values (for SCSIPort) or **DriverParameter** registry values by using a semicolon (';") before and/or after the SDI security registry value. For example if the **DriverParameter** value already contains "abc def", then after adding the SDI security, the **DriverParameter** value contains "abc def;SDI=Full;".

Editor's Note 11: no standard exists for formatting this string; JNI uses spaces to separate items. Adaptec has used /XXX=yyy.

Editor's Note 12: DriverParameters is not defined by the current DDK, but KnowledgeBase article 133706 mentions it. HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services. Passed in HwScsiFindAdapter as ArgumentString. Either q12x00\Parameters\Devices\DriverParameters to pass the values to all HBAs of that driver type "q12x00"

Editor's Note 13: it appears that SCSIPort uses DriverParameters but StorPort uses DriverParameter.

The registry value content should be identified as valid only if the SDI descriptor matches exactly the ASCII string in table 6.

Table 6 — Windows registry encoding of SDI security levels

SDI security level	Windows Registry DriverParameter value
None	“;SDI=None;”
Restricted	“;SDI=Restricted;”
Limited	“;SDI=Limited;” or no value (i.e., this is the default setting)
Full	“;SDI=Full;”

Editor’s Note 14: CSMI included a path and value for Windows monolithic port drivers (not SCSIPort/StorPort based). Should we bother?

Editor’s Note 15: can only detect these at driver initialization time. Have to reboot (or at least restart the driver) to change the security level when done this way.

4.3 Linux®

4.3.1 Function invocation

For Linux, SDI is defined as a set of function codes that are submitted using the **ioctl** function call (defined in <sys/ioctl.h>):

```
int ioctl (int d; int request, ...)
```

with the following parameters:

a) *d*: specifies an open file descriptor;
b) *request:* specifies which SDI function to execute. Control codes are defined in 6.1; and
c) third argument: specifies a pointer to an input buffer containing an IOCTL_HEADER data structure.

The IOCTL_HEADER is a reference to the typedef of the struct _IOCTL_HEADER on the Linux platform.

4.3.2 Input

The **ioctl** function accepts a IOCTL_HEADER data structure containing the following fields:

a) *IOControllerNumber* (platform-specific): The I/O controller number for drivers that support multiple I/O controllers (i.e., adapters);
b) *Length*: Length of the SDI data structure buffer including IOCTL_HEADER. At a minimum this should be the sizeof(SDI_xxxx_xxxx_BUFFER) associated with the SDI control code. A larger buffer may be supplied;
c) *ReturnCode* (platform-independent): Initialized to 0;
d) *Timeout* (platform-independent): Time in seconds to wait before the SDI function is considered to have failed. See Timeouts; and
e) *Direction* (platform-specific): specifies the direction of data flow through the **ioctl** function. SDI_DATA_READ (i.e., 0) specifies that data be returned by the **ioctl** function. SDI_DATA_WRITE (i.e., 1) specifies that data be provided to the **ioctl** function.

4.3.3 Output

The **ioctl** function shall return 0 for success with a IOCTL_HEADER data structure with the following fields:

a) *IOControllerNumber*. Same as input;

b) *Length*. Same as input;
c) *ReturnCode*: Indicates the resulting status of the SDI function. Return codes are defined in 5.1;
d) *Timeout*. Same as input; and
e) *Direction*. Same as input.

4.3.4 Structure Definitions

The following data structures are used:

```
typedef struct _IOCTL_HEADER {
      __u32 IOControllerNumber;
      __u32 Length;
      __u32 ReturnCode;
      __u32 Timeout;
      __u16 Direction;
} IOCTL_HEADER, *PIOCTL_HEADER;
```

4.3.5 Security

Since the SDI functions can only be issued by an application with root security access, no specific protection mechanisms are required or provided for Linux.

There is also no provision for the namespace signature, because the SDI control codes on the Linux platform should prevent a namespace collision.

4.4 Novell® NetWare®

4.4.1 Platform requriements

There are no platform requirements.

4.4.2 Function invocation

For NetWare, SDI is defined as a set of I/O control codes that are submitted using the **NPA_HACB_Passthru()** API.

The IOCTL_HEADER is a reference to the typedef of the struct _IOCTL_HEADER on the NetWare platform. Definition of this data structure is provided below. Unlike the Windows or Linux versions, this data structure is minimal since most IOCTL details are already contained within the NetWare Peripheral Architecture (NWPA) Host bus Adapter Control Block (HACB) structure. See the Novell Developer's Kit (NDK) for complete information on NWPA and HACB definitions. The information provided here is for convenience only.

4.4.3 HACB Usage

Application NLMs rely on the use of the Novell Media Manager (MM) for discovery of drivers, adapters, and devices. Driver pass through calls are used where necessary to identify and acquire hardware-device specific information. Where passthroughs and MM calls cannot deliver required information, a vendor unique set of Host Bus Adapter Control Block (HACB) IOCtrls (SDI) are defined.

The NWPA specification allows for a number of methods for implementing vendor unique HACB calls. The method defined by this standard is using the HACB *hacbType* field set to 0. The HACB contains a 28-byte *command* overlay area (i.e., union) of HACBStruct to define each vendor unique IOCTL. SDI shall use the *host* data structure in that area:

```
// HACB Command Block Overlay Area
      struct /* HACB Type = 0: Host Adapter Cmd */
      {
            LONG function;
            LONG parameter0;
            LONG parameter1;
            LONG parameter2;
            BYTE reserved[12];
```

```
        } host;
```

The function field is used for IOCTL definition, leaving 24 bytes of space to define additional parameters. These 24 bytes are insufficient to allow for a common usage of the SDI IOCTLs across different operating system platforms. The following will allow for the usage of the SDI IOCTLs within the confines of the NetWare Peripheral Architecture.

All NetWare SDI IOCTLs are issued with the Data_Direction_Bit set to WRITE within the HACB *controlInfo* field (e.g., 0x00000002). The SDI IOCTL data structure buffer is always sent to the driver in the HACB **vDataBufferPtr* field. Upon IOCTL return (to a HACB WRITE) the driver sends the SDI IOCTL data structure buffer back to the calling application using the HACB **vErrorSenseBufferPtr* field.

NOTE 3 - By design the NWPA HACB process is unidirectional; thus when a READ IOCTL is issued, the memory referenced by the HACB *pdataBufferPointer* is only to be used for reading data from the driver, not for transporting data to the driver. When a WRITE IOCTL is issued, the HACB *pdataBufferPointer* is only to be used for sending information to the driver, not for reading data from the driver. However, the *vErrorSenseBufferPtr* is always available as a data transport by the driver for both READ and WRITE operations.

4.4.4 Input

For each SDI command, the HACB data structure fields shall be set as follows:

a) *hacbPutHandle*: Specifies the handle identifying the current HACB;
b) *hacbCompletion*: Initialized to 0;
c) *controlInfo*: The application shall set the Data_Direction_Bit to 1 (i.e., 0x00000002);
d) *hacbType*: Set to 0x0000 (i.e., adapter-specific Host command structures).
e) *timeoutAmount*: Time in seconds to wait before the SDI function is considered to have failed. See Timeouts;
a) *deviceHandle*: NWPA-supplied handle for a specific registered device. Obtained via **NPA_Return_DeviceHandle()**;
b) *dataBufferLen*: Length in bytes of the SDI command data structure buffer;
c) *vDataBufferPtr*: Virtual address pointer to the SDI command data structure buffer. The data structure is SDI IOCTL command dependent;
d) *pDataBufferPtr*: Physical address of the buffer pointed at by vDataBufferPtr;
e) *errorSenseBufferLen*: Same as dataBufferLen;
f) *vErrorSenseBufferPtr*: Same as vDataBufferPtr; and
g) *pErrorSenseBufferPtr*: Physical address of the buffer pointed at by vErrorSenseBufferPtr;
h) *reserved*: Reserved;
i) *hamSpace*: and
j) *command*: Uses the *host* structure:
   A) *command.host.function*: Specifies the SDI function;
   B) *command.parameter0*: Set to 0;
   C) *command.parameter1*: Set to 0;
   D) *command.parameter2*: Set to 0;
   E) *command.reserved*: Set to 0.

4.4.5 Output

For each SDI command, the driver shall return information within the following HACB data structure fields:

a) *hacbCompletion*: Per NWPA specifications, the return status of this HACB;
b) *errorSenseBufferLen*: WORD aligned length of SDI command data structure buffer; and
c) *vErrorSenseBufferPtr*: Virtual address pointer to the SDI command data structure buffer.

4.4.6 Structure Definitions

The following data structures are used:

```
typedef struct _IOCTL_HEADER {
      long lLength; // size SDI IOCTL specific command data structure
```

```
        unsigned long ulReturnCode; // SDI return code
    } IOCTL_HEADER;
```

4.4.7 Security

Since the SDI functions can only be issued by an application with administrative security access, no specific protection mechanisms are required or provided for NetWare platforms.

5 Return codes

5.1 Return codes

Editor's Note 16: make this an early section in chapter 6

Editor's Note 17: consider deleting the list of functions that use each code, which may become a maintenance nightmare

Table 7 defines the return codes that are returned in the *ReturnCode* field of the IOCTL_HEADER structure on completion of a function call.

Table 7 — *ReturnCode* field (part 1 of 3)

SDI return code (SDI_...)	SDI functions that return (CC_SDI_...)	Description
STATUS_SUCCESS	GET_DRIVER_INFO GET_CNTLR_CONFIG GET_CNTLR_STATUS FIRMWARE_DOWNLOAD GET_RAID_INFO GET_RAID_CONFIG GET_PHY_INFO SET_PHY_INFO GET_LINK_ERRORS SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH GET_SATA_SIGNATURE GET_SCSI_ADDRESS GET_DEVICE_ADDRESS TASK_MANAGEMENT GET_CONNECTOR_INFO PHY_CONTROL	SDI function completed successfully.
STATUS_FAILED	GET_DRIVER_INFO GET_CNTLR_CONFIG GET_CNTLR_STATUS GET_RAID_INFO GET_RAID_CONFIG GET_PHY_INFO SET_PHY_INFO GET_LINK_ERRORS SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH GET_SATA_SIGNATURE GET_SCSI_ADDRESS GET_DEVICE_ADDRESS TASK_MANAGEMENT GET_CONNECTOR_INFO PHY_CONTROL	SDI function failed to complete. This is the non-specific default for an error condition that does not meet a more specific definition.

Table 7 — *ReturnCode* field (part 2 of 3)

SDI return code (SDI_...)	SDI functions that return (CC_SDI_...)	Description
BAD_CNTL_CODE	Any reserved code	The SDI function code is invalid or unknown.
INVALID_PARAMETER	GET_DRIVER_INFO GET_CNTLR_CONFIG GET_CNTLR_STATUS FIRMWARE_DOWNLOAD GET_RAID_INFO GET_RAID_CONFIG GET_PHY_INFO SET_PHY_INFO GET_LINK_ERRORS SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH GET_SATA_SIGNATURE GET_SCSI_ADDRESS GET_DEVICE_ADDRESS TASK_MANAGEMENT PHY_CONTROL	The SDI data structure contained an invalid parameter on input. No additional information is provided.
SECURITY_VIOLATION	SET_PHY_INFO SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH TASK_MANAGEMENT SET_PHY_INFO PHY_CONTROL	The SDI data structure contained a directive to write information to the physical device and the SDI security level does not allow the operation.
RAID_SET_OUT_OF_RANGE	GET_RAID_CONFIG	*URaidSetIndex* is out of range.
PHY_INFO_CHANGED	SET_PHY_INFO	Phy information was successfully changed.
PHY_INFO_NOT_CHANGEABLE	SET_PHY_INFO	Phy information could not be changed. Indicates that the driver does not support changing the phy information.
LINK_RATE_OUT_OF_RANGE	SET_PHY_INFO PHY_CONTROL	The link rate was not supported by the hardware.
PHY_DOES_NOT_EXIST	SET_PHY_INFO GET_LINK_ERRORS SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH GET_SATA_SIGNATURE PHY_CONTROL	Specified phy does not exist.
PHY_DOES_NOT_MATCH_PORT	SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH	The phy and port combination does not exist
PHY_CANNOT_BE_SELECTED	SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH	Specified phy cannot be selected

Table 7 — *ReturnCode* field (part 3 of 3)

SDI return code (SDI_...)	SDI functions that return (CC_SDI_...)	Description
SELECT_PHY_OR_PORT	SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH	Return code indicating that either phy or port needs to be selected
PORT_DOES_NOT_EXIST	SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH	Specified port does not exist.
PORT_CANNOT_BE_SELECTED	SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH	Specified port cannot be selected.
CONNECTION_FAILED	SMP_PASSTHROUGH SSP_PASSTHROUGH STP_PASSTHROUGH TASK_MANAGEMENT	Connection failed.
NO_SATA_DEVICE	GET_SATA_SIGNATURE	Specified phy is not connected to a SATA device or has not completed a SATA OOB sequence.
NO_SATA_SIGNATURE	GET_SATA_SIGNATURE	Specified phy has not received the initial Register Device To Host FIS from the SATA device
SCSI_EMULATION	STP_PASSTHROUGH	Use the SCSI emulation CDB for passing SATA commands
NOT_AN_END_DEVICE	GET_SCSI_ADDRESS TASK_MANAGEMENT	The OS specific platform address cannot be returned because the device is not an end device
NO_SCSI_ADDRESS	GET_SCSI_ADDRESS TASK_MANAGEMENT	No OS specific platform address was found for this SAS address
NO_DEVICE_ADDRESS	GET_DEVICE_ADDRESS TASK_MANAGEMENT	No SAS address was found for this OS specific platform address

6 SDI functions

6.1 SDI functions overview

The SDI function code is provided, based on the platform (see 4.2 for Windows, 4.3 for Linux, and 4.4 for NetWare), as either an element of a buffer structure that is submitted as part of the device I/O control or as an argument to the device I/O control call. In either case, an SDI data structure buffer is provided as the content of the device I/O control call. The SDI data structure buffer has the general form of:

```
typedef struct _SDI_xxxx_xxxx_BUFFER {
      IOCTL_HEADER IoctlHeader;
} SDI_xxxx_xxxx_BUFFER, *PSDI_xxxx_xxxx_BUFFER;
```

The SDI data structure buffer provides as input the necessary information to specify the SDI function desired. It also provides space for any resulting data requested by the SDI function. The application using the SDI function codes shall ensure that enough memory has been allocated to contain any requested data. If the memory provided is too small a SDI error is returned from the device I/O control call.

Editor's Note 18: remove bolding below

Table 8 lists the SDI functions. The constants specified by the “Timeout” column should be specified in the *Timeout* field of the IOCTL_HEADER structure on submission of the device I/O control call. The “Security” column indicates which security level applies to the function for Windows (see 4.2.6).

Table 8 — SDI functions

SDI function	Timeout	Minimum required security level	Required	Reference
CC_SDI_GET_DRIVER_INFO	SDI_ALL_TIMEOUT	Restricted	M	6.2
CC_SDI_GET_CNTLR_CONFIG	SDI_ALL_TIMEOUT	Restricted	O	6.3
CC_SDI_GET_CNTLR_STATUS	SDI_ALL_TIMEOUT	Restricted	M	6.4
CC_SDI_FIRMWARE_DOWNLOAD	SDI_ALL_TIMEOUT	Limited	O	6.5
CC_SDI_GET_RAID_INFO	SDI_RAID_TIMEOUT	Restricted	M if RAID [a]	6.6
CC_SDI_GET_RAID_CONFIG	SDI_RAID_TIMEOUT	Restricted	M if RAID [a]	6.7
CC_SDI_GET_PHY_INFO	SDI_TIMEOUT	Restricted	M	6.8
CC_SDI_SET_PHY_INFO	SDI_TIMEOUT	Full	M	6.9
CC_SDI_GET_LINK_ERRORS	SDI_TIMEOUT	Restricted	M	6.10
CC_SDI_SMP_PASSTHROUGH	SDI_TIMEOUT	See function definition	M if SMP [b]	6.11
CC_SDI_SSP_PASSTHROUGH	SDI_TIMEOUT	See function definition	M if SSP [c]	6.12
CC_SDI_STP_PASSTHROUGH	SDI_TIMEOUT	See function definition	M if STP or SATA [d]	6.13
CC_SDI_GET_SATA_SIGNATURE	SDI_TIMEOUT	Restricted	M if SATA [e]	6.14
CC_SDI_GET_SCSI_ADDRESS	SDI_TIMEOUT	Restricted	M	6.15
CC_SDI_GET_DEVICE_ADDRESS	SDI_TIMEOUT	Restricted	M	6.16
CC_SDI_TASK_MANAGEMENT	SDI_TIMEOUT	Full	M	6.17
CC_SDI_GET_CONNECTOR_INFO	SDI_TIMEOUT	Restricted	M	6.18
CC_SDI_PHY_CONTROL	SDI_TIMEOUT	Full	O	6.19

[a] Mandatory if the controller supports RAID; optional otherwise.
[b] Mandatory if the controller supports SMP; optional otherwise.
[c] Mandatory if the controller supports SSP; optional otherwise.
[d] Mandatory if the controller supports STP and/or directly attached SATA devices; optional otherwise.
[e] Mandatory if the controller supports directly attached SATA devices; optional otherwise.

Editor’s Note 19: defined the control code assignments here (just offsets; each OS would have a different base value). Change the values from thosein 04-284r0.

Editor’s Note 20: change or drop the CC_ prefix. It means Control Code right now. Just the rest of

the name (SDI_...)may conflict with structure names, though.

6.2 CC_SDI_GET_DRIVER_INFO

6.2.1 Behavior

The CC_SDI_GET_DRIVER_INFO SDI function requests descriptive and version information about the device driver. The information returned should be consistent with any file information provided on the platform OS for the driver.

6.2.2 Input

This function accepts a SDI_DRIVER_INFO_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4; and
b) *Information*: All fields set to 0.

6.2.3 Output

This function shall return a SDI_DRIVER_INFO_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Information.szName*: Name of the binary driver. May contain a string of up to 80 ASCII characters including a null termination. Should reference the base name of the driver, without a file extension;
c) *Information.szDescription*: Description of the driver. May contain a string of up to 80 ASCII characters including a null termination. Should reference the vendor, product family and model information;

Editor's Note 21: 04-284r0 had 81 byte string fields; switched to 80 bytes

Editor's Note 22: ASCII is appropriate for szName. Is ASCII appropriate for the Description field, or is Unicode UCS-2 more appropriate for modern OSes? If so, it will need to be 160 bytes long to allow the same number of characters.

d) *Information.usMajorRevision*: Major revision of the driver;
e) *Information.usMinorRevision*: Minor revision of the driver;
f) *Information.usBuildRevision*: Build revision of the driver;
g) *Information.usReleaseRevision*: Release revision of the driver;

Editor's Note 23: Revision number formats may need to be OS specific. Or, just provide an ASCII string field.

Editor's Note 24: Windows INF files include DriverVer=mm/dd/yyyy[,w.x.y.z], where mm/dd/yyyy specify the date of the "driver package" (driver files and .inf) - the most recent date of any file in the package. / can be replaced by -. The optional w.x.y.z has integers greater than 0 (although 0 seems to be supported too) but less than 65535. These values are displayed in the Device Manager as the Driver Date and Driver Version. A driver resource file includes an additional internal version number provided in the double-DWORD value FILEVERSION (not used by Windows) and the FileVersion character string (displayed by Windows); internal format x.xx.00.xxxx and external string x.xx.xxxx.

Editor's Note 25: For Netware drivers, x.yy.z (displayed as 1.00c or 1.00.03). Each are numbers.

Editor's Note 26: For Linux drivers, x.y.z-a (major.minor.subminor-pass). x and y are linux kernel numbers. z is also a number. a may be an ASCII string.

h) *Information.usSDIMajorRevision*: Revision of this standard that the driver supports. The driver should return the constant SDI_MAJOR_REVISION; and
i) *Information.usSDIMinorRevision*: Revision of tthis standard that the driver supports. The driver should return the constant SDI_MINOR_REVISION.

Editor's Note 27: Use the SPC-3 style version descriptor instead - one 16-bit field.

6.2.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_DRIVER_INFO {
   __u8  szName[81];
   __u8  szDescription[81];
   __u16 usMajorRevision;
   __u16 usMinorRevision;
   __u16 usBuildRevision;
   __u16 usReleaseRevision;
   __u16 usSDIMajorRevision;
   __u16 usSDIMinorRevision;
} SDI_DRIVER_INFO, *PSDI_DRIVER_INFO;

typedef struct _SDI_DRIVER_INFO_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_DRIVER_INFO Information;
} SDI_DRIVER_INFO_BUFFER, *PSDI_DRIVER_INFO_BUFFER;
```

Editor's Note 28: Excerpts from sdi.h included piecemeal like this will easily become out of date and incorrect. Consider structuring sdi.h as a shell that includes 19 .h files, one per function. Include each of those .h files in the appropriate section. A preprocessor could be provided tomerge them into a single .h file for real use.

6.3 CC_SDI_GET_CNTLR_CONFIG

6.3.1 Behavior

The CC_SDI_GET_CNTLR_CONFIG SDI function requests descriptive and version information about the hardware, firmware and boot BIOS associated with a storage controller.

6.3.2 Input

This function accepts a SDI_CNTLR_CONFIG_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4; and
b) *Configuration*: All fields set to zero.

6.3.3 Output

This function shall return a SDI_CNTLR_CONFIG_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Configuration.uBaseIoAddress*: Base I/O Address of the controller. If the controller has more than one base I/O address, this field shall indicate the lowest one used;
c) *Configuration.BaseMemoryAddress*: Base memory address of the controller. If the controller has more than one base memory address, this field shall indicate the lowest one used;
d) *Configuration.uBoardID*: 32-bit subsystem ID from the controller's PCI configuration space. Bits 0 – 15 contain the subsystem vendor ID and bits 16 – 31 contain the subsystem ID as defined by the PCI specification;
e) *Configuration.usSlotNumber*: The physical slot number of the controller in the system. If the driver cannot determine the physical slot number, it shall return the SLOT_NUMBER_UNKNOWN (i.e., 0xFFFF);
f) *Configuration.bControllerClass*: Indicates the class of the controller (e.g., HBA or RAID). This shall be set to SDI_CNTLR_CLASS_HBA (i.e., 0x05);
g) *Configuration.bIoBusType*: System I/O bus type of the controller. Shall be set to one of the following:
   A) SDI_BUS_TYPE_PCI (i.e., 0x03): if the host bus adapter is in a PCI slot; or
   B) SDI_BUS_TYPE_PCMCIA (i.e., 0x04): if the host bus adapter is in a PCMCIA slot;

Editor's Note 29: where did the bus types come from?

---

h) *Configuration.BusAddress*: The I/O bus address (i.e., bus number, device number, and function number) of the controller, if applicable;

Editor's Note 30: replace BaseMemoryAddr, uBoardID, usSlotNumber, and bIoBusType with whatever OS-specific information is used by that OS to identify cards and their drivers.

---

i) *Configuration.szSerialNumber*: Controller serial number. Contains a string of up to 80 ASCII characters including a null termination character. Should reference the serial number of the controller. If the value is unknown, then the field shall be set to 0;
j) *Configuration.usMajorRevision*: Major revision of the controller firmware. If the value is unknown, then the field shall be set to 0;
k) *Configuration.usMinorRevision*: Minor revision of the controller firmware. If the value is unknown, then the field shall be set to 0;
l) *Configuration.usBuildRevision*: Build revision of the controller firmware. If the value is unknown, then the field shall be set to 0;
m) *Configuration.usReleaseRevision*: Release revision of the controller firmware. If the value is unknown, then the field shall be set to 0;

Editor's Note 31: version numbers with this 4 number structure may not suit all vendors. An ASCII string might be more palatable.

---

n) *Configuration.usBIOSMajorRevision*: Major revision of the controller boot BIOS. If the value is unknown, then the field shall be set to 0;
o) *Configuration.usBIOSMinorRevision*: Minor revision of the controller boot BIOS. If the value is unknown, then the field shall be set to 0;
p) *Configuration.usBIOSBuildRevision*: Build revision of the controller boot BIOS. If the value is unknown, then the field shall be set to 0;
q) *Configuration.usBIOSReleaseRevision*: Release revision of the controller boot BIOS. If the value is unknown, then the field shall be set to 0;

Editor's Note 32: BIOS is too x86 specific. EFI doesn't call itself a BIOS. PCI uses "expansion ROM". Also, maybe an ASCII string would better fit a variety of HBA vendors.

r) *uControllerFlags*: Controller subclass definition. One or more of the following constants may be used:
   A) SDI_CNTLR_SAS_HBA: controller is a SAS HBA;
   B) SDI_CNTLR_SAS_RAID: controller is a SAS HBA with RAID support;
   C) SDI_CNTLR_SATA_HBA: controller is a SATA HBA;
   D) SDI_CNTLR_SATA_RAID: controller is a SATA HBA with RAID support;

Editor's Note 33: Change ControllerFlags to a bitmask with just 3 bits used so far for SAS, SATA, and RAID. OR them together for SAS_RAID and SATA_RAID combinations.

   E) SDI_CNTLR_FWD_SUPPORT: controller supports firmware download SDI function code CC_SDI_FIRMWARE_DOWNLOAD;
   F) SDI_CNTLR_FWD_ONLINE: controller supports online update of firmware;
   G) SDI_CNTLR_FWD_SRESET: controller requires soft reset to initiate a firmware update. The driver manages coordinating the download to ensure outstanding IOs are not impacted;
   H) SDI_CNTLR_FWD_HRESET: controller requires a hard reset to initiate a firmware update. The driver forces the controller to a power-up state and re-initializes the controller as necessary;
   I) SDI_CNTLR_FWD_RROM: controller supports a redundant copy of the ROM image;

s) *Configuration.usRromMajorRevision*: Major revision of the redundant controller firmware. If the value is unknown, then the field shall be set to 0;
t) *Configuration.usRromMinorRevision*: Minor revision of the redundant controller firmware. If the value is unknown, then the field shall be set to 0;
u) *Configuration.usRromBuildRevision*: Build revision of the redundant controller firmware. If the value is unknown, then the field shall be set to 0;
v) *Configuration.usRromReleaseRevision*: Release revision of the redundant controller firmware. If the value is unknown, then the field shall be set to 0;
w) *Configuration.usRromBIOSMajorRevision*: Major revision of the redundant controller boot BIOS. If the value is unknown, then the field shall be set to 0;
x) *Configuration.usRromBIOSMinorRevision*: Minor revision of the redundant controller boot BIOS. If the value is unknown, then the field shall be set to 0;
y) *Configuration.usRromBIOSBuildRevision*: Build revision of the redundant controller boot BIOS. IIf the value is unknown, then the field shall be set to 0;
z) *Configuration.usRromBIOSReleaseRevision*: Release revision of the redundant controller boot BIOS. If the value is unknown, then the field shall be set to 0; and

Editor's Note 34: see comments above about version numbers

aa) *Configuration.bReserved[7]*: This field shall be set to 0.

6.3.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_PCI_BUS_ADDRESS {
   __u8  bBusNumber;
   __u8  bDeviceNumber;
   __u8  bFunctionNumber;
   __u8  bReserved;
} SDI_PCI_BUS_ADDRESS, *PSDI_PCI_BUS_ADDRESS;
```

```
typedef union _SDI_IO_BUS_ADDRESS {
   SDI_PCI_BUS_ADDRESS PciAddress;
   __u8  bReserved[32];
} SDI_IO_BUS_ADDRESS, *PSDI_IO_BUS_ADDRESS;

typedef struct _SDI_CNTLR_CONFIG {
   __u32 uBaseIoAddress;
   struct {
      __u32 uLowPart;
      __u32 uHighPart;
   } BaseMemoryAddress;
   __u32 uBoardID;
   __u16 usSlotNumber;
   __u8  bControllerClass;
   __u8  bIoBusType;
   SDI_IO_BUS_ADDRESS BusAddress;
   __u8  szSerialNumber[81];
   __u16 usMajorRevision;
   __u16 usMinorRevision;
   __u16 usBuildRevision;
   __u16 usReleaseRevision;
   __u16 usBIOSMajorRevision;
   __u16 usBIOSMinorRevision;
   __u16 usBIOSBuildRevision;
   __u16 usBIOSReleaseRevision;
   __u32 uControllerFlags;
   __u16 usRromMajorRevision;
   __u16 usRromMinorRevision;
   __u16 usRromBuildRevision;
   __u16 usRromReleaseRevision;
   __u16 usRromBIOSMajorRevision;
   __u16 usRromBIOSMinorRevision;
   __u16 usRromBIOSBuildRevision;
   __u16 usRromBIOSReleaseRevision;
   __u8  bReserved[7];
} SDI_CNTLR_CONFIG, *PSDI_CNTLR_CONFIG;

typedef struct _SDI_CNTLR_CONFIG_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_CNTLR_CONFIG Configuration;
} SDI_CNTLR_CONFIG_BUFFER, *PSDI_CNTLR_CONFIG_BUFFER;
```

6.4 CC_SDI_GET_CNTLR_STATUS

6.4.1 Behavior

The CC_SDI_GET_CNTLR_STATUS SDI function requests the current status of the controller.

6.4.2 Input

This function accepts a SDI_CNTLR_STATUS_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4; and
b) *Status*: All fields set to to 0.

6.4.3 Output

This function shall return a SDI_CNTLR_STATUS_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Status.uStatus*: Current status of the controller. Should contain one of the following values:
   A) SDI_CNTLR_STATUS_GOOD: operating normally;
   B) SDI_CNTLR_STATUS_FAILED: the controller has failed. No I/O is allowed to the controller in this state;
   C) SDI_CNTLR_STATUS_OFFLINE: the controller is in a transitional state and is currently inaccessible. It has not failed, but no I/O is allowed to the controller in this state; or
   D) SDI_CNTLR_STATUS_POWEROFF: the controller slot is powered off. It may have been failed before but currently does not have power to the slot;
c) *Status.uOfflineReason*: If the *Status.uStatus* field is set to SDI_CNTLR_STATUS_OFFLINE, the reason it is so. Contains one of the following values:
   A) SDI_OFFLINE_REASON_NO_REASON: unknown reason;
   B) SDI_OFFLINE_REASON_INITIALIZING: the driver is in the process of initializing the controller and bringing it online;
   C) SDI_OFFLINE_REASON_BACKSIDE_BUS_DEGRADED: the physical interface to the SAS or SATA domain is in a degraded state; or
   D) SDI_OFFLINE_REASON_BACKSIDE_BUS_FAILURE: the physical interface to the SAS or SATA domain has failed;

   and

d) *Status.bReserved*: This field shall be set to 0.

6.4.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_CNTLR_STATUS {
   __u32 uStatus;
   __u32 uOfflineReason;
   __u8  bReserved[28];
} SDI_CNTLR_STATUS, *PSDI_CNTLR_STATUS;

typedef struct _SDI_CNTLR_STATUS_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_CNTLR_STATUS Status;
} SDI_CNTLR_STATUS_BUFFER, *PSDI_CNTLR_STATUS_BUFFER;
```

6.5 CC_SDI_FIRMWARE_DOWNLOAD

6.5.1 Behavior

The CC_SDI_FIRMWARE_DOWNLOAD SDI function allows the controller firmware to be updated online. This is an optional function. The driver indicates support for this function in the CC_SDI_CNTLR_CONFIG control. For the function to be successful, the *uControllerFlags* field in the SDI_CNTLR_CONFIG structure must include both SDI_CNTLR_FWD_SUPPORT and SDI_CNTLR_FWD_ONLINE. Either SDI_CNTLR_FWD_SRESET or SDI_CNTLR_FWD_HRESET must be set depending on the upgrade reset behavior. The driver and controller are responsible for validating the integrity of the ROM image before attempting to upgrade.

Editor's Note 35: should this be done through an initiator LUN implementing the WRITE BUFFER command (or the controller LUN for RAID controllers)? Problems: That could require a new peripheral device type (or revival of the processor peripheral device type). Would have to provide a Class driver for that device type. There is no direct way to confirm which LUN is the initiator LUN (remote devices could also use the processor device type). Security archicture would have to

change.

6.5.2 Input

This function accepts a SDI_FIRMWARE_DOWNLOAD_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *Information.uBufferLength*: Specifies the length of the ROM image being downloaded in uDataBuffer;
c) *Information.uDownloadFlags*: Control for the firmware download operation. Contains one or more of the following values:
   A) SDI_FWD_VALIDATE: validate the download image, but do not upgrade the image. If this operation is not supported, then return the *Information.uStatus* field set to SDI_FWD_REJECT;
   B) SDI_FWD_SOFT_RESET: download operation initiates a soft reset to the controller after the ROM image has been upgraded. The driver manages all I/O until the controller has returned to a ready state. If a soft reset is insufficient to complete a firmware download operation then the *Information.uStatus* field shall be set to SDI_FWD_REJECT and the upgrade operation shall not be initiated; and/or
   C) SDI_FWD_HARD_RESET: download operation initiates a hard reset to the controller after the ROM image has been upgraded. The driver suspends all I/O until the controller has returned to a ready state. If a hard reset is insufficient to complete a firmware download operation then the *Information.uStatus* field shall be set SDI_FWD_REJECT and the upgrade operation shall not be initiated;
d) *Information.bReserved[32]*: This field shall be set to 0;
e) *Information.usStatus*: This field shall be set to 0;
f) *Information.usSeverity*: This field shall be set to 0; and
g) *bDataBuffer*: Represents the first byte of the ROM image that is being written to the controller.

Editor's Note 36: need to describe bDataBuffer[1] better

6.5.3 Output

This function shall return a SDI_FIRMWARE_DOWNLOAD_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Information.uBufferLength*: Same as input;
c) *Information.uDownloadFlags*: Same as input;
d) *Information.bReserved[32]*: Same as input;
e) *Information.uStatus*: Status of the firmware download operation. Contains one of the following values:
   A) SDI_FWD_SUCCESS: download operation was successful;
   B) SDI_FWD_FAILED: download operation has failed. No I/O is allowed to the controller;
   C) SDI_FWD_USING_RROM: download operation has failed and the controller is using the redundant ROM image;
   D) SDI_FWD_REJECT: download operation was rejected. The ROM image was corrupted or incorrect for this controller;
   E) SDI_FWD_DOWNREV: download operation was successful. However, the ROM image was an earlier revision than the executing image;
f) *Information.uSeverity*: The severity code for the uStatus. Should contain one of the following values:
   A) SDI_FWD_INFORMATION: uStatus is informational only;
   B) SDI_FWD_WARNING: uStatus is indicating a condition that may be helpful for diagnostic purposes;
   C) SDI_FWD_ERROR: uStatus is indicating a recoverable error condition; or
   D) SDI_FWD_FATAL: uStatus is indicating a fatal error condition.

   and

g) *bDataBuffer*: Represents the first byte of t the ROM image that was to be written to the controller.

6.5.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_FIRMWARE_DOWNLOAD {
   __u32 uBufferLength;
   __u32 uDownloadFlags;
   __u8   bReserved[32];
   __u16 usStatus;
   __u16 usSeverity;
} SDI_FIRMWARE_DOWNLOAD, *PSDI_FIRMWARE_DOWNLOAD;

typedef struct _SDI_FIRMWARE_DOWNLOAD_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_FIRMWARE_DOWNLOAD Information;
   __u8 bDataBuffer[1];
} SDI_FIRMWARE_DOWNLOAD_BUFFER, *PSDI_FIRMWARE_DOWNLOAD_BUFFER;
```

6.6 CC_SDI_GET_RAID_INFO

6.6.1 Behavior

The CC_SDI_GET_RAID_INFO SDI function requests information on the number of RAID volumes and number of physical drives on a controller. The RAID solution may be implemented within the driver (i.e., software RAID) or by firmware on the controller. If the uControllerFlags in the SDI_CNTLR_CONFIG structure indicates that the controller supports RAID, then the driver that implements this specification shall support this SDI function; otherwise the driver may respond to this function code with a generic IO error (see Submitting SDI Control Codes).

6.6.2 Input

This function accepts a SDI_RAID_INFO_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4; and
b) *Information*: All fields shall be set to 0.

6.6.3 Output

This function shall return a SDI_RAID_INFO_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Information.uNumRaidSets*: Number of logical RAID volumes (or sets) currently defined. If no volumes (or sets) have been defined, then a 0 value is returned;
c) *Information.uMaxDrivesPerSet*: Maximum number of physical drives within a logical RAID volume. This may be an absolute maximum or the actual maximum currently defined for all volumes. This value will be used to allocate memory for the CC_SDI_GET_RAID_CONFIG SDI function; and
d) *Information.bReserved[92]*: This field shall be set to 0.

6.6.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_RAID_INFO {
   __u32 uNumRaidSets;
   __u32 uMaxDrivesPerSet;
   __u8  bReserved[92];
} SDI_RAID_INFO, *PSDI_RAID_INFO;

typedef struct _SDI_RAID_INFO_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_RAID_INFO Information;
```

```
} SDI_RAID_INFO_BUFFER, *PSDI_RAID_INFO_BUFFER;
```

6.7 CC_SDI_GET_RAID_CONFIG

6.7.1 Behavior

The CC_SDI_GET_RAID_CONFIG SDI function requests information for a specified RAID set on a controller that supports RAID. To obtain the information for all the logical RAID sets defined; this SDI function shall be called for each RAID set of the controller. If the *uControllerFlags* in the SDI_CNTLR_CONFIG structure indicates that the controller supports RAID, then the driver that implements this specification shall support this SDI function; otherwise the driver may respond to this function code with a generic IO error (see Submitting SDI Control Codes).

6.7.2 Input

This function accepts a SDI_RAID_CONFIG_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *Configuration.uRaidSetIndex*: Contains the number of the RAID set for which information is being requested. A calling routine would increment this value to enumerate the information for all RAID sets. If this value exceeds the number of RAID sets (see CC_SDI_GET_RAID_INFO), then the SDI function shall return the *IoctlHeader.ReturnCode* field set to SDI_RAID_SET_OUT_OF_RANGE;
c) *Configuration.uCapacity*: This field shall be set to 0;
d) *Configuration.uStripeSize*: This field shall be set to 0;
e) *Configuration.bRaidType*: This field shall be set to 0;
f) *Configuration.bStatus*: This field shall be set to 0;
g) *Configuration.bInformation*: This field shall be set to 0;
h) *Configuration.bDriveCount*: This field shall be set to 0; and
i) *Configuration.bReserved[20]*: This field shall be set to 0;

6.7.3 Output

This function shall return a SDI_RAID_CONFIG_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Configuration.uRaidSetIndex*: Same as input;
c) *Configuration.uCapacity*: Contains the capacity of the RAID set in mebibytes (i.e., MiB);
d) *Configuration.uStripeSize*: Contains the stripe size of the RAID set in kibibytes (i.e., KiB);
e) *Configuration.bRaidType*: Contains the basic RAID type of the RAID set. Contains one of:
   A) SDI_RAID_TYPE_NONE: indicates the RAID set is composed of a single drive;
   B) SDI_RAID_TYPE_0: indicates the RAID set is a striped set, with no fault tolerance;
   C) SDI_RAID_TYPE_1: indicates the RAID set is a mirrored set;
   D) SDI_RAID_TYPE_10: indicates the RAID set is a striped mirror set;
   E) SDI_RAID_TYPE_5: indicates the RAID set is a parity set supporting single drive failure;
   F) SDI_RAID_TYPE_15: indicates the RAID set is an advanced parity set;
   G) SDI_RAID_TYPE_6: indicates the RAID set is an advanced parity set supporting dual drive failures; or
   H) SDI_RAID_TYPE_OTHER: indicates the RAID set type configuration does not match the standard types;

Editor's Note 37: Maybe just return an ASCII (or Unicode UCS-2) string as the RAID type.

---

f) *Configuration.bStatus*: Contains the status of the RAID set. Should be one of:
   A) SDI_RAID_SET_STATUS_OK: indicates the RAID set is operational;
   B) SDI_RAID_SET_STATUS_DEGRADED: indicates the RAID set is no longer functioning in a fault tolerant mode;

C) SDI_RAID_SET_STATUS_REBUILDING: indicates the RAID set is rebuilding. This implies a degraded operation. Once the rebuild completes successfully, the status will change to SDI_RAID_SET_STATUS_OK. If the rebuilding process fails, the status will be updated appropriately;
D) SDI_RAID_SET_STATUS_FAILED: indicates the RAID set has failed. There is no guarantee on the operational behavior of the RAID set and data loss has occurred or is imminent;

g) *Configuraton.bInformation*: Contains clarifying information for *Configuration.bStatus* results. The actual content depends on the *Configuration.bStatus* result. Should be:
A) If *Configuration.bStatus* is set to SDI_RAID_SET_STATUS_OK, then *Configuration.bInformation* shall be 0;
B) If *Configuration.bStatus* is set to SDI_RAID_SET_STATUS_DEGRADED, then *Configuration.bInformation* shall contain the failed drive index number;
C) If *Configuration.bStatus* is set to SDI_RAID_SET_STATUS_REBUILDING, then *Configuration.bInformation* shall contain the percentage complete. The value shall be in the range of 0 to 100 (0h to 64h);
D) If *Configuration.bStatus* is set to SDI_RAID_SET_STATUS_FAILED, then *Configuration.bInformation* shall be 0 or vendor specific. Since the failure modes could include drive or controller failures, *Configuration.bInformation* may provide a vendor specific error code to indicate which component led to the failed status;

h) *Configuration.bDriveCount*: Contains the number of drives in the RAID set and in turn the number of SDI_RAID_DRIVES data structures that will exist;

i) *Configuration.Drives[n]:* Contains one SDI_RAID_DRIVES data structure for each physical drive which is used in the RAID set, each containing the following fields:
A) *Configuration.Drives[n].bModel*: Contains 40 ASCII characters indicating the drive model number:
a) For SAS drives, this is the concatenation of the 8-byte VENDOR IDENTIFICATION field and the 16-byte PRODUCT IDENTIFICATION field from the standard INQUIRY data (see SPC-3) with 16 ASCII space characters; or
b) For SATA drives, this is the 40-byte MODEL NUMBER field from the IDENTIFY DEVICE data (see ATA/ATAPI-7 V1) with each pair of bytes swapped to create a valid ASCII string format;
B) *Configuration.Drives[n].bFirmware*: Contains 8 ASCII characters indicating the drive firmware revision level.
a) For SAS drives, this is the concatenation of the PRODUCT REVISION LEVEL field from the standard INQUIRY data (see SPC-3) with 4 ASCII space characters;
b) For SATA drives, this is from the FIRMWARE REVISION field in the IDENTIFY DEVICE data (see ATA/ATAPI-7 V1) with each pair of bytes swapped to create a valid ASCII string format;
C) *Configuration.Drives[n].bSerialNumber*: Contains 40 ASCII characters indicating the drive serial number:
a) For SAS drives, this is the first 40 bytes of the PRODUCT SERIAL NUMBER field from the Unit Serial Number VPD page (see SPC-3), if any, concatenated with ASCII space characters;
b) For SATA drives, this is from the SERIAL NUMBER field in the IDENTIFY DEVICE data (see ATA/ATAPI-7 V1) with each pair of bytes swapped to create a valid ASCII string format;

Editor's Note 38: all the above strings changed to filled with ASCII spaces not nulls

---

D) *Configuration.Drives[n].bSASAddress*: Contains the SAS address of the physical drive. If the drive does not have a SASAddress (e.g., a directly attached SATA drive), then this field shall be set to 0;
E) *Configuration.Drives[n].bSASLun*: Contains the SCSI logical unit number of the physical drive. If the drive does not have a SCSI logical unit number (e.g., a directly attached SATA drive), then this field shall be set to 0;

Editor's Note 39: SASLun is poorly named. physical drive vs. logical unit terminology needs work.

F) *Configuration.Drives[n].bDriveStatus*: Indicates the status of the physical drive. Contains one of:
   a) SDI_DRIVE_STATUS_OK: indicates the physical drive is operational;
   b) SDI_DRIVE_STATUS_DEGRADED: indicates the physical drive has posted a SMART notification (e.g., an ATA SMART event or a SCSI informational exception condition) to the controller;
   c) SDI_DRIVE_STATUS_REBUILDING: indicates the physical drive is the target drive of a RAID set rebuild. Once the rebuild completes successfully, the status will change to SDI_DRIVE_STATUS_OK. If the rebuilding process fails, the status will be updated appropriately; or
   d) SDI_DRIVE_STATUS_FAILED: indicates the physical drive has posted unrecoverable errors to the controller or has triggered a vendor specific action to remove the physical drive from the RAID set. There is no guarantee on the operational behavior of the drive and data loss has occurred or is imminent;

G) *Configuration.Drives[n].bDriveUsage*: Indicates whether the physical drive is part of the RAID set. Contains one of:
   a) SDI_DRIVE_CONFIG_NOT_USED: indicates the physical drive is not part of a RAID set;
   b) SDI_DRIVE_CONFIG_MEMBER: indicates the physical drive is part of this RAID set; or
   c) SDI_DRIVE_CONFIG_SPARE: indicates the physical drive is part of this RAID set as a hot swap spare.

and

j) *Configuration.Drives[n].bReserved*: This field shall be set to 0.

6.7.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_RAID_DRIVES {
   __u8  bModel[40];
   __u8  bFirmware[8];
   __u8  bSerialNumber[40];
   __u8  bSASAddress[8];
   __u8  bSASLun[8];
   __u8  bDriveStatus;
   __u8  bDriveUsage;
   __u8  bReserved[22];
} SDI_RAID_DRIVES, *PSDI_RAID_DRIVES;

typedef struct _SDI_RAID_CONFIG {
   __u32 uRaidSetIndex;
   __u32 uCapacity;
   __u32 uStripeSize;
   __u8  bRaidType;
   __u8  bStatus;
   __u8  bInformation;
   __u8  bDriveCount;
   __u8  bReserved[20];
   SDI_RAID_DRIVES Drives[1];
} SDI_RAID_CONFIG, *PSDI_RAID_CONFIG;

typedef struct _SDI_RAID_CONFIG_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_RAID_CONFIG Configuration;
} SDI_RAID_CONFIG_BUFFER, *PSDI_RAID_CONFIG_BUFFER;
```

6.8 CC_SDI_GET_PHY_INFO

6.8.1 Behavior

The CC_SDI_GET_PHY_INFO SDI function requests information about the physical characteristics and interconnect to the SATA or SAS domain.

6.8.2 Input

This function accepts a SDI_PHY_INFO_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4; and
b) *Information*: All fields shall be set to to 0.

6.8.3 Output

This function shall return a SDI_PHY_INFO_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Information.bNumberOfPhys*: Contains the number of phys (real or virtual) supported by this controller. It is possible for a controller and/or driver to contain a virtual phy that supports one or more of the SAS protocols. A management or test application should not assume that all phys are real;
c) *Information.bReserved*: This field shall be set to 0;
d) *Information.Phy[0 - 31]*: Contains 32 SDI_PHY_ENTITY data structures each of which defines the physical characteristics and provides information on the device attached to each interconnect;
e) *Information.Phy[n].Identify*: Contains a data structure with information that will be transferred to the attached device during a link reset sequence as defined in the SAS specification. If the controller is a SATA implementation, then the link reset sequence is not transmitted, but the content of this data structure will define this controller as a SATA solution;
f) *Information.Phy[n].Identify.bDeviceType*: Contains the SAS device type. Should be one of the following:
   A) SDI_PHY_UNUSED: indicates that the phy cannot be attached to a physical device;
   B) SDI_END_DEVICE: indicates that the phy will have the characteristics of a SAS end device. A SATA controller would define a SATA device as an end device;

Editor's Note 40: reword above.

---

g) *Information.Phy[n].Identify.bInitiatorPortProtocol*: Contains information on which SAS initiator protocols are supported by this initiator on this phy. Should be one or more of the following:
   A) SDI_PROTOCOL_SATA: indicates the controller may support a directly attached SATA device. This protocol bit is used to notify the management or test application about the SATA capabilities of a controller, it will be masked out from the data provided during a SAS link reset sequence. A SAS controller may support this protocol. A SATA controller shall support this protocol;
   B) SDI_PROTOCOL_SMP: indicates the controller may support connection to a SAS expander device. A SAS controller shall support this protocol. A SATA controller may support this protocol;
   C) SDI_PROTOCOL_STP: indicates the controller may support connection to a tunneled SATA device. A SAS or SATA controller may support this protocol;
   D) SDI_PROTOCOL_SSP: indicates the controller may support connection to a SAS end device. A SAS controller shall support this protocol. If a SATA controller supports this protocol, then it is by definition a SAS controller;

Editor's Note 41: This really returns the full byte containing the Initiator Port bits

---

h) *Information.Phy[n].Identify.bTargetPortProtocol*: Contains information on which SAS target protocols are supported by this initiator on this phy. Initiators are not required to support any target protocols, so

this field would typically be 0. However an initiator may have target capabilities and in that event, this field should be one or more of the following:

A) SDI_PROTOCOL_SATA: indicates the controller may respond as a SATA device. This protocol bit is used to notify the management or test application about the SATA capabilities of a controller, it will be masked out from the data provided during a SAS link reset sequence. A SAS or SATA controller may support this protocol;
B) SDI_PROTOCOL_SMP: indicates the controller may respond as a SAS expander device. A SAS or SATA controller may support this protocol;
C) SDI_PROTOCOL_STP: indicates the controller may respond as a tunneled SATA device. A SAS or SATA controller may support this protocol;
D) SDI_PROTOCOL_SSP: indicates the controller may respond as a SAS end device. A SAS controller may support this protocol;

i) *Information.Phy[n].Identify.bSASAddress*: Contains the SAS address in MSB order. A SATA controller, or a SAS controller with a SATA drive directly attached, shall return a 0 for this field;
j) *Information.Phy[n].Identify.bPhyIdentifier*: Contains the phy identifier of this phy. The range of the value must be from 0 to (bNumberOfPhys – 1). This value is restricted to a maximum of 254 (FEh), because FFh is a reserved identifier used to indicate a “don’t care” for other SDI functions;
k) *Information.Phy[n].bPortIdentifier*. Contains the port identifier associated with this phy. The range of the value shall be from 0 to (bNumberOfPhys - 1). Multiple phys may be associated with the same port, because of wide links in SAS. For example, a 4 wide link (phys 0 - 3) from an initiator to an expander all reference one port (port 0);
l) *Information.Phy[n].bNegotiatedLinkRate*: Contains the current link rate of this phy. Should be one of the following:

A) SDI_LINK_RATE_UNKNOWN: indicates the link may currently be unconnected, or that a link rate does not apply, as is the case with a virtual phy;
B) SDI_PHY_DISABLED: indicates the phy has been disabled;
C) SDI_LINK_RATE_FAILED: indicates that a link rate negotiation has failed. In this case, there appears to be a device connected, because the link reset sequence has been initiated, but communication was not established;
D) SDI_SATA_SPINUP_HOLD: indicates that a link has detected a SATA device attached and is in a wait state to release a spin-up hold. A SATA drive will use this mechanism to stage the power surges associated with spin-up;
E) SDI_SATA_PORT_SELECTOR: indicates that a link has detected a SATA port selector device is attached;
F) SDI_LINK_RATE_1_5_GBPS: indicates that a link was established at 1.5 Gbps;
G) SDI_LINK_RATE_3_0_GBPS: indicates that a link was established at 3.0 Gbps;
H) SDI_LINK_VIRTUAL: indicates that a link is available to a virtual device;

m) *Information.Phy[n].bMinimumLinkRate*: Contains the minimum link rate for this phy. This field incorporates information for both the programmed and hardware link rate. Should be one of the following:

A) SDI_LINK_RATE_1_5_GBPS: indicates the minimum link rate for this phy is 1.5 Gbps; or
B) SDI_LINK_RATE_3_0_GBPS: indicates the minimum link rate for this phy is 3.0 Gbps;

In combination with one of the following:

A) SDI_PROGRAMMED_LINK_RATE_1_5_GBPS: indicates the minimum link rate programmed for this phy is 1.5 Gbps; or
B) SDI_PROGRAMMED_LINK_RATE_3_0_GBPS: indicates the minimum link rate programmed for this phy is 3.0 Gbps;

n) *Information.Phy[n].bMaximumLinkRate*: Contains the maximum link rate for this phy. This field incorporates information for both the programmed and hardware link rate. Should be one of the following:

A) SDI_LINK_RATE_1_5_GBPS: indicates the maximum link rate for this phy is 1.5 Gb/s;
B) SDI_LINK_RATE_3_0_GBPS: indicates the maximum link rate for this phy is 3.0 Gb/s;

In combination with one of the following:

A) SDI_PROGRAMMED_LINK_RATE_1_5_GBPS: indicates the maximum link rate programmed for this phy is 1.5 Gbps;

B) SDI_PROGRAMMED_LINK_RATE_3_0_GBPS: indicates the maximum link rate programmed for this phy is 3.0 Gbps;

o) *Information.Phy[n].bPhyChangeCount*: Contains the current count of BROADCAST(CHANGE) primitives received on this phy. This count needs to be updated according to the SAS specfication. A SATA controller should update this count anytime a hotplug event is detected. If the SATA controller does not support hotplug detection, then this value should remain 0;

p) *Information.Phy[n].bAutoDiscover*: Contains the current state of the discover process for the SAS Domain. The auto-discover process may begin at power on or may be initiated by an OS platform specific method. For example, an IOCTL_SCSI_RESCAN_BUS I/O control function on Windows will initiate auto-discover. Auto-discovery can only be interrupted in a vendor specific manner. Set to:
   A) SDI_DISCOVER_NOT_SUPPORTED: indicates that auto-discover is not supported. A SATA controller should set this value;
   B) SDI_DISCOVER_NOT_STARTED: indicates that auto-discover is supported, but has not begun;
   C) SDI_DISCOVER_IN_PROGRESS: indicates that the auto-discover process is in progress. Any address translation or routing errors that occur during this period should be retried;
   D) SDI_DISCOVER_COMPLETE: indicates that the auto-discover process has completed successfully; or
   E) SDI_DISCOVER_ERROR: indicates that the auto-discover process has completed with a vendor unique or topology error. The driver may have a vendor unique mechanism to determine where the error occurred. A management or test application may need to examine the topology to determine the cause of the error. Address translation, routing errors, or command errors may result if this state is entered;

q) *Information.Phy[n].Attached*: Contains a SDI_IDENTIFY data structure with information that defines the attached device. If the attached device is a SATA device, then the controller will generate a pseudo representation of the information;

r) *Information.Phy[n].Attached.bDeviceType*: Contains the SAS device type attached to this phy (i.e., byte 0 of the IDENTIFY address frame). This field should contain one of the following:
   A) SDI_NO_DEVICE_ATTACHED: indicates that the phy is not currently attached to a device. A SATA controller shall use this value if no device is attached;
   B) SDI_END_DEVICE: indicates that the phy is connected to a SAS target or a SATA device. A SATA controller shall use this value if a SATA device is attached;
   C) SDI_EDGE_EXPANDER_DEVICE: indicates that the phy is connected to a SAS edge expander; or
   D) SDI_FANOUT_EXPANDER_DEVICE: indicates that the phy is connected to a SAS fanout expander;

Editor's Note 42: need an AND mask to properly validate the field contents. Cannot use == or | or || since there are other fields in the byte.

---

s) *Information.Phy[n].Attached.bRestricted_Byte1*: Contains byte 1 of the IDENTIFY address frame. If the attached device is a SATA device, then this field shall be set to 0;

Editor's Note 43: rename all such fields with their byte numbers

---

t) *Information.Phy[n].Attached.bInitiatorPortProtocol*: Contains information on which SAS initiator protocols are supported by the attached device (i.e., byte 2 of the IDENTIFY address frame). This field may contain one or more of the following:
   A) SDI_PROTOCOL_SATA: indicates the attached device is a directly attached SATA host device. If this bit is set it indicates a topology error;
   B) SDI_PROTOCOL_SMP: indicates the attached device supports generating SMP requests;
   C) SDI_PROTOCOL_STP: indicates the attached device supports generating STP commands;
   D) SDI_PROTOCOL_SSP: indicates the attached device supports generating SSP commands;

u) *Information.Phy[n].Attached.bTargetPortProtocol*: Contains information on which SAS target protocols are supported by the attached device (i.e., byte 3 of the IDENTIFY address frame). This field should contain one or more of the following:
   A) SDI_PROTOCOL_SATA: indicates the attached device is a directly attached SATA device;
   B) SDI_PROTOCOL_SMP: indicates the attached device supports receiving SMP requests;
   C) SDI_PROTOCOL_STP: indicates the attached device supports receiving STP commands; and/or
   D) SDI_PROTOCOL_SSP: indicates the attached device supports receiving SSP commands;
v) *Information.Phy[n].Attached.bRestricted_Bytes4to11*: Contains bytes 4 through 11 of the IDENTIFY address frame. If the attached device is a SATA device, then this field shall be set to 0;

Editor's Note 44: better named Restricted4to11

w) *Information.Phy[n].Attached.bSASAddress*: Contains the SAS address of the attached device (i.e., bytes 12 through 19 of the IDENTIFY address frame). If the attached device is a SATA device, then this field shall be set to 0;
x) *Information.Phy[n].Attached.bPhyIdentifier*: Contains the phy identifier of the attached device (i.e., byte 20 of the IDENTIFY address frame). If the attached device is a SATA device, then this field shall be set to 0;
y) *Information.Phy[n].Attached.bSignalClass*: Contains byte 21 of the IDENTIFY address frame. If the attached device is a SATA device, then this field shall be set to 0; and
z) *Information.Phy[n].Attached.bReserved_Bytes22to27*: Contains bytes 22 through 27 of the IDENTIFY address frame. If the attached device is a SATA device, then this field shall be set to 0.

Editor's Note 45: better names Reserved22to27

6.8.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_IDENTIFY {
   __u8  bDeviceType;
   __u8  bRestricted_Byte1;
   __u8  bInitiatorPortProtocol;
   __u8  bTargetPortProtocol;
   __u8  bRestricted_Bytes4to11[8];
   __u8  bSASAddress[8];
   __u8  bPhyIdentifier;
   __u8  bSignalClass;
   __u8  bReserved_Bytes22to27[6];
} SDI_IDENTIFY, *PSDI_IDENTIFY;

typedef struct _SDI_PHY_ENTITY {
   SDI_IDENTIFY Identify;
   __u8  bPortIdentifier;
   __u8  bNegotiatedLinkRate;
   __u8  bMinimumLinkRate;
   __u8  bMaximumLinkRate;
   __u8  bPhyChangeCount;
   __u8  bAutoDiscover;
   __u8  bReserved[2];
   SDI_IDENTIFY Attached;
} SDI_PHY_ENTITY, *PSDI_PHY_ENTITY;

typedef struct _SDI_PHY_INFO {
```

```
    __u8  bNumberOfPhys;
    __u8  bReserved[3];
    SDI_PHY_ENTITY Phy[32];
} SDI_PHY_INFO, *PSDI_PHY_INFO;

typedef struct _SDI_PHY_INFO_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_PHY_INFO Information;
} SDI_PHY_INFO_BUFFER, *PSDI_PHY_INFO_BUFFER;
```

6.9 CC_SDI_SET_PHY_INFO

6.9.1 Behavior

The CC_SDI_SET_PHY_INFO SDI function requests that the physical characteristics of a phy be changed.

Even though this SDI function is required, the changing of physical characteristics is not required. If the driver does not supporting changing any characteristics, it shall return an *IoctlHeader.ReturnCode* field set to SDI_PHY_INFO_NOT_CHANGEABLE and not perform any further behavior.

Upon completion of a phy characteristic change, the driver shall return an *IoctlHeader.ReturnCode* field set to SDI_PHY_INFO_CHANGED and shall initiate a link reset sequence.

6.9.2 Input

This function accepts a SDI_SET_PHY_INFO_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *Information.bPhyIdentifier*: Contains the phy identifier of the phy to modify;
c) *Information.bNegotiatedLinkRate*: Contains the directive to negotiate a new link rate or to disable the phy. Should be one of the following:
   A) SDI_LINK_RATE_NEGOTIATE: specifies that controller shall negotiate a new link rate constrained by the *Information.bProgrammedMinimumLinkRate* and *Information.bProgrammedMaximumLinkRate* values provided;
   B) SDI_LINK_RATE_PHY_DISABLED: indicates that the phy should be disabled. The values for *Information.bProgrammedMinimumLinkRate* and *Information.bProgrammedMaximumLinkRate* will be updated after the phy has been disabled. A link reset sequence will not occur;
d) *Information.bProgrammedMinimumLinkRate*: Contains the value used to update the minimum programmed link rate for this phy. If the value is outside the range of the hardware minimum and maximum link rates or greater than *Information.bProgrammedMaximumLinkRate* then the SDI function shall return the *IoctlHeader.ReturnCode* field set to SDI_LINK_RATE_OUT_OF_RANGE. Should be one of the following:
   A) SDI_PROGRAMMED_LINK_RATE_UNCHANGED: specifies that the programmed minimum link rate shall not be changed;
   B) SDI_PROGRAMMED_LINK_RATE_1_5_GBPS: specifies that the programmed minimum link rate shall be updated to 1.5 Gbps;
   C) SDI_PROGRAMMED_LINK_RATE_3_0_GBPS: specifies that the programmed minimum link rate shall be updated to 3.0 Gbps;
e) *Information.bProgrammedMaximumLinkRate*: Contains the value used to update the maximum programmed link rate for this phy. If the value is outside the range of the hardware minimum and maximum link rates or less than bProgrammedMinimumLinkRate then the SDI function shall return the *IoctlHeader.ReturnCode* field set to SDI_LINK_RATE_OUT_OF_RANGE. Should be one of the following:
   A) SDI_PROGRAMMED_LINK_RATE_UNCHANGED: specifies that the programmed maximum link rate shall not be changed;
   B) SDI_PROGRAMMED_LINK_RATE_1_5_GBPS: specifies that the programmed maximum link rate shall be updated to 1.5 Gbps;
   C) SDI_PROGRAMMED_LINK_RATE_3_0_GBPS: specifies that the programmed maximum link rate shall be updated to 3.0 Gbps;

f) *Information.bSignalClass*: This field shall be set to 0; and
g) *Information.bReserved*: This field shall be set to 0.

6.9.3 Output

This function shall return a SDI_SET_PHY_INFO_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Information.bPhyIdentifier*: Same as input;
c) *Information.bNegotiatedLinkRate*: Same as input;
d) *Information.bProgrammedMinimumLinkRate*: Same as input;
e) *Information.bProgrammedMaximumLinkRate*: Same as input;
f) *Information.bSignalClass*: Same as input; and
g) *Information.bReserved*: Same as input.

6.9.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_SET_PHY_INFO {
   __u8  bPhyIdentifier;
   __u8  bNegotiatedLinkRate;
   __u8  bProgrammedMinimumLinkRate;
   __u8  bProgrammedMaximumLinkRate;
   __u8  bSignalClass;
   __u8  bReserved[3];
} SDI_SET_PHY_INFO, *PSDI_SET_PHY_INFO;

typedef struct _SDI_SET_PHY_INFO_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_SET_PHY_INFO Information;
} SDI_SET_PHY_INFO_BUFFER, *PSDI_SET_PHY_INFO_BUFFER;
```

6.10 CC_SDI_GET_LINK_ERRORS

6.10.1 Behavior

The CC_SDI_GET_LINK_ERRORS SDI function requests information on the link errors associated with a specific phy. If the controller cannot support tracking of one or more of the errors indicated, then the associated error counter should contain 0. If the controller can track one or more of the errors indicated, then the reset flag must be supported. If the controller does not support any of the link error counters, then it shall return an *IoctlHeader.ReturnCode* field set to SDI_STATUS_FAILED.

6.10.2 Input

This function accepts a SDI_LINK_ERRORS_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *Information.bPhyIdentifier*: Specifies the phy identifier of the phy for which link error information shall be returned. If the phy identifier specified is to an unsupported or non-existing phy, then the SDI function shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_EXIST;
c) *Information.bResetCounts*: Contains a flag to reset the error counts on return. Should be set to one of the following:
   A) SDI_LINK_ERROR_DONT_RESET_COUNTS: specifies that the error counts shall not be reset; or
   B) SDI_LINK_ERROR_RESET_COUNTS: specifies that the error counts shall be reset;
d) *Information.bReserved*: This field shall be set to 0;
e) *Information.uInvalidDwordCount*: This field shall be set to 0;
f) *Information.uRunningDisparityErrorCount*: This field shall be set to 0;
g) *Information.uLossOfDwordSyncCount*: This field shall be set to 0; and

h) *Information.uPhyResetProblemCount*: This field shall be set to 0.

6.10.3 Output

This function shall return a SDI_LINK_ERRORS_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Information.bPhyIdentifier*: Same as input;
c) *Information.bResetCounts*: Same as input;
d) *Information.uInvalidDwordCount*: Indicates the number of invalid dwords received by the phy (see the SMP REPORT PHY ERROR LOG function in SAS);
e) *Information.uRunningDisparityErrorCount*: Indicates the number of dwords with disparity errors received by the phy (see the SMP REPORT PHY ERROR LOG function in SAS);
f) *Information.uLossOfDwordSyncCount*: Indicates the number of loss of dword synchronizations detected by the phy (see the SMP REPORT PHY ERROR LOG function in SAS); and
g) *Information.uPhyResetProblemCount*: Indicates the number of phy reset problems detected by the phy (see the SMP REPORT PHY ERROR LOG function in SAS).

6.10.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_LINK_ERRORS {
   __u8  bPhyIdentifier;
   __u8  bResetCounts;
   __u8  bReserved[2];
   __u32 uInvalidDwordCount;
   __u32 uRunningDisparityErrorCount;
   __u32 uLossOfDwordSyncCount;
   __u32 uPhyResetProblemCount;
} SDI_LINK_ERRORS, *PSDI_LINK_ERRORS;

typedef struct _SDI_LINK_ERRORS_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_LINK_ERRORS Information;
} SDI_LINK_ERRORS_BUFFER, *PSDI_LINK_ERRORS_BUFFER;
```

6.11 CC_SDI_SMP_PASSTHROUGH

6.11.1 Behavior

The CC_SDI_SMP_PASSTHROUGH SDI function provides a method of sending generic SMP requests to a specific SAS address. Any driver that implements this specification and supports the SMP protocol shall support this SDI function; otherwise the driver must respond to this function code with a generic IO error (see Submitting Control Codes).

6.11.2 Security

A driver shall return the *IoctrlHeader.ReturnCode* field set to SDI_SECURITY_VIOLATION if the security level is insufficient to perform the requested function.

SMP function codes 00h to 7Fh (i.e., the read functions) shall always be allowed. SMP function codes 80h to FFh (i.e., the write functions) shall be allowed only if the security level is FULL.

6.11.3 Input

This function accepts a SDI_SMP_PASSTHROUGH_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *Parameters.bPhyIdentifier*: Specifies the phy identifier of the phy that should be used to issue the SMP request. The value shall be in the range of 0x00 to 0xFE representing phy identifiers, or shall be

set to SDI_USE_PORT_IDENTIFIER (i.e., 0xFF) specifying that the *Parameters.bPortIdentifier* field be used instead. The driver may generate an error due to the phy identifier for the following reasons:

A) If the driver does not support sending SMP requests to a phy identifier and only supports sending SMP requests to a bPortIdentifier, it shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_CANNOT_BE_SELECTED;
B) If the phy identifier is out of the range of valid phys, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_EXIST;
C) The phy identifier cannot be associated with bPortIdentifier. If bPhyIdentifier is intended to reference the phy and the bPortIdentifier value is not SDI_IGNORE_PORT, then the bPhyIdentifier and bPortIdentifier must have the proper association. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_MATCH_PORT;
D) The phy identifier has a value of SDI_USE_PORT_IDENTIFIER and the bPortIdentifier has a value of SDI_IGNORE_PORT. The driver cannot determine where to send the SMP request. Either bPhyIdentifier should reference a valid phy or bPortIdentifier must reference a valid port. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_SELECT_PHY_OR_PORT;

c) *Parameters.bPortIdentifier*: Specifies the port identifier of the port that should be used to issue the SMP request. The value shall be in the range of 0x00 to 0xFEh representing port identifiers, or shall be set to SDI_IGNORE_PORT (i.e., 0xFF). The driver may generate an error due to the port identifier for the following reasons:
   A) If the driver does not support sending SMP requests to a port identifier and only supports sending SMP requests to a bPhyIdentifier, it shall return the *IoctlHeader.ReturnCode* field set to SDI_PORT_CANNOT_BE_SELECTED;
   B) If the port identifier is out of range of valid ports, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PORT_DOES_NOT_EXIST;
   C) The port identifier cannot be associated with bPhyIdentifier. If bPortIdentifier is intended to reference the port and the bPhyIdentifier value is not SDI_USE_PORT_IDENTIFIER, then the bPortIdentifier and bPhyIdentifier must have the proper association. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_MATCH_PORT;
   D) The port identifier has a value of SDI_IGNORE_PORT and the bPhyIdentifier has a value of SDI_USE_PORT_IDENTIFIER. The driver cannot determine where to send the SMP request. Either bPhyIdentifier should reference a valid phy or bPortIdentifier must reference a valid port. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_SELECT_PHY_OR_PORT;
d) *Parameters.bConnectionRate*: Specifies the connection rate directive for the driver connection manager. The field shall be set to one of the following:
   A) SDI_LINK_RATE_NEGOTIATED: specifies that the connection should be opened at the highest allowable negotiated rate for the destination device. The resulting rate will be the lowest common denominator of link rates along a connection pathway;
   B) SDI_LINK_RATE_1_5_GBPS: specifies that the connection should be attempted at 1.5 Gbps; or
   C) SDI_LINK_RATE_3_0_GBPS: specifies that the connection should be attempted at 3.0 Gbps. This connection rate may not succeed if an intermediate physical link is less than 3.0 Gbps;
e) *Parameters.bDestinationSASAddress*: Specifies the SAS address of the destination device in MSB order;
f) *Parameters.uRequestLength*: Specifies the length of the function specific content in *Parameters.Request*: The length should be in LSB order and should not include the CRC bytes. The driver will be responsible for appending the proper CRC to the request at the uRequestLength offset using the function specific content;
g) *Parameters.Request*: Specifies the function specific content for the SMP request;
h) *Parameters.Request.bFrameType*: Specifies the SMP frame type (e.g., 40h)(see SAS);
i) *Parameters.Request.bFunction*: Specifies the SMP function to request (see SAS); and
j) *Parameters.Request.bAdditionalRequestBytes*: Specifies the payload bytes for the SMP function requested. Any unused bytes shall be set to 0x00.

6.11.4 Output

This function shall return a SDI_SMP_PASSTHROUGH_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Parameters.bPhyIdentifier*: Same as input;

c) *Parameters.bPortIdentifier*: Same as input;
d) *Parameters.bConnectionRate*: Same as input;
e) *Parameters.bReserved*: Same as input;
f) *Parameters.bDestinationSASAddress*: Same as input;
g) *Parameters.uRequestLength*: Same as input;
h) *Parameters.Request*: Same as input;
i) *Parameters.bConnectionStatus*: Contains the results of the connection request:
   A) SDI_OPEN_ACCEPT: indicates the connection response was OPEN_ACCEPT and the SMP request was submitted;
   B) SDI_OPEN_REJECT_BAD_DESTINATION:indicates the connection response was OPEN_REJECT (BAD DESTINATION). No request was submitted;
   C) SDI_OPEN_REJECT_RATE_NOT_SUPPORTED: indicates the connection response was OPEN_REJECT (CONNECTION RATE NOT SUPPORTED). No request was submitted;
   D) SDI_OPEN_REJECT_NO_DESTINATION: indicates the connection response was OPEN_REJECT (NO DESTINATION). No request was submitted;
   E) SDI_OPEN_REJECT_PATHWAY_BLOCKED: indicates the connection response was OPEN_REJECT (PATHWAY BLOCKED). No request was submitted;
   F) SDI_OPEN_REJECT_PROTOCOL_NOT_SUPPORTED: indicates the connection response was OPEN_REJECT (PROTOCOL NOT SUPPORTED). No request was submitted;
   G) SDI_OPEN_REJECT_RESERVE_ABANDON: indicates the connection response was OPEN_REJECT (RESERVED ABANDON 0), OPEN_REJECT (RESERVED ABANDON 1), OPEN_REJECT (RESERVED ABANDON 2), or OPEN_REJECT (RESERVED ABANDON 3). No request was submitted;
   H) SDI_OPEN_REJECT_RESERVE_CONTINUE: indicates the connection response was OPEN_REJECT (RESERVED CONTINUE 0), or OPEN_REJECT (RESERVED CONTINUE 1). No request was submitted;
   I) SDI_OPEN_REJECT_RESERVE_INITIALIZE: indicates the connection response was OPEN_REJECT (RESERVED INITIALIZE 0) or OPEN_REJECT (RESERVED INITIALIZE 1). No request was submitted;
   J) SDI_OPEN_REJECT_RESERVE_STOP: indicates the connection response was OPEN_REJECT (RESERVED STOP 0) or OPEN_REJECT (RESERVED STOP 1). No request was submitted;
   K) SDI_OPEN_REJECT_RETRY: indicates the connection response was OPEN_REJECT (RETRY). No request was submitted;
   L) SDI_OPEN_REJECT_STP_RESOURCES_BUSY: indicates the connection response was OPEN_REJECT (STP RESOURCES BUSY). No request was submitted; or
   M) SDI_OPEN_REJECT_WRONG_DESTINATION: indicates the connection response was OPEN_REJECT (WRONG DESTINATION). No request was submitted;
j) *Parameters.bReserved2*: Same as input;
k) *Parameters.bResponseBytes*: Contains the number of valid bytes in the *Parameters.Response* data structure. The CRC bytes of the response may be included in the count;
l) *Parameters.Response*: Contains the function specific response. See the SAS specification;
m) *Parameters.Response.bFrameType*: Contains the SMP response type (i.e., byte 0). Should be 41h;
n) *Parameters.Response.bFunction*: Contains the SMP function (i.e., byte 1);
o) *Parameters.Response.bFunctionResult*: Contains the SMP function result (i.e., byte 2); and
p) *Parameters.Response.bReserved*: Contains the SMP function response byte 3; and
q) *Parameters.Response.bAdditionalRequestBytes*: Contains the payload bytes for the SMP function response. Any trailing unused bytes shall be set to 0x00.

6.11.5 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_SMP_REQUEST {
   __u8  bFrameType;
   __u8  bFunction;
   __u8  bReserved[2];
   __u8  bAdditionalRequestBytes[1016];
```

```
} SDI_SMP_REQUEST, *PSDI_SMP_REQUEST;

typedef struct _SDI_SMP_RESPONSE {
   __u8  bFrameType;
   __u8  bFunction;
   __u8  bFunctionResult;
   __u8  bReserved;
   __u8  bAdditionalResponseBytes[1016];
} SDI_SMP_RESPONSE, *PSDI_SMP_RESPONSE;

typedef struct _SDI_SMP_PASSTHROUGH {
   __u8  bPhyIdentifier;
   __u8  bPortIdentifier;
   __u8  bConnectionRate;
   __u8  bReserved;
   __u8  bDestinationSASAddress[8];
   __u32 uRequestLength;
   SDI_SMP_REQUEST Request;
   __u8  bConnectionStatus;
   __u8  bReserved2[3];
   __u32 uResponseBytes;
   SDI_SMP_RESPONSE Response;
} SDI_SMP_PASSTHROUGH, *PSDI_SMP_PASSTHROUGH;

typedef struct _SDI_SMP_PASSTHROUGH_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_SMP_PASSTHROUGH Parameters;
} SDI_SMP_PASSTHROUGH_BUFFER, *PSDI_SMP_PASSTHROUGH_BUFFER;
```

6.12 CC_SDI_SSP_PASSTHROUGH

6.12.1 Behavior

The CC_SDI_SSP_PASSTHROUGH SDI function provides a method of sending generic SSP requests to a specific SAS address. Any driver that implements this specification and supports the SSP protocol shall support this SDI function; otherwise the driver may respond to this function code with a generic IO error (see Submitting Control Codes).

6.12.2 Security

A driver shall return the *IoctrlHeader.ReturnCode* field set to SDI_SECURITY_VIOLATION if the security level is insufficient to process the requested function. Only the SCSI commands listed in table 9 shall be allowed if the security level is not FULL.

Table 9 — SCSI commands allowed without full security access

SCSI command	Reference
INQUIRY	SPC-3
LOG SENSE	SPC-3
MODE SENSE (6)/(10)	SPC-3
READ BUFFER	SPC-3
READ CAPACITY	SBC-2
READ DEFECT DATA	SBC-2
READ (6)/(10)/(12)/(16)	SBC-2
REPORT LUNS	SPC-3
REQUEST SENSE	SPC-3
TEST UNIT READY	SPC-3
VERIFY (6)/(10)/(12)/(16)	SBC-2
WRITE BUFFER [a]	SPC-3

[a] To support download of microcode and for link validation, the WRITE BUFFER command requires Limited access. The end device is responsible for ensuring that any download microcode operation performed is validated with proper vendor, model and checksum associations.

6.12.3 Input

This function accepts a SDI_SSP_PASSTHROUGH_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *Status*: Initialized to 0's;
c) *Parameters.bPhyIdentifier*: Specifies the phy identifier of the phy that should be used to issue the request. The value shall be in the range of 0 to 254 (0x00 to 0xFE) or be the directive SDI_USE_PORT_IDENTIFIER. The driver may generate an error due to the phy identifier for the following reasons:
   A) The phy identifier cannot be selected because the driver does not support sending SMP requests to a phy identifier. The driver may support sending SMP requests to a bPortIdentifier only. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_CANNOT_BE_SELECTED;
   B) The phy identifier is out of range of valid phys. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_EXIST;
   C) The phy identifier cannot be associated with bPortIdentifier. If bPhyIdentifier is intended to reference the phy and the bPortIdentifier value is not SDI_IGNORE_PORT, then the bPhyIdentifier and bPortIdentifier must have the proper association. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_MATCH_PORT;
   D) The phy identifier has a value of SDI_USE_PORT_IDENTIFIER and the bPortIdentifier has a value of SDI_IGNORE_PORT. The driver cannot determine where to send the SMP request. Either bPhyIdentifier should reference a valid phy or bPortIdentifier must reference a valid port. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_SELECT_PHY_OR_PORT;
d) *Parameters.bPortIdentifier*: Contains the port identifier that specifies which port should be used to issue the request. The value must be in the range of 0 to 254 (0 to FEh) or be the directive

SDI_IGNORE_PORT. The driver may generate an error due to the port identifier for the following reasons:

A) The port identifier cannot be selected because the driver does not support sending SMP requests to a port identifier. The driver may support sending SMP requests to a bPhyIdentifier only. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PORT_CANNOT_BE_SELECTED;
B) The port identifier is out of range of valid ports. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PORT_DOES_NOT_EXIST;
C) The port identifier cannot be associated with bPhyIdentifier. If bPortIdentifier is intended to reference the port and the bPhyIdentifier value is not SDI_USE_PORT_IDENTIFIER, then the bPortIdentifier and bPhyIdentifier must have the proper association. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_MATCH_PORT;
D) The port identifier has a value of SDI_IGNORE_PORT and the bPhyIdentifier has a value of SDI_USE_PORT_IDENTIFIER. The driver cannot determine where to send the SMP request. Either bPhyIdentifier should reference a valid phy or bPortIdentifier must reference a valid port. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_SELECT_PHY_OR_PORT;

e) *Parameters.bConnectionRate*: Contains the connection rate directive for the driver connection manager. Should be one of the following:
   A) SDI_LINK_RATE_NEGOTIATED: specifies that the connection should be opened at the highest allowable negotiated rate for the destination device. The resulting rate will be the lowest common denominator of link rates along a connection pathway;
   B) SDI_LINK_RATE_1_5_GBPS: specifies that the connection should be attempted at 1.5 Gbps;
   C) SDI_LINK_RATE_3_0_GBPS: specifies that the connection should be attempted at 3.0 Gbps. This connection rate may not succeed if an intermediate physical link is less than 3.0 Gbps;
f) *Parameters.bReserved*: This field shall be set to 0;
g) *Parameters.bDestinationSASAddress*: Contains the SAS address of the destination device in MSB order;
h) *Parameters.bLun*: Contains the LUN of the target physical device to address. Equivalent to the SSP LOGICAL UNIT NUMBER field in an SSP information unit;
i) *Parameters.bCDBLength*: Contains the length of the CDB in *Parameters.bCDB*;
j) *Parameters.bAdditionalCDBLength*: Contains the length of valid dwords in *Parameters.bAdditionalCDB*. Shall be in the range of 0 to 6;
k) *Parameters.bReserved*2 This field shall be set to 0;
l) *Parameters.bCDB*: Contains the CDB bytes for the specific SCSI command to send;
m) *Parameters.uFlags*: Contains the directive that tells the SSP link and transport layers whether the command is expected to send or receive data. Should be one or more of the following:
   A) SDI_SSP_READ: specifies that the data transfer will be from the destination device;
   B) SDI_SSP_WRITE: specifies that the data transfer will be to the destination device;
   C) SDI_SSP_UNSPECIFIED: specifies that there will be no data transfer, or the data transfer direction is unknown and any data received until the ITL nexus is completed should be retained;
   D) SDI_SSP_TASK_ATTRIBUTE_SIMPLE: specifies that the Task attribute for the SSP command information unit should be set to SIMPLE;
   E) SDI_SSP_TASK_ATTRIBUTE_HEAD_OF_QUEUE: specifies that the Task attribute for the SSP command information unit should be set to HEAD_OF_QUEUE;
   F) SDI_SSP_TASK_ATTRIBUTE_ORDERED: specifies that the Task attribute for the SSP command information unit should be set to ORDERED;
   G) SDI_SSP_TASK_ATTRIBUTE_ACA:specifies that the Task attribute for the SSP command information unit should be set to ACA;

NOTE 4 - Only one of the Task attribute flags may be included in *Parameters.uFlags*. If more than one is specified, then all are ignored and the result equivalent to SDI_SSP_TASK_ATTRIBUTE_SIMPLE.

n) *Parameters.bAdditionalCDB*: Specifies the additional CDB bytes, if any;
o) *Parameters.uDataLength*: Specifies the length of *bDataBuffer* in LSB order; and
p) *bDataBuffer*: Specifies any data that is being written to the device (for write commands) or provides a memory space for any data that is being read from the device (for read commands).

Editor's Note 46: there should be two data pointers to support bidirectional transfers, with two length fields. Or, use one big field with part for read data and part for write data. first length field would cover either read or write data for unidirectional commands, and would cover write data for bidirectional commands. Second length field would cover read data for bidirectional commands.

6.12.4 Output

This function shall return a SDI_SSP_PASSTHROUGH_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Parameters.bPhyIdentifier*: Same as input;
c) *Parameters.bPortIdentifier*: Same as input;
d) *Parameters.bConnectionRate*: Same as input;
e) *Parameters.bDestinationSASAddress*: Same as input;
f) *Parameters.bLun*: Same as input;
g) *Parameters.bCDBLength*: Same as input;
h) *Parameters.bAdditionalCDBLength*: Same as input;
i) *Parameters.bCDB*: Same as input;
j) *Parameters.uFlags*: Same as input;
k) *Parameters.bAdditionalCDB*: Same as input;
l) *Parameters.uDataLength*: Same as input;
m) *Status*: Contains the SSP status structure for the SSP command;
n) *Status.bConnectionStatus*: Contains the results of the connection request. See the *Status.bConnectionStatus* field in the SDI_SMP_PASSTHROUGH command (see 6.11.4);
o) *Status.bDataPresent*: Contains the directives that indicate what has been returned in bResponse. Should be one of the following:
   A) SDI_SSP_NO_DATA_PRESENT: see SCSI specification;
   B) SDI_SSP_RESPONSE_DATA_PRESENT: see SCSI specification;
   C) SDI_SSP_SENSE_DATA_PRESENT: see SCSI specification;
p) *Status.bStatus*: Contains the SCSI status code;
q) *Status.bResponseLength*: Contains the number of valid bytes in the bResponse field in MSB order;
r) *Status.bResponse*: Contains the response bytes in MSB order. The interpretation of the data depends on the directive in the bDataPresent field;

Editor's Note 47: this is intended to carry either sense data or response data (or both) - whatever is in the RESPONSE frame (based on the DataPres field). Change names bResponseLength and bResponse. Note that the Task Management function also uses this structure but only to return response data, not sense data.

Editor's Note 48: italicize bResponse, etc. above

s) *uDataBytes*: Contains the number of valid bytes in bDataBuffer, in LSB order; and

Editor's Note 49: uDataBytes should name something like Length or Num in its name

t) *bDataBuffer*: Contains any data that has been written to the device (write commands) or contains any data that has been read from the device (read commands).

6.12.5 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_SSP_PASSTHROUGH {
   __u8  bPhyIdentifier;
   __u8  bPortIdentifier;
   __u8  bConnectionRate;
   __u8  bReserved;
   __u8  bDestinationSASAddress[8];
   __u8  bLun[8];
   __u8  bCDBLength;
   __u8  bAdditionalCDBLength;
   __u8  bReserved2[2];
   __u8  bCDB[16];
   __u32 uFlags;
   __u8  bAdditionalCDB[24];
   __u32 uDataLength;
} SDI_SSP_PASSTHROUGH, *PSDI_SSP_PASSTHROUGH;

typedef struct _SDI_SSP_PASSTHROUGH_STATUS {
   __u8  bConnectionStatus;
   __u8  bReserved[3];
   __u8  bDataPresent;
   __u8  bStatus;
   __u8  bReponseLength[2];
   __u8  bResponse[256];
} SDI_SSP_PASSTHROUGH_STATUS, *PSDI_SSP_PASSTHROUGH_STATUS;

typedef struct _SDI_SSP_PASSTHROUGH_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_SSP_PASSTHROUGH Parameters;
   SDI_SSP_PASSTHROUGH_STATUS Status;
__u32 uDataBytes;
   __u8  bDataBuffer[1];
} SDI_SSP_PASSTHROUGH_BUFFER, *PSDI_SSP_PASSTHROUGH_BUFFER;
```

Editor's Note 50: fix indenting above

6.13 CC_SDI_STP_PASSTHROUGH

6.13.1 Behavior

The CC_SDI_STP_PASSTHROUGH SDI function provides a method of sending generic STP or SATA commands to a specific SAS address. Any driver that implements this specification and supports the STP or SATA protocols shall support this function; otherwise the driver may respond to this function code with a generic IO error (see Submitting Control Codes). A driver that emulates STP or SATA devices as SCSI devices may require that commands directed to STP or SATA devices be directed to the SCSI link and transport layer. To facilitate sending generic STP or SATA commands with that restriction an alternative mechanism using a special SCSI command to wrap SATA commands may be provided (see Error! Reference source not found.). A driver may direct the upper level application to use the alternative method by returning the *IoctrlHeader.ReturnCode* field set to SDI_SCSI_EMULATION.

6.13.2 Security

A driver shall return the *IoctrlHeader.ReturnCode* field set to SDI_SECURITY_VIOLATION if the security level is insufficient to complete the requested function. Only the ATA commands listed in table 10 shall be allowed if the security level is not FULL.

Table 10 — ATA commands allowed without full security access

ATA command	Reference
CHECK POWER MODE	ATA/ATAPI-7 V1
DOWNLOAD MICROCODE [a]	ATA/ATAPI-7 V1
EXECUTE DEVICE DIAGNOSTICS	ATA/ATAPI-7 V1
FLUSH CACHE/FLUSH CACHE EXT	ATA/ATAPI-7 V1
IDENTIFY DEVICE/IDENTIFY PACKET DEVICE	ATA/ATAPI-7 V1
NOP	ATA/ATAPI-7 V1
PACKET [b]	ATA/ATAPI-7 V1
READ BUFFER	ATA/ATAPI-7 V1
READ DMA/READ DMA EXT/READ DMA QUEUED/READ DMA QUEUED EXT	ATA/ATAPI-7 V1
READ LOG EXT	ATA/ATAPI-7 V1
READ MULTIPLE/READ MULTIPLE EXT	ATA/ATAPI-7 V1
READ NATIVE MAX ADDRESS/READ NATIVE MAX ADDRESS EXT	ATA/ATAPI-7 V1
READ SECTOR(S)/READ SECTOR(S) EXT	ATA/ATAPI-7 V1
READ VERIFY SECTOR(S)/READ VERIFY SECTOR(S) EXT	ATA/ATAPI-7 V1
SMART	ATA/ATAPI-7 V1
WRITE BUFFER [c]	ATA/ATAPI-7 V1

[a] To support download of microcode, the DOWNLOAD MICROCODE command requires Limited access. The end device is responsible for ensuring that any download operation performed is validated with proper vendor, model and checksum associations.
[b] The SCSI command being sent with PACKET shall be processed according to the SCSI command security access (see table 9 in 6.12).
[c] For link verification.

6.13.3 Input

This function accepts a SDI_STP_PASSTHROUGH_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *Parameters.bPhyIdentifier*: Contains the phy identifier that specifies which phy should be used to issue the request. The value must be in the range of 0 to 254 (0 to FEh) or be the directive SDI_USE_PORT_IDENTIFIER. The driver may generate an error due to the phy identifier for the following reasons:
   A) The phy identifier cannot be selected because the driver does not support sending SMP requests to a phy identifier. The driver may support sending SMP requests to a bPortIdentifier only. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_CANNOT_BE_SELECTED;
   B) The phy identifier is out of range of valid phys. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_EXIST;
   C) The phy identifier cannot be associated with bPortIdentifier. If bPhyIdentifier is intended to reference the phy and the bPortIdentifier value is not SDI_IGNORE_PORT, then the bPhyIdentifier and bPortIdentifier must have the proper association. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_MATCH_PORT;

D) The phy identifier has a value of SDI_USE_PORT_IDENTIFIER and the bPortIdentifier has a value of SDI_IGNORE_PORT. The driver cannot determine where to send the SMP request. Either bPhyIdentifier should reference a valid phy or bPortIdentifier must reference a valid port. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_SELECT_PHY_OR_PORT;

c) *Parameters.bPortIdentifier*: Contains the port identifier that specifies which port should be used to issue the request. The value must be in the range of 0 to 254 (0 to FEh) or be the directive SDI_IGNORE_PORT. The driver may generate an error due to the port identifier for the following reasons:
   A) The port identifier cannot be selected because the driver does not support sending SMP requests to a port identifier. The driver may support sending SMP requests to a bPhyIdentifier only. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PORT_CANNOT_BE_SELECTED;
   B) The port identifier is out of range of valid ports. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PORT_DOES_NOT_EXIST;
   C) The port identifier cannot be associated with bPhyIdentifier. If bPortIdentifier is intended to reference the port and the bPhyIdentifier value is not SDI_USE_PORT_IDENTIFIER, then the bPortIdentifier and bPhyIdentifier must have the proper association. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_MATCH_PORT;
   D) The port identifier has a value of SDI_IGNORE_PORT and the bPhyIdentifier has a value of SDI_USE_PORT_IDENTIFIER. The driver cannot determine where to send the SMP request. Either bPhyIdentifier should reference a valid phy or bPortIdentifier must reference a valid port. The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_SELECT_PHY_OR_PORT;

d) *Parameters.bConnectionRate*: Contains the connection rate directive for the driver connection manager. Should be one of the following:
   A) SDI_LINK_RATE_NEGOTIATED: specifies that the connection shall be opened at the highest allowable negotiated rate for the destination device. The resulting rate will be the lowest common denominator of link rates along a connection pathway;
   B) SDI_LINK_RATE_1_5_GBPS: specifies that the connection shall be attempted at 1.5 Gbps;
   C) SDI_LINK_RATE_3_0_GBPS: specifies that the connection shall be attempted at 3.0 Gbps. This connection rate may not succeed if an intermediate link is less than 3.0 Gbps;

e) *Parameters.bDestinationSASAddress*: Contains the SAS address of the destination device in MSB order;

f) *Parameters.bCommandFIS*: Contains the SATA command FIS (27h). See the SATA specification;

g) *Parameters.uFlags*: Contains the directive that tells the STP and/or SATA link and transport layers whether the command is expected to send or receive data. Should be one or more of the following:
   A) SDI_STP_READ: specifies that the data transfer will be from the destination device;
   B) SDI_STP_WRITE: specifies that the data transfer will be to the destination device;
   C) SDI_STP_UNSPECIFIED: specifies that there will be no data transfer, or the data transfer direction is unknown and any data received should be retained;
   D) SDI_STP_PIO: specifies the command follows the SATA PIO state machine for completion;
   E) SDI_STP_DMA: specifies the command follows the SATA DMA state machine for completion;
   F) SDI_STP_PACKET: specifies the command follows the SATA packet state machine for completion;
   G) SDI_STP_DMA_QUEUED: specifies the command follows the SATA DMA queued state machine for completion;
   H) SDI_STP_EXECUTE_DIAG: specifies the command follows the SATA execute diagnostic state machine for completion; and/or
   I) SDI_STP_RESET_DEVICE: specifies that a soft reset is being performed;

   and

h) *bDataBuffer*: Contains any data that is being written to the device (write commands) or provides a memory space for any data that is being read from the device (read commands).

6.13.4 Output

This function shall return a SDI_STP_PASSTHROUGH_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;

b) *Parameters.bPhyIdentifier*: Same as input;
c) *Parameters.bPortIdentifier*: Same as input;
d) *Parameters.bConnectionRate*: Same as input;
e) *Parameters.bReserved*: Same as input;
f) *Parameters.bDestinationSASAddress*: Same as input;
g) *Parameters.bReserved2*: Same as input;
h) *Parameters.bCommandFIS*: Same as input;
i) *Parameters.uFlags*: Same as input;
j) *Parameters.uDataLength*: Same as input;
k) *Status*: Contains the STP status structure for the STP or SATA command;
l) *Status.bConnectionStatus*: Contains the results of the connection request. See the *Status.bConnectionStatus* field in the SDI_SMP_PASSTHROUGH command (see 6.11.4);
m) *Status.bReserved*: This field shall be set to 0;
n) *Status.bStatusFIS*: Contains the SATA status FIS (34h). See SATA specification;
o) *Status.uSCR*: Contains the status control registers. The contents of uSCR are be updated at the completion of the command. Register level polling is not intended;
p) *uDataBytes*: Contains the number of valid bytes in bDataBuffer, in LSB order; and
q) *bDataBuffer*: Contains any data that has been written to the device (write commands) or contains any data that has been read from the device (read commands).

6.13.5 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_STP_PASSTHROUGH {
   __u8  bPhyIdentifier;
   __u8  bPortIdentifier;
   __u8  bConnectionRate;
   __u8  bReserved;
   __u8  bDestinationSASAddress[8];
   __u8  bReserved2[4];
   __u8  bCommandFIS[20];
   __u32 uFlags;
   __u32 uDataLength;
} SDI_STP_PASSTHROUGH, *PSDI_STP_PASSTHROUGH;

typedef struct _SDI_STP_PASSTHROUGH_STATUS {
   __u8  bConnectionStatus;
   __u8  bReserved[3];
   __u8  bStatusFIS[20];
   __u32 uSCR[16];
} SDI_STP_PASSTHROUGH_STATUS, *PSDI_STP_PASSTHROUGH_STATUS;

typedef struct _SDI_STP_PASSTHROUGH_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_STP_PASSTHROUGH Parameters;
   SDI_STP_PASSTHROUGH_STATUS Status;
__u32 uDataBytes;
__u8  bDataBuffer[1];
} SDI_STP_PASSTHROUGH_BUFFER, *PSDI_STP_PASSTHROUGH_BUFFER;
```

Editor's Note 51: fix indenting above

6.14 CC_SDI_GET_SATA_SIGNATURE

6.14.1 Behavior

The CC_SDI_GET_SATA_SIGNATURE SDI function provides a method of obtaining the initial SATA signature (i.e., the initial Register Device to Host FIS) from a directly attached SATA device. The signature may be used to identify whether a SATA device supports the PACKET command set or whether it is a unique SATA device (like a port multiplier). Any driver that implements this specification and supports directly attached SATA devices shall support this SDI function; otherwise the driver may respond to this function code with a generic IO error (see Submitting Control Codes).

6.14.2 Input

This function accepts a SDI_SATA_SIGNATURE_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *Signature.bPhyIdentifier*: Contains the phy identifier that is being queried for a SATA signature. The driver may generate an error due to the phy identifier for the following reasons:
   A) If the phy does not have a SATA device directly attached or the phy has not completed the link reset sequence, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_NO_SATA_DEVICE;
   B) If the phy has not received the initial register device to host FIS from the SATA device, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_NO_SATA_SIGNATURE; and
   C) If the phy does not exist, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_PHY_DOES_NOT_EXIST;
c) *Signature.bReserved*: This field shall be set to 0; and
d) *Signature.bSignatureFIS*: This field shall be set to 0.

6.14.3 Output

This function shall return a SDI_SATA_SIGNATURE_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Signature.bPhyIdentifier*: Same as input; and
c) *Signature.bReserved*: Same as input; and
d) *Signature.bSignatureFIS*: Contains the initial register device to host FIS (34h) from the SATA device. Only the signature bytes are required to be valid, the remainder of the FIS may be 0 filled. If the FIS type is valid, (i.e. 34h) then the entire FIS is assumed to be valid (i.e. as returned from the device).

6.14.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_SATA_SIGNATURE {
   __u8  bPhyIdentifier;
   __u8  bReserved[3];
   __u8  bSignatureFIS[20];
} SDI_SATA_SIGNATURE, *PSDI_SATA_SIGNATURE;

typedef struct _SDI_SATA_SIGNATURE_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_SATA_SIGNATURE Signature;
} SDI_SATA_SIGNATURE_BUFFER, *PSDI_SATA_SIGNATURE_BUFFER;
```

6.15 CC_SDI_GET_SCSI_ADDRESS

6.15.1 Behavior

The CC_SDI_GET_SCSI_ADDRESS SDI function provides a method of obtaining the OS specific platform address for a SAS address.

The driver may generate an error on this request for the following reasons:

a) If the SAS address is to an expander device, the driver shall return the *IoctrlHeader.ReturnCode* field set to SDI_NOT_AN_END_DEVICE; and
b) If the SAS address does not have an associated OS specific address, the driver shall return the *IoctrlHeader.ReturnCode* field set to SDI_NO_SCSI_ADDRESS.

6.15.2 Input

This function accepts a SDI_GET_SCSI_ADDRESS_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *bSASAddress*: Specifies the SAS address of the device, in MSB order;
c) *bSASLun*: Specifies the SCSI logical unit number of the device, in MSB order;
d) *bHostIndex*: This field shall be set to 0;
e) *bPathId*: This field shall be set to 0;
f) *bTargetId*: This field shall be set to 0; and
g) *bLun*: This field shall be set to 0.

Editor's Note 52: path 0, target 0, lun 0 is valid which could be misleading in the Output. However, the caller is not supposed to use the results in these fields if an error is returning.

This function shall return a SDI_GET_SCSI_ADDRESS_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *bSASAddress*: Same as input;
c) *bSASLun*: Same as input;
d) *bHostIndex*: Indicates the enumerated index of the driver instance (for example, the n value in "SCSIn" under Windows). An FFh indicates the value is invalid;
e) *bPathId*: Indicates the path (i.e., bus or port) identifier of the device;
f) *bTargetId*: Indicates the target identifier of the device; and
g) *bLun*: Indicates the logical unit number of the device.

6.15.3 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_GET_SCSI_ADDRESS_BUFFER {
   IOCTL_HEADER IoctlHeader;
   __u8  bSASAddress[8];
   __u8  bSASLun[8];
   __u8  bHostIndex;
   __u8  bPathId;
   __u8  bTargetId;
   __u8  bLun;
} SDI_GET_SCSI_ADDRESS_BUFFER, *PSDI_GET_SCSI_ADDRESS_BUFFER;
```

6.16 CC_SDI_GET_DEVICE_ADDRESS

6.16.1 Behavior

The CC_SDI_GET_DEVICE_ADDRESS SDI function provides a method of obtaining the SAS address of a device from an OS specific platform address.

Editor's Note 53: use of "DEVICE" is problematic since that term has specific meanings in SCSI. SCSI and even Windows use the term "peripheral device" to represent a logical unit (different from the target device, that can contain many different logical units each with a different peripheral

device type).Would SDI_GET_PERIPHERAL_DEVICE_ADDRESS or SDI_GET_SAS_ADDRESS_LUN be better?

The driver may generate an error on this request for the following reasons:

a) The OS specific platform address does not have a SAS address. The driver shall return the *IoctrlHeader.ReturnCode* field set to SDI_NO_DEVICE_ADDRESS.

6.16.2 Input

This function accepts a SDI_GET_DEVICE_ADDRESS_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *bHostIndex*: Specifies the enumerated index of the driver instance (for example, the n value in “SCSIn” under Windows). An FFh indicates the value is invalid;
c) *bPathId*: Specifies the path (i.e., bus or port) identifier of the device;
d) *bTargetId*: Specifies the target identifier of the device;
e) *bLun*: Specifies the logical unit number of the device;
f) *bSASAddress*: This field shall be set to to 0; and
g) *bSASLun*: This field shall be set to to 0.

6.16.3 Output

This function shall return a SDI_GET_DEVICE_ADDRESS_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *bHostIndex*: Same as input;
c) *bPathId*: Same as input;
d) *bTargetId*: Same as input;
e) *bLun*: Same as input;
f) *bSASAddress*: Indicates the SAS address of the device, in MSB order; and
g) *bSASLun*: Indicates the SAS logical unit number of the device, in MSB order.

6.16.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_GET_DEVICE_ADDRESS_BUFFER {
   IOCTL_HEADER IoctlHeader;
   __u8  bHostIndex;
   __u8  bPathId;
   __u8  bTargetId;
   __u8  bLun;
   __u8  bSASAddress[8];
   __u8  bSASLun[8];
} SDI_GET_DEVICE_ADDRESS_BUFFER, *PSDI_GET_DEVICE_ADDRESS_BUFFER;
```

6.17 CC_SDI_TASK_MANAGEMENT

6.17.1 Behavior

The CC_SDI_TASK_MANAGEMENT SDI function provides a method of sending a hard reset sequence or a TASK frame to the specified OS specific platform address.

Editor’s Note 54: if the device has a wide target port, which phy of the HBA or expander is used to send a hard reset sequence? Lowest, highest, any, all? Move HARD_RESET_SEQUENCE into its

own SDI function which includes a phy number. Task management functions are different beasts.

6.17.2 Security

The driver shall return the *IoctrlHeader.ReturnCode* field set to SDI_SECURITY_VIOLATION if the security level is not FULL.

6.17.3 Input

This function accepts a SDI_SSP_TASK_IU_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *Parameters.bHostIndex*: Specifies the enumerated index of the driver instance (for example, the n value in “SCSIn” under Windows). An FFh indicates the value is invalid;
c) *Parameters.bPathId*: Specifies the path (i.e., bus or port) identifier of the device;
d) *Parameters.bTargetId*: Specifies the target identifier of the device;
e) *Parameters.bLun*: Specifies the logical unit number of the device;
f) *Parameters.uFlags*: Specifies one or more of the following:
   A) SDI_TASK_IU: When set, the *Parameters.uTagOfTaskToBeManaged* and *Parameters.bTaskManagementFunction* fields contain the information to be provided in a TASK frame. If set, the SDI_HARD_RESET_SEQUENCE shall not be set;
   B) SDI_HARD_RESET_SEQUENCE: When set, the driver shall issue a hard reset sequence to the OS specific platform address. If the device is directly attached, the HBA does this; if the device is attached to an expander, the driver sends an SMP PHY CONTROL function to that expander requesting a HARD_RESET phy operation. There should no delay inserted by the driver after issuing the hard reset sequence (i.e., it should not wait for the sequence to complete). If set, the SDI_TASK_IU shall not be set;
   C) SDI_SUPPRESS_RESULT: Optional flag when set, the OS low-level driver shall suppress reporting the task management event to the upper level driver;
g) *Parameters.uTagOfTaskToBeManaged*: Specifies the tag of the task to be managed. If the task management function does not use tag value, then this field shall be set to zero;

Editor’s Note 55: Some TMFs don’t care about the LUN (formerly TARGET RESET; upcoming I_T NEXUS RESET). Need to require this be passed in as 0 or just make it a don’t care?

h) *Parameters.bTaskManagementFunction*: Specifies the contents of the TASK MANAGEMENT FUNCTION field of the TASK frame (see SAS). This should be set to one of the following:
   A) SDI_SSP_ABORT_TASK;
   B) SDI_SSP_ABORT_TASK_SET;
   C) SDI_SSP_CLEAR_TASK_SET;
   D) SDI_SSP_LOGICAL_UNIT_RESET;
   E) SDI_SSP_CLEAR_ACA; or
   F) SDI_SSP_QUERY_TASK;
i) *Parameters.uInformation*: Specifies application-specific information about why this task management function is being sent. May be set to one of the following:
   A) SDI_SSP_TEST: Specifies that the task management request was sent as part of a general test procedure;
   B) SDI_SSP_EXCEEDED: Specifies that the task management request was sent to terminate an outstanding command that has exceeded a time limit;
   C) SDI_SSP_DEMAND: Specifies that the task management request was sent on demand from an application; or
   D) SDI_SSP_TRIGGER: Specifies that the task management request is being used as a trigger event by an application;

   and

j) *Status*: This field shall be set to 0.

6.17.4 Output

This function shall return a SDI_SSP_TASK_IU_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Parameters.bHostIndex*: Same as input;
c) *Parameters.bPathId*: Same as input;
d) *Parameters.bTargetId*: Same as input;
e) *Parameters.bLun*: Same as input;
f) *Parameters.uFlags*: Same as input;
g) *Parameters.uTagOfTaskToBeManaged*: Same as input;
h) *Parameters.bTaskManagementFunction*. Same as input;
i) *Parameters.uInformation*: Same as input;
j) *Status*: Contains the SSP status structure for the SSP command;
k) *Status.bConnectionStatus*: Indicates the results of the connection request. See the *Status.bConnectionStatus* field in the SDI_SMP_PASSTHROUGH command (see 6.11.4);
l) *Status.bDataPresent*: Indicates the contents of byte 10 of the RESPONSE information unit (i.e., the DATAPRES field)(see SAS). This field should be set to one of the following:
   A) SDI_SSP_NO_DATA_PRESENT: Neither response data nor sense data is present;
   B) SDI_SSP_RESPONSE_DATA_PRESENT: Response data is present;
   C) SDI_SSP_SENSE_DATA_PRESENT: Sense data is present;
m) *Status.bStatus*: Indicates the SCSI status code;
n) *Status.bResponseLength*: Indicates the number of valid bytes in the *Status.bResponse* field in MSB order; and
o) *Status.bResponse*: Contains the response data bytes in MSB order. The interpretation of the data depends on the directive in the bDataPresent field.

6.17.5 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_SSP_TASK_IU {
   __u8  bHostIndex;
   __u8  bPathId;
   __u8  bTargetId;
   __u8  bLun;
   __u32  uFlags;
   __u32  uTagOfTaskToBeManaged;
   __u32 uReserved;
   __u8 bTaskManagementFunction;
   __u8 bReserved[7];
   __u32 uInformation;
} SDI_SSP_TASK_IU, *PSDI_SSP_TASK_IU;

typedef struct _SDI_SSP_TASK_IU_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_SSP_TASK_IU Parameters;
   SDI_SSP_PASSTHROUGH_STATUS Status;
} SDI_SSP_TASK_IU_BUFFER, *PSDI_SSP_TASK_IU_BUFFER;
```

Editor's Note 56: should this reuse SDI_SSP_PASSTHROUGH_STATUS or should it have its own structure?

6.18 CC_SDI_PHY_CONTROL

6.18.1 Behavior

The CC_SDI_PHY_CONTROL SDI function provides a method of determining and setting the phy characteristics of the controller. The phy control features include: low level reset control, SATA port selection control, phy signal control, and phy pattern generation. Since this function supports functions that are tightly coupled with hardware implementations, full support for every phy signal control is not required. If the hardware is capable of supporting a specific phy signal control then the associated function should be supported.

6.18.2 Security

The driver shall return the *IoctrlHeader.ReturnCode* field set to SDI_SECURITY_VIOLATION if the security level is not FULL.

6.18.3 Spinup behavior model

This SDI function supports controls that may affect the spinup behavior of devices. The programming model used to define this function assumes that the device spinup window is global for the controller. This means that when end devices are directly connected to the controller across the controller phys there is only one window of opportunity to spinup a device. This prevents power supply overload conditions caused by multiple devices spinning up at the same time. The model further assumes that the spinup window is enabled by a token that is passed from phy to phy starting with phy 0 and wrapping around from the last phy back to phy 0. The spinup rate defined in this SDI function is intended to specify the time the token wait before being passed to the next phy in the loop. As an example if a controller has 4 phys and the spinup rate is set for 3 seconds, then;

1) At 0 seconds:
   A) Phy 0 outputs a NOTIFY(SPINUP) primitive for SAS or send a COMWAKE in response to a COMINIT for SATA;
   B) Phy 1 remains idle;
   C) Phy 2 remains idle; and
   D) Phy 3 remains idle;
2) At 3 seconds:
   A) Phy 0 remains idle;
   B) Phy 1 output a NOTIFY(SPINUP) primitive for SAS or send a COMWAKE in response to a COMINIT for SATA;
   C) Phy 2 remains idle; and
   D) Phy 3 remains idle;
3) At 6 seconds:
   A) Phy 0 remains idle;
   B) Phy 1 remains idle;
   C) Phy 2 outputs a NOTIFY(SPINUP) primitive for SAS or send a COMWAKE in response to a COMINIT for SATA; and
   D) Phy 3, remains idle;
4) At 9 seconds:
   A) Phy 0 remains idle;
   B) Phy 1 remains idle;
   C) Phy 2 remains idle; and
   D) Phy 3 outputs a NOTIFY(SPINUP) primitive for SAS or send a COMWAKE in response to a COMINIT for SATA;
5) At 12 seconds, repeat from step 1

From this example a 3 second rate translates into a minimum 9 second waiting period for all devices to be given an opportunity to spinup.

A model that is not global in nature should also be supported by this SDI function, but may have a different interaction between the spinup rate provided and the actual device ready times.

6.18.4 Phy signal control behavior model

The SDI function supports controls that may affect communication with end devices.

The driver should limit the possible range of controls to ensure excessive voltages are not generated by the phy.

The programming model assumes that any signal level changes will occur when the phy is in an inactive state and will be followed by either a link reset sequence or a hard link reset sequence.

The pattern generation behavior assumes that the controller receivers will ignore any input from the end device (if any) and simply provide a constant stream of data based on the pattern requested.

If the controller has any active IO outstanding at the time a pattern generation behavior is requested The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_FAILED and the requested function shall not be performed.

6.18.5 Input

This function accepts a SDI_PHY_CONTROL_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *uFunction*: Specifies the function to perform and may be one of the following:
   A) SDI_PC_LINK_RESET: Specifies that the specified phy should perform a link reset sequence. The phy identifier (see bPhyIdentifier) specifies which phy should participate in this function. Depending on the remaining parameters in the structure one of the following behaviors is performed:
      a) If the length of control (see bLengthOfControl), number of controls (see bNumberOfControls) and control structure (see Control) properly define one or more phy controls, then after going to the common mode state and prior to initiating the first COMRESET the phy control(s) should be used to update the current phy settings;
      b) If the length of control, number of controls, and control structure do not properly define one or more phy controls, then the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and the link reset sequence shall not be performed;
      c) If the length of control, number of controls and control structure are all 0 filled, then a link reset sequence is performed without altering the current phy settings;
   B) SDI_PC_HARD_RESET: Specifies that the specified phy should perform a hard link reset sequence. The phy identifier specified by the *bPhyIdentifier* field specifies which phy should participate in this function. Depending on the remaining parameters in the structure one of the following behaviors shall be performed:
      a) If the length of the control structure specified by the *usLengthOfControl* field, the number of controls specified by the *bNumberOfControls* field, and the control structure specified by the *Control* field properly define one or more phy controls, then after going to the common mode state and prior to initiating the first COMRESET the phy control(s) should be used to update the current phy settings;
      b) If the length of the control structure, the number of controls, and the control structure do not properly define one or more phy controls, then the driver shall return the *IoctlHeader.ReturnCode* fieldset to SDI_STATUS_INVALID_PARAMETER and the link reset sequence shall not be performed;
      c) If the length of control, the number of controls, and the control structure are all 0 filled, then a hard link reset sequence shall be performed without altering the current phy settings;
   C) SDI_PC_PHY_DISABLE: Specifies that the specified phy should be disabled. The phy identifier (see bPhyIdentifier) specifies which phy should participate in this function. The length of control (see bLengthOfControl) the number of controls (see bNumberOfControls) and control (see Control) structures should all be 0 filled;
   D) SDI_PC_GET_PHY_SETTINGS: Specifies that the necessary number of SDI_PHY_CONTROL structures should be updated to reflect the current phy settings for each control type (see *bType*) and rate (see *bRate*) supported. For example if the SAS controller supports SATA and SAS devices at 1.5 Gbps and 3.0 Gbps link rates, then 4 SDI_PHY_CONTROL structures should be returned. The order of the structures returned is not defined. The phy identifier (see

bPhyIdentifier) specifies which phy should participate in this function. The length of control (see *bLengthOfControl*) the number of controls (see *bNumberOfControls*) and control (see *Control*) structures should all be 0 filled on input;

c) *bPhyIdentifier*: Contains the phy identifier of the phy to control or query;
d) *usLengthOfControl*: Contains the length of the phy control structure. If the length is required for the function and is incorrect then The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and the value shall be updated to reflect the correct length on return;
e) *bNumberOfControls*: Contains the number of SDI_PHY_CONTROL elements in the Control array. If the number of controls is required for the function and is incorrect then the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and the value shall be updated to reflect the correct number of controls on return;
f) *bReserved*: This field shall be set to 0;
g) *uLinkFlags*: Contains flags that define basic link behavior and may be one or more of the following:
   A) SDI_PHY_ACTIVATE_CONTROL: specifies that the link behavior provided should be performed. If this flag is not set, then the link behavior will not be modified during input or are not active during output.;
   B) SDI_PHY_UPDATE_SPINUP_RATE: specifies that the spinup rate (see uSpinupRate) should be used to alter the repetition rate of NOTIFY(SPINUP) primitives for SAS or the release interval of COMWAKE in response to a COMINIT for SATA. The notify spinup rate may be global in nature across all phys, so the application must compensate for this by validating the resulting value by using the SDI_PC_GET_PHY_SETTINGS after updating all phys. If this flag is not supported, then the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and no change shall occur; or
   C) SDI_PHY_AUTO_COMWAKE: specifies that there is no release interval for COMWAKE in response to a COMINIT for SATA. This means that a SATA drive will be released to spinup immediately. If set in conjunction with SDI_PHY_UPDATE_SPINUP_RATE, then The driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and no change shall occur. If this flag is not supported, then the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and no change shall occur;
h) *bSpinupRate*: Contains the repetition rate at which the NOTIFY(SPINUP) primitive is generated on this phy for SAS devices or the release interval of COMWAKE in response to a COMINIT for SATA.. The value is in seconds. A 0 value indicates that the NOTIFY(SPINUP) primitive generation is disabled for SAS or COMWAKE is not released in response to a COMINIT for SATA. The result is that a device should stay in the non-spinup state indefinitely. If the value is out of range, then the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and the maximum value of the spinup rate shall be set;
i) *bLinkReserved*: This field shall be set to 0;
j) *uVendorUnique[8]*: Contains vendor unique information. Vendor is responsible for positively detecting the validity of the data provided. Each dword shall be initialized to 0 when not providing vendor unique information;
k) *Control[]*: The elements of this SDI_PHY_CONTROL data structure contain phy signal controls. If the control structure is required for the function and is incorrect then the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER;
l) *Control[].bType*: Specifies the device type of the control and may be one of the following:
   A) SDI_SATA: Specifies that the phy settings should be applied when a SATA device is attached to the phy; or
   B) SDI_SAS: Specifies that the phy settings should be applied when a SAS device is attached to the phy;
m) *Control[].bRate*: Contains the link rate for which the setting or control applies and may be one of the following:
   A) SDI_LINK_RATE_UNKNOWN: Specifies that the structure content is unknown on input or invalid on output;
   B) SDI_LINK_RATE_1_5_GPBS: Specifies that the structure content is valid for a 1.5 Gbps link rate;
   C) SDI_LINK_RATE_3_0_GBPS: Specifies that the structure content is valid for a 3.0 Gbps link rate;
n) *Control[].bReserved*: This field shall be set to 0;

o) *Control[].uVendorUnique*: Contains vendor unique information. Vendor is responsible for positively detecting the validity of the data provided. Should be initialized to 0 when not providing vendor unique information;
p) *Control[].uTransmitterFlags*: Contains flags that define the transmitter characteristics or link characteristics. The value may be one of the following:
   A) SDI_PHY_PREEMPHASIS_DISABLED: Specifies that preemphasis on the transmitter should be disabled during input or is disabled during output;
q) *Control[].bTransmitterAmplitude*: Specifies the step offset from the default setting that the transmitter shall use to establish the transmitter driver voltage amplitude. The field value should be treated as a 2's-complement signed value that can range from –128 to +127. If the step requested is out of range of the transmitter capability, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER. A value of 0 should always be accepted, even if the control is not supported;
r) *Control[].bTransmitterPreemphasis*: Specifies the step offset from the default setting that the transmitter shall use to establish the transmitter driver voltage preemphasis. The field value should be treated as a 2's-complement signed value that can range from –128 to +127. If the step requested is out of range of the transmitter capability, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER. A value of 0 should always be accepted, even if the control is not supported;
s) *Control[].bTransmitterSlewRate*: Specifies the step offset from the default setting that the transmitter shall use to establish the transmitter driver voltage slew rate. The field value should be treated as a 2's-complement signed value that can range from –128 to +127. If the step requested is out of range of the transmitter capability, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER. A value of 0 should always be accepted, even if the control is not supported;
t) *Control[].bTransmitterReserved*: This field shall be set to 0;
u) *Control[].bTransmitterVendorUnique*: Specifies vendor unique information. Vendor is responsible for positively detecting the validity of the data provided. Should be initialized to 0 when not providing vendor unique information;
v) *Control[].bReceiverFlags*: Specifies flags that define the receiver characteristics or link characteristics. The value may be one or more of the following:
   A) SDI_PHY_ACTIVATE_CONTROL: Specifies that the receiver controls provided should be updated to the current settings. If this flag is not set, then the receiver controls will not be modified during input or are not active during output; or
   B) SDI_PHY_EQUALIZATION_DISABLED: Specifies that any receiver equalization should be disabled during input or is disabled during output;
w) *Control[].bReceiverThreshold*: Contains the step offset from the default setting that the receiver shall use to establish the receiver signal detection threshold. The field value should be treated as a 2's-complement signed value that can range from –128 to +127. If the step requested is out of range of the receiver capability, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER. A value of 0 should always be accepted, even if the control is not supported;
x) *Control[].bReceiverEqualizationGain*: Specifies the step offset from the default setting that the receiver shall use to establish the receiver signal equalization gain. The field value should be treated as a 2's-complement signed value that can range from –128 to +127. If the step requested is out of range of the receiver capability, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER. A value of 0 should always be accepted, even if the control is not supported;
y) *Control[].bReceiverReserved*: This field shall be set to 0;
z) *Control[].bReceiverVendorUnique*: Specifies vendor unique information. Vendor is responsible for positively detecting the validity of the data provided. Should be initialized to 0 when not providing vendor unique information;
aa) *Control[].uPatternFlags*: Specifies flags that define whether the phy should enter a pattern generation mode. The value may be one or more of the following:
   A) SDI_PHY_ACTIVATE_CONTROL: Specifies that the pattern generation mode should be activated. If this flag is not set, then the pattern generation mode is not activited during input or is not active during output. If this flag is set, then the phy will remain in pattern generation mode until

another link reset is initiated. Only a single phy control may have this bit set at any one time. If multiple phy controls have this bit set, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and pattern generation shall be aborted;

B) SDI_PHY_FIXED_PATTERN: Specifies that the fixed pattern should be used in pattern generation. This bit may not be used in conjunction with the SDI_PHY_USER_PATTERN bit. If both are set to one, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and pattern generation shall be aborted;

C) SDI_PHY_USER_PATTERN: Specifies that the user pattern should be used in pattern generation. This bit may not be used in conjunction with the SDI_PHY_FIXED_PATTERN bit. If both are set then the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and pattern generation shall be aborted;

D) SDI_PHY_DISABLE_SCRAMBLING: Specifies that the phy should disable data scrambling during input or data scrambling is disabled during output;

E) SDI_PHY_DISABLE_ALIGN: Specifies that the phy should disable ALIGN and/or NOTIFY insertion during input, or ALIGN and/or NOTIFY insertion is disabled during output; or

F) SDI_PHY DISABLE_SSC: Specifies that the phy should disable spread spectrum clocking during input or spread spectrum clocking is disabled during output;

ab) *Control[].bFixedPattern*: Contains the SAS or SATA specification pattern and may be one of the following:

A) SDI_PHY_CJPAT: specifies that the pattern used should be the CJPAT as defined in the SATA and/or SAS specification; or

B) SDI_PHY_ALIGN: specifies that the pattern used should be the ALIGN[1] repeated value;

ac) *Control[].bUserPatternLength*: Contains the length in bytes of the user pattern buffer (see bUserPattern). The value must be less than the number of elements in the user pattern buffer;

ad) *Control[].UserPatternBuffer[]*: Contains an array of SDI_CHARACTER elements that define the user data pattern. If the user pattern length and SDI_PHY_USER_PATTERN bit are not set then this array should be 0 filled. If the type flags (see bTypeFlags) for the user pattern are not supported or the user pattern is not supported, the driver shall return the *IoctlHeader.ReturnCode* field set to SDI_STATUS_INVALID_PARAMETER and pattern generation shall be aborted;

ae) *Control[].UserPatternBuffer[].bTypeFlags*: Contains flags that define the type of character to generate and may be one or more of the following:

A) SDI_PHY_POSITIVE_DISPARITY: Specifies that the character should have a running disparity that is positive;

B) SDI_PHY_NEGATIVE_DISPARITY: Specifies that the character should have a running disparity that is negative; or

C) SDI_PHY_CONTROL_CHARACTER: Specifies that the character should be encoded as a control character;

and

af) *Control[].UserPatternBuffer[].bValue*: Specifies the base value used to generate the character.

6.18.6 Output

This function shall return a SDI_PHY_CONTROL_BUFFER data structure with the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4;
b) *uFunction*: Same as input;
c) *bPhyIdentifier*: Same as input;
d) *bLengthOfControl*: See input definition;
e) *bNumberOfControls*: See input definition;
f) *uVendorUnique*: Contains vendor unique information;
g) *Control[].bType*: See input definition;
h) *Control[].bRate*: See input definition;
i) *Control[].uVendorUnique*: Contains vendor unique information;
j) *Control[].uTransmitterFlags*: See input definition;
k) *Control[].bTransmitterAmplitude*: See input definition;
l) *Control[].bTransmitterPreemphasis*: See input definition;
m) *Control[].bTransmitterSlewRate*: See input definition;

n) *Control[].bTransmitterReserved*: Same as input;
o) *Control[].bTransmitterVendorUnique*: Contains vendor unique information;
p) *Control[].bReceiverFlags*: See input definition;
q) *Control[].bReceiverThreshold*: See input definition;
r) *Control[].bReceiverEqualizationGain*: See input definition;
s) *Control[].bReceiverReserved*: Same as input;
t) *Control[].bReceiverVendorUnique*: Contains vendor unique information;
u) *Control[].uPatternFlags*: See input definition;
v) *Control[].bFixedPattern*: See input definition;
w) *Control[].bUserPatternLength*: See input definition; and
x) *Control[].bUserPatternBuffer*: See input definition.

6.18.7 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_CHARACTER {
   __u8  bTypeFlags;
   __u8  bValue;
} SDI_CHARACTER, *PSDI_CHARACTER;

typedef struct _SDI_PHY_CONTROL {
   __u8  bType;
   __u8  bRate;
   __u8  bReserved[6];
   __u32 uVendorUnique[8];
   __u32 uTransmitterFlags;
   __i8  bTransmitterAmplitude;
   __i8  bTransmitterPreemphasis;
   __i8  bTransmitterSlewRate;
   __i8  bTransmitterReserved[13];
   __u8  bTransmitterVendorUnique[64];
   __u32 uReceiverFlags;
   __i8  bReceiverThreshold;
   __i8  bReceiverEqualizationGain;
   __i8  bReceiverReserved[14];
   __u8  bReceiverVendorUnique[64];
   __u32 uPatternFlags;
   __u8  bFixedPattern;
   __u8  bUserPatternLength;
   __u8  bPatternReserved[6];
   SDI__SAS_CHARACTER UserPatternBuffer[16];
} SDI_PHY_CONTROL, *PSDI_PHY_CONTROL;

typedef struct _SDI_PHY_CONTROL_BUFFER {
   IOCTL_HEADER IoctlHeader;
   __u32 uFunction;
   __u8  bPhyIdentifier;
   __u16  usLengthOfControls;
   __u8  bNumberOfControls;
   __u8  bReserved[4];
   __u32 uLinkFlags;
   __u8 bSpinupRate;
   __u8 bLinkReserved[7];
   __u32 uVendorUnique[8];
   SDI_PHY_CONTROL Control[1];
} SDI_PHY_CONTROL_BUFFER, *PSDI_PHY_CONTROL_BUFFER;
```

6.19 CC_SDI_GET_CONNECTOR_INFO

6.19.1 Behavior

The CC_SDI_GET_CONNECTOR_INFO SDI function provides a method for obtaining the connector information for a controller.

6.19.2 Input

This function accepts a SDI_GET_CONNECTOR_INFO_BUFFER data structure containing the following fields:

a) *IoctlHeader*: see the IOCTL_HDR definition in 4.2.3, 4.3.2, and 4.4.4; and
b) *Reference[0 - 31]*: Each field shall be set to 0.

6.19.3 Output

This function shall return a SDI_GET_CONNECTOR_INFO_BUFFER data structure with the following fields:

a) *IoctlHeader.ReturnCode*: Return codes are defined in 5.1;
b) *Reference[0 - 31]*: Contains the reference structure for up to 32 phys. The number of valid reference structures corresponds to the number of phys defined in the CC_SDI_GET_PHY_INFO function;
c) *Reference.bConnector*[16]: Contains a null terminated ASCII string that is the reference designator for the component that provides physical connectivity for the phy;
d) *Reference.uPinout*: Contains the pinout identifier for the phy in the connector component and shall be one or move of the following:
   A) SDI_CON_UNKNOWN: indicates that the phy pinout is unknown;
   B) SDI_CON_SFF_8482: indicates that the phy is pinned out as a single lane SFF-8482 connector;
   C) SDI_CON_SFF_8470_LANE_1: indicates that the phy is attached to physical link 0 in a SAS external connector (i.e., an SFF-8470 connector);
   D) SDI_CON_SFF_8470_LANE_2: indicates that the phy is attached to physical link 1 in a SAS external connector;
   E) SDI_CON_SFF_8470_LANE_3: indicates that the phy is attached to physical link 2 in a SAS external connector;
   F) SDI_CON_SFF_8470_LANE_4: indicates that the phy is attached to physical link 3 in a SAS external connector;
   G) SDI_CON_SFF_8484_LANE_1: indicates that the phy is pinned out as lane 1 in a SAS internal wide connector (i.e., an SFF-8484 connector);
   H) SDI_CON_SFF_8484_LANE_2: indicates that the phy is pinned out as lane 2 in a SAS internal wide connector;
   I) SDI_CON_SFF_8484_LANE_3: indicates that the phy is pinned out as lane 3 in a SAS internal wide connector; or
   J) SDI_CON_SFF_8484_LANE_4: indicates that the phy is pinned out as lane 4 in a SAS internal wide connector;
e) *Reference.bLocation*: Contains the location identifier for the connector and will be one or more of the following:
   A) SDI_CON_UNKNOWN: indicates that the connector location is unknown;
   B) SDI_CON_INTERNAL: indicates that the connector is positioned for connecting to devices internal to a system;
   C) SDI_CON_EXTERNAL: indicates that the connector is positioned for connecting to devices external to a system;
   D) SDI_CON_SWITCHABLE: indicates that the phy is switchable between an internal or external connector;
   E) SDI_CON_AUTO: indicates that the phy will auto detect activity on an internal or external connector and switch;
   F) SDI_CON_NOT_PRESENT: indicates that the phy is not physically present, as is the case with a virtual phy; or
   G) SDI_CON_NOT_CONNECTED: indicates that the phy is not physically connected to a connector of any type;

and

f) *Reference.bReserved*: This field shall be set to 0.

6.19.4 Structure Definitions

The following data structures are used:

```
typedef struct _SDI_GET_CONNECTOR_INFO {
   __u32 uPinout;
   __u8  bConnector[16];
   __u8  bLocation;
   __u8  bReserved[15];
} SDI_CONNECTOR_INFO, *PSDI_CONNECTOR_INFO;

typedef struct _SDI_CONNECTOR_INFO_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_CONNECTOR_INFO Reference[32];
} SDI_CONNECTOR_INFO_BUFFER, *PSDI_CONNECTOR_INFO_BUFFER;
```

Annex A
(normative)

Header file

A.1 Header file

This is the C language header file defining all the SDI data structures and constants.

Editor's Note 57: header file lacks NetWare content

Editor's Note 58: need a separate header file for each OS. Linux kernel won't accept Windows & Netware overhead even ifdef'd out.

Editor's Note 59: check use of decimal vs hex numbers in constants

Editor's Note 60: drop typedef struct and just use struct _SDI_XYZ everywhere

```
/*****************************************************************************

Module Name:

   SDI.H


Abstract:

   This file contains constants and data structure definitions used by
drivers
   that support the SAS Driver Interface specification for
   SAS or SATA in either the Windows or Linux.

   This should be considered as a reference implementation only.  Changes may
   be necessary to accommodate a specific build environment or target OS.


*****************************************************************************/

#ifndef _SDI_H_
#define _SDI_H_

// SDI Specification Revision, the intent is that all versions of the
// specification will be backward compatible after the 1.00 release.
// Major revision number, corresponds to xxxx. of SDI specification
// Minor revision number, corresponds to .xxxx of SDI specification
#define SDI_MAJOR_REVISION   0
#define SDI_MINOR_REVISION   82
```

```
/*****************************************************************************/
/* TARGET OS LINUX SPECIFIC CODE                                             */
/*****************************************************************************/

#ifdef _linux

// Linux base types

#include <linux/types.h>

// pack definition

#define SDI_BEGIN_PACK(x)    pack(x)
#define SDI_END_PACK         pack()

// IOCTL Control Codes
// used as the ioctl() request parameter

// Control Codes requiring SDI_ALL_SIGNATURE
#define CC_SDI_GET_DRIVER_INFO    0xCC770001
#define CC_SDI_GET_CNTLR_CONFIG   0xCC770002
#define CC_SDI_GET_CNTLR_STATUS   0xCC770003
#define CC_SDI_FIRMWARE_DOWNLOAD  0xCC770004

// Control Codes requiring SDI_RAID_SIGNATURE
#define CC_SDI_GET_RAID_INFO      0xCC77000A
#define CC_SDI_GET_RAID_CONFIG    0xCC77000B

// Control Codes requiring SDI_SIGNATURE
#define CC_SDI_GET_PHY_INFO       0xCC770014
#define CC_SDI_SET_PHY_INFO       0xCC770015
#define CC_SDI_GET_LINK_ERRORS    0xCC770016
#define CC_SDI_SMP_PASSTHROUGH    0xCC770017
#define CC_SDI_SSP_PASSTHROUGH    0xCC770018
#define CC_SDI_STP_PASSTHROUGH    0xCC770019
#define CC_SDI_GET_SATA_SIGNATURE 0xCC770020
#define CC_SDI_GET_SCSI_ADDRESS   0xCC770021
#define CC_SDI_GET_DEVICE_ADDRESS 0xCC770022
#define CC_SDI_TASK_MANAGEMENT    0xCC770023
#define CC_SDI_GET_CONNECTOR_INFO 0xCC770024

// Control Codes requiring SDI_PHY_SIGNATURE
#define CC_SDI_PHY_CONTROL        0xCC77003C

#pragma SDI_BEGIN_PACK(8)

// IOCTL_HEADER
typedef struct _IOCTL_HEADER {
   __u32 IOControllerNumber;// FIXFIX add a space
__u32 Length;
__u32 ReturnCode;
__u32 Timeout;
__u16 Direction;
} IOCTL_HEADER, *PIOCTL_HEADER;

#pragma SDI_END_PACK
```

```
#endif  # linux

/*****************************************************************************/
/* TARGET OS WINDOWS SPECIFIC CODE                                           */
/*****************************************************************************/

#ifdef _WIN32

// windows IOCTL definitions

#ifndef _NTDDSCSIH_
#include <ntddscsi.h>
#endif

// pack definition

#if defined _MSC_VER
   #define SDI_BEGIN_PACK(x)    pack(push,x)
   #define SDI_END_PACK         pack(pop)
#elif defined __BORLANDC__
   #define SDI_BEGIN_PACK(x)    option -a##x
   #define SDI_END_PACK         option -a.
#else
   #error “SDISAS.H - Must externally define a pack compiler designator.”
#endif

// base types

#define __u8    unsigned char
#define __u32   unsigned long
#define __u16   unsigned short

#define __i8    char

// IOCTL Control Codes
// (IoctlHeader.ControlCode)

// Control Codes requiring SDI_ALL_SIGNATURE
#define CC_SDI_GET_DRIVER_INFO    1
#define CC_SDI_GET_CNTLR_CONFIG   2
#define CC_SDI_GET_CNTLR_STATUS   3
#define CC_SDI_FIRMWARE_DOWNLOAD  4

// Control Codes requiring SDI_RAID_SIGNATURE
#define CC_SDI_GET_RAID_INFO      10
#define CC_SDI_GET_RAID_CONFIG    11

// Control Codes requiring SDI_SIGNATURE
#define CC_SDI_GET_PHY_INFO       20
#define CC_SDI_SET_PHY_INFO       21
#define CC_SDI_GET_LINK_ERRORS    22
#define CC_SDI_SMP_PASSTHROUGH    23
#define CC_SDI_SSP_PASSTHROUGH    24
#define CC_SDI_STP_PASSTHROUGH    25
#define CC_SDI_GET_SATA_SIGNATURE 26
#define CC_SDI_GET_SCSI_ADDRESS   27
#define CC_SDI_GET_DEVICE_ADDRESS 28
```

```
#define CC_SDI_TASK_MANAGEMENT    29
#define CC_SDI_GET_CONNECTOR_INFO 30

// Control Codes requiring SDI_PHY_SIGNATURE
#define CC_SDI_PHY_CONTROL        60

#define IOCTL_HEADER SRB_IO_CONTROL
#define PIOCTL_HEADER PSRB_IO_CONTROL

#endif # _WIN32

/*************************************************************************/
/* TARGET OS NOT DEFINED ERROR                                           */
/*************************************************************************/

#if (!_WIN32 && !_linux)
   #error “Unknown target OS.”
#endif

/*************************************************************************/
/* OS INDEPENDENT CODE                                                   */
/*************************************************************************/

/* * * * * * * * * Class Independent IOCTL Constants * * * * * * * * * */

// Return codes for all IOCTL’s regardless of class
// (IoctlHeader.ReturnCode)

#define SDI_STATUS_SUCCESS              0
#define SDI_STATUS_FAILED               1
#define SDI_STATUS_BAD_CNTL_CODE        2
#define SDI_STATUS_INVALID_PARAMETER    3
#define SDI_STATUS_WRITE_ATTEMPTED      4

// Signature value
// (IoctlHeader.Signature)

#define SDI_ALL_SIGNATURE    “SDIALL”

// Timeout value default of 60 seconds
// (IoctlHeader.Timeout)

#define SDI_ALL_TIMEOUT      60

//  Direction values for data flow on this IOCTL
// (IoctlHeader.Direction, Linux only)
#define SDI_DATA_READ    0
#define SDI_DATA_WRITE   1

// I/O Bus Types
// ISA and EISA bus types are not supported
// (bIoBusType)

#define SDI_BUS_TYPE_PCI       3
#define SDI_BUS_TYPE_PCMCIA    4

// Controller Status
```

```
// (uStatus)

#define SDI_CNTLR_STATUS_GOOD     1
#define SDI_CNTLR_STATUS_FAILED   2
#define SDI_CNTLR_STATUS_OFFLINE  3
#define SDI_CNTLR_STATUS_POWEROFF 4

// Offline Status Reason
// (uOfflineReason)

#define SDI_OFFLINE_REASON_NO_REASON             0
#define SDI_OFFLINE_REASON_INITIALIZING          1
#define SDI_OFFLINE_REASON_BACKSIDE_BUS_DEGRADED 2
#define SDI_OFFLINE_REASON_BACKSIDE_BUS_FAILURE  3

// Controller Class
// (bControllerClass)

#define SDI_CNTLR_CLASS_HBA    5

// Controller Flag bits
// (uControllerFlags)

#define SDI_CNTLR_SAS_HBA   0x00000001
#define SDI_CNTLR_SAS_RAID  0x00000002
#define SDI_CNTLR_SATA_HBA  0x00000004
#define SDI_CNTLR_SATA_RAID 0x00000008

// for firmware download
#define SDI_CNTLR_FWD_SUPPORT  0x00010000
#define SDI_CNTLR_FWD_ONLINE   0x00020000
#define SDI_CNTLR_FWD_SRESET   0x00040000
#define SDI_CNTLR_FWD_HRESET   0x00080000
#define SDI_CNTLR_FWD_RROM     0x00100000

// Download Flag bits
// (uDownloadFlags)
#define SDI_FWD_VALIDATE       0x00000001
#define SDI_FWD_SOFT_RESET     0x00000002
#define SDI_FWD_HARD_RESET     0x00000004

// Firmware Download Status
// (usStatus)
#define SDI_FWD_SUCCESS        0
#define SDI_FWD_FAILED         1
#define SDI_FWD_USING_RROM     2
#define SDI_FWD_REJECT         3
#define SDI_FWD_DOWNREV        4

// Firmware Download Severity
// (usSeverity>
#define SDI_FWD_INFORMATION    0
#define SDI_FWD_WARNING        1
#define SDI_FWD_ERROR          2
#define SDI_FWD_FATAL          3

/* * * * * * * * * SAS RAID Class IOCTL Constants  * * * * * * * * */
```

```
// Return codes for the RAID IOCTL’s regardless of class
// (IoctlHeader.ControlCode)

#define SDI_RAID_SET_OUT_OF_RANGE       1000

// Signature value
// (IoctlHeader.Signature)

#define SDI_RAID_SIGNATURE    “SDIARY”

// Timeout value default of 60 seconds
// (IoctlHeader.Timeout)

#define SDI_RAID_TIMEOUT      60

// RAID Types
// (bRaidType)
#define SDI_RAID_TYPE_NONE     0
#define SDI_RAID_TYPE_0        1
#define SDI_RAID_TYPE_1        2
#define SDI_RAID_TYPE_10       3
#define SDI_RAID_TYPE_5        4
#define SDI_RAID_TYPE_15       5
#define SDI_RAID_TYPE_6        6
#define SDI_RAID_TYPE_OTHER    255

// RAID Status
// (bStatus)
#define SDI_RAID_SET_STATUS_OK          0
#define SDI_RAID_SET_STATUS_DEGRADED    1
#define SDI_RAID_SET_STATUS_REBUILDING  2
#define SDI_RAID_SET_STATUS_FAILED      3

// RAID Drive Status
// (bDriveStatus)
#define SDI_DRIVE_STATUS_OK          0
#define SDI_DRIVE_STATUS_REBUILDING  1
#define SDI_DRIVE_STATUS_FAILED      2
#define SDI_DRIVE_STATUS_DEGRADED    3

// RAID Drive Usage
// (bDriveUsage)
#define SDI_DRIVE_CONFIG_NOT_USED 0
#define SDI_DRIVE_CONFIG_MEMBER   1
#define SDI_DRIVE_CONFIG_SPARE    2

/* * * * * * * * * * SAS HBA Class IOCTL Constants * * * * * * * * * */

// Return codes for SAS IOCTL’s
// (IoctlHeader.ReturnCode)
#define SDI_PHY_INFO_CHANGED            SDI_STATUS_SUCCESS
#define SDI_PHY_INFO_NOT_CHANGEABLE     2000
#define SDI_LINK_RATE_OUT_OF_RANGE      2001

#define SDI_PHY_DOES_NOT_EXIST          2002
#define SDI_PHY_DOES_NOT_MATCH_PORT     2003
```

```
#define SDI_PHY_CANNOT_BE_SELECTED      2004
#define SDI_SELECT_PHY_OR_PORT          2005
#define SDI_PORT_DOES_NOT_EXIST         2006
#define SDI_PORT_CANNOT_BE_SELECTED     2007
#define SDI_CONNECTION_FAILED           2008

#define SDI_NO_SATA_DEVICE              2009
#define SDI_NO_SATA_SIGNATURE           2010
#define SDI_SCSI_EMULATION              2011
#define SDI_NOT_AN_END_DEVICE           2012
#define SDI_NO_SCSI_ADDRESS             2013
#define SDI_NO_DEVICE_ADDRESS           2014

// Signature value
// (IoctlHeader.Signature)
#define SDI_SIGNATURE    “SDISAS”

// Timeout value default of 60 seconds
// (IoctlHeader.Timeout)

#define SDI_TIMEOUT      60

// Device types
// (bDeviceType)
#define SDI_PHY_UNUSED               0x00
#define SDI_NO_DEVICE_ATTACHED       0x00
#define SDI_END_DEVICE               0x10
#define SDI_EDGE_EXPANDER_DEVICE     0x20
#define SDI_FANOUT_EXPANDER_DEVICE   0x30

// Protocol options
// (bInitiatorPortProtocol, bTargetPortProtocol)
#define SDI_PROTOCOL_SATA   0x01
#define SDI_PROTOCOL_SMP    0x02
#define SDI_PROTOCOL_STP    0x04
#define SDI_PROTOCOL_SSP    0x08

// Negotiated and hardware link rates
// (bNegotiatedLinkRate, bMinimumLinkRate, bMaximumLinkRate)
#define SDI_LINK_RATE_UNKNOWN  0x00
#define SDI_PHY_DISABLED       0x01
#define SDI_LINK_RATE_FAILED   0x02
#define SDI_SATA_SPINUP_HOLD   0x03
#define SDI_SATA_PORT_SELECTOR 0x04
#define SDI_LINK_RATE_1_5_GBPS 0x08
#define SDI_LINK_RATE_3_0_GBPS 0x09
#define SDI_LINK_VIRTUAL       0x10

// Discover state
// (bAutoDiscover)
#define SDI_DISCOVER_NOT_SUPPORTED   0x00
#define SDI_DISCOVER_NOT_STARTED     0x01
#define SDI_DISCOVER_IN_PROGRESS     0x02
#define SDI_DISCOVER_COMPLETE        0x03
#define SDI_DISCOVER_ERROR           0x04

// Programmed link rates
```

```
// (bMinimumLinkRate, bMaximumLinkRate)
// (bProgrammedMinimumLinkRate, bProgrammedMaximumLinkRate)
#define SDI_PROGRAMMED_LINK_RATE_UNCHANGED 0x00
#define SDI_PROGRAMMED_LINK_RATE_1_5_GBPS  0x08
#define SDI_PROGRAMMED_LINK_RATE_3_0_GBPS  0x09

// Link rate
// (bNegotiatedLinkRate in SDI_SET_PHY_INFO)
#define SDI_LINK_RATE_NEGOTIATE      0x00
#define SDI_LINK_RATE_PHY_DISABLED   0x01

// Signal class
// (bSignalClass in SDI_SET_PHY_INFO)
#define SDI_SIGNAL_CLASS_UNKNOWN     0x00
#define SDI_SIGNAL_CLASS_DIRECT      0x01
#define SDI_SIGNAL_CLASS_SERVER      0x02
#define SDI_SIGNAL_CLASS_ENCLOSURE   0x03

// Link error reset
// (bResetCounts)
#define SDI_LINK_ERROR_DONT_RESET_COUNTS   0x00
#define SDI_LINK_ERROR_RESET_COUNTS        0x01

// Phy identifier
// (bPhyIdentifier)
#define SDI_USE_PORT_IDENTIFIER   0xFF

// Port identifier
// (bPortIdentifier)
#define SDI_IGNORE_PORT           0xFF

// Programmed link rates
// (bConnectionRate)
#define SDI_LINK_RATE_NEGOTIATED  0x00
#define SDI_LINK_RATE_1_5_GBPS    0x08
#define SDI_LINK_RATE_3_0_GBPS    0x09

// Connection status
// (bConnectionStatus)
#define SDI_OPEN_ACCEPT                          0
#define SDI_OPEN_REJECT_BAD_DESTINATION          1
#define SDI_OPEN_REJECT_RATE_NOT_SUPPORTED       2
#define SDI_OPEN_REJECT_NO_DESTINATION           3
#define SDI_OPEN_REJECT_PATHWAY_BLOCKED          4
#define SDI_OPEN_REJECT_PROTOCOL_NOT_SUPPORTED   5
#define SDI_OPEN_REJECT_RESERVE_ABANDON          6
#define SDI_OPEN_REJECT_RESERVE_CONTINUE         7
#define SDI_OPEN_REJECT_RESERVE_INITIALIZE       8
#define SDI_OPEN_REJECT_RESERVE_STOP             9
#define SDI_OPEN_REJECT_RETRY                    10
#define SDI_OPEN_REJECT_STP_RESOURCES_BUSY       11
#define SDI_OPEN_REJECT_WRONG_DESTINATION        12

// SSP Flags
// (uFlags)
#define SDI_SSP_READ           0x00000001
#define SDI_SSP_WRITE          0x00000002
```

```
#define SDI_SSP_UNSPECIFIED    0x00000004

#define SDI_SSP_TASK_ATTRIBUTE_SIMPLE         0x00000000
#define SDI_SSP_TASK_ATTRIBUTE_HEAD_OF_QUEUE  0x00000010
#define SDI_SSP_TASK_ATTRIBUTE_ORDERED        0x00000020
#define SDI_SSP_TASK_ATTRIBUTE_ACA            0x00000040

// SSP Data present
// (bDataPresent)
#define SDI_SSP_NO_DATA_PRESENT         0x00
#define SDI_SSP_RESPONSE_DATA_PRESENT   0x01
#define SDI_SSP_SENSE_DATA_PRESENT      0x02

// STP Flags
// (uFlags)
#define SDI_STP_READ           0x00000001
#define SDI_STP_WRITE          0x00000002
#define SDI_STP_UNSPECIFIED    0x00000004
#define SDI_STP_PIO            0x00000010
#define SDI_STP_DMA            0x00000020
#define SDI_STP_PACKET         0x00000040
#define SDI_STP_DMA_QUEUED     0x00000080
#define SDI_STP_EXECUTE_DIAG   0x00000100
#define SDI_STP_RESET_DEVICE   0x00000200

// Task Management Flags
// (uFlags)
#define SDI_TASK_IU               0x00000001
#define SDI_HARD_RESET_SEQUENCE   0x00000002
#define SDI_SUPPRESS_RESULT       0x00000004

// Task Management Functions
// (bTaskManagement)
#define SDI_SSP_ABORT_TASK           0x01
#define SDI_SSP_ABORT_TASK_SET       0x02
#define SDI_SSP_CLEAR_TASK_SET       0x04
#define SDI_SSP_LOGICAL_UNIT_RESET   0x08
#define SDI_SSP_CLEAR_ACA            0x40
#define SDI_SSP_QUERY_TASK           0x80

// Task Management Information
// (uInformation)
#define SDI_SSP_TEST           1
#define SDI_SSP_EXCEEDED       2
#define SDI_SSP_DEMAND         3
#define SDI_SSP_TRIGGER        4

// Connector Pinout Information
// (uPinout)
#define SDI_CON_UNKNOWN              0x00000001
#define SDI_CON_SFF_8482             0x00000002
#define SDI_CON_SFF_8470_LANE_1      0x00000100
#define SDI_CON_SFF_8470_LANE_2      0x00000200
#define SDI_CON_SFF_8470_LANE_3      0x00000400
#define SDI_CON_SFF_8470_LANE_4      0x00000800
#define SDI_CON_SFF_8484_LANE_1      0x00010000
#define SDI_CON_SFF_8484_LANE_2      0x00020000
```

```
#define SDI_CON_SFF_8484_LANE_3      0x00040000
#define SDI_CON_SFF_8484_LANE_4      0x00080000

// Connector Location Information
// (bLocation)

// same as uPinout above...
// #define SDI_CON_UNKNOWN              0x01
#define SDI_CON_INTERNAL             0x02
#define SDI_CON_EXTERNAL             0x04
#define SDI_CON_SWITCHABLE           0x08
#define SDI_CON_AUTO                 0x10
#define SDI_CON_NOT_PRESENT          0x20
#define SDI_CON_NOT_CONNECTED        0x80

/* * * * * * * * SAS Phy Control Class IOCTL Constants * * * * * * * * */

// Return codes for SAS Phy Control IOCTL's
// (IoctlHeader.ReturnCode)

// Signature value
// (IoctlHeader.Signature)
#define SDI_PHY_SIGNATURE    "SDIPHY"

// Phy Control Functions
// (bFunction)

// values 0x00 to 0xFF are consistent in definition with the SMP PHY CONTROL
// function defined in the SAS spec
#define SDI_PC_NOP                   0x00000000
#define SDI_PC_LINK_RESET            0x00000001
#define SDI_PC_HARD_RESET            0x00000002
#define SDI_PC_PHY_DISABLE           0x00000003
// 0x04 to 0xFF reserved...
#define SDI_PC_GET_PHY_SETTINGS      0x00000100

// Link Flags
#define SDI_PHY_ACTIVATE_CONTROL     0x00000001
#define SDI_PHY_UPDATE_SPINUP_RATE   0x00000002
#define SDI_PHY_AUTO_COMWAKE         0x00000004

// Device Types for Phy Settings
// (bType)
#define SDI_UNDEFINED 0x00
#define SDI_SATA      0x01
#define SDI_SAS       0x02

// Transmitter Flags
// (uTransmitterFlags)
#define SDI_PHY_PREEMPHASIS_DISABLED    0x00000001

// Receiver Flags
// (uReceiverFlags)
#define SDI_PHY_EQUALIZATION_DISABLED   0x00000001

// Pattern Flags
// (uPatternFlags)
```

```
// #define SDI_PHY_ACTIVATE_CONTROL     0x00000001
#define SDI_PHY_DISABLE_SCRAMBLING      0x00000002
#define SDI_PHY_DISABLE_ALIGN           0x00000004
#define SDI_PHY_DISABLE_SSC             0x00000008

#define SDI_PHY_FIXED_PATTERN           0x00000010
#define SDI_PHY_USER_PATTERN            0x00000020

// Fixed Patterns
// (bFixedPattern)
#define SDI_PHY_CJPAT                   0x00000001
#define SDI_PHY_ALIGN                   0x00000002

// Type Flags
// (bTypeFlags)
#define SDI_PHY_POSITIVE_DISPARITY      0x01
#define SDI_PHY_NEGATIVE_DISPARITY      0x02
#define SDI_PHY_CONTROL_CHARACTER       0x04

// Miscellaneous
#define SLOT_NUMBER_UNKNOWN   0xFFFF

/*****************************************************************************/
/* DATA STRUCTURES                                                           */
/*****************************************************************************/

/* * * * * * * * * * Class Independent Structures * * * * * * * * * */

#pragma SDI_BEGIN_PACK(8)

// CC_SDI_DRIVER_INFO
typedef struct _SDI_DRIVER_INFO {
   __u8  szName[81];
   __u8  szDescription[81];
   __u16 usMajorRevision;
   __u16 usMinorRevision;
   __u16 usBuildRevision;
   __u16 usReleaseRevision;
   __u16 usSDIMajorRevision;
   __u16 usSDIMinorRevision;
} SDI_DRIVER_INFO,  *PSDI_DRIVER_INFO;

typedef struct _SDI_DRIVER_INFO_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_DRIVER_INFO Information;
} SDI_DRIVER_INFO_BUFFER,  *PSDI_DRIVER_INFO_BUFFER;

// CC_SDI_CNTLR_CONFIGURATION
typedef struct _SDI_PCI_BUS_ADDRESS {
   __u8  bBusNumber;
   __u8  bDeviceNumber;
   __u8  bFunctionNumber;
   __u8  bReserved;
} SDI_PCI_BUS_ADDRESS,  *PSDI_PCI_BUS_ADDRESS;

typedef union _SDI_IO_BUS_ADDRESS {
   SDI_PCI_BUS_ADDRESS PciAddress;
```

```
    __u8  bReserved[32];
} SDI_IO_BUS_ADDRESS,  *PSDI_IO_BUS_ADDRESS;

typedef struct _SDI_CNTLR_CONFIG {
    __u32 uBaseIoAddress;
    struct {
        __u32 uLowPart;
        __u32 uHighPart;
    } BaseMemoryAddress;
    __u32 uBoardID;
    __u16 usSlotNumber;
    __u8  bControllerClass;
    __u8  bIoBusType;
    SDI_IO_BUS_ADDRESS BusAddress;
    __u8  szSerialNumber[81];
    __u16 usMajorRevision;
    __u16 usMinorRevision;
    __u16 usBuildRevision;
    __u16 usReleaseRevision;
    __u16 usBIOSMajorRevision;
    __u16 usBIOSMinorRevision;
    __u16 usBIOSBuildRevision;
    __u16 usBIOSReleaseRevision;
    __u32 uControllerFlags;
    __u16 usRromMajorRevision;
    __u16 usRromMinorRevision;
    __u16 usRromBuildRevision;
    __u16 usRromReleaseRevision;
    __u16 usRromBIOSMajorRevision;
    __u16 usRromBIOSMinorRevision;
    __u16 usRromBIOSBuildRevision;
    __u16 usRromBIOSReleaseRevision;
    __u8  bReserved[7];
} SDI_CNTLR_CONFIG,  *PSDI_CNTLR_CONFIG;

typedef struct _SDI_CNTLR_CONFIG_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_CNTLR_CONFIG Configuration;
} SDI_CNTLR_CONFIG_BUFFER,  *PSDI_CNTLR_CONFIG_BUFFER;

// CC_SDI_CNTLR_STATUS
typedef struct _SDI_CNTLR_STATUS {
    __u32 uStatus;
    __u32 uOfflineReason;
    __u8  bReserved[28];
} SDI_CNTLR_STATUS,
  *PSDI_CNTLR_STATUS;

typedef struct _SDI_CNTLR_STATUS_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_CNTLR_STATUS Status;
} SDI_CNTLR_STATUS_BUFFER,  *PSDI_CNTLR_STATUS_BUFFER;

// CC_SDI_FIRMWARE_DOWNLOAD
typedef struct _SDI_FIRMWARE_DOWNLOAD {
    __u32 uBufferLength;
    __u32 uDownloadFlags;
```

```
    __u8  bReserved[32];
    __u16 usStatus;
    __u16 usSeverity;
} SDI_FIRMWARE_DOWNLOAD,  *PSDI_FIRMWARE_DOWNLOAD;

typedef struct _SDI_FIRMWARE_DOWNLOAD_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_FIRMWARE_DOWNLOAD Information;
    __u8  bDataBuffer[1];
} SDI_FIRMWARE_DOWNLOAD_BUFFER,  *PSDI_FIRMWARE_DOWNLOAD_BUFFER;

// CC_SDI_RAID_INFO
typedef struct _SDI_RAID_INFO {
    __u32 uNumRaidSets;
    __u32 uMaxDrivesPerSet;
    __u8  bReserved[92];
} SDI_RAID_INFO,  *PSDI_RAID_INFO;

typedef struct _SDI_RAID_INFO_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_RAID_INFO Information;
} SDI_RAID_INFO_BUFFER,  *PSDI_RAID_INFO_BUFFER;

// CC_SDI_GET_RAID_CONFIG
typedef struct _SDI_RAID_DRIVES {
    __u8  bModel[40];
    __u8  bFirmware[8];
    __u8  bSerialNumber[40];
    __u8  bSASAddress[8];
    __u8  bSASLun[8];
    __u8  bDriveStatus;
    __u8  bDriveUsage;
    __u8  bReserved[30];
} SDI_RAID_DRIVES,   *PSDI_RAID_DRIVES;

typedef struct _SDI_RAID_CONFIG {
    __u32 uRaidSetIndex;
    __u32 uCapacity;
    __u32 uStripeSize;
    __u8  bRaidType;
    __u8  bStatus;
    __u8  bInformation;
    __u8  bDriveCount;
    __u8  bReserved[20];
    SDI_RAID_DRIVES Drives[1];
} SDI_RAID_CONFIG,   *PSDI_RAID_CONFIG;

typedef struct _SDI_RAID_CONFIG_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_RAID_CONFIG Configuration;
} SDI_RAID_CONFIG_BUFFER,  *PSDI_RAID_CONFIG_BUFFER;


/* * * * * * * * * SAS HBA Class Structures * * * * * * * * * */

// CC_SDI_GET_PHY_INFO
typedef struct _SDI_IDENTIFY {
```

```
    __u8  bDeviceType;
    __u8  bRestricted_Byte1;
    __u8  bInitiatorPortProtocol;
    __u8  bTargetPortProtocol;
    __u8  bRestricted_Bytes4to11[8];
    __u8  bSASAddress[8];
    __u8  bPhyIdentifier;
    __u8  bSignalClass;
    __u8  bReserved_Bytes22to27[6];
} SDI_IDENTIFY,  *PSDI_IDENTIFY;

typedef struct _SDI_PHY_ENTITY {
    SDI_IDENTIFY Identify;
    __u8  bPortIdentifier;
    __u8  bNegotiatedLinkRate;
    __u8  bMinimumLinkRate;
    __u8  bMaximumLinkRate;
    __u8  bPhyChangeCount;
    __u8  bAutoDiscover;
    __u8  bReserved[2];
    SDI_IDENTIFY Attached;
} SDI_PHY_ENTITY,  *PSDI_PHY_ENTITY;

typedef struct _SDI_PHY_INFO {
    __u8  bNumberOfPhys;
    __u8  bReserved[3];
    SDI_PHY_ENTITY Phy[32];
} SDI_PHY_INFO,  *PSDI_PHY_INFO;

typedef struct _SDI_PHY_INFO_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_PHY_INFO Information;
} SDI_PHY_INFO_BUFFER,  *PSDI_PHY_INFO_BUFFER;

// CC_SDI_SET_PHY_INFO
typedef struct _SDI_SET_PHY_INFO {
    __u8  bPhyIdentifier;
    __u8  bNegotiatedLinkRate;
    __u8  bProgrammedMinimumLinkRate;
    __u8  bProgrammedMaximumLinkRate;
    __u8  bSignalClass;
    __u8  bReserved[3];
} SDI_SET_PHY_INFO,  *PSDI_SET_PHY_INFO;

typedef struct _SDI_SET_PHY_INFO_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_SET_PHY_INFO Information;
} SDI_SET_PHY_INFO_BUFFER,  *PSDI_SET_PHY_INFO_BUFFER;

// CC_SDI_GET_LINK_ERRORS
typedef struct _SDI_LINK_ERRORS {
    __u8  bPhyIdentifier;
    __u8  bResetCounts;
    __u8  bReserved[2];
    __u32 uInvalidDwordCount;
    __u32 uRunningDisparityErrorCount;
    __u32 uLossOfDwordSyncCount;
```

```
    __u32 uPhyResetProblemCount;
} SDI_LINK_ERRORS,  *PSDI_LINK_ERRORS;

typedef struct _SDI_LINK_ERRORS_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_LINK_ERRORS Information;
} SDI_LINK_ERRORS_BUFFER,  *PSDI_LINK_ERRORS_BUFFER;

// CC_SDI_SMP_PASSTHROUGH
typedef struct _SDI_SMP_REQUEST {
    __u8  bFrameType;
    __u8  bFunction;
    __u8  bReserved[2];
    __u8  bAdditionalRequestBytes[1016];
} SDI_SMP_REQUEST,  *PSDI_SMP_REQUEST;

typedef struct _SDI_SMP_RESPONSE {
    __u8  bFrameType;
    __u8  bFunction;
    __u8  bFunctionResult;
    __u8  bReserved;
    __u8  bAdditionalResponseBytes[1016];
} SDI_SMP_RESPONSE,  *PSDI_SMP_RESPONSE;

typedef struct _SDI_SMP_PASSTHROUGH {
    __u8  bPhyIdentifier;
    __u8  bPortIdentifier;
    __u8  bConnectionRate;
    __u8  bReserved;
    __u8  bDestinationSASAddress[8];
    __u32 uRequestLength;
    SDI_SMP_REQUEST Request;
    __u8  bConnectionStatus;
    __u8  bReserved2[3];
    __u32 uResponseBytes;
    SDI_SMP_RESPONSE Response;
} SDI_SMP_PASSTHROUGH,  *PSDI_SMP_PASSTHROUGH;

typedef struct _SDI_SMP_PASSTHROUGH_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_SMP_PASSTHROUGH Parameters;
} SDI_SMP_PASSTHROUGH_BUFFER,  *PSDI_SMP_PASSTHROUGH_BUFFER;

// CC_SDI_SSP_PASSTHROUGH
typedef struct _SDI_SSP_PASSTHROUGH {
    __u8  bPhyIdentifier;
    __u8  bPortIdentifier;
    __u8  bConnectionRate;
    __u8  bReserved;
    __u8  bDestinationSASAddress[8];
    __u8  bLun[8];
    __u8  bCDBLength;
    __u8  bAdditionalCDBLength;
    __u8  bReserved2[2];
    __u8  bCDB[16];
    __u32 uFlags;
    __u8  bAdditionalCDB[24];
```

```
    __u32 uDataLength;
} SDI_SSP_PASSTHROUGH,  *PSDI_SSP_PASSTHROUGH;

typedef struct _SDI_SSP_PASSTHROUGH_STATUS {
    __u8  bConnectionStatus;
    __u8  bReserved[3];
    __u8  bDataPresent;
    __u8  bStatus;
    __u8  bResponseLength[2];
    __u8  bResponse[256];
} SDI_SSP_PASSTHROUGH_STATUS,  *PSDI_SSP_PASSTHROUGH_STATUS;

typedef struct _SDI_SSP_PASSTHROUGH_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_SSP_PASSTHROUGH Parameters;
    SDI_SSP_PASSTHROUGH_STATUS Status;
    __u32 uDataBytes;
    __u8  bDataBuffer[1];
} SDI_SSP_PASSTHROUGH_BUFFER,  *PSDI_SSP_PASSTHROUGH_BUFFER;

// CC_SDI_STP_PASSTHROUGH
typedef struct _SDI_STP_PASSTHROUGH {
    __u8  bPhyIdentifier;
    __u8  bPortIdentifier;
    __u8  bConnectionRate;
    __u8  bReserved;
    __u8  bDestinationSASAddress[8];
    __u8  bReserved2[4];
    __u8  bCommandFIS[20];
    __u32 uFlags;
    __u32 uDataLength;
} SDI_STP_PASSTHROUGH,  *PSDI_STP_PASSTHROUGH;

typedef struct _SDI_STP_PASSTHROUGH_STATUS {
    __u8  bConnectionStatus;
    __u8  bReserved[3];
    __u8  bStatusFIS[20];
    __u32 uSCR[16];
} SDI_STP_PASSTHROUGH_STATUS,  *PSDI_STP_PASSTHROUGH_STATUS;

typedef struct _SDI_STP_PASSTHROUGH_BUFFER {
    IOCTL_HEADER IoctlHeader;
    SDI_STP_PASSTHROUGH Parameters;
    SDI_STP_PASSTHROUGH_STATUS Status;
    __u32 uDataBytes;
    __u8  bDataBuffer[1];
} SDI_STP_PASSTHROUGH_BUFFER,  *PSDI_STP_PASSTHROUGH_BUFFER;

// CC_SDI_GET_SATA_SIGNATURE
typedef struct _SDI_SATA_SIGNATURE {
    __u8  bPhyIdentifier;
    __u8  bReserved[3];
    __u8  bSignatureFIS[20];
} SDI_SATA_SIGNATURE,  *PSDI_SATA_SIGNATURE;

typedef struct _SDI_SATA_SIGNATURE_BUFFER {
    IOCTL_HEADER IoctlHeader;
```

```
   SDI_SATA_SIGNATURE Signature;
} SDI_SATA_SIGNATURE_BUFFER,  *PSDI_SATA_SIGNATURE_BUFFER;

// CC_SDI_GET_SCSI_ADDRESS
typedef struct _SDI_GET_SCSI_ADDRESS_BUFFER {
   IOCTL_HEADER IoctlHeader;
   __u8  bSASAddress[8];
   __u8  bSASLun[8];
   __u8  bHostIndex;
   __u8  bPathId;
   __u8  bTargetId;
   __u8  bLun;
} SDI_GET_SCSI_ADDRESS_BUFFER,   *PSDI_GET_SCSI_ADDRESS_BUFFER;

// CC_SDI_GET_DEVICE_ADDRESS
typedef struct _SDI_GET_DEVICE_ADDRESS_BUFFER {
   IOCTL_HEADER IoctlHeader;
   __u8  bHostIndex;
   __u8  bPathId;
   __u8  bTargetId;
   __u8  bLun;
   __u8  bSASAddress[8];
   __u8  bSASLun[8];
} SDI_GET_DEVICE_ADDRESS_BUFFER,  *PSDI_GET_DEVICE_ADDRESS_BUFFER;

// CC_SDI_TASK_MANAGEMENT
typedef struct _SDI_SSP_TASK_IU {
   __u8  bHostIndex;
   __u8  bPathId;
   __u8  bTargetId;
   __u8  bLun;
   __u32 uFlags;
   __u32 uQueueTag;
   __u32 uReserved;
   __u8  bTaskManagementFunction;
   __u8  bReserved[7];
   __u32 uInformation;
} SDI_SSP_TASK_IU,  *PSDI_SSP_TASK_IU;

typedef struct _SDI_SSP_TASK_IU_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_SSP_TASK_IU Parameters;
   SDI_SSP_PASSTHROUGH_STATUS Status;
} SDI_SSP_TASK_IU_BUFFER,  *PSDI_SSP_TASK_IU_BUFFER;

// CC_SDI_GET_CONNECTOR_INFO

typedef struct _SDI_GET_CONNECTOR_INFO {
   __u32 uPinout;
   __u8  bConnector[16];
   __u8  bLocation;
   __u8  bReserved[15];
} SDI_CONNECTOR_INFO,  *PSDI_CONNECTOR_INFO;

typedef struct _SDI_CONNECTOR_INFO_BUFFER {
   IOCTL_HEADER IoctlHeader;
   SDI_CONNECTOR_INFO Reference[32];
```

```
} SDI_CONNECTOR_INFO_BUFFER,  *PSDI_CONNECTOR_INFO_BUFFER;

// CC_SDI_PHY_CONTROL
typedef struct _SDI_CHARACTER {
   __u8  bTypeFlags;
   __u8  bValue;
} SDI_CHARACTER,  *PSDI_CHARACTER;

typedef struct _SDI_PHY_CONTROL {
   __u8  bType;
   __u8  bRate;
   __u8  bReserved[6];
   __u32 uVendorUnique[8];
   __u32 uTransmitterFlags;
   __i8  bTransmitterAmplitude;
   __i8  bTransmitterPreemphasis;
   __i8  bTransmitterSlewRate;
   __i8  bTransmitterReserved[13];
   __u8  bTransmitterVendorUnique[64];
   __u32 uReceiverFlags;
   __i8  bReceiverThreshold;
   __i8  bReceiverEqualizationGain;
   __i8  bReceiverReserved[14];
   __u8  bReceiverVendorUnique[64];
   __u32 uPatternFlags;
   __u8  bFixedPattern;
   __u8  bUserPatternLength;
   __u8  bPatternReserved[6];
   SDI_CHARACTER UserPatternBuffer[16];
} SDI_PHY_CONTROL,  *PSDI_PHY_CONTROL;

typedef struct _SDI_PHY_CONTROL_BUFFER {
   IOCTL_HEADER IoctlHeader;
   __u32 uFunction;
   __u8  bPhyIdentifier;
   __u16 usLengthOfControl;
   __u8  bNumberOfControls;
   __u8  bReserved[4];
   __u32 uLinkFlags;
   __u8  bSpinupRate;
   __u8  bLinkReserved[7];
   __u32 uVendorUnique[8];
   SDI_PHY_CONTROL Control[1];
} SDI_PHY_CONTROL_BUFFER,  *PSDI_PHY_CONTROL_BUFFER;

#pragma SDI_END_PACK

#endif // _SDI_H_
```