AT&T

UNIX® SYSTEM V/386 RELEASE 4

Programmer's Guide: SCSI Driver Interface

UNIX Software Operation

AT&T

UNIX® SYSTEM V/386 RELEASE 4

Programmer's Guide: SCSI Driver Interface

UNIX Software Operation

Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-957556-1

UNIX
PRESS
A Prentice Hall Title

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Figures and Tables

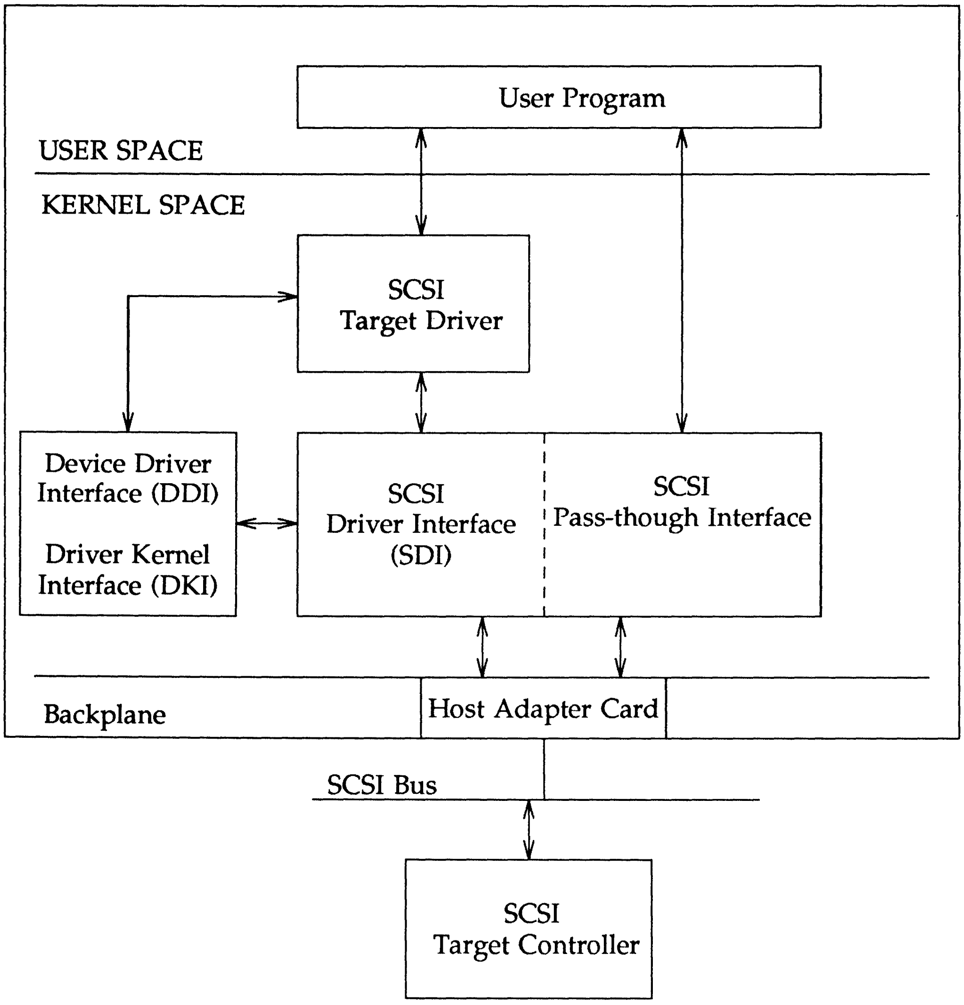
1 Introduction

Overview

SCSI Driver Interface

The *Programmer's Guide: SCSI Driver Interface (SDI)* is a machine-independent mechanism for writing SCSI target drivers to access a SCSI device. Figure 1-1 illustrates the SCSI hardware and software.

Figure 1-1: SCSI Software Architecture



SDI interacts with the SCSI hardware and provides a driver or user program a way in which to access a SCSI device.

Relationship of SDI and the Device Driver Interface

A SCSI target driver is written using the functions of the Device Driver Interface (DDI). DDI is a library of functions. The DDI functions are incorporated in the system software as part of UNIX® System V Release 4.

SDI augments the DDI functions to provide a set of services for exchanging commands and data with the SCSI device and for allocating and freeing data structures. SDI includes input/output controls that give user programs direct command access to a SCSI device and to the SDI. These controls allow circumvention of the target driver for user testing of SCSI devices, state determination, and diagnostics. The SDI functions are incorporated in the "SCSI Support Package" you receive with your SCSI host adapter feature card for the 386 computer.

A properly written target driver using DDI and SDI functions can be ported directly between any 386 family member equipped with SCSI. Driver portability is possible when strict compliance to the SDI and DDI functions and structures is maintained. DDI has a wide variety of features that offer a driver developer an opportunity to write drivers that are faster to code, usable on a variety of computers, and easier to maintain.

SCSI consists of both hardware and software. SCSI hardware includes a special feature card called the host adapter, and a cable for connecting SCSI devices called the SCSI bus. The host adapter provides a communication pipeline between a target driver and the SCSI bus. The host adapter connects a AT&T 386 computer to the machine-independent SCSI bus.

Using This Manual

The audience for this manual is driver developers wishing to write or maintain a SCSI target driver.

The following is an overview of each chapter in this manual:

Chapter 1 — Introduction
An introduction to this manual. Also described are special programming considerations.

Chapter 2 — SDI Input/Output Controls
a description of the special SDI input/output controls.

Chapter 3 — Functions
the SDI functions that are used to code a driver.

Chapter 4 — Structures
the data structures used when creating a driver and for use when sending a command with the input/output controls.

Index a topical index.

Name Referencing Conventions

A reference numbering scheme is used throughout this manual and in the other books in the Open Architecture documentation series. On driver routine names, structure names, and function names, a code is suffixed that describes the origin of each name. The code is in the format:

(D*ni*)

Where:

D =	Driver interface designator
n =	Name type: 2 = Driver Routine, 3 = Function, 4 = Structure
i =	(optional) interface type: P = DDI, I = SDI, X = Block and Character Interface (if omitted, the reference is to a driver routine).

For example, sdi_send(D3I) is a SDI function, ver_no(D4P) is a PDI structure.

Type Style Conventions

The conventions given in Table 1-1 are used throughout the text.

Table 1-1: Conventions Used In This Book

Item	Type Style	Example
Book Titles	*Italics*	*Programmer's Guide*
C bitwise operators (|&~)	FULL CAPITAL LETTERS	OR
C `typedef` declarations	`Constant Width`	`dma_tuple`
Driver prefix	*Italics*	*prefix*`close`(D1)
Driver routines	`Constant Width`	`strategy`(D1) routine
Error values	FULL CAPITAL LETTERS	EINTR
File names	`Constant Width`	`/usr/include/sys/ddi.h`
Flag names	FULL CAPITAL LETTERS	B_WRITE
Functions	`Constant Width`	`sdi_send`(D2I)
Function arguments	*Italics*	*bp*
Keyboard keys	Key	CTRL-d
Structure members	`Constant Width`	`b_flags`
Structure names	`Constant width`	`ver_no`(D2I) structure
Symbolic constants	FULL CAPITAL LETTERS	CE_CONT
UNIX system C commands	`Constant Width` with section reference	`ioctl`(2)

Programming Considerations

This section provides information on how to code a SCSI target driver. Described are the differences between coding driver routines when using the SDI or when using the DDI.

SDI Driver Routine Considerations

Routines are called from a number of sources: the kernel through the bdevsw and cdevsw, at initialization time and at startup, and by other routines. The driver routines written for a SCSI target driver have the same attributes as those of any other block driver with only two exceptions, you must include an `ioctl`(D2) routine, and you should not include an interrupt routine that is called by the kernel. Since the SCSI peripherals are not directly connected to the backplane, the normal interrupt routine (`intr`) would never get called and therefore would be useless.

A complete description of driver routines is provided in Chapter 2 of the *Device Driver Interface/Driver-Kernel Interface (DDI/DKI) Reference Manual.*

Table 1-2 summarizes the driver routines that may be used to code a SCSI target driver.

Table 1-2: Target Driver Routine Summary

Routine Name	Description
prefix`close` (*device-number, flag, otype,cred-pointer*)	Close a device
prefix`init` ()	Initialize a device
prefix`ioctl` (*device-number, command, arg, mode, cred-pionter, rval-pointer*)	I/O control
prefix`open` (*device-number, flag, otype,cred-pointer*)	Open a device
prefix`print` (*device-number, string*)	Display error
prefix`read` (*device-number, uio-pointer,cred-pointer*)	Read data
prefix`start` ()	Start device access
prefix`strategy` (*buf-pointer*)	Block device I/O
prefix`write` (*device-number,uio-pointer, cred-pointer*)	Write data

Interrupts

The interrupt routines of the SCSI target drivers are different from those of typical hardware drivers because the hardware they control is not directly connected to the I/O bus on the 6386 computer. Hardware drivers using non-SCSI interfaces with associated hardware connected directly to the I/O bus of a 6386 computer have interrupt routines that are called by the operating system.

The interrupt routine in a SCSI target driver is called by the host adapter software with a pointer to an sb(D4I) structure as the input argument. The host adapter controls hardware on the 6386 computer SCSI Bus.

Before sending the job, assign a value to the `sc_int` member of the scb(D4I) structure. When a target driver completes, the driver passes the address of the SCSI interrupt routine to the host adapter in the sb(D4I) structure (specifically using `sc_int` member structure). Target drivers may define different interrupt routines for different operations.

ioctl

An SCSI target driver must provide an `ioctl` routine with handling for the B_GETDEV and B_GETTYPE pass-through interface commands. The following code provides an example of how these two commands might be handled:

```
#include "sys/types.h"
#include "sys/vtoc.h"
#include "sys/sdi.h"
#include "sys/scsi.h"
#include "sys/errno.h"
#include "sys/open.h"
#include "sys/sdi_edt.h"

   dev_t ipt_dev;
      ...
   switch(ioctl-cmd);
      ...
         case B_GETTYPE:
            if (copyout("scsi",
               ((struct bus_type *) arg)->bus_name, 5))
            {
               return(EFAULT);
            }
            if (copyout("driver-prefix",
               ((struct bus_type *) arg)->drv_name, 5))
            {
               return(EFAULT);
            }
            return(0);
         case B_GETDEV;
            sdi_getdev(&dk->dk_addr, &ipt_dev);
            if (copyout(&ipt_dev, arg, sizeof(ipt_dev)))
            {
               return(EFAULT);
            }
            return(0);
```

The bus_type structure is defined in /usr/include/sys/sdi_edt.h. In this example, "5" is for "scsi" and a terminating NULL and assumes a four-character driver prefix and a terminating NULL character.

SCSI Bus Reset

The SCSI bus can reset in the case of a software or hardware failure. (Reset means that all jobs using the SCSI bus are interrupted and returned without being completed with the sc_comp_code member of the scb structure set to SDI_RESET.) After a reset, SCSI target controllers may take up to 5 seconds to reset themselves depending on the peripheral device. During this interval, all SCSI jobs are blocked and not sent over the SCSI bus. Once the times has elapsed, job processing on the bus is continued. Computers equipped with differential multi-host functionality (presently unsupported on 6386) are given an additional second to send immediate commands to regain device control (SCSI RESERVE command). A target driver regains control so that pending job requests can complete without intervention from other hosts on the bus. Should the driver disregard this opportunity, a target driver on another host could access the device and reserve it for its use.

Suggested Reading

You must obtain the *AT&T SCSI (Small Computer System Interface) Definition, Select Code 305-013*. This manual is required when sending commands to a SCSI device (using either the pass-through interface or the `sdi_send` or `sdi_icmd` functions described in this manual.

2 SDI Input/Output Controls

Introduction

SDI provides a set of Input/Output (I/O) controls that can be used from a user program. These controls permit a program to:

- Get information from a target driver or from the host adapter.
- Reset a SCSI bus or a target controller.
- Send a command to a SCSI device. This control is also called the pass-through interface.

Most of the controls are built into SDI, but some must be provided by a SCSI target driver.

This chapter describes information about using the `ioctl`(2) system call with the SDI I/O controls. Following this section is a separate section on each set of controls.

In Table 2-1, the "Purpose" describes what the SDI I/O controls perform. The "Control" is the `ioctl`(2) control name. The "Source" column describes from where the information is provided. (For example, if the source is TD, then a target driver must provide the information in its `ioctl`(D2) routine.) The "Computer" indicates on which computer the control can be accessed. The "Header File" indicates the name of the header file that defines the I/O control. All files are in the `/usr/include/sys` directory.

Table 2-1: SDI I/O Controls

Purpose	Control	Source	Header File
Get Information	B_GETDEV	TD †	**sdi_edt.h**
	B_GETTYPE	HA	**sdi_edt.h**
	B_GETTYPE	TD	**sdi_edt.h**
	GETVER	HA	**scsi.h**
	HA_VER	HA	**sdi.h**
Reset bus or target controller	SDI_BRESET	HA	**sdi.h**
	SDI_TRESET	HA	**sdi.h**
Send a command	SDI_SEND	HA	**sdi.h**

NOTE

Abbreviations:

TD = SCSI Target Driver HA = SCSI Host Adapter

NOTE

A program must have super-user permissions to send an SDI I/O control.

The format for the `ioctl`(2) command is:

`ioctl` (*file-descriptor, control-name, argument*)

Where:

file-descriptor	is generated by an open(2) call to the SCSI host adapter device file.
control-name	the name of the SDI I/O control.
argument	an optional argument required by the I/O control.

Table 2-2 indicates the second and third arguments to the `ioctl` system call, and the origin of the first argument. The first argument to the `ioctl`(2) command is always a file descriptor passed from a previous open(2) call. The path (first) argument to the open(2) call defines the origins of the `ioctl` file descriptor.

Table 2-2: SDI ioctl Arguments

First Argument *origin*	**Second Argument** *control-name*	**Third Argument**
GEN	HA_VER	struct `ver_no` * pointer
GEN	SDI_BRESET	long `ddi_dev_t` — minor device number of a device on the SCSI bus
PT	SDI_SEND	struct sb * pointer
PT	SDI_TRESET	none

The meaning of the abbreviations in Table 2-2 are:

BRESET	Bus Reset
GEN	device number for a specific device. This is the major number for the host adapter with the 0xff minor number.
HA	Host Adapter
PT	Pass-Through device number for a specific device.
TRESET	Target device controller Reset
VER	Version number

The Host Adapter Minor Number

Table 2-3 describes the minor number format for the host adapter driver I/O controls:

Table 2-3: Host Adapter Minor Number

CONTENTS:	0	Controller				Logical Unit		
BITS:	7	6	5	4	3	2	1	0

This controller addressing is used rather than the address provided in the SCSI Control Block. Requests for pass-through or controller reset are rejected if addressed to the host adapter. The minor number is only used by the host adapter for the SDI_SEND and SDI_TRESET controls.

The minor number 0xff defaults to the host adapter driver and should be used when using general controls such as those for getting information.

Get Information

These I/O controls provide a variety of information sources that can be obtained through the `ioctl`(2) commands. This information is useful for initiating other I/O controls and for verifying the usability of the current software or hardware versions.

This information includes the:

- Bus type and driver type (using the B_GETTYPE command) of either the host adapter or of the target driver
- Target driver device type (using the B_GETDEV command)
- The host adapter version number (using the GETVER command)
- Contents of the ver_no structure (using the HA_VER command). This structure contains the host adapter release number, the computer type, and the release number of the SCSI software.

The procedure for accessing information is:

- Obtain a file descriptor by executing an open(2) system call on the host adapter or target driver special device file.
- Send the command to either the host adapter or to the target driver using the `ioctl`(2) command.

ERRORS

The following error can be returned by SDI in `errno` (errors returned by the target driver are dependent on the design of the driver). B_GETTYPE and HA_VER return EFAULT if the information cannot be copied to user space.

GETVER does not return an error.

EXAMPLE

The following example illustrates the use of a get information I/O control.

```
char *special_device;             /* Path name of target device */
int file_des;                     /* File descriptor */
dev_t pass_thru_device;           /* Pass through device number */


/*
 *  Open the special device file so that we may query the target
 *  driver for the pass-through major minor numbers of the device.
 */
if ((file_des = open(special_device, O_RDONLY)) < 0)
{
        fprintf(stderr,"Special device file open failed0);
}

/*
 *  Perform the B_GETDEV command to obtain the pass-through
 *  device number for this particular device.
 */
if (ioctl(file_des, B_GETDEV, &pass_thru_device) < 0)
{
fprintf(stderr,"Call to get pass-through device number failed0);
}
```

Reset a Bus or Target Controller

The SDI I/O controls provide a command to reset a SCSI bus and a command to reset a target controller.

The method for executing these commands is:

1. Use B_GETDEV to determine the major and minor numbers of the host adapter handling this device.
2. Open the device special file to obtain the host adapter major and minor numbers.
3. Use mktemp(3C) to generate a unique host adapter node name, then append this new name to the end of the pathname provided with the device name.
4. Use mknod(2) to create the general access host adapter node so that the pass-through interface can be used. NOTE: On SDI_BRESET, when you use makedev to form the device number (as the last argument of mknod) use 0xff instead on a minor number. On SDI_TRESET, use the pass-through minor number.
5. Open the host adapter node for access by the pass-through interface.
6. Reset the bus or the target controller.

ERRORS

SDI_BRESET does not return any value in errno or in SCB.sc_comp_code.

SDI_TRESET returns EBUSY (jobs are outstanding at the device or the device is not active) in errno. In addition, SCB.sc_comp_code may be set to SDI_PROGRES (job not complete) during the time when the sb is being sent to SDI.

EXAMPLE

```
#define HATEMP      "HAXXXXXX"

  char *special_device;       /* Path name of target device */
  char *pass_through[];       /* Path name of pass-through device */
  int file_des;               /* File descriptor */
  dev_t pass_thru_device;     /* Pass-through device number */
```

```
/*
 *  Open the special device file so that we may query the target
 *  driver for the pass-through major/minor numbers of the device.
 */
if ((file_des = open(special_device, O_RDONLY)) < 0)
{
        fprintf(stderr,"Special device file open failed\n");
}

/*
 *  Perform the B_GETDEV command to obtain the pass-through
 *  device number for this particular device.
 */
if (ioctl(file_des, B_GETDEV, &pass_thru_device) < 0)
{
fprintf(stderr,"Call to get pass-through device number failed\n");
}

(void) close(file_des);

/*
 *  Generate the pass-through special device file for
 *  this device using mktemp().  To create the node in
 *  the same directory as the target special device file,
 *  the new pass-through name is appended on to the
 *  path name of the target special device file.
 */

/* Copy the special device file name into the pass-through name */
(void) strcpy(pass_through, special_device);

/*  Search to the last '/'.  Append the newly created pass-through
 *  device file onto the end.  If no '/' was found, then use the
 *  current directory.
 */
if ((ptr = strrchr(pass_through, '/')) != NULL)
{
        (void) strcpy(++ptr, mktemp(HATEMP));
}
else
{
        (void) strcpy(pass_through, mktemp(HATEMP));
}
```

```
/*
 *  Make the pass-through device node using the pass-through
 *  major number of the target device and the general use
 *  pass-through minor number (0xff).
 */
if (mknod(pass_through, (S_IFCHR | S_IREAD | S_IWRITE),
          makedev(pass_thru_device.maj,0xff)) < 0)
{
        fprintf(stderr,"Unable to make pass-through node\n");
}

/*
 *  Now that the pass-through node has been created, perform
 *  an open it so that the command can be issued.
 */
if ((file_des = open(pass_through, O_RDWR)) < 0)
{
        fprintf(stderr,"Open of pass-through device failed\n");

}

/*  Issue the bus reset ioctl to the Host Adapter driver.
 *  Pass it the pass-through minor number as an argument
 *  so it can determine which bus to reset.
 */
if( ioctl(file_des, SDI_BRESET, pass_thru_device.min) < 0 )
{
        fprintf(stderr,"SCSI bus reset failed\n");
}
```

Send a SCSI Command (Pass-Through Interface)

The SCSI command pass-through interface gives a user program direct access to a SCSI device. By permitting user programs to act in a manner similar to a target driver, the overhead of instructions needed to make device-specific requests can be removed from a target driver. An example of device-specific requests are the instructions required to format a disk. With the pass-through interface, a user program can select different instruction packages for different vendors' drives. Not only can many instructions be removed from a driver, but a driver can be made to work on a wider range of different drives. In addition, a driver need not be updated as frequently when a device changes.

The pass-through interface gives you a means of evaluating new peripherals and controllers without developing a driver, checking device states, and eliminating duplication of driver code.

Pass-Through Use Considerations

CAUTION When a user program opens access to the pass-through interface, all other jobs heading for the device using a target driver are blocked until the pass-through access is closed. This means any use of `read`(2) or `write`(2) during pass-through access fails. In addition, a process using pass-through cannot use system calls that require target driver access. If a process opens the pass-through interface and then executes a system call that accesses the device, the system call fails. When the system call generates an SDI_SEND, the process will hang. (Some `ioctl` commands may not generate an SDI_SEND.) If your pass-through interface access requires device reads and writes, you must do so either before pass-through is started or with CDB commands utilizing SDI_SEND (explained later in this chapter).

The following must be observed when writing a program that accesses the pass-through interface:

- One major-minor number pair is associated with each logical unit
- Permissions on the special device files must be set so that only the owner has read-write permissions

- Only one process may have a special device file open. Processing of jobs from the target driver to the logical unit is suspended while the device file is open. For the 6386 computer, this is identical to the suspension which takes place after a check condition.
- The only `sb_type` value which can be used is ISCB_TYPE. (`sb_type` is a member of the sb structure.)

NOTE The pass-through interface requires that a target driver include access to the two `ioctl` commands, B_GETDEV and B_GETTYPE. Implementation of these two commands for a target driver is discussed in Chapter 1 of this manual.

The pass-through interface provides a command for sending a SCSI command directly to a SCSI device. Any command described in the *AT&T SCSI (Small Computer System Interface) Definition, Select Code 305-013* can be sent directly to the SCSI device. A sample of the functionality provided by these commands is:

- Data copying
- Device formatting
- Mode selecting
- Preventing/Allowing media removal
- Reserve/Release unit
- Read/Write data

Commands are sent by creating a Command Descriptor Block (defined by the scs and scm structures and described in the *ANSI Small Computer System Interface (SCSI), X3T9, 2/82-2, Revision 17B*. Refer to the discussion of the scs and scm structures in Chapter 4 of this manual for more information.

The method for using the SDI_SEND command is:

1. Open target driver
2. Use B_GETDEV to obtain the pass-through host adapter device number
3. Close access to the target driver

4. Use mknod(2) to create a host adapter node
5. Open the pass-through interface
6. Create sb and CDB
7. Send the command to the SCSI device

ERRORS

SDI_SEND can return the following errors in `errno`:

- EBUSY — jobs are queued for the device (therefore, the device cannot be accessed).
- EFAULT — an attempt to copy an sb or an scb to or from a user program failed.
- EINVAL — `sb.sb_type` is not set to ISCB_TYPE or `SCB.sc_mode` is set to the unsupported SCB_LINK value.
- ENOMEM — memory cannot be allocated for data from the sent CDB or for returned data.

SDI_SEND can return the following values in `SCB.sc_comp_code`:

- SDI_HAERR (host adapter failure or parity error)
- SDI_PROGRES (the job is not complete during time when sb is being sent to the host)

EXAMPLE

```
#define HATEMP  "HAXXXXXX"

  char *special_device;      /* Path name of target device */
  char *pass_through[];      /* Path name of pass-through device */
  int file_des;              /* File descriptor */
  dev_t pass_thru_device;    /* Pass-through device number */
  struct sb sb, *sb_ptr;     /* SCSI block and pointer */
  struct scs scs;            /* SCSI command block */
  char buffer[512];          /* Data buffer area */
```

```
/*
 *  Open the special device file so that we may query the target
 *  driver for the pass-through major minor numbers of the device.
 */
if ((file_des = open(special_device, O_RDONLY)) < 0)
{
        fprintf(stderr,"Special device file open failed\n");
}

/*
 *  Perform the B_GETDEV command to obtain the pass-through
 *  device number for this particular device.
 */
if (ioctl(file_des, B_GETDEV, &pass_thru_device) < 0)
{
fprintf(stderr,"Call to get pass-through device number failed\n");
}

(void) close(file_des);

/*
 *  Generate the pass-through special device file for
 *  this device using mktemp().  To create the node in
 *  the same directory as the target special device file,
 *  the new pass-through name will be appended on to the
 *  path name of the target special device file.
 */

/* Copy the special device file name into the pass-through name */
(void) strcpy(pass_through, special_device);

/*  Search to the last '/'.  Append the newly created pass-through
 *  device file onto the end.  If no '/' was found, then use the
 *  current directory.
 */
if ((ptr = strrchr(pass_through, '/')) != NULL)
{
        (void) strcpy(++ptr, mktemp(HATEMP));
}
else
{
        (void) strcpy(pass_through, mktemp(HATEMP));
}
```

```
/*
 *  Make the pass-through device node using the pass-through
 *  major number of the target device and the general use
 *  pass-through minor number (0xff).         */
if (mknod(pass_through, (S_IFCHR | S_IREAD | S_IWRITE),
          makedev(pass_thru_device.maj,0xff)) < 0)
{
        fprintf(stderr,"Unable to make pass-through node\n");
}

/*
 *  Now that the pass-through node has been created, perform
 *  an open it so that the command can be issued.
 */
if ((file_des = open(pass_through, O_RDWR)) < 0)
{
        fprintf(stderr,"Open of pass-through device failed\n");

}

/*
 *  Set of the SCSI command descriptor block.  In this
 *  example the command will be a read of block 256 on
 *  logical unit 0.
 */
scs.ss_op = SS_READ;
scs.ss_lun = 0;
scs.ss_addr = 256;
scs.ss_len = 1;
scs.ss_cont = 0;

/* Set up the SB */
sb_ptr = &sb;
sb_ptr->sb_type = ISCB_TYPE;

/*  Fill in the command address and size, the data transfer
 *  address and the amount of data to be transferred, and
 *  set the transfer mode to be a read from the device. */
sb_ptr->SCB.sc_cmdpt = SCS_AD(scs);
sb_ptr->SCB.sc_cmdsz = SCS_SZ(scs);
sb_ptr->SCB.sc_datapt = data_buffer;
sb_ptr->SCB.sc_datasz = 512;
sb_ptr->SCB.sc_mode = SCB_READ;
```

```
/* Set the timeout to 5 seconds */
sb_ptr->SCB.sc_time = 5000;


/*
 *  Issue the SDI_SEND ioctl to send the command to
 *  the Host Adapter driver.  The third argument is the
 *  pointer to the SB.
 */

if( ioctl(file_des, SDI_SEND, sb_ptr) < 0 )
{
        fprintf(stderr,"SCSI read command failed\n");
}
```

3 Functions

Introduction

This chapter describes the SDI functions for the target driver.

The SDI consists of eight functions. Each function is described one to a page with the following headings used to describe each aspect of the command:

NAME	The function name and a brief description
SYNOPSIS	How a function appears in the source code
ARGUMENTS	A description of each argument
DESCRIPTION	The description of the function
RETURN VALUE	The value returned when the function is called from a driver
LEVEL	Whether the function can be called from the base level only or from the base and interrupt levels.
EXAMPLE	An example usage of the SDI function

A summary of the functions is:

SDI Function	Description	Computer Type	Interrupt Usable?
`sdi_freeblk`(*pt*)	Release a SCSI block	Any	Yes
`sdi_getblk`()	Get a SCSI block	Any	No
`sdi_getdev`(*addr, dev*)	Get pass-through device number	Any	Yes
`sdi_icmd`(*pt*)	Perform command now	Any	Yes
`sdi_init`()	Initialize HA driver	Any	No
`sdi_name`(*addr, name*)	Get controller name	Any	Yes
`sdi_send`(*pt*)	Send command to device	Any	Yes
`sdi_translate`(*pt,bflags,procp*)	Translate SCB virtual address	Any	No

The `Computer Type` column indicates on which computers the function can be called. "Any" indicated in the `Computer Type` column refers to the type of computer the function is supported under. The `Interrupt Usable?` column indicates whether the function can be called from within an interrupt routine. HA stands for Host Adapter.

NAME

sdi_freeblk — release a previously allocated SCSI block

SYNOPSIS

```
long
sdi_freeblk(pt)
struct sb *pt;
```

ARGUMENT

pt	pointer to the sb (SCSI block) structure

DESCRIPTION

sdi_freeblk returns an sb to the free block pool. The sb_type member of the sb structure is checked to ensure that a valid sb is returned.

RETURN VALUE

The normal return is SDI_RET_OK. A return value of SDI_RET_ERR indicates an error with the pointer.

LEVEL

Base or Interrupt

EXAMPLE

This function is typically used after a job completes. In this example diskfreejob cleans up after a disk job completes and is called with a pointer to a struct job. job contains information about the disk job including a pointer to the sb for the job. sdi_freeblk is called with the pointer to the job structure as part of the clean-up operation.

```
struct job {
        struct sb *j_sbptr;
         ...

}

diskfreejob(jp)
struct job *jp; {
        /* Perform job clean up */
        ...

        /*  Return SB to SDI */
        if( sdi_freeblk(jp->j_sbptr) != SDI_RET_OK)
        {
         /* SB rejected - print error message */
         cmn_err(CE_WARN, "DISK: SB rejected by SDI.");
        }

        ...

}
```

NAME

sdi_getblk — allocate a SCSI block for the target driver

SYNOPSIS

```
struct sb *
sdi_getblk()
```

DESCRIPTION

sdi_getblk allocates an sb from SDI. Only sdi_getblk should be used to allocate an sb. This function may sleep and should not be called at interrupt level. sc_comp_code is set to SDI_UNUSED in the returned sb.

SDI may add fields to the end of the sb for use by SDI. This implies that target drivers may not use sb structures allocated themselves. Target drivers must allocate an sb structure with sdi_getblk. Some of the information added at the end of the sb includes physical addresses. If these addresses do not exist, the computer will panic.

RETURN VALUE

Pointer to an sb structure

LEVEL

Base Only

EXAMPLE

In the example, diskopen1 is called by the open routine for a disk target driver. In this routine, a disk-specific structure is initialized the first time the disk is accessed. This initialization includes allocating an sb for sending request sense commands to the disk. The following structure is used in the example.

```
struct disk_st {
      long disk_state;              /* State of this disk */
      struct scsi_ad disk_addr;      /* Major/Minor number of device
*/
      struct sb *disk_fltreq;       /* SCSI block for request sense
*/
      ...
};
diskopen1(major, minor)
long major, minor;
{
      struct disk_st *disk;
      /* Based on the major and minor numbers of the disk,
       * index into the array of disk structures and get the
       * pointer to the one for this disk.
       */
      disk = &Disk[diskintmin(major, minor)];
      /* Check to see if this disk has been initialized */
      if ((disk->disk_state & DISK_INIT) == 0)
         {
            /* This is the first access to the disk so initialize
             * some of the data structures for the disk.
             */
```

```
              /* Get an SB for request sense jobs for this disk */
              disk->disk_fltreq = sdi_getblk();

              /* Fill in the major and minor numbers and the
               * logical unit number in the address structure.
               */
              disk->disk_addr.sa_major = major;
              disk->disk_addr.sa_minor = minor;
              disk->disk_addr.sa_lun = LUN(minor);
        }
        ...
```

NAME

sdi_getdev — convert device number to pass-through device number

SYNOPSIS

```
void
sdi_getdev(addr, dev)
struct scsi_ad *addr;
dev_t dev;
```

ARGUMENTS

addr pointer to the scsi_ad (SCSI device address) structure. The pass-through device number is returned in sa_major and sa_minor members of the structure.

dev device major/minor number pair.

DESCRIPTION

sdi_getdev translates a device major/minor number pair into the pass-through interface major/minor number pair for that device. The pass-through major/minor number is returned in the addr structure.

RETURN VALUE

None

LEVEL

Base or Interrupt

EXAMPLE

A target driver uses the pass-through device number when logging non-buffer related errors. The following example shows how sdi_getdev can be used after an unsuccessful call to sdi_icmd.

```
struct  sb *sb_ptr;
   struct  scsi_ad  dk_addr;
   dev_t  pt_dev;
   ...

   /* Call sdi_icmd to send an immediate command */
      if (sdi_icmd(sb_ptr) != SDI_RET_OK)
   {
      /* The call was unsuccessful.  Print an error message,
       * get the pass-through device number, and log an error
       * against the device.
       */
       cmn_err(CE_WARN, "DISK: Bad SB type to SDI. ");
       sdi_getdev(&disk_addr, &pt_dev);
       lognberr(pt_dev.maj, pt_dev.min, 0, 0, 0,
                &disk->disk_stat.ios, 0, 0, 0);
   }
   ...
```

NAME

sdi_icmd — perform requested operation immediately

SYNOPSIS

```
int
sdi_icmd(pt)
struct sb *pt;
```

ARGUMENT

pt	pointer to the sb (SCSI block) structure. The sb_type member of the sb structure must be set to either SFB_TYPE or ISCB_TYPE.

DESCRIPTION

sdi_icmd sends an immediate sb to a device. Immediate means that this function bypasses queued scbs and immediately accesses the device to perform the requested operation. This function is typically used during error handling.

In contrast to an operation using an sfb, operations using an scb send the job to the requested logical unit.

Coming in at immediate-priority, operations using an sfb are executed in the order submitted and take priority over scb operations. Only one immediate of each command type (SFB_TYPE or ISCB_TYPE) may be outstanding to a particular logical unit.

RETURN VALUE

A return code of SDI_RET_OK indicates that the request is in progress and the target driver interrupt routine will be called. A return code of SDI_RET_ERR indicates that the type field is invalid. After a logical unit queue is resumed, all outstanding immediate control and function blocks are returned before the next normal command is returned.

LEVEL

Base or Interrupt

EXAMPLE

The following example shows how an sb is re-sent using sdi_icmd when the completion code indicates that a retry (SDI_RETRY) is requested. diskint is an example disk target driver interrupt routine.

```
void
diskint(sb_ptr)
struct sb *sb_ptr;
{
     ...

     /* Check the completion code of the SCB to see if the
      * command needs to be retried.
      */
```

```
          if (sb_ptr->SCB.sc_comp_code & SDI_RETRY )
          {
              /* Retry the command request using sdi_icmd */
                 if (sdi_icmd(sb_ptr) != SDI_RET_OK)
          {
              /* If the return value of sdi_icmd is not OK,
               * print an error message.
               */
               cmn_err(CE_WARN, "DISK: Bad SB type to SDI.");
          }
          return;
      }
      ...
```

NAME

`sdi_init` — initialize the host adapter

SYNOPSIS

```
extern int sdi_started
void
sdi_init()
```

DESCRIPTION

`sdi_init` initializes the host adapter to accept SDI functions. The `sdi_started` flag is provided so that a target driver can determine if another target driver may have already called `sdi_init`. If `sdi_started` is zero, `sdi_init` must be called.

RETURN VALUE

None

LEVEL

Base Only

EXAMPLE

This example shows the sequence of instructions used to test `sdi_started` and call `sdi_init`.

```
extern int sdi_started
    ...
    /* Check to see if the Host Adapter driver has
     * already been started.
     */
    if( !sdi_started )
    {
            /* Call the Host Adapter driver init routine */
            sdi_init();
    }
    ...
```

NAME

sdi_name — get name of addressed controller

SYNOPSIS

```
void
sdi_name(addr, name)
struct scsi_ad *addr;
char *name;
```

ARGUMENTS

addr	pointer to the scsi_ad (SCSI device address) structure
name	string containing the device *name*

DESCRIPTION

sdi_name decodes a device number into a character string so that the device number can be displayed (with cmn_err). The controller name is copied into *name*. The returned string may be as long as 48 bytes. (You must allocate 49 bytes.) Access to the controller name can only be used for display; the driver should not attempt to decode the string for other uses. The string returned is "SLOT # TC #", with the slot number of the host adapter card filled in and the target controller ID filled in. On the 3B4000 computer, the string returned is "SLIC # Controller #". On the 3B4000, if the device is on an adjunct processor, the string returned is "Controller #".

RETURN VALUE

None

LEVEL

Base or Interrupt

EXAMPLE

This example is a sample print routine in a disk target driver. The arguments to the cmn_err function are the device number and a string to display (on the system console).

```
diskprint(dev, str)
dev_t dev;
char *str;
{
   char name[49];        /* Character array for device name */
   struct scsi_ad addr; /* SCSI address structure          */

   /* Fill in the SCSI device address based on the device
      number */
   addr.sa_major = emajor(dev);
   addr.sa_minor = eminor(dev);
   addr.sa_lun = LUN(eminor(dev));
```

```
        /* Call sdi_name with the address of the SCSI address
         * structure and a pointer to the character array.
         */
        sdi_name(&addr, name);

        /* Print the error message */
           cmn_err(CE_WARN, "%s, Unit %d, Partition %d:  %s", name,
                   addr.sa_lun, eminor(dev) & PARTMASK, str);
}
```

NAME

sdi_send — send SCSI command to the controller

SYNOPSIS

```
long
sdi_send(pt)
struct sb *pt;
```

ARGUMENT

pt	pointer to the sb (SCSI block) structure

DESCRIPTION

sdi_send accepts a pointer to an sb (SCSI block) and sends the SCSI command to the controller for routing to a specific SCSI device. The SCSI block must have been allocated from the host adapter pool of SCSI blocks and the addresses translated via the sdi_translate function. The type field must be SCB_TYPE. Commands sent via this function are executed in the order they are received.

RETURN VALUE

SDI_RET_OK return indicates that the request is in progress and the target driver interrupt handler will be called. SDI_RET_RETRY indicates that SDI cannot accept the job at this time, and it should be retried later. SDI_RET_ERR indicates the sb_type is invalid. When a device is opened for pass through, SDI_RET_RETRY is returned.

LEVEL

Base or Interrupt

EXAMPLE

This example shows how sdi_send is used. disksend is an example disk target driver routine that is called internally within the target driver to send a command to a device. It is passed a pointer to an SCSI block.

```
disksend(sb_ptr)
struct sb *sb_ptr;
{
    int sendret;        /* sdi_send return value */
    extern int sendid;  /* timeout ID for retry  */

        /* Call sdi_send with the SB pointer for the job */
        if ((sendret = sdi_send(sb_ptr)) != SDI_RET_OK)
        {
           /* If sdi_send returned retry, set up a timeout to
            * submit the job later
            */
           if (sendret == SDI_RET_RETRY)
           {
              /* Call timeout and save the ID */
                 sendid = timeout(disksendt, sb_ptr, LATER);
                 return;
           }
        else
        {
            /* The Host Adapter driver could not process the job.
             * Print an error message.
             */
            cmn_err(CE_WARN, "DISK: Bad SB type to SDI. ");
            continue;
        }
    }
    ...
```

NAME

sdi_translate — translate scb virtual addresses

SYNOPSIS

```
void
sdi_translate(pt, bflags, procp)
struct sb *pt;
int bflags;
proc_t procp;
```

ARGUMENTS

pt	pointer to the sb (SCSI block) structure
bflags	the b_flags member of the buf_t (buffer header) structure
procp	pointer to the procp_t process pointer

DESCRIPTION

sdi_translate is called to allow the SDI to perform machine-specific base level virtual to physical address translation for the host adapter. This function is called each time an scb is assembled for transmission before the sdi_send or sdi_icmd functions are called.

IMPORTANT: It is the CDB aspect of the scb that requires translation; therefore, the sfb must never be run through sdi_translate. Another important consideration is that if the data area is not a contiguous segment of memory, the B_PHYS flag must be set. Especially when allocating more than 2K (2048 bytes) of memory. (This flag is defined in ddi.h.)

The sb_type, sc_cmdpt, sc_cmdsz, sc_datapt, sc_datasz, and sc_link fields must be valid. The *bflags* argument is the same as the b_flags member of the buf_t(D4P) structure. The B_READ and B_PHYS flags are used by sdi_translate. The target driver must guarantee that the data area and command area are locked into memory and are accessible by the requester. sdi_translate should not be called if the data address is supplied by SDI.

In the case of a block read or write, the data area is locked automatically by the kernel.

This function may sleep and must be called while executing as the requesting process.

RETURN VALUE

None.

LEVEL

Base Only

EXAMPLE

In this example, the values that must be initialized prior to the call to sdi_translate are set in the sb structure.

```
struct sb  *sb_ptr;        /* SCSI Block */
struct scb *scb;           /* SCSI control block */
struct scs *cmd;           /* SCSI command */
buf_t  *bp;                /* Buffer pointer */
char *buffer;              /* Buffer for data */
unsigned int size;         /* Size of the buffer */
unsigned short mode;       /* Direction of the transfer */

...

/* Set the command address and the command size */
scb->sc_cmdpt = SCS_AD(cmd);
scb->sc_cmdsz = SCS_SZ;

/* Set the data address and the data size */
scb->sc_datapt = buffer;
scb->sc_datasz = size;

sdi_translate(sb_ptr, bp->b_flags, procp());

...
```

4 Structures

Introduction

This chapter describes the structures that are accessed when writing or maintaining SCSI target drivers. Five data structures are used by SDI to handle data transmission between the target drivers and the SCSI host adapter. Additional structures, the `scm` and the `scs` structures, are used to send a command from a user program to a SCSI device using the pass-through interface. All header files names are shown with only the file name. All header files referenced in this manual are found in the `/usr/include/sys` directory.

The structures discussed in this chapter are:

- `sb` SCSI Block Structure
- `scb` SCSI Control Block Structure
- `scm` and `scs` SCSI Command Structures
- `scsi_ad` SCSI Device Address Structure
- `sfb` SCSI Function Block Structure
- `ver_no` SCSI Version Number Structure

sb(D4I)

Header File: sdi.h

This structure defines the SCSI block which can be either the scb (SCSI control block) or sfb structures (SCSI function block). The sb_type indicates whether the sb structure contains an scb or sfb structure.

IMPORTANT: The target driver must only allocate a SCSI block using sdi_getblk(D3I). Allocation by any other means causes the computer to panic. In addition, if one sb structure is copied, the new structure must be processed with sdi_translate(D3I).

Use the sb structure when you call an SDI function that sends a request to a SCSI device (either the sdi_send or sdi_icmd functions).

The method for using the sb structure is:

1. Use the sdi_getblk function to allocate a SCSI block.
2. Set sb_type appropriately.
3. Assign values to the members of the structure as appropriate.
4. Call the appropriate function to send the data structure.

Refer to the individual sections on the scb and sfb structures for more detailed information on structure use.

The members of the sb structure are:

Type	**Member**	**Description**
`unsigned long`	`sb_type;`	`/* Type of SDI block */`
`union{`		
`struct scb`	`b_scb;`	`/* SCSI control block */`
`struct sfb`	`b_sfb;`	`/* SCSI function block */`
`}sb_b;`		

The members of this structure are:

sb_type	indicates whether the sb_b structure is an scb or an scb structure. Values may be:

ISCB_TYPE	sb_b is an immediate scb (used with sdi_icmd)
SCB_TYPE	sb_b is an scb (used with sdi_send)
SFB_TYPE	sb_b is an sfb (used with sdi_icmd)

b_scb	the SCSI Control Block structure
b_sfb	the SCSI Function Block structure

scb(D4I)

Header File: sdi.h

The SCSI Control Block structure is used to send a command to a SCSI device. The scb contains a pointer to a Command Descriptor Block (CDB) that describes the command to the target controller. (The CDB command information is created using either the scm(D4I) or the scs(D4I) structures, described later in this chapter. The CDB contains the operation code of the instruction you wish to send and other command-specific data.)

The method of using the scb structure is:

1. Fill in the appropriate information in the CDB, put the address of the CDB in the sc_cmdpt member of the scb structure, and set sc_cmdsz.
2. Set the sb_type member of the sb structure to indicate whether this structure is being used by sdi_icmd(D3I) or for sdi_send(D3I).
3. Set the sc_link member of the scb structure to NULL.
4. Set the sc_datapt member of the scb structure to the virtual address of the data area, and set sc_datasz. NOTE: Always assign values to sc_datapt and sc_datasz if the command requires a data area. You must set these two fields to NULL.
5. Call sdi_translate(D3I) to resolve virtual to physical addressing. Before calling this function, these fields must be set: sb,type, sc_cmdpt, sc_cmdsz, sc_datapt, sc_datasz, and sc_link.
6. Call either sdi_send or sdi_icmd to send the CDB to the SCSI device.

IMPORTANT: After the scb is sent, do not change the information in this structure or anything referenced by it, for example, in the structure that describes the CDB, until after the job completes.

On successful job completion, sc_comp_code is set to SDI_ASW (all seems well).

For an error condition, check sc_comp_code for SDI_CKSTAT, then check sc_status to determine the type of error that was returned by the target controller. (When an error occurs on a SCSI device, the error is passed to the controller and then through the firmware of the host adapter. SDI acknowledges this interaction and sets the error code from the target controller in sc_status and sets sc_comp_code to SDI_CKSTAT.)

If sc_comp_code is not SDI_CKSTAT or SDI_ASW, then the value in bits 0-27 of sc_comp_code indicates the nature of the error and the value in bits 28-31 indicates how to process the error. NOTE: As an alternative, you may wish to check bits 28-31 for SDI_ERROR and then test bits 0-27 for more specific information. A scenario for this usage is shown in the explanation for sc_comp_code later in this section.

The members of the scb structure are:

Type	Member	Description
unsigned long	sc_comp_code;	/* Current job status */
void	(*sc_int)();	/* Target Driver interrupt handler */
caddr_t	sc_cmdpt;	/* Target command */
caddr_t	sc_datapt;	/* Data area */
long	sc_wd;	/* Target driver word */
time_t	sc_time;	/* Time limit for job */
struct scsi_ad	sc_dev;	/* SCSI device address */
unsigned short	sc_mode;	/* Mode flags for current job */
unsigned char	sc_status;	/* Target status byte */
char	sc_fill;	/* Fill byte */
struct sb	*sc_link;	/* Link to next scb command */
long	sc_cmdsz;	/* Size of command */
long	sc_datasz;	/* Size of data */
long	sc_resid;	/* Bytes to transfer after data */

The fields that the host adapter can change are: sc_comp_code, sc_status, and sc_time.

More information on each member of the scb structure follows:

sc_comp_code
: the job completion status. This member is tested in the interrupt routine after the job has completed. Use sc_comp_code by testing for SDI_ASW (normal return). If SDI_ASW is not present, test bits 28-31 to determine if SDI_ERROR was set to indicate an error occurred. The remaining values in bits 28-31 indicate how to process the error. The values for bits 28-31 are covered in the following text. Values for bits 0-27 are covered at the end of the scb structure section. Refer to the header file for more information on how to extract the values in each bit position. Values for bits 28-31 are:

SDI_ERROR	an error occurred.
SDI_RETRY	the error is unrelated to this job and the job should be retried (device dependent).
SDI_MESS	a message describing this event has been printed on the console and logged.
SDI_SUSPEND	the host adapter has suspended sending normal commands to the logical unit and the target driver is responsible for resuming the queue. Immediate commands can still be sent with the `sdi_icmd` function, but the SCSI device cannot be opened for pass-through.

`sc_int`	a pointer to a target driver interrupt routine. This routine is called when the job associated with the SCSI Control Block completes. The interrupt routine is called with a single argument which is a pointer to the address of the `sb` of the job. The interrupt routine runs at the SCSI interrupt level (12). The functions in the interrupt routine must not call `pdi_sleep`(D3P), have user context, or run in the context of any particular process. `sc_int` is called when the job sending the information to the SCSI device completes. If `sc_int` is NULL, no interrupt routine is called when the job completes.
`sc_cmdpt`	a virtual address pointing to the start of a target controller command with the size indicated by the `sc_cmdsz` member. The SCSI command pointed at by `sc_cmdpt` is sent with no interpretation by the SDI software. The command area must be in kernel space and contiguous in physical memory. You must allocate your own data structure to ensure contiguous physical memory.
`sc_datapt`	a virtual address pointer pointing to the start of the data area for the given command with the size indicated by the `sc_datasz` member.
`sc_wd`	provided for use by the target drivers. This member is not examined or changed by SDI; you can use this member for any purpose.

`sc_time`	`sc_time` is the maximum number of milliseconds SDI should wait for the job to complete. The timing begins when the command is sent to the controller. The completion status must be returned before the timer runs out. If a time out occurs, The processing of queued jobs for that controller is suspended until it is resumed by the target driver. If the `sc_time` member is zero, the job is not timed. This timing should only be used to ensure the completion of the job and not for performance measurements. The returned value of `sc_time` indicates the actual amount of time that the job took and the resolution is in minutes.
`sc_dev`	SCSI device address (an instance of the `scsi_ad`(D4I) structure).
`sc_mode`	This member is any special modes for this job. The SCB_HAAD bit in the `sc_mode` field refers to the data address which is pointed to by the `sc_datapt` element. If the SCB_HAAD bit is set in the `sc_mode` field, then the data address was supplied by SDI, otherwise it was supplied by the target driver. When SCB_PARTBLK is set in the mode field, it should indicate that the data area does not define the complete transfer. In this case the `sc_resid` field indicates how many more bytes to expect in the transfer. These extra bytes are not transferred between system memory and the SCSI bus. If the transfer is a write, zeros are sent to the controller.
`sc_status`	contains the value returned by the target controller. If a CHECK CONDITION status is returned, the host adapter suspends processing of commands to that device. (CHECK CONDITION is defined in the *AT&T SCSI Definition, Select Code 305-013.*)
`sc_fill`	(not supported.) A character with which an incomplete data block is padded. All incomplete data blocks are always padded with zeros.
`sc_link`	(not supported.) You must set this member to NULL before calling `sdi_translate`.

`sc_cmdsz`	command size in bytes.
`sc_datasz`	data size in bytes of the requested input or output buffer.
`sc_resid`	(not supported by the 6386 Computer.) The number of bytes not transferred to/from the target controller. This is used for partial block transfers. Residue bytes which are received from the target controller are discarded by SDI.

sc_comp_code Values

An alphabetic summary of the completion codes follows. The third column indicates the hexadecimal value associated with the completion code value. The fourth column indicates on which computer type the use of the completion code is supported. The fifth column describes which flags in bit positions 28-31 of `sc_comp_code` are enabled by the completion code. (E is SDI_ERROR, M is SDI_MESS, R is SDI_RETRY, S is SDI_SUSPEND, and "-" indicates that the respective flag is not enabled.)

sc_comp_code value	Description	Hex Value	Computer Type	Flags
SDI_ABORT	Command was aborted	0x05	unsupported	EMRS
SDI_ASW	Job completed normally	0x01	6386	----
SDI_CKSTAT	Check the status byte	0x0e	6386	E-RS
SDI_CRESET	Reset caused by this unit	0x07	6386	E-RS
SDI_HAERR	Host adapter error	0x0b	6386	EM-S
SDI_LINKF0	Linked cmd done without flag	0x02	unsupported	----
SDI_LINKF1	Linked cmd done with flag	0x03	unsupported	----
SDI_MEMERR	Memory fault	0x0c	unsupported	E-R-
SDI_MISMAT	Parameter mismatch	0x12	6386	EMRS
SDI_NOALLOC	This block not allocated	0x00	6386	----
SDI_NOSELE	SCSI bus select failed	0x11	6386	E--S
SDI_NOTEQ	Addressed device not present	0x08	6386	EM--
SDI_ONEIC	More than one immediate request	0x17	6386	E---
SDI_OOS	Device out of service	0x10	6386	EM--
SDI_PROGRES	Job in progress	0x13	6386	----
SDI_QFLUSH	Job was flushed	0x04	6386	EMR-
SDI_RESET	Reset detected on the bus	0x06	6386	EMRS
SDI_SBUSER	SCSI bus error	0x0d	unsupported	E-RS
SDI_SCBERR	SCB error	0x0f	6386	E---
SDI_SFBERR	SFB error	0x19	6386	E---
SDI_SHORT	TC did not transfer all data	0x1b	unsupported	E-RS
SDI_TCERR	Target protocol error detected	0x1a	unsupported	E--S
SDI_TIME	Job timed out	0x09	unsupported	E-RS
SDI_UNUSED	Job not in use	0x14	unsupported	----
SDI_V2PERR	vtop failed	0x08	unsupported	EM--

The `sc_comp_code` values are defined as follows:

`Values for the 6386 Computer:`

SDI_SCBERR	the SCSI control block contains an error or an invalid type.

SDI_ASW	(all seems well) the job completed without an error.
SDI_CKSTAT	check the status byte. This value is set when the target controller returns a status other than good.
SDI_HAERR	a problem occurred between SDI and the host adapter controller. Possible causes are I/O bus parity or a failed host adapter.
SDI_NOALLOC	the requested block was not allocated to a target driver by SDI. If a target driver detects this value, panic the system (using the `pdi_cmn_err`(D3P) function). `sc_comp_code` is set to this value when the SCSI block is released from the target driver.
SDI_NOSELE	SDI timed out trying to select the controller.
SDI_ONEIC	more than one immediate request has been sent.
SDI_OOS	SDI_OOS indicates that the firmware is not operational.
SDI_PROGRES	the job is not complete (set by SDI in the `sdi_icmd` and `sdi_send` functions)
SDI_QFLUSH	when the target driver requested a job queue flush for a device, all jobs in the queue are returned with this completion code set.
SDI_RESET	SDI detected a reset on the SCSI bus. All outstanding or jobs queued at the target controller are returned to the target drivers with this condition code set. If the job is being controlled by SDI, but has not been sent out on the bus, the job is not returned. This code is also returned if a target driver requests that a target controller be reset.
SDI_SFBERR	there is an error in one of the fields in the `sfb` structure.
SDI_TIME	SDI timed out a job.
SDI_UNUSED	the host adapter is not using the control structure. SDI sets the `sc_comp_code` to this value when it allocates a SCSI block for a target driver.

scm(D4I) and scs(D4I)

Header File: `scsi.h` (both structures)

Both the scm and scs structures are used by target drivers and with the pass-through interface to send a SCSI command to a SCSI device. The scs structure defines the layout for a group 6 (six-byte command length) Command Descriptor Block (CDB); scm is a group 10 (ten-byte command length) CDB. Both types of CDBs are described in the *ANSI Small Computer System Interface (SCSI), X3T9, 2/82-2, Revision 17B*. Refer to the ANSI manual for more information on individual structure members. The sm_pad0 member ensures that the sm_addr member does not cross a 32-bit word boundary.

The members of the scm structure are:

Type	Member	Description
int	sm_pad0 : 16;	/* 16-bit pad */
int	sm_op : 8;	/* Opcode */
int	sm_lun : 3;	/* Logical unit number */
int	sm_res1 : 5;	/* reserved field */
unsigned	sm_addr;	/* Block address */
int	sm_res2 : 8;	/* reserved field */
int	sm_len : 16;	/* Transfer length */
int	sm_cont : 8;	/* Control byte */

NOTE Because of the sm_pad0 member, you must add 2 to the address of the scm structure when specifying it in the scb structure.

The members of the scs structure are:

Type	Member	Description
int	ss_op : 8;	/* Opcode */
int	ss_lun : 3;	/* Logical unit number */
int	ss_addr : 21;	/* Block address */
int	ss_len : 8;	/* Transfer length */
int	ss_cont : 8;	/* Control byte */

scsi_ad(D4I)

Header File: sdi.h

The SCSI device address structure is used by every scb or sfb structure with the appropriate SCSI device. SDI interprets the external major and minor numbers, the logical unit number, and the extended logical unit number to send the scb or sfb to the correct device.

The members of the scsi_ad structure are:

```
Type            Member      Description

long            sa_major;   /* Major number                   */
long            sa_minor;   /* Minor number                   */
unsigned char   sa_lun;     /* logical unit number            */
unsigned char   sa_exlun;   /* extended logical unit number */
short           sa_fill;    /* Fill word                      */
```

This structure consists of the device number which is passed to the target driver by the kernel, and logical unit number. The sa_major and sa_minor members (external major/minor numbers) are long integers to allow for future expansion of the major and minor numbers.

NOTE Use of the sa_exlun member is not supported for the 6386 computer.

For example, if the target controller wanted to access extended logical unit 0x021a, then sa_lun would equal 0x82, and sa_exlun would equal 0x1a.

sfb(D4I)

Header File: `sdi.h`

The SCSI function block serves as a mechanism for sending control information from a target driver, to the host adapter, or to a SCSI device. (The `scb` is sent to the device, the `sfb` is generally sent to all other receiving entities.)

An `sfb` is sent to the device when an abort or reset message is required. (Abort and reset messages have a different protocol than do the commands sent to the SCSI device with a CDB.)

The members of the `sfb` structure are:

Type	Member	Description
`unsigned long`	`sf_comp_code;`	`/* Current job status*/`
`void`	`(*sf_int)();`	`/* Target Driver interrupt handler */`
`struct scsi_ad`	`sf_dev;`	`/* SCSI device address */`
`unsigned long`	`sf_func;`	`/* Function to be performed */`
`int`	`sf_wd`	`/* Target driver word */`

The `sf_comp_code` is identical to the `sc_comp_code` in the `scb`, and it takes on the same values.

The `sf_int` and `sf_dev` entries are used the same as in the `scb` structure. The only field which the host adapter changes in the `sfb` structure is `sf_comp_code`.

The `sf_func` member indicates the operation to be performed.

SFB_ABORTM	requests that an abort message be sent to the addressed logical unit.
SFB_FLUSHR	requests that a logical queue unit be flushed.
SFB_NOPF	requests the target driver to not perform an operation.
SFB_RESETM	requests that SDI send a bus device reset message to the addressed controller.
SFB_RESUME	requests that a queue permit normal job flow to a logical unit. This command is used after SFB_SUSPEND, but no error results if SFB_RESUME is called first.

SFB_SUSPEND	requests that a queue be suspended. This indicates that normal job flow to the logical unit is halted until the queue is resumed by the target driver.

ver_no(D4I)

Header File: `sdi.h`

The version number structure is used to ensure that the version of SDI is appropriate for the target drivers.

The members of this structure are:

Type	Member	Description
`unsigned char`	`sv_release;`	`/* Release number */`
`unsigned char`	`sv_machine;`	`/* Computer type */`
`short`	`sv_modes;`	`/* SCSI Release Number */`

`sv_release` indicates SDI release number (set to 1 for the first release). On the 3B4000 computer, the release is set in the SHA_RELEASE constant defined in `had.h`. On the 3B2 computer, the release is hard coded as 1.

`sv_machine` indicates the type of computer you are using. Valid values are:

CDB	Command Descriptor Block. Information that describes a command that is sent to the SCSI device from a target driver or from a user program. The commands described by the CDB are listed in the *AT&T SCSI Definition*. The scm(D4I) and scs(D4I) structures are used describe the information in a CDB to SDI.
device	A single logical unit that may be attached to a target controller or directly to the SCSI bus. If a target controller has four disk drives attached to it, each disk drive constitutes a device.
EDT	Equipped Device Table. A list maintained by the operating system of all circuit boards (feature cards on a 3B2 computer) that are attached to a computer.
HA	Host Adapter. A machine-dependent hardware component that provides access to the machine-independent SCSI bus. The driver associated with the host adapter contains the SDI functions.
host computer	The computer to which the SCSI bus is attached.

multi-host functionality	A feature on the 3B4000 Computer that permits more than one computer to share a SCSI bus. LI "pass-through interface" A feature of SDI that permits a user program to send a command directly to a SCSI device without using a target driver.
PDI	Portable Driver Interface. A library of functions used to code a SCSI target driver.
sb(D4I)	The SCSI Block structure that contains either an `scb` or `sfb` structure.
scb(D4I)	The SCSI Command Block structure that contains data for identifying a command descriptor block to SDI, for handling completion status, and for handling interrupts.
scm(D4I)	The SCSI Command Medium-size structure that contains data for creating a ten-byte command descriptor block.
scs(D4I)	The SCSI Command Small-size structure that contains data for creating a six-byte command descriptor block.
SCSI	Small Computer System Interface. A hardware and software standard that treats computer peripherals as modules. These modules can be added more easily and in greater numbers than have been previously available for small computer systems.
SCSI bus	A cable that connects SCSI devices and controllers to a computer.
scsi_ad(D4I)	The SCSI device address structure that contains data identifying the address of a specific SCSI device.
SDI	A set of structures and functions that permit a driver to access a SCSI device.
sdi_freeblk(D3I)	A function used to release a previously allocated SCSI block.
sdi_getblk(D3I)	A function used to allocate a SCSI block for the target driver.

sdi_getdev(D3I)	A function used to convert device number to pass-through device number.
sdi_icmd(D3I)	A function used to perform requested operation immediately.
sdi_init(D3I)	A function used to initialize the host adapter.
sdi_name(D3I)	A function used to get name of addressed controller.
sdi_send(D3I)	A function used to send SCSI command to the controller.
sdi_translate(D3I)	A function used to translate scb virtual addresses.
sfb(D4I)	The SCSI Function Block structure that contains data for sending an immediate command to SDI.
target driver	a UNIX operating system driver that controls a particular class of a device, such as a disk drive or a tape driver.
ver_no(D4I)	The SCSI Version Number structure that identifies the current release and version number of SCSI hardware and software components.

Index

320–710

UNIX
PRESS

A Prentice Hall Title

ISBN 0-13-9575