Writing a SCSI Device Driver for Linux

Rickard E. Faith

Mon Jun 7 21:30:37 1993

Contents

1 Introduction

In this paper, I have presented an overview of the information necessary to write a low-level SCSI device driver for LINUX. The author of such a driver is expected to have a thorough knowledge of C programming, but does not need to have previous experience writing kernel or device driver code. References to other resources have been provided—access to these resources will make writing and debugging a low-level driver much easier.

1.1 LINUX: The Operating System

LINUX is a freely available UNIX[1]-like operating system for 386 and i486 systems.[2] The kernel, written from scratch by Linus Torvalds,[3] contains *no* proprietary source code and is available for use without licensing fees. Linus distributes the kernel freely under the terms of the GNU General Public License.[4]

Since its introduction in November 1991, LINUX has matured rapidly. The kernel contains support for SCSI disks and tapes, sound cards, mice, TCP/IP, and several file systems. The C library contains POSIX, System V, and BSD compatible routines. LINUX installations are capable of running large software packages such as TEX and the X Window System.

1.2 Why You Want to Write a SCSI Driver

Currently, the LINUX kernel contains drivers for the following SCSI host adapters: Adaptec 1542, Adaptec 1740, Future Domain TMC-1660/TMC-1680, Seagate ST-01/ST-02, UltraStor 14F, and Western Digital WD-7000. You may want to write your own driver for an unsupported host adapter. You may also want to re-write or update one of the existing drivers.

2 What is SCSI?

The foreword to the SCSI-2 standard draft [ANS] gives a succinct definition of the Small Computer System Interface and briefly explains how SCSI-2 is related to SCSI-1 and CCS:

[1]UNIX is a registered trademark of AT&T Bell Laboratories.

[2]386 and i486 are trademarks of Intel Corporation.

[3]Internet mail address: torvalds@kruuna.Helsinki.FI

[4]For those interested, a copy of the GNU General Public License is available via anonymous ftp from `prep.ai.mit.edu`. Copies can also be obtained by writing to the Free Software Foundation, Inc., 675 Massachusetts Ave., Cambridge, MA 02139, USA.

The SCSI protocol is designed to provide an efficient peer-to-peer I/O bus with up to 8 devices, including one or more hosts. Data may be transferred asynchronously at rates that only depend on device implementation and cable length. Synchronous data transfers are supported at rates up to 10 mega-transfers per second. With the 32 bit wide data transfer option, data rates of up to 40 megabytes per second are possible.

SCSI-2 includes command sets for magnetic and optical disks, tapes, printers, processors, CD-ROMs, scanners, medium changers, and communications devices.

In 1985, when the first SCSI standard was being finalized as an American National Standard, several manufacturers approached the X3T9.2 Task Group. They wanted to increase the mandatory requirements of SCSI and to define further features for direct-access devices. Rather than delay the SCSI standard, X3T9.2 formed an ad hoc group to develop a working paper that was eventually called the Common Command Set (CCS). Many disk products were designed using this working paper in conjunction with the SCSI standard.

In parallel with the development of the CCS working paper, X3T9.2 began work on an enhanced SCSI standard which was named SCSI-2. SCSI-2 included the results of the CCS working paper and extended them to all device types. It also added caching commands, performance enhancement features, and other functions that X3T9.2 deemed worthwhile. While SCSI-2 has gone well beyond the original SCSI standard (now referred to as SCSI-1), it retains a high degree of compatibility with SCSI-1 devices.

2.1 SCSI Phases

The "SCSI bus" transfers data and state information between interconnected SCSI devices. A single transaction between an "initiator" and a "target" can involve up to 8 distinct "phases." These phases are almost entirely determined by the target (e.g., the hard disk drive). The current phase can be determined from an examination of five SCSI bus signals, as shown in Table 1 [LXT91, p. 57].

Some controllers (notably the inexpensive Seagate controller) require direct manipulation of the SCSI bus—other controllers automatically handle these low-level details. Each of the eight phases will be described in detail.

BUS FREE Phase The BUS FREE phase indicates that the SCSI bus is idle and is not currently being used.

ARBITRATION Phase The ARBITRATION phase is entered when a SCSI device attempts to gain control of the SCSI bus. Arbitration can start only if

-SEL	-BSY	-MSG	-C/D	-I/O	PHASE
HI	HI	?	?	?	BUS FREE
HI	LO	?	?	?	ARBITRATION
I	I&T	?	?	?	SELECTION
T	I&T	?	?	?	RESELECTION
HI	LO	HI	HI	HI	DATA OUT
HI	LO	HI	HI	LO	DATA IN
HI	LO	HI	LO	HI	COMMAND
HI	LO	HI	LO	LO	STATUS
HI	LO	LO	LO	HI	MESSAGE OUT
HI	LO	LO	LO	LO	MESSAGE IN

I = Initiator Asserts, T = Target Asserts, ? = HI or LO

Table 1: SCSI Bus Phase Determination

the bus was previously in the BUS FREE phase. During arbitration, the arbitrating device asserts its SCSI ID on the DATA BUS. For example, if the arbitrating device's SCSI ID is 2, then the device will assert `0x04`. If multiple devices attempt simultaneous arbitration, the device with the highest SCSI ID will win. Although ARBITRATION is optional in the SCSI-1 standard, it is a required phase in the SCSI-2 standard.

SELECTION Phase After ARBITRATION, the arbitrating device (now called the initiator) asserts the SCSI ID of the target on the DATA BUS. The target, if present, will acknowledge the selection by raising the -BSY line. This line remains active as long as the target is connected to the initiator.

RESELECTION Phase The SCSI protocol allows a device to disconnect from the bus while processing a request. When the device is ready, it reconnects to the host adapter. The RESELECTION phase is identical to the SELECTION phase, with the exception that it is used by the disconnected target to reconnect to the original initiator. Drivers which do not currently support RESELECTION do not allow the SCSI target to disconnect. RESELECTION should be supported by all drivers, however, so that multiple SCSI devices can simultaneously process commands. This allows dramatically increased throughput due to interleaved I/O requests.

COMMAND Phase During this phase, 6, 10, or 12 bytes of command information are transferred from the initiator to the target.

DATA OUT and DATA IN Phases During these phases, data are transferred between the initiator and the target. For example, the DATA OUT phase transfers data from the host adapter to the disk drive. The DATA IN phase transfers data from the disk drive to the host adapter. If the SCSI command does not require data transfer, then neither phase is entered.

STATUS Phase This phase is entered after completion of all commands, and allows the target to send a status byte to the initiator. There are nine valid status bytes, as shown in Table 2 [ANS, p. 77]. Note that since bits[5] 1–5 are used for the status code (the other bits are reserved), the status byte should be masked with `0x3e` before being examined.

Value†	Status
0x00	GOOD
0x02	CHECK CONDITION
0x04	CONDITION MET
0x08	BUSY
0x10	INTERMEDIATE
0x14	INTERMEDIATE-CONDITION MET
0x18	RESERVATION CONFLICT
0x22	COMMAND TERMINATED
0x28	QUEUE FULL

† After masking with 0x3e

Table 2: SCSI Status Codes

The meanings of the three most important status codes are outlined below:

GOOD The operation completed successfully.

CHECK CONDITION An error occurred. The REQUEST SENSE command should be used to find out more information about the error (see section 2.2).

BUSY The device was unable to accept a command. This may occur during a self-test or shortly after power-up.

MESSAGE OUT and MESSAGE IN Phases Additional information is transferred between the target and the initiator. This information may regard the status of an outstanding command, or may be a request for a change of protocol. Multiple MESSAGE IN and MESSAGE OUT phases may occur during a single

[5] Bit 0 is the least significant bit.

SCSI transaction. If RESELECTION is supported, the driver must be able to correctly process the SAVE DATA POINTERS, RESTORE POINTERS, and DISCONNECT messages. Although required by the SCSI-2 standard, some devices do not automatically send a SAVE DATA POINTERS message prior to a DISCONNECT message.

2.2 SCSI Commands

Each SCSI command is 6, 10, or 12 bytes long. The following commands must be well understood by a SCSI driver developer.

REQUEST SENSE Whenever a command returns a CHECK CONDITION status, the high-level LINUX SCSI code automatically obtains more information about the error by executing the REQUEST SENSE. This command returns a sense key and a sense code (called the "additional sense code," or ASC, in the SCSI-2 standard [ANS]). Some SCSI devices may also report an "additional sense code qualifier" (ASCQ). The 16 possible sense keys are described in Table 3. For information on the ASC and ASCQ, please refer to the SCSI standard [ANS] or to a SCSI device technical manual.

Sense Key	Description
`0x00`	NO SENSE
`0x01`	RECOVERED ERROR
`0x02`	NOT READY
`0x03`	MEDIUM ERROR
`0x04`	HARDWARE ERROR
`0x05`	ILLEGAL REQUEST
`0x06`	UNIT ATTENTION
`0x07`	DATA PROTECT
`0x08`	BLANK CHECK
`0x09`	(Vendor specific error)
`0x0a`	COPY ABORTED
`0x0b`	ABORTED COMMAND
`0x0c`	EQUAL
`0x0d`	VOLUME OVERFLOW
`0x0e`	MISCOMPARE
`0x0f`	RESERVED

Table 3: Sense Key Descriptions

TEST UNIT READY This command is used to test the target's status. If the target can accept a medium-access command (e.g., a READ or a WRITE), the command returns with a GOOD status. Otherwise, the command returns with a CHECK CONDITION status and a sense key of NOT READY. This response usually indicates that the target is completing power-on self-tests.

INQUIRY This command returns the target's make, model, and device type. The high-level LINUX code uses this command to differentiate among magnetic disks, optical disks, and tape drives (the high-level code currently does not support printers, processors, or juke boxes).

READ and WRITE These commands are used to transfer data from and to the target. You should be sure your driver can support simpler commands, such as TEST UNIT READY and INQUIRY, before attempting to use the READ and WRITE commands.

3 Getting Started

3.1 Dangers of Kernel Hacking

Writing code for the LINUX kernel (or for any kernel) is somewhat different from writing normal C code. The most striking differences are that static variables are not initialized to zero and that the common functions found in the standard C library are not available.

3.1.1 Uninitialized Static Variables are Evil

The ANSI C standard [KR88, p. 40] and the current C++ description [ES91, pp. 19 and 150] both guarantee that static variables are, by default, initialized to zero. While the merits of relying on this default initialization are debatable, static variables in the kernel are *not* automatically initialized to zero.[6] To avoid many debugging nightmares, all otherwise uninitialized static variables should be *explicitly* initialized to zero.

3.1.2 `libc` Unavailable

The kernel maintains a small set of commonly used library functions, which permit file manipulation (e.g., `open()`, `close()`), string manipulation (e.g., `strcmp()`), and

[6]Some boot/loader programs (e.g., LILO) can zero memory before loading the kernel. This will make memory appear *as if* static variables were, indeed, initialized to zero. However, when the same kernel is placed on a floppy disk, and the system is booted from that floppy, the uninitialized static variables will contain random data. Because of this behavior, a kernel which works fine when booted from the hard disk sometimes fails mysteriously when booted from the floppy.

character type testing (e.g., `isdigit()`). Most other standard C library functions, however, are *not* available.

3.1.3 Commonly Used Kernel Functions

The kernel contains several special library functions. The most commonly used function is `printk()`. This function acts exactly like `printf()`, except its output will be sent to the console or to the system log facility.

Another commonly used kernel function is `panic()`. This function takes a single parameter, which is a message to be printed on the console explaining why the system has just crashed. This function should be used if the driver enters an unrecoverable error state in which the only option is to shut down the system.

The author of a low-level device driver will need to have an understanding of how interruptions are handled by the kernel. At minimum, the kernel functions that disable (`cli()`) and enable (`sti()`) interruptions should be understood. The scheduling functions (e.g., `schedule()`, `sleepon()`, and `wakeup()`) may also be needed by some drivers. A detailed explanation of these functions is outside the scope of this paper.

3.2 Before You Begin: Gathering Tools

Before you begin to write a SCSI driver for Linux, you will need to obtain several resources.

The most important is a bootable Linux system—preferably one which boots from an IDE, RLL, or MFM hard disk. During the development of your new SCSI driver, you will rebuild the kernel and reboot your system many times. Programming errors may result in the destruction of data on your SCSI drive *and* on your non-SCSI drive. *Back up your system before you begin.*

The installed Linux system can be quite minimal: the GCC compiler distribution (including libraries and the binary utilities), an editor, and the kernel source are all you need. Additional tools like `od`, `hexdump`, and `less` will be quite helpful. All of these tools will fit on an inexpensive 20-30 MB hard disk.[7].

Documentation is essential. At minimum, you will need a technical manual for your host adapter. Since Linux is freely distributable, and since you (ideally) want to distribute your source code freely, avoid non-disclosure agreements (NDA). Most NDA's will prohibit you from releasing your source code—you might be allowed to release an object file containing your driver, but this is simply not acceptable in the Linux community at this time.

A manual that explains the SCSI standard will be helpful. Usually the technical manual for your disk drive will be sufficient, but a copy of the SCSI standard will

[7]A used 20 MB MFM hard disk and controller should cost less than US$100.

often be helpful.[8]

Before you start, make hard copies of `hosts.h`, `scsi.h`, and one of the existing drivers in the LINUX kernel. These will prove to be useful references while you write your driver.

4 The LINUX SCSI Interface

The high-level SCSI interface in the LINUX kernel manages all of the interaction between the kernel and the low-level SCSI device driver. Because of this layered design, a low-level SCSI driver need only provide a few basic services to the high-level code. The author of a low-level driver does not need to understand the intricacies of the kernel I/O system and, hence, can write a low-level driver in a relatively short amount of time.

Two main structures (`Scsi_Host` and `Scsi_Cmnd`) are used to communicate between the high-level code and the low-level code. The next two sections provide detailed information about these structures and the requirements of the low-level driver.

5 The `Scsi_Host` Structure

The `Scsi_Host` structure serves to describe the low-level driver to the high-level code. Usually, this description is placed in the device driver's header file in a C preprocessor definition, as shown in Figure 1.

The `Scsi_Host` structure is presented in Figure 2. Each of the fields will be explained in detail later in this section.

5.1 Variables in the `Scsi_Host` Structure

In general, the variables in the `Scsi_Host` structure are not used until after the `detect()` function (see section 5.2.1) is called. Therefore, any variables which cannot be assigned before host adapter detection should be assigned during detection. This situation might occur, for example, if a single driver provided support for several host adapters with very similar characteristics. Some of the parameters in the `Scsi_Host` structure might then depend on the specific host adapter detected.

[8] The October 17, 1991, draft of the SCSI-2 standard document is available via anonymous ftp from `sunsite.unc.edu` in `/pub/Linux/development/scsi-2.tar.Z`, and is available for purchase from Global Engineering Documents (2805 McGaw, Irvine, CA 92714), (800)-854-7179 or (714)-261-1455. Please refer to document X3.131-199X. In early 1993, the manual cost US$60–70.

```
#define FDOMAIN_16X0  { "Future Domain TMC-16x0",          \
                         fdomain_16x0_detect,              \
                         fdomain_16x0_info,                \
                         fdomain_16x0_command,             \
                         fdomain_16x0_queue,               \
                         fdomain_16x0_abort,               \
                         fdomain_16x0_reset,               \
                         NULL,                             \
                         fdomain_16x0_biosparam,           \
                         1, 6, 64, 1 ,0, 0}
#endif
```

Figure 1: Device Driver Header File

```
typedef struct
{
  char                *name;
  int                 (* detect)(int);
  const char          *(* info)(void);
  int                 (* queuecommand)(Scsi_Cmnd *,
                       void (*done)(Scsi_Cmnd *));
  int                 (* command)(Scsi_Cmnd *);
  int                 (* abort)(Scsi_Cmnd *, int);
  int                 (* reset)(void);
  int                 (* slave_attach)(int, int);
  int                 (* bios_param)(int, int, int []);
  int                 can_queue;
  int                 this_id;
  short unsigned int sg_tablesize;
  short               cmd_per_lun;
  unsigned            present:1;
  unsigned            unchecked_isa_dma:1;
} Scsi_Host;
```

Figure 2: The Scsi_Host Structure

5.1.1 name

`name` holds a pointer to a short description of the SCSI host adapter.

5.1.2 can_queue

`can_queue` holds the number of outstanding commands the host adapter can process. Unless RESELECTION is supported by the driver and the driver is interrupt-driven,[9] this variable should be set to 1.

5.1.3 this_id

Most host adapters have a specific SCSI ID assigned to them. This SCSI ID, usually 6 or 7, is used for RESELECTION. The `this_id` variable holds the host adapter's SCSI ID. If the host adapter does not have an assigned SCSI ID, this variable should be set to -1 (in this case, RESELECTION cannot be supported).

5.1.4 sg_tablesize

The high-level code supports "scatter-gather," a method of increasing SCSI throughput by combining many small SCSI requests into a few large SCSI requests. Since most SCSI disk drives are formatted with 1:1 interleave,[10] the time required to perform the SCSI ARBITRATION and SELECTION phases is longer than the rotational latency time between sectors.[11] Therefore, only one SCSI request can be processed per disk revolution, resulting in a throughput of about 50 kilobytes per second. When scatter-gather is supported, however, average throughput is usually over 500 kilobytes per second.

The `sg_tablesize` variable holds the maximum allowable number of requests in the scatter-gather list. If the driver does not support scatter-gather, this variable should be set to `SG_NONE`. If the driver can support an unlimited number of grouped requests, this variable should be set to `SG_ALL`. Some drivers will use the host adapter to manage the scatter-gather list and may need to limit `sg_tablesize` to the number that the host adapter hardware supports. For example, some Adaptec host adapters require a limit of 16.

[9] Some of the early LINUX drivers were not interrupt driven and, consequently, had very poor performance.

[10] "1:1 interleave" means that all of the sectors in a single track appear consecutively on the disk surface.

[11] This may be an over-simplification. On older devices, the actual command processing can be significant. Further, there is a great deal of layered overhead in the kernel: the high-level SCSI code, the buffering code, and the file-system code all contribute to poor SCSI performance.

5.1.5 cmd_per_lun

The SCSI standard supports the notion of "linked commands." Linked commands allow several commands to be queued consecutively to a single SCSI device. The `cmd_per_lun` variable specifies the number of linked commands allowed. This variable should be set to 1 if command linking is not supported. At this time, however, the high-level SCSI code will not take advantage of this feature.

Linked commands are fundamentally different from multiple outstanding commands (as described by the `can_queue` variable). Linked commands always go to the same SCSI target and do not necessarily involve a RESELECTION phase. Further, linked commands eliminate the ARBITRATION, SELECTION, and MESSAGE OUT phases on all commands after the first one in the set. In contrast, multiple outstanding commands may be sent to an arbitrary SCSI target, and *require* the ARBITRATION, SELECTION, MESSAGE OUT, and RESELECTION phases.

5.1.6 present

The `present` bit is set (by the high-level code) if the host adapter is detected.

5.1.7 unchecked_isa_dma

Some host adapters use Direct Memory Access (DMA) to read and write blocks of data directly from or to the computer's main memory. LINUX is a virtual memory operating system that can use more than 16 MB of physical memory. Unfortunately, on machines using the ISA bus[12], DMA is limited to the low 16 MB of physical memory.

If the `unchecked_isa_dma` bit is set, the high-level code will provide data buffers which are guaranteed to be in the low 16 MB of the physical address space. Drivers written for host adapters that do not use DMA should set this bit to zero. Drivers specific to EISA bus[13] machines should also set this bit to zero, since EISA bus machines allow unrestricted DMA access.

5.2 Functions in the Scsi_Host Structure

5.2.1 detect()

The `detect()` function's only argument is the "host number," an index into the `scsi_hosts` variable (an array of type `struct Scsi_Host`). The `detect()` function

[12] The so-called "Industry Standard Architecture" bus was introduced with the IBM PC/XT and IBM PC/AT computers.

[13] The "Extended Industry Standard Architecture" bus is a non-proprietary 32-bit bus for 386 and i486 machines.

should return a non-zero value if the host adapter is detected, and should return zero otherwise.

Host adapter detection must be done carefully. Usually the process begins by looking in the ROM area for the "BIOS signature" of the host adapter. On PC/AT-compatible computers, the use of the address space between `0xc0000` and `0xfffff` is fairly well defined. For example, the video BIOS on most machines starts at `0xc0000` and the hard disk BIOS, if present, starts at `0xc8000`. When a PC/AT-compatible computer boots, every 2-kilobyte block from `0xc0000` to `0xf8000` is examined for the 2-byte signature (`0x55aa`) which indicates that a valid BIOS extension is present [Nor85].

The BIOS signature usually consists of a series of bytes that uniquely identifies the BIOS. For example, one Future Domain BIOS signature is the string

```
FUTURE DOMAIN CORP. (C) 1986-1990 1800-V2.07/28/89
```

found exactly five bytes from the start of the BIOS block.

After the BIOS signature is found, it is safe to test for the presence of a functioning host adapter in more specific ways. Since the BIOS signatures are hard-coded in the kernel, the release of a new BIOS can cause the driver to mysteriously fail. Further, people who use the SCSI adapter exclusively for LINUX may want to disable the BIOS to speed boot time. For these reasons, if the adapter can be detected safely without examining the BIOS, then that alternative method should be used.

Usually, each host adapter has a series of I/O port addresses which are used for communications. Sometimes these addresses will be hard coded into the driver, forcing all LINUX users who have this host adapter to use a specific set of I/O port addresses. Other drivers are more flexible, and find the current I/O port address by scanning all possible port addresses. Usually each host adapter will allow 3 or 4 sets of addresses, which are selectable via hardware jumpers on the host adapter card.

After the I/O port addresses are found, the host adapter can be interrogated to confirm that it is, indeed, the expected host adapter. These tests are host adapter specific, but commonly include methods to determine the BIOS base address (which can then be compared to the BIOS address found during the BIOS signature search) or to verify a unique identification number associated with the board. For MCA bus[14] machines, each type of board is given a unique identification number which no other manufacturer can use—several Future Domain host adapters, for example, also use this number as a unique identifier on ISA bus machines. Other methods of verifying the host adapter existence and function will be available to the programmer.

[14]The "Micro-Channel Architecture" bus is IBM's proprietary 32 bit bus for 386 and i486 machines.

Requesting the IRQ After detection, the `detect()` routine must request any needed interrupt or DMA channels from the kernel. There are 16 interrupt channels, labeled IRQ 0 through IRQ 15. The kernel provides two methods for setting up an IRQ handler: `irqaction()` and `request_irq()`.

The `request_irq()` function takes two parameters, the IRQ number and a pointer to the handler routine. It then sets up a default `sigaction` structure and calls `irqaction()`. The code[15] for the `request_irq()` function is shown in Figure 3. I will limit my discussion to the more general `irqaction()` function.

```
int request_irq( unsigned int irq, void (*handler)( int ) )
{
  struct sigaction sa;

  sa.sa_handler  = handler;
  sa.sa_flags    = 0;
  sa.sa_mask     = 0;
  sa.sa_restorer = NULL;
  return irqaction( irq, &sa );
}
```

Figure 3: The `request_irq()` Function

The declaration[16] for the `irqaction()` function is

```
int irqaction( unsigned int irq, struct sigaction *new )
```

where the first parameter, `irq`, is the number of the IRQ that is being requested, and the second parameter, `new`, is a structure with the definition[17] shown in Figure 4.

In this structure, `sa_handler` should point to your interrupt handler routine, which should have a definition similar to the following:

```
void fdomain_16x0_intr( int irq )
```

where `irq` will be the number of the IRQ which caused the interrupt handler routine to be invoked.

[15] LINUX 0.99.7 kernel source code, `linux/kernel/irq.c`
[16] LINUX 0.99.5 kernel source code, `linux/kernel/irq.c`
[17] LINUX 0.99.5 kernel source code, `linux/include/linux/signal.h`

```
struct sigaction
{
  __sighandler_t sa_handler;
  sigset_t       sa_mask;
  int            sa_flags;
  void           (*sa_restorer)(void);
};
```

Figure 4: The `sigaction` Structure

The `sa_mask` variable is used as an internal flag by the `irqaction()` routine. Traditionally, this variable is set to zero prior to calling `irqaction()`.

The `sa_flags` variable can be set to zero or to `SA_INTERRUPT`. If zero is selected, the interrupt handler will run with other interrupts enabled, and will return via the signal-handling return functions. This option is recommended for relatively slow IRQ's, such as those associated with the keyboard and timer interrupts. If `SA_INTERRUPT` is selected, the handler will be called with interrupts disabled and return will avoid the signal-handling return functions. `SA_INTERRUPT` selects "fast" IRQ handler invocation routines, and is recommended for interrupt driven hard disk routines. The interrupt handler should turn interrupts on as soon as possible, however, so that other interrupts can be processed.

The `sa_restorer` variable is not currently used, and is traditionally set to `NULL`.

The `request_irq()` and `irqaction()` functions will return zero if the IRQ was successfully assigned to the specified interrupt handler routine. Non-zero result codes may be interpreted as follows:

`-EINVAL` Either the IRQ requested was larger than 15, or a `NULL` pointer was passed instead of a valid pointer to the interrupt handler routine.

`-EBUSY` The IRQ requested has already been allocated to another interrupt handler. This situation should never occur, and is reasonable cause for a call to `panic()`.

The kernel uses an Intel "interrupt gate" to set up IRQ handler routines requested via the `irqaction()` function. The Intel i486 manual [Int90, p. 9-11] explains the interrupt gate as follows:

> Interrupts using...interrupt gates...cause the TF flag [trap flag] to be cleared after its current value is saved on the stack as part of the saved contents of the EFLAGS register. In so doing, the processor prevents

> instruction tracing from affecting interrupt response. A subsequent IRET [interrupt return] instruction restores the TF flag to the value in the saved contents of the EFLAGS register on the stack.
>
> ... An interrupt which uses an interrupt gate clears the IF flag [interrupt-enable flag], which prevents other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores the IF flag to the value in the saved contents of the EFLAGS register on the stack.

Requesting the DMA channel Some SCSI host adapters use DMA to access large blocks of data in memory. Since the CPU does not have to deal with the individual DMA requests, data transfers are faster than CPU-mediated transfers and allow the CPU to do other useful work during a block transfer (assuming interrupts are enabled).

The host adapter will use a specific DMA channel. This DMA channel will be determined by the `detect()` function and requested from the kernel with the `request_dma()` function. This function takes the DMA channel number as its only parameter and returns zero if the DMA channel was successfully allocated. Non-zero results may be interpreted as follows:

`-EINVAL` The DMA channel number requested was larger than 7.

`-EBUSY` The requested DMA channel has already been allocated. This is a very serious situation, and will probably cause any SCSI requests to fail. It is worthy of a call to `panic()`.

5.2.2 `info()`

The `info()` function merely returns a pointer to a static area containing a brief description of the low-level driver. This description, which is similar to that pointed to by the `name` variable, will be printed at boot time.

5.2.3 `queuecommand()`

The `queuecommand()` function sets up the host adapter for processing a SCSI command and then returns. When the command is finished, the `done()` function is called with the `Scsi_Cmnd` structure pointer as a parameter. This allows the SCSI command to be executed in an interrupt-driven fashion. Before returning, the `queuecommand()` function must do several things:

1. Save the pointer to the `Scsi_Cmnd` structure.
2. Save the pointer to the `done()` function in the `scsi_done()` function pointer in the `Scsi_Cmnd` structure. See section 5.2.4 for more information.

3. Set up the special `Scsi_Cmnd` variables required by the driver. See section 6 for detailed information on the `Scsi_Cmnd` structure.

4. Start the SCSI command. For an advanced host adapter, this may be as simple as sending the command to a host adapter "mailbox." For less advanced host adapters, the ARBITRATION phase is manually started.

The `queuecommand()` function is called *only* if the `can_queue` variable (see section 5.1.2) is non-zero. Otherwise the `command()` function is used for all SCSI requests. The `queuecommand()` function should return zero on success (the current high-level SCSI code presently ignores the return value).

5.2.4 done()

The `done()` function is called after the SCSI command completes. The single parameter that this command requires is a pointer to the same `Scsi_Cmnd` structure that was previously passed to the `queuecommand()` function. Before the `done()` function is called, the `result` variable must be set correctly. The `result` variable is a 32 bit integer, each byte of which has specific meaning:

Byte 0 (LSB) This byte contains the SCSI STATUS code for the command, as described in section 2.1.

Byte 1 This byte contains the SCSI MESSAGE, as described in section 2.1.

Byte 2 This byte holds the host adapter's return code. The valid codes for this byte are given in `scsi.h` and are described below:

`DID_OK`	No error.
`DID_NO_CONNECT`	SCSI SELECTION failed because there was no device at the address specified.
`DID_BUS_BUSY`	SCSI ARBITRATION failed.
`DID_TIME_OUT`	A time-out occurred for some unknown reason, probably during SELECTION or while waiting for RESELECTION.
`DID_BAD_TARGET`	The SCSI ID of the target was the same as the SCSI ID of the host adapter.
`DID_ABORT`	The high-level code called the low-level `abort()` function (see section 5.2.6).
`DID_PARITY`	A SCSI PARITY error was detected.

DID_ERROR	An error occurred which lacks a more appropriate error code (for example, an internal host adapter error).
DID_RESET	The high-level code called the low-level `reset()` function (see section 5.2.7).
DID_BAD_INTR	An unexpected interrupt occurred *and* there is no appropriate way to handle this interrupt.

Note that returning `DID_BUS_BUSY` will force the command to be retried, whereas returning `DID_NO_CONNECT` will abort the command.

Byte 3 (MSB) This byte is for a high-level return code, and should be left as zero by the low-level code.

Current low-level drivers do not uniformly (or correctly) implement error reporting, so it may be better to consult `scsi.c` to determine exactly how errors should be reported, rather than exploring existing drivers.

5.2.5 `command()`

The `command()` function processes a SCSI command and returns when the command is finished. When the original SCSI code was written, interrupt-driven drivers were not supported. The old drivers are much less efficient (in terms of response time and latency) than the current interrupt-driven drivers, but are also much easier to write. For new drivers, this command can be replaced with a call to the `queuecommand()` function, as demonstrated in Figure 5.[18]

The return value is the same as the `result` variable in the `Scsi_Cmnd` structure. Please see sections 5.2.4 and 6 for more details.

5.2.6 `abort()`

The high-level SCSI code handles all timeouts. This frees the low-level driver from having to do timing, and permits different timeout periods to be used for different devices (e.g., the timeout for a SCSI tape drive is nearly infinite, whereas the timeout for a SCSI disk drive is relatively short).

The `abort()` function is used to request that the currently outstanding SCSI command, indicated by the `Scsi_Cmnd` pointer, be aborted. After setting the `result` variable in the `Scsi_Cmnd` structure, the `abort()` function returns zero. If `code`, the second parameter to the `abort()` function, is zero, then `result` should be set to `DID_ABORT`. Otherwise, `result` shoudl be set equal to `code`. If `code` is not zero, it is usually `DID_TIME_OUT` or `DID_RESET`.

[18] LINUX 0.99.5 kernel, `linux/kernel/blk_drv/scsi/aha1542.c`, written by Tommy Thorn.

```
static volatile int internal_done_flag    = 0;
static volatile int internal_done_errcode = 0;
static void         internal_done( Scsi_Cmnd *SCpnt )
{
  internal_done_errcode = SCpnt->result;
  ++internal_done_flag;
}

int aha1542_command( Scsi_Cmnd *SCpnt )
{
  aha1542_queuecommand( SCpnt, internal_done );

  while (!internal_done_flag);
  internal_done_flag = 0;
  return internal_done_errcode;
}
```

Figure 5: Example `command()` Function

Currently, none of the low-level drivers is able to correctly abort a SCSI command. The initiator should request (by asserting the `-ATN` line) that the target enter a MESSAGE OUT phase. Then, the initiator should send an ABORT message to the target.

5.2.7 reset()

The `reset()` function is used to reset the SCSI bus. After a SCSI bus reset, any executing command should fail with a `DID_RESET` result code (see section 5.2.4).

Currently, none of the low-level drivers handles resets correctly. To correctly reset a SCSI command, the initiator should request (by asserting the `-ATN` line) that the target enter a MESSAGE OUT phase. Then, the initiator should send a BUS DEVICE RESET message to the target. It may also be necessary to initiate a SCSI RESET by asserting the `-RST` line, which will cause all target devices to be reset. After a reset, it may be necessary to renegotiate a synchronous communications protocol with the targets.

5.2.8 slave_attach()

The `slave_attach()` function is *not* currently implemented. This function would be used to negotiate synchronous communications between the host adapter and the target drive. This negotiation requires an exchange of a pair of SYNCHRONOUS DATA TRANSFER REQUEST messages between the initiator and the target. This exchange should occur under the following conditions [LXT91]:

> A SCSI device that supports synchronous data transfer recognizes it has not communicated with the other SCSI device since receiving the last "hard" RESET.
>
> A SCSI device that supports synchronous data transfer recognizes it has not communicated with the other SCSI device since receiving a BUS DEVICE RESET message.

5.2.9 bios_param()

Linux supports the MS-DOS[19] hard disk partitioning system. Each disk contains a "partition table" which defines how the disk is divided into logical sections. Interpretation of this partition table requires information about the size of the disk in terms of cylinders, heads, and sectors per cylinder. SCSI disks, however, hide their physical geometry and are accessed logically as a contiguous list of sectors. Therefore, in order to be compatible with MS-DOS, the SCSI host adapter will "lie" about its geometry. The physical geometry of the SCSI disk, while available, is seldom used as the "logical geometry." (The reasons for this involve archaic and arbitrary limitations imposed by MS-DOS.)

Linux needs to determine the "logical geometry" so that it can correctly modify and interpret the partition table. Unfortunately, there is no standard method for converting between physical and logical geometry. Hence, the `bios_param()` function was introduced in an attempt to provide access to the host adapter geometry information.

The `size` parameter is the size of the disk in sectors. Some host adapters use a deterministic formula based on this number to calculate the logical geometry of the drive. Other host adapters store geometry information in tables which the driver can access. To facilitate this access, the `dev` parameter contains the drive's device number. Two macros are defined in `linux/fs.h` which will help to interpret this value: `MAJOR(dev)` is the device's major number, and `MINOR(dev)` is the device's minor number. These are the same major and minor device numbers used by the standard Linux `mknod` command to create the device in the `/dev` directory. The `info` parameter points to an array of three integers that the `bios_param()` function will fill in before returning:

[19] MS-DOS is a registered trademark of Microsoft Corporation.

`info[0]` Number of heads

`info[1]` Number of sectors per cylinder

`info[2]` Number of cylinders

The information in `info` is *not* the physical geometry of the drive, but only a *logical* geometry that is identical to the *logical* geometry used by MS-DOS to access the drive. The distinction between physical and logical geometry cannot be overstressed.

6 The `Scsi_Cmnd` Structure

The `Scsi_Cmnd` structure,[20] as shown in Figure 6, is used by the high-level code to specify a SCSI command for execution by the low-level code. Many variables in the `Scsi_Cmnd` structure can be ignored by the low-level device driver—other variables, however, are extremely important.

6.1 Reserved Areas

6.1.1 Informative Variables

`host` is an index into the `scsi_hosts` array.

`target` stores the SCSI ID of the target of the SCSI command. This information is important if multiple outstanding commands or multiple commands per target are supported.

`cmnd` is an array of bytes which hold the actual SCSI command. These bytes should be sent to the SCSI target during the COMMAND phase. `cmnd[0]` is the SCSI command code. The `COMMAND_SIZE` macro, defined in `scsi.h`, can be used to determine the length of the current SCSI command.

`result` is used to store the result code from the SCSI request. Please see section 5.2.4 for more information about this variable. This variable *must* be correctly set before the low-level routines return.

6.1.2 The Scatter-Gather List

`use_sg` contains a count of the number of pieces in the scatter-gather chain. If `use_sg` is zero, then `request_buffer` points to the data buffer for the SCSI command, and `request_bufflen` is the length of this buffer in bytes. Otherwise, `request_buffer` points to an array of `scatterlist` structures, and `use_sg` will indicate how many such structures are in the array. The use of `request_buffer` is non-intuitive and confusing.

[20] Linux 0.99.7 kernel, `linux/kernel/blk_drv/scsi/scsi.h`

```
typedef struct scsi_cmnd
{
  int               host;
  unsigned char     target,
                    lun,
                    index;
  struct scsi_cmnd *next,
                   *prev;

  unsigned char     cmnd[10];
  unsigned          request_bufflen;
  void             *request_buffer;

  unsigned char     data_cmnd[10];
  unsigned short    use_sg;
  unsigned short    sglist_len;
  unsigned          bufflen;
  void             *buffer;

  struct request    request;
  unsigned char     sense_buffer[16];
  int               retries;
  int               allowed;
  int               timeout_per_command,
                    timeout_total,
                    timeout;
  unsigned char     internal_timeout;
  unsigned          flags;

  void (*scsi_done)(struct scsi_cmnd *);
  void (*done)(struct scsi_cmnd *);

  Scsi_Pointer      SCp;
  unsigned char    *host_scribble;
  int               result;

} Scsi_Cmnd;
```

Figure 6: The Scsi_Cmnd Structure

Each element of the `scatterlist` array contains an `address` and a `length` component. If the `unchecked_isa_dma` flag in the `Scsi_Host` structure is set to 1 (see section 5.1.7 for more information on DMA transfers), the address is guaranteed to be within the first 16 MB of physical memory. Large amounts of data will be processed by a single SCSI command. The length of these data will be equal to the sum of the lengths of all the buffers pointed to by the `scatterlist` array.

6.2 Scratch Areas

Depending on the capabilities and requirements of the host adapter, the scatter-gather list can be handled in a variety of ways. To support multiple methods, several scratch areas are provided for the exclusive use of the low-level driver.

6.2.1 The scsi_done() Pointer

This pointer should be set to the `done()` function pointer in the `queuecommand()` function (see section 5.2.3 for more information). There are no other uses for this pointer.

6.2.2 The host_scribble Pointer

The high-level code supplies a pair of memory allocation functions, `scsi_malloc()` and `scsi_free()`, which are guaranteed to return memory in the first 16 MB of physical memory. This memory is, therefore, suitable for use with DMA. The amount of memory allocated per request *must* be a multiple of 512 bytes, and *must* be less than or equal to 4096 bytes. The total amount of memory available via `scsi_malloc()` is a complex function of the `Scsi_Host` structure variables `sg_tablesize`, `cmd_per_lun`, and `unchecked_isa_dma`.

The `host_scribble` pointer is available to point to a region of memory allocated with `scsi_malloc()`. The low-level SCSI driver is responsible for managing this pointer and its associated memory, and should free the area when it is no longer needed.

6.2.3 The Scsi_Pointer Structure

The `SCp` variable, a structure of type `Scsi_Pointer`, is described in Figure 7. The variables in this structure can be used in *any* way necessary in the low-level driver. Typically, `buffer` points to the current entry in the `scatterlist`, `buffers_residual` counts the number of entries remaining in the `scatterlist`, `ptr` is used as a pointer into the buffer, and `this_residual` counts the characters remaining in the transfer. Some host adapters require support of this detail of interaction—others can completely ignore this structure.

```
typedef struct scsi_pointer
{
  char               *ptr;             /* data pointer */
  int                this_residual;    /* left in this buffer */
  struct scatterlist *buffer;          /* which buffer */
  int                buffers_residual; /* how many buffers left */

  volatile int       Status;
  volatile int       Message;
  volatile int       have_data_in;
  volatile int       sent_command;
  volatile int       phase;
} Scsi_Pointer;
```

Figure 7: The `Scsi_Pointer` Structure

The second set of variables provide convenient locations to store SCSI status information and various pointers and flags.

7 Acknowledgements

Thanks to Drew Eckhardt, Michael K. Johnson, Karin Boes, Devesh Bhatnagar, and Doug Hoffman for reading early versions of this paper and for providing many helpful comments. Special thanks to my official COMP-291 (Professional Writing in Computer Science) "readers," Professors Peter Calingaert and Raj Kumar Singh.

References

[ANS] *Draft Proposed American National Standard for Information Systems: Small Computer System Interface – 2 (SCSI-2)*. Irvine, California: Global Engineering Documents. (X3T9.2/86-109, revision 10h, October 17, 1991).

[ES91] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1991.

[Int90] Intel. *i486 Processor Programmer's Reference Manual*. Intel/McGraw-Hill, 1990.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice Hall, 2nd edition, 1988.

[LXT91] *LXT SCSI Products: Specification and OEM Technical Manual*. San Jose, California: Maxtor Corporation, 1991.

[Nor85] Peter Norton. *The Peter Norton Programmer's Guide to the IBM PC*. Bellevue, Washington: Microsoft Press, 1985.