University of Montana
ScholarWorks at University of Montana

Theses, Dissertations, Professional Papers Graduate School

1993

Design of the software interface for a multimaster bus system.

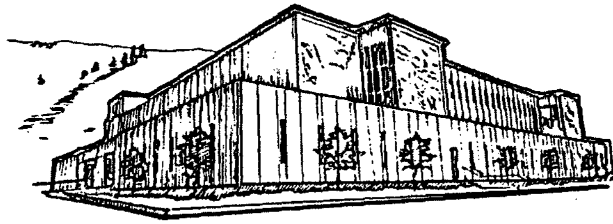
Jie. Yang
The University of Montana

Follow this and additional works at: http://scholarworks.umt.edu/etd

Recommended Citation
Yang, Jie., "Design of the software interface for a multimaster bus system." (1993). *Theses, Dissertations, Professional Papers.* Paper 5118.

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Theses, Dissertations, Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mail.lib.umt.edu.

JIE YANG



Maureen and Mike
MANSFIELD LIBRARY

The University of
Montana

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

*** Please check "Yes" or "No" and provide signature***

Yes, I grant permission X
No, I do not grant permission ___

Author's Signature [signature]

Date: 1/21/94

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.

The Design of the Software

Interface for a Multimaster Bus System

By

Jie Yang

Presented in partial fulfillment of the requirements

for the degree of

Master of Science

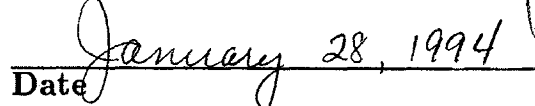
University of Montana

1993

Approved by

Chairman, Board of Examiners

Dean, Graduate School



Date

UMI Number: EP40582

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI EP40582

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

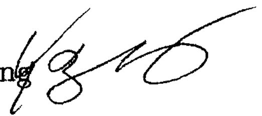
gift 1-7-94

Jie Yang., M.S., December 1993 Computer Science

The Design of the Software Interface for a Multimaster Bus System

(44 pp.)

Director: Dr. Youlu Zheng



Abstract

This work deals with the design and implementation of the software interface for the Small Computer System Interface (SCSI) system, which is a parallel, multimaster I/O bus that provides a standard interface between computer and peripherial devices, and SCSI device drivers on the BSD/386 Operating System.

The SCSI is a bus structure that expects intelligence in all the devices it connects, and works with those devices in terms of the data to be exchanged rather than low-level hardware functions. This frees up resources on the BSD/386 and other UNIX systems, which are quite happy to leave the hardware-level details up to peripheral intelligence.

Compared with the protocol of conventional device driver, the protocol of SCSI device driver treats peripherials as logical devices that use a well defined set of commands, which eliminates hardware incompatibilities. It is possible for one SCSI bus to communicate up to seven physical devices through a single SCSI device driver.

Table of Contents

1. Project Formulation **1**

1.1. **Multimaster Bus – SCSI** 1

1.2. **An Introduction To The Project** 3

2. BSD/386 Operating System **5**

2.1. **Overall Structure of the BSD/386 Operating System** 5

2.2. **Buffer Cache** 8

2.3. **Buffer Cache Interface** 10

2.4. **Device Driver** 12

2.5. **System Configuration** 15

3. Basic Concept of SCSI Host Bus **17**

3.1. **What is SCSI?** 17

3.2. **SCSI Phases** 18

3.3. **Transaction** 22

3.4. **SCSI Commands** 23

4. The Design of SCSI Host Adapter Driver Interface **27**

4.1. **Basic Requirements of The SCSI Device Driver** 27

4.2. **Specification of SCSI** 27

4.3. **BSD/386 SCSI Host Adapter Driver** 28

4.4. **Overall Structure** 29

5. The Implementation of the SCSI Host Adapter Driver Interface **31**

5.1. **The Configuration Driver Structure** 31

5.1.1. **cd_devs** 32

5.1.2. **cd_name** 32
5.1.3. **cd_devsize** 34
5.1.4. **Function xxprobe()** 34
5.1.5. **Function xxattach()** 35
5.2. **SCSI Host Bus Adapter Driver Structure** 37
5.2.1. **hd_icmd – SCSI Immediate Command Function** 38
5.2.2. **hd_dump – SCSI Dumping Function** 38
5.2.3. **hd_start – Restarting SCSI Bus Function** 38
5.2.4. **hd_go – Start I/O Transfer Function** 40
5.2.5. **hd_reset – Reseting SCSI Adapter Function** 40

6. Conclusion **41**
6.1. **Optimization** 41
6.1.1. **The Third Generation SCSI** 41
6.1.2. **Object-Oriented Host Adapter Driver** 42
6.1.3. **The Standardized Design and Implementation** 42

List of Figures

1 Block Diagram of the System Kernel 6

2 Driver Entry Point 13

3 SCSI Bus Phases 20

4 The 6–Byte Command Descriptor Block 24

5 SCSI Device Driver Interface 30

6 The data structure of configuration driver 32

7 Struct fdomain_softc 33

8 The `intr_establish()` function 36

9 The data structure of SCSI Host Adapter Driver 39

10 The class of the SCSI Host Adapter Driver 43

List of Tables

1 Buffer Head Descriptions 9

2 SCSI Bus Phase Determination 18

3 SCSI Status Codes 21

4 Sense Key Descriptions 25

Acknowledgements

I would like to offer special thanks to Dr. Youlu Zheng, Dr. James R. Ullrich, and Dr. weiming Hao, the members of my project committee. The time they have spent advising me and proofing drafts could have been spent in more pleasant ways.

Thanks go to the Department of Computer Science for purchasing the 486 system for my experiement, and thanks also go to BSDI for providing BSD/386 Operating System and the Future Domain SCSI Board for this project.

I should thank all of those who have helped me along the way to where I am now. Parents, professors, instructors, wonderful friends in the LINUX world and fellow students have all influenced my education and are all due this thanks; without them I would be a very different person.

1. Project Formulation

1.1. Multimaster Bus – SCSI

A bus is physically embodied in the connectors that carry its signals, and the logic on each board that implements the bus protocol and connection. Essentially, there are three major types of buses: The system bus, the I/O bus, and the memory bus. Personal computers often use only I/O bus, with the CPU and memory having a close non-bus connection.

In any I/O bus transaction, there is a master and a slave. The master initiates the transaction, and the slave responds. In the XT and AT buses, there is generally only one master, the motherboard. The ESIA[1] and SCSI[2] buses are multimaster buses. The basic use of the multimaster capability is to allow I/O cards to perform true direct memory access (DMA) and to access data from main memory of the central processor independently .

XT and AT buses are 8-bit or 16-bit bus. These buses lack auto-configuration and high-performance multimaster capabilities.

In comparison with a single master bus, such as an XT or AT bus, SCSI is a parallel, multimaster I/O that provides a standard interface between computer and peripheral devices. Despite its misleading name[3], SCSI is fast becoming the method of choice for connecting disks, tape drives, CD-ROMs, WORM (write once, read many times) drives, and communication devices to computers of all sizes. The SCSI gets all devices to cooperate with each other.

Like Micro Channel, VME, and NuBus, SCSI is a true bus in the sense that it defines standard physical and electrical connections for devices. The fundamental and

[1]**ESIA** is Extended Industry Standard Architecture

[2]**SCSI** is Small Computer System Interface

[3]**SCSI** incorrectly implies that it is useful only for small computer

distinguishing difference between SCSI and the other buses is that SCSI facilitates the use of many diverse peripherals. Its communication protocols treat peripherals as logical devices that use a defined set of commands, which eliminates hardware incompatibilities. It is possible for a SCSI bus to communicate, through a single device driver, to many physical devices (up to seven).

The SCSI specification outlines several commands for many device types: Hard disk (random access), tape (sequential access), printers, and others. The specification includes mandatory, optional, and vendor-unique commands. The SCSI bus treats a hard disk as a defined-capacity random-access device that responds to standard read, write and format commands. A built-in SCSI controller translates the commands into interface specific control signals.

The SCSI hardware specification description details the physical characteristics, such as the cable, signal pin-outs, connector types, and so on. There are two commonly used types of SCSI connectors: A standard 50-pin cable for connecting internal peripherals, and a centronics-style 50-pin cable for connecting external devices.

The SCSI lets the physical bus exceed the bound of a typical microcomputer chassis. You can "stretch" the SCSI bus via a cable of up to 6 meters in length by using single-ended drivers, or via a cable of up to 25 meters in length by using differential drivers. This eliminates the problem of trying to pack every peripheral into the computer. It also allows you to connect large peripheral devices, such as scanners, that wouldn't fit otherwise.

This combination of connectivity and economy of hardware delivers many capabilities to users with demanding I/O needs. SCSI's standard connectors allow easy connection to many peripherals, while its standard command set simplifies the system design. The end user can select SCSI peripherals from several different vendors with the assurance that they will be compatible. SCSI's versatile nature has helped it to emerge as a standard on most companies' system-level and storage peripheral

products.

Command queuing is another important SCSI feature. If one or more hosts make many requests for a peripheral device, that peripheral (if it implements queuing) can queue up commands for later execution. This allows a controller to optimize I/O by implementing strategies such as elevator seeks algorithm.

1.2. An Introduction To The Project

Low-cost PCs are making UNIX hotter than ever. Prices for IBM-compatible desktop computers have been dropping dramatically in recent years. With prices plummeting and performance zooming, there has never been a better time to put UNIX, the multitasking and multiuser 32-bit Operating System, on a PC.

Much of the advantage of running UNIX on a PC comes from the nature of the hardware itself. Unlike RISC workstations, PCs are standardized, high-volume commodity products. PC hardware from one vendor can generally be swapped with that of any of hundreds of other competing vendors without a problem. However, PC-related peripherals are not absolutely interchangeable: Many network adapters and multiport boards require customized device drivers, for example. But with the right device drivers, the same add-in boards will work in almost any PC.

For computer science students who want to delve deeply – but inexpensively – into the internals of a real multitasking system, PC-UNIX is a good choice.

More and more companies are working on the PC-UNIX. Currently, there are more than 15 versions of PC-UNIX, including BSD/386.

The BSD/386 is a UNIX[4]-like operating system for 386 and i486 system.[5] It is based on the Net2 distribution of BSD 4.3 which does not require a UNIX source

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

[5]386 and i486 are trademarks of Intel Corporation.

code license from USL[6]. BSD/386 provides support for a variety of I/O devices and the installation is reasonably easy. The complete source code of the UNIX operating system from the BSDI[7] makes it possible for one to dig into the kernel of UNIX, study, implement and develop some part of the kernel. The code is also very helpful for establishing an operating system course lab. The students not only have an opportunity to get the secret of the implementation of the UNIX operating system, but also get a chance to learn how to manage and maintain the UNIX system.

The current BSD/386 is still a gamma version, that means there are many bugs and problems that need to be solved. The purpose of the project is to study the kernel of the BSD/386, and improve its I/O Subsystem through the design and implementation of the software interface for a popular multimaster bus - the Future Domain TMC18c50 SCSI Board. After the project is finished and refined, the resulting software and documentation will be included in the new release of BSD/386 by BSDI.

[6]USL is UNIX System Laboratories.

[7]BSDI is Berkeley Software Design, Inc.

2. BSD/386 Operating System

2.1. Overall Structure of the BSD/386 Operating System

In the overall architecture of the UNIX system, the hardware provides the operating system with basic services. The operating system interacts directly with the hardware, providing common services to programs and insulating them from hardware idiosyncrasies. Viewing the system as a set of layers, the operating system is commonly called the system kernel, or the kernel, emphasizing its isolation from user programs. User level programs such as the shell and editors interact with the kernel by invoking a well defined set of system calls. The system calls instruct the kernel to do various operations for the calling program and exchange data between the kernel and the program.

Figure 1 shows three levels: User, kernel, and hardware. The system call and library interface represent the border between user programs and the kernel. System calls look like regular function calls in C programs, and libraries map these function calls to the primitives needed to enter the operating system.

The UNIX system (kernel) supports the illusions that the file system has "places" and that processes have "life". The two entities, file and process, are the two central concepts in the UNIX system model. Figure 1 gives a block diagram of the kernel, showing various modules and their relationships to each other. In particular, it shows the file subsystem on the left and the process control subsystem on the right, the two major components of the kernel.

The *process control subsystem* is responsible for process synchronization, interprocess communication, process scheduling, and memory management.

BSD/386 supports a multitasking environment. Each task of execution is termed a *process*. The *context* of a UNIX process consists of user-level state, including the contents of its address space and the run-time environment, and kernel-state, which

INTRODUCTION TO THE KERNEL

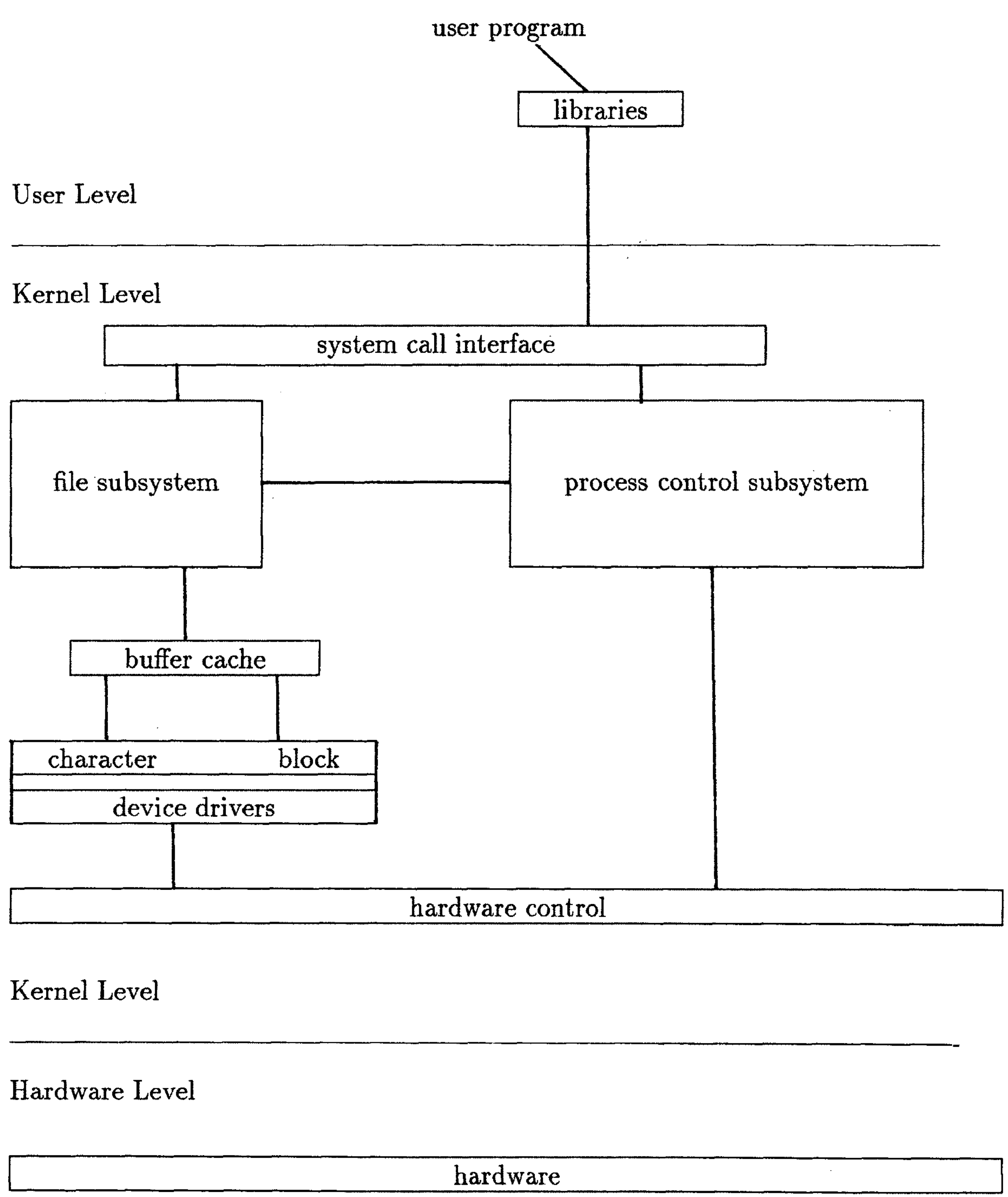


Figure 1: Block Diagram of the System Kernel

includes scheduling parameters, resource control, and identification information. The context includes everything used by the kernel in providing services for the process. Users can create processes, control a process' execution, and receive notification when the process' execution status changes. Every process is assigned a unique value, termed a *process identifier* (PID). This value is used by the kernel to identify a process when reporting status changes to a user, and by a user when referencing a process in a system call.

The file subsystem manages files, allocates file space, administers free space, controls access to files, and retrieves data for users. A regular file is a linear array of bytes, and can be read and written starting at any location in the file. The kernel distinguishes no record boundaries in regular files, although many programs recognize line-feed characters as distinguishing the ends of lines, and other programs may impose other structure. No system-related information about a file is kept in the file itself, but the file system stores a small amount of ownership, protection, and usage information with each file.

A filename component is a string of up to 255 characters. These filenames are stored in a type of file called a directory. The information in a directory about a file is called a directory entry and includes, in addition to the filename, a pointer to the file itself. Directory entries may refer to other directories as well as to plain files. A hierarchy of directories and files is thus formed, and is called a file system. A file system may include not only plain files and directories, but also references to other objects, such as devices and sockets.

The file system implementation converts the user abstraction of a file as an array of bytes into a structure imposed by the underlying physical medium. Although the user may wish to write a single byte to a file, the disk can read and write only in multiples of sectors. Here, the system must arrange to read in the sector containing the byte to be modified, to replace the affected byte, and to write the sector back to

the disk. This operation of converting random access to an array of bytes into reads and writes of disk sectors is known as block I/O.

First, the system breaks the user's request into a set of operations to be done on each logical block of the file. Logical blocks describe block-sized pieces of a file. The system calculates the logical blocks by dividing the array of bytes into filesystem-sized pieces. Thus, if a filesystem's block size is 8192 bytes, then logical block 0 would contain bytes 0 to 8192, logical block 1 would contain bytes 8192 to 16,383, and so on. If the user's request is incomplete, the process is repeated with the next logical block of the file.

The data in each logical block is stored in a physical block on the disk. A physical block is the location on the disk to which the system maps a logical block. A physical disk block is constructed from one or more contiguous sectors. For a disk with 512-byte sectors, an 8192-byte file system block would be built up from 16 contiguous sectors. Although the contents of a logical block are contiguous on disk, the logical blocks of the file need not be laid out contiguously.

The file subsystem accesses file data using a buffering mechanism that regulates data flow between the kernel and secondary storage devices. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel.

2.2. Buffer Cache

The kernel could read and write directly to and from the disk for all file system accesses, but system response time and throughput would be poor because of the slow disk transfer rate. The kernel therefore attempts to minimize the frequency of disk access by keeping a pool of internal data buffers, called the buffer cache[8], which contains the data in recently used disk blocks.

[8] The buffer cache discussed here is a software structure that should not be confused with hardware caches that speed memory references

Figure 1 shows that the position of the buffer cache module in the kernel architecture is between the file subsystem and (block) device drivers. When reading data from the disk, the kernel attempts to read from the buffer cache. If the data is already in the cache, the kernel does not have to read from the disk. If the data is not in the cache, the kernel reads the data from the disk and caches it, using an algorithm that tries to save as much good data in the cache as possible. Similarly, data being written to disk is cached so that it will be there if the kernel later tries to read it. The kernel also attempts to minimize the frequency of disk write operations by determining whether the data must actually be stored on disk or whether it is transient data that will soon be overwritten. Higher-level kernel algorithms instruct the buffer cache module to pre-cache data or to delay-write data to maximize the caching effect. Depending on available memory, a system may be configured with anything from a hundred to a thousand buffers. The larger the number of buffers, the longer a given disk block can be retained in memory, and the greater the chance that disk I/O can be avoided.

HASH LINK
QUEUE LINK
FLAGS
DEVICE
BLOCK NUMBER
BYTE COUNT
BUFFER SIZE
BUFFER POINTER TO BUFFER CONTENTS (8K)

Table 1: Buffer Head Descriptions

Table 1 shows the format of a buffer. The buffer is composed of two parts. The

first part is the buffer header, which contains information used to find the buffer and to describe its contents. The content information includes the device (i.e., disk and partition on that disk), the starting physical block number (counting from the beginning of the partition), and the number of bytes contained in the buffer. The flag's entry tracks status information about the buffer, such as whether the buffer contains useful data, whether the buffer is in use, and whether the data must be written back to the disk before the buffer can be reused.

The second part is the actual buffer contents. There is a pointer to the data and a field that shows the size of the data buffer. The buffer size is always at least as big as the fragment size. Data is maintained separately from the header to allow easy manipulation of the buffer sizes with the page-mapping hardware. If the headers were prepended, either each header would have to be on a page by itself or the kernel would have to avoid remapping buffer pages that contained headers.

The size of buffer requests from the file system range from 512-byte fragments up to 8192-byte full-sized blocks. If many small files are being accessed, then many small buffers are needed. Alternatively, if several large files are being accessed, then fewer large buffers are needed. To allow the system to adapt efficiently to these changing needs, each buffer is allocated 8192 bytes of virtual memory, but the address space is not fully populated with physical memory. Initially, each buffer is assigned 2048 bytes of physical memory. As smaller buffers are allocated, they give up their excess physical memory to other buffers that need to hold more than 2048 bytes.

2.3. Buffer Cache Interface

The internal kernel interface to the buffer pool is simple. The file system allocates and fills buffers by calling the *bread()* routine. *Bread()* takes a device, a block number, and a size, and returns a pointer to a locked buffer. Any other process that tries to access the buffer will be put to sleep until the buffer is released. A buffer can be

released in one of four ways. If the buffer has not been modified, it can simply be released through use of *brels()*. If the buffer has been modified, it is called dirty. Dirty buffers must eventually be written back to the disk. Three routines are available based on the urgency with which the data must be written to disk. In the typical case, *bdwrite()* is used; it assumes that the buffer probably will be modified again soon, so should be marked as dirty, but should not be immediately written to disk. The heuristic is that, if the buffer will be modified again soon, the disk I/O would be wasted. Because the buffer is held for an average of 15 seconds before it is written, a process doing many small writes will not repeatedly access the disk. If the buffer has been completely filled, then it is unlikely to be written again soon, so it should be released with *bawrite()*. *Bawrite()* schedules a write to disk, but allows the caller to continue running while the output completes. The final case is *bwrite()*, which ensures that the disk write is complete before proceeding. Because this mechanism can introduce a long latency to the requester, it is used only when the process explicitly requests the behavior (such as the *fsync* system call), or when the operation is critical to ensure the consistency of the file system in case of a system crash.

The primary interface to getting a buffer is through *bread()*, which is called with a request for a data block of a specified size on a specified disk. *Bread()* first calls *getblk()* to find out whether the data block is available in a buffer that is already in memory. If the block is available in a buffer, *getblk()* calls *notavail()* to take the buffer off whichever free list it is on and to mark it busy; *bread()* can then return the buffer to the caller.

If the disk block is not already in memory, *getblk()* calls *newbuf()* to allocate a new buffer. The new buffer is then passed to *brealloc()* to ensure that there will not be any overlap with other buffers. Next, the buffer is passed to *allocbuf()*, which ensures that the buffers has the right amount of physical memory. *Getblk()* then returns the buffer to *bread()* marked busy and unfilled. Noticing that the buffer is

unfilled, *bread()* passes it to the disk driver to have the data read in. When the disk read completes, the buffer can be returned.

To maintain the consistency of the file system, *brealloc()* ensures that a disk block is mapped into at most one buffer. The final task in allocating a buffer is to ensure that the buffer has enough physical memory allocated to it; this task is handled by *allocbuf()*.

2.4. Device Driver

A hardware device is a peripheral such as a disk or tape drive, terminal multiplexer, or network controller. For each hardware device supported by UNIX, a software module termed a device driver is required. The device driver provides a consistent interface to various hardware devices by hiding device-specific details from the rest of the UNIX kernel and user applications.

UNIX must provide two internal interfaces that a device driver may support. These interfaces permit a device to be used in two different ways: As a block-oriented device suitable for holding one or more file systems, or as an unstructured device that is potentially suitable for use by the terminal I/O system.

A block-device interface, as the name indicates, supports only block-oriented I/O operations. The block-device interface uses the buffer cache to minimize the number of I/O requests that actually require an I/O operation, and to synchronize with file system operations on the same device. All I/O is done to or from I/O buffers that reside in the kernel's address space. This approach requires at least one memory-to-memory copy operation to satisfy a user request, but also allows UNIX to support I/O requests of nearly arbitrary size and alignment. A character-device interface comes in two styles depending on the characteristics of the underlying hardware device.

Internal to the system, I/O devices are accessed through a fixed set of entry points provided by each device's device driver. The set of entry points varies according

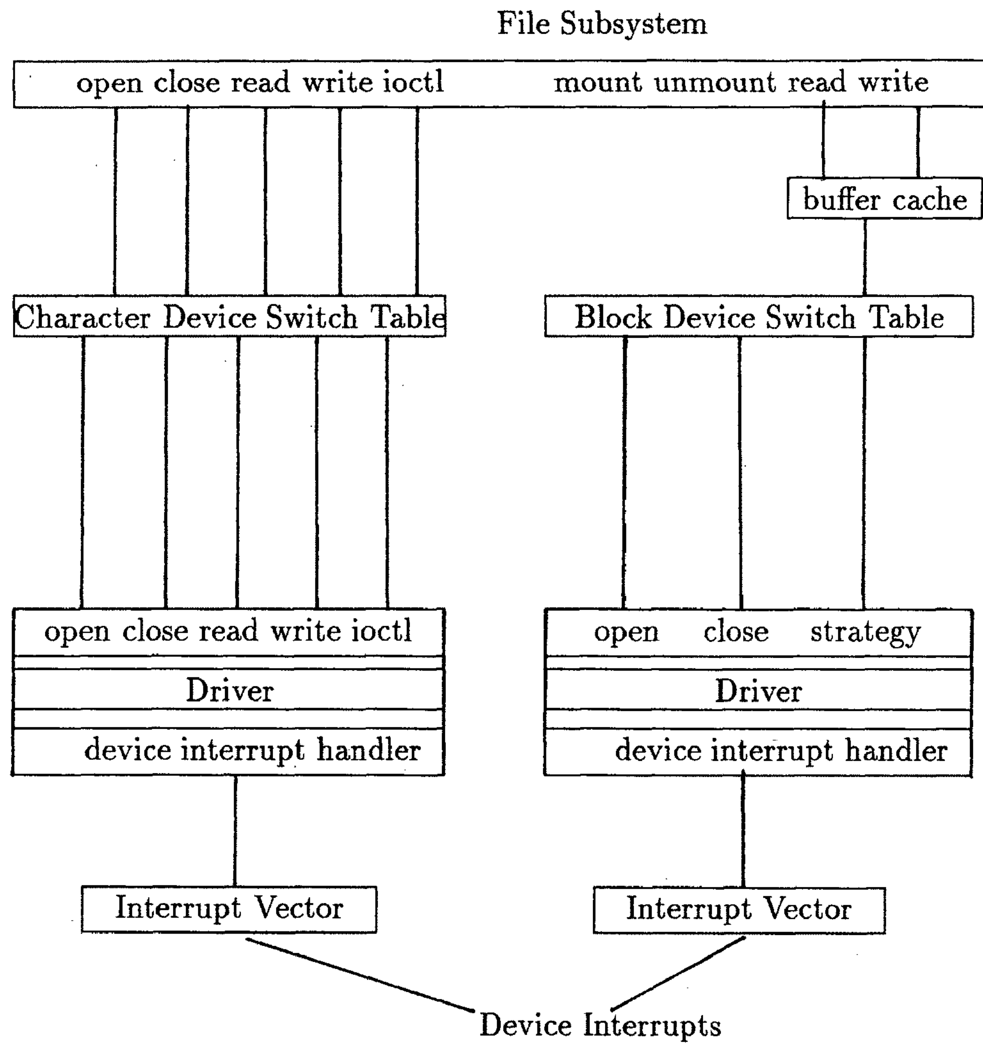


Figure 2: Driver Entry Point

to whether the I/O device supports a block- or character-device interface. For a block-device interface, a device driver is described by a *bdevsw* structure, whereas for character-device interface, it accesses a *cdevsw* structure.

Devices are identified by a device number that is constructed from a major and a minor device number. The major device number uniquely identifies the type of device (actually the device driver) and it is the index of the device's entry in the block- or character-device table. Devices that support both block- and character-device interfaces have two major device numbers, one for each table. The minor device number is interpreted solely by the device driver and is used by the driver to identify to which, of potentially many, hardware devices an I/O request refers. For magnetic tapes, for example, minor device numbers are used to identify a specific controller and tape transport.

Every device driver must have entries in at least one of the switch tables. The position of each driver's entry in the array that forms the table is determined by its major number. There are two data tables called the character device switch table and the block device switch table. Drivers can appear in both tables if the driver is both character and block type. This is usually done to support raw character I/O on a block device.

The kernel to driver interface is described by the block device switch table and the character device switch table (Figure 2). Each device type has entries in the table that direct the kernel to the appropriate driver interfaces for the system calls. The *open* and *close* system calls for a device file funnel through the two device switch tables, according to the file type. The *mount* and *unmount* system calls also invoke the device open and close procedures for block devices. *Read*, *write*, and *ioctl* system calls for character special files pass through the respective procedures in the character device switch table. *Read* and *write* system calls for block devices and for files on mounted file systems invoke the algorithms of the buffer cache, which invoke the

device strategy procedure.

The hardware to driver interface consists of machine-dependent control registers or I/O instructions for manipulating devices and interrupt vectors: When a device interrupt occurs, the system identifies the interrupting device and calls the appropriate interrupt handler. Obviously, software devices such as the kernel profile driver do not have a hardware interface, but other interrupt handlers may call a "software interrupt handler" directly.

Device drivers deal with both I/O bus addresses and virtual addresses. Implementations of BSD/386 use a demand-paged virtual memory system. The addresses used in source and executable code in device drivers and in almost all other kernel and user code are never physical memory addresses. They are virtual addresses, which are translated by hardware at run time into physical memory references. The translations are made with the aid of address translation tables, called page tables, which are created and maintained by the kernel.

The method used by system hardware to translate I/O bus addresses is that CPU references to some portion of physical memory are placed on the particular I/O bus, either as memory or as I/O references. DMA (Direct Memory Access) device references to memory may be translated into physical memory addresses by I/O mapping hardware.

Device drivers execute in system virtual address space; in most cases the relationship of this address space to physical memory is unknown. On the BSD/386 system, explicit mappings must be set up between kernel virtual address space and the I/O bus. Kernel initialization and configuration routines will set up these mappings.

2.5. System Configuration

Device drivers are code modules that must be incorporated into the executable image of the kernel. For a particular device driver, the kernel needs to know what

entry points exist and what those entry points are. The device driver needs to know what physical devices are attached to the system and what their addresses are. This information is stored in a variety of driver-specific data structures.

Although the data structure can be built mostly as C code, with small amounts of machine-specific assembly language for interrupt handlers on some systems, the configuration data structures are normally constructed by a special system configuration utility. The configuration utility takes a description of what device drivers are available and what devices are attached to the system, and then constructs one or more configuration files. These are compiled or assembled, then linked with the main part of the operating system, along with the device drivers, to produce a complete kernel.

System configuration is the procedure by which administrators specify parameters that are installation dependent. Some parameters specify the sizes of kernel tables, such as the process table, inode table, file table, and the number of buffers to be allocated for the buffer pool. Other parameters specify device configuration, telling the kernel which devices are included in the installation and information about their switch tables, address tables, and data structures.

3. Basic Concept of SCSI Host Bus

3.1. What is SCSI?

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.

3.2. SCSI Phases

When the SCSI bus operates, it makes orderly transitions between bus states known as *phases.* The phase determines the direction and content of the data lines. 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 2.

-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 2: SCSI Bus Phase Determination

The eight possible phases are BUS FREE, ARBITRATION, SELECTION, RE SELECTION, COMMAND, DATA, STATUS, and MESSAGE. The last four of these are called *information transfer phases.*

The phase diagram in Figure 3 shows the relationships between the phase and

the possible phase transitions. The system always comes up in the BUS FREE phase or reenters this phase after the bus is reset. A system can be nonarbitrating (a) or arbitrating (b); if arbitration is not implemented, there is no ARBITRATION or RESELECTION phase. Nonarbitrating systems usually consist of a single host and a single peripheral controller; ARBITRATION and RESELECTION are not necessary because the host is always in control, and there is no need for a disconnect/reconnect operation.

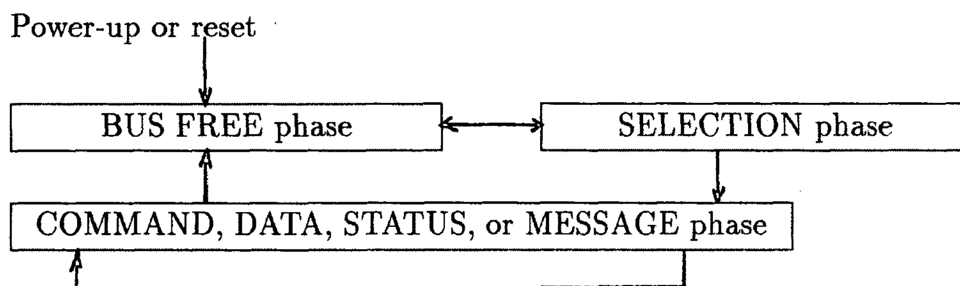
In the BUS FREE phase, the BSY (Busy) signal is not asserted (as it is in all the other phases).

In the ARBITRATION phase, all would-be bus masters compete for control of the bus. This phase begins when an initiator, or a target that wants to get back in touch with an initiator after being disconnected, attempts to gain control of the SCSI bus. Each potential master asserts the BSY signal (which is a wired OR, so there is no electrical conflict) and sets the data bit (0 through 7) corresponding to its SCSI ID. The device with the highest ID wins, and the others then back off.

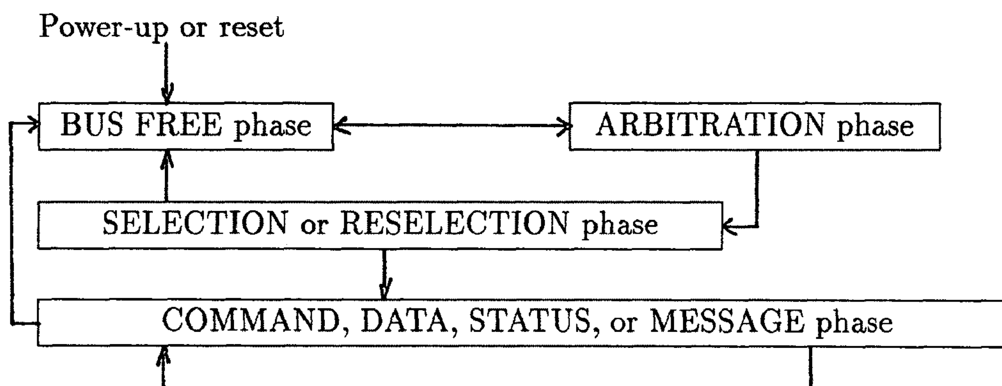
In the SELECTION phase, an initiator selects a target for a command by placing the target's ID on the data lines and asserting the SEL (Select) signal. If the system is nonarbitrating, the initiator does not need to compete for the bus and can skip to this phase from the BUS FREE phase. At the end of this phase, the target (if it exists) takes over control of the bus timing and phase transitions for the remainder of the transaction.

The RESELECTION phase occurs when a target wins the arbitration and reestablishes contact with an initiator that previously sent it a command. The target places the initiator's ID on the data lines and asserts the I/O signal, as well as SEL, to distinguish this phase from a SELECTION phase.

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



(a) SCSI phases without arbitration



(b) SCSI phases with arbitration

Figure 3: SCSI Bus Phases

In the DATA OUT and DATA IN phases, data is 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.

The STATUS 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 3. Note that since bits[9] 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 3: SCSI Status Codes

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

- GOOD : The operation completed successfully.

[9] Bit 0 is the least significant bit.

- CHECK CONDITION : An error occurred. The REQUEST SENSE command should be used to find out more information about the error.
- BUSY : The device was unable to accept a command. This may occur during a self-test or shortly after power-up.

In the 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 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.

The system cycles through one or more information transfer phases. The target uses the MSG (Message), C/D (Control/Data), and I/O signals to guide the system through the phases.

A typical SCSI transaction would consist of a COMMAND phase, followed by a series of DATA IN or DATA OUT phases, followed by a STATUS phase and a MESSAGE IN phase (in which the target sends the mandatory "Command Complete" message). However, the initiator can cause the target to enter the MESSAGE OUT phase (and accept a message) by asserting the ATN (Attention) signal on the bus. It can also reset the bus at any time by asserting RST (Reset).

3.3. Transaction

These phases can be grouped into a *transaction* (a sequence of phases that starts and ends in the BUS FREE phase). Interestingly, it is the target – not the initiator – of the SCSI transaction that determines the sequence of phases from the command

it has been asked to process. The initiator finds out what phase the bus is in by watching the SCSI control lines.

Commands and data can be transferred either asynchronously or synchronously during the information transfer phases. During an asynchronous transfer, the REQ (Request) and ACK (Acknowledge) signals operate in lockstep with the transfer. On a transfer from initiator to target, the target asserts REQ when it is ready for data, and the host asserts ACK when the data is on the bus. The target deasserts REQ when it latches the data, and the initiator, seeing this, deasserts ACK. When data is sent from target to initiator, the REQ line indicates that the target has placed data on the bus, and ACK indicates that the initiator has latched the data.

If the target and initiator agree, however, they can avoid waiting for handshake signals by "windowing" the transfer. The target pulses REQ for each byte of data, and the initiator will eventually pulse ACK the same number of times, but they are allowed to get ahead of one another. This is a synchronous transfer.

3.4. SCSI Commands

The original SCSI standard was developed at a time when each equipment manufacturer used a different set of commands for its devices. All SCSIs, therefore had loose requirements for commands, and almost none of them were mandatory. However, the specification did specify classes and required formats for the commands.

Each SCSI command is sent to a device as a *command descriptor block.* The first byte of each block is the operation code, which in turn has two fields. The first is a *group code* (contained in the three most significant bits) which indicates the type of command and the number of bytes it contains, and the second is a *command code* which specifies the command itself.

Figure 4 shows the layout of a 6-byte (group0) command descriptor block. The larger formats (i.e., 10–byte commands) are similar but leave room for larger addresses

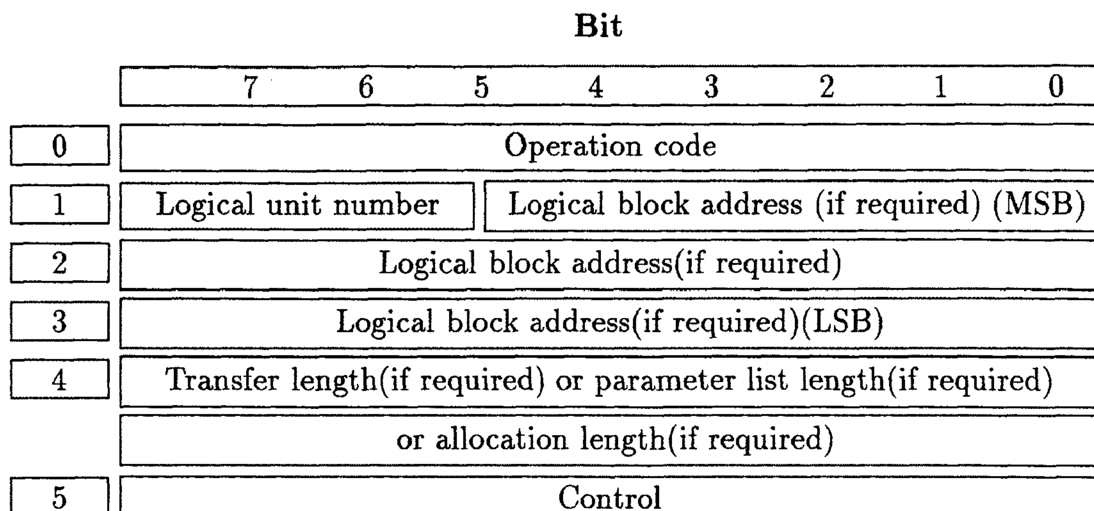


Figure 4: The 6–Byte Command Descriptor Block

and transfer lengths. The eight groups of command codes are divided by length. Group 0 contains 6-byte commands, groups 1 contains 10-byte commands, and group 2 contains 12-byte commands. The other groups are either reserved or vendor-specific. A command descriptor block always ends with a *control byte*, which contains flags that allow several commands to be linked together in a sequence and sent all at once.

Command linking is a powerful SCSI feature. By sending a sequence of linked commands, an initiator can avoid the delays involved in waiting for a command to complete, rearbitrating for the bus, and issuing another command. For instance, suppose the host wants to find a disk block that contains a certain byte sequence and read it into memory. If it sends a SEARCH DATA EQUAL command followed by a READ command to an intelligent SCSI disk drive, the drive will automatically return the correct data with no further intervention.

The following commands are very important to a SCSI driver developer.

- RQUEST SENSE : Whenever a command returns a CHECK CONDITION status, the high-level BSD/386 SCSI code automatically obtains more information

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 4: Sense Key Descriptions

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). Some SCSI devices may also report an "additional sense code qualifier" (ASCQ). The 16 possible sense keys are described in Table 4. For information on the ASC and ASCQ, we can refer to the SCSI standard or to a SCSI device technical manual.

- 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 BSD/386 code uses this command to differentiate among magnetic disks, optical disks, and tape drives.
- READ and WRITE : These commands are used to transfer data to and from the target.

4. The Design of SCSI Host Adapter Driver Interface

4.1. Basic Requirements of The SCSI Device Driver

In the BSD/386 Operating System, the SCSI device driver is divided into three layers, the high-level SCSI interface, middle-level SCSI interface and low-level SCSI interface. The high-level SCSI interface, usually called SCSI device driver, manages all of the interaction between the kernel and the low-level SCSI interface through the middle-level SCSI interface, which is the generic part. The middle-level SCSI interface communicates with both high-level and low-level SCSI interfaces. Because of this layered design, the low-level SCSI driver, usually called SCSI Host Adapter Driver, needs 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.

4.2. Specification of SCSI

SCSI host adapters vary in complexity from a simple parallel port that directly reads and writes the SCSI bus control and data lines to a self-contained controller board with its own microprocessor that handles queues of commands for multiple SCSI buses.

The Future Domain TMC–1800 SCSI Host Interface provides a low-cost, high-performance way to interface SCSI peripheral devices to IBM PC/AT, 80386, and Micro Channel Architecture (MCA) systems. It operates as a host interface in SCSI initiator mode and supports up to seven SCSI targets, each with up to eight logic units attached. Arbitration and reselection are supported.

The TMC–1800 is designed to facilitate easy integration of a SCSI interface into an IBM PC/AT, 80386, or MCA system. The Host Interface is optimized for the INTEL 8086/8088 and 80286/80386 families of microprocessors. The TMC–1800 provides

the following features:

- Incorporates all SCSI drivers and receivers for PC/AT or Micro Channel Architecture (MCA) buses.
- SCSI–2 compatible;
- Supports asynchronous and synchronous SCSI;
- Uses FIFO for very-high-speed data transfer;
- Provides full parity generation and checking;
- Provides a configurable interrupt level, and I/O and memory base addresses;
- Requires no host DMA channels;
- Requires no host system memory;
- Includes 8k x 8-bit static RAM;
- Performs full SCSI arbitration in hardware;
- Low power consumption using CMOS technology;

4.3. BSD/386 SCSI Host Adapter Driver

The SCSI bus is a cable that connects device controllers to a host adapter. The host adapter is a board that converts computer-independent SCSI protocol to specific information for a particular computer. The SCSI Bus is also an industry standard local I/O bus that allows certain *device classes*, such as random-access read/write disks, sequential access tape drives, image scanners, or CD-ROM to be added to a system without modifying the generic system hardware or software. This provides the operating system with a certain amount of device independence within *device classes.*

The SCSI interface provides logical rather than physical addressing, so a logical unit may be only a part of a physical device.

In BSD/386 operating system, the SCSI device driver does not communicate directly with the hardware. Rather, it communicates with SCSI Host Adapter Driver. The SCSI Device Driver has all the usual driver entry points. The SCSI Host Adapter Driver has only a subset of those routines. It is responsible for getting access to the SCSI bus, sending a CDB (Command Descriptor Block) to the selected target (physical) device, sending or receiving any data packet associated with the CDB, and storing the status byte at the conclusion of each command. The overall structure of the SCSI Device Driver Interface is shown in figure 5.

4.4. Overall Structure

The overall structure of SCSI Interface, including the SCSI Host Adapter Driver, Generic Host Adapter Driver and SCSI Device Driver, is a subset of the I/O system structure. Figure 5 shows its overall structure. SCSI device drivers communicate with the Host Adapter Driver through a set of entry points that hide the implementation details of the host adapter from the SCSI device driver. Each specific Host Adapter Driver is a subclass of the Generic Host Adapter Driver. The SCSI Host Adapter Driver is the lowest level in the I/O subsystem. It communicates with the Generic Host Adapter Driver and sends and receives messages to or from physical devices at the same time. From the over structure, we can see whenever we want to add any kind of SCSI Board to the Operating System, we only need to write a SCSI Host Adapter Driver for this specific board. The high-level SCSI device driver should not be changed.

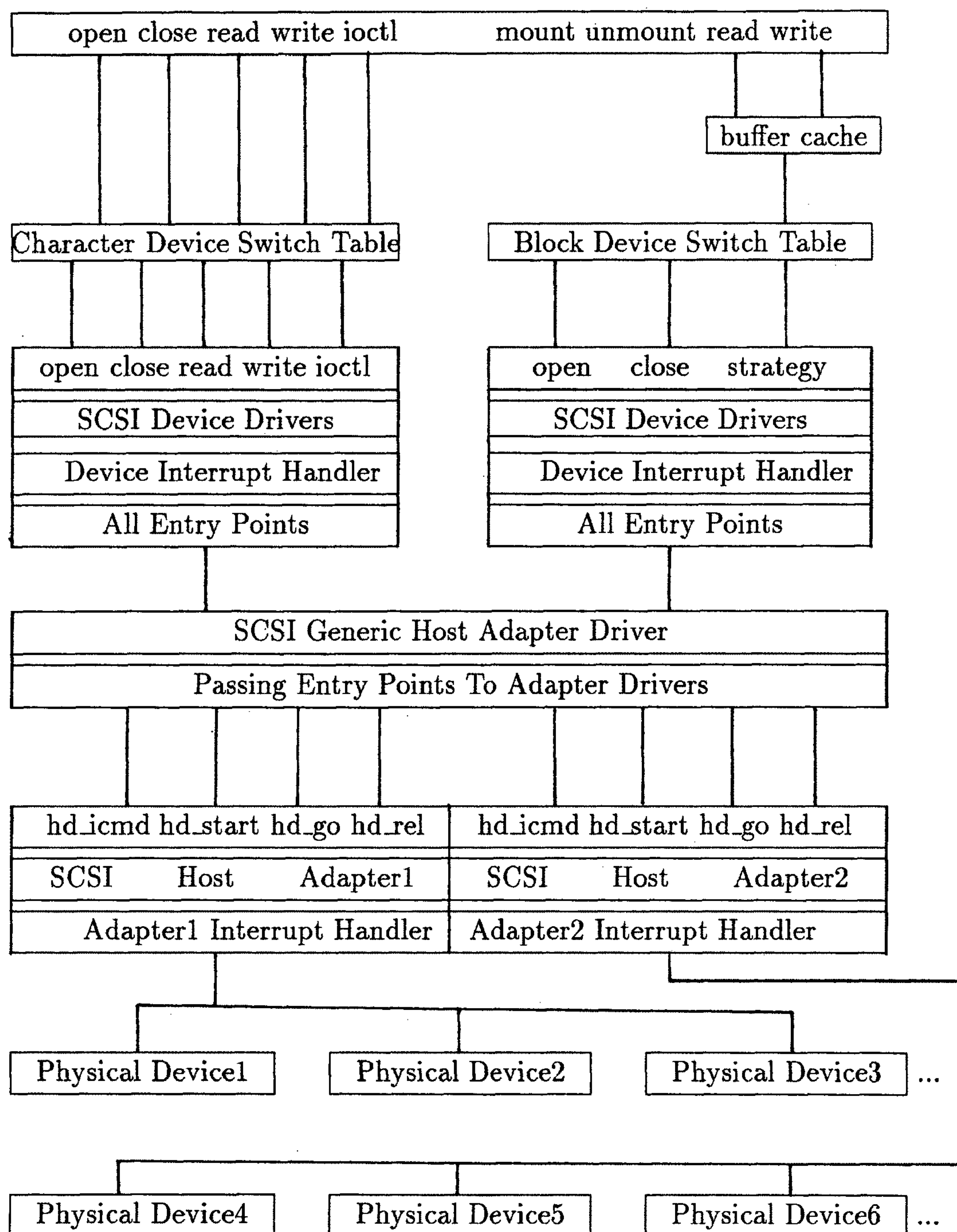


Figure 5: SCSI Device Driver Interface

5. The Implementation of the SCSI Host Adapter Driver Interface

The SCSI software interface is designed in two layers: the lower layer consists of the SCSI Adapter Driver, including generic host adapter driver and host adapter driver, and the upper layer consists of SCSI Device Drivers. The SCSI Host Adapter Driver handles the actual hardware interface to the SCSI bus; it sends and receives commands but does not interpret the contents of the command. The upper layer traditionally controls a particular device or device class–building I/O requests containing SCSI commands and sending them to the Host Adapter Driver to be passed on to the device. This shields the SCSI device driver from the details of the hardware interface. The SCSI Host Adapter Driver also is shielded from the unique requirements of different devices that can be attached to the bus.

Two main structures (`cfdriver` and `hbadriver`) are used to communicate between the high-level code and the low-level code. The following sections will provide detailed information about these structures and the requirements of the low-level driver.

5.1. The Configuration Driver Structure

The Data Structure of the Configuration Driver (cfdriver) is defined in Figure 6. The system assumes that the source code of your driver declares a cfdriver structure named xxdriver. This structure contains information relevant to the device driver as a whole, as opposed to information about individual devices or adapter. It differs in several important ways from the device and adapter structures. For one thing, it contains a number of pointers to driver functions. These pointers, like those in *cdvsw* and *bdevsw*, are used by the kernel as entry points into the driver. For another, it is initialized not by the configuration system, but within the driver source code itself – in fact, several of the routines in xxdriver are actually called by the kernel

autoconfiguration process to complete the driver-related kernel initialization.

```
struct cfdriver
{
    void        **cd_devs;
    char        *cd_name;
    cfmatch_t   cd_match;
    void        (*cd_attach) __P((struct device *,
                                  struct device *, void *));
    size_t      cd_devsize;
    void        *cd_aux;
    int         cd_ndevs;
};
```

Figure 6: The data structure of configuration driver

5.1.1. cd_devs

Variable cd_devs contains the name of a device supported by this driver. This field takes the form of a regular null-terminated C string. Fill in this field if a controller is used.

5.1.2. cd_name

Varible cd_name holds a pointer to a short description of the SCSI host adapter.

5.1.3. cd_devsize

Varible cd_devsize is the size – in bytes – of the memory that the kernel allocates for this host adapter driver' data structure. This field is initialized with a value identical to that which xxprobe returns upon success. This is the amount of space that needs to be mapped into the system memory by the autoconfiguration code. For the drivers of the Future Domain SCSI adapter, this size is for struct fdomain_softc, whose data structure is shown in Figure 7.

5.1.4. Function xxprobe()

The probe function takes three pointers as its arguments. Two of them point to two globe data structures, called struct cfdata, and struct isa_attach_args. These data structures show the base port addresses of all the devices and the relationships between the devices.

This function is called for every adapter and every independent device given in the kernel config file. It determines whether the device/controller is actually installed. The probe function should return 1 if the host adapter board is probed successfully, and return 0 otherwise.

5.1.5. Function xxattach()

This function is called during the autoconfiguration process, where it does preliminary setup and initialization for a device or controller. It is commonly used within disk and tape drivers to perform setup tasks such as reading of labels, and in character drivers for tasks like initializing interrupt vectors and reserving blocks of memory. This function will return 1 if the host adapter is attached, and zero otherwise.

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 users who have this host adapter to use a specific set of I/O port addresses. Other drivers are more flexible, and can 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[10] 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

[10]The "Micro-Channel Architecture" bus is IBM's proprietary 32 bit bus for 286, 386 and i486 machines.

the host adapter's existence and function will be available to the programmer.

Requesting the IRQ During attaching, the *attach()* 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 a method for setting up an IRQ handler: *intr_establish()*.

The *intr_establish()* function takes three parameters, the IRQ number, a pointer to the handler routine, and the device type. The prototype of the function *intr_establish()* is shown in Figure 8.

```
int intr_establish(irq, ih, devtype )
    int                 irq;
    struct intrhand     *ih;
    enum devclass       devtype;
```

Figure 8: The `intr_establish()` function

The first parameter, *irq*, is the number of the IRQ that is being requested, and it causes the interrupt handler routine to be invoked; the second parameter, ih, is a pointer that points to the specific host adapter driver interrupt handler; the third parameter, devtype, is the device type (disk, tape, etc.).

The kernel uses an Intel "interrupt gate" to set up IRQ handler routines requested via the *irqaction()* function. The Intel i486 manual 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.

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 uses a specific DMA channel (for instance, Adaptec Board uses channel 5). This DMA channel will be determined by the *attach()* function and is requested from the kernel with the *at_dma_cascade (drq)* function. This function takes the DMA channel number as its only parameter and allocates the DMA channel.

Some SCSI Boards, like the Future Domain SCSI, don't require DMA channel and system memory. Instead, they use the built-in 8k x 8-bit external static RAM for buffering data to/from the SCSI data port by setting FIFO data transmission bit on.

5.2. SCSI Host Bus Adapter Driver Structure

The SCSI Host Bus Adapter Driver is called by a high level SCSI device driver. Its data structure is defined as shown in Table 9. In general, the variables in the `Hbadriver` structure are not used until after the *attach()* function is called. Therefore,

any variables that cannot be assigned before the host adapter is attached 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 `Hbadriver` structure might then depend on the specific host adapter attached.

5.2.1. hd_icmd – SCSI Immediate Command Function

The SCSI Command has a 6–byte command or a 10–type command. In this function, the initiator first allocates enough space for the whole command structure, which includes command code, logical block address, data length, and control code. it then sends the SCSI command block to the target and waits for ACK from the target. If it gets ACK from the control line, data will be sent to the target. Upon finishing, the first byte of the command block is set to FREE, which tells the target that is the end of the transaction. From the control line, the initiator senses the signal, and frees the bus.

5.2.2. hd_dump – SCSI Dumping Function

This function will dump all the data in the memory into a physical device, if the system crashes accidently.

5.2.3. hd_start – Restarting SCSI Bus Function

The task of hd_start() is straightforward: Examine the next request on the device request queue and tell the controller to start the I/O operation. Then it call SCSI Adapter-resource routines (hd_go function) to request resources needed in doing the transfer. The system allocates these resources dynamically, rather than statically at boot time.

```
struct hbadriver
{
   int          (*hd_icmd) __P((struct hba_softc *,
                                int targ,
                                struct scsi_cdb *cmd,
                                caddr_t addr, int len,
                                int rw));
   int          (*hd_dump) __P((struct hba_softc *,
                                int targ,
                                struct scsi_cdb *cmd
                                caddr_t addr, int len,
                                int isphys));
   scstart_fn  hd_start;
   scbusgo_fn  hd_go
   scbusrel_fn hd_rel;
   void         (*hd_reset) __P((struct hba_softc *,
                                int));
};
```

Figure 9: The data structure of SCSI Host Adapter Driver

5.2.4. hd_go – Start I/O Transfer Function

This function is called by hd_start function for allocating resource. When resources become available (for instance, the adapter is not in use), it calls back to the SCSI immediate command to begin the I/O data transfer. If the adapter is busy, that request will be placed on a queue of device request.

5.2.5. hd_reset – Reseting SCSI Adapter Function

The *reset()* function is used to reset the SCSI bus. After a SCSI bus reset, any executing command fails, and all the devices that connect to the bus are reset.

To 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.

6. Conclusion

The compelling factors behind SCSI's sudden popularity boost are its variety and compatibility. The old SCSI is SCSI-1 (such as Adaptec Board). It was written as a catch-all, to accommodate a range of bus users with widely varying needs. As it turned out, SCSI-1 still wasn't loose enough. Also it maxes out at about 10-megabits-per-second transmission speed. On the BSD/386 Operating System, the Adaptec transfers data at 5-megabits-per-second. Compared to SCSI-1, SCSI-2 (for instance, Future Domain SCSI Board) extends the 8-bit data path of standard SCSI to 16-bit or 32-bit (depending on its mode), to directly service the wider I/O buses on newer small computers. The transmission speed reaches up to 40-megabits.

The original goal of the project: to design an efficient software interface for the I/O subsystem of the modern operating system, to develop and implement a simple, reliable, and an efficient SCSI Host Adapter Driver that runs on the BSD/386, has been achieved. BSD/386 will start to take advantages of SCSI-2 with superior performance when the current project is incorperated into the new release of BSD/386.

6.1. Optimization

6.1.1. The Third Generation SCSI

A SCSI-3 committee is now working on a general updating of SCSI, mostly to keep it abreast of advances in bus technology. All these variations can talk to each other on a lowest-common-denominator basis. This is generally accomplished by starting a SCSI conversation at the lowest level, after which a higher-capability device will propose switching the conversation to its natural level. If the other device can not do this, the conversation merely continues as it started. SCSI devices can also ask each other to identify their device types to speed up the process.

6.1.2. Object-Oriented Host Adapter Driver

From Figure 5, we can see the relation between each level of the device driver. The SCSI Host Adapter Driver is a subclass of the Generic Host Adapter Driver, which is connected to SCSI Device Drivers, which are subclasses of the I/O subsystem, in turn. Each subclass has its unique and distinct attributes and functions, and also inherits some common attributes and functions.

Object–Oriented Programming Languages (like C++) focus on code model definition, using an inheritance concept. In order to take advantage of the inheritance concept, an Object-Oriented Programming Language could be used in the future to simplify the implementation and shorten the program. For example, two main structures (`cfdriver` and `hbadriver`) can be combined into a class called connection_class. Its definition is shown in Figure 10. However, this design needs much exploiting work for the implementation of the lowest level in the device driver subsystem, which, in general, is hardware dependent.

6.1.3. The Standardized Design and Implementation

The SCSI device driver is shielded from the details of the hardware interface, and the SCSI Host Adapter Driver is also shielded from the unique requirements of different devices that can be attached to the bus. These features make it possible that one SCSI device driver works with many SCSI Host Adapter Boards, and one SCSI Host Adapter Driver works with many types of SCSI device drivers and different peripheral devices. Compared with the conventional device driver, this is a significant progress. All the designs and implementations of the SCSI device drivers are based on SCSI standard.

Even though one SCSI Host Adapter Driver can work with many types of the peripheral devices, it works with only one type of SCSI Boards. Is it possible to

```
class connection_class  {
   private: char       *cd_name;
            size_t     cd_devsize;
   public: cfmatch_t  cd_match;
           void       (*cd_attach) __P((struct device *,
                                        struct device *, void *));
           int         (*hd_icmd) __P((struct hba_softc *,
                                       int targ,
                                       struct scsi_cdb *cmd,
                                       caddr_t addr, int len,
                                       int rw));
           int         (*hd_dump) __P((struct hba_softc *,
                                       int targ,
                                       struct scsi_cdb *cmd
                                       caddr_t addr, int len,
                                       int isphys));
           scstart_fn  hd_start;
           scbusgo_fn  hd_go
           scbusrel_fn hd_rel;
           void        (*hd_reset) __P((struct hba_softc *,
                                       int));
};
```

Figure 10: The class of the SCSI Host Adapter Driver

make the writing of SCSI Host Adapter Driver standardized? Theoretically, that is possible. On the same UNIX system, the structure of the SCSI Host Adapter Driver is able to be kept consistent, no matter what kind of SCSI Host Adapter Board is installed. Furthermore, for the different SCSI Host Adapter Drivers, the connection between the high-level and low-level codes is similar, and it can be made standard. Generally speaking, for the standardized implementation of the SCSI Host Adapter Driver, the system will provide

- A standard structure of the SCSI Host Adapter Driver.
- A standard software connector between the SCSI device driver and the SCSI Host Adapter Driver.
- A standard basic service between the connector and the user functions with standard definition.

Under this circumstance, a system programmer is required to provide a set of specific SCSI Host Adapter hardware-dependent functions, which also have standard function definitions.

The standardized implementation of the Host Adapter Driver will make it easier for the system programmer to develop a specific SCSI Host Adapter Driver in the future.

References

[1] Janet I. Egan and Thomas J. Teixeira , "*Writing a Unix Device Driver*" John Wiley and Sons, New York, NY, 1992

[2] Maurice J. Bach, "*Design of Unix Operating System*" Prentice–Hall, New York, NY, 1986

[3] Peter kettle and Steve Statler, "*Writing Device Drivers for SCO UNIX*" Prentice–Hall, New York, NY, 1992

[4] Samuel J. Leffler, Marshall Kirk Mckusick, Michael J. Karels and John S. Quarterman, "*The Design and Implementation of the 4.3BSD UNIX Operating System*" Addison–Wesley, New York, NY, 1987

[5] Berkeley, *BSD/386 V1.0 Installation Notes, Berkeley Software Design, INC.*

[6] Walter Zintz , "SCSI's Busting Out All Over" , *UNIXWORD*, Oct. 1992, pp. 76–79

[7] Future Domain Corporation, "*TMC-1800 SCSI CHIP SPECIFICATION*", FDC–1800T Specification Errata Sheet, P/N: 02-01800-303-00

[8] William Frederick Jolitz and Lynne Greer Jolitz, "*Porting UNIX to the 386 Device Drivers*" Dr. Dobb's Journal, March 1992, pp. 44–50

[9] Roger C. Alford "*The Evolution of ESDI*", BYTE, june 1990, pp. 297–306

[10] Mark L. Van Name and Bill Catchings "*486 EISA Machines: A Slow Start in the Fast Lane*", BYTE, Oct. 1990, pp. 172–176

[11] Nick Baran "*EISA Arrives*", BYTE, Nov. 1989, pp. 93–98

[12] L. Brett Glass, "*Inside EISA*", BYTE, Nov. 1989, pp. 417–425

[13] Corey Sandler, "*New Controller Makes SCSI Palatable to PCs*", BYTE, Nov. 1990, pp. 205–208

[14] George White, "*A Bus Tour*", BYTE, Sep. 1989, pp. 296–302

[15] Bruce Van Dyke, "*SCSI: The I/O Standard Evolves*", BYTE, Fall 1990, pp. 187–191

[16] L. Brett Glass, "*Part 1: The SCSI Bus*", BYTE, Feb. 1990, pp. 267–274

[17] L. Brett Glass, "*Part 2: The SCSI Bus*", BYTE, March 1990, pp. 291–298

[18] Lynne McCue, "*Writing a SCSI Device Driver: A Pass-Through Implementation*" , AIXpert, Feb. 1993, pp. 24–31

[19] Edwin C. Perkins, "*Should Your Next UNIX Workstation be a PC?*", UNIXWORD, Feb. 1993, pp. 90–96