ABSTRACT

SMALL COMPUTER SYSTEM INTERFACE (SCSI) UNIVERSAL SERVICES FOR THE TURBONET PARALLEL COMPUTER

by
Artak Ohan Melkonian

TurboNet is a parallel computer with a shared-memory and message-passing hybrid architecture. It employs two boards, with four digital signal processors (DSPs) each, and a host FORCE SPARC CPU-2CE board with a SCSI bus.

Software has been developed in this thesis to provide SCSI services to programs running on the DSPs. DSP programs can therefore fully control assigned SCSI devices at the SCSI command level. Transfer control modifiers ensure compatibility with most SCSI devices.

The software provides service for three SCSI access levels. The "su" SCSI universal device driver is built into the host computer's kernel and is a gateway to the SCSI bus from user contexts. The "hscsid" SCSI request server daemon is an interrupt driven link between the DSP programs and the "su" driver. The Hydra SCSI utilities can be included in programs to make SCSI programming easier.

SMALL COMPUTER SYSTEM INTERFACE (SCSI) UNIVERSAL SERVICES FOR THE TURBONET PARALLEL COMPUTER

by
Artak Ohan Melkonian

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

January 1996

APPROVAL PAGE

SMALL COMPUTER SYSTEM INTERFACE (SCSI) UNIVERSAL SERVICES FOR THE TURBONET PARALLEL COMPUTER

Artak Ohan Melkonian

Dr. Sotirios Ziavras, Thesis Advisor — Date
Associate Professor of Electrical and Computer Engineering,
and Computer Science, NJIT

Dr. Constantine Manikopoulos, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Edwin Hou, Committee Member — Date
Assistant Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Artak Ohan Melkonian

Degree: Master of Science in Electrical Engineering

Date: January 1996

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1996

- Bachelor of Science in Computer Engineering,
  State Engineering University of Armenia, Yerevan, Armenia, 1992

Major: Electrical Engineering

This thesis is dedicated to
my beloved wife Jenny

ACKNOWLEDGMENT

I wish to express my true gratefulness to my thesis advisor, Professor Sotirios G. Ziavras, who helped me to go through this research providing me his inestimable and contributive support and guidance. Special thanks to Professors Constantine Manikopoulos and Edwin Hou for their essential participance as members of the committee.

I am sincerely thankful to the President of the Republic of Armenia Levon Ter-Petrosian for providing continuous funding for this research.

I would like also to appreciate the support given by Professor Deran Hanesian from my very first day at the NJIT.

Special thanks also to Christopher Jackson from Sun Special Projects for kindly answering my questions about system level programming.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

Table **Page**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Short Description of Small Computer System Interface (SCSI)

SCSI is a local input/output bus primarily intended for connecting host computers to a wide range of peripheral intelligent devices, including direct access devices (e.g. fixed and flexible magnetic disks), sequential access devices (e.g. magnetic tape), processors, printers, write-once-read-multiple (WORM) optical devices, CD-ROM optical disks, scanners, optical memory devices, medium changer devices, and communication devices. ANSI has defined the SCSI standard in its X3.131-19XX documents, where X3.131-1986 [1] is defined as SCSI-1, and X3.131-1990 [2] as SCSI-2.

A SCSI bus can connect up to 8 devices; however, one of them usually is a host computer. Each device can have its own command set. Every SCSI device may consist of 8 logical units (LUN) addressed separately. SCSI devices are intelligent units capable of understanding and performing standard sets of commands, returning detailed status information and providing full control by controlling operation systems.

SCSI-1 has been defined as an 8-bit local bus with up to 5Mbyte/sec throughput and supports only a subset of the device types mentioned above. SCSI-2 supports all of them and provides a more sophisticated set of bus and device control facilities. There exist some enhancements of the 8-bit SCSI-2. The Fast-SCSI provides a throughput of up to 10Mbyte/sec. The Wide-SCSI uses a 16-bit bus. The Fast-and-Wide SCSI can achieve up to 20Mbyte/sec throughput.

SCSI has been used widely in midrange to small multi-user workstations (Sun, DEC, IBM, Silicon Graphics, etc.) and some personal computers (primarily the Apple Macintosh). In the PC market, SCSI comes as an advanced alternative to IDE (ATA bus) interface, for users who want to take advantage of multiple devices on a single bus.

In general, a SCSI transfer is controlled by specific bus signals and their timing combinations, that determine the following bus phases: BUS FREE, ARBITRATION, SELECTION, RESELECTION, COMMAND, DATA IN/DATA OUT, STATUS and MESSAGE. Messaging activities, taking place in the MESSAGE phase, define low-level control and status exchange between a transfer initiator and a target. The initiator transfers SCSI commands in the COMMAND phase as an opaque array of bytes, which are to be interpreted by the target's processor. Data is transferred in DATA phases. The other bus phases provide control of bus states, arbitration, and handshaking.

1.2 The TurboNet Parallel Computer

The TurboNet system is a parallel computer employing a hybrid architecture, i.e. it possesses message-passing and shared-memory capabilities. Currently the system consists of two Hydra boards, each of them containing four Texas Instruments TMS320C40 digital-signal processors (DSP), two VME bus analyzers, and a host SPARC system running Solaris 1. All of these components are attached to a VME backplane.

The eight DSPs are connected as a hypercube, for the implementation of message passing. However, all eight DSPs can access the shared memory as well.

The host system has a complete SPARCstation 2 architecture with an embedded VME bus. Along with other standard SPARCstation I/O ports, the host system supports SCSI buses.

1.3 Motivation and Objectives

Although TurboNet is a very powerful computational engine and can also run CPU/data-intensive application programs, input/output of large amounts of data from the SCSI port of the host system to the Hydra boards, and vice versa, is of very limited functionality. The Hydra boards originally could not control dedicated SCSI devices, i.e. issue SCSI commands, get status information and transfer data from the shared memory to the SCSI bus and vice versa.

Before writing the software described in this thesis, the control and data flow in general was as follows. The host program initialized all input data residing on the Solaris controlled fixed disks, then downloaded the DSP program(s) and the data into the Hydra boards via the VME bus and finally, ran the DSP program(s) and uploaded the results from the boards onto the system disks.

As it could be easily seen, the scenario described above has several disadvantages. The DSPs on the Hydra boards had been used as slave coprocessors of the master host computer. The data had been kept on fixed disks using the UNIX filesystem, therefore the DSP programs could not arbitrarily manage the data on their own dedicated disks. Moreover, this approach would not allow the Hydra boards to fully control arbitrary SCSI devices (scanners, printers, WORM or optical storage devices) on the command level.

The objective of this work was to develop software running on different levels of the TurboNet system, enabling SCSI command-level custom control by the Hydra boards and providing system-level SCSI services to programs running on the host or on the DSPs.

The first level of the SCSI custom support is a universal SCSI device driver embedded in the SunOS kernel of the SPARC host system. It supports a flexible interface (specific data structures and services) for user SCSI requests. The SCSI Universal "su" device driver can support almost all kinds of SCSI devices, and the standard or vendor specific commands associated with them.

The second level of the SCSI custom support is a host-side SCSI request server, implemented as a UNIX daemon, which provides data structures and SCSI request forms for application programs running on Hydra boards. It implements the host side of the host-Hydra interprocess communication protocol, which is fully interrupt driven. It actually serves as a link between the "su" SCSI universal device driver and the host-Hydra program pairs. Due to this, the Hydra DSPs become the master initiators of SCSI transfers.

The third level of the SCSI custom support contains a simple set of Hydra SCSI utility functions and data structures. Only the basic calls are included; however, the open architecture allows application programmers to add a variety of their own specific functions. The access to the Hydra SCSI services is designed to be very easy and straightforward for use and customization.

1.4 Outline

The thesis is organized as follows. Following this introduction, Chapter 2 briefly describes the TurboNet system, including an overview of the TMS320C40 DSP and the Hydra architecture, characteristics and the Hydra device driver and the library for Solaris 1. The SPARC host system, the Solaris 1 kernel and the SCSI device driver interface overview are also included. Chapters 3, 4 and 5 present three levels of SCSI support software: the SCSI Universal device driver, the SCSI request server deamon, and the DSP SCSI utilities, respectively. Chapter 6 gives some performance results, draws conclusions and presents further research directions.

CHAPTER 2

THE TURBONET: A MESSAGE-PASSING AND SHARED-MEMORY HYBRID ARCHITECTURE

This chapter presents the TurboNet parallel computer. The hardware overview of the TMS320C40 DSPs, the Hydra boards, the host computer board and the VME backplane is given. The Hydra device driver and the utility library for the Solaris 1 are introduced. Short description of the Solaris 1 kernel and SCSI device drivers is also included.

2.1 System and Interconnections Overview

The base for the TurboNet system is a 21-slot 6U VME backplane with a power supply and 2 bays for full-size I/O devices (e.g. fixed or flexible disks). The SPARC host board is attached to the first slot of the VME backplane, and is configured as a VME system controller. Two Solaris-controlled system disks and one Hydra controlled fixed disk are attached to the SCSI port of the host board. The Sun monitor and keyboard are the console of the host board. The system is connected to the campus network via an Ethernet adapter.

The VME backplane contains also two VME-bus logic analyzers, which are controlled by a DEC dumb terminal via a RS-232 port.

The main processing engine of the system consists of two (currently) Hydra boards, which contain four TMS320C40 digital signal processors each. Both Hydra boards can be accessed from the controlling VT220 terminal through their RS-232 ports. This is mostly used for setting up hardware parameters and examining the memory.

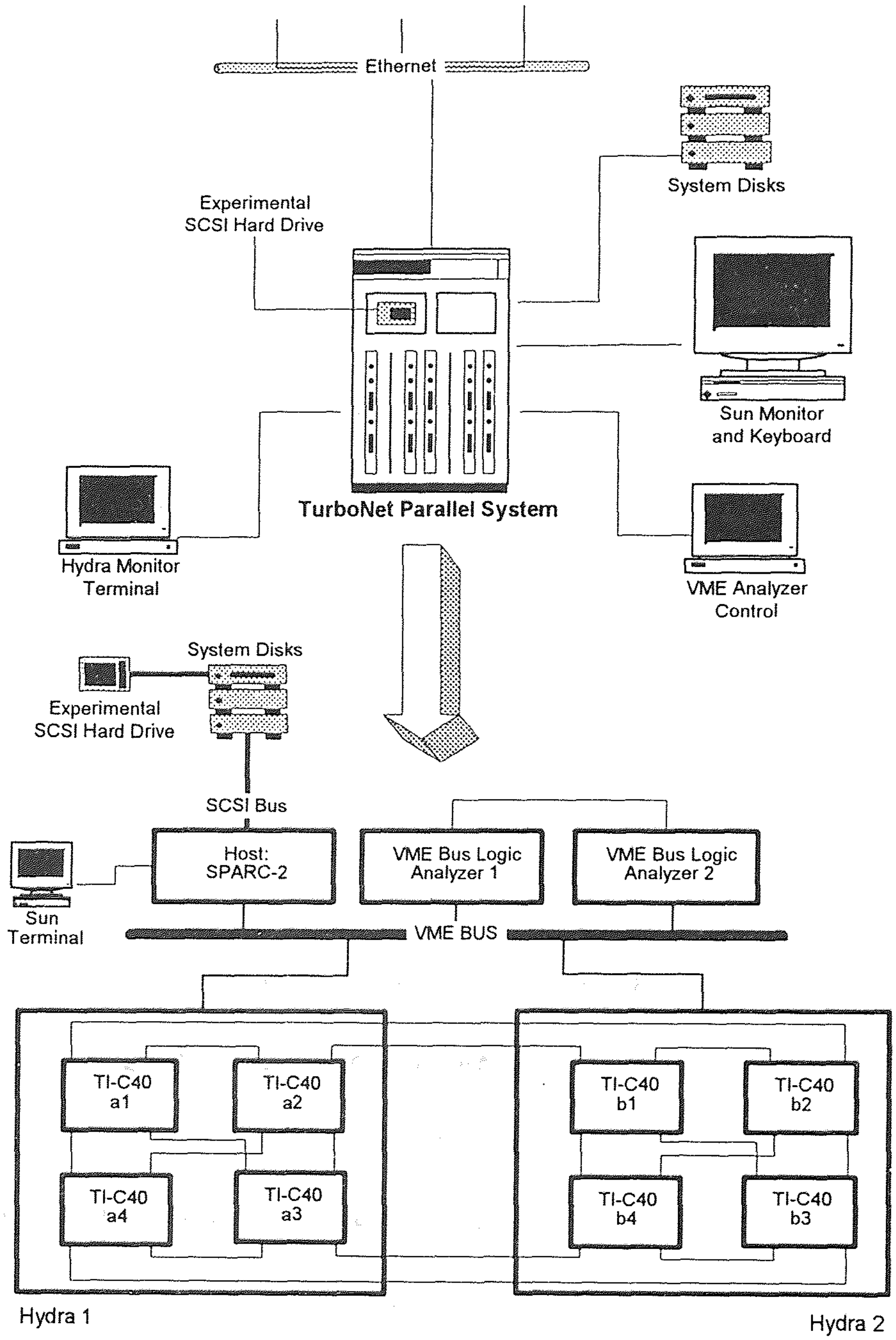

Figure 2.1 The interconnection diagram of TurboNet.

2.2 The Texas Instruments TMS320C40 Digital Signal Processor

Texas Instruments TMS320C40 [3] DSPs are the main processing elements of TurboNet and have the following primary features:

- Six 8-bit bi-directional half-duplex communication ports for high speed (up to 20-Mbyte) interprocessor communication.
- Six-channel DMA coprocessor for concurrent I/O and CPU operation, thereby maximizing sustained CPU performance by alleviating the CPU of burdensome I/O.
- High-performance DSP CPU capable of 275 MOPS and 320 Mbytes/sec of data-transfer rate.
- Two identical external data and address buses supporting shared-memory systems and high data rate, single-cycle transfers. There exist two 32 bit data buses called the Global Bus and the Local Bus; each of them is capable of addressing 2 Gwords (x32 bits) of address space for a total of 4 Gwords addressable by each C40.
- On-chip analysis module supporting efficient, state of the art parallel processing debugging with on-chip hardware breakpoints.
- Bootloader ROM, 512-byte on-chip program cache and dual-access/single-cycle RAM (2 Kwords) for increased memory access performance.
- Separate internal program, data, and DMA coprocessor buses for support of massive concurrent I/O of program and data throughput, thereby maximizing sustained CPU performance.

2.3 The Hydra Multi-DSP Boards

2.3.1 Hardware Description

The Ariel Hydra [4] is a single-slot 6U VME card containing 4 TMS320C40 DSPs, with two of them on the base board and two on the daughter-board. Each DSP owns two static RAM banks on its local and global buses, and can access them without interfering with the operation of the other DSPs. The static memory banks can be as large as 16K or 64K or 256K (the latter is only for DSPs on the base board) 32-bit words. For the current configuration of TurboNet, the DSPs of the first Hydra have 64K static RAM each, while the DSPs on the other board have 16K.

Each board can have a shared memory resource either as large as 1M, 4M or 16M 32-bit words for DRAM configuration, or 128K to 1M 32-bit words for SRAM configuration. The DSPs can gain access to the shared memory through the Internal Shared Bus (ISB). All the DSPs requesting the ISB will get an access with rotating priority. If the ISB is in use, a requesting device will be held in a wait-state until the resources are available. However, the chips controlling the VME bus interface always can take priority over the other requesters. The ISB arbitration logic changes the ownership only when the current master is done using the bus, when the current master crosses a DRAM page boundary, or when a DRAM refresh cycle occurs. The latter happens every 15μs, and the current ISB master loses the bus when DRAM control circuitry requests it for a refresh operation. All the ISB requesters will be held in wait states until the end of the refresh cycle, after which a device with the higher priority will become an ISB master.

Since the DRAM refresh cycle is not "transparent", because of the hardware design of the Hydra board, there cannot be a continuous ISB access operation any longer than the DRAM refresh period. Although this fact may only affect the ISB performance when on-board masters access the bus, or the host accesses it through the Hydra device driver, the use of the DMA on the host board can cause a serious problem.

The Hydra VME bus interface includes a VME Interface Controller chip (VIC) and a VME Address Controller chip (VAC). These devices operate as a set, and provide a fully functional interface between the VME bus and the ISB. The VIC chip translates the ISB control signals to the equivalent VME control signals during the VME bus master cycles. It also converts the VME control signals into the ISB signals when the Hydra is a VME bus slave. The VIC can act as a VME interrupter and interrupt handler, and can operate as a Slot 1 VME bus controller.

The second major device in the VME bus interface, the VAC, is a programmable memory map address controller which works in conjunction with the VIC. It contains programmable registers to allow the user to easily define memory maps for both the VME address bus and the ISB address bus.

Every TMS320C40 DSP on a Hydra board is connected to the other three through a 20Mbyte/sec, 8-bit bi-directional ports, which are connected to each other without any extra 'glue' logic. Since every DSP has six ports, the remaining three ports for each DSP are brought to the front panel of the Hydra board for external connections. This allows users to create custom configurations of message passing parallel machines. In our system, these ports as used to create a hypercube parallel configuration with eight DSPs on two Hydra boards, as shown in Figure 2.1.

The DSP bootstrap and initialization are performed by the DSP#1, which is the only processor that can access the Boot EPROM and the EEPROM. When the Hydra board powers up, the boot loader program resident in the DSP#1 starts the boot program in the Boot EPROM, which runs hardware diagnostics, configures the Hydra internal registers, the VME interface chips and boots other three DSPs through their communication ports. Hydra boards keep their setup and operating parameters in the EEPROMs. Each EEPROM contains such parameters as the register values for the Hydra internal registers and the VIC/VAC, the DRAM size, the sizes of the local and global memories, the serial port setup, etc.

2.3.2 The HydraMon Monitor

The DSP bootstrap procedure is a part of the HydraMon monitor program, which controls the interaction of the Hydra boards with the outside world. It has three main sections: startup, terminal and host. The startup section is the bootstrap procedure. The interactive command-line type interface between users and the Hydra board through a dumb terminal and RS-232 port are controlled by the terminal section. The user can configure the board, and examine or modify the memory. The host section is the most important one, because it handles all the communication between the Hydra board and the host computer through the VME bus. The HydraMon on a 4-DSP Hydra board reserves 16640 bytes from the top of the shared memory for the host section of the monitor. This space, beside containing some control information, is used by some data transfer library routines.

HydraMon can service the following requests from the host system:

- **BootADSP**

  Boots a DSP from a communication port with the monitor.

- **CopyStuff**

  Copies data from the specified source to the specified destination. The data is copied as 1-Kbytes parts.

- **Run**

  Starts an execution at the specified address.

- **Halt**

  Halts the program that is currently running and enters a dead loop inside the monitor. Brings a DSP to a known state.

- **HostIntNumber**

  Sets the VME interrupt number that is used with the trap 7 VME interrupt service. This number originally resides in the Hydra device driver configuration.

- **HostIntVector**

  Sets the VME interrupt vector that is used with the trap 7 VME interrupt service. This number also originally resides in the Hydra device driver configuration.

- **DisableKeyInt**

  Disconnects the UART input data ready interrupt from the DSP 1 NMI. Actually disables the terminal keyboard.

- **EnableKeyInt**

  Connects the UART input-data-ready interrupt to the DSP 1 NMI.

- **UserInt**

  Executes an interrupt handler service, that was previously registered by the DSP user program. The user program enters the interrupt handler's entry point address into the DSP trap table at the appropriate vector. The host can then request UserInt with the trap number, which will execute the registered interrupt handler routine. The user program can register up to four interrupt handlers at the trap numbers 0x9 through 0xC.

HydraMon also provides two services to the DSP programs:

- **GetBoardInfo**

  Activated by invoking trap 0x8. Returns a pointer to the `hydra_conf` structure, which is defined in `hydra.h` of the monitor source and contains the configuration information for the Hydra board. Note that different revisions of Hydra return pointers to different data structures, e.g. revision 'E' and revision 'C', which we currently have in our system.

- **HostInt**

  During the initialization process, the host informs HydraMon which VME interrupt vector and number to use. By invoking trap 0x7, the user can assert a

VME interrupt with the specified vector, which will be caught by the Hydra device driver on the host system and will be delivered to a user process as a UNIX signal.

The UserInt and HostInt HydraMon services are the key points of the interrupt driven approach of the Hydra SCSI device driver.

2.3.3 Hydra Device Driver and Utility Library for Solaris 1

The VC40DSP, the Hydra device driver, is a traditional UNIX device driver written for SunOS and targeted for a Sun 3 or Sun 4 architecture. In general, the device driver can be loadable for SunOS beyond 4.1.2, but it is not the case for our system, because the host system runs a Solaris 1 (SunOS 4.1.3) with the '4c' kernel architecture, and has a non-native VME bus extension. The VC40DSP provides full control of the multiple Hydra boards and has an interrupt service capability.

Several standard UNIX system calls, such as `open()`, `close()`, `read()`, `write()`, `mmap()`, `ioctl()`, are used to interact with the Hydra device driver. These system calls are fully described in the second section of the SunOS Reference Manual [5]. Each DSP on the Hydra boards is presented as a separate character device special file `vc40xy` in the `/dev` directory of the host system, where the x is a lowercase letter (e.g. 'a', 'b') and represents the board number. The y represents the number of a DSP in a board and can be from 1 to 4.

Most of the Hydra device-driver calls have their user friendly forms as the Hydra library functions. These calls control DSP states, transfer data to/from the Hydra memory and the special registers, map the shared memory, etc. The following library functions are used in the host-level SCSI-request server daemon (see Chapter 5), and are fully described in [6]:

`c40_enint()`

Enables Hydra-to-host interrupt generation and links a UNIX signal to a Hydra interrupt.

`c40_getinfo()`

Returns information about a DSP.

`c40_load()`

Loads a TMS320C40 executable COFF format code to a DSP and extracts the symbol table information from COFF.

`c40_map_shmem()`

Returns a pointer to the Hydra's shared memory.

`c40_reset()`

Resets a DSP to its power-on state.

`c40_run()`

Causes a DSP to begin executing a program from a specified address.

`c40_trap()`

Causes a DSP to execute a specified software trap.

2.4 Force SPARC CPU-2CE Host System: Hardware Description

The Force SPARC CPU-2CE [7] is a complete SPARCstation 2 architecture implementation with Sbus expansion on a single 6U VME bus slot. The system offers DMA supported SCSI and Ethernet ports along with audio, keyboard/mouse, and two serial channels with full modem support. The Sbus sockets allow the installation of any of the over 300 available Sbus cards, such as graphics frame buffers.

The central processing unit is a 40-MHz SPARC (Scaleable Processor ARChitecture) 32-bit RISC chip set. At the 40-MHz, it has 28.5 MIPS integer performance and 4.2 MFLOPS floating-point performance. The board in our system has 64 MB RAM, and a 16K x 16 cache memory.

The DMA chip provides a DMA and data assembly-disassembly function for both the Ethernet and SCSI interfaces. The DMA ASIC contains a 32-byte FIFO buffer for each interface and performs a DMA in 16-byte bursts when the alignment and transfer length permit.

The SCSI controller is a NCR 53C90A controller chip providing SCSI-1 functionality and can transfer up to 5MB/sec in synchronous mode (2.5 MBytes/sec typically) and up to 3MB/sec in asynchronous mode (1.75 MBytes/sec typically). The SCSI data goes through the D-channel of the DMA ASIC. The SCSI port features automatic termination adjustment depending on external devices connected to it. The SCSI signals are brought to the front panel and to the VME P2 connector for additional flexibility. Single-ended mode is only supported. The host SPARC CPU-2CE board acts as a SCSI initiator, which can control up to seven SCSI devices.

The VME bus interface of the SPARC CPU-2CE board is built using the Sun VME chip, the S4-VME, which provides a complete 32-bit solution. The master and slave mode operations, the VME bus interrupt service and the system controller functions are fully implemented. The VME bus interface is controlled by a proprietary driver for Solaris 1. This driver is necessary for the Hydra device driver operation.

2.5 The Kernel and SCSI Device Driver Interface

2.5.1 Solaris 1 Kernel and Device Drivers

All the I/O and special-purpose hardware devices in Solaris 1 are controlled by their respective device drivers [11-13]. Device drivers can be built-into the kernel or be loadable. While the Solaris 1 kernel is intended to provide an I/O, virtual memory management and scheduling interface to user processes, i.e. isolate them from the hardware, the device drivers usually provide certain interfaces to the kernel itself such as standardizing the hardware access mechanisms, freeing it from device specific processing, and providing ways to make arbitrary and portable hardware configurations using the same kernel core.

In general, there are two types of Solaris 1 device drivers: structured or block, and unstructured or character drivers. The block device drivers usually control devices that can contain mountable filesystems. The kernel accesses these drivers using the buffer-caching mechanism, which expects these devices to be random-accessible. As the name for this type of drivers states, the transfers are done using blocks of data, because the actual hardware devices, which are capable of containing filesystems, are usually disk devices (fixed disks, CD-ROMs, even RAM-disks). Such devices act as and/or are block I/O devices and allow a random access. Almost always, the block device drivers can perform a byte-oriented or so called raw I/O, but that is only an imitation and is supported using the same actual implementation for the block I/O.

The character device drivers, oppositely, are intended to transfer byte-oriented data. Examples of them are serial, printer, frame buffer, audio, network devices, etc. These devices cannot contain filesystems because of the fact that they do not contain a random accessible media. Character device drivers sometimes may contain routines and data structures specific to the block drivers; however, this does not change their byte oriented nature and it is done only because it is convenient to do so.

Developing Solaris 1 device drivers is a complex process which includes many sophisticated issues such as conforming kernel requirements for a specific type of a driver, working with the kernel resources, the virtual memory and the other device drivers, run-time debugging, etc. Since the device drivers are parts of the kernel, which runs in the supervisor mode, it is very easy to cause a damage to the system when running a buggy code. Non-loadable drivers are especially hard to debug, because even after a little modification of the driver code the kernel should be compiled, installed, and the machine rebooted to see the changes, which takes a significant amount of time.

2.5.2 I/O Processing Path Up to the Device Driver Entry Points

In the Solaris 1 environment, user processes usually gain I/O access using the standard system calls, which can be applied to any file with appropriate access permissions. An I/O system call can be viewed as a special kernel routine, which runs in the supervisor mode, may activate a certain device driver, perform any necessary I/O, if possible, and return data and a related status information. The caller process is suspended while the system call is being processed. The user requests to operate on regular files are carried out by the kernel using its buffer cache mechanism, which is transparent to the user programs and actually uses the block devices.

Solaris 1 maintains a system of "special" files, which represent the actual devices. Any operation request on these "special" files will be converted into calls to the entry points in respective device drivers, which, if the operation is permitted and feasible, will perform the requested I/O. The "special" files are contained in the `/dev` directory and, like the regular files, have their permission and owner settings. However, the i-nodes for these files contain device specific information such as the major and minor numbers and the device type (block, character, FIFO, or socket). The table below represents a sample listing of "special" device files in the `/dev` directory:

Table 2.1 A sample listing of the `/dev` directory

```
  .    .    .

crw-rw-rw-   1 root      111,    0 Jun   9 14:57 rsu0
crw-rw-rw-   1 root      111,    1 Jun   9 14:57 rsu1
crw-rw-rw-   1 root      111,    2 Jun   9 14:57 rsu2
crw-rw-rw-   1 root      111,    3 Jun   9 14:57 rsu3

  .    .    .
```

The first character in the permissions string is the device type ('c' stands for character in the sample list above). All the other permission settings determine the device accessibility. The third column represents the owner of the device. The fourth and fifth columns contain the device major and minor numbers, and will be explained below. The ninth column shows the device file names. Since these files contain special information, they can only be created by the `mknod` command [5,10].

When a user process requests an access to a "special" file using the file name, the kernel checks for the access permissions, and if the operation is permissible, retrieves the device type information and the major and minor numbers from the file's i-node. The major number is used then to allocate the device driver's entry points in the `cdevsw` or `bdevsw` Solaris 1 kernel structure arrays [11], depending on the type of the device (`cdevsw` is for character devices, `bdevsw` is for block devices).

Table 2.2 SunOS 4.1.3 kernel `cdevsw` structure in `/usr/kvm/sys/sys/conf.h`

```
struct cdevsw {
        int      (*d_open)();
        int      (*d_close)();
        int      (*d_read)();
        int      (*d_write)();
        int      (*d_ioctl)();
        int      (*d_reset)();
        int      (*d_select)();
        int      (*d_mmap)();
        struct   streamtab *d_str;
        int      (*d_segmap)();
};
```

The following piece is the `cdevsw` structure array element for "su" SCSI universal device driver and is contained in the `/usr/kvm/sys/sun/conf.c`. As it can be seen in this example, the "su" driver has only three operational entry points: `su_open`, `su_close` and `su_ioctl`.

Table 2.3 An element in the `cdevsw` structure array in `/usr/kvm/sys/sun/conf.c`

```
{
    su_open,    su_close,    nulldev,    nulldev,    /*111*/
    su_ioctl,   nulldev,     seltrue,    0,          0,   0,
},
```

Then, the kernel calls the respective entry point function in the driver according to the system call from the user process. This causes the driver to try to perform the necessary operation. Note that the Sun SCSI Common Architecture (SCSA) implementation for a 'sun4c' Solaris 1 kernel also defines structures to keep the SCSI subsystem's own copy of the entry points, the device initialization and attach routines, and some other SCSA specific important configuration information.

There are also two standard routines, `nulldev` and `nodev`, that can be configured instead of real driver entry points. The first one does nothing, thus it silently ignores the call. The latter ignores the call also, but returns an error code. This is for calls that are to be considered as errors for a particular device driver.

When calling an entry point function of a device driver, beside the other parameters, the kernel passes also the minor number, which can be interpreted internally. The "su"

SCSI device driver uses this number to distinguish among the driver instances, i.e. since it can control up to four SCSI devices, this number shows which SCSI device, and thus, which driver instance the access is to. This can be clear when looking at Table 2.1. Each of the rsu* "special" files can be configured to represent a separate SCSI device, and, for example, an access to the 'rsu1' will generate a call to the "su" driver with minor number 1 (shown on the fifth column of the second row of Table 2.1), and will activate the driver instance 1 with the assigned SCSI device.

2.5.3 Summary of Standard Device Driver and Kernel Support Routines Used

The following Solaris 1 standard device driver routines are used in the "su" SCSI universal device driver, and are fully described in [11-13]. Note that these are not library routines, and are to be written by the device driver programmer. By a convention, the device driver routines are named as `xx_routine()`, where the `xx` is a short descriptor name chosen for a driver, e.g. an attachment routine for the "su" driver would be `su_attach()`. This also assures uniqueness of the symbols when linking the kernel.

`su_attach()`

Does a boot-time, device-specific initialization. Sets up and initializes the local data for the driver instance. It is a driver entry point.

`su_close()`

Closes the access to the device, resetting the local data. It is a driver entry point.

`su_identify()`

Requests an identification information from the SCSI device and initializes the SCSA structures accordingly. It is a driver entry point.

`su_ioctl()`

Performs requested control operations according to request codes and parameters. This is the main SCSI command and data transfer facility for the "su" device driver. It is a driver entry point.

`su_minphys()`

Determines the "chunk" size for transfers done by parts, when transfers should not tie up too much system resources. The routine returns a size less then or equal to the `maxphys` kernel label.

`su_open()`

Opens the access to the device, initializing the local data for the access time. Assures that the SCSI device is used by one user process at a time. It is a driver entry point.

`su_strategy()`

While used in block device drivers as a main transfer entry point, it is not an entry point in the "su" device driver and is used with conjunction with the `physio()` kernel support routine.

The following Solaris 1 standard kernel support routines are also used in the "su" SCSI device driver and are fully described in [11].

`copyin()`

Moves data from the user to the kernel space.

`iodone()`

Indicates the I/O complete condition by setting the `B_DONE` flag in a buffer header and wakes up a waiting process.

`iowait()`

Waits for the I/O to complete and does a `sleep` on a calling process.

`kmem_zalloc()`

Allocates a memory space from the kernel heap and fills with zeros.

`kmem_free()`

Returns an allocated space to the kernel heap.

`panic()`

Dumps a kernel core image, prints a message and trace information, and reboots the machine in case of a fatal error.

`physio()`

While usually it is used as block I/O service routine for raw (byte oriented) transfers, in the "su" driver it is used as a convenient mechanism to lock down user memory pages when performing a transfer and breaking down the data into 'chunks' with a size determined by the `su_minphys()`. Calls `su_strategy()`.

`printf()`

Kernel printing function, which outputs directly on the console. Mainly used for error messages.

`spl6()`

Sets the processor priority level to the highest (15). Used to start a critical section.

`splx()`

Resets the processor priority level. Used to end a critical section.

Note that the "su" device driver uses also some special kernel data structures (e.g. `buf` and `uio`), which are described in detail in [10,12,13].

2.6 The Sun Common SCSI Architecture (SCSA)

The Solaris 1 SCSI device driver interface for 'sun4c' and 'sun4m' kernel architectures is called Sun Common SCSI Architecture (SCSA) [9, 10, 12, 13]. The SCSA for Solaris 1 is essentially the same for Solaris 2.0 and 2.1. However, Solaris 2.2 and higher versions have some important extensions.

In the SCSA definition, the Solaris SCSI subsystem has two levels. The lower level is the Host Adapter driver (HA), which has such important functions as controlling the SCSI chip, following the SCSI low-level protocol, reserving the DMA resources, if needed, and transferring data and SCSI commands to/from SCSI devices. However, the actual SCSI data and commands are only an opaque array of bytes for the HA. It cannot perform any device-specific processing nor handle any user requests directly. The HA, depending on the OS version, can provide some advanced services like an Automatic Request Sense and a Tagged Command Queuing. The role of the HA is to provide services to the higher-level Target drivers and to handle their transfer requests concurrently.

The higher-level Target driver actually controls a SCSI device in a specific way corresponding to the type of the device. For example, the "sd" device driver Solaris 1 is the primary Sun SCSI fixed disk device driver, which handles all the block transfer requests from the kernel using some optimal disk access strategy. Target drivers never concern about actual hardware transfer details, however, they are responsible for creating meaningful command and data packets for the controlled SCSI device, and they make control decisions based on the returned status information both from the HA (about lower level transfer details) and from the SCSI device itself (e.g. media specific errors). The 'sd'

Sun disk driver, since it deals with the data sensitive system disks, can perform a global action on the SCSI bus also i.e. request to reset the whole bus in some controlled fashion.

2.6.1 Concept of a "Universal" SCSI Target Driver

Although most of the SCSI device drivers for Solaris 1 perform device-specific processing themselves, it is possible to write a driver which does not perform any specific processing itself, but has a certain way to accept arbitrary SCSI commands from a user process, pack them properly in a recognizable format and pass them to the HA for a further transfer. This "universal" Target driver can accept also a special control information such as adjusting the transfer parameters, the timeout values, setting the HA capabilities, etc. The "su" SCSI "universal" device driver, which is described in the next chapter, is such a driver. The complete lack of a decision making capability of the "universal" driver may cause some difficulties, but it is still possible to give the driver some control authority without taking away the full freedom of the custom control of the SCSI devices.

CHAPTER 3

THE TURBONET "SU" - SCSI UNIVERSAL DEVICE DRIVER FOR SOLARIS 1

The "su" - the SCSI universal device driver for the TurboNet hybrid-architecture parallel-processing system is described in this chapter. The second section describes the configuration and compilation process to create a new kernel with a built-in "su" device driver. A comprehensive reference for the driver-specific system calls and the control commands is given. The complete source code of the device driver can be found in Appendix A.

3.1 Overview

The "su" is a SCSA compliant `dev_info` style SCSI device driver, and uses data and control flow structures for the autoconfiguration process specific to the Sun Openproms interface. The driver is designed to support up to four SCSI devices, although there are not any strict limitations, and, after a very minor change, it can be compiled to handle more SCSI devices. The driver is built into the kernel and initializes during the booting process.

As it was described in the previous chapter, the "su" is a universal SCSI device driver, thus, its main function is being a link between the user process and the actual SCSI device. Almost all types of SCSI devices can be accessed and controlled by the driver, assuming that the user process knows what specific SCSI command and status information set should be used to insure the proper operation of the device.

In general, the driver accepts SCSI commands, checks for the integrity and validity, and packs and passes them to the Host Adapter (HA), which then sends them to the actual SCSI device. As it was mentioned, HA itself does not interpret the information contained in SCSI commands, which are treated as opaque arrays of bytes. After the completion of the transfer, the "su" driver returns to the user process the state and status information from HA about the actual physical transfer, as well as directly from the SCSI device the SCSI status and sense information.

Due to the specific nature of SCSI devices, only one user process can have access to a particular SCSI device using the "su" driver. All other accesses are denied until the process holding the ownership of the driver instance and, thus, the actual device, closes and releases it.

Solaris 1 defines a special user process interface - `uscsi`, for accessing some specific features of certain SCSI drivers. For example, the Solaris "sr" driver, which controls SCSI CD-ROM devices, has an `uscsi` interface to access musical compact disks, and implements a command set to control them.

For the "su" universal SCSI device driver, the `uscsi` interface, which is implemented as a `ioctl()` system call [5, 10-13], is the main transfer mechanism (see Section 3.4.3), and has some driver-specific extensions. Since the operation of SCSI devices is controlled by specific command and data sets, it would not make sense to implement 'read' and 'write' calls for a universal SCSI device driver, because for every transfer there is much more information to be passed to the device than only the data pointers and sizes. Instead, the `ioctl()` interface is used, which is conveniently programmable on the driver and the user sides, and, beside that, allows an easy extension of the functionality of the device driver. In addition to `uscsi`, the `ioctl()` interface is also used to implement all the driver specific control and set/get functions.

The "su" universal SCSI device driver has the following primary features:

- Controls up to four SCSI devices independently.
- Can support a wide range of SCSI devices including, but not limited, direct-access, sequential-access, printer, processor, scanner, WORM, CD-ROM SCSI devices, etc.
- Supports Group 0, 1 and 5 standard and vendor unique SCSI commands [1,2].
- Performs user-selectable continuous or multi-part transfers of block or byte oriented data, utilizing the DMA facility of the SPARC host board, and using a special block addressing mechanism for the multi-part transfer type when sending SCSI commands with relative block addressing.
- Retries the unsuccessful SCSI commands, if enabled, up to 256 times.
- Can utilize the SCSA Tagged Command Queuing mechanism.
- Dedicated REQUEST SENSE SCSI command mechanism for fast and easy use.
- Provides an interface to fully control the transfer parameters, including the selection of block or byte oriented, as well as the continuous or multi-part accesses, the programming of the data part size, the transfer timeout rate, the

HA packet flags, the number of retries and the logical block size. Can test and modify the HA capabilities.

- If requested, returns a detailed status information about the last SCSI transfer.

3.2 Configuring and Building a New Kernel with the "su" SCSI Device Driver

In order to use the "su" device driver in the Solaris 1 environment, a new kernel should be built and installed. This process is quite straightforward and involves the following steps (assuming that the users have some kernel configuration experience, the Solaris 1 object code license and a default system directory/file structure):

1. All the SCSI devices that are to work with the "su" driver should be connected to the system SCSI bus(es), should have unique SCSI IDs (0 to 6) and the devices at both ends of the SCSI chain should be properly terminated (see the installation and user manuals for the respective SCSI devices and the SPARC system). The target ID numbers of the devices should be noted for further use in Step 3.

   Example: The user is connecting a SCSI fixed disk (ID = 4, logical unit = 0) to the first SCSI bus of the system, and a SCSI scanner (ID = 1, logical unit = 0) to the second SCSI bus following the requirements above.

2. Only the "superuser" can perform the system reconfiguration and, therefore, "root" access privileges are required.

3. In the `/usr/sys/sun4c/conf/` directory, a new kernel configuration file should be created making a new copy of the last one, but with some new name `NEWCONFIG`. The new file should contain records about newly added SCSI devices also. The section for a particular SCSI bus in the system starts with keywords `scsibus`*n* `at esp`, where n is the number of the SCSI bus starting from 0. Several lines may follow the latter line, which define SCSI device assignments to the system SCSI drivers and their instances. For example, the line `disk su0 at scsibus0 target 4 lun 0` defines that the SCSI device with ID = 4 and logical unit = 0 (`target 4 lun 0`) on the first SCSI bus (`scsibus0`) is to be controlled by the first instance of the "su" driver (`su0`). All the newly added SCSI devices should be defined following that format.

   Example: Assume the last kernel configuration file is HYDRA, therefore the new one may be SU-HYDRA, which is an exact copy of HYDRA, but has lines commented and added:

   In the section starting with `scsibus0 at esp`, comment a line and add another one:

   ```
   # tape st0 at scsibus0 target 4 lun 0
   disk su0 at scsibus0 target 4 lun 0
   ```

   And in the section starting with `scsibus1 at esp`, comment a line and add another one too:

```
# disk sd5 at scsibus1 target 1 lun 0
disk su1 at scsibus1 target 1 lun 0
```

4. In the `/usr/sys/sun4c/conf/` directory, the file `files` should be backed up, and the original should be modified adding a line to define the source code path of the "su" driver, i.e. just after the line:

   ```
   scsi/targets/st.c        optional su scsibus
   ```

   add a line:

   ```
   scsi/targets/su.c        optional su scsibus
   ```

   Example: Copy the file `files` in the `/usr/sys/sun4c/conf/` directory to `files.nosu` and modify the original as described above.

5. The actual C source code of the "su" device driver consists of three text files - `su.c`, `sudrv.h` and `sudef.h`, which should be copied to the `/usr/sys/scsi/targets` directory for further compilation. It is preferable also to have a read-only copy of the `sudef.h` include file in the `/usr/include/scsi/targets/` directory.

6. The file `conf.c` in the `/usr/sys/sun` directory, which contains the `bdevsw` and `cdevsw` structure arrays, described in the last chapter, should be backed up and the original should be modified to contain a `cdevsw` structure specific to the "su" driver. First of all, in the file `conf.c` just before the line:

   ```
   struct cdevsw   cdevsw[] =
   ```

the following lines should be added:

```
#include "su.h"
#if NSU > 0
extern int su_open(), su_close(), su_ioctl();
#else
#define su_open         nodev
#define su_close        nodev
#define su_ioctl        nodev
#endif
```

Afterwards, the following lines should be added just after the last element of the `cdevsw` structure array but before the concluding curly close brace:

```
{
        su_open,     su_close,    nulldev,     nulldev,
        su_ioctl,    nulldev,     seltrue,     0, 0, 0,
},
```

Note the number of the new array element in the `cdevsw` structure array, which will be used in the next step. The numbers for the standard array elements are usually given in the file included in the C language comment brackets, e.g. `/*107*/`. The number of the newly added element can be easily determined by adding one to the given number of the preceding element. The resulting number is the major number of the "su" driver (see Chapter 2).

Example: Copy the file `conf.c` in the `/usr/sys/sun/` directory to `conf.c.nosu` and modify the original as described above, noting the major number, which is assumed to be 111 in this particular example.

7. The shell script `MAKEDEV.su` taking the major number as a parameter should be executed in the `/dev/` directory, and will create four device "special" files named `rsu0 - rsu3`, which will be used to access the driver and the actual SCSI devices. The files `rsu0 - rsu3` are assigned 0 to 3 minor numbers, respectively.

   Example: Copy and execute `MAKEDEV.su 111` in the `/dev/` directory, where 111 was the major number from the previous step. The `/bin/ls -l /dev/rsu*` command will show the newly created "special" files with their respective major and minor numbers.

8. The `/usr/kvm/config NEWCONFIG` command should be executed in the `/usr/sys/sun4c/conf` directory, where `NEWCONFIG` is the name of the newly created kernel configuration file from Step 3. It will result in creation of the `/usr/sys/sun4c/NEWCONFIG` directory, which will have everything necessary inside (object and some C source files) to build a new kernel with the embedded "su" device driver.

   Example: Execute `/usr/kvm/config SU-HYDRA` in the `/usr/sys/sun4c/conf` directory. Upon success, it will create the `/usr/sys/sun4c/SU-HYDRA` directory, the contents of which can be viewed by executing the commands:

   ```
   cd /usr/sys/sun4c/SU-HYDRA
   /bin/ls -l.
   ```

9. A new kernel is created by executing the `/bin/make` command in the `/usr/sys/sun4c/NEWCONFIG` directory. The new `vmunix` kernel file will have the same configuration as the previous working one; however, it will have the embedded "su" SCSI device driver also.

   Example: Change directory to `/usr/sys/sun4c/HYDRA`, and execute the `/bin/make` command. After some processing, it will create a new `vmunix` kernel file.

10. The remaining task is copying the new kernel file into the root directory, previously renaming the old one for an emergency and booting the machine. The following sequence of commands will be the safest (`NEWCONFIG` is the name of the newly created configuration and the object files directory):

```
/bin/cp /usr/sys/sun4c/NEWCONFIG/vmunix /vmunix+
/bin/cp /vmunix /vmunix-
/bin/mv /vmunix+ /vmunix
/usr/etc/shutdown -r now
```

   The last command will reboot the machine with the new kernel. If hardware and software setup is done without any mistakes, the machine will boot, and the kernel will start the "su" driver too, which will print the type and vendor information about the SCSI devices assigned to it.

3.3 Autoconfiguration and Device Driver Initialization During Boot

Each time the kernel with the embedded "su" device driver boots, and there are one or more SCSI devices configured to be controlled by the driver, it calls the `su_identify()` entry point function of the "su" driver for each configured device to check the existence of it, and possibly to get inquiry information (see Appendix A for the source code). The kernel learns about the device driver entry points from the array of the SCSA `scsi_conf` structures, which contain a basic configuration information about each SCSI device, such as HA and the target device driver entry points, the SCSI ID, the logical unit number and the SCSI bus number the device resides on. The driver entry points are represented by a pointer to the kernel `dev_ops` structure [10,12,13].

When calling the `su_identify()` function, the kernel passes a pointer of a `scsi_device` SCSA structure [10,12,13] as a parameter. This structure initially contains only the necessary information about the SCSI ID and logical unit number of the device to be controlled, and a pointer to the next similar structure for another SCSI device in the system. Actually, SCSA links all the `scsi_device` structures for all the configured SCSI devices through these pointers, where the first structure in this chain is pointed by the `sd_root` global kernel variable.

The `su_identify()` function, using the supplied information, tries to connect the actual SCSI device and check its status. The SCSA library `scsi_slave()` routine is intended for this task. It tries to remove the SCSI UNIT_ATTENTION condition of the device, most probably occurred just because of the recent SCSI bus reset during the boot process, and tries to send a SCSI INQUIRY command to get some type and vendor information about the device. Depending on the return code of the `scsi_slave()` routine, the rest of the `su_identify()` function either initializes the driver instance data for the particular device and marks the device present by notifying the kernel, or, when the SCSI device does not respond (it may be off or disconnected), marks it absent, which makes the kernel to deny access requests from user processes on the corresponding "special" file, which represents the actual SCSI device.

In the case of success, the `su_identify()` function also prepares a SCSA `scsi_pkt` structure, intended for a special procedure that quickly sends a SCSI REQUEST_SENSE command [1,2], which is assumed to be used much during a typical device operation. The SCSI REQUEST_SENSE command usually reads a useful extended state information, when the device returns a SCSI CHECK_CONDITION status.

3.4 "SU" Universal SCSI Device Driver User's Guide

The information in this section is the complete reference of the "su" device driver function calls as well as the related parameters and options. All these calls can be used from the user context, assuming that device "special" files have proper access permissions for a particular user ID.

3.4.1 Opening and Closing SCSI Devices Through the "su" Device Driver

In order to open an access to a particular SCSI device controlled by the "su" device driver, a user program should make an `open()` system call [5], which, if the access is possible, will return a positive device descriptor. The `path` parameter should specify the device "special" file corresponding to the desired SCSI device. The `flags` parameter can be one of the following two values:

`O_RDWR`

The SCSI device can be opened for the read and write operations.

`O_RDONLY`

Any data transfer from a user memory buffer to the actual SCSI device is not allowed; however all the other types of operations are allowed.

Note that the latter flag will only block the operations involving actual data transfer to a device on the SCSI bus, and will not affect such operations as setting parameters of the driver instance, or sending SCSI commands to a device.

As it was mentioned before, a "su" driver has an exclusive access lock facility, which does not allow the reopening of an already opened device. This is done because of the specific nature of SCSI devices, which use complex control mechanisms. The device can be released by the `close()` system call, which takes the device descriptor as a parameter and closes the access.

In case of errors, the `open()` system call returns -1 and sets the `errno` global variable to indicate the error. The following error conditions may occur:

`ENXIO`

The SCSI device cannot be found.

`EBUSY`

The SCSI device has been already opened.

3.4.2 "su" Device Driver Control Interface

The "su" device driver defines a special custom control facility for SCSI devices. It not only provides an interface for sending SCSI commands to a device and initiating transfers, but also for fine tuning the transfer parameters, and for performing utility functions. All the mentioned operations are implemented as requests for the `ioctl()` system call, which is well suited for these purposes.

In order to use the functions described below, a user program must include the `/usr/include/scsi/targets/sudef.h` file into the source code. This file includes and defines all the data structures and definitions necessary for taking a full advantage of the "su" device driver capabilities (see Appendix A). Since the contents of this file are the only "su" specific user interface to the device driver, every statement will be explained thoroughly.

The `sudef.h` file includes the following constant values, which mostly define some limits on the parameters that the control requests use:

`SU_SENSE_LENGTH`

Lngth of the SCSI extended sense (equals 16).

`SU_MAXUNIT_NUM`

Maximum number of the driver instances, i.e. the SCSI devices controlled by the "su" driver (equals 4).

`SU_STATUS_LEN`

Length of the returned SCSI status (equals 1).

`SU_DRIVER_VER`

Current version number of the "su" device driver.

`SU_NUM_RETRIES`

Default number of retries of an unsuccessful SCSI command (equals 16). The actual number of retries can be changed using the `SU_SET_RETRY_NUM` ioctl request.

`SU_MAX_NUM_RETRIES`

Maximum number of retries of an unsuccessful SCSI (equals 256).

`SU_MAX_CHUNK_SIZE`

Maximum size of a data part when performing a multi-part data transfer (equals 262144). The actual part size can be changed using the `SU_SET_CHUNK_SIZE` ioctl request.

`SU_MIN_CHUNK_SIZE`

Minimum size of a data part when performing a multi-part data transfer (equals system logical block size `DEV_BSIZE`, which is 512).

`SU_MAX_TIME_RATE`

Maximum timeout rate (equals 10). The timeout rate and the ioctl request associated with it will be described below.

`SU_MAX_CBD_LEN`

Maximum size of a SCSI command accepted, limited by the Group 5 commands (equals 12).

The "su" device driver implements 13 ioctl requests, which provide a full control of the assigned SCSI device as well as the transfer and the access parameters. The generic form of the C language statement to issue an ioctl request to the "su" driver is:

```
err = ioctl(dev, SU_XXXXX, data_ptr);
```

where dev is the device descriptor, returned by a previously called open() system call, and indicates that this ioctl() system call is to refer to a particular SCSI device. The SU_XXXXX parameter is the defined keyword of the request, and the data_ptr is the pointer to the data structure corresponding to the request. In most cases, the device driver uses certain fields of the structure pointed at by the data_ptr to perform the request and/or fills them with the resulting values when returning. The return value of err for a successful ioctl() call is 0, while the value of -1 indicates error and the errno global variable is set to indicate the type of it. The following error conditions are possible:

ENXIO

The SCSI device cannot be found.

ENODEV

The SCSI device is not open.

EFAULT

Bad address (usually a null data pointer).

EINVAL

Invalid argument or argument is not in its range.

ENOTTY

Invalid request code.

In addition of these, the USCSICMD and SU_REQUEST_SENSE request keywords, since they actually transfer data through the SCSI bus, may return additional error conditions, which will be noted when describing each keyword.

3.4.3 Description of the "su" Device Driver Control Requests

The comprehensive description of the control requests follows, which includes the defined keyword of the request, the related data structure, if applicable, and the description of the operation. For a correct operation of a control request, a pointer to a properly allocated data structure, given in the "Specific Data Structure" section of a request description, should be passed as the third parameter (`data_ptr`) of the `ioctl()` system call,

Request Keyword:

`SU_RESET_DEV`

Specific Data Structure:

none

Description:

Resets the SCSI device to its initial state.

Request Keyword:

`SU_GET_INQ_DATA`

Specific Data Structure:

`struct scsi_inquiry`

Description:

Returns the inquiry data about the device type, the product name, the revision number, the vendor information and the other device specific parameters acquired during the device driver initialization (see the section 3.3). The format of the inquiry data is described in [1,2], and the definition of the `scsi_inquiry` SCSA structure can be found in the `/usr/sys/scsi/generic/inquiry.h` file in a Solaris 1 environment.

Request Keyword:

`SU_REQUEST_SENSE`

Specific Data Structure:

`struct scsi_extended_sense`

Description:

Sends the SCSI REQUEST SENSE command to the device and returns the SCSI extended sense information for the `SU_SENSE_LENGTH` bytes. The format of the extended sense data is described in [1,2], and the definition of the `scsi_sense` SCSA structure can be found in the `/usr/sys/scsi/generic/sense.h` file in a Solaris 1 environment.

This request is usually sent, when, as a result of the previous SCSI operation, the device responded with the SCSI CHECK_CONDITION status. Since this situation happens quite frequently, it is convenient to have a fast built-in mechanism for requesting the sense information without the need of forming a new SCSI command for this purpose and sending it from a user process.

Usually, when returning the sense, the SCSI device sets the sense key field of the extended sense data format to an appropriate condition code. The following is the list of the most common sense keys, although every vendor can define its own ones depending on the type of the device or the operating modes.

Table 3.1 The most common SCSI extended sense keys

Condition	**Hex Code**
Recoverable Error	0x01
Not Ready	0x02
Medium Error	0x03
Hardware Error	0x04
Illegal Request	0x05
Unit Attention	0x06
Write Protect	0x07
Blank Check	0x08
Vendor Unique	0x09
Copy Aborted	0x0A
Aborted Command	0x0B
Equal	0x0C
Volume Overflow	0x0D
Miscompare	0x0E
Reserved	0x0F

Request Keyword:

```
SU_GET_CAP
```

Specific Data Structure:

```
struct su_dev_cap {
        int     cap;
        int     value;
};
```

Description:

Gets a value of the requested capability parameter of HA for the SCSI device. The HA has several operating parameters called capabilities. In order to get the value of the particular capability, the user program should set the `cap` field of the `su_dev_cap` structure to the number of the capability and perform the request. If the call is successful, the driver will return the capability value set in the `value` field.

Table 3.2 The Host Adapter capabilities

SCSI_CAP_DMA_MAX	0
SCSI_CAP_MSG_OUT	1
SCSI_CAP_DISCONNECT	2
SCSI_CAP_SYNCHRONOUS	3
SCSI_CAP_WIDE_XFER	4
SCSI_CAP_PARITY	5
SCSI_CAP_INITIATOR_ID	6
SCSI_CAP_UNTAGGED_QING	7
SCSI_CAP_TAGGED_QING	8

Request Keyword:

```
SU_SET_CAP
```

Specific Data Structure:

```
struct su_dev_cap {
        int     cap;
        int     value;
};
```

Description:

Sets a value of the requested capability parameter of HA for the SCSI device.

For information about the HA capabilities, refer to the description of the `SU_GET_CAP` control request. In order to set the value of the particular capability, the user program should set the `cap` field of the `su_dev_cap` structure to the number of the capability, as well as the `value` field to the desired value for it, and perform the request. If the system call is not successful for any reason, it will return -1 with `errno` set to `EINVAL`.

Note that not every HA capability can be altered. Since the capabilities may affect the operation of the low-level SCSI protocol, a great care should be taken when modifying the system set capabilities.

Request Keyword:

```
SU_GET_RESULTS
```

Specific Data Structure:

```
struct su_results {
        long     resid;
        u_char   reason;
        u_char   state;
        u_char   statistics;
};
```

Description:

Gets detailed state and status information about the last SCSI transfer. The fields of the `su_results` structure represent actual values returned by HA. Thus, having this information, the user program can keep track of the actual transfer process and make appropriate decisions.

The `resid` field of the `su_results` structure returns the number of data bytes not transferred. After a successful transfer, this field should be equal to 0. Another important status indicator of the SCSI transfer is the `reason` field, which shows the reason of a SCSI transfer failure, if any. The value of 0 means a successful transfer. The `reason` field can show the following failure reasons defined in the `/usr/sys/scsi/scsi_pkt.h` file in a Solaris 1 environment:

Table 3.3 The SCSI transfer failure reasons

Reason	**Code**	**Description**
CMD_CMPLT	0	No transport errors- normal completion

CMD_INCOMPLETE	1	Transport stopped with not normal state
CMD_DMA_DERR	2	DMA direction error occurred
CMD_TRAN_ERR	3	Unspecified transport error
CMD_RESET	4	SCSI bus reset destroyed command
CMD_ABORTED	5	Command transport aborted on request
CMD_TIMEOUT	6	Command timed out
CMD_DATA_OVR	7	Data Overrun
CMD_CMD_OVR	8	Command Overrun
CMD_STS_OVR	9	Status Overrun
CMD_BADMSG	10	Message not Command Complete
CMD_NOMSGOUT	11	Target refused to go to Message Out Phase
CMD_XID_FAIL	12	Extended Identify message rejected
CMD_IDE_FAIL	13	Initiator Detected Error message rejected
CMD_ABORT_FAIL	14	Abort message rejected
CMD_REJECT_FAIL	15	Reject message rejected
CMD_NOP_FAIL	16	No Operation message rejected
CMD_PER_FAIL	17	Message Parity Error message rejected
CMD_BDR_FAIL	18	Bus Device Reset message rejected
CMD_ID_FAIL	19	Identify message rejected
CMD_UNX_BUS_FREE	20	Unexpected Bus Free Phase occurred

The `state` field reflects the actual stage of the low-level SCSI transfer that HA was in when the failure occurred, if any. As far as HA proceeds with a SCSI transfer, it fills the corresponding bits of an internal variable, which then is represented by the `state` field after the transfer. The value of 0x1f means a successful transfer. The Solaris 1 environment defines the following stages of the low level SCSI transfer:

Table 3.4 The SCSI transfer stages

Stage	Hex Code	Description
STATE_GOT_BUS	0x01	SCSI bus arbitration succeeded
STATE_GOT_TARGET	0x02	Target successfully selected
STATE_SENT_CMD	0x04	Command successfully sent
STATE_XFERRED_DATA	0x08	Data transfer took place
STATE_GOT_STATUS	0x10	SCSI status received

The `statistics` field adds some information to the main status given by the previous fields. Only the following values are defined in the Solaris 1 environment:

Table 3.5 Some SCSI transfer statistics

Event	Hex Code	Description
STAT_DISCON	0x1	Command experienced a disconnect
STAT_SYNC	0x2	Command did a synchronous data transfer
STAT_PERR	0x4	Command experienced a SCSI parity error

Request Keyword:

`SU_SET_CHUNK_SIZE`

Specific Data Structure:

`long chunk_size`

Description:

Sets the size of the part for the internal part-by-part transfer option. This size should be in multiples of the device logical block size, if using the block transfer option. The lower and upper limits for the size are set by the `SU_MIN_CHUNK_SIZE` and `SU_MAX_CHUNK_SIZE` constant values, respectively. The part size initially is set to the system default size, and can be reset anytime by issuing the `SU_SET_CHUNK_SIZE` request with a 0 argument.

Request Keyword:

`SU_SET_RETRY_NUM`

Specific Data Structure:

`int retry_num`

Description:

Sets the number of retry operations for a failed SCSI command. If set to 0, no retries will be performed. The upper limit for this number is the `SU_SET_RETRY_NUM` constant value.

Request Keyword:

`SU_GET_SCSI_ID`

Specific Data Structure:

```
struct su_scsi_id {
        u_char  bus_id: 4,
        u_char  dev_id: 4;
};
```

Description:

Returns SCSI ID information about the controlled device. The `bus_id` field of the `su_scsi_id` structure contains the bus ID, which is a number showing which SCSI bus the device is connected to in the given system. The `dev_id` field contains the SCSI target ID for the device.

Request Keyword:

`SU_SET_TIME_RATE`

Specific Data Structure:

```
int time_rate
```

Description:

Sets the SCSI command timeout rate. The argument for this request should be a positive number. The actual time in seconds for a SCSI command to be completed without causing a timeout is calculated then by the formula:

$$allocated\ time = 10 + \frac{number\ of\ bytes\ to\ transfer\ *\ timeout\ rate}{512},$$

where the *timeout rate* is the value set by this control request, and is initially set to 1.

As it can be seen from the formula, the SCSI device, beside the specified allocated time for each 512 bytes, has 10 seconds additionally allocated for preparation operations, e.g. getting on-line, seeking access, etc.

Request Keyword:

`SU_SET_PKT_FLAGS`

Specific Data Structure:

`long pkt_flags`

Description:

The internal process of sending a SCSI command to the device involves a step of preparing a special packet of information about all the transfer attributes and passing it to HA, which processes it accordingly. A certain field in this SCSA defined packet structure is intended for special flags, which tell HA how to perform the transfer request included in the packet.

The `SU_SET_PKT_FLAGS` control request allows the user program to control this aspect of the transfer process too. They are set to 0 upon initializing the device driver. Once set, the flags are attached to every outgoing packet unless modified or reset. The flags are also used to control the HA tagged-command-queuing capability. In the Solaris 1 environment, the following flags are accepted by the device driver:

Table 3.6 The SCSA packet flags

Packet flag	Hex Code	Description
FLAG_NOINTR	0x0001	Run command without interrupts

FLAG_NODISCON	0x0002	Run command without disconnects
FLAG_SUBLUN	0x0004	Use the `sublun` field in `pkt_address`
FLAG_NOPARITY	0x0008	Run command without parity checking
FLAG_HTAG	0x1000	Run as HEAD OF QUEUE tagged command
FLAG_OTAG	0x2000	Run as ORDERED QUEUE tagged command
FLAG_STAG	0x4000	Run as SIMPLE QUEUE tagged command

Request Keyword:

`SU_SET_BLOCK_SIZE`

Specific Data Structure:

`long block_size`

Description:

Sets the size of the logical block used when performing a block transfer. This size preferably should be equivalent to the logical block size of the SCSI device, although can be different for some special applications. The initial logical block size is set to 512.

Request Keyword:

```
USCSICMD
```

Specific Data Structure:

```
struct uscsi_cmd {
        caddr_t          uscsi_cdb;
        int              uscsi_cdblen;
        caddr_t          uscsi_bufaddr;
        int              uscsi_buflen;
        unsigned char    uscsi_status;
        int              uscsi_flags;
};
```

Description:

Sends a SCSI command to the device, and, if applicable, performs the actual data transfer. This request is the main mechanism to control the SCSI device, and as an argument uses a pointer to a `uscsi_cmd` structure originally defined in the `/usr/sys/scsi/impl/uscsi.h` file in the Solaris 1 environment.

In order to form a correct SCSI command and specify proper parameters and flags for a desired SCSI operation, a user program should satisfy the requirements for values of all the fields of the `uscsi_cmd` structure.

The first field of the `uscsi_cmd` structure is `uscsi_cdb`, which is to point to a properly allocated array of bytes representing the actual SCSI command [1,2]. As it was mentioned above, the 'su' device driver accepts only Group 0,1 and 5 SCSI commands, therefore the next `uscsi_cdblen` field of the structure, which is to contain the SCSI command length, can accept only the values 6, 10 and 12, respectively. Note that the first byte of a SCSI command specifies the command group, and any inconsistency between the latter and the `uscsi_cdblen` field value will result to an `EINVAL` error condition.

If the specified SCSI command is to perform a data transfer, the valid data buffer pointer and the buffer length should be given in the `uscsi_buffaddr` and `uscsi_buflen` fields, respectively. Otherwise, the `uscsi_buflen` fields should contain 0 to insure correct operation.

The `uscsi_flags` field is intended to contain operational flags, which determine the transfer modes and options. Currently, the "su" device driver accepts two SCSA defined and three "su" driver specific flags. The description below contains their defined keyword, the hexadecimal code representing the actual bit position, as well as their effect onto the USCSI request processing.

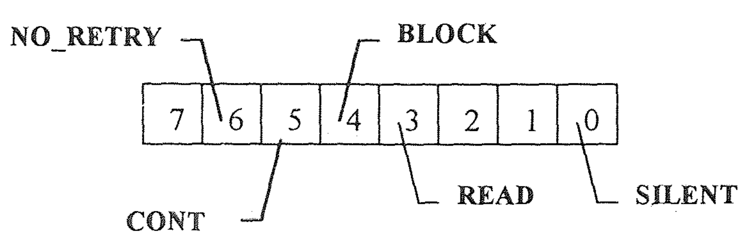

Figure 3.1 Operational flags in the `uscsi_flags` field defined for the "su" driver

`USCSI_SILENT`   `0x01`

SCSA defined flag. Causes the device driver not to print error messages on the console, and thus, to the system log.

`USCSI_READ` `0x08`

SCSA defined flag. If set, specifies a "read" operation from the SCSI device. Note that for "write" operation, as well as for the no-data SCSI commands this bit should be 0.

`USCSI_BLOCK` `0x10`

A "su" specific flag. Must be set when sending a block transfer SCSI command. Enforces the device driver to interpret the address and data count fields [1,2] in a SCSI command as a logical block address and a logical block count, respectively. The number in the `uscsi_buflen` field, as well as the data part size (if transferring data by parts) set by the `SU_SET_CHUNK_SIZE` request, must be in multiples of the device's actual logical block size, which should be the same as the block size set by the `SU_SET_BLOCK_SIZE` request.

The device driver will transfer data by parts, unless the `USCSI_CONT` flag is set. When transferring data by parts, a request with a SCSI command with relative block addressing [1,2] will create several actual SCSI commands (for each data part); the address and block counts will be set automatically by the driver.

`USCSI_CONT` `0x20`

A "su" specific flag. If set, this flag makes the device driver not to transfer data by parts, which may boost the transfer speed, but may also affect the overall system performance because it would tie up such system resources during the transfer, as the SCSI bus and DMA. Beside that, transfers of large amounts of data cannot be performed without dividing them into parts, because of the limited DMA resources, and will generate error conditions.

The SCSI commands, not following the generic format for the Group 0, 1, and 5 commands [1,2], as well as the vendor specific commands, which have different meanings for the address and data count fields, must be sent with the `USCSI_CONT` flag to ensure the desired operation. Failure to do so may cause unexpected results. This is due to the generic nature of the "su" device driver.

`USCSI_NO_RETRY          0x40`

A "su" specific flag. Disables the unsuccessful SCSI command retry capability of the "su" device driver. If not set, the driver will retry the failed SCSI command until encountering a success, but not more than the maximum number of times, which is 16 by default, and can be changed by the `SU_SET_RETRY_NUM` request.

It should be noted that SCSI commands with the relative block addressing are preferable to send with this flag set, because retrying such commands may sometimes cause unexpected results.

The `uscsi_status` field of the `uscsi_cmd` structure is intended to contain a SCSI status byte [1,2], which is sent by the device after processing a SCSI command. It usually represents the result of the last SCSI operation. In some applications, the most frequently encountered case is the SCSI CHECK_CONDITION status, which shows that the device detected an error during processing the command and has more detailed status information, which can be retrieved by the SCSI REQUEST_SENSE command.

For certain SCSI commands, even if the `USCSICMD` control request returns 0, indicating that the transfer was successful, it is strongly advisable for a user program to check the SCSI status and to perform an appropriate action, before sending another SCSI command to the device.

On failure, since the `ioctl()` system call with the `USCSICMD` request deals with physical I/O, it returns -1 and, beside the standard error codes mentioned above, may set some specific error codes in the `errno` global variable:

`EACCES`

"Write" operation is not permitted.

`ENOMEM`

Kernel memory allocation failed.

`EBUSY`

SCSI device is busy.

`EIO`

Input/output error.

CHAPTER 4

THE "HSCSID" - HOST LEVEL SCSI REQUEST SERVER DAEMON

The "su" SCSI universal device driver, described in the previous chapter, provides a full SCSI device control interface only to the processes running on the host side of TurboNet. In order to access the provided SCSI control services from the Hydra side as a SCSI request initiator, a special host-Hydra interprocess communication protocol has been developed, which converts the complete "su" device driver interface into a convenient mechanism that can be used by programs running on the Hydra boards. This chapter describes the host-Hydra interprocess communication protocol.

4.1 The "hscsid" UNIX Daemon Process

4.1.1 Overview

The host-Hydra interprocess communication protocol is implemented with a Solaris 1 "hscsid" daemon, which is fully interrupt driven. As some other UNIX daemons, the "hscsid" forks and, disassociating itself from the controlling terminal, sleeps waiting for signals to process. Multiple "hscsid" daemons may be configured to start at the machine boot time, although they may be started and killed at any desired time. The maximum number of simultaneously running "hscsid" daemons depends on the system resources. A variety of daemon-host program-Hydra program combinations can be used for specific applications.

Each "hscsid" daemon can gain access to any of the SCSI devices controlled by the "su" device driver upon requests from the assigned Hydra board. All the communication between the daemons and the Hydra boards is carried out using the shared memory of the boards.

The "hscsid" SCSI server request daemon has the following primary features:

- Can access up to four SCSI devices controlled by the "su" device driver.
- Uses a 256-byte dedicated SCSI request block in the Hydra shared memory as a host-Hydra shared data structure.
- Handles 14 SCSI requests of the host-Hydra interprocess communication protocol.
- Performs buffered SCSI transfers for parallel Hydra DSP and SCSI operation using the "su" SCSI universal device driver.
- Uses a memory mapped fast access to the whole Hydra shared memory.

The complete source of the "hscsid" SCSI request server daemon can be found in the Appendix B.

4.1.2 The Operation Sequence

The actual host-Hydra interprocess communication protocol consists of ten steps, as follows. Note that the protocol uses some steps involving the Hydra device driver and the HydraMon monitor on Hydra boards, which are standard for the Hydra product, and are marked with the "*" .

1. The Hydra DSP program fills the shared request structure appropriately.

2*. It invokes the HydraMon VME interrupter service by calling trap 7 and continues the program execution.

3*. HydraMon generates a VME interrupt with assigned number and vector.

4*. The Hydra device driver catches the Hydra interrupt and sends an assigned UNIX signal to a host side user program registered to receive the signal.

5. The host-side user program, which has the UNIX process ID of the running "hscsid" daemon, sends a UNIX `SIGUSR1` signal to it, if it is appropriate (beside the SCSI services, the interrupts can be used for other purposes too).

6. The daemon, which might have been sleeping, activates its interrupt handler function upon receiving the signal, and analyses the SCSI request block in the Hydra shared memory, which was previously mapped in the daemon's virtual memory space. Any other `SIGUSR1` signals are blocked until the interrupt handler function returns.

7. The daemon tries to perform the requested service by calling 'su' SCSI universal device driver. If a SCSI transfer is requested, the transfer handler function transfers data from the SCSI device to the Hydra shared memory or vice versa. Upon completion, the daemon sets status and error information, and sends a Hydra interrupt request to the Hydra device driver.

8*. The Hydra device driver requests the HydraMon to invoke the UserInt service, which is intended to call user registered functions on Hydra with specified trap numbers.

9*. HydraMon calls the given trap number to invoke the previously registered SCSI interrupt handling function in the Hydra DSP program.

10. The SCSI interrupt handler function analyses the status information in the shared request structure and performs an appropriate action.

4.1.3 Command Line Arguments

The "hscsid" daemon should be started with two arguments:

```
hscsid dsp trapnum
```

where the `dsp` argument should be a "special" file name representing a DSP on a Hydra board. All accesses to a Hydra board are done using this "special" file.

The second argument, `trapnum`, specifies the DSP trap number that will be used when requesting a DSP interrupt upon returning status and error information to the DSP specified with the first argument. For proper operation of the daemon, the specified DSP should run a program, which registers an interrupt handler function with the specified trap number. The argument can be specified as a decimal, octal (as \XXX), or hexadecimal (as 0xX) number.

Example: `hscsid /dev/vc40a1 0x9`

Note that for Hydra boards of up to revision E only four trap numbers are available - 0x9 through 0xC.

4.2 Hydra SCSI Requests

4.2.1 Request Block Structure

The SCSI request block is 256 bytes long and resides in the Hydra shared memory, just before the space reserved by HydraMon, which takes 16640 bytes at the top of the memory [4].

The actual request data structure is defined in the `shared.h` include file as follows:

Table 4.1 Hydra SCSI request data structure

```
struct hydra_request {
        unsigned long           dev_num;
        unsigned long           req_code;
        unsigned long           params[PARAM_NUM];
        unsigned long           result;
};
```

The `dev_num` field specifies the SCSI device, and, thus, the "su" device driver instance to be used. For the current implementation of the "su" driver, this field can contain a value from 0 to 3.

The second field, `req_code`, contains the code for an operation requested, which will be described in details below.

The request parameters are specified in the `params[]` array, which has 61 elements in the current implementation. The array is intended to pass the operating arguments for the requests, as well as to get the status and other types of information. For each request, only a certain subset of the array elements is used, while the others are simply ignored.

After completion of a request, the "hscsid" daemon sets the `result` field of the structure with a number representing the status of the completion. For successful calls, it contains 0. Requests with illegal request codes return -1. All other values should be treated as error codes corresponding to the `errno` global variable set by the "su" device driver (See Sections 3.4.2 and 3.4.3).

As it can be easily seen, this request structure is universal enough to be used with any kind of host-Hydra interprocess communication protocol, which can be either interrupt driven or status polling. New request codes can be added to the existing ones, expanding the operational capabilities (not only for SCSI).

4.2.2 Request Codes

Since the Hydra SCSI requests reflect the actual "su" device driver control requests, there is no need to describe each one again, and detailed information about the parameter assignments can be found in the printout of the include file `shared.h` (see Appendix B).

The following is a brief description of the Hydra SCSI requests, for which there is additional information:

Request Keyword

`CLOSE_DEV`

Description

Closes the SCSI device specified in the `dev_num` field of the request block. Useful, because the "hscsid" daemon will never close the device itself, if not requested so, holding the device ownership and preventing other users from using the device. Note that this is not a "su" device driver control request, and it only closes the device using the `close()` system call.

Request Keyword

`GET_INQ_DATA`

Description

Invokes the `SU_GET_INQ_DATA` "su" device driver control request and returns some useful fields from the SCSI inquiry structure. For the complete inquiry data returned from a device, a DSP can request `SCSI_CMD` with the SCSI INQUIRY command.

Request Keyword

`REQUEST_SENSE`

Description

Invokes the `SU_REQUEST_SENSE` "su" device driver control request. Note that the pointer to the `params[0]` can be casted as a pointer to a `scsi_extended_sense` structure (from `scsi_incl.h` file), which would be very convenient to work with.

4.3 Accessing and Activating the "hscsid" Daemon

After the "hscsid" daemon has been started, it can be either in the idle (sleeping, swapped) or in the active states. In order to activate the daemon, the host side program of the host-Hydra program pair should send a `SIGUSR1` signal to the daemon using its UNIX process ID. Upon receiving the signal, the daemon will assume that there was a SCSI request from the DSP specified with the first argument on its command line, and will start analyzing the request block and try to perform the specified request, after completion of which (successful or not) it will request to interrupt the DSP for returning the status and error information.

If it is necessary to terminate the daemon, the user side program can send any of the `SIGTERM`, `SIGINT`, `SIGHUP` and `SIGQUIT` signals to terminate it. The UNIX `kill pid` command can be used from a UNIX shell prompt or from within shell scripts, where the `pid` is the process ID of the daemon.

Since the daemon may be configured to start when booting, any system-related error messages, e.g. failures to open devices or map memory blocks, are logged into the system log, which is usually the `/var/adm/messages` file in the Solaris 1 environment.

CHAPTER 5

HYDRA SCSI UTILITIES AND DATA STRUCTURES

This chapter presents several useful simple utilities and data structures, which make the SCSI access programming much easier for Hydra boards. Solaris 1 has very convenient definitions for SCSI data structures, which can also be included in Hydra SCSI programs. Some terminal output utilities are also described.

5.1 Source Files

In order to make use of the defined Hydra SCSI utilities and data structures, a Hydra DSP program source should include the `hy_scsi_util.h` file, which contains the definitions and include files for all the utilities and data structures. Beside that, the `hy_scsi_util.c` file and `hydra_int.asm` file should be compiled, assembled and linked with the actual user program. The complete source code for these and some other included files can be found in Appendix C. Reference [14] is a comprehensive user's guide for the C compiler and linker for Hydra DSPs. The `scsi_incl.h` file includes many useful data structures and definitions reflecting the Sun SCSI implementation, which can be used in Hydra DSP programs.

Due to the open architecture, the Hydra DSP programmers can create their own application specific routines and data structures, as well as host-Hydra interprocess communication protocol requests.

5.2 Utility Routines and Definitions

5.2.1 Working with TMS320C40 Built-In Timer

These definitions can be used as routines in C code, and are intended to access the DSP built-in timer:

Function

```
RESET_TIMER()
```

Description

Resets the Timer 0 of a DSP.

Function

```
SET_PERIOD(p)
```

Description

Sets period of the Timer 0 of a DSP. For a detailed timer description and operation modes, see [3].

Function

```
GET_TIMER()
```

Description

Returns the current value in the Timer 0 of a DSP.

Function

```
ELAPSED_TIME(st, end)
```

Description

Used for measuring time intervals. When the `st` argument is the timer value at the beginning of an event, and the `end` argument is the value at the completion of the event, the defined function returns the elapsed time in microseconds.

5.2.2 DSP Control Functions

In order to utilize the Hydra SCSI protocol, a DSP user program should access and modify some system and processor attributes or generate interrupts. These simple functions are actually implemented in the TMS320C40 assembly language, but can be called from a C program if correctly assembled and linked (see Section 5.1). They implement such operations as installing DSP trap handlers, enabling/disabling DSP interrupts, getting Hydra board configuration, etc.

Function

```
void GIE_On()
```

Description

Enables DSP interrupts.

Function

```
void GIE_Off()
```

Description

Disables DSP interrupts.

Function

```
void EnCache()
```

Description

Enables DSP cache.

Function

```
void SetIntVect(trapnum, handler_function)
```

Description

Installs a trap handler function into the DSP trap table. The user is restricted to use only traps 0x9 through 0xC for the `trapnum` argument, which is the trap number. The `handler_function` argument is the function to be called when the corresponding trap is called. The Texas Instruments TMS320 C compiler [14] restricts the function name definition for the trap handler functions. They must be declared as `c_int`*`nn`*`()`, where *nn* is a two digit number. There are some other restrictions also, but if the function complies with the conventions of the TMS320 C compiler, there should not be any problems.

Function

```
void HostInt()
```

Description

Generates a VME interrupt, which can be delivered to a specified user program on the host side as a UNIX signal.

Function

```
hydra_conf *GetConfig()
```

Description

Returns a pointer to a data structure, which is defined in the Hydra include files, and represents the Hydra board configuration. It contains information about the board revision level, the memory sizes, the CPU clock frequency, the settings for serial ports, etc.

Function

```
void Idle()
```

Description

Stops DSP program execution, which can be resumed, if the DSP is reset or any interrupts arrive. Useful for sleep-and-wait-for-interrupt situations.

5.2.3 Terminal Output Functions

These functions make it possible for a DSP user program to output some information and error messages to a "dumb" terminal using the Hydra's RS-232 port. Two generic and three SCSI specific messaging functions are available. Input facilities are not included yet.

Function

```
void c40_putchar(char c)
```

Description

From the original Ariel Hydra library. An implementation of the C `putchar()` function.

Function

```
void c40_printf(char *fmt, ...)
```

Description

From the original Ariel Hydra library. It is a limited implementation of the C `printf()` function, and outputs to the Hydra RS-232 port. Note that in the `fmt` argument, the function only recognizes the `%d, %f, %x, %c, %s` formats and the `\n` escape character.

Function

```
void c40_perror(char *msg)
```

Description

Prints an error message containing the message specified with the `msg` argument, followed by an error number and a brief error description. The error description strings and the corresponding error numbers can be found in the `hy_scsi_util.c` file (see the Appendix C).

Function

```
void perror_scsi()
```

Description

Prints an error message about a SCSI request additionally containing the request code.

Function

```
void print_sense(s)
```

Description

Useful when printing a message about returned SCSI sense key [1,2]. Prints a descriptive message about the SCSI sense key specified with the `s` argument.

5.3 Useful SCSI Data Structures

An incredible amount of SCSI data structures can be found in the `scsi_inc.h` file (see Appendix C), almost all contents of which are originally copyright of Sun Microsystems, Inc. The data structures are usable with the TMS320C40, which always accesses data as long words. Some other compatibility problems are corrected too.

CHAPTER 6

PERFORMANCE RESULTS AND CONCLUSIONS

Some performance results of the Hydra SCSI services are presented here for multi-user environments of the TurboNet computer. The chapter also draws conclusions and presents further research objectives.

6.1 Performance Results

In order to measure the performance of the Hydra SCSI services, a test Hydra DSP program is written (see Appendix D) which uses the SCSI services to access an experimental 100 Mbyte SCSI fixed disk connected to the SCSI bus of the host computer. The time measurements are done using the DSP's built-in timer. The results have 0.0001 sec accuracy in the worst case.

The test program performs three different types of accesses: non-SCSI, SCSI control/info, and SCSI transfer. The total time elapsed for every access includes the processing times for all three levels of the Hydra SCSI services, from the instant the DSP requests a SCSI service through the instant it gets interrupted as a notification of service completion. All three types of accesses are processed through the host level "hscsid" SCSI request server daemon and the "su" SCSI universal device driver (see section 4.1.2 for the detailed operation sequence).

As an example of a non-SCSI request, the GET_INQ_DATA SCSI service is used, which does not cause an access to the test SCSI fixed disk. However, it does access the "su" driver to get previously acquired SCSI inquiry data.

The second type of SCSI access, namely SCSI control/info, is represented in the test program by the SCSI READ CAPACITY command [1,2], which returns 4 bytes containing the disk capacity parameters.

In order to test the typical Hydra SCSI throughput in different multi-user environments for typical SCSI configurations (i.e. not only disks), a 18 ms access time (i.e. relatively slow) SCSI fixed drive is used, daisy-chained with the system disks on the first SCSI bus of the host computer. The program transfers 3 Mbytes of the Hydra shared memory to the fixed disk and vice versa. The transfer is done in parts (see Chapter 3), the size of each part being set to the system default size for the first two transfers, and to 128 Kbytes for the last two transfers.

When testing the performance, three different typical multi-user system load environments are simulated. The fully-configured Solaris 1 system is connected to the Ethernet network and has mounted NFS partitions. The first series of tests (named low-load) are performed when only one user is logged in using the console shell. The second environment (named mid-load) is simulated with three users logged in and running two processes remotely, and a fourth one using the Sun Open Windows interface on the console running 15 processes, including the shells. Although the third environment (named high-load) has the same number of users running the same number of processes as the second one, in addition three processes concatenate in the background 20 files of 1.5-Mbytes each using the UNIX `cat` command.

The test program is also a good example to give users an idea of how to write DSP programs using the implemented Hydra SCSI services. Appendix D contains the complete sources of the host side and the Hydra side programs.

The following is the comparison table of the performance results for the Hydra SCSI services in the abovementioned different environments using one SCSI device with a typical speed. The results are averaged in each category.

Table 6.1 Some performance results of the Hydra SCSI services in terms of elapsed time

Request Type / System Load	Low-Load	Mid-Load	High-Load
Non-SCSI, μs	604.53	604.55	4066.5
SCSI control/info, μs	3200.7	4620.5	5545.5
3-Mbytes SCSI write (default part size), s	5.5743	5.8554	6.7052
3-Mbytes SCSI read (default part size), s	5.1919	5.4327	5.8465
3-Mbytes SCSI write (128-Kbytes part size), s	5.6868	6.2179	6.6644
3-Mbytes SCSI read (128-Kbytes part size), s	5.1865	5.4416	5.8913

As we can see from the table, and as other experiments show, reducing the part size for transfers results in increased transfer times. This happens because of the overhead caused by the pauses among consecutive part transfers. Note that an extensive increase of this part size for achieving better performance is not possible in a Solaris 1 environment, because of system setup and operation limitations. Doing so would cause continuous DMA or other system failures, not mentioning compromising the access performance of other devices on the SCSI bus.

The best approach to achieve maximum performance is finding the optimal number of bytes that a particular system can transfer at a time, which depends on many factors including DMA capability and the amount of memory the kernel can allocate for this purpose. The optimal transfer size would be in multiples of the SCSI device logical block size, if it is a block device. Although the "su" device driver provides a continuous data transfer option using the `USCSI_CONT` flag (see Chapter 3), usage of it should be avoided for large amounts of data, again because of the reasons indicated above.

6.2 Conclusions and Further Research Objectives

The Hydra SCSI universal services for TurboNet are powerful tools intended to expand the I/O capabilities for Hydra DSP programs. The actual goal of this work was to design universal tools, which can be easily used, modified and expanded. The device driver services and the host-Hydra intercommunication protocol can be customized to suit the user's specific needs. Due to the universal approach of the services, a large number of SCSI devices can be accessed and controlled. Since the SCSI command protocol is sometimes complex and differs from device to device, different access modifiers are provided to ensure maximum coverage of the controllable SCSI devices for the Hydra SCSI services.

TurboNet is a parallel processing system that often requires parallel I/O solutions for high performance. The current version of the "su" device driver supports up to four SCSI devices, which can be accessed simultaneously from the Hydra boards assuming that the needed number of the "hscsid" SCSI request server daemons run. Since the DSPs in the system will use some shared resources, such as the SCSI request blocks or the actual data buffers in the shared memories, there may be certain exclusive lock mechanisms to ensure data integrity and correct operation.

Although the software set described in this work provides a low-level access to the SCSI devices, it provides full control over all the SCSI device aspects, which is as important as the use of assembly language to work with a system hardware.

Future research objectives include topics such as: parallel I/O using several DSPs to control respective SCSI devices; including a SCSI port attached directly to the VME bus and development of accompanying software (drivers, etc.); and creating and fine-tuning a special configurable parallel filesystem for different DSP interconnection schemes.

APPENDIX A

C SOURCE OF THE "SU" SCSI UNIVERSAL DEVICE DRIVER FOR SOLARIS 1

```
/*
 * SU - SCSA compatible universal SCSI driver
 * Definitions and user data structures
 * sudef.h 12/11/95
 * Artak O. Melkonian, All Rights Reserved, 1994, 1995
 * Department of Electrical and Computer Engineering
 * New Jersey Institute of Technology
 */

#ifndef    _scsi_targets_sudef_h
#define    _scsi_targets_sudef_h


#include <scsi/scsi.h>
#include <scsi/impl/uscsi.h>

#define SU_SENSE_LENGTH    SENSE_LENGTH    /* length of extended
sense */
#define SU_MAXUNIT_NUM     4     /* max number of devices */
#define SU_STATUS_LEN 1    /* length of status code */
#define    SU_DRIVER_VER   "1.0" /* current version */
#define SU_NUM_RETRIES     16    /* defualt num of retries */
```

```
#define SU_MAX_NUM_RETRIES 256  /* maximum num of retries */
#define SU_MAX_CHUNK_SIZE  1024*256   /* maximum chunk size */
#define SU_MIN_CHUNK_SIZE  DEV_BSIZE  /* minimum chunk size */
#define SU_MAX_TIME_RATE   10   /* maximum time rate */
#define SU_MAX_CDB_LEN     CDB_GROUP5 /* up to Group 5 commands
*/

/*
 * structure to be used SU_SET_CAP/SU_GET_CAP
 * ioctl requests
 */
struct    su_dev_cap {
     int  cap; /* capability defined in scsi/impl/services.h */
     int  value;    /* value of capability */
};

/*
 * structure containing resulting data from last SCSI command
 * to be used with SU_GET_RESULTS ioctl request
 * possible values for fields are defined in scsi/scsi_pkt.h
 */
struct    su_results {
     long resid;    /* data bytes not transferred */
     u_char    reason;   /* command completion reason */
     u_char    state;    /* state of command reached */
     u_char    statistics;    /* some statistics */
};
```

```
/* SCSI bus id and target device id data */
struct     su_scsi_id {
      u_char     bus_id     : 4,
      dev_id     : 4;
};

/*
 * implementation specific additional flag for uscsi_flags
 * in uscsi_cmd structure defined in scsi/impl/uscsi.h
 *
 * MUST be set when issuing block transfer SCSI commands
 *
 * if set, this flag enforces su driver to interpret address and
count
  * of data as logical block address and number of blocks,
respectively,
 * and to use appropriate algorithm
 *
 * uscsi_buflen length MUST be in multiplies of device's logical
 * block size, which is set by SU_SET_BLOCK_SIZE ioctl request
 *
 * transfer chunks size, set by SU_SET_CHUNK_SIZE ioctl request,
 * MUST be in multiplies of device's logical block size,
 * which is set by SU_SET_BLOCK_SIZE ioctl request
 */
#define USCSI_BLOCK  0x10 /* block transfer SCSI command */
```

```
/*
 * implementation specific additional flag for uscsi_flags
 * in uscsi_cmd structure defined in scsi/impl/uscsi.h
 *
 * if set, this flag enforces su driver to transfer data without
 * dividing it to smaller chunks and may affect overall system
performance
 * when used for large data, however may boost the su transfer
performance
 *
 * MUST be set when issuing some control and vendor specific
commands, i.e.
 * commands not following exactly the generic command format (Gr
0, 1 and 5),
 * to SCSI device, because these commands usually don't contain
 * large data and/or data should not be divided to chunks and/or
 * address and count fields do not correspond to the generic
command format
 * failure to do so may cause unexpected results
 */
#define USCSI_CONT    0x20 /* don't divide data into chunks */

/*
 * implementation specific additional flag for uscsi_flags
 * in uscsi_cmd structure defined in scsi/impl/uscsi.h
 *
```

```
 * if set, this flag enforces su driver not to retry
unsuccessfull
 * commands, and may be used with commands with relative
addressing, etc.,
 * since device can perform unexpected operations
 */
#define USCSI_NO_RETRY     0x40 /* don't retry unsuccessfull cmd
*/


/*
 * Definitions for ioctl requests, except USCSICMD
 */


/* reset SCSI device */
#define    SU_RESET_DEV    _IO(u, 0)


/*
 * defined in scsi/impl/uscsi.h
 * supports only Group 0, Group 1 and Group 5 SCSI commands
 *
 * if USCSI_BLOCK flag is set, uscsi_buflen length MUST be in
 * multiplies of device's logical block size, which is set by
 * SU_SET_BLOCK_SIZE ioctl request
 */
/* USCSICMD */


/* get inquiry data, defined in scsi/generic/inquiry.h */
```

```
#define    SU_GET_INQ_DATA _IORN(u, 2, SUN_INQSIZE)

/*   send   REQUEST   SENSE   and   get   sense,   defined   in
scsi/generic/sense.h */
#define    SU_REQUEST_SENSE_IORN(u, 3, SU_SENSE_LENGTH)

/* get HA capabilities for device */
#define    SU_GET_CAP _IOWR(u, 4, struct su_dev_cap)

/* set HA capabilities for device */
#define    SU_SET_CAP _IOW(u, 5, struct su_dev_cap)

/*   get   results   from   last   SCSI   transfer,   defined   in
scsi/scsi_pkt.h */
#define SU_GET_RESULTS     _IOR(u, 6, struct su_results)

/*
 * set size of internal transfer chunk
 * if 0, driver will use system default size
 */
#define SU_SET_CHUNK_SIZE  _IOW(u, 7, long)

/* set number of retries for a failed SCSI cmd */
#define SU_SET_RETRY_NUM   _IOW(u, 8, int)

/* get SCSI bus id and target device id */
#define    SU_GET_SCSI_ID  _IOR(u, 9, struct su_scsi_id)
```

```
/*
 * set completion time rate for SCSI command transport in
sec/512bytes,
 * if 0, driver will use 1 sec/512bytes default rate
 */
#define SU_SET_TIME_RATE   _IOW(u, 10, int)

/*
 * set optional flags for performing SCSI commands
 * flags are defined in scsi/scsi_pkt.h
 * driver will use no flags as default
 */
#define SU_SET_PKT_FLAGS   _IOW(u, 11, long)

/*
 * set device logical block size for SCSI commands using block
 * transfer, default value is DEV_BSIZE = 512 for most of
systems
 */
#define SU_SET_BLOCK_SIZE  _IOW(u, 12, long)

#endif    /* _scsi_targets_sudef_h */
```

```
/*
 * SU - SCSA compatible universal SCSI driver
 * su.c include file
 * sudrv.h 12/11/95
 * Artak O. Melkonian, All Rights Reserved, 1994, 1995
 * Department of Electrical and Computer Engineering
 * New Jersey Institute of Technology
 */

/*
 * Debugging macro. Messages go on console
 */
#define DEBUG
#ifdef DEBUG
int su_deb = 0;
#define DPRINT(v, fmt, p0, p1, p2) \
     if (su_deb > (v)) printf(fmt, p0, p1, p2); \
     else {}
#else
#define DPRINT(v, fmt, p0, p1, p2)     ;
#endif
/**********************************************/

#define SU_TIMEOUT    10    /* min time for pkt */
#define SU_COMMAND_RETRY   1     /* send SCSI command again */
#define SU_COMMAND_ERROR   2     /* unrecoverable cmd error */
#define SU_COMMAND_DONE    3     /* command was successfull */
```

```
/*
 * Definitions for unit_flags field in struct su_device
 */
#define    UNIT_OPEN  0x01  /* unit is open */
#define    TAGGED_QING     0x02 /* tagged queueing enabled */
#define    WRITE_EN   0x04  /* write enabled */
#define SILENT  0x08 /* no error messages on console */
#define BLOCK_CMD    0x10  /* block transfer SCSI command */
#define CONT_TRAN    0x20  /* don't divide data into chunks */
#define NO_RETRY0x40  /* don't retry SCSI command again */

/*
 * private data for each unit
 * pointer to data of this type will be stored in sd_private
 * field of scsi_device structure for each individual device
 */
struct su_device  {

     /* various operational flags */
     u_char     unit_flags;
     /* ready pkt for REQUEST SENSE command */
     struct scsi_pkt *unit_reqsense_pkt;
     /* pkt flags to be set during transfer */
     long unit_pkt_flags; /* 0 as default */
     /* uscsi status */
     unsigned char   unit_uscsi_status;
```

```
    /*
     * buf pointer for current transfer, note:
     * b_forw points to unit's scsi_device structure
     * b_back points to unit's su_device structure
     */
    struct buf *unit_bp;
    /* current scdb and its length*/
    union scsi_cdb  *unit_scdb;
    int  unit_scdb_len;
    /* results of last pkt transportation */
    struct su_results    *unit_results;
    /* size for su_minphys, if 0, use kernel minphys function
*/
    long unit_chunk_size;
    /* number of retries and retry counter */
    int  unit_num_retries;
    int  unit_retry_count;
    /* comletion time rate in sec/512bytes */
    int  unit_time_rate;
    /* device block size, default is DEV_BSIZE */
    long unit_block_size;
    /* offset, to be added to starting address of SCSI cmd */
    long unit_addr_offset;
};

/*
 * Driver messages printed on console
```

```
 */
char *su_drv_message[] = { /*#*/
      "Error: maximum supported unit number is", /*0*/
      "Cannot allocate memory.\n",    /*1*/
      "Kernel memory error ...\n",    /*2*/
      "Device exists, cannot identify.\n", /*3*/
      "Universal   SCSI   driver.   (C)1995   Artak   Melkonian.
Release:", /*4*/
      "Removable",    /*5*/
      "SCSI-",   /*6*/
      "Device",  /*7*/
      "Vendor",  /*8*/
      "Product", /*9*/
      "DMA Error: data exceeded maximum DMA size.\n", /*A*/
      "SCSI error occured: retrying command ...\n",   /*B*/
      "Transport  error  occured  when  retrying  command,  giving
up.\n",    /*C*/
      "SCSI command completion error, giving up.\n",  /*D*/
};
```

```
/*
 * SU - SCSA compatible universal SCSI driver
 * Driver module: dev_info style,
 * for SunOS 4.1.3 sun4c architecture
 * su.c    12/11/95
 * Artak O. Melkonian, All Rights Reserved, 1994, 1995
 * Department of Electrical and Computer Engineering
 * New Jersey Institute of Technology
 */

#include "sudef.h"
#include "sudrv.h"



/* pointers for scsi_device structures of supported devices */
static struct scsi_device  *su_units[SU_MAXUNIT_NUM];

int su_identify(), su_attach(), su_open(), su_close();
su_strategy(), su_ioctl();
void su_comp();

extern int nulldev();
extern int nodev();

/*
 * dev_ops structure for su driver: driver entry points
 */
```

```
#ifdef OPENPROMS
struct dev_ops su_ops = {
      1,
      su_identify,
      su_attach,
      su_open,
      su_close,
      nulldev,   /* no read() entry point */
      nulldev,   /* no write() entry point */
      nodev,     /* no strategy() entry point */
      nodev,     /* no dump() entry point */
      nulldev,   /* no size() entry point */
      su_ioctl   /* controls SCSI commands, data transfer and
internals */
};

#else
NO OPENPROMS;    /* no SCSA support */
#endif

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_identify function:
 * called by kernel during boot
 * checks if corresponding device exists and responds, and if
so,
 * initializes certain fields of scsi_device,
```

```
 * creates pkt for REQUEST SENSE command to be used later
 */

int
su_identify(devp)

/* scsi_device structure partially filled by HA */
struct scsi_device    *devp;
{
     int  unit_num = devp->sd_dev->devi_unit;   /* get minor
number */
     register struct su_device  *unit_su_device;
     struct scsi_pkt *rs_pkt;

     /* test if minor number is in its range */
     if (unit_num >= SU_MAXUNIT_NUM) {
     printf("su%d: %s %d\n",
     unit_num, su_drv_message[0], SU_MAXUNIT_NUM - 1);
     return(0);
     }

     /* test the device and fill sd_inq structure accordingly
*/
     switch(scsi_slave(devp,0)) {

     /* memory or system failure */
     case SCSIPROBE_NOMEM:
```

```
case SCSIPROBE_FAILURE:
printf("su%d: %s", unit_num, su_drv_message[1]);
return(0);

/* no respont from specified target or no device */
case SCSIPROBE_NORESP:
return(0);

/* no identification data */
case SCSIPROBE_NONCCS:
printf("su%d: %s", unit_num, su_drv_message[3]);
return(0);
default:
return(0);

/* device exists and responds */
case SCSIPROBE_EXISTS:
{}    /* go ahead */
}

/* allocate memory for su_device structure */
unit_su_device =
(struct su_device *)kmem_zalloc(sizeof(struct su_device));
if (!unit_su_device) {     /* if could not, panic ! */
printf("su%d: (01) %s", unit_num, su_drv_message[1]);
panic(su_drv_message[2]);  /* REBOOT ! */
}
```

```
	/* initializing unit_flags field in su_device */
	unit_su_device->unit_flags = 0;

	/* allocate memory for unit_bp in unit_su_device and set
some fields */
	unit_su_device->unit_bp = (struct buf
*)kmem_zalloc(sizeof(struct buf));
	if (!(unit_su_device->unit_bp)) {	/* if could not,
panic ! */
	printf("su%d: (02) %s", unit_num, su_drv_message[1]);
	panic(su_drv_message[2]);  /* REBOOT ! */
	}

	/* allocate memory for unit_results in unit_su_device */
	unit_su_device->unit_results =
	(struct su_results *)kmem_zalloc(sizeof(struct
su_results));
	if (!(unit_su_device->unit_results)) {	/* if could not,
panic ! */
	printf("su%d: (03) %s", unit_num, su_drv_message[1]);
	panic(su_drv_message[2]);  /* REBOOT ! */
	}

	/* keep important pointers in unit_bp */
	unit_su_device->unit_bp->b_forw = (struct buf *)devp;
```

```
unit_su_device->unit_bp->b_back = (struct buf
*)(unit_su_device);

/* make su_device private structure pointed by sd_private
*/
(struct su_device *)(devp->sd_private) = unit_su_device;

/* keep this devp in corresponding su_units element */
su_units[unit_num] = devp;

/*
 *Allocate pkt for unit_su_device->unit_reqsense_pkt,
 *we will need it later for REQUEST SENSE command,
 *will return extended sense to sd_sense of the unit,
 *which is allocated of length SU_SENSE_LENGTH
 */
rs_pkt = get_pktiopb(&(devp->sd_address),
(caddr_t *)&(devp->sd_sense), CDB_GROUP0,
SU_STATUS_LEN, SU_SENSE_LENGTH, B_READ, NULL_FUNC);
if (!rs_pkt) {  /* if could not, panic ! */
printf("su%d: (04) %s", unit_num, su_drv_message[1]);
panic(su_drv_message[2]);  /* REBOOT ! */
}

/* keep its pointer to its proper place */
unit_su_device->unit_reqsense_pkt = rs_pkt;
rs_pkt->pkt_pmon = -1;     /* no performance monitoring */
```

```
      rs_pkt->pkt_comp = su_comp;     /* install completion
routine*/
      rs_pkt->pkt_time = 0;
      rs_pkt->pkt_private = (opaque_t)(unit_su_device->unit_bp);

      /*
       * put REQUEST SENSE command description block
       * and flags into rs_pkt
       */
      makecom_g0(rs_pkt, devp, FLAG_NODISCON,
SCMD_REQUEST_SENSE,
      0, SU_SENSE_LENGTH);

      devp->sd_dev->devi_driver = &su_ops;  /* install driver
entry points */
      devp->sd_present = 1; /* mark present */

      return(1); /* return OK */
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_strspcpy function:
 * copies a string of length n from source_str to dest_str
 * until the first space character
 * called from su_attach
 */
```

```
static
su_strspcpy(dest_str, source_str, n)
register char   *dest_str;
register char   *source_str;
int   n;
{
      register int    i;

      for(i = 0; i < n; i++) {
      *(dest_str + i) = *(source_str + i);
      if (*(dest_str + i) == ' ') {
      *(dest_str + i) = '\0';
      return;
      }
      }
      *(dest_str + n) = '\0';
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_attach function:
 * prints inquiry information,
 * tests and tries to enable tagged queueing capability
 */

int
su_attach(devp)
struct scsi_device   *devp;
```

```
{
        struct scsi_inquiry   *inq_data;
        int   unit_num = devp->sd_dev->devi_unit;
        char str[17];   /* string for inquiry info */

        printf("\t%s %s\n", su_drv_message[4], SU_DRIVER_VER);

        /*
         * printing device identification info from inquiry
         */
        inq_data = devp->sd_inq;
        ASSERT(inq_data != NULL);
        printf("\tsu%d: ", unit_num);
        if (inq_data->inq_rmb)      /* Removable ? */
        printf("%s ", su_drv_message[5]);
        printf("%s %s%d %s, ", scsi_dname(inq_data->inq_dtype),
        su_drv_message[6], inq_data->inq_ansi, su_drv_message[7]);
        su_strspcpy(str, inq_data->inq_vid, 8);
        printf("%s: \'%s\', ", su_drv_message[8], str);
        su_strspcpy(str, inq_data->inq_pid, 16);
        printf("%s: \'%s\'.\n", su_drv_message[9], str);

        /*
         * try to enable tagged queueing for the unit,
         * set flag accordingly in unit_flags
         */
        if (scsi_ifsetcap(&(devp->sd_address),
```

```
      scsi_capstrings[SCSI_CAP_TAGGED_QING], 1, 1) == 1)
      ((struct su_device *)devp->sd_private)->
      unit_flags |= TAGGED_QING;
      else
      ((struct su_device *)devp->sd_private)->
      unit_flags &= ~TAGGED_QING;


}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_done function:
 * finish up current transfer
 * destroy pkt, if any and wake up sleeping user process
 */

su_done(bp)
struct buf *bp;
{
      ASSERT(bp != NULL);
      DPRINT(0, "su%d debug: su_done called\n", minor(bp-
>b_dev), 0, 0);

      /* destroy pkt, if there is any and it isn't reqsense_pkt
*/
      if (bp->av_forw)
      scsi_pktfree((struct scsi_pkt *)(bp->av_forw));
```

```
        /* reset unit_retry_count */
        ((struct su_device *)(bp->b_back))->unit_retry_count = 0;

        /* wake up user process and set B_DONE */
        iodone(bp);
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_strategy function:
 * called from su_uscsi() and prepares and transports
 * SCSI pkt and and does data transfer, if there is need
 * notice: this is not a driver entry point
 */

su_strategy(bp)
register struct buf   *bp;
{
        struct scsi_device    *devp = (struct scsi_device *)(bp-
>b_forw);
        register struct su_device  *unit_su_device =
        (struct su_device *)(bp->b_back);
        struct scsi_pkt *pkt;
        int   unit_num = minor(bp->b_dev);
        int   transport_state;/* state of transport */
        union scsi_cdb  *cdb = unit_su_device->unit_scdb;
        union scsi_cdb  *cdb_in_pkt;
```

```
/*
 * number of logical blocks to be transfered when using
block algorithm
 * used as count parameter in outgoing SCSI command
 */
long blk_count; /* number of blocks */
/* just to be short */
long *offset =
&(unit_su_device->unit_addr_offset);

DPRINT(0, "su%d debug: su_strategy called\n", unit_num, 0,
0);

bp->b_flags &= ~(B_DONE | B_ERROR);   /* initialize flags
*/
bp->b_resid = 0;/* this will be ok hopefully */

/* allocate pkt with or witout DMA support depending on
b_bcount */
if (bp->b_bcount)
/* allocate pkt with associated DMA token */
pkt = scsi_resalloc(&(devp->sd_address),
unit_su_device->unit_scdb_len, SU_STATUS_LEN,
(opaque_t)bp, NULL_FUNC);
else
/* allocate pkt without any DMA support */
pkt = scsi_pktalloc(&(devp->sd_address),
```

```
unit_su_device->unit_scdb_len, SU_STATUS_LEN, NULL_FUNC);

/* save pkt pointer in bp field */
(struct scsi_pkt *)(bp->av_forw) = pkt;

/* check if pkt has been allocated */
if (!pkt) {
if (!(unit_su_device->unit_flags & SILENT))
printf("su%d: %s", unit_num, su_drv_message[1]);
bp->b_flags |= B_ERROR;
bp->b_error = ENOMEM;
bp->b_resid = bp->b_bcount;
su_done(bp);    /* finish, if could not allocate pkt */
return;
}

/* save pointer of cdb in pkt */
cdb_in_pkt = (union scsi_cdb *)(pkt->pkt_cdbp);

/* make common part of SCSI command for pkt and fill some
pkt fields */
MAKECOM_COMMON(pkt, devp, unit_su_device->unit_pkt_flags,
cdb->scc_cmd);
bcopy(cdb, cdb_in_pkt, unit_su_device->unit_scdb_len);

/*
```

```
        * if there is data transfer AND it may be done chunk by
chunk,
        * fill standart address and count fields cleverly
        */
       if ((bp->b_bcount != 0) && !(unit_su_device->unit_flags &
CONT_TRAN)) {
       /*
        * make specific address and count parameters according
             * to BLOCK_CMD flag and unit_addr_offset, using
algorithm
             * to support access by chunks, if necessary
             */
       if (unit_su_device->unit_flags & BLOCK_CMD) {
       /* use block algorithm */
       blk_count = bp->b_bcount /
       unit_su_device->unit_block_size;
       switch GETGROUP(cdb) {
           case CDB_GROUPID_0:    /* Group 0 cmd */
       /* if this is sequential device */
       if ((devp->sd_inq->inq_dtype &
       DTYPE_MASK) == DTYPE_SEQUENTIAL) {
       if (!(cdb->t_code & 0x01)) {
       bp->b_flags |= B_ERROR;
       bp->b_error = EINVAL;
       bp->b_resid = bp->b_bcount;
       su_done(bp);
       return;    /* no fixed bit in cmd */
```

```
}
FORMG0COUNT_S(cdb_in_pkt, blk_count);
} else {   /* other device */
FORMG0ADDR(cdb_in_pkt,
GETG0ADDR(cdb) + *offset);
FORMG0COUNT(cdb_in_pkt, blk_count);
}
break;

    case CDB_GROUPID_1:
/* check if relative addressing */
if (!(cdb->g1_reladdr & 0x01)) {/* no */
FORMG1ADDR(cdb_in_pkt,
GETG1ADDR(cdb) + *offset);
} else     /* yes */
if (*offset)
FORMG1ADDR(cdb_in_pkt, 0);
FORMG1COUNT(cdb_in_pkt, blk_count);
break;

    case CDB_GROUPID_5:
/* check if relative addressing */
if (!(cdb->scc5_reladdr & 0x01)) {    /* no */
FORMG5ADDR(cdb_in_pkt,
GETG5ADDR(cdb) + *offset);
} else     /* yes */
if (*offset)
```

```
FORMG5ADDR(cdb_in_pkt, 0);
FORMG5COUNT(cdb_in_pkt, blk_count);
break;
}
DPRINT(0, "b_bcount %d, blk_count %d, offset %d\n",
bp->b_bcount, blk_count, *offset);
*offset += blk_count;
} else {   /* use byte algorithm */
/* Group 0 cmd and not a seq. device with set fixed bit*/
if ((GETGROUP(cdb) == CDB_GROUPID_0) &&
!(((devp->sd_inq->inq_dtype &
DTYPE_MASK) == DTYPE_SEQUENTIAL) &&
(cdb->t_code & 0x01))) {
FORMG0COUNT_S(cdb_in_pkt, bp->b_bcount);
} else {   /* all other situations */
bp->b_flags |= B_ERROR;
bp->b_error = EINVAL;
bp->b_resid = bp->b_bcount;
su_done(bp);
return;
}
}
}

/* fill out other pkt fields */
pkt->pkt_private = (opaque_t)bp;
pkt->pkt_comp = su_comp;   /* install completion routine */
```

```
pkt->pkt_time = SU_TIMEOUT +
(bp->b_bcount * unit_su_device->unit_time_rate) / 512;
pkt->pkt_pmon = -1;   /* no performance monitoring */

/* transport pkt */
transport_state = pkt_transport(pkt);

/* check state and perform appropriate action */
if (transport_state != TRAN_ACCEPT) {
if (transport_state == TRAN_BUSY)
bp->b_error = EBUSY;
else {
bp->b_error = EIO;
if (!(unit_su_device->unit_flags & SILENT))
printf("su%d: %s", unit_num, su_drv_message[10]);
}
bp->b_flags |= B_ERROR;
bp->b_resid = bp->b_bcount;
su_done(bp);
}
return;
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_comp function:
(interrupt)
```

```
 * called after performing a SCSI command as a completion
routine,
 * analyses resulting situation, performs appropriate actions,
 * retries command, if possible
 */

void
su_comp(pkt)
register struct scsi_pkt   *pkt;
{
     struct buf *bp = (struct buf *)(pkt->pkt_private);
     struct scsi_device   *devp = (struct scsi_device *)(bp-
>b_forw);
     register struct su_device  *unit_su_device =
     (struct su_device *)(bp->b_back);
     int  unit_num = minor(bp->b_dev);
     int  op;  /* what to do */

     DPRINT(0, "su%d debug: su_comp called\n", unit_num, 0, 0);

     /* analyze command completion */
     if ((pkt->pkt_reason != CMD_CMPLT) ||
     ((pkt->pkt_state & STATE_GOT_STATUS) == 0)) {
     /* too bad, cmd wasn't completed or no status received */
     if ((unit_su_device->unit_retry_count++ <
     unit_su_device->unit_num_retries) &&
     !(unit_su_device->unit_flags & NO_RETRY)) {
```

```
if (pkt == unit_su_device->unit_reqsense_pkt)
bzero((caddr_t)(devp->sd_sense),
SU_SENSE_LENGTH);
op = SU_COMMAND_RETRY;
} else {
op = SU_COMMAND_ERROR;
bp->b_error = EIO;
}
} else if (*(u_char *)(pkt->pkt_scbp) & STATUS_MASK) {
/* status ok ? */
op = SU_COMMAND_ERROR;
/* set uscsi status field */
     unit_su_device->unit_uscsi_status =
*(u_char *)(pkt->pkt_scbp);
/* no error message */
unit_su_device->unit_flags |= SILENT;
DPRINT(0, "su%d debug: status %x\n", unit_num,
*(u_char *)(pkt->pkt_scbp) & STATUS_MASK, 0);
} else {
if (pkt->pkt_resid) { /* if transfer isn't done */
op = SU_COMMAND_ERROR;
bp->b_resid += pkt->pkt_resid;
bp->b_error = EIO;
} else     /* SCSI command was successful */
op = SU_COMMAND_DONE;
}
```

```
/* perform operation according to op */
switch (op) {
case SU_COMMAND_RETRY:
if (pkt_transport(pkt) == TRAN_ACCEPT) {
if (!(unit_su_device->unit_flags & SILENT))
printf("su%d: %s", unit_num,
su_drv_message[11]);
break;
} else {
bp->b_error = EIO;
if (!(unit_su_device->unit_flags & SILENT))
printf("su%d: %s", unit_num,
su_drv_message[12]);
}
/* FALLTHROUGH */

case SU_COMMAND_ERROR:
bp->b_resid = bp->b_bcount;
bp->b_flags |= B_ERROR;
if (!(unit_su_device->unit_flags & SILENT))
printf("su%d: %s", unit_num, su_drv_message[13]);
/* FALLTHROUGH */

case SU_COMMAND_DONE:
/* copy result fields from pkt into unit_results */
bcopy((caddr_t)&(pkt->pkt_resid),
unit_su_device->unit_results,
```

```
      sizeof(struct su_results));

      /* finish it */
      su_done(bp);
      }
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_open function:
 * checks minor number and
 * sets UNIT_OP flag if everithing is ok
 * sets WRITE_EN flag according to user open() mode
 */

su_open(dev, flags)
dev_t dev;
int   flags;
{
      int   unit_num = minor(dev);
      struct     scsi_device     *devp;
      register struct su_device  *unit_su_device;
      int   pri;

      DPRINT(0, "su%d debug: su_open called\n", unit_num, 0, 0);

      /* test if minor number is in its range */
      if (unit_num >= SU_MAXUNIT_NUM)
```

```
return(ENXIO);  /* return error if not */

/* test if there is present device with this minor
number*/
if ((devp = su_units[unit_num]) == (struct scsi_device
*)0)
return(ENXIO);  /* return error if not */

ASSERT(devp->sd_private != NULL);
unit_su_device = (struct su_device *)(devp->sd_private);

pri = spl6();   /* begin critical section */
if (unit_su_device->unit_flags & UNIT_OPEN)     /* if
already open */
return(EBUSY);  /* return busy */
unit_su_device->unit_flags |= UNIT_OPEN;   /* set UNIT_OPEN
flag */
splx(pri); /* exit critical section */

/* test if write is enabled and set the flag accordingly
*/
if (flags & FWRITE) {
unit_su_device->unit_flags |= WRITE_EN;
DPRINT(0, "su%d debug: write enabled\n", unit_num, 0, 0);
} else
unit_su_device->unit_flags &= ~WRITE_EN;
```

```
    /* initialize some fields */
    unit_su_device->unit_uscsi_status = 0;
    unit_su_device->unit_scdb = NULL;
    unit_su_device->unit_scdb_len = 0;
    ASSERT(unit_su_device->unit_results != NULL);
    bzero((caddr_t)(unit_su_device->unit_results),  /*
initialize */
    sizeof(struct su_results));
    unit_su_device->unit_chunk_size = 0;  /* default */
    unit_su_device->unit_num_retries = SU_NUM_RETRIES;    /*
default */
    unit_su_device->unit_retry_count = 0;
    unit_su_device->unit_time_rate = 1;   /* default */
    unit_su_device->unit_pkt_flags = 0;   /* default */
    unit_su_device->unit_block_size = DEV_BSIZE;     /* default
*/
    unit_su_device->unit_addr_offset = 0; /* initialize */

    ASSERT(devp->sd_sense != NULL);
    bzero((caddr_t)(devp->sd_sense), SU_SENSE_LENGTH);    /*
initialize */

    return(0);
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_close function:
```

```
 * clear UNIT_OPEN flag
 */

su_close(dev, flags)
dev_t dev;
int   flags;
{
      int   unit_num = minor(dev);
      struct     scsi_device     *devp;

      DPRINT(0, "su%d debug: su_close called\n", unit_num, 0,
0);

      if ((devp = su_units[unit_num]) == (struct scsi_device
*)0)
      return(ENXIO);

      /* clear UNIT_OPEN flag */
      ASSERT(devp->sd_private != NULL);
      ((struct su_device *)devp->sd_private)->
      unit_flags &= ~UNIT_OPEN;

      return(0);
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_minphys function:
```

```
 * determines size of transfer chunks and is called by physio.
 * if unit_chunk_size is 0 (default), return minphys value
supplied
 * by kernel, otherwise use value in unit_chunk_size, set by
 * calling ioctl with SU_SET_CHUNK_SIZE request.
 */

su_minphys(bp)
struct     buf   *bp;
{

      struct     su_device  *unit_su_device =
      (struct su_device *)(bp->b_back);
      long  su_chunk_size;

      DPRINT(0, "su%d debug: su_minphys called\n", minor(bp-
>b_dev), 0, 0);

      ASSERT(unit_su_device != NULL);
      su_chunk_size = unit_su_device->unit_chunk_size;
      if (!(su_chunk_size)) {    /* if zero */
      minphys(bp);    /* default function */
      return;
      }
      if (bp->b_bcount > su_chunk_size)     /* otherwise */
      bp->b_bcount = su_chunk_size;   /* limit size */
 }
```

```
/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_uscsi function:
 * performs uscsi request, called from su_ioctl
 * single threading is enforced by physio in DMA case
 * or iowait() using B_DONE mechanizm
 * which does not return until transfer is done
 * dev     - device major + minor numbers
 * su_uscsi_cmd - pointer to valid uscsi_cmd structure
 * wr - write enable flag
 */

static int
su_uscsi(dev, su_uscsi_cmd, wr, errorp)
dev_t dev;
register struct uscsi_cmd  *su_uscsi_cmd;
u_char     wr;
{
     struct scsi_device   *devp = su_units[minor(dev)];
     register struct su_device  *unit_su_device =
     (struct su_device *)(devp->sd_private);
     caddr_t    cdb;
     int   dir;  /* B_READ or B_WRITE */
     long orig_chunk_size =
     unit_su_device->unit_chunk_size;
     int   c_len;     /* length of cmd */
     int   error;
```

```
/* check length of cdb and validity of cdb pointer */
if ((!su_uscsi_cmd->uscsi_cdblen) || (!su_uscsi_cmd-
>uscsi_cdb) ||
(su_uscsi_cmd->uscsi_cdblen > SU_MAX_CDB_LEN))
return(EINVAL);

/* check for valid flags and parameters for block SCSI
transfer */
if (su_uscsi_cmd->uscsi_flags & USCSI_BLOCK) {
if (((su_uscsi_cmd->uscsi_buflen /
unit_su_device->unit_block_size) *
unit_su_device->unit_block_size) !=
su_uscsi_cmd->uscsi_buflen)
return(EINVAL);
if (((unit_su_device->unit_chunk_size /
unit_su_device->unit_block_size) *
unit_su_device->unit_block_size) !=
unit_su_device->unit_chunk_size)
return(EINVAL);
}

/* determine transfer direction */
dir = (su_uscsi_cmd->uscsi_flags & USCSI_READ) ? B_READ :
B_WRITE;

/* check for write permission */
```

```
if ((dir == B_WRITE) && (!wr))
return(EACCES); /* no write permission */

/* allocate memory for cdb */
cdb = kmem_zalloc((size_t)su_uscsi_cmd->uscsi_cdblen);
if (!cdb)
return(ENOMEM);

/* keep in su_device */
unit_su_device->unit_scdb = (union scsi_cdb *)cdb;

/* copy cdb in */
if (copyin(su_uscsi_cmd->uscsi_cdb, cdb, su_uscsi_cmd-
>uscsi_cdblen)) {
kmem_free(cdb, (size_t)su_uscsi_cmd->uscsi_cdblen);
return(EFAULT); /* if copyin fails */
}

/* check for validity of cmd group and its length */
switch (GETGROUP(unit_su_device->unit_scdb)) {
case CDB_GROUPID_0:
c_len = CDB_GROUP0;
break;
case CDB_GROUPID_1:
c_len = CDB_GROUP1;
break;
case CDB_GROUPID_5:
```

```
c_len = CDB_GROUP5;
break;
default:   /* not Group 0, 1, or 5 SCSI command */
ASSERT(cdb != NULL);
kmem_free(cdb, (size_t)su_uscsi_cmd->uscsi_cdblen);
return(EINVAL);
}
if (su_uscsi_cmd->uscsi_cdblen != c_len)
return(EINVAL);

/* copy length of cdb */
unit_su_device->unit_scdb_len = su_uscsi_cmd-
>uscsi_cdblen;

/* set SILENT flag according to USCSI_SILENT */
if (su_uscsi_cmd->uscsi_flags & USCSI_SILENT)
unit_su_device->unit_flags |= SILENT;

/* set BLOCK_CMD flag according to USCSI_BLOCK */
if (su_uscsi_cmd->uscsi_flags & USCSI_BLOCK)
unit_su_device->unit_flags |= BLOCK_CMD;

/*
 * if USCSI_CONT flag is set,
 * set unit_chunk_size to uscsi_buflen
 * and set CONT_TRAN flag
 */
```

```
if (su_uscsi_cmd->uscsi_flags & USCSI_CONT) {
unit_su_device->unit_chunk_size = su_uscsi_cmd-
>uscsi_buflen;
unit_su_device->unit_flags |= CONT_TRAN;
}

/* set NO_RETRY flag according to USCSI_NO_RETRY */
if (su_uscsi_cmd->uscsi_flags & USCSI_NO_RETRY)
unit_su_device->unit_flags |= NO_RETRY;

/* determine if DMA transfer needed */
if (su_uscsi_cmd->uscsi_buflen) {
/* DMA transfer needed */
struct iovec    i_iov;     /* internally allocated */
struct uio i_uio;     /* uio structure*/
struct uio *su_uio = &i_uio;

/* prepare su_uio structure */
bzero((caddr_t)&i_iov, sizeof(struct iovec));
bzero((caddr_t)&i_uio, sizeof(struct uio));
su_uio->uio_iov = &i_iov;
su_uio->uio_iov->iov_base = su_uscsi_cmd->uscsi_bufaddr;
su_uio->uio_iov->iov_len = su_uscsi_cmd->uscsi_buflen;
su_uio->uio_iovcnt = 1;
su_uio->uio_offset = 0;
su_uio->uio_segflg = UIO_USERSPACE;
su_uio->uio_fmode = 0;
```

```
su_uio->uio_resid = su_uscsi_cmd->uscsi_buflen;

/* perform physical I/O */
error = physio(su_strategy, unit_su_device->unit_bp, dev,
dir,
su_minphys, su_uio);
} else {   /* no data, no DMA */
/* prepare unit_bp */
unit_su_device->unit_bp->b_flags = dir;
unit_su_device->unit_bp->b_dev = dev;
unit_su_device->unit_bp->b_bcount = 0;
unit_su_device->unit_bp->b_blkno = 0;

/* call strategy routine for dataless SCSI cmd */
su_strategy(unit_su_device->unit_bp);

/* wait (sleep on user process) until B_DONE */
error = iowait(unit_su_device->unit_bp);
}

/* destroy current cdb */
ASSERT(cdb != NULL);
kmem_free(cdb, (size_t)su_uscsi_cmd->uscsi_cdblen);

/* reset some flags */
unit_su_device->unit_flags &=
~(SILENT | BLOCK_CMD | CONT_TRAN | NO_RETRY);
```

```
    /* reset SCSI address offset */
    unit_su_device->unit_addr_offset = 0;

    /* restore unit_chunk_size */
    unit_su_device->unit_chunk_size = orig_chunk_size;

    return(error);
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_request_sense function:
 * sends REQUEST SENSE command using already allocated
unit_reqsense_pkt
 * and copies received sense into buffer pointed by data
 */

static int
su_request_sense(dev, data)
dev_t dev;
caddr_t    data;
{
    struct scsi_device    *devp = su_units[minor(dev)];
    register struct su_device  *unit_su_device =
    (struct su_device *)(devp->sd_private);
    struct buf *bp = unit_su_device->unit_bp;
    int   transport_state;
```

```
int   error;

DPRINT(0, "su%d debug: su_request_sense called\n",
minor(dev), 0, 0);

bp->b_flags &= ~(B_DONE | B_ERROR);   /* initialize flags
*/
bp->av_forw = NULL;   /* su_done won't destroy
unit_reqsense_pkt */

/* transport unit_reqsense_pkt */
transport_state = pkt_transport(unit_su_device-
>unit_reqsense_pkt);

/* check state and perform appropriate action */
if (transport_state != TRAN_ACCEPT) {
if (transport_state == TRAN_BUSY)
bp->b_error = EBUSY;
else
bp->b_error = EIO;
bp->b_flags |= B_ERROR;
bp->b_resid = bp->b_bcount;
su_done(bp);
}

/* wait for completion */
error = iowait(bp);
```

```
        /* copy sense data and reset sd_sense */
        if (!error)
        bcopy((caddr_t)(devp->sd_sense), data, SU_SENSE_LENGTH);
        bzero((caddr_t)(devp->sd_sense), SU_SENSE_LENGTH);

        /* reset status, which may be changed in su_comp */
        unit_su_device->unit_uscsi_status = 0;

        return(error);
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> su_ioctl function:
 * performs ioctl requests, including user SCSI commands,
 * control operations etc.
 * valid ioctl requests are defined in /sys/scsi/targets/sudef.h
 */

su_ioctl(dev, cmd, data, flag)
dev_t dev;
register int    cmd;
caddr_t    data;
{
        int   unit_num = minor(dev);
        struct scsi_device    *devp;
        struct su_device*unit_su_device;
```

```
struct scsi_pkt *pkt;
u_char     wr_en;     /* write enabled flag */
int   error;

DPRINT(0, "su%d debug: su_ioctl with cmd = %d\n",
minor(dev), cmd, 0);

/* test if minor number is in its range */
if (unit_num >= SU_MAXUNIT_NUM)
return(ENXIO);  /* return error if not */

/* test if unit structures are ok and unit is opened*/
if ((devp = su_units[unit_num]) == (struct scsi_device
*)0)
return(ENXIO);  /* return error if not */
ASSERT(devp->sd_private != NULL);
unit_su_device = (struct su_device *)(devp->sd_private);
if (!(unit_su_device->unit_flags & UNIT_OPEN))
return(ENODEV);

/* set write enabled flag according to open mode */
wr_en = unit_su_device->unit_flags & WRITE_EN;

/* perform ioctl request */
switch (cmd) {
case SU_RESET_DEV:   /* reset the device */
return((scsi_reset(&(devp->sd_address), RESET_TARGET)) ?
```

```
0 : EIO);

case USCSICMD:  /* issue user SCSI command */
if (data == NULL)
return(EFAULT);

/* send USCSI command */
error = su_uscsi(dev, data, wr_en);

/* copy SCSI status and reset it in su_device*/
((struct uscsi_cmd *)data)->uscsi_status =
unit_su_device->unit_uscsi_status;
unit_su_device->unit_uscsi_status = 0;

/*
 * if status is not zero, return 0 error code
 * to copy uscsi_cmd structure back
 */
return((((struct uscsi_cmd *)data)->uscsi_status &
STATUS_MASK) ? 0 : error);

case SU_GET_INQ_DATA: /* return stored inquiry data */
if (data == NULL)
return(EFAULT);
bcopy((caddr_t)(devp->sd_inq), data,
(SU_GET_INQ_DATA >> 16) & 0x00ff);   /* size */
return(0);
```

```
case SU_REQUEST_SENSE:    /* send REQUEST SENSE and return
sense */
if (data == NULL)
return(EFAULT);
return(su_request_sense(dev, data));

case SU_GET_CAP:/* get HA capabilities for unit */
if (data == NULL)
return(EFAULT);
if ((((struct su_dev_cap *)data)->value =
scsi_ifgetcap(&(devp->sd_address),
scsi_capstrings[((struct su_dev_cap *)data)->
cap], 1)) != -1)
return(0);
else
return(EINVAL);

case SU_SET_CAP:/* set HA capabilities for unit */
if (data == NULL)
return(EFAULT);
if (scsi_ifsetcap(&(devp->sd_address),
scsi_capstrings[((struct su_dev_cap *)data)->
cap], ((struct su_dev_cap *)data)->value, 1)
== 1)
return(0);
else
```

```
return(EINVAL);

case SU_GET_RESULTS: /* get results from last SCSI
transfer*/
if (data == NULL)
return(EFAULT);
bcopy((caddr_t)(unit_su_device->unit_results), data,
sizeof(struct su_results));
return(0);

case SU_SET_CHUNK_SIZE:    /* set chunk size for su_minphys
*/
if (data == NULL)
return(EFAULT);
if ((*(long *)data > SU_MAX_CHUNK_SIZE) ||
(*(long *)data < SU_MIN_CHUNK_SIZE))
return(EINVAL);
unit_su_device->unit_chunk_size = *(long *)data;
return(0);

case SU_SET_RETRY_NUM:     /* set max num of retries */
if (data == NULL)
return(EFAULT);
if ((*(int *)data > SU_MAX_NUM_RETRIES) ||
(*(int *)data < 0))
return(EINVAL);
unit_su_device->unit_num_retries = *(int *)data;
```

```
return(0);

case SU_GET_SCSI_ID: /* return bus and target ids */
if (data == NULL)
return(EFAULT);
((struct su_scsi_id *)data)->bus_id =
devp->sd_dev->devi_parent->devi_unit;
((struct su_scsi_id *)data)->dev_id =
devp->sd_address.a_target;
return(0);

case SU_SET_TIME_RATE:     /* set time rate in sec/512bytes
*/
if (data == NULL)
return(EFAULT);
if ((*(int *)data > SU_MAX_TIME_RATE) ||
(*(int *)data < 0))
return(EINVAL);
unit_su_device->unit_time_rate = *(int *)data;
return(0);

case SU_SET_PKT_FLAGS:     /* set pkt flags */
if (data == NULL)
return(EFAULT);
unit_su_device->unit_pkt_flags = *(long *)data;
return(0);
```

```
case SU_SET_BLOCK_SIZE:
if (data == NULL)
return(EFAULT);
if (*(long *)data <= 0)
return(EINVAL);
unit_su_device->unit_block_size = *(long *)data;
return(0);


default:
return(ENOTTY);
}
}
```

```
#! /bin/sh
#
# MAKEDEV.su v. 1.00  12/11/95
#
# MAKEDEV for su - SCSI universal driver
#
# synopsis:
#     MAKEDEV.su char
#
#     where char is the respective char major
#     number of the driver in a particular system.
#

char=$1;
umask 0;
max_unit_num=4;
unit_num=0;
mode=666;

while [ $unit_num -ne $max_unit_num ]
do
      /etc/mknod rsu$unit_num c $char $unit_num
      chmod $mode rsu$unit_num
      unit_num=`expr $unit_num + 1`;
done

   umask 77
```

APPENDIX B

C SOURCE OF THE "HSCSID" HOST LEVEL SCSI REQUEST SERVER DAEMON

```
/*
 * Host side deamon running on Sun SPARC station performing
 * as a link
 * between Hydra DSP programs and su universal SCSI driver
 * hscsid.c     12/11/95
 * usage: hscsid dsp trapnum
 * Artak O. Melkonian, All Rights Reserved, 1994, 1995
 * Department of Electrical and Computer Engineering
 * New Jersey Institute of Technology
 */

#include <scsi/targets/sudef.h>
#include <vc40dsp.h>
#include <stdio.h>
#include <sys/types.h>
#include <syslog.h>
#include "shared.h"


#define SUDEVFILE     "/dev/rsu "
#define HYDRA INT     SIGUSR1
```

```
#define BAD_DEV REQ_NUM
#define OPEN_ERRREQ_NUM + 1

int   su_d[SU_MAXUNIT_NUM];
char  su_dev_name[sizeof(SUDEVFILE)] = SUDEVFILE;
int   c40_d, i;  /* c40 device descriptor */
struct scsi_inquiry  inq; /* SCSI inquiry info */
long  err;
caddr_t    shmem_ptr; /* hydra shared mem ptr */
struct hydra_request  *hy_req;   /* parameter block ptr */
char  *data;
int   trapnum;

/* signal handlers */
void hydra_int_handler();
void term_handler();


/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> main() function
 */

main(argc, argv)
int   argc;
char  *argv[];
{
      int   i;
```

```
struct vc40info hydra_info;

/* check command line */
if (argc != 3) {
printf("usage:  hscsid dsp trapnum\n");
printf("   dsp - special file for dsp\n");
printf("   trapnum - dsp trap number to use\n");
exit(1);
}
if ((trapnum = (int)strtol(argv[2], NULL, 0)) <= 0) {
printf("hscsid: trapnum out of range.\n");
exit(1);
}

/* fork and disassociate controlling terminal */
err = fork();
if (err > 0)
exit(0);
if (err == -1) {
printf("hscsid: couldn't fork");
exit(1);
}
setsid();

/* initialize descriptors */
for (i = 0; i < SU_MAXUNIT_NUM; i++)
su_d[i] = -1;
```

```
/* setup syslog */
openlog(argv[0], LOG_PID, LOG_DAEMON);

/* open Hydra board */
if ((c40_d = open(argv[1], O_RDWR)) <= 0) {
syslog(LOG_ERR, "failed to open Hydra DSP %s. Exiting.",
argv[1]);
exit(1);
}

/* get DSP configuration information */
if (c40_getinfo(c40_d, &hydra_info) != 0) {
syslog(LOG_ERR, "failed to get DSP info. Exiting.");
exit(1);
}

/* allocate space for data buffer */
data = (char *)malloc(hydra_info.dram_size -
MO_BYTES_PER_DSP *
hydra_info.numdsp - HY_REQ_SIZE);
if (!data) {
syslog(LOG_ERR, "failed to allocate memory for data
buffer.");
syslog(LOG_ERR, "requested size = %d",
hydra_info.dram_size - MO_BYTES_PER_DSP *
hydra_info.numdsp - HY_REQ_SIZE);
```

```
syslog(LOG_ERR, "Exiting.");
exit(1);
}

/* install interrupt handlers */
signal(HYDRA_INT, hydra_int_handler);
signal(SIGTERM, term_handler);
signal(SIGINT, term_handler);
signal(SIGHUP, term_handler);
signal(SIGQUIT, term_handler);

/*
 * map in Hydra shared memory excluding top memory
 * portions used by Hydra monitor and parameter block
 * note that on failure, c40_map_shmem returns -1 !
 */
if ((shmem_ptr = (caddr_t)c40_map_shmem(c40_d, 0,
hydra_info.dram_size - MO_BYTES_PER_DSP *
hydra_info.numdsp -
HY_REQ_SIZE)) == (caddr_t)-1) {
syslog(LOG_ERR, "failed to map Hydra shared memory.
Exiting.");
exit(1);
} else
syslog(LOG_INFO, "Shmem address: 0x%lx.", shmem_ptr);

/*
```

```
 * map in Hydra shared memory portion for parameter block
 * just before space used by Hydra monitor
 * note that on failure, c40_map_shmem returns -1 !
 */
if ((hy_req = (struct hydra_request *)c40_map_shmem(c40_d,
hydra_info.dram_size - MO_BYTES_PER_DSP *
hydra_info.numdsp -
HY_REQ_SIZE, HY_REQ_SIZE)) == (struct hydra_request *)-1)
{
syslog(LOG_ERR, "failed to map Hydra shared memory.
Exiting.");
exit(1);
} else
syslog(LOG_INFO, "Reqmem address: 0x%lx.", hy_req);

/* sleep, wait for signals */
while(1) {
pause();
}
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> send_scsi_cmd() function
 * sends SCSI command and transfers data, if any
 */
int
send_scsi_cmd(d_num)
```

```
int   d_num;
{
      int   error;
      struct uscsi_cmdcmd;
      struct su_results     res;

      /* prepare uscsi cmd */
      cmd.uscsi_cdb = (caddr_t)&(hy_req->params[0]);
      cmd.uscsi_cdblen = hy_req->params[3];
      cmd.uscsi_buflen = hy_req->params[5] * WORD_SIZE;
      cmd.uscsi_bufaddr = cmd.uscsi_buflen ? data : NULL;
      cmd.uscsi_flags = hy_req->params[7];

      /* copy data from shmem, if write operation */
      if (!(cmd.uscsi_flags & USCSI_READ))
      memcpy(data, ((caddr_t)((unsigned long)shmem_ptr +
      hy_req->params[4] * WORD_SIZE)), cmd.uscsi_buflen);

      /* send command */
      error = ioctl(su_d[d_num], USCSICMD, &cmd);
      hy_req->params[6] = cmd.uscsi_status;

      /* copy data to shmem, if read operation */
      if (cmd.uscsi_flags & USCSI_READ)
      memcpy(((caddr_t)((unsigned long)shmem_ptr +
      hy_req->params[4] * WORD_SIZE)), data, cmd.uscsi_buflen);
```

```
        return(error);
}


/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> write_inq_data() function
 * writes appropriate inquiry data to Hydra
 */
int
write_inq_data(d_num)
int   d_num;
{
        int   i;


        /* get inquiry info */
        if (err = ioctl(su_d[d_num], SU_GET_INQ_DATA, &inq) == -1)
{
        perror("host_scsi: failed to get SCSI inquiry info");
        return(err);
        }


        /* copy appropriate inquiry fields */
        hy_req->params[0] = inq.inq_dtype;
        hy_req->params[1] = inq.inq_rmb;
        hy_req->params[2] = inq.inq_qual;
        hy_req->params[3] = inq.inq_ansi;
        hy_req->params[4] = inq.inq_aenc;
        hy_req->params[5] = inq.inq_rdf;
```

```
    hy_req->params[6] = inq.inq_reladdr;
    hy_req->params[7] = inq.inq_wbus32;
    hy_req->params[8] = inq.inq_wbus16;
    hy_req->params[9] = inq.inq_linked;
    hy_req->params[10] = inq.inq_cmdque;
    hy_req->params[11] = inq.inq_sftre;

    for(i = 0; i < 8; i++)
    hy_req->params[12 + i] = inq.inq_vid[i];
    for(i = 0; i < 16; i++)
    hy_req->params[20 + i] = inq.inq_pid[i];
    for(i = 0; i < 4; i++)
    hy_req->params[36 + i] = inq.inq_revision[i];

    return(GOOD);
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> hydra_int_handler() function
 * when Hydra interrupts host system, this function is called
 * no parameter range checking is done, su driver will do itself
 */
void
hydra_int_handler()
{
    register int    d_num;     /* which SCSI device */
    unsigned int    i_temp;
```

```
struct su_results      trans_res; /* transport results */
struct su_scsi_id      scsi_id;   /* target ID info */
struct su_dev_cap      cap; /* capability and value */

/*
 *check if device with number hy_req->dev_num has been
opened
 */
d_num = hy_req->dev_num;
if (!(d_num >= 0) && (d_num < SU_MAXUNIT_NUM))
hy_req->req_code = BAD_DEV;

/*
 * check if su device is open
 * and try to open, if it isn't
 */
su_dev_name[sizeof(SUDEVFILE) - 2] = 0x30 + d_num;
if (su_d[d_num] == -1) {
if ((su_d[d_num] = open(su_dev_name, O_RDWR)) < 0)
hy_req->req_code = OPEN_ERR;
}

/*
 * perform appropriate request
 * or handle errors (last two cases)
 */
switch(hy_req->req_code) {
```

```
case CLOSE_DEV:

su_d[d_num] = -1;

err = close(su_dev_name);


case RESET_DEV:

err = ioctl(su_d[d_num], SU_RESET_DEV);

break;


case SCSI_CMD:

err = send_scsi_cmd(d_num);

break;


case GET_INQ_DATA:

err = write_inq_data(d_num);

break;


case REQUEST_SENSE:

err = ioctl(su_d[d_num], SU_REQUEST_SENSE,

&(hy_req->params[0]));

break;


case GET_CAP:

cap.cap = hy_req->params[0];

err = ioctl(su_d[d_num], SU_GET_CAP, &cap);

hy_req->params[1] = cap.value;

break;
```

```
case SET_CAP:
cap.cap = hy_req->params[0];
cap.value = hy_req->params[1];
err = ioctl(su_d[d_num], SU_SET_CAP, &cap);
break;

case GET_RESULTS:
err = ioctl(su_d[d_num], SU_GET_RESULTS, &trans_res);
hy_req->params[0] = trans_res.resid;
hy_req->params[1] = trans_res.reason;
hy_req->params[2] = trans_res.state;
hy_req->params[3] = trans_res.statistics;
break;

case SET_CHUNK_SIZE:
err = ioctl(su_d[d_num], SU_SET_CHUNK_SIZE,
&(hy_req->params[0]));
break;

case SET_RETRY_NUM:
i_temp = hy_req->params[0];
err = ioctl(su_d[d_num], SU_SET_RETRY_NUM, &i_temp);
break;

case GET_SCSI_ID:
err = ioctl(su_d[d_num],
```

```
SU_GET_SCSI_ID, &scsi_id);
hy_req->params[0] = scsi_id.bus_id;
hy_req->params[1] = scsi_id.dev_id;
break;

case SET_TIME_RATE:
i_temp = hy_req->params[0];
err = ioctl(su_d[d_num], SU_SET_TIME_RATE, &i_temp);
break;

case SET_PKT_FLAGS:
err = ioctl(su_d[d_num], SU_SET_PKT_FLAGS,
&(hy_req->params[0]));
break;

case SET_BLOCK_SIZE:
err = ioctl(su_d[d_num], SU_SET_BLOCK_SIZE,
&(hy_req->params[0]));
break;

case BAD_DEV:
errno = EBADF;
err = -1;
break;

case OPEN_ERR:
err = -1;
```

```
        break;

        default:
        errno = ILL_REQ;
        err = -1;
        }

        /* set result and interrupt Hydra */
        hy_req->result = err ? errno : GOOD;
        if (c40_trap(c40_d, trapnum) != 0) {
        syslog(LOG_ERR, "failed to interrupt DSP. Exiting.");
        exit(1);
        }
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> term_handler() function
 * terminates program gracefully
 */
void
term_handler()
{
        int  i;

        /* close devices and exit*/
        for (i = 0; i < SU_MAXUNIT_NUM; i++)
        if (su_d[i] != -1)
```

```
    close(su_d[i]);
    close(c40_d);
    exit();
}
```

```
/*
 * Definitions and data structures shared by host
 * and hydra programs
 * shared.h     12/11/95
 * Artak O. Melkonian, All Rights Reserved, 1994, 1995
 * Department of Electrical and Computer Engineering
 * New Jersey Institute of Technology
 */

/* DSP word size */
#define WORD_SIZE     4

/* size of shared memory space per DSP reserved by Hydra monitor
*/
#define MO_WORDS_PER_DSP   1040 /* DSP words */
#define MO_BYTES_PER_DSP   MO_WORDS_PER_DSP * WORD_SIZE    /*
bytes */

/* DSP trap numbers */
#define TRAP9   0x9

/*
 * Hydra request codes:
 * requests described in scsi/targets/sudef.h
 * specific SCSI terms are described in SCSI standart
 * and in scsi directory of include files
 *
```

```
 * parameter directions:
 * > -     Hydra to host
 * < -     host to Hydra (return)
 */

#define RESET_DEV     1
/*
      no params
 */

#define SCSI_CMD2
/*    params[0]:>      SCSI command block, byte 0..3
        ...
      params[2]:>      SCSI command block, byte 8..11
      params[3]:>      length of command block (6; 10; 12
usually)
      params[4]:>      buffer offset from Hydra shared memory
base (words)
      params[5]:>      length of buffer in Hydra shared memory
(words)
      params[6]:<      returned SCSI status
      params[7}:>      USCSI flags
 */

#define GET_INQ_DATA  3
/* complete inquiry info can be obtained issuing INQUIRY command
*/
```

```
/*
     params[0]:<     device qualifier and device type
     params[1]:<     set, if removable
     params[2]:<     device type qualifier
     params[3]:<     ANSI version
     params{4]:<     async. event notification capability
     params[5]:<     inquiry response data format
     params[6]:<     supports relative addressing
     params[7]:<     supports 32 bit wide data transfers
     params[8]:<     supports 16 bit wide data transfers
     params[9]:<     supports linked commands
     params[10]:<    supports command queueing
     params[11]:<    supports Soft Reset option
     params[12]:<    vendor ID, byte 0
       ...
     params[19]:<    vendor ID, byte 7
     params[20]:<    product ID, byte 0
       ...
     params[35]:<    product ID, byte 15
     params[36]:<    revision level, byte 0
       ...
     params[39]:<    revision level, byte 3
 */

#define REQUEST_SENSE 4
/* complete sense can be obtained also issuing REQUEST SENSE
command */
```

```
/*
     params[0]:<     SCSI sense, byte 0..3
       ...
     params[3]:<     SCSI sense, byte 16..19

     Note that the pointer to params[0] can be casted as a
pointer
     to a struct scsi_extended_sense (from hydra/scsi_incl.h)
     which would be very convenient to work with.
 */

#define GET_CAP 5
/*
     params[0]:>     capability
     params[1]:<     value
 */

#define SET_CAP 6
/*
     params[0]:>     capability
     params[1]:>     value
 */

#define GET_RESULTS  7
/*
     params[0]:<     data bytes not transfered
     params[1]:<     command completion reason
```

```
     params[2]:<     state of command reached
     params[3]:<     some statistics
 */

#define SET_CHUNK_SIZE     8
/*
     params[0]:>     transfer chunk size
 */

#define SET_RETRY_NUM 9
/*
     params[0]:>     number of command retries
 */

#define GET_SCSI_ID   10
/*
     params[0]:<     bus ID
     params[1]:<     target ID
 */

#define SET_TIME_RATE 11
/*
     params[0]:>     completion time rate in sec/512bytes
 */

#define SET_PKT_FLAGS 12
```

```
     params[0]:>      transport packet flags
 */

#define SET_BLOCK_SIZE     13
/*
     params[0]:>      logical block size for block transfer
commands
 */

/*
 * standart result codes
 */
#define GOOD    0
#define ILL_REQ 0xffffffff /*illegal request */

/*
 * parameter block used to exchange request and result
information
 * between host and hydra. Hydra fills it in its shared memory,
 * host reads it, performs request and returns result
 */

/* NOTE that c40 compiler will calculate number of words, not
bytes! */
#define HY_REQ_SIZE  sizeof(struct hydra_request)

#define PARAM_NUM    61
```

```
struct hydra_request {
      unsigned long   dev_num;
      unsigned long   req_code;
      unsigned long   params[PARAM_NUM];
      unsigned long   result;
};
```

APPENDIX C

SOURCES OF THE HYDRA SCSI UTILITIES

```
/*
 * Header file of some useful SCSI or other utilities for Hydra
 * hy_scsi_util.h    12/11/95
 * Portions are copyright (C) 1993 Ariel Corp.
 */

#include <stdlib.h>
#include <stddef.h>
#include "hy_scsi.h"

/* useful definitions for DSP timer */
#define RESET_TIMER() (*(unsigned long *)0x00100020 |= 960)
#define SET_PERIOD(p) (*(unsigned long *)0x00100028 = (unsigned
long)(p))
#define GET_TIMER()   (*(unsigned long *)0x00100024)
#define ELAPSED_TIME(st, end)    (((end) - (st)) * 0.1)
#define SH_MEM(offset)     (*(unsigned long *)(offset))
#define DEV_BSIZE     512
#define RESET_CDB()   hy_req->params[0] = 0, hy_req->params[1] =
0, \
      hy_req->params[2] = 0
```

```
#define SET_USCSI_FLAGS(flags)  hy_req->params[7] |= (flags)

#define CLEAR_USCSI_FLAGS()     hy_req->params[7] = 0x0

/* Hydra standart configuration structures */
typedef struct{
      unsigned long baud;
      int parity, bits;
      } UART_config;

typedef struct{
      int local, global;
      } SramSize;

typedef struct{
      UART_config uartA, uartB;
      unsigned long dram_size, cpu_clock, checksum;
      SramSize sram1_size, sram2_size, sram3_size, sram4_siz
      unsigned   long   l_dram_base,   l_dram_space,   l_jtag_
l_jtag_space;
      unsigned long daughter;
      char revision;
      } hydra_conf;

extern void writeVIC();
extern unsigned long readVIC();
extern void writeVAC();
```

```
extern unsigned long readVAC();
extern char c40_putchar();
extern void c40_printf();
extern void c40_perror();
extern void perror_scsi();
extern void print_sense();

/* functions defined in hydra_int.asm */
extern void GIE_On();
extern void GIE_Off();
extern void EnCache();
extern void SetIntVect();
extern void HostInt();
extern hydra_conf *GetConfig();
extern void Idle();
```

```
/*
 * Header file of some useful SCSI or other utilities for Hydra
 * hy_scsi.h    12/11/95
 * Portions are copyright (C) 1993 Ariel Corp.
 */

#include <errno.h>
#include "shared.h"
#include "scsi_incl.h"

/* request block */
extern struct hydra_request     *hy_req;
```

```
/*
 * Some useful SCSI or terminal output utilities for Hydra
 * hydra_util.c 06/25/95
 * Portions are copyright (C) 1993 Ariel Corp.
 */

#include <ctype.h>
#include <stdarg.h>
#include <math.h>
#include "hy_scsi.h"

/* Hydra SCSI request block */
struct hydra_request *hy_req;

/* extended sense key messages */
char *ext_sense_key[] = {
      "No sense",
      "Recoverable error",
      "Not ready",
      "Medium error",
      "Hardware error",
      "Illegal request",
      "Unit attention",
      "Write protect",
      "Blank check",
      "Vendor unique",
      "Copy aborted",
```

```
      "Aborted command",
      "Equal",
      "Volume overflow",
      "Miscompare",
      "Unknown sense key"
      };

/*
 * error messages for c40_perror
 */
char ue_msg[] = "Unknown error";
/* new error messages can be added without changing anything
else */
char *err_msg[] = {
      /*00*/     "No error",
      /*01*/     "",
      /*02*/     "",
      /*03*/     "",
      /*04*/     "",
      /*05*/     "I/O error",
      /*06*/     "No such device or address",
      /*07*/     "",
      /*08*/     "",
      /*09*/     "Bad file number",
      /*10*/     "",
      /*11*/     "",
      /*12*/     "Not enough memory",
```

```
    /*13*/     "Permission denied",
    /*14*/     "Bad address",
    /*15*/     "",
    /*16*/     "Device busy",
    /*17*/     "",
    /*18*/     "",
    /*19*/     "No such device",
    /*20*/     "",
    /*21*/     "",
    /*22*/     "Invalid argument",
    /*23*/     "",
    /*24*/     "",
    /*25*/     "Invalid request"
    };

void writeVIC( unsigned long add, unsigned long data )
{
    *((unsigned long *)(((0xFFFC0000 | add) >> 2) |
0xB0000000)) = data;
}

unsigned long readVIC( unsigned long add )
{
    return( (*((unsigned long *)(((0xFFFC0000 | add) >> 2) |
0xB0000000))) &0xFF );
}
```

```
void writeVAC( unsigned long add, unsigned long data )
{
	*((unsigned long *)(((0xFFFD0000 | (add << 8)) >> 2) |
0xB0000000)) = (data << 16);
}

unsigned long readVAC( unsigned long add )
{
	return( (*((unsigned long *)(( ((0xFFFD0000 | (add << 8))
>> 2) | 0xB0000000))) >> 16) & 0xFFFF );
}

char c40_putchar( char ch )
{
	int i;

	writeVIC( 0x27, 0xa1 );    /* Disable this interrupt */

	/* Wait until the transmitter is ready */
	while( !(readVAC(0x25) & (unsigned long)0x100) );

	/* Wait until the transmitter is ready again due to VAC
bug */
	while( !(readVAC(0x25) & (unsigned long)0x100) );

	/* Write character to transmitter */
	writeVAC( 0x1E, ch << 8 );
```

```
        for( i=0 ; i < 100 ; )
        i++;

        writeVIC( 0x27, 0x01 );    /* Re-enable this interrupt */

        return( ch );
}

void putstr( char *buf )
{
        int i;

        for( i=0 ; buf[i] != '\0' ; i++ )
        c40_putchar( buf[i] );
}

void xtoa( unsigned long hexval, char *buf )
{
        unsigned long mask=0x0F0000000, i;
        unsigned long temp;

        for( i=0 ; i < 8 ; i++, mask >>= 4 )
        {
        temp = hexval & mask;
        temp >>= (7-i)*4;
        buf[i] = (temp < 10) ? 48+temp : 55+temp;
```

```
        }
        buf[8] = '\0';
}

void ftoa( float fval, char *buf )
{
        int index=0, count, exponent;
        double temp1, temp2;

        if( fval < 0 )
        {
        buf[index++] = '-';
        fval = -fval;
        }

        exponent = fval!=0.0?log10( fval ):0;

        fval /= pow( (double)10.0, (double) exponent );


        if( (fval > -1.0) && (fval < 1.0) && (fval != 0.0) )
        {
        fval *= 10;
        exponent--;
         buf[index++] = '0' + (int)fval;
        fval -= (int)fval;
        buf[index++] = '.';
```

```
}
else
{
buf[index++] = '0' + (int)fval;
fval -= (int)fval;
buf[index++] = '.';
}
for( count=0 ; count < 4 ; count++ )
{
fval *= 10;
buf[index++] = '0' + (int)fval;
 fval -= (int)fval;
}

if( exponent )
{
buf[index++] = 'x';
buf[index++] = '1';
buf[index++] = '0';
buf[index++] = 'e';
ltoa( exponent, buf+index );
}
else
{
buf[index] = '\0';
}
}
```

```
void c40_printf( char *fmt, ... )
{
      va_list ap;
      char *p, *sval, cval;
      long ival;
      float fval;
      char buf[16];

      va_start( ap, fmt );
      for( p=fmt ; *p ; p++ )
      {
      switch( *p )
      {
      case '\n' :
      c40_putchar( (int) 10 );
      c40_putchar( (int) 13 );
      break;
      case '%' :
      switch( *++p )
      {
      case 'd' :
      ival = va_arg( ap, int );
      ltoa( ival, buf, 10 );
      putstr( buf );
      break;
      case 'f' :
```

```
fval = va_arg( ap, float );
ftoa( fval, buf );
putstr( buf );
break;
case 'x' :
ival = va_arg( ap, int );
xtoa( ival, buf );
putstr( buf );
break;
case 'c' :
cval = va_arg( ap, char );
c40_putchar( cval );
break;
case 's' :
sval = va_arg( ap, char * );
putstr( sval );
break;
default :
c40_putchar( *p );
break;
}
break;
default:
c40_putchar( *p );
break;
}
}
```

```
    va_end( ap );
}

void c40_perror(char *msg)
{
    c40_printf("%s: (E%d) %s.\n", msg, errno,
    ((err_msg[errno] != "") &&
    (errno < (sizeof(err_msg)/sizeof(char *))) && (errno >=
0)) ?
    err_msg[errno] : ue_msg);
}

void perror_scsi()
{
    errno = hy_req->result;
    c40_printf("\nRequest: (%d), ", hy_req->req_code);
    c40_perror("got error when accessing SCSI device");
    stop_prog();
}

void print_sense(s)
unsigned long   s;
{
    c40_printf("Sense key: %s.\n", ((s >= 0) && (s < 0xf)) ?
    ext_sense_key[s] : ext_sense_key[0xf]);

}
```

```
FP    .set AR3

; Some TMS320C40 assembly language code
; to perform low-level operations

      .globl     _SetIntVect
_SetIntVect:
      ldep  ivtp,ar0

      ldi   sp,ar1
      ldi   *-ar1(1),ir0    ;Get interrupt to set

      ldi   *-ar1(2),r0     ;Get  pointer  to  interrupt  handler
routine
      sti   r0,*+ar0(ir0)

      rets

 ; enable global interrupts
      .globl     _GIE_On

 _GIE_On:
      or    02000h,st

      rets

 ; disable global interrupts
```

```
        .globl      _GIE_Off

_GIE_Off:
        ldi   02000h,r0
        not   r0

        and   r0,st

        rets

; call trap 0x7 to generate host interrupt
        .globl      _HostInt
_HostInt:
        trap  7h

        rets

; call trap 0x8 to get pointer to copy of Hydra configuration in
R0
        .globl      _GetConfig
_GetConfig:
        trap  8h

        rets

; enable cache
        .globl      _EnCache
```

```
_EnCache:
     or    02000h,st

     rets

; stop CPU and wait for interrupts
     .globl     _Idle
_Idle:
     idle

     rets

     .end
```

APPENDIX D

EXAMPLE HOST-HYDRA PROGRAM PAIR USING HYDRA SCSI SERVICES

```
/*
 * Host side program running on Sun SPARC
 * station for Hydra board
 * host_scsi.c  12/11/95
 * usage: host_scsi hscsid_pid
 * Artak O. Melkonian, All Rights Reserved, 1994, 1995
 * Department of Electrical and Computer Engineering
 * New Jersey Institute of Technology
 */
#include <vc40dsp.h>
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>

#define C40DEVFILE    "/dev/vc40b1"
#define HYDRA_EXE     "../hydra_side/hydra_scsi.x40"
#define NUMSYMS (sizeof(symnames)/sizeof(char *))
#define HYDRA_INT     SIGUSR1

int  c40_d;      /* c40 device descriptor */
long err;
```

```
pid_t hscsid_pid;
char  *symnames[] = { /* symnames in DSP prog */
      "_shmem_base"
      };
struct symtab   symtable[NUMSYMS];

/* termination signals handler */
void hydra_int_handler();
void term_handler();


/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> main() function
 */

main(argc, argv)
int  argc;
char *argv[];
{
      int  i;
      struct vc40info hydra_info;
      u_long     e_addr;    /* DSP entry address */

      /* check command line */
      if (argc != 2) {
      printf("usage:  host_scsi hscsid_pid\n");
      printf("   hscsid_pid - pid of hscsid server deamon\n");
      exit(1);
```

```
}

hscsid_pid = (pid_t)strtol(argv[1], NULL, 0);
if (hscsid_pid <= 2) {
printf("hscsid: trapnum out of range.\n");
exit(1);
}

/* open Hydra board */
if ((c40_d = open(C40DEVFILE, O_RDWR)) <= 0) {
perror("host_scsi: failed to open Hydra DSP");
exit(1);
}

if (c40_reset(c40_d) != 0) {
perror("host_scsi: failed to reset DSP");
exit(1);
}

/* get and print DSP configuration information */
if (c40_getinfo(c40_d, &hydra_info) != 0) {
perror("host_scsi: failed to get DSP info");
exit(1);
}
printf("%s: %d-DSP Hydra board with %d Mbyte DRAM.\n",
C40DEVFILE,                                   hydra_info.numdsp,
hydra_info.dram_size/1024/1024);
```

```
fflush(stdout);

/*
 * loading DSP program and getting entry address of
 * DSP program in Hydra memory, and symtab info too
 */
if (c40_load(c40_d, HYDRA_EXE, &e_addr,
NUMSYMS, symnames, symtable) == 0) {
printf("host_scsi: failed to load DSP program: %s\n",
cofferr);
exit(1);
}
printf("DSP program loaded. Entry address: 0x%lx\n",
e_addr);

/* check if all symbols are defined by c40_load */
err = 0;
for(i = 0; i < NUMSYMS; i++) {
if (symtable[i].type == T_UNDEF) {
printf("host_scsi: undefined symbol \'%s\'\n",
symnames[i]);
err = 1;
}
}
if (err)
exit(1);
```

```
/* enable Hydra interrupts */
if (c40_enint(c40_d, HYDRA_INT) != 0) {
perror("host_scsi: failed to enable Hydra interrupts");
exit(1);
}

/* install interrupt handlers */
signal(HYDRA_INT, hydra_int_handler);
signal(SIGTERM, term_handler);
signal(SIGINT, term_handler);
signal(SIGHUP, term_handler);
signal(SIGQUIT, term_handler);

/* run DSP code */
if (c40_run(c40_d, e_addr) != 0) {
perror("host_scsi: failed to run DSP program");
exit(1);
}

/* sleep, wait for signals */
while(1) {
pause();
}
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> hydra_int_handler() function
```

```
 * process Hydra interrupt
 */
void
hydra_int_handler()
{
        kill(hscsid_pid, HYDRA_INT);
}


 /*
  * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> term_handler() function
  * terminates program gracefully
  */
 void
 term_handler()
 {
        int   i;

        close(c40_d);
        printf("DSP halted.\n");
        fflush(stdout);
        exit();
 }
```

```
/*
 * Hydra side program running on TMS320C40 DSP
 * and using Hydra SCSI services
 * hydra_scsi.c 12/11/95
 * Artak O. Melkonian, All Rights Reserved, 1994, 1995
 * Department of Electrical and Computer Engineering
 * New Jersey Institute of Technology
 */

#include <stdlib.h>
#include <stddef.h>
#include <hy_scsi_util.h>

/* some useful macros */
#define cdbp    ((union scsi_cdb *)(&(hy_req->params[0])))

#define VERSION "1.0"
#define REQUEST_HOST()     HostInt(), Idle()

unsigned long   shmem_base;
unsigned long   shmem_size;
hydra_conf *conf;
unsigned long   num_blocks;
unsigned long   block_size;     /* in bytes */

/* trap 0x9 handler */
void c_int01()
```

```
{
    /* just wakes up the CPU from idle state */
}

void stop_prog()
{
    c40_printf("DSP program terminated.");
    while(1);
}

unsigned long req_sense()
{
    hy_req->req_code = REQUEST_SENSE;
    REQUEST_HOST();
    if (hy_req->result != GOOD) {
    errno = hy_req->result;
    c40_perror("Could not request sense");
    stop_prog();
    }
    return(((struct    scsi_extended_sense    *)&(hy_req-
>params[0]))->es_key);
}

void reset_target()
{
    hy_req->dev_num = 0;
    hy_req->req_code = RESET_DEV;
```

```
        REQUEST_HOST();
        if (hy_req->result != GOOD)
        perror_scsi();
        /* clear UNIT_ATTENTION state after reset by requesting
sense */
        (void)req_sense();
}

/*
 * >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> strspcpy function:
 * copies a string of length n from source_str to dest_str
 * until the first space character
 */

strspcpy(dest_str, source_str, n)
register char   *dest_str;
register char   *source_str;
int   n;
{
        register int    i;

        for(i = 0; i < n; i++) {
        *(dest_str + i) = *(source_str + i);
        if (*(dest_str + i) == ' ') {
        *(dest_str + i) = '\0';
        return;
        }
```

```
    }
    *(dest_str + n) = '\0';
}

void main()
{
    unsigned long   data_length;
    char  text[80];
    unsigned long   st1, et1, st2, et2;

    /*
     * set timer period
     * with 100ns cycle (40MHz clock),
     * timer can count 7.16 min before overflow
     */
    SET_PERIOD(0xffffffff);
    RESET_TIMER();

    /*  install  interrupt  handler  and  enable  cache  and
interrupts */
    EnCache();
    SetIntVect(TRAP9, c_int01);
    GIE_On();

    /* print greeting message */
    c40_printf("%sSCSI  Disk  Control  Demo  Program,  ver.  %s,
%s\n",
```

```
"□[H□[2J", VERSION, "Copyright (C) Artak Melkonian");
c40_printf("   Low   level   SCSI   control   by   Hydra   DSP
board\n");
c40_printf("   using   SCSI   bus   of   host   FORCE   CPU-2CE
(SPARC)\n\n");



/* get board configuration */
conf = GetConfig();
shmem_base = conf->l_dram_base;
shmem_size = conf->dram_size * 1024;
shmem_size = 1024 * 1024;  /* needed for old Hydra */
c40_printf("Shared memory base: %xH\n", shmem_base);
c40_printf("Shared memory size: %d Mbytes\n\n",
(shmem_size / (1024 * 1024)) * WORD_SIZE);

/* initializing hydra_request parameter block */
hy_req = (struct hydra_request *)(shmem_base + shmem_size
-
MO_WORDS_PER_DSP * (conf->daughter ? 4 : 2) -
HY_REQ_SIZE);

/* get scsi ID info for target */
hy_req->dev_num = 0;
hy_req->req_code = GET_SCSI_ID;
REQUEST_HOST();
if (hy_req->result == GOOD)
```

```
c40_printf("Controlling SCSI bus %d target %d\n",
hy_req->params[0], hy_req->params[1]);
else
perror_scsi();

/* get inquiry info for target */
st1 = GET_TIMER();
hy_req->dev_num = 0;
hy_req->req_code = GET_INQ_DATA;
st2 = GET_TIMER();
REQUEST_HOST();
et2 = GET_TIMER();
if (hy_req->result == GOOD) {
if (hy_req->params[0] & DTYPE_MASK != DTYPE_DIRECT) {
c40_printf("Target is not a direct access device !\n");
stop_prog();
}
if (hy_req->params[1])
c40_printf("Removable ");
c40_printf("Direct   Access   SCSI-%d   Device,   ",   hy_req-
>params[3]);
strspcpy(text, &(hy_req->params[12]), 8);
c40_printf("Vendor: \'%s\', ", text);
strspcpy(text, &(hy_req->params[20]), 16);
c40_printf("Product: \'%s\'.\n", text);
} else
perror_scsi();
```

```
et1 = GET_TIMER();
c40_printf("□[7m No SCSI request: %f us, waiting for host:
%f us □[m\n",
ELAPSED_TIME(st1, et1), ELAPSED_TIME(st2, et2));

/* reset target */
reset_target();

/* get disk capacity (READ CAPACITY) */
st1 = GET_TIMER();
hy_req->dev_num = 0;
hy_req->req_code = SCSI_CMD;
data_length = 2;/* words */
RESET_CDB();
cdbp->scc_cmd = SCMD_READ_CAPACITY;   /* only this needed
*/
hy_req->params[3] = CDB_GROUP1;
hy_req->params[4] = 0;
hy_req->params[5] = data_length;
CLEAR_USCSI_FLAGS();
SET_USCSI_FLAGS(USCSI_READ | USCSI_CONT);
st2 = GET_TIMER();
REQUEST_HOST();
et2 = GET_TIMER();
if ((hy_req->result == GOOD) && (hy_req->params[6] ==
USCSI_STATUS_GOOD)) {
num_blocks = SH_MEM(shmem_base);
```

```
block_size = SH_MEM(shmem_base + 1);
c40_printf("Disk has %d logical blocks %d bytes each,
total space: %d Mbytes.\n",
num_blocks, block_size, num_blocks * block_size /
1048576);
} else
perror_scsi();
et1 = GET_TIMER();
c40_printf("□[7m SCSI cntl request: %f us, waiting for
host: %f us □[m\n",
ELAPSED_TIME(st1, et1), ELAPSED_TIME(st2, et2));

/* set appropriate block size */
if (block_size != DEV_BSIZE) {
hy_req->req_code = SET_BLOCK_SIZE;
hy_req->params[0] = block_size;
REQUEST_HOST();
if (hy_req->result != GOOD)
perror_scsi();
}

/* test if disk is ready (TEST UNIT READY) */
hy_req->req_code = SCSI_CMD;
RESET_CDB();
MAKECOM_COMMON(&(hy_req->params[0]), SCMD_TEST_UNIT_READY,
0);
hy_req->params[3] = CDB_GROUP0;
```

```
hy_req->params[5] = 0;      /* no data */
CLEAR_USCSI_FLAGS();
REQUEST_HOST();
if (hy_req->result == GOOD) {
if ((hy_req->params[6] != USCSI_STATUS_GOOD) &&
(req_sense() == KEY_NOT_READY)) {
c40_printf("Disk is not ready... Trying to start... ");
hy_req->req_code = SCSI_CMD;
RESET_CDB();
MAKECOM_COMMON(&(hy_req->params[0]),
SCMD_START_STOP, 0);
cdbp->scc_b4 = 0x01;  /* start */
hy_req->params[3] = CDB_GROUP0;
REQUEST_HOST();
if (hy_req->result == GOOD) {
if (hy_req->params[6] !=
USCSI_STATUS_GOOD) {
c40_printf("Failed.\n");
perror_scsi();
} else {
RESET_CDB();
MAKECOM_COMMON(&(hy_req->params[0]),
SCMD_TEST_UNIT_READY, 0);
REQUEST_HOST();
if ((hy_req->result == GOOD) &&
(hy_req->params[6] ==
USCSI_STATUS_GOOD)) {
```

```
c40_printf("Done.\n");
} else
perror_scsi();
}
} else
perror_scsi();
}
} else
perror_scsi();

/*
 * write 1M of shared memory onto disk
 * driver will write data by chunks (system default size)
 */
c40_printf("Writing 1M of shared memory onto disk using
data chunks in SCSI driver.\n");
st1 = GET_TIMER();
hy_req->req_code = SCSI_CMD;
data_length = 262144; /* DSP words */
RESET_CDB();
MAKECOM_G0(cdbp, SCMD_WRITE, 0, 0,
(data_length * WORD_SIZE) / block_size);
hy_req->params[3] = CDB_GROUP0;
hy_req->params[4] = 0;
hy_req->params[5] = data_length;
CLEAR_USCSI_FLAGS();
SET_USCSI_FLAGS(USCSI_BLOCK);
```

```
st2 = GET_TIMER();
REQUEST_HOST();
et2 = GET_TIMER();
if (hy_req->result == GOOD)
if (hy_req->params[6] == USCSI_STATUS_CHECK)
print_sense(req_sense());
et1 = GET_TIMER();
c40_printf("□[7m SCSI data request: %f us, waiting for
host: %f us □[m\n",
ELAPSED_TIME(st1, et1), ELAPSED_TIME(st2, et2));

/*
 * write 1M of shared memory onto disk
 * driver will write data by chunks
 */
hy_req->req_code = SET_CHUNK_SIZE;
hy_req->params[0] = 1048576;
REQUEST_HOST();

c40_printf("Writing 1M of shared memory onto disk using
large data chunks in SCSI driver.\n");
st1 = GET_TIMER();
hy_req->req_code = SCSI_CMD;
data_length = 262144; /* DSP words */
RESET_CDB();
MAKECOM_G0(cdbp, SCMD_WRITE, 0, 0,
(data_length * WORD_SIZE) / block_size);
```

```
hy_req->params[3] = CDB_GROUP0;
hy_req->params[4] = 0;
hy_req->params[5] = data_length;
CLEAR_USCSI_FLAGS();
SET_USCSI_FLAGS(USCSI_BLOCK);
st2 = GET_TIMER();
REQUEST_HOST();
et2 = GET_TIMER();
if (hy_req->result == GOOD)
if (hy_req->params[6] == USCSI_STATUS_CHECK)
print_sense(req_sense());
et1 = GET_TIMER();
c40_printf("□[7m SCSI data request: %f us, waiting for
host: %f us □[m\n",
ELAPSED_TIME(st1, et1), ELAPSED_TIME(st2, et2));

/*
 * read 1M of disk into shared memory
 * driver will read data by chunks
 */
c40_printf("Reading 1M of disk into shared memory using
data chunks in SCSI driver.\n");
st1 = GET_TIMER();
hy_req->req_code = SCSI_CMD;
data_length = 262144; /* DSP words */
RESET_CDB();
MAKECOM_G0(cdbp, SCMD_READ, 0, 0,
```

```
(data_length * WORD_SIZE) / block_size);
hy_req->params[3] = CDB_GROUP0;
hy_req->params[4] = 0;
hy_req->params[5] = data_length;
CLEAR_USCSI_FLAGS();
SET_USCSI_FLAGS(USCSI_BLOCK | USCSI_READ);
st2 = GET_TIMER();
REQUEST_HOST();
et2 = GET_TIMER();
if (hy_req->result == GOOD)
if (hy_req->params[6] == USCSI_STATUS_CHECK)
print_sense(req_sense());
et1 = GET_TIMER();
c40_printf("□[7m SCSI data request: %f us, waiting for
host: %f us □[m\n",
ELAPSED_TIME(st1, et1), ELAPSED_TIME(st2, et2));



stop_prog();
}
```

```
/*
 * Memory allocation map for TMS320C40 linker
 * hydra_scsi.lnk    12/11/95
 * Artak O. Melkonian, All Rights Reserved, 1994, 1995
 * Department of Electrical and Computer Engineering
 * New Jersey Institute of Technology
 */

/* SPECIFY THE MEMORY MAP OF HYDRA PROGRAM */

MEMORY
{
      INT_ROM:   org = 0x000000  len = 0x1000    /* INTERNAL ROM
*/
      INT_RAM0:  org = 0x2FF800  len = 0x400     /* INTERNAL RAM
BLOCK 0 */
      INT_RAM1:  org = 0x2FFC00  len = 0x400     /* INTERNAL RAM
BLOCK 1 */
      L_SRAM:    org = 0x40001200  len = 0xec00  /*  LOCAL  BUS
SRAM */
      G_SHMEM:   org = 0x8d000000 len = 0xfefc0  /*  GLOBAL  BUS
SHMEM */
      G_SRAM:    org = 0xc0000000 len = 0x10000  /*  GLOBAL  BUS
SRAM */
}
```

```
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    .text:  > G_SRAM   /* EXECUTABLE CODE */
    .cinit: > G_SRAM   /* INITIALIZATION TABLES */
    .const: > G_SRAM   /* CONSTANTS */
    .stack: > G_SRAM   /* SYSTEM STACK */
    .sysmem:     > G_SRAM   /* DYNAMIC MEMORY (HEAP) */
    .bss:   > G_SRAM   /* GLOBAL & STATIC VARIABLES */
}
```

REFERENCES

1. Systems Small Computer System Interface (SCSI). American National Standard for Information, ANSI X3T9.2/82.-2 - Rev. 17B, 1986.

2. Systems Small Computer System Interface - 2 (SCSI-2). American National Standard for Information, ANSI X3T9.2/86 - 109 - Rev. 10C - X3T9/89-042, 1990.

3. TMS320C4x User's Guide. Texas Instruments, 2564090-9721 revision A, 1991.

4. User's Manual for the V-C40 Hydra. Ariel, Version 0.60, 1994.

5. SunOS Reference Manual. Sun Microsystems, Sun Part Num.: 800-3827-10, 1990.

6. V-C40 Utility Library. Ariel, Computer file: vc40lib.doc, v 1.2, 1993.

7. SPARC CPU-2CE Technical Reference Manual. Force Computers, Part Num.: 049-12441-102 ver. A1, 1993.

9. Stirling P. *A Solaris 2 SCSA Tutorial,* Computer file, 1993.

10. Solaris 2.1 Online Reference Manual. Sun Microsystems, 1992.

11. Writing Device Drivers SunOS 4.1. Sun Microsystems, Sun Part Num: 800-3851-10, 1990.

12. Writing Device Drivers SunOS 5.1. Sun Microsystems, 1992.

13. Writing Device Drivers SunOS 5.2. Sun Microsystems, 1993.

14. TMS320 Floating-Point DSP Optimizing C Compiler User's Guide. Texas Instruments, 2576391-9721 rev. A, 1991.

15. Hipson P. *Advanced C Programming* SAMS Publishing, Indianapolis, Indiana, 1992.

16. Solaris 1.x (SunOS 4.x) Handbook for SMCC Peripherals. Sun Microsystems, Part No.: 801-2424-10, rev. A, 1992.

17. VIC068A/VAC068A User's Guide. Cypress Semiconductor, 1992.