FILE MANAGER – DISK DRIVER INTERFACE FOR

OPERATING SYSTEM PRAGMATICS CSC 159

A Project

Presented to the faculty of the Department of Electrical and Electronic Engineering

California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Engineering

by

Jayaram Prabhu Durairaj

SPRING
2013

FILE MANAGER – DISK DRIVER INTERFACE FOR

OPERATING SYSTEM PRAGMATICS CSC 159

A Project

by

Jayaram Prabhu Durairaj

Approved by:

_____, Committee Chair
Dr. John Clevenger

_____, Second Reader
Chung –E Wang, Ph. D

_____
Date

Student: Jayaram Prabhu Durairaj

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

_____, Graduate Coordinator _____
Dr. Suresh Vadhva Date

Department of Electrical and Electronic Engineering

Abstract

of

FILE MANAGER – DISK DRIVER INTERFACE FOR

OPERATING SYSTEM PRAGMATICS CSC 159

by

Jayaram Prabhu Durairaj

Operating Systems are one of the critical architectural components every Computer Engineering Student should be aware of. Most Operating System course-work is strong on theory and weak on practical. Operating System Pragmatics CSC 159 course at CSUS explains the fundamental principles in great detail, including process management, inter-process communication, semaphores, message passing, input/output, character device drivers, memory management, interrupts, paging and virtual memory. The main aspect of this project is to add an Interface for File Systems and Disk Drivers to the existing course-work model.

This project is an extension of existing kernel module designed from the course work CSC 159. The File System Design for this project will be based on collective knowledge gathered during research with many open source File Systems. The Project comprises three stages in development. Each stage includes completion and loading of a module in to the kernel.

In stage one, Device Driver for IDE Disk is designed. Both the Interrupt and Busy Polling Mechanisms are used for obtaining status of IDE drive, reading data and writing data. Timer Synchronization and Buffer Management plays an important role to be considered for the design.

In stage two, FAT16 File System is designed with basic design taken from an embedded system file system. The Boot Block, FAT Area, Data Area are all manually set by the File System Initialization Routine. The Total Capacity of the File System is also set manually.

The above two stages are involved in modules that are considered as part of the existing Operating System, but in the final stage, specific tools are created for loading files to disks apart from operating on the designed existing kernel modules. This feature helps the instructors to load files to the IDE Disk without the operating system being run. This feature is planned to be used in case of emergencies like accidental meta-data over write or kernel crashes.

_____, Committee Chair
Dr. John Clevenger

_____
Date

ACKNOWLEDGEMENTS

Apart from the efforts of me, the success of any project depends largely on the encouragement and guidelines of the guide. I take this opportunity to express my gratitude to Dr. John Clevenger who has been instrumental in the successful completion of this project. I would like to show him my greatest appreciation. I cannot say thank you enough for his tremendous support and help. I feel motivated and encouraged every time I attend his meeting. Without his encouragement and guidance, this project would not have materialized. His guidance and support was vital for the success of the project.

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Operating Systems are one of the critical architectural components every Computer Engineering Student should be aware of. This piece of software is responsible for management of computer operations and resource management. It is the base system that handles hardware operations and user applications. CSC 159 course works builds an operating system from the scratch, which is not something that is done in recent days. CSC 159 course work has the potential to serve as a source for software innovation for students to build upon it. The course work has extensive tools and documentation for students to start writing their own operating system. These tools can also evolve giving students the ability to build bottom up systems. Students are expected to ramp up and code many major components in a small time span of one semester. This system software, which manages and controls the activities of the system, is very explicit and small in its purpose. The hardware, for which the Operating System software is written, is referred to as target system. The target system used in this course work is an INTEL Pentium machine. CSC 159 puts together lots of basic kernel operations. Some of the data structures used are not dynamic allocated for better understanding and simplicity. The very core of this course work is to introduce students to a very important system in a much simpler way.

Most Operating System course-works are strong on theory and weak on practical applications. The importance of different components that form the base of an Operating System is really a tough one to decipher from online open source projects available online say UBUNTU, OPENSUSE, FEDORA, MINIX and many others since they have evolved with the open source community for long time. Hence to understand these basic components in pragmatic details, CSC 159 coursework helps. Operating System Pragmatics CSC 159 course at CSUS explains the

fundamental principles an Operating System must implement, such as Inter Process Communication, Process Management, Memory Management and Input-Output Processing in great details with some external and internal hardware. Performance is never taken into consideration here but coding details are given more importance. At the end of CSC 159 course work our team ended up with a small microkernel operating system called NanOS which is around 14 files with approximately 3000 lines of codes which could handle some hardware and software Interrupts, spawn and kill Processes, exchange messages between processes, manage memory for created processes and mange some devices like printer and UART ports. These components were all covered and coded in one semester and this serves as the existing code base over which the project is going to get built on. NanOS handles all the above-mentioned implementations in a static mode, where all data are store on RAM (Random Access Memory). The data is not persistent since a reboot will reset the system. Data are not stored in any storage device like a flash or hard disk, so that they can be retrieved again for usage. Once the operating system is reloaded, all the information gets reset and reloaded again. Also NanOS has drivers that can control UART ports and Printer ports, but other drivers like Graphics Driver, Disk Driver are not available since they cannot be covered in the whole course work taking time into consideration.

Students need the ability to use a file system, and the corresponding disk driver, without having to write it themselves, and at the same time there needs to be an external mechanism to add data into that file system so the student Operating System can access it for various applications such as editors, compilers, etc. This project provides the tools and source code to accomplish the above-mentioned task. This project's base idea originates from this aspect. Creating a Graphics Driver is outside the scope of this project. This project will concentrate on creating a Disk Driver for NanOS and a simple File System that can manage and manipulate files

in a very basic way. ATA/IDE Device Driver is taken into consideration for implementation. There are many open source file systems like Extended File Systems say ext2, ext3, ext4 but FAT16 File System is considered for this project because of its simplicity and easy to use data structures.

The main goal of this project is to give students and professors the ability to move or transfer files from host to target systems. This must be achieved in two ways, one in a blind manner where the sole purpose is to just transfer files and the second one is to integrate the code base in the students created operating system so that they can make use of the file system to create and manipulate files on the fly for their specific usage. The User Guide at the end will help in the usage options and the remaining chapters will explain the project in detail. This project requires the reader to have some basic knowledge on operating system concepts and data structures. This project is an extension of existing end product of CSC 159 course work. Though simpler forms of coding standard are followed, concepts like pointers and linked lists are required since they are not emphasized in the chapters. Also optimization and structure of the code base are not given much importance.

CHAPTER 2

OVERVIEW OF THE PROJECT

2.1. Existing NANOS Architecture

An Operating System is the longest running program on a computer. The primary purpose of the Operating System is to share computer resources such a memory, CPU time, disk space, etc. The CSC 159 course work helps in understanding how to create an Operating System from the scratch. A microkernel OS architecture is considered for the course work. The kernel that describes the core part of the Operating System, gives importance to all the details that needs to be handled, based on the hardware used. One advantage of the microkernel approach is moving program-visible features out of the kernel with ease. Operating System has many responsibilities beyond performing reads and writes from storage media and these operations does not need the kernel to be involved. Device (Disk) Driver takes control on performing reads and writes on requests from kernel. File System modules can be replaced with ease in microkernel design. By making all the Operating System modules follow Request-Response interface, the complexity is kept simple. All the other system level operations are handled by kernel code such as maintaining synchronization between processes, handling hardware ports and interrupts. At the end of NanOS project, our team had a basic functioning microkernel Operating System that can handle Inter process Communication, Process Management, Memory Management, Interrupt Handlers, and common Device Drivers such as printer and UART Terminals. Project consists of totally 18 files with approximately 3000 lines of code.

CSC 159 course work provides extensive tools to achieve the successful run of the newly created operating system on the target. A suite of tools called SPEDE System Programmer's Educational Development Environment provides the ability to run the code under a debugger so that every line of code can be traced under debugger control. This is intended for INTEL Pentium

CPUs. SPEDE supports C++ and ANSI C and provides language compilers, linkers and assemblers. It also has a set of tools to make development easier and faster for students. The resulting machine language program is compiled, linked and assembled on the host system and then transferred using tools to target system using UART port connection between the systems. SPEDE Libraries were used extensively to support some of the basic features in INTEL Pentium hardware. All the libraries were included under a common header file named "spede.h". This file was created with extensive comments that explain the functions and defines that were chosen from each file. The SPEDE-2000 Lab Manual also explains each of the function calls in detail. Some of the functions that were used are listed as follows

1. printf() (from spede/stdio.h)
2. memset(), strcpy(), strtok() (from spede/string.h)
3. clock(), time() (from spede/time.h)
4. assert() (from spede/assert.h)
5. inportb(), outportb(), inportw() (form spede/machine/io.h),
6. get_idt_base(), get_cs(), set_cr3() (from spede/machine/proc_reg.h)

These functions offload minute details that students have to deal with the internals of the hardware and hence important Operating System components can be concentrated upon.

The following figure gives an overview of existing architecture of NANOS represented in blue blocks and the newly added modules specifically for this project represented in red blocks.

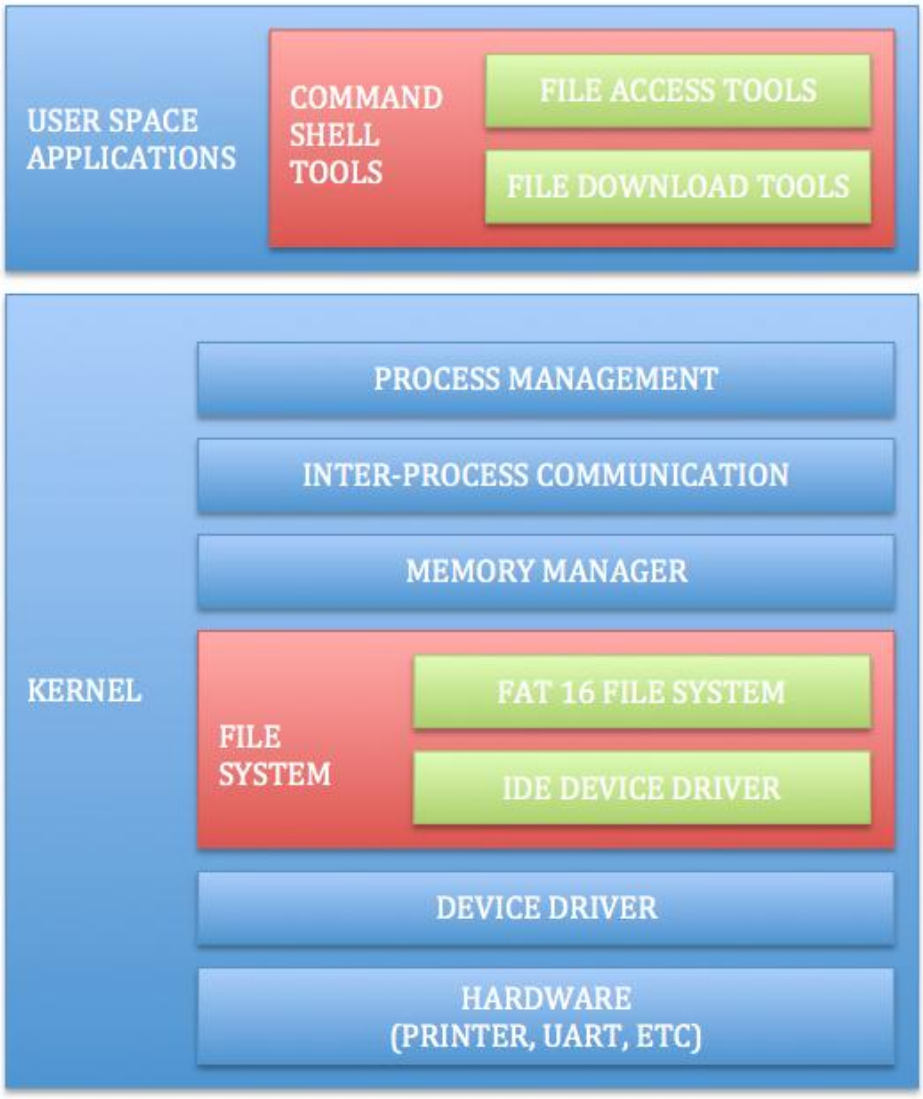

Figure 2.1
PROJECT OVERVIEW

2.2. Features Implemented

Christopher Giese's ATA ("IDE") hard drive and ATAPI CD-ROM demo code [1] is used as a base for constructing Disk Device Driver. This piece of code have lot of issues such as byte count read from disk must be always a multiple of 2048 otherwise they cannot be reliable.

There are lots of known blocking delays. This code is not fully interrupt driven and is not suitable for high performance OS.

This device driver is handled by files ata.c, ata_process.c. These files handle both the low-level hardware communication and the process that is used as a driver, which provides an interface for other processes to handle the disk such as File System. This code handles disk communication using three segments. The first segment deals with ATA/ATAPI data structures and device probing modules; the second segment deals with probing partitions, which may be available on the disk drives, and the third segment deals with modules that are used to handle the Hard Disk Controller (HDC) interrupts. The reference code has been modified to fit the need of this project and most of the structures and module implementation are not optimized for high performance. Also almost all the techniques used in this project like device probing and partition accessing is straightforward and simple. These details will be explained in chapter " Disk Driver Overview".

For simplicity, FAT 16 file system is considered for the design of this project. FullFAT, a High Performance, Thread-Safe Embedded FAT File-System created by James Walmsley that is distributed with GNU General Public License, is considered for reference [2]. FullFAT supports many features that are available in commercial file systems such as write buffer and read buffer caching on files, optimizations on unaligned access of data and more unit test codes for almost all modules. However much of the complex features are disabled for simplicity purposes. Some of the major components that are used for this project include

1. FAT File-System that handles FAT access and traversal.
2. IOMAN I/O Manager that handles IO buffers for FullFAT safely. It provides a simple static interface to the rest of FullFAT to manage buffers. It also defines the public interfaces for Creating and Destroying a FullFAT IO object.

3. FILEIO FILE I/O Access that provides an interface to allow File I/O on a mounted IOMAN for the FAT file-system.
4. DIR that handles Directory Traversal and access
5. MEMORY that has FullFAT Memory Access Routines, which handles memory, access in a portable way and provides simple, fast, and portable access to memory routines. These are only used to read data from buffers. That is LITTLE ENDIAN due to the FAT specification.
6. BLK Block Calculator that handle Block Number conversions.
7. FFTERM provides a simple platform for independent and extendable console/terminal program running over multiple, thread-safe environment. However, only some of the modules of this block are included in the project and other are disabled or not used.
8. SAFETY features that Process Safety for FullFAT, which provides semaphores, and thread-safety for FullFAT. This option is disabled.
9. TIME Real-Time Clock Interface that allows FullFAT to time-stamp files. This option is disabled.
10. This also has Independent implementation of MD5 for file protection but it is disabled.
11. This system maintains its own set of data structures and some cross references were done for tweaking the code.

The file system requires partition details, which are taken from the MBR, Master Boot Record. Though device driver maintains these details, since FullFAT has unique data structures to access the details, the structure type casting is performed in many modules. These details will be explained in chapter " File System Overview".

Another important part of this project is to provide students the ability to download files from host system to target system. This is achieved using Embedded Kermit with true sliding

windows, authored by Frank da Cruz, Columbia University in the City of New York [3]. UART port on the system is used for communication and the mentioned Kermit program is used to achieve the safe reliable transfer of data between the systems. USB ports are used from the host system with the help of USB to UART cables. In NanOS, UART ports are managed using device drivers that use Interrupt driven model. Due to timing issues in communication between host and target system using the Embedded Kermit Protocol, the Device drivers were changed to operate on polled IO mode. NanOS had the interrupt driven mode since the device drivers were written to handle communication between terminals. A single module that can handle both the sending and receiving functions handles this requirement. However only the receiving model has been incorporated in the NanOS and it can be extended to provide ability to transfer files form target to host system. In the Shell process, one of the NanOS processes, predefined function calls are added so that this downloading feature can be configured, started and stopped, as the user wants. Since the files are downloaded from host system, there is another part of script that needs to be run so that the communication can be achieved. This package on the host system has configuration files, which control various parameters required by both the host and target systems. They also have control on which directories can be transferred from host system to target system. Two modes of operation can be considered based on how the user wants this option enabled. This can be used either blindly just to download files to the target machine or the source files can be added to the users project so that the whole package can be used in their own Operating System. These details will be explained in chapter " File Download Overview".

Process scheduling and CPU time slicing are handled in the kernel code using trap structures and switching on timer interrupts. The whole process can be better understood while following the execution of main program. Once the DLI image is created (the executable file) and downloaded to target machine the main function defined in main.c starts running. Static data

structures, with defined size, such as ready queue, available queue, sleep queue, available semaphore queue, run-time user stacks which are used to hold process details necessary for context/process switching, semaphores, message boxes and some basic global variables are initialized in the main function. Necessary Interrupt service vectors and routines are enabled. For this project we enable hardware Timer Interrupt, Primary and Secondary Hard Disk Controller Interrupts. Some of the software interrupts used are as follows

1. Spawn Process to create new process
2. Sleep Process to make a process idle in between switching
3. Message Send an Receive for IPC

The above features are handled using files main.c (InitControl function) with SPEDE libraries. All the processes that are part of initial services to the operating system are added to the active process list. Process scheduling is handled using modules named Scheduler() (defined in main.c) and Timer Interrupt, which are responsible for changing the state of processes and process queue management. All the interrupts are trapped and processed using module Kernel() (defined in main.c), each interrupts is marked with an id and respective hardware and software interrupted kernel services are carried out accordingly. Explanation on Process management, Interrupt handling, Inter Process communication and synchronization using Message passing interface and semaphores are beyond the scope of this project. More details regarding the data structures and modules used in the NANOS Kernel can be found in CSC 159 course workbook "Building your first Operating System, A Microkernel approach" [4].

2.3. Device Setup

Details regarding IBM Pentium PC Target System considered for this project:

1. Intel Pentium II Processor @ 400MHz clock

2. Cache Memory 512K, Base Memory 640K, Extended Memory 261120K
3. Diskette Drive A 1.44M, 3.5 in
4. Primary Master Disk LBA, UDMA 2, 40018MB
5. Secondary Master Disk CDROM, UDMA 2
6. Display Type EGA/VGA
7. Serial Ports 3F8, 2F8
8. Parallel Port 378
9. DRAM Type SDRAM
10. SPD On Module Yes
11. Data Integrity Check Non-ECC
12. PCI Devices IDE Controller, Serial Bus Controller.
13. Multimedia Device and Display Controller

Any System with Linux distribution such as UBUNTU, FEDORA, etc. can be used as a host system. The host system considered for this particular project is a HP laptop

Notable Host system details are as follows

1. Intel® Core™ 2 DUO P7350 @ 2.00 GHz
2. RAM 2.00 GB
3. Sound and Video Controllers
4. Universal Serial Bus Controllers
5. All other components that can be found in any modern laptop
6. System running UBUNTU 10.04.1 LTS Codename LUCID
7. Linux version used 2.6.32-38

There is a caveat to be considered while using SPEDE Environment setup for development using a laptop. SPEDE requires communication between the target and the host systems using

UART ports connected by a NULL-MODEM cable. However modern laptops do not have UART ports built in; hence we have to resort to other means to bridge this communication gap. USB ports can be used as UART ports using a USB Serial Cable, where the system's available USB ports can be converted to UART ports for communication with the target system. Apart from physical hardware conversion of the ports, there is some software tweaking to be done since SPEDE Environment requires some basic settings for reliable usage. Two important programs in SPEDE are FLAMES and FLASH, which run on target system and host system for downloading the compiled executable file of machine instructions into targets memory. In LINUX all the devices and ports are treated as files and they can be located at path /dev. One of the host serial ports that will be used by FLASH on LINUX needs to be /dev/ttyS0 and these are hardcoded; to change these parameters, FLASH code base has to be tweaked and recompiled, but we can resort to another simple technique which takes advantage of LINUX file system. All the USB ports are listed as /dev/ttyUSBx; since these ports can handle serial UART connections, they can be renamed as /dev/ttySx and can function in a similar way like a UART port. Hence we can change the port, that is connected to the USB Serial cable, referred as /dev/ttyUSB0 to /dev/ttyS0 and SPEDE setup can handle the communication as it would with an UART port. The following bash script can be run after the cable has been plugged to the port. This requires administrative privileges on host system.

```
#! /bin/bash
sudo cp /dev/ttyS0 /dev/ttyS0.bkup
sudo rm /dev/ttyS0
sudo ln -sf /dev/ttyUSB0 /dev/ttyS0
```

One down side is that once the cable has been reset or pulled over, the device names have to be reset for proper reliable functioning.

The hard disk on the target machine has to be setup with configuration so that it supports both the execution of the MSDOS FLAMES program and the required disk space for NanOS to store and retrieve data. Now the executable file can be downloaded using FLASH and FLAMES as it is performed in CSC 159 lab. For more details on the system setup “SPEDE-2000 Lab Manual” [5] can be referred and explanation on the details of SPEDE is outside the scope of this project.

SPEDE setup requires MSDOS to be run on the target system. Since the communication between target and host system needs FLASH and FLAMES to be run on host and target system respectively. Sometimes FLAMES can be run from a floppy drive on target system. For NanOS to use the hard disk connected to the IDE port of the target system, they need to be pre formatted for identification since only formatting control is give to the users (students/professors) and selecting the size of the drive is not provided. Hence the size of the drive usable by NanOS has to be set. For this purpose we use FDISK utility of MSDOS. The step-by-step procedure to set up the Hard Drive can be found in the User Guide. NanOS File System and Disk Device Driver can access newly created Extended partition. The total size of the hard disk used by NanOS in this project is 8033 Megabytes. The partition details are explained in the chapter "Disk Driver Overview".

CHAPTER 3

DISK DRIVER OVERVIEW

The Disk Driver Structure can be represented with the following block diagram.

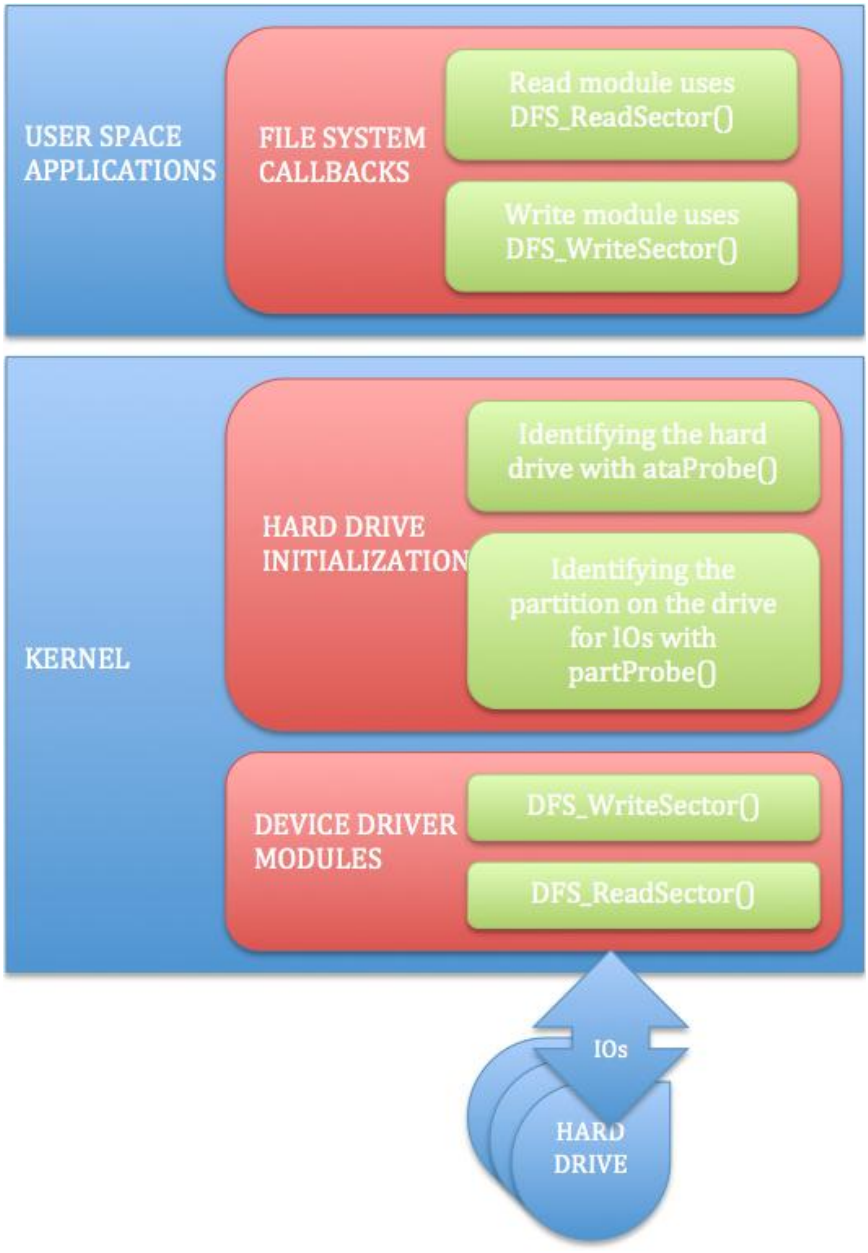

Figure 3.1
Disk Driver Overview

3.1. IDE Driver Details

Integrated Device Electronics (IDE) is one standard for connecting hard drives to PC. AT Attachment (ATA) is essentially the same as IDE. First, we will explore the hardware controller structure and the logic behind the IDE Hard Drives [6]. The block diagram in the following picture [11] explains the IDE Hard disk hardware details that are required for the background of this project. IDE hard disks have movable parts and immovable parts, where the immovable parts are controlled with read and write registers that can be programmed. Some of the data and interrupt signals are not shown in the diagram.

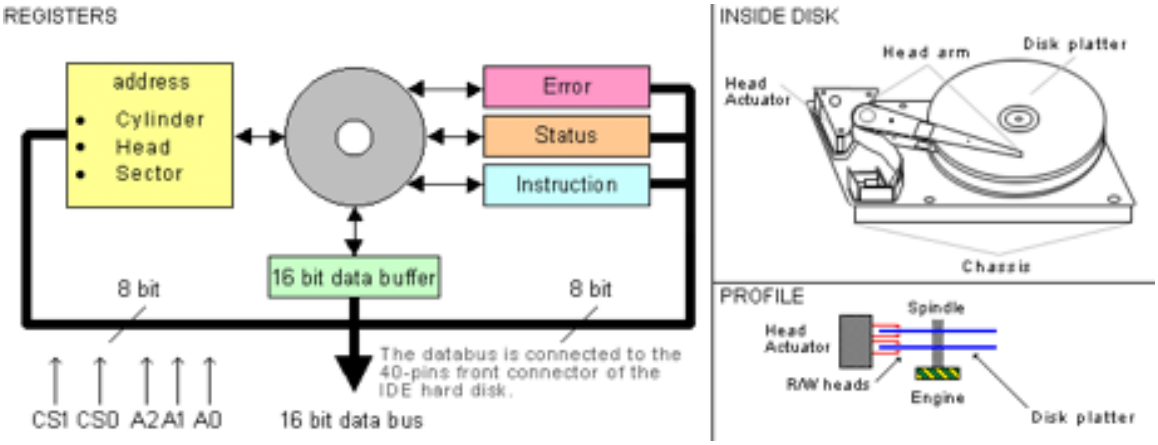

Figure 3.2
Hard Drive Organization [11]

The signals A0, A1, A2, /CS0, /CS1, /WR, /RD and /RESET are the outputs from the controller to the IDE bus. The signals IRQ and /ACT are outputs from the IDE bus to the controller; IRQ can be tri-stated by the IDE device when two devices are connected to the IDE bus; ACT can drive a LED. The signals D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, D14, D15 are bi-directional. These act as output from the controller to the IDE bus when writing, output from the IDE device to the controller when reading information.

There are five important modules in the IDE hardware, which should be considered for properly understanding IDE. They can be listed as Addressing, Data-Buffer Handling, Error Reporting, Status Reporting and Command Instructions. Most motherboards have built in Primary and Secondary IDE controllers. The target system used in this project has a Primary Master Disk and no Primary Slave Disk. These IDE controllers use the following standard I/O addresses:

1) Primary IDE controller: 1F0h to 1F7h and 3F6h to 3F7h
2) Secondary IDE controller: 170h to 177h and 376h to 377h

Each I/O address maps to a specific register on the IDE controller. The following is a list of each I/O address used by ATA controllers and the corresponding register. The I/O addresses are provided are for the Primary IDE controller.

1) 1F0 (Read and Write): Data Register
2) 1F1 (Read): Error Register
3) 1F1 (Write): Features Register
4) 1F2 (Read and Write): Sector Count Register
5) 1F3 (Read and Write): LBA Low Register
6) 1F4 (Read and Write): LBA Mid Register
7) 1F5 (Read and Write): LBA High Register
8) 1F6 (Read and Write): Drive/Head Register
9) 1F7 (Read): Status Register
10) 1F7 (Write): Command Register
11) 3F6 (Read): Alternate Status Register
12) 3F6 (Write): Device Control Register

Some of the details of the registers [7] and the bits used in the project are explained below.

1) The head and device write register helps setup of master/slave selection and the head number.
    a) bits 3 - 0, Head number
    b) bit 4, Master slave select
    c) bits 7 - 5, Bytes per sector coding. In IDE, only 512 bytes per sector are supported
2) Only 2 bits are used from the 8-bit Interrupt and Reset Register (a write register), which are tabulated below.
    a) bit 1, IRQ enable, when 1 disk will generate interrupts for command completion and do not generate interrupts when 0
    b) bit 2, Reset bit, a software reset for programmers control
3) The 8-bit Status Register is one of the most important read registers that give control on the IDE controller based on its current status. The following table explains the status register bits.
    a) bit 0, Error bit
    b) bit 1, Index Pulse, for each revolution of the disk, this bit pulses to one.
    c) bit 2, ECC bit, for this project ECC corrections are not enabled
    d) bit 3, DRQ bit, this bit implies data is either available to be read or written
    e) bit 4, SKC bit, seek has been executed successfully
    f) bit 5, WFT bit, indicates write error occurred
    g) bit 6, RDY bit, disk is ready and has finished its power up
    h) bit 7, BSY bit, disk is busy

4) Another important and necessary detail of any controller is their error description for better error handling. Error register, which is a read register, indicates what happened when a command failed. The error status is obtained from the Status register and the description can be read from this register. They are self-explanatory.
    a) bit 0, AMNT bit, Address Mark was Not Found
    b) bit 1, TK0NF, the track 0 is undetectable which implies the disk is not usable.
    c) bit 2, ABRT bit, the command has been aborted, probably because the command was illegal
    d) bit 3, MCR bit, Media Change Requested.
    e) bit 4, IDNF, Sector ID not found
    f) bit 5, MC bit, Media has been changed.
    g) bit 6, UNC bit, uncorrectable data error occurred
    h) bit 7, Reserved

The details and structure also correspond to the same secondary IDE controller addresses. They are like similar data structures at different places, which will have similar elements. Thus, for example, if the secondary IDE controller's data register is at 170h (primary IDE data register can be found at 1F0h), the secondary controller's error and features register is at 171h (primary controller’s error and feature register can be found at 1F1h) and so on.

Addressing IDE drives can be handled in two modes. They can be called LBA mode and CHS (Cylinder Head Sector) mode. In this project, LBA mode is considered for implementation, but a brief overview on CHS mode is also given. In CHS mode data on the drive is located using 3-register values cylinder, head and sector. The following diagram explains the detailed reference for all three components.

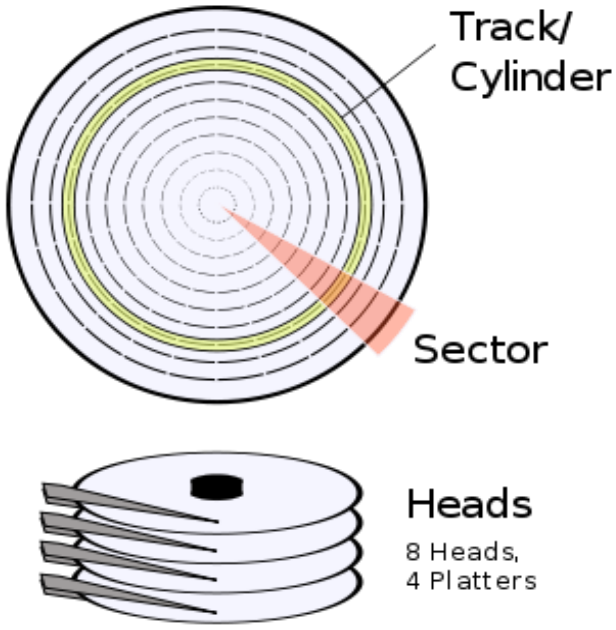

Figure 3.3
Heads, Sectors and Cylinders [11]

In IDE, each sector consists of 256 words, which is equivalent to 512 bytes. The sector is an 8-bit register that can be used to address these 256 values ranging from 0 to 255. The head address is also an 8-bit register. In a hard disk, each disk platter consists of 2 read/write heads.

The cylinder-track defines the data position based on the read/write head position on the disk platter. If the size of the hard disk is 6448 MB and it has 13328 cylinders, 15 heads and 63 sectors, then the mathematics as defined before would prove the equivalence.

13328*15*63*256 = 3224309760 words = 6448619520 bytes

The LBA mode is the simplest way of handling data from hard disks. This mode is a bit simpler to use. This mode also uses the head and cylinder, but also a register called sector number. Together these registers will define a 32-bit address, which point to the data block that needs to

be read or written. Each data block is still 256 words. The picture below shows how the register forms the 32-bit address. In LBA mode, numbers of sectors, cylinder or heads are not considered for identifying a required block. The defined address as a 32-bit number from 0 to total disk size can be used for IO operations. For example, a 6448MB disk will have LBA address from 0 to 12594960 (0 to C02F10 hex). This mode uses a 32-bit register that comprises all the before explained head, cylinder and sector register values that gives the programmer the ability to access data with one single address.

ATA commands are issued by writing commands to the command register. The pseudo code structure will be explained in the next section. All the commands are defined in the header files ata.h. ATAPI is an extension of ATA, which defines an interface for sending SCSI commands to the ATA devices. SCSI devices are storage devices that respond to predefined SCSI commands. A standard has been established for SCSI specification and most of the modern enterprise devices are SCSI devices or its successor SAS devices. ATAPI is specifically designed to control CD-ROM drives. ATAPI uses packet-based concept for communicating with drives (sending and receiving data and commands). We will concentrate more on ATA commands since for this project only Hard Disks are used and no CD-ROM access is implemented.

3.2. Device Driver Components

NanOS ATA/IDE device driver implementation is made in 2 files ata.c and ata.h. The header file defines structures that are required to carryout some of the basic commands and #defines that describe the hex values for the error, status and instructions as outlined in the ATA specification. Most of the #defines are self explanatory with comments. For a successful Disk IO operation or transaction following three data structures must be considered, they are listed in Appendix B.1 .

1) ATA or ATAPI command structure

2) Generalized drive info structure

3) Partition info

The Drive Command Structure consists of elements that define the ATA command details for a successful command request execution. This structure is considered in the simplest form, although many other elements could have been considered for performance and extensive details. Drive Specification Structure is used to store details of the hard disk such as the number of cylinders, heads and sectors, along with IO addresses as explained in the previous section. Partition details will be explained in detail in the File system structure. Another important structure that maintains the identify information received from the hard disk when performing an identify command is ATA Identify Structure. This structure includes vendor specific data with capability (DMA mode or LBA mode) and capacity details. They are listed in Appendix B.2 .
The Instruction command register can be used to write the Drive Command Bytes as shown in Appendix B.3 for required Disk IO Operation.

A variable hd_intr_occurd, which is defined as a global integer, keeps track of the interrupt status of the hard drive, since it is required in other files such as ata_process.c, which is an interface for File system. Module awaitInterrupt waits with a timeout value given as input in microseconds until interrupt(s) given by another input bit-mask IRQMask occur. If the controller is busy with another operation then this value cannot be retrieved. This function returns a non-zero mask value if interrupt occurred, any other scenario a zero is returned on timeout. Modules ataProbe and ataSelect that takes IO address are used to check for the existence of the hard disk and select the hard disk if more than one hard disk is present. In this project we used only one hard disk.

Another important module ataCmd that takes Drive Command Structure as input, is used to perform read/write operation on the specific drive. This function returns 0 if the command is successful, and the following values for any other scenario

1) 1, if drive could not be selected
2) 2, if unsupported command
3) 3, if command timed out
4) 4, if bad/questionable drive status after command
5) 5, if writing to read-only drive.

Module partProb probes for partition information necessary for any file system mounting. This is just a sanity check performed during Device driver initialization. The partition table handling is done in the File System level.

3.3. Device Driver Interface

This section explains the files ata_process.c and ata_process.h, which forms an interface for any File System to perform basic reads and writes. The following two modules are the simplest modules to perform reads and writes of any available sector on the hard drive.

1) DFS_ReadSector that takes a buffer pointer, start sector, number of sectors and a unit number specifying the drive (if multiple drive exists) and reads the data into the buffer provided.
2) DFS_WriteSector that takes a buffer pointer, start sector, number of sectors and a unit number specifying the drive (if multiple drive exists) and writes the data into the sectors provided.

Both the above modules are similar in functionality except for the flow of data between memory (RAM) and drive (external memory). The algorithm is explained below.

1) Input buffer to store data on read or provide data on write, a sector value to read from or write into, number of sectors, device unit
2) Form a Drive Command structure for read/write operation.
3) Loop sector values from 0 to given input limit
4) Perform read/write operation using ataCmd
5) If the output of the above command is successful, then move data either from input buffer on write or to input buffer on read. If the output fails, return non-zero values.
6) Loop through step 4

CHAPTER 4

FILE SYSTEM OVERVIEW

The File Allocation Table (FAT) [9] file system is a simple file system originally designed for small disks and simple folder structures. The FAT file system is named for its method of organization, the file allocation table, which resides at the beginning of the volume. To protect the volume, two copies of the table are kept, in case one becomes damaged. In addition, the file allocation tables and the root folder must be stored in a fixed location so that the files needed to start the system can be correctly located. A volume formatted with the FAT file system is allocated in clusters. The default cluster size is determined by the size of the volume. For the FAT file system, the cluster number must fit in 16 bits and must be a power of two. The figure below illustrates how the overall data organization on the disk with boot structures and FAT File System.

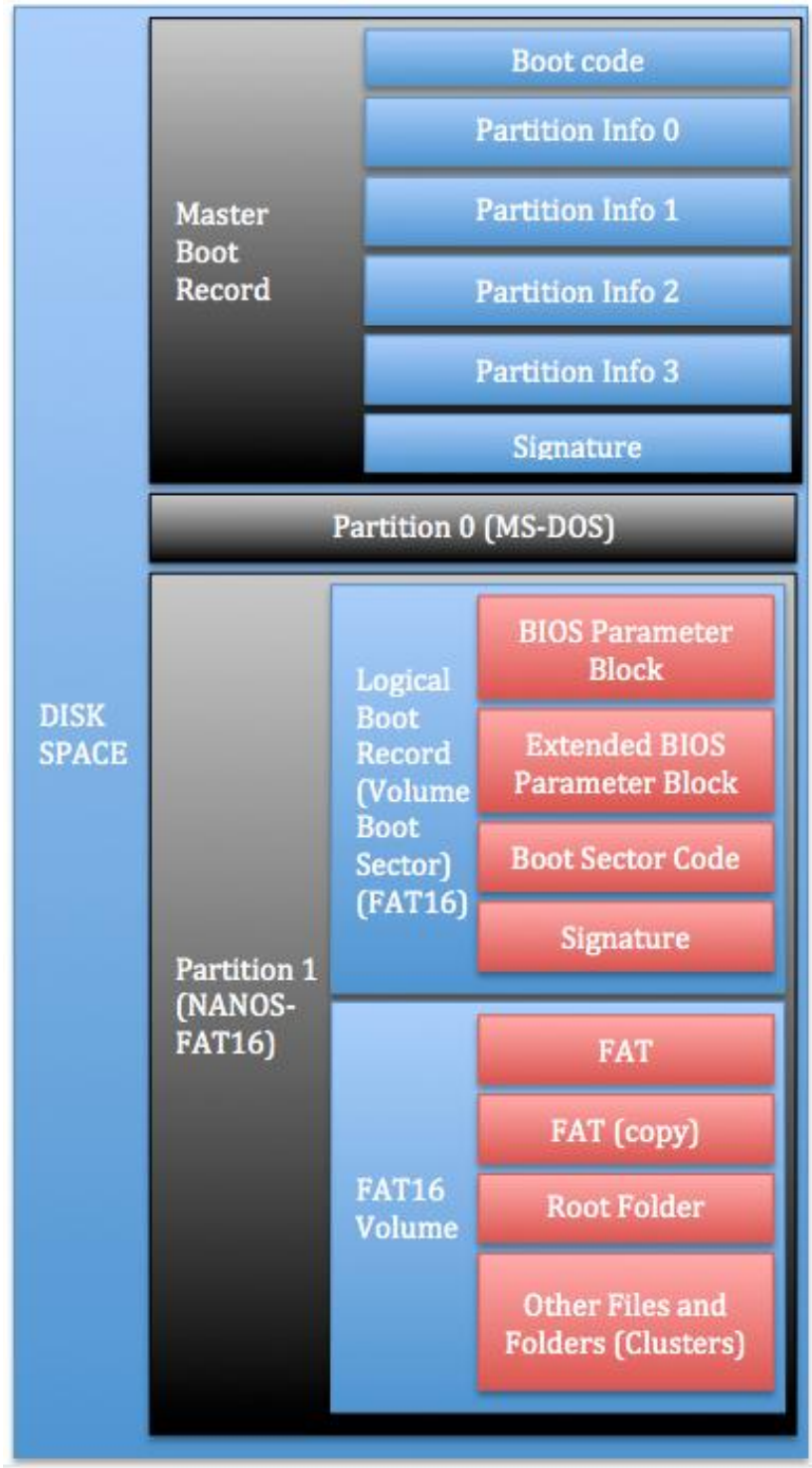

Figure 4.1
Disk Data Arrangement

4.1. FAT 16 File System Details

The Partition Boot Sector contains information that the file system uses to access the volume. On x86-based computers, the Master Boot Record uses the Partition Boot Sector on the system partition to load the operating system kernel files. Master Boot Record (MBR) is the traditional way of storing partition details in the boot sector of the hard drive, popularized by IBM PC. The very first sector of a hard drive contains an MBR Bootstrap program and a partition table. A Partition Table helps in the logical division of the hard disk and supports the notion of multiple Operating System and File System on the same hard disk. The following describes the structure of the Master Boot Record and Partition Table [9].

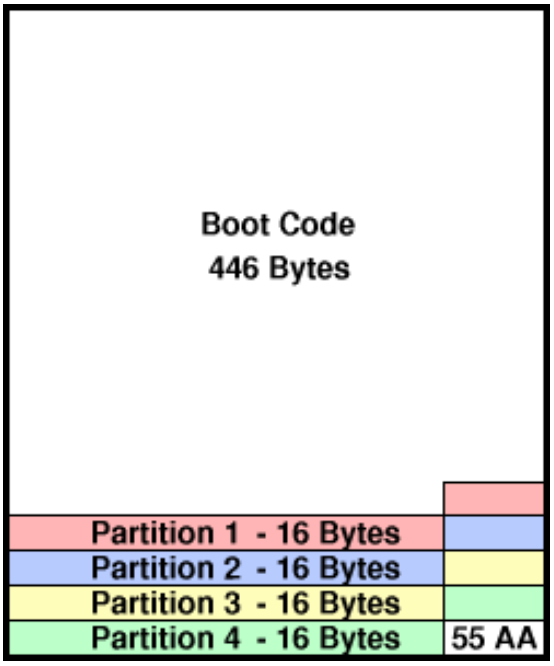

Figure 4.2
MBR Layout

The code area covers 440 bytes from the offset and there is another optional field of 4 bytes, which is used to identify the hard disk uniquely. The other 2 bytes are usually NULL or 0x0000.

The following four entries are used to maintain information regarding the Partition table starting address and each entry is 16 bytes long. One important factor to note is that these entries are not aligned on dword boundaries. The entries are not required to be in any kind of order. At least one of the partitions must be marked active. Usually the primary partition is marked as the partition that has MSDOS on it (refer Appendix A.1). Boot Signature or MBR signature marks the end of the MBR information. This usually consists of 0x55 at offset 0x00FE and 0xAA at offset 0x00FF. The partition table details give the whole structure of the partition on the hard disk with information such as Starting and Ending Head, Cylinder and Sector details. There are also 4 bytes of value that give the starting sector of the hard disk, and another 4 bytes that count the number of sectors included in the partition. The partition table structure is shown below.

Figure 4.3
16 Byte Partition Table Entry

For this project, the user has to find out which partition is usable for the NanOS File System. This value have been hard coded which can be modified using the define NANOS_PARTITION in the file ata_process.h. The MBR formats are becoming obsolete because of the limitation in the 32 bits design of LBA fields, which might overflow when dealing with a hard drive with a capacity greater than 2 Terabyte. Hence, new MBR design, GUID Partition Table (GPT), with 48 bit LBA fields has been introduced. This updated partition table standard is more complex in design. To handle all the sectors, cylinders and heads, there are many on-disk structure formats used which are named as File System. For this project, all the blocks are 512 byte sectors. In FAT File Systems, these units are grouped together to form an

allocation unit. Allocation unit forms the unit of storage in FAT File System. For simplicity, consider that each allocation unit comprises one sector, which means these terms can be interchanged for explanation purpose of the File System Layout.

The basic layout of FAT 16 File System is shown in Figure 4.1. The starting block should be considered starting at block zero of the partition on which the File System is to be loaded. The area can be divided into 4 regions:

1) Boot Block (1 block)
2) File Allocation Table (Depends on File System Size)
3) Disk Root Directory (Variable – selected when disk is formatted)
4) File Data Area (The rest of the disk)

4.2. File System Components

The boot block is used to load the operating system to memory by running the special bootstrap program. But in the course CSC 159, the Operating System is not loaded using this method. The data structures from dosfs.h describe the Master Boot Record and Partition table entries are shown in Fig 4.1, Fig 4.2 and Fig 4.3 (listed in Appendix B.4 and Appendix B.5).

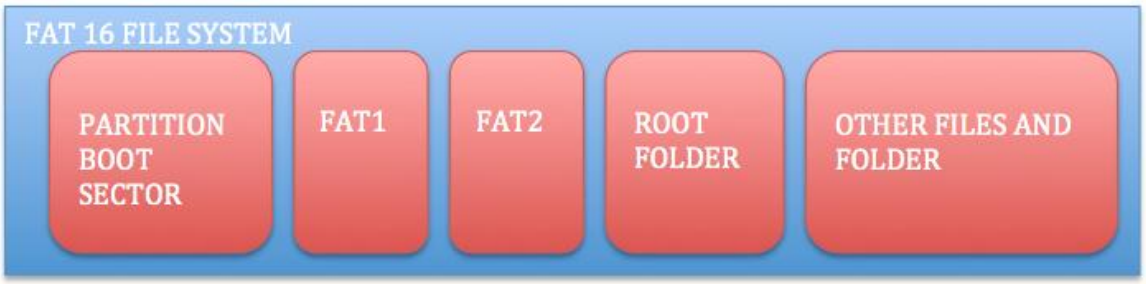

Figure 4.4
FAT16 File System Structure

Modules defined in dosfs.c and dosfs.h handle FAT File System Initialization and Partition detection and allocation. File System Structure Management and Maintenance are the

core functions of this section. These structures are mainly used to control the formatting, reading, writing and modifying the details of an existing File System, or to create a new File System from scratch on the disk. The Boot Block in FAT File System also holds several important data areas, which help to describe the File System. The block diagram in Fig 4.1 can be considered for better understanding of these modularized structures along with data structures Logical Boot Record structure (volume boot sector), BIOS Parameter Block structure (FAT12/16), Extended BIOS Parameter Block structure (FAT12/16) (shown in Appendix B.6, B.7, B.8).

The files and directories management of the file System is accomplished using FullFAT. The core of the working FAT16 File System is adapted from an open source project named FullFAT written by James Walmsley. FullFAT File System [10] has flexible Open Source Licensing for usage in student projects. The structures mentioned above are also maintained in FullFAT. They are not explained here to avoid recurrence. FullFAT has online documentation [12]. The other structures such as File Allocation Table, Directories and Files are explained for better understanding on the working FAT 16 File System. All the files used in this project from FullFAT are prefixed with "ff_". Some of the structures in FullFAT are listed as examples for exploring the critical component of the FAT File System.

File Allocation Table is responsible for the classification of FAT into FAT-12, FAT-16 and FAT-32. Here we will be considering the FAT-16 File System. These occupy one or more blocks at the start, immediately followed by the boot block as shown in Fig 4.4 . If there is more than one FAT table, these tables will be placed one after another. FAT-16 get its name from the size of the entry in the FAT table, here each entry is 2 bytes long (16 bits). The disk data area is divided into clusters, which is the same as an allocation unit except they start from in the data area and not the start of the partition. Hence,

Cluster number = Allocation unit – A constant

A constant denotes the space between the start of the disk and the start of the data area. For each entry in a FAT-16 table, there is a cluster assigned. Entry N relates to cluster N. Clusters are numbered starting from 2, since the first byte is a copy of the media descriptor which is the magic byte used to classify disks; the second byte is set to 0xFF. Each cluster contains a successor cluster number, the number of the cluster that follows the current cluster. The last cluster in a (cluster list) file will have an entry 0xFFFF. Files ff_fatdef.h, ff_fat.c, ff_fat.h defines all the necessary variables for the FAT File System to be laid out on the hard disk.

The root directory contains an entry for each file whose name appears at the top of the root level of the file system. Other directories can also be listed under root directory and they are usually mentioned as subdirectories. The main difference between root directory and sub directories is that root directory occupies statically allocated space on the disk when it is formatted, and has an upper limit on the number of files and directories it can hold. The structure of a file and a directory are similar and is 32 bytes in size. A block (512 bytes) can contain 16 entries. The following structure in FullFAT explains the structure of an entry. Files ff_dir.c, ff_dir.h, ff_file.c, ff_file.h contain all the modules necessary for creation, deletion, and modification of files and directories. A directory entry is usually mentioned as a dirent. The Data Structure for a DIRENT in FullFAT is shown in Appendix B.9 . Filenames “FileName” in the 'dirent' structure are only 11 bytes long in FAT-16, including the extension, which are 3 bytes. A Long file name is not supported in the project. The first byte of the file name indicates its status. The following list enumerates the important file status descriptions.

1) 0x00 Filename is never used
2) 0xE5 The file name has been used but its deleted
3) 0x05 The first character is actually 0xE5
4) 0x2E This is a directory entry, not a normal file

5) Any Other Bits This is the first character of a real filename.

The dot separator between the file name and extension is implied and is not stored anywhere. If the extension is less than 3 characters, they are padded with special characters. There are 8 file attributes to be considered, and the 1-byte element “Attrib” provides information regarding the properties and the permissions of the dirent. Each bit corresponds to unique information and they are combined to give all the details about the file. They are listed as follows.

1) 0x01 indicates that the file is read only.
2) 0x02 indicates a hidden file. Such files can be displayed if it is really required.
3) 0x04 indicates a system file. These are hidden as well.
4) 0x08 indicates a special entry containing the disk's volume label, instead of describing a file. This kind of entry appears only in the root directory.
5) 0x10 the entry describes a subdirectory.
6) 0x20 this is the archive flag. This can be set and cleared by the programmer or user, but is always set when the file is modified. Backup programs use it.
7) 0x40 Reserved; must be set to 0.
8) 0x80 Reserved; must be set to 0.

File Date and Time are handled by structure FF_SYSTEMTIME as shown in Appendix B.10. FullFAT source code contains some more files that are not required by this project, but they still remain in the source directory to keep the project intact, such as ff_md5.c, ff_md5.h, ff_string.c, ff_string.h, ff_safety.h, and ff_safety.c. File ff_types.h describes all the type casted data types used specifically for FullFAT. Files ff_memory.c, ff_memory.h provided ability to deal with short and long integers in Big Endian and Little Endian formats. The most important component of the FullFAT File System resides in files ff_ioman.c, ff_ioman.h. These files provide I/O Manager for handling IO buffers for FullFAT safely. They provide a simple static interface to the

rest of FullFAT to manage buffers. It also defines the public interfaces for creating and destroying a FullFAT IO object. The structures to define partition tables for FullFAT and handling the IO Buffers are listed in Appendix B.11 and B.12. This can be modified for better performance.

Appendix B.13 lists out interfaces that are used to handle creation and deletion of I/O Manager, registering and un-registering block devices, mounting and un-mounting partitions. I/O Manager modules are responsible for detecting partition, FAT16 boot block, and setting up the whole environment for file management modules. The File System Process in NanOS uses these modules for maintaining and managing the files and directories, as the user wants. I/O manager is unique for each device and it has mechanisms like registering and mounting which help maintain FullFAT in several hard disks at the same time.

Various error codes are pre defined in files ff_error.c, ff_error.h for displaying error information captured during the process of file management.

4.3. File System Interface

I/O manager maintains the environment required for full functioning of FullFAT. But FullFAT has the function interfacing structure shown in Appendix B.14 to describe the block device driver interface found in ff_ioman.h, which completes the connections between the FullFAT and ATA Device Driver covered in chapter 3. Each I/O manager can be used on only one device, hence for different devices separate I/O managers must be maintained. This is achieved by registering and un-registering the drivers created for accessing the specific devices. In this project only one device is used, therefore one I/O manger is used. Registering a device driver with FullFAT must adhere to the specification provided by modules F_WRITE_BLOCKS and FF_READ_BLOCKS listed in Appendix B.15. Then these above modules are used all over the FullFAT system for accessing the hard disk information without any hassles while handling

the interrupts and registers. Since FullFAT maintains its own partition and FAT information structures, the above modules are used to load the structures FF_PARTITION shown in Appendix B.16, which has similar elements as maintained in LBR or *PLBR structures described in chapter 3 for device drivers.

4.4. File System Tools

This sections gives insight into how the command line options are handled and the processes communicate between each other to accomplish file management tasks and file download tasks. Initialization (init) Process creates all the processes and feeds them with information so that communication can be established between the newly spawned processes. This is accomplished using message passing. Message passing between the processes is accomplished using pre-known process ids of the communicating processes. For this reason, Init process spawns the new processes in a certain order and then sends a message to the process, which requires messages from its communication process. The processes gets spawned in the following order

1) Process for inputs from Standard Input from Console (StdConsoleIn).
2) Process for sending data to Standard Output (StdConsoleOut).
3) Command Shell process for handling user input commands (Shell).
4) File System process that handles user input commands for File Management & Manipulation (fs_process).
5) File Download process that handles file download commands (fdl_process).

The following messages are exchanged between Initialization (init) process and other above listed processes:

1) Process id of StdConsoleIn, StdConsoleOut, fs_process and fdl_process are sent to Shell.

2) Process id of Shell is sent to StdConsoleIn, StdConsoleOut , fs_process and fdl_process
3) Process id of fdl_process is sent to fs_process
4) Process id of fs_process is sent to fdl_process

After the exchange of the above messages the core function of the processes gets initiated, before that they are blocked by a message passing mechanism (Refer Appendix B.18). The processes implementation can be found at proc.c and proc.h. Lets explore the Shell and File System processes used in this project. The following block diagram shows the spawning process at initialization and message passing interaction between the processes.

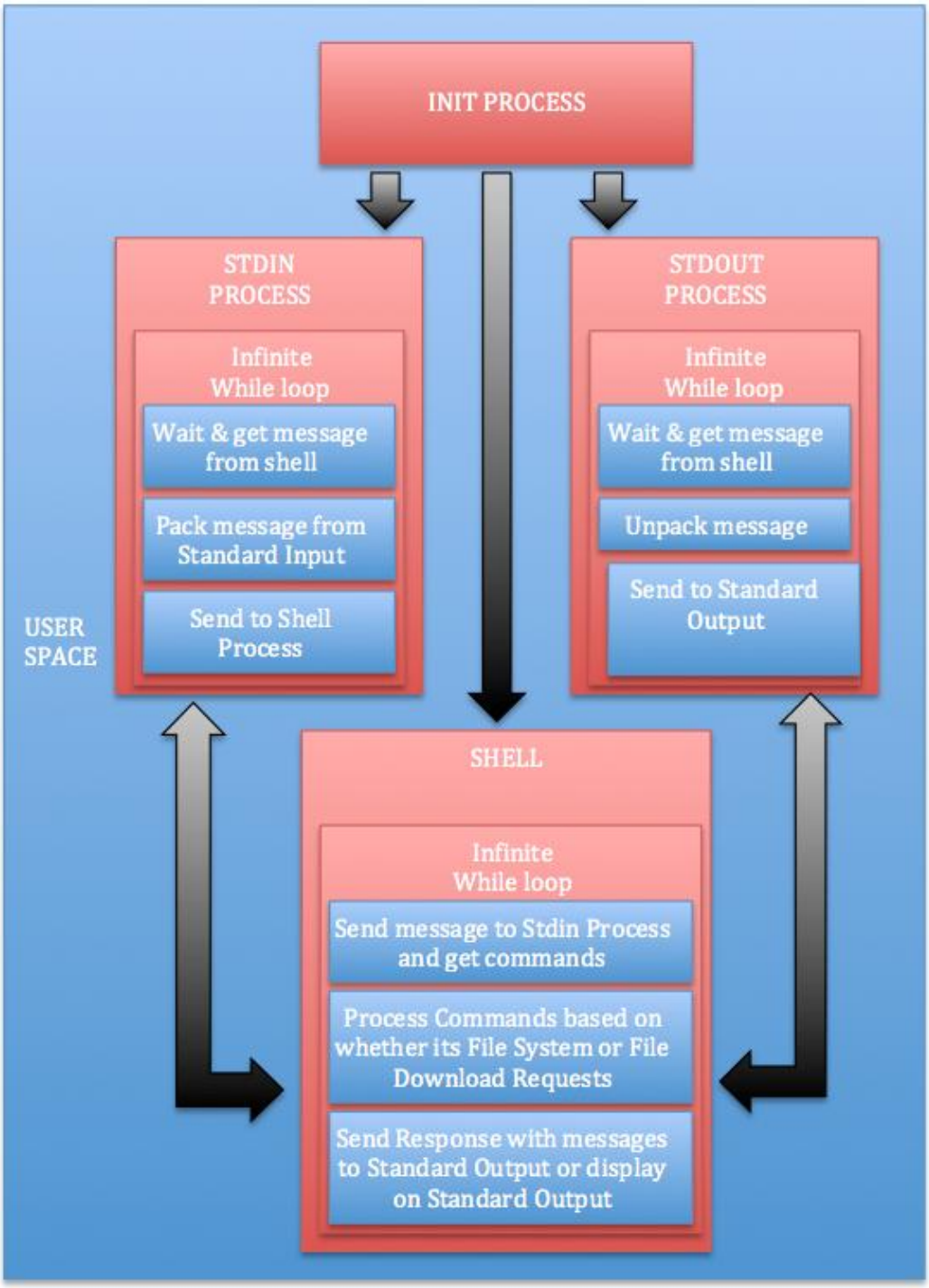

Figure 4.5
Process Spawning at Initialization

Command Shell process is one of the processes that get spawned at the start of the NanOS Operating System as shown in Fig 4.5. Shell process that is designed in NanOS deals with communication between two terminals. These terminals are connected to the UART port of the target system. The Shell process handles the interrupts of the 2 ports and creates a simulated chat session where messages between the terminals are exchanged. These are accomplished using UART Interrupt Driven Device Drivers with semaphores and a message passing mechanism. For this project, Shell process will handle the input and output to the target monitor in the simplest way. There are three processes involved in the full functioning of the shell. They are the modules StdConsoleIn, StdConsoleOut and Shell present in proc.c. These modules are spawned using the pre existing spawn kernel function, which creates a process and runs on the target system. The communications between the processes are handled using message passing interface implemented in CSC 159 course work. These processes are simple, infinite while loops with proper inter process communication established using message-passing interface. Lets explore each of the processes. The process names are same as module names.

StdConsoleIn process gets spawned from the Init process. It waits for the message from the Init process from which it can acquire the Shell process id. It enters an infinite while loop which waits for a request from a command Shell process. When the request comes in, it processes constantly for input characters from the keyboard and stores them in a string buffer whose pointer is stored in an element inside the message structure. The message is sent as a response to the request and again waits for a request from the command shell. The input function used, collect characters from the keyboard, is cons_getchar() (taken from spede library). Another option of echoing is provided which decides whether the display of characters typed is allowed or not. The pseudo code is shown below.

1. Get message from (init) Initialization process and store the shell process id
2. Infinite while loop
    a. Echo mode on by default.
    b. Wait for request from shell to get collect input from keyboard.
    c. Check for echo mode from the message received.
    d. Collect characters till Enter key is encountered.
    e. Pack the message and send it to Shell process.

StdConsoleOut module gets spawned from the Init process and its structure is very similar to the StdConsoleIn process. It waits for the message from the Init process from which it can acquire the shell process id. Then it enters an infinite while loop, which waits for a request from command shell process. When the request comes in, it obtains the string from the buffer pointer in the message and displays them on the screen using cons_putchar() (taken from spede library). It then sends a dummy response to complete the task, and starts waiting on the request from shell process. Before the processing of commands, the process is synchronized so that it can move further only after the file system has been loaded. This is achieved by making the process waiting for a message from fs_process (this can be achieved by using semaphores too). The pseudo code is shown below.

1. Get message from (init) Initialization process and store the StdConsoleIn, StdConsoleOut, fs_process and fdl_process ids.
2. Infinite while loop
    a. Maintain command prompt “=>”.
    b. Send a request to collect input from user through keyboard with echo mode on.
    c. Receive a response holding the command line data received.

   d. If the commands received from messages are for file system, send message to fs_process and wait for response from the process.
   e. If the commands received from messages are for file download, send message to fdl_process and wait for response from the process.
   f. If they messages are not identifiable, "Not an Internal or External Function" is printed.
   g. Collect characters from the message and display it on the screen.
   h. Pack the message with dummy values and send it to Shell process.

Command Shell process uses the above 2 processes for display functions, and handles the data for displaying and collecting for the other 2 core processes fs_process and fdl_process. When the shell process received commands from the processes fs_process and fdl_process, they are differentiated using another module consoleCommandSet in ff_command.c, which identifies what commands are defined for the processes fs_process and fdl_process, and then forwards the information to the appropriate processes for execution. The pseudo code for shell process is shown below.

1. Get message from (init) Initialization process and store the shell process id.
2. Get message from file system process after successful file system initialization.
3. Infinite while loop
   a. Send request to StdConsoleIn process to collect data keyboard with echo mode on.
   b. Collect characters from the message's string buffer and identify the type of command using the module consoleCommandSet.

c. If the commands received from messages are for file system, send request to file system process and obtain response.

d. If the commands received from messages are for file download, send request to file download process and obtain response.

e. If they messages are not identifiable, "Not an Internal or External Function" is printed.

File System Process is the core component of the whole project, and maintains the file system modules and services the requests from the user via command shell. This process receives the process ids of Shell process and File Download process from Init process. As explained in the previous section, both the processes wait for a message from the File System process. This message will be sent only after the file system has been set and mounted along with the structure for handling file data input and output. The above actions are performed using a module named fs_init(), which returns a FF_IOMAN structure used for the complete file system operations. Another module filesystem_consoleSet() is used to initialize all the shell commands. Following these steps, the process goes into an infinite request-response loop serving file system requests such as create file, delete file and many other operations. Lets look into the file system process pseudo code.

1. Initialize FF_IOMAN and FFT_CONSOLE_SET.
2. Wait until a message is sent from Init process and obtain the Shell & File Download process ids.
3. If File system is initialized successfully and IOMAN object is received, follow the steps
    a. Set command line parameters using filesystem_consoleSet.
    b. Send a message to shell and file download process so that they can start sending requests.

c. Infinite while loop

   i. Receive a message, either from Shell or File Download processes.

   ii. Process the string from the message using FFTerm_ExecCommand defined in ff_term.c (part of FullFAT).

   iii. Send response back to the process that issued the request.

4. If the File system is not initialized, then display “File System Initialization failed”.

File System Initialization (fs_init module) is one of the important modules, which helps the FullFAT system to initialize the file system. fs_init module is found in file ata_process.c. This process defines various FullFAT variables for handling FullFAT buffers, Input/output Manager, Block Device Drivers, Partitions and files system Environment structures. Initialization includes the following three steps as explained in chapter 4.2.

1. Creating of Input/output manager
2. Registering Block Device driver
3. Mounting the Partition on the hard disk

FullFAT comes with built in APIs for managing a set of shell commands for file system management requests. These are incorporated and added to the shell command analysis using the module consoleCommandSet in proc.c file. The file system commands are added to the set using the function FFTerm_AddExCmd in FullFAT that does the string parsing, identifying, and executing of appropriate modules.

CHAPTER 5

FILE DOWNLOAD OVERVIEW

This is another core part of the project, which deals with downloading files and folders from the host system to the target system. The user can run a script on host machine with required files in specific folders as set in the configuration file and commands “dwnldC” (Configure Download) & “dwnldS” (Start Download) on target machine can be used to complete the process of download. The target side function is handled by fdl_process module in the file proc.c. This process is similar to a request-response model where it serves a request from a shell process. The following pseudo code describes the structure of the file download process.

1. Wait until a message is sent from Initialization (init) process and obtain the shell & file system process ids.
2. Get message from file system process after successful file system initialization.
3. Variable nanosdl_config_check set to 0, which is used to track the order of execution of file transfer, config file followed by data files.
4. If File Download process is initialized successfully, (Infinite while loop)
    a. Receive a message from shell.
    b. Process the string from the message using basic string compare.
        i. If it is ‘dwnldC’, check for nanosdl_config_check status and initiate transfer of config file from host to target, and the system is ready to receive files and folders from host.
        ii. If it is ‘dwnldS’, check for nanosdl_config_check status and initiate transfer of data files from host to target, and the system is reset after the completion of the module.
    c. Send a status message back to shell process.

5. If File Download process does not initialize properly, then display “File Download Initialization failed”.

The following block diagram displays the state machine that the file download process comprises. These state machines are managed using simple variable checking under “IF” conditional statements.

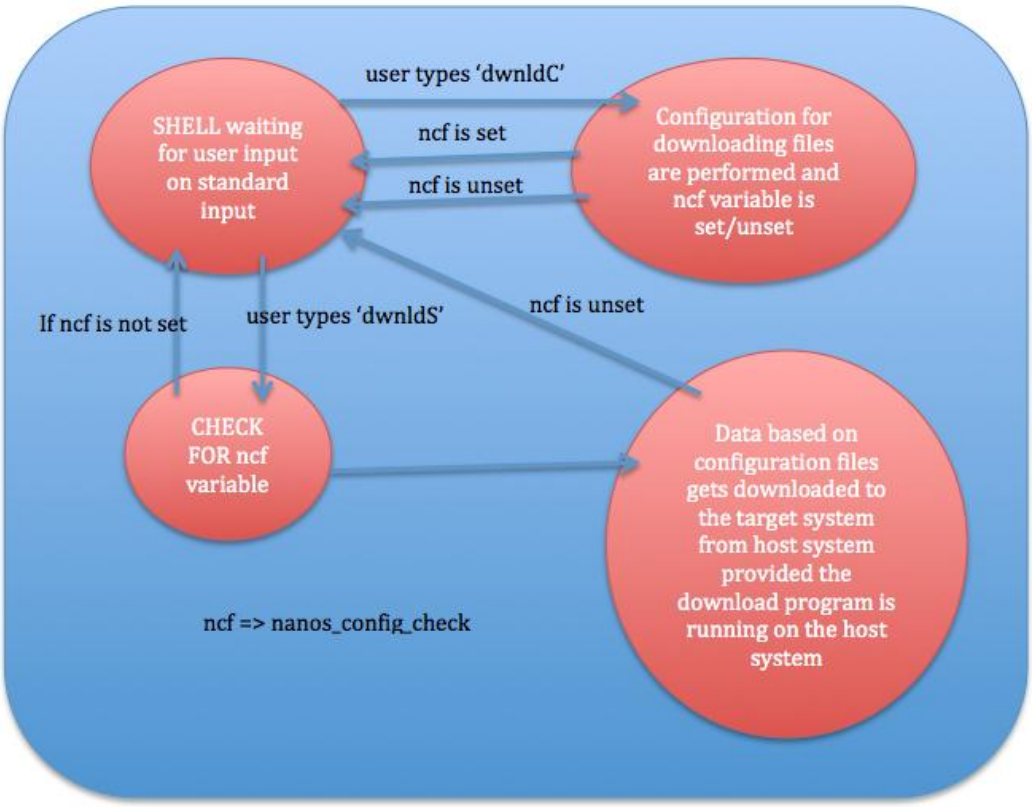

Figure 5.1
File Download State Transition

The flag nanosdl_config_check is used to check the state of the file download to verify whether it has finished config phase or not. Only after the completion of config phase can the data phase begin, since some structures get set in the NanOS system. The fdl_init module in file file_transfer.c initializes the data structures and resets the variables used to assist the file

download process. These structures are described in section 5.2. These initializations include the setting up UART terminals speed and other serial communication details as discussed before. The 2 important command modules mentioned above execute respective modules to accomplish the task. The modules are config_transfer() and file_transfer() . These modules are responsible for the transfer of config file and the data files (described in section 5.2). These modules in turn use the transfer() function that accomplishes the serial transfer of data using kermit protocol as explained in the section 5.2 and 5.3. After the successful completion of the module “config_transfer”, the module “file_transfer” gets executed. Execution of file_transfer before the config_transfer cannot be done and this state is maintained by the variable nanosdl_config_check. The help messages from the command line executions will display proper usage of the commands. The usage can be well understood from the User Guide (Appendix A.2)

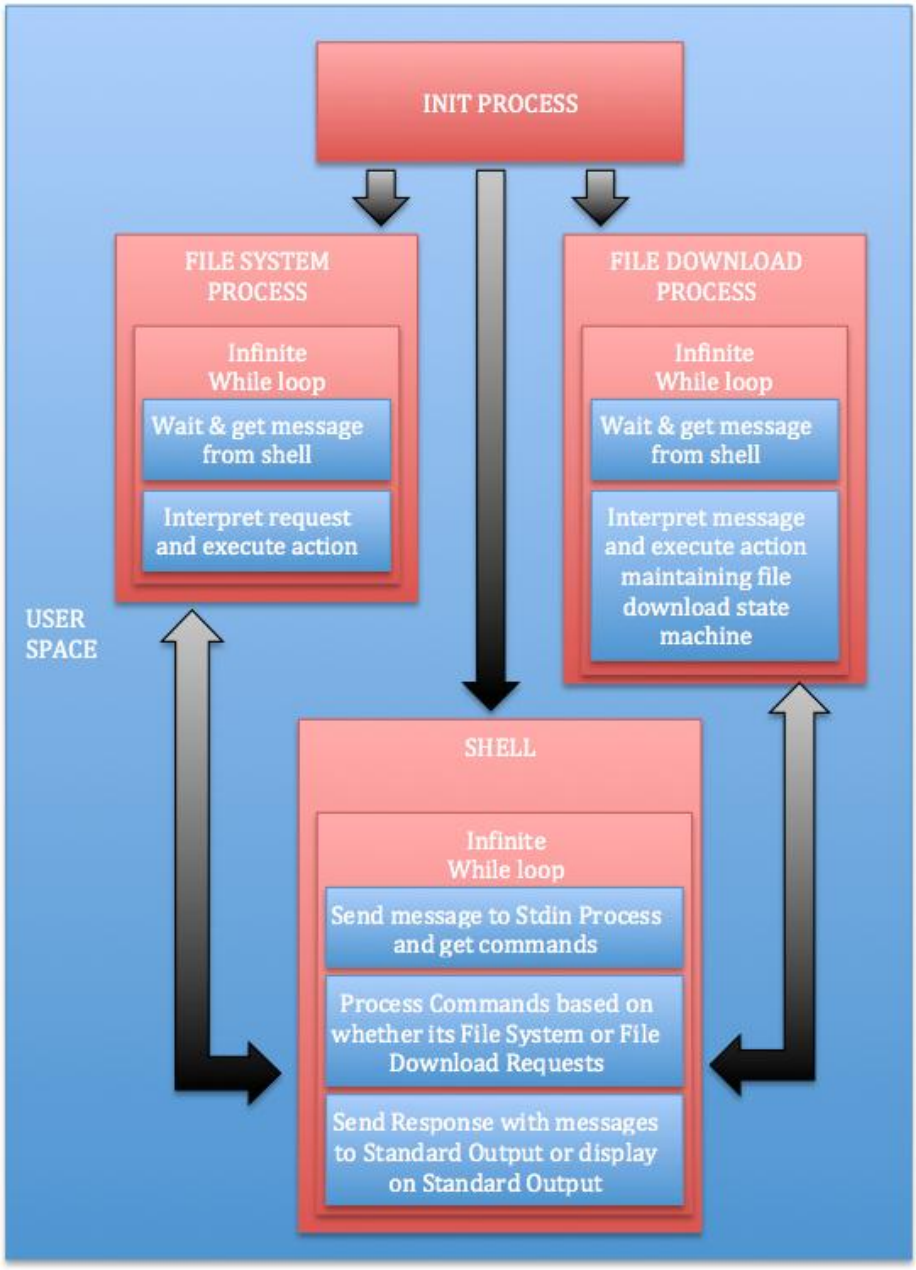

Figure 5.2
Processes Communication with Shell Process

5.1. Kermit Protocol Details

This part of the project provides the ability for NanOS to download files from the host system to the hard disk, so that they can be accessed by the file system mounted on the hard disk by NanOS. This project uses another UART port apart from the UART port used for downloading and debugging NanOS from the host system. This is accomplished using Kermit protocol [11]. Kermit is the name of a file-transfer and -management protocol, and a suite of computer programs for many types of computers that implements that protocol as well as other communication functions ranging from terminal emulation to automation of communications tasks through a high-level cross-platform scripting language. The software is transport-independent, operating over TCP/IP connections in traditional clear-text mode or secured by SSH, SSL/TLS, or Kerberos IV or V, as well as over serial-port connections, modems, and other communication methods (X.25, DEC net, various LAN protocols such as NETBIOS and LAT, parallel ports, etc., on particular platforms).

The Kermit Project was founded at the Columbia University Computer Center (now CUIT) for establishing a standard protocol for serial communication. There are a variety of Kermit protocol flavors available. For this project, an E-Kermit protocol is considered that has been published with an Open Source license. EK (Embedded Kermit, E-Kermit) is an implementation of the Kermit file transfer protocol written in ANSI C and designed for embedding in devices or firmware, and is used in real-time applications. EK performs two functions: sending files and receiving files. It is compact, portable, and fully reentrant. On Intel (CISC) it's about 15K; reducing buffer sizes and eliminating optional or unwanted features can achieve smaller sizes. EK does not include client/server functions, a command or script programming language, character-set conversion, transport encryption, or any form of communications or file input/output. It does not dial modems, make neither connections, nor does

it have a built-in TCP/IP stack or interface to an external one. If you need these features, then you should use a full Kermit program, such as C-Kermit or Kermit 95.

EK is not an application itself; it's a subroutine to be called from your master application. It is useful only when supplied with the master application or calling environment as well as the file and communications I/O routines. The calling environment must, in turn, make and configure the communications connection if one is required and not already open. EK includes the following Kermit Protocol features:

1. Long packets
2. Sliding windows with Go-Back-to-N error recovery (true selective repeat in EKSW).
3. Repeat-count compression
4. Control-character prefixing and un-prefixing
5. 8th-bit prefixing (for transferring 8-bit data on 7-bit links) (= parity)
6. Attribute packets (type, size, and date)
7. Sending and receiving single or multiple files.
8. Automatic per-file text/binary mode switching.
9. All three block check types (6- and 12-bit checksum, 16-bit CRC).
10. Status reports (protocol state, file name, size, timestamp, bytes so far).
11. Transfer cancellation by either party.

The following Kermit Protocol features are not implemented:

1. Sliding windows with selective retransmission
2. Character sets
3. Locking shifts
4. Client/server

Timeouts would be the responsibility of the Kermit program on both ends of the connection. This project consists of a NanOS process running Kermit protocol that can be executed using NanOS terminal. It can communicate with another program on the host machine running the same version of Kermit protocol. There were timeout issues in the open source code on the host side that needed some hacking based on the UART driver design on the target side. They will be explored in the next section.

5.2. File Download Components

Embedding E-Kermit protocol source code package in NanOS is accomplished by addition of 6 files. Core Kermit protocol is handled in files kermit.c and kermit.h. The I/O for kermit protocol is handles in kermitio.c and kermitio.h. The initialization and portability with NanOS is handled in file_transfer.c and file_transfer.h. The package used on the host side uses all the original file names that come with the open source package. They have usage manual, which will help users to execute them on Linux terminal with proper command line arguments. Kermit protocol documentation can be found online [3] which has extensive explanation on the details of the sliding window method, file transferring ability and the format of packets used.

Lets explore the host side package changes, command line options, scripts and configuration settings. NanOS project consist of a folder named “nanos_dl” which has the following folders and files.

1. config (no extension)
2. nanos_dl.sh
3. nanostxfr3n6 (no extension)
4. Folder named “download”
5. Folder named “eksw094_s”

Folder “download” contains all the files that are to be downloaded to the target system. This folder can have sub directories that can be transferred to the target system too. Folder “eksw094_s” contain files that are necessary for compiling the source code of e-kermit protocol into an executable that can initiate and manage file transfer between the systems. There are source files named as unixio.c and unixio.h, which are responsible for accessing the device driver on the system for protocol communication. These files by default contain modules that are written for Unix-to-Unix systems communication. These files have to be tweaked for reliable communication with NanOS since timing difference in bit transfers has to be handles. Binary mode transfer accomplishes the communication. UART ports are used with baud rate 9600, 8 data bits, no parity check and 1 stop-bit. The same setting is set in the NanOS UART handling device drivers. There are three modules in unixio.h that are important to explore for the project.

1. Module ‘devopen’ (Appendix B.18), this routine should get the current device settings and save them. It should open the device. If the device is a serial port, this module set the speed, stop-bits, flow control, etc. It returns 0 on failure, 1 on success. Refer Appendix B.19 for modified code.
2. Module ‘readpkt’ (Appendix B.20), this routine reads data of length len from UART port read buffer and transfers it to pointer p. When reading a packet, this function looks for start of Kermit packet, then reads everything between it and the end of the packet into the indicated buffer. It returns the number of bytes read; 0 on timeout or other possibly correctable error; 1 on fatal error, such as loss of connection, or no buffer to read into.
3. Module ‘tx_data’ (Appendix B.21) , this routine transfers data of transmit length n from pointer p. It returns values based on write I/O error. The definitions of return values and data structures used can be found in kermit.h and cdefs.h. Timeout values must be handled for writes since the target system cannot handle much faster communication.

Hence for this project, the function usage of module ‘write’ (Appendix B.22), has been altered. Instead of writing all the bytes at once, each byte is transferred to target in a loop so that it can read the data in much slower pace than expected. One-second timeout values are given between each transfer of byte using the functions ‘usleep’.

The config (no extension) file contains configurations settings as key-value pairs, which are expected to be in the specific format as shown in Appendix B.23. The config file settings are used for the target system to identify the folder settings and names for successful transfer of requested files from the host. The entry “host_src_dir” specifies the folder from which the target system must expect data to arrive; if data from a different folder gets transferred from the host by a user error or accident, then those files will not be accepted. The entry “target_src_dir” specifies the name of the directory into which the files must be placed on the target system. If the directory does not exist, a new directory will be created. The entry “endfile_name” specifies the unique file name that gets transferred to the target system instructing the end of transfer so that any number of files in the “host_src_dir” folder can be transferred. The final entry “overwrite” specifies the mode of write operation on the target system. If the files that are transferred already exist in the target file system, this setting specifies whether these files must be overwritten or the write operation must be overridden.

The file “nanos_dl.sh” is a shell script that handles all the transfers of files from host system to target system, which encompasses the transfer of config file (followed by user command from target system to initiate file transfer); transfer of files in the host source directory and transfer of a unique file signals the end of transfer to the target system. The contents of the script can be seen at Appendix B.24.

The file “nanostxfr3n6” was chosen randomly to be a unique name for the target system to identify that that file transfer initiated and ongoing is coming to an end so that data

structures used on the target system can be reset. This file name can be modified to be anything unique so that the file names being transferred from the host source directory do not confuse the target system code. This name is changed in the config file so that the data structure used in the NanOS target system identifies the change during the config file transfer.

5.3. File Download Interface

The target system UART device drivers are handled in polled I/O mode. The interrupt driven mechanism originally adopted and explained in the CS 159 course work is modified to fit the needs of file download ability. The NanOS source files irq34.c and irq34.h contains all initializations and implementation of modules for handling UART communication. The structure shown in Appendix B.25 is used to define the terminal parameters often used in the driver modules. This is a part of NanOS source code.

Three important modules that need special attention for NanOS to support file download form host system through UART are explained below.

1. The module "TerminalInit_polledio" is used to set the UART ports on the target system to required settings for successful completion between the target and host system. The header file in the spede library spede/machine/rs232.h consists of defines handling the port names and numbers for detailed use. The definition can be seen at Appendix B.26.
2. The module “put_serial_char_polledio” defines the polled I/O module for sending data byte by byte from NanOS. The definition can be seen at Appendix B.27.
3. The module “get_serial_char_polledio” defines the polled I/O module for reading data byte by byte to NanOS. The definition can be seen at Appendix B.28.

The e-kermit source code has been added to the NanOS source files as kermit.c, kermit.h, kermitio.c, kermitio.h and kdefs.h. The same kermit source code that was used on the host system to initiate the file download is used in receiving mode on the target side. Some modules in the raw source code have to be tweaked so that they would work in receiving mode. Some of the changes in the interfacing modules that help in accessing the UART ports on the target side are discussed below. Kermit structure "k_data" handles the whole communication, which gets initialized in module transfer() in file file_tranfer.c. The files kermitio.h and kermitio.c implements the following modules that are used by core "kermit" module in kermit.c for reading and writing data packets using the UART ports and File System.

1. Module "readpkt" handles the incoming data on the UART ports and forms packets that are managed by the protocol. The implementation is shown in Appendix B.29.
2. Module "tx_data" handles the outgoing data to the UART ports, and sends the packets that are managed by the protocol. The implementation is shown in Appendix B.30.
3. The critical component on the NanOS side that handles the downloaded files is module "writefile". This helps in writing the files to the file system with the data gathered and stored in the buffer by kermit protocol. This module in turn uses modules "mkdir_dl" and "mkfile_dl" which are modified version of making files and directories taken from "FullFAT". These modules further use FF_MkDir and FF_Write functions with global I/O Manager structures defined to handle file management. This module also handles the different phases of the file download.
    a. "config phase" helps in obtaining the configuration file and initialize required data structures that will be used for all the files that are transferred or downloaded.

b. “data phase” helps in transferring raw data which gets assembled to files in NanOS file system;
c. “end phase” marks the end of file transfer that is identified by download of a specific file, which is stored in target system and also set in the data structures during “config phase”.

The file download data structure is defined in Appendix B.31. This structure stores all the configuration information and uses the data for the complete file download process. The transformation from “config phase” to “data phase” is done by a very basic global variable check on variable “nanosdl_config_check” which gets set based on the command line execution as discussed in Appendix A.2. The data phase handling of this is the complicated process of various string processing and creation and deletion of files and folders based on the config settings parameter “overwrite”. The end phase is also handled here just by identifying the name of the file, which is preset and needs to be changed if they are altered in the configuration file. The code handling of “config phase” and “data phase” can be viewed in Appendix B.31.

There are command line options on the NanOS for initiating the file download is explained in USER GUIDE Appendix A.2.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This project can be considered as small step in realizing an optimized and robust file system for the CSC 159 Operating Systems Pragmatics Course Work. Major areas can be concentrated on for robustness and optimization. Optimization was not considered for the design of this project. Major work on optimization can be considered in areas of disk device driver design and file system design.

Disk-scheduling algorithms that help in determining the motion of disk's arm and head and can service the disk reads and writes faster can be implemented for improving performance. FAT16 file system is a simple file system that does not give robustness in terms of catastrophic events such as power loss and system crash. To handle such scenarios file systems must include journaling, a journaling file system keeps track of changes made in a journal and helps in recovery of the system from catastrophic scenarios. This also reduces the time to bring a system online and makes the system less susceptible to data corruption. There are lots of open source projects that can be used as reference for improving the project. Some of the notable ones are various versions of Linux Extended file systems and many existing disk scheduling algorithms in Linux source like Anticipatory scheduling. Also adding code for using cache in the system might help in increasing the performance factor. The next phase of the project can be adding some of the above-mentioned functions so that the Operating System developed by students can have a robust file system for their course-work.

Appendix A

USER GUIDE

1. Hard Drive Setup on Target System

The size of the Hard Drive used by NanOS has to be set using FDISK utility of MSDOS. Following step-by-step procedure can be used to achieve the required Hard Disk setup on target system.

1) To start FDISK from a start-up disk, put the disk in drive a: and press CTRL+ALT+DEL to restart the computer. At the command prompt on the drive a:, type FDISK.
2) To start FDISK from a hard disk, press CTRL+ALT+DEL to restart the computer. At the command prompt on the drive c: , type FDISK.
3) Since we have hard disk greater than 512 MB, if a screen appears with option of selecting FAT16/32, select yes.
4) The following screen can manage Creation and Deletion of primary/extended partition. All the available partitions can be deleted or a primary partition can be maintained for running MSDOS. The deletion of partitions must be performed in the following order:
    a. Any non-MS-DOS partitions.
    b. Any logical drives in the extended MS-DOS partition
    c. Any extended MS-DOS partition
    d. The existing primary MS-DOS partition.
5) Create an Extended partition, which can be accessed by NanOS File System and Disk Device Driver.
6) Primary Partitions are usually created for installation of MSDOS in the hard disk. This project requires creation of an extended partition that will be handled by NanOS FAT16 file system.

7) The system setup used by NanOS development environment has the following disk configuration.

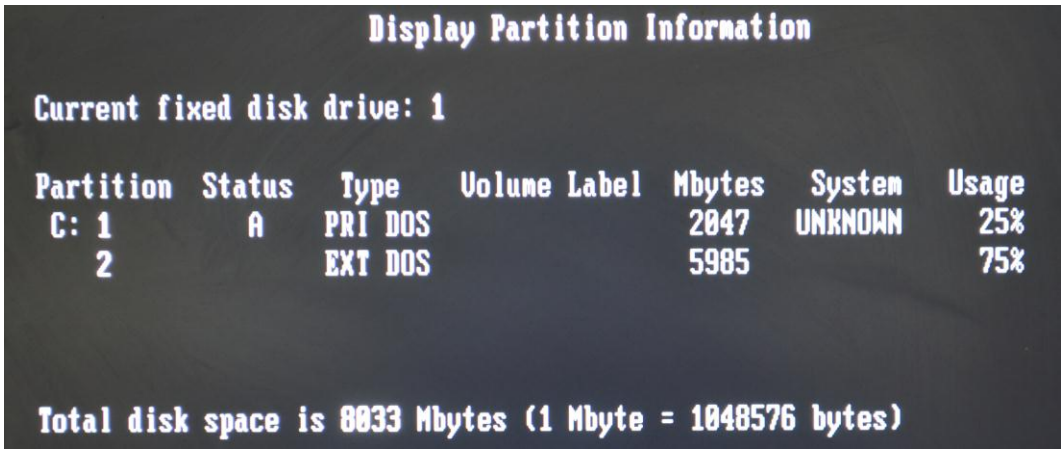

Figure 6.1
Display Partition Information

At the end of the setup, the total size of the hard disk used by NanOS would be 5985 Megabytes.

2. File Download Process Usage

1) Booting NanOS, using SPEDE tools once the required executable files have been generated, they have to be loaded using the debugger via serial communication. Many messages will be displayed on the monitor regarding setting up of the target system, once a command prompt is seen, the NanOS is considered to be ready.
2) After booting the NanOS on the target machine, after a series of file system and device driver self-explanatory messages, wait for the login and enter username and password (identical username and password will pass through the validation). After successful login, wait for the prompt. If the login is disabled, then the prompt will be shown without requesting login credentials. Now NanOS is ready for taking supported user commands.

3) On the host system, copy the folder “nanos_dl”, which is provided along with this project to desired location. The file “config” can be modified as required which includes the following settings
    a. The source directory on the host system from which the files and folders will be copied. (default folder “download” inside folder “nanos_dl”).
    b. The destination folder that specifies the path where the files and folders will be transferred to the target file system. The file system on target is designed to work Unix-like and the root is “/”. The command “pwd” can be used to display the current path.
    c. Default filename that will be transferred indicating the end of file transfer.
    d. Overwrite protection where if a file or folder exists on the target file system, enabling this option will overwrite the existing ones and vice versa.
4) After the above settings or conditions are achieved, type “dwnldC” on target command prompt, which will put the target system in a state where the system terminal is waiting for a configuration file to be downloaded to start the file transfer.
5) Now from the terminal on the host system, change the directory to “nanos_dl” folder and type “bash nanos_dl.sh”, which will start the downloading of the configuration file.
6) Once the downloading of configuration file is complete, the target system terminal will display a success message and host system terminal will be waiting for a message to start the file transfer from target system.
7) Now type “dwnldS” on the target system which will kick-start the file transfer between the systems and wait for the display of success message on the terminal.

3. Adding File Download Option to the existing CSC 159 course work projects.

1) There are 3 large modules that are to be added for the project to be used with any course-work projects. These modules are maintained in different files.
    a. Device Driver Module
    b. File System Module
    c. File Transfer Module
2) For accessing Device Driver Modules, the following files must be added.
    a. ata.c , ata.h
    b. ata_process.c , ata_process.h

    The following functions defined in ata_process.c can be used for performing disk reads and write.
    a. int DFS_ReadSector (u8 *buffer, u32 sector, u32 count, u8 unit);
    b. int DFS_WriteSector (u8 *buffer, u32 sector, u32 count, u8 unit);
3) For accessing File System modules, the following files must be added and the
    a. dosfs.c , dosfs.h
    b. ff_blk.c , ff_blk.h
    c. ff_cmd.c , ff_cmd.h
    d. ff_crc.c , ff_crc.h
    e. ff_dir.c , ff_dir.h
    f. ff_error.c . ff_error.h
    g. ff_fat.c , ff_fat.h
    h. ff_file.c , ff_file.h
    i. ff_hash.c , ff_hash.h
    j. ff_ioman.c , ff_ioman.h

k. ff_md5.c , ff_md5.h

l. ff_memory.c , ff_memory.h

m. ff_safety.c , ff_safety.h

n. ff_string.c , ff_string.h

o. ff_time.c , ff_time.h

p. ff_types.h , ff_fatdef.h , ff_config.h

q. fullfat.h

Terminal commands for requesting file system processing can be performed using the following functions defined in FFTerm.h.

a. FFT_CONSOLE_SET *FFTerm_CreateConsoleSet (FF_T_SINT32 *pError);

b. FF_T_SINT32 FFTerm_AddCmd (FFT_CONSOLE_SET *pConsoleSet, const FF_T_INT8 *pa_cmdName, FFT_FN_COMMAND pa_fnCmd, const FFT_ERR_TABLE *pa_ErrTable);

4) For accessing File Transfer Modules, the following files must be added.

a. kermit.c , kermit.h

b. kermitio.c , kermitio.h

c. file_transfer.c

Looking into the module “file_transfer” inside file_transfer.c can perform this functionality.

Appendix B

SOURCE CODE REFERENCE

1. Disk IO Data Structures

```
/* ATA or ATAPI command structure */

typedef struct{

        u32  Blk;        /* in SECTORS */

        u32  Count;      /* in BYTES */

        u8   Dev;

        u8   Cmd;        /* DRV_CMD_RD or DRV_CMD_WR */

        u8 * Data;

} DriveCmd;


/* Generalized drive info structure */

typedef struct{

u16  Flags;

        u8   DrvSel;                    /* ATA, ATAPI only (LUN for SCSI?) */

        u8   MultSect;                  /* ATA only */

        u16  Sects, Heads, Cyls;        /* CHS ATA only */

        u16  IOAdr;

} DriveSpec;


/* Partition info*/

typedef struct{
```

```
    u32     PartNo;             /* Part No */
    u8      FileDescriptor;     /* File Descriptor */
    u32     StartLBA;           /* Starting Sector */
    u32     TotalLBAs;          /* Total no of Sectors */
} PartitionInfo;
```

2. ATA Identify Structure

```
typedef struct{
    u16 Config; // 0 see spec
    u16 PhysCyls; // 1 Nof Logical Cylinders
    u16 Res2; // 2
    u16 PhysHeads; // 3 Nof Logical Heads
    u32 Vendor4_5; // 4 to 5
    u16 PhysSects; // 6 Nof Logical Sectors per Logical track
    u16 Vendor7_9[3]; // 7 to 9
    u8  SerialNum[10*2]; // 10 to 19  ASCII serial number
    u32 Obs20_21; // 20 to 21
    u16 NofVendorBytesRWLongCmd;// 22 Nof Vendor Specific bytes for Read/Write Long Cmds
    u8  FirmwareRev[4*2]; // 23 to 26 Firmware Revision ASCII
    u8  Model[20*2]; // 27 to 46 Model No ASCII
    u16 RWMultSupp; // 47 RWMultipleSupport;
    u16 Res48; // 48
    u16 Capability; // 49 see spec
    u16 Res50; // 50
```

u16 PIODataTxCycTimeMode; // 51 PIO Data Transfer Cycle Timing mode

u16 Obs52; // 52

u16 FieldValidity; // 53 Field Value

u16 LogCyls; // 54 Number of current logical cylinders

u16 LogHeads; // 55 Number of current logical heads

u16 LogSectsPerTrack; // 56 Number of current logical sectors per logical track

u32 LBASects; // 57 to 58 Current capacity in sectors

u16 MultSect; // 59 Multiple sector setting

u32 UsrAddrSects; // 60 to 61 Total number of user addressable sectors

u16 Obs62; // 62

u16 DMAInfoMult; // 63 Multiword DMA transfer

u16 FCPIOMode; // 64 Flow control PIO transfer modes supported

u16 MinMultiWrdDMATxCycTime; // 65 Minimum multiword DMA transfer cycle time per word

u16 DevMultiWrdDMACycTime; // 66 Device recommended multiword DMA cycle time

u16 MinPIOTxCycTimeNoFC; // 67 Minimum PIO transfer cycle time without flow control

u16 MinPIOTxCycTimeIORDY; // 68 Minimum PIO transfer cycle time with IORDY

u16 Res69_79[10]; // 69 to 79

u16 MajVerNo; // 80 Major version number

u16 MinorVerNo; // 81 Minor version number

u32 CmdSetSup; // 82 to 83 Command sets supported see spec

u16 Res84_127[43]; // 84 to 127

u16 SecurityStat; // 128

```
    u16 Vendor129_159[30]; // 129 to 159
    u16 Res160_255[49]; // 160 to 255
} ataid;
```

3. The following defines explain the drive command bytes. The Instruction command register explained in the previous section will be written with one of the following values:

```
/* ATA drive command bytes */
#define ATA_CMD_RD      0x20        /* read one sector */
#define ATA_CMD_WR 0x30         /* write one sector */
#define ATA_CMD_PKT     0xA0        /* ATAPI packet cmd */
#define ATA_CMD_PID     0xA1        /* ATAPI identify */
#define ATA_CMD_RDMUL   0xC4        /* read multiple sectors */
#define ATA_CMD_WRMUL   0xC5        /* write multiple sectors */
#define ATA_CMD_ID      0xEC        /* ATA identify */
```

4. Master Boot Record structure from dosfs.h

```
typedef struct _tagMBR {
    u8 bootcode[0x1be];     // boot sector
    PTINFO ptable[4];               // four partition table structures
    u8 sig_55;                      // 0x55 signature byte
    u8 sig_aa;                      // 0xaa signature byte
} MBR, *PMBR;
```

5. Partition table entry structure from dosfs.h

```
typedef struct _tagPTINFO {
    u8      active;             // 0x80 if partition active
```

```
    u8      start_h;        // starting head
    u8      start_cs_l;             // starting cylinder and sector (low byte)
    u8      start_cs_h;             // starting cylinder and sector (high byte)
    u8      type;                   // type ID byte
    u8      end_h;                  // ending head
    u8      end_cs_l;               // ending cylinder and sector (low byte)
    u8      end_cs_h;               // ending cylinder and sector (high byte)
    u8      start_0;        // starting sector# (low byte)
    u8      start_1;
    u8      start_2;
    u8      start_3;        // starting sector# (high byte)
    u8      size_0;                 // size of partition (low byte)
    u8      size_1;
    u8      size_2;
    u8      size_3;                 // size of partition (high byte)
} PTINFO, *PPTINFO;
```

6. Logical Boot Record structure (volume boot sector) from dosfs.h

```
typedef struct _tagLBR {
    u8 jump[3];             // JMP instruction
    u8 oemid[8];            // OEM ID, space-padded
    BPB bpb;                // BIOS Parameter Block
    union {
      EBPB ebpb;   // FAT12/16 Extended BIOS Parameter Block
     //EBPB32 ebpb32;// FAT32 EBP Block not used
```

```
        } ebpb;
        u8 code[448];           // boot sector code
        u8 sig_55;              // 0x55 signature byte
        u8 sig_aa;              // 0xaa signature byte
} LBR, *PLBR;
```

7. BIOS Parameter Block structure (FAT12/16) from dosfs.h

```
typedef struct _tagBPB {
        u8 bytepersec_l;        // bytes per sector low byte (0x00)
        u8 bytepersec_h;        // bytes per sector high byte (0x02)
        u8 secperclus;          // sectors/cluster (1,2,4,8,16,32,64,128 are valid)
        u8 reserved_l;          // reserved sectors low byte
        u8 reserved_h;  // reserved sectors high byte
        u8 numfats;             // number of FAT copies (2)
        u8 rootentries_l;       // no of root dir entries low byte (0x00 normally)
        u8 rootentries_h;       // no of root dir entries high byte (0x02 normally)
        u8 sectors_s_l;  // small num sectors low byte
        u8 sectors_s_h; // small num sectors high byte
        u8 mediatype;           // media descriptor byte
        u8 secperfat_l;  // sectors per FAT low byte
        u8 secperfat_h; // sectors per FAT high byte
        u8 secpertrk_l;  // sectors per track low byte
        u8 secpertrk_h; // sectors per track high byte
        u8 heads_l;             // heads low byte
        u8 heads_h;             // heads high byte
```

```
    u8 hidden_0;        // hidden sectors low byte
    u8 hidden_1;        // (note - this is the no of MEDIA sectors before
    u8 hidden_2;    // 1st sector of VOLUME - we rely on the MBR instead)
    u8 hidden_3;        // hidden sectors high byte
    u8 sectors_l_0; // large num sectors low byte
    u8 sectors_l_1;
    u8 sectors_l_2;
    u8 sectors_l_3; // large num sectors high byte
} BPB, *PBPB;
```

8. Extended BIOS Parameter Block structure (FAT12/16) from dosfs.h

```
typedef struct _tagEBPB {
    u8 unit;        // int 13h drive#
    u8 head;        // archaic, used by Windows NT-class OSes for flags
    u8 signature;       // 0x28 or 0x29
    u8 serial_0;        // serial#
    u8 serial_1;        // serial#
    u8 serial_2;        // serial#
    u8 serial_3;        // serial#
    u8 label[11];       // volume label
    u8 system[8];       // file system ID
} EBPB, *PEBPB;
```

9. Directory Entry Data Structure (DIRENT) in FullFAT,

```
typedef struct {
    FF_T_INT8   FileName[FF_MAX_FILENAME];
```

```
    FF_T_UINT8    Attrib;
    FF_T_UINT32 Filesize;
    FF_T_UINT32  ObjectCluster;

#ifdef FF_TIME_SUPPORT
    FF_SYSTEMTIME     CreateTime;          ///< Date and Time Created.
    FF_SYSTEMTIME     ModifiedTime;  ///< Date and Time Modified.
    FF_SYSTEMTIME     AccessedTime;  ///< Date of Last Access.
#endif
    //---- Book Keeping for FF_Find Functions
    FF_T_UINT16  CurrentItem;
    FF_T_UINT32  DirCluster;
    FF_T_UINT32  CurrentCluster;
    FF_T_UINT32 AddrCurrentCluster;
    //FF_T_UINT8  NumLFNs;
} FF_DIRENT;
```

10. File Date and Time Data Structure from FullFAT,

```
typedef struct {
    FF_T_UINT16  Year;        ///< Year      (e.g. 2009).
    FF_T_UINT16 Month;        ///< Month     (e.g. 1 = Jan, 12 = Dec).
    FF_T_UINT16  Day;         ///< Day       (1 - 31).
    FF_T_UINT16 Hour;         ///< Hour      (0 - 23).
    FF_T_UINT16 Minute;       ///< Min       (0 - 59).
    FF_T_UINT16 Second;       ///< Second    (0 - 59).
```

} FF_SYSTEMTIME;

11. IO Buffer Data Structure in FULLDAT,

typedef struct {

FF_T_UINT32 Sector; ///< The LBA of the Cached sector.

FF_T_UINT32 LRU; ///< For the Least Recently Used algorithm.

FF_T_UINT16 NumHandles; ///< Number of objects using this buffer.

FF_T_UINT16 Persistence; ///< For the persistence algorithm.

FF_T_UINT8 Mode; ///< Read or Write mode.

FF_T_BOOL Modified; ///< If the sector was modified since read.

FF_T_BOOL Valid; ///< Initially FALSE.

*FF_T_UINT8 *pBuffer; ///< Pointer to the cache block.*

} FF_BUFFER;

12. Input Output Manager Structure in FullFAT,

typedef struct {

*FF_BLK_DEVICE *pBlkDevice; // Pointer to a Block device description.*

*FF_PARTITION *pPartition; // Pointer to a partition description.*

*FF_BUFFER *pBuffers; // Pointer to the first buffer description.*

*void *pSemaphore;*

// < Pointer to a Semaphore object.

// (For buffer description modifications only!).

*void *FirstFile; // Pointer to the first File object.*

*FF_T_UINT8 *pCacheMem; // Pointer to a block of memory for the cache.*

FF_T_UINT32 LastReplaced;

// Marks which sector was last replaced in the cache.

FF_T_UINT16 BlkSize; //The Block size that IOMAN is configured to.

FF_T_UINT16 CacheSize; // Size of the cache in number of Sectors.

FF_T_UINT8 PreventFlush; // Flushing to disk only allowed when 0

FF_T_UINT8 MemAllocation; // Bit-Mask identifying allocated pointers.

FF_T_UINT8 Locks;

// Lock Flag for FAT & DIR Locking etc

// This must be accessed via a semaphore).

FF_T_UINT32 status; // added for nanos download support

} FF_IOMAN;

13. Function definitions for creating and destroying IO Managers, Registering and Unregistering Block Devices on the system and Mounting and Unmounting Partitions in FullFAT.

*FF_IOMAN *FF_CreateIOMAN (FF_T_UINT8 *pCacheMem, FF_T_UINT32 Size, FF_T_UINT16 BlkSize, FF_ERROR *pError);*

*FF_ERROR FF_DestroyIOMAN (FF_IOMAN *pIoman);*

*FF_ERROR FF_RegisterBlkDevice (FF_IOMAN *pIoman, FF_T_UINT16 BlkSize,*

*FF_ERROR FF_UnregisterBlkDevice (FF_IOMAN *pIoman);*

*FF_ERROR FF_MountPartition (FF_IOMAN *pIoman, FF_T_UINT8 PartitionNumber);*

*FF_ERROR FF_UnmountPartition (FF_IOMAN *pIoman);*

14. Structure used to maintain the interface that access the device driver modules.

typedef struct {

FF_WRITE_BLOCKS fnWriteBlocks; // Func Ptr, to write a block(s) from blk device.

FF_READ_BLOCKS fnReadBlocks; // Func Ptr, to read a block(s) from a blk device.

FF_T_UINT16 devBlkSize; // Block size that the driver deals with.

*Void *pParam; // Pointer to some parameters e.g. for a Low-Level Driver Handle*

} FF_BLK_DEVICE;

15. Function Interface Definition in FullFAT for accessing Device Driver modules.

*typedef FF_T_SINT32 (*FF_WRITE_BLOCKS) (FF_T_UINT8 *pBuffer, FF_T_UINT32 SectorAddress, FF_T_UINT32 Count, void *pParam);*

*typedef FF_T_SINT32 (*FF_READ_BLOCKS) (FF_T_UINT8 *pBuffer, FF_T_UINT32 SectorAddress, FF_T_UINT32 Count, void *pParam);*

16. Partition Details maintained in the following structure in FullFAT.

```
typedef struct {
        //FF_T_UINT8  ID;       //< Partition Incremental ID number.
        FF_T_UINT8    Type;    //< Partition Type Identifier.
        FF_T_UINT16  BlkSize; //< Size of a Sector Block in bytes.
        FF_T_UINT8     BlkFactor; //< Scale Factor for blocksizes above 512!
        //FF_T_INT8    Name[FF_MAX_PARTITION_NAME];
//< Partition Identifier e.g. c: sd0: etc.
        FF_T_INT8      VolLabel[12];   //< Volume Label of the partition.
        FF_T_UINT32  BeginLBA; //< LBA start address of the partition.
        FF_T_UINT32  PartSize; //< Size of Partition in number of sectors.
        FF_T_UINT32  FatBeginLBA; //< LBA of the FAT tables.
        FF_T_UINT8    NumFATS; //< Number of FAT tables.
        FF_T_UINT32  SectorsPerFAT; //< Number of sectors per Fat.
        FF_T_UINT8    SectorsPerCluster; //< Number of sectors per Cluster.
        FF_T_UINT32  TotalSectors;
        FF_T_UINT32  DataSectors;
        FF_T_UINT32  RootDirSectors;
```

FF_T_UINT32 FirstDataSector;

FF_T_UINT16 ReservedSectors;

FF_T_UINT32 ClusterBeginLBA; //< LBA of first cluster.

FF_T_UINT32 NumClusters; //< Number of clusters.

FF_T_UINT32 RootDirCluster;

//< Cluster number of the root directory entry.

FF_T_UINT32 LastFreeCluster;

FF_T_UINT32 FreeClusterCount; //< Records free space on mount.

FF_T_BOOL PartitionMounted;

//< FF_TRUE if the partition is mounted, otherwise FF_FALSE.

#ifdef FF_PATH_CACHE

FF_PATHCACHE PathCache[FF_PATH_CACHE_DEPTH];

FF_T_UINT32 PCIndex;

#endif

} FF_PARTITION;

17. The message structure is shown below

```
typedef struct {
    int sender;              // sender
    int send_tick;           // time sent
    int numbers[NUM_NUM];    // several integers can be useful as msgs
    char bytes[NUM_BYTE];    // some chars can be useful as msgs
} msg_t;
```

The define NUM_NUM holds value 10 and NUM_BYTE holds value 101, which means each message can have 10 integer values and a string of 100 characters in length (for 1 null character at

end). For simplicity, the numbers are ordered so that each of the array element index can be used to hold sender process id, message command, message count (usually not used) and remaining messages. These are defined in file proc.h.

```
// message indexes
#define MB_MSG_SENDER_PID       0
#define MB_MSG_CMD              1
#define MB_MSG_COUNT            2
#define MB_MSG_1            3
#define MB_MSG_2            4
#define MB_MSG_3            5
#define MB_MSG_4            6
#define MB_MSG_5            7
#define MB_MSG_6            8
#define MB_MSG_7            9
```

MB_MSG_CMD define is used mainly to identify the difference between commands exchanged between different processes.

18. Definition of module devopen in unixio.c,

*devopen (char *ttyname, long baud)*

19. Modified part of the code under the module devopen, the following lines were added to alter the UART communication settings

```
settty.c_cflag |= CS8; //8-N-1
```

20. Definition of module readpkt in unixio.c,

*readpkt (struct k_data *k, UCHAR * p, int len, int fc),*

21. Definition of module tx_data in unixio.c, *tx_data (struct k_data *k, UCHAR * p, int n),* in file unixio.c

22. Modified part of the code under the module tx_data, definition shown in Appendix B.19, for handling the interrupts on transfer of acknowledgements from target ports data,

```
while (n > 0)
{               /* Keep trying till done */
for(i=0;i<n;i++)
{
        x = write (ttyfd, (p), 1);
        usleep (100000);
        if (x == -1 && max > 0)
        {
                usecs = k->s_maxlen * (10000000L / k->baud);
                usleep (usecs);
                continue;
        }
        if (x < 0 || max < 1)     // Errors are fatal
                return (X_ERROR);
        n -= x; // decrease the transferred data length
        p += x; // move the data pointer for new data to be transferred.
}
```

23. The config file format that is maintained in the host system for downloading files.

```
host_src_dir=/home/nanos_dl/download/
target_src_dir=/dwnld
```

endfile_name=/home/nanos_dl/nanostxfr3n6

overwrite=1

24. In the above nanos_dl.sh script, the details from the config file have been hardcoded, but the script can be modified so that they can read from the config file that should exist in the same directory.

```
#!/bin/bash
# modify the script so that the config file changes the nof files
# based on no files identified by this script
tgt_files_path="/home/nanos_dl/download"
config_file_path="/home/nanos_dl/config"
endtxfr_file_path="/home/nanos_dl/nanostxfr3n6"
fine_transfer_out="eksw094_s/eksw"
uart_out=/dev/ttyUSB1
file=`find $tgt_files_path -type f -print| wc -l`
dir=`find $tgt_files_path -type d -print | wc -l`
# Result
echo "Total number of files in $tgt_files_path directory is $file"
echo "Total number of directory in $tgt_files_path directory is $dir"
file_list=`find $tgt_files_path -type f -name \* -print`
echo "sending config file.... $config_file_path"
$fine_transfer_out $uart_out 9600 -s $config_file_path
echo "to make target start receiving files type dwnldS on target console"
for file in $file_list
do
```

```
 echo "sending .... $file"
 $fine_transfer_out $uart_out 9600 -s $file
done
echo "sending specific file name to end transfer.... $endtxfr_file_path"
$fine_transfer_out $uart_out 9600 -s $endtxfr_file_path
```

25. Structure that sets the UART COM Ports on the target system.

```
terminal_t terminals[NUM_TERM]=
{
  {COM2_IOBASE, 3}, // IO port base and IRQ #
};
```

26. Module TerminalInit_polledio.

```
void TerminalInit_polledio(int term_num) {
int BAUD_RATE = 9600;
int divisor = 115200 / BAUD_RATE;
// abbrevs and usage
// CFCR Char Format Control Reg, MSR Modem Status Reg, IIR Intr Indicator Reg
// MCR Modem Control, IER Intr Enable , LSR Line Status Regs
// ERXRDY Enable Recv Ready, ETXRDY Enable Xmit Ready
// LSR_TSRE Line Status Reg Xmit+Shift Regs Empty
outportb(terminals[term_num].io_base + CFCR, CFCR_DLAB);
// CFCR_DLAB is 0x80
outportb(terminals[term_num].io_base + BAUDLO,LOBYTE(divisor));
outportb(terminals[term_num].io_base + BAUDHI,HIBYTE(divisor));
outportb(terminals[term_num].io_base + CFCR,CFCR_8BITS ); //8-N-1
```

```
outportb(terminals[term_num].io_base + IER,0);
// raise DTR & RTS of the serial port to start read/write
outportb(terminals[term_num].io_base + MCR, MCR_DTR | MCR_RTS |   MCR_IENABLE);
outportb(terminals[term_num].io_base + IER,0);
//FIFO stuff
outportb(terminals[term_num].io_base + FIFO, FIFO_ENABLE | FIFO_TRIGGER_8);
outportb(terminals[term_num].io_base + FIFO, FIFO_RCV_RESET | FIFO_XMT_RESET);
}
```

27. Module get_serial_char_polledio .

```
u8 get_serial_char_polledio() {
  while(!(inportb(terminals[FT_TERM].io_base + LSR) & LSR_RXRDY)){
  }
  return (inportb(terminals[FT_TERM].io_base + DATA) & 0x7F);
}
```

28. Module put_serial_char_polledio .

```
void put_serial_char_polledio(u8 ch) {
  while(!(inportb(terminals[FT_TERM].io_base + LSR) & LSR_TXRDY)){
  }
  outportb(terminals[FT_TERM].io_base + DATA), ch);
}
```

29. Module readpkt in NanOS,

```
int readpkt (struct k_data *k, UCHAR * p, int len, int fc) {
 while (1) {
    x = get_serial_char_polledio();
```

```
    c = (k->parity) ? x & 0x7f : x & 0xff;    // Strip parity
    ….
   return (-1);
 }
}
```

30. Module tx_data in NanOS,

```
int tx_Data (struct k_data *k, UCHAR * p, int len, int fc) {
 ..
 put_serial_buf_polledio(p, n);
}
```

31. The file download data structure,

```
struct nanos_dwnld_strct
{
  short txfr_status;            /* Current status */
  short overwrite;
  UCHAR host_dir_name[FN_MAX];  /* Name of current file */
  UCHAR trg_dir_name[FN_MAX];   /* Date of file */
  UCHAR endfile_name[FN_MAX];   /* Date of file */
};
```

31. The config phase and data phase code handling for file download process,

```
int writefile (…) {
……
if(nanosdl_config_check == 0){
tn = 0;  ts = 0;
```

```
    for(i=0;i<n;i++){
      tn++;
      if(*(s+i) ==  10){
      memcpy(sample, &s[ts], tn);
      ts = i+1;  tn1 = 0;
      for(j=0;j<tn;j++){
      tn1++;
      if(*(sample+j) ==  '='){ // =
      config_value[0] = '\0';
      memcpy(config_value, &sample[tn1], tn-tn1-1);
      config_value[tn-tn1-1]='\0';
      switch(index){
        case 0: strcpy(nanosdl_str.host_dir_name,config_value);
          break;
        case 1: strcpy(nanosdl_str.trg_dir_name,config_value);
          break;
        case 2: strcpy(nanosdl_str.endfile_name,config_value);
          break;
case 3: if(strcmp(OVERWRITE,config_value) == 0)
     nanosdl_str.overwrite = 1;
          else nanosdl_str.overwrite = 0;
            break;
    }
    index++;
```

```
} } tn = 0; } else {
 // data phase
 cons_printf("\n entering data_phase ....");
 rcv_fnmlen = strlen(k->filename);
 //check for end of transfer
 if(strcmp(k->filename,nanosdl_str.endfile_name) == 0){
  nanosdl_str.txfr_status = 1;
  return rc;
 }
 cfg_host_dir_fnmlen = strlen(nanosdl_str.host_dir_name);
 if(cfg_host_dir_fnmlen > rcv_fnmlen) {
 } else {
  memcpy(prefix_str,k->filename,cfg_host_dir_fnmlen);
  prefix_str[cfg_host_dir_fnmlen+1]='\0';
  if(strcmp(prefix_str,nanosdl_str.host_dir_name) == 0){
  memset(prefix_str,0,cfg_host_dir_fnmlen+1);
  memcpy(prefix_str,&(k->filename)[cfg_host_dir_fnmlen],rcv_fnmlen-cfg_host_dir_fnmlen);
  prefix_str[rcv_fnmlen-cfg_host_dir_fnmlen+1]='\0';
  result = strtok(prefix_str,dir_delim);
  nof_directories = 0;
 strcpy(dir_nanoscmd, nanosdl_str.trg_dir_name);
  if(mkdir_dl(dir_nanoscmd) == 0){
  } else {
    cons_printf("directory %s exists",dir_nanoscmd);
```

```
    }
    while(result != NULL){
memcpy(&directories_inpath[nof_directories],result,strlen(result));
        nof_directories++;
        result = strtok(NULL,dir_delim);
}
for(i=0;i<nof_directories;i++){
      strcat(dir_nanoscmd, "/");
      strcat(dir_nanoscmd, directories_inpath[i]);            if(i == nof_directories-1){
         if(mkfile_dl(dir_nanoscmd, s, n) == 0){
        printf("\nfile %s created successfully",k->filename);
          } else {
                printf("\nfile %s exists",k->filename);
          }
        }else{
    if(mkdir_dl(dir_nanoscmd) == 0){
      printf("\ndirectory %s created successfully",dir_nanoscmd);
    } else {
      printf("\ndirectory %s exists",dir_nanoscmd);
  }}}
  if(nof_directories == 0){
  } } else {
  printf("\nthe received file is not obtained from configured host folder");
  }} }
```

References

[1] ATA (IDE) hard drive detection code, by Chris Giese <geezer@execpc.com>, http://bos.asmhackers.net/docs/ata/snippet_1/ata-id.c

[2] FullFAT, fully featured FAT12/16 and FAT32 file system driver, by James Walmsley, http://www.fullfat-fs.co.uk/Home/About

[3] Embedded Kermit, authored by Frank da Cruz, Columbia University, NY, http://www.columbia.edu/kermit/ek.html

[4] "Building your first Operating System, A Microkernel approach", version 0.2.1, by Brian Witt and John Clevenger

[5] "SPEDE-2000 Lab Manual", by John Clevenger and BrianWitt

[6] IDE Hard Disk experiments, A website that explains how IDE works, http://hem.passagen.se/communication/ide.html

[7] X3T10/2008D: Information Technology - AT Attachment-3 Interface (ATA-3), draft

[8] File Allocation Table File System Reference, http://www.ntfs.com/fat-systems.htm

[9] Partition Table Reference, http://www.pjrc.com/tech/8051/ide/fat32.html

[10] FullFAT Open Source FAT File System, http://wiki.fullfat-fs.co.uk/wiki/Main_Page

[11] Documentation on E-Kermit Protocol, http://www.columbia.eduhttp://www.ntfs.com/fat-systems.htmhttp://www.ntfs.com/fat-systems.htm