IMPLEMENTATION OF A NETWORK ATTACHED STORAGE DEVICE ON AN ARM PLATFORM

A Project Report
submitted in partial fulfilment
of
the requirements for the award of the degree of

MASTER OF TECHNOLOGY

in

COMMUNICATION SYSTEMS

by

LT CDR SANTOSH KUMAR

EE05M022

Under the guidance of

Dr. TG Venkatesh

DEPARTMENT OF ELECTRICAL ENGINEERINGINDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

JUNE 2007

IMPLEMENTATION OF A NETWORK ATTACHED STORAGE DEVICE ON AN ARM PLATFORM

A Project Report
submitted in partial fulfilment
of
the requirements for the award of the degree of

MASTER OF TECHNOLOGY

in

COMMUNICATION SYSTEMS

by

LT CDR SANTOSH KUMAR

EE05M022

Under the guidance of

Dr. TG Venkatesh

DEPARTMENT OF ELECTRICAL ENGINEERINGINDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

JUNE 2007

CERTIFICATE

This is to certify that the report titled "**IMPLEMENTATION OF NETWORK ATTACHED STORAGE DEVICE ON AN ARM PLATFORM**", submitted by **Lt Cdr Santosh Kumar**, to the Indian Institute of Technology, Madras, for the award of the degree of Master of Technology, is a bona fide record of the work done by him under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

(Dr. TG Venkatesh)
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai
Date: 18th Jun 2007

ACKNOWLEDGMENTS

I would like to place on record my gratitude to the Indian Navy for this opportunity to undergo Post Graduate programme in this premier institution.

I wish to express my heartfelt gratitude to my project guide **Dr TG Venkatesh**, for his inspiring guidance, invaluable support and unflagging attention throughout the project.

I deeply appreciate the valuable suggestions and help offered by Mr V Krishna of iWave Systems for guiding me through the obstacles that I encountered while coding.

Lt Cdr Santosh Kumar

EE05M022

ABSTRACT

The ARM architecture (Advanced RISC Machine) is a 32-bit RISC processor architecture developed by ARM Limited and is widely used in a number of embedded designs. Due to their power saving features, ARM CPUs are dominant in the mobile electronics market, where low power consumption is a critical requirement.

The ARM family accounts for most of the 32-bit embedded CPUs,making it one of the most used 32-bit architectures in the world. A characteristic feature of ARM processors is their low electric power consumption, which makes them particularly suitable for use in portable devices. Almost all modern mobile phones and personal digital assistants contain ARM CPUs, making them the most widely-used 32-bit microprocessor family in the world, more so than the better-known 32-bit Pentium 4 processors found in many PCs.

This aim of the project is to develop a "**Network Attached Storage**" device which will enable various clients on various machine to access the storage device. The first step to implement this is to enable the detection of USB MSD (Mass Storage Device) by the arm board. This includes implementation of a Mass Storage Device (MSD) driver with the framework provided by Atmel. The frame work code provided by Atmel needs to be loaded to the flash everytime the board is booted up. This could be avoided by patching the MSD code to the latest Linux Kernel source code, making an image of the Kernel and porting it to the board.

After enabling the MSD , Samba Server has been compiled to be used for the ARM environment. The binary file are stored in the Hard disk and the server runs from there. Various clients can be configured from various windows / Linux machines.

A web sever using Apache and Perl script is being implemented to make the implementation more user friendly and easier access over the internet/intranet.

LIST OF FIGURES

The first step to implement was to enable the detection of USB MSD (Mass Storage Device) by the arm board. It requires developing of a code which shall enables various devices like hard disk, pen drive etc to be recognized. The USB mass storage device class is a set of computing communications protocols defined by the USB Implementers Forum that run on the Universal Serial Bus[(11)]. The standard provides an interface to a variety of storage devices. A USB flash drive typically implements the USB mass storage device class. This has been done using two methods:

- **As an Application**: This application needs to be loaded to the SDRAM location whenever the board boots up. This is done using the serial link between the PC and the AT91RM9200 board.
- **Linux Kernel level support**: A kernel image is prepared with USB MSD(Mass Storage Device) driver features and then ported to the specified Flash location.

It is easier and better to implement the Linux Kernel Level support. This Linux kernel has been compiled using the cross compiler and an image has been prepared. It can then be ported to the Flash Memory of the AT91RM9200 board.

Samba is a suite of Unix applications that speak the Server Message Block (SMB) protocol. Microsoft Windows operating systems and the OS/2 operating system use SMB to perform client-server networking for file and printer sharing and associated operations. By supporting this protocol, Samba enables computers running Unix to get in on the action, communicating with the same networking protocol as Microsoft Windows and appearing as another Windows system on the network from the perspective of a Windows client.

The Samba Server-Client configuration has implemented by cross compiling the Samba Server code for the ARM platform. The binary files generated are stored in the Mass Storing Device (ie hard disk) and are being soft linked to the Linux environment. It can therefore be accessed from anywhere in the network and various services are achievable. Hence it can then easily be recognized on various Windows/ Linux platforms.

Apache webserver has been used for web hosting with Perl module for scripting and controlling various resources. Perl script is used to generate HTML pages. The HTML pages are internally linked to each other. The web server used controls all password authentications and provides all the resources required by the user.

1.2 Scope and Organization of the Report

This report is about development of Network attached storage Device. Other than bringing out the work done, the report has been organized to meet the following requirements:

- ARM architecture.
- USB Specifications.
- Description of the Code for USB MSD(Mass Storage Device) driver
- Kernel Level support for USB.
- Configuration of Samba Server for ARM platform
- Configuring Apache Server for user friendly access.

The Project is organized into Eight chapters given in **Fig. 1.1**.

The following outlines the contents of the report. First chapter is introduction.

Chapter 2 Reviews of ARM Technology.

Chapter 3 Briefly explains ARM architecture and a small note on AT91RM9200.

Chapter 4 MSD Device Specifications & description of code

Chapter 5 Linux Kernel level support for the USB device.

Chapter 6 Configuring Samba Server for ARM.

Chapter 7 Configuring Apache for web access.

Chapter 8 Results and Conclusion

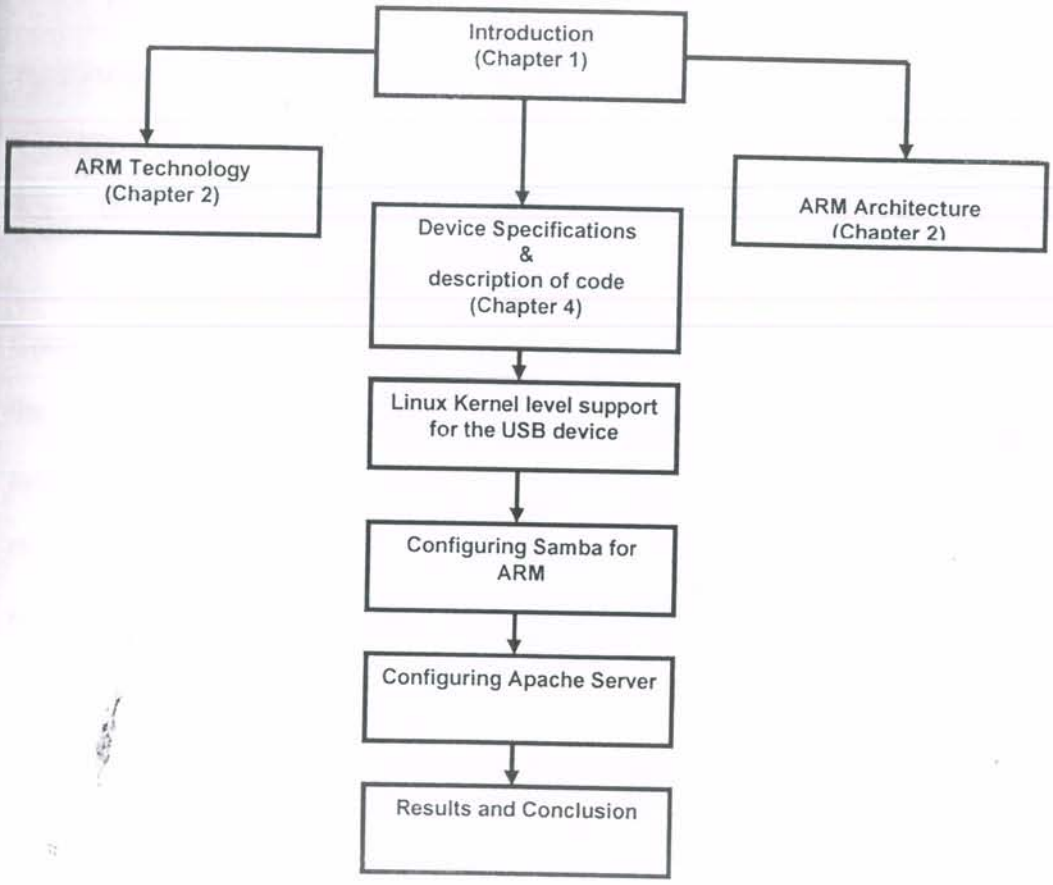


Figure 1.1: Report Organization

CHAPTER 2

ARM Technology

The ARM technology has become tremendously popular nowadays. The modularity and extensibility inherent in choices is a perfect match for the needs of a mobile device in a dynamic environment.

2.1 Development of ARM

The ARM design was started in 1983 as a development project at Acorn Computers Ltd. The first developed version was called ARM1 by April 1985, and the first "real" production systems as ARM2. The ARM2 featured a 32-bit data bus, a 26-bit address space giving a 64 Mbyte address range and sixteen 32-bit registers. One of these registers served as the (word aligned) program counter with its top 6 bits and lowest 2 bits holding the processor status flags. The ARM2 was the simplest useful 32-bit microprocessor in the world, with only 30,000 transistors. Much of this simplicity comes from not having microcode and, like most CPUs of the day, not including any cache. This simplicity led to its low power usage, while performing better than the Intel 80286. A successor, ARM3, was produced with a 4KB cache, which further improved performance.[(9)]

The first models of ARM6 were released in 1991, and was used as ARM 610 based main CPU in Risc PC computers. The core has remained largely the same size through-out these changes. ARM2 had 30,000 transistors, while the ARM6 grew to only 35,000. The Original Design Manufacturer combines the ARM

These registers such as the stack pointer are automatically switched when entering a different processor mode. This design allows fast processing of interrupts as the handler code does not need to manually switch to a new stack. The ARM architecture includes the following RISC features:

- Load/store architecture
- No support for misaligned memory accesses (now supported in ARMv6 cores)
- Orthogonal instruction set
- Large 16 Œ 32-bit register file
- Fixed opcode width of 32 bits to ease decoding and pipelining, at the cost of decreased code density
- Mostly single-cycle execution
- Conditional execution of most instructions, reducing branch overhead and compensating for the lack of a branch predictor
- Arithmetic instructions alter condition codes only when desired
- 32-bit barrel shifter which can be used without performance penalty with most arithmetic instructions and address calculations
- Powerful indexed addressing modes
- Simple, but fast, 2-priority-level interrupt subsystem with switched register banks

2.2 Advancement in ARM Technology

Advanced ARM processors have a 16-bit instruction set, called Thumb, perhaps related to the conditional execution facility using four bits of every instruction. In Thumb, the smaller opcodes have less functionality. For example, only branches can be conditional, and many opcodes cannot access all of the

CPU's registers. However, the shorter opcodes give improved code density overall, even though some operations require more instructions. Particularly in situations where the memory port or bus width is con-strained to less than 32 bits, the shorter Thumb opcodes allows greater performance than with 32-bit code because of the more efficient use of the limited memory bandwidth. The first processor with Thumb technology was the ARM7TDMI. All ARM9 and later families, including XScale have included Thumb technology.

2.3 AT91RM9200 based on ARM9 Technology

The AT91RM9200 is a complete system-on-chip built around the ARM920T ARM Thumb processor. It incorporates a rich set of system and application peripherals and standard interfaces in order to provide a single-chip solution for a wide range of compute-intensive applications that require maximum functionality at minimum power consumption at lowest cost.**[(8)]**

The AT91RM9200 incorporates a high-speed on-chip SRAM workspace, and a low-latency External Bus Interface (EBI) for seamless connection to whatever configuration of off-chip memories and memory-mapped peripherals is required by the application. The EBI incorporates controllers for synchronous DRAM (SDRAM), Burst Flash and Static memories and features specific circuitry facilitating the interface for NAND Flash/Smart Media and Compact Flash. The Advanced Interrupt Controller (AIC) enhances the interrupt handling performance of the ARM920T processor by providing multiple vectored, prioritized interrupt sources and reducing the time taken to transfer to an interrupt handler.

2.4 Mass Storage Device

The MSD(Mass Storage Device) class defines how devices such as a hard disk, a USB floppy disk drive or a disk-on key shall operate on the USB.**[(1)]** These devices are referred to as mass storage devices, since they usually offer a high storage capacity. When plugged to a PC, a device complying to the MSD specification is accessed like any other disk on the system. Many devices use the MSD class in various ways. The simplest use is for disk-on-keys, which offer a portable storage with a high capacity compared to traditional floppy disks. External USB hard-drives are also common; they enable quick and easy connection to any system.**[(4)]**

2.5 Conclusion

ARM CPUs are found in all corners of consumer electronics, from portable devices (PDAs, mobile phones, media players, handheld gaming units, and calculators) to computer peripherals (hard drives, desktop routers). Therefore the advantage of using ARM processor for Network Attached is its very high speed 200-MIPS advanced 32-bit RISC processor and low electric power consumption.

CHAPTER 3

ARM Architecture

The ARM architecture, is the most widely used 16/32-bit embedded RISC processor. It is an open architecture that provides unparalleled levels of compatibility and design reusability, combined with superior performance per watt and code density. The industry-proven Thumb instruction set is an extension to the ARM architecture. The Thumb instruction set requires only a 16-bit wide system data bus, thereby using less power, offering a smaller footprint, and reducing overall system cost. The Thumb instruction set features a subset of the most commonly used 32-bit ARM instructions, which have been compressed into 16-bit wide codes to provide excellent code densities. On execution, these 16-bit instructions are decompressed transparently and in real-time to full 32-bit instructions without performance loss.**[(8)]** The Thumb-2 core technology instruction set builds on the success of Thumb, to in-crease the power of ARM microprocessor cores available to developers of low cost, high performance systems. Thumb-2 technology provides enhanced levels of performance, energy efficiency, and code density for a wide range of embedded applications.

ARM has enhanced many cores by extending the instruction set to include 16-bit and 32-bit arithmetic capabilities. These DSP-enhanced cores enable products that re-quire a mixture of DSP and control functionality to be

3.2 AT91RM9200

The AT91RM9200 is a complete system-on-chip built around the ARM920T ARM Thumb processor. It incorporates a rich set of system and application peripherals and standard interfaces in order to provide a single-chip solution for a wide range of compute-intensive applications that require maximum functionality at minimum power consumption at lowest cost.[(9)]

The AT91RM9200 incorporates a high-speed on-chip SRAM workspace, and a low-latency External Bus Interface (EBI)for seamless connection to whatever configuration of off-chip memories and memory-mapped peripherals is required by the application. The EBI incorporates controllers for synchronous DRAM (SDRAM), Burst Flash and Static memories and features specific circuitry facilitating the interface for NAND Flash/ SmartMedia and Compact Flash. The Advanced Interrupt Controller (AIC) enhances the interrupt handling performance of the ARM920T processor by providing multiple vectored, prioritized interrupt sources and reducing the time taken to transfer to an interrupt handler.

The Peripheral DMA Controller (PDC) provides DMA channels for all the serial peripherals, enabling them to transfer data to or from on-and off-chip memories with-out processor intervention. This reduces the processor overhead when dealing with transfers of continuous data streams. The AT91RM9200 benefits from a new generation of PDC which includes dual pointers that simplify significantly buffer chaining. The set of Parallel I/O (PIO) controllers multiplex the peripheral input/output lines with general purpose data I/Os for maximum flexibility in device configuration. An input change interrupt, open drain capability and programmable pull-up resistor is included on each line. The Power Management Controller (PMC) keeps system power consumption to a minimum by selectively enabling/disabling the processor and various peripherals under

software control. It uses an enhanced clock generator to provide a selection of clock signals including a slow clock (32 kHz) to optimize power consumption and performance at all times. The AT91RM9200 integrates a wide range of standard interfaces including Ethernet 10/100 Base-T Media Access Controller (MAC), which provides connection to a extensive range of external peripheral devices and a widely used networking layer. In addition, it provides an extensive set of peripherals that operate in accordance with several industry standards, such as those used in audio, telecom, Flash Card, infrared and Smart Card applications. The block diagram for the device is given in **Fig 3.1**(taken from the technical manual for AT91RM9200 at **www.atmel.com /dyn/ resources/ prod*documents/* 1768*s.pdf***)

3.3 Conclusion

The ARM Architecture has various additional advantages as compared to various other microcontrollers which makes it suitable for various high speed application. Thereby it is now used in all fields of modern technology.

The SCSI transparent command set comprises all SCSI-related specifications, such as SCSI Primary Commands (SPC), SCSI Block Commands (SBC), and so on. A command will be issued by the host to determine exactly with which standard the device is compliant. The protocol used by the device is specified in its Interface descriptor.**[(2)]**

4.2 Transport Protocols

There are actually two different transport protocols for the MSD class:

- Control/Bulk/Interrupt (CBI) transport
- Bulk-Only Transport (BOT)

CBI has become obsolete and is being completely replaced by BOT.

4.3 Architecture

4.3.1 Interfaces

An MSD device only needs one single interface. It should display the MSD class code (08h) in the b-Interface-Class field, the corresponding data transfer protocol code in the b-Interface-Subclass field, and finally the transport protocol code in the b-Interface-Protocol field.**[(3)]**

4.3.2 Endpoints

There are three endpoints (when using the Bulk-Only Transport protocol) necessary for MSD devices. The first one is the Control endpoint 0, and is used for class-specific requests and for clearing Halt conditions on the other two endpoints. Endpoints are halted in response to errors and host bad behavior during data transfers, and the CLEAR-FEATURE request is consequently used to return them to a functional state. The other two endpoints, which are of type Bulk, are used for transferring commands and

data over the bus. There are Bulk-IN and Bulk-OUT endpoint given in Fig. 4.2

4.3.2.1 Class-specific requests

A device can feature one or more Logical Unit (LU). Each of these units will be treated as a separate disk when plugged to a computer. A device can have up to 15 logical units. The GET-MAX-LUN request is issued by the host to determine the max-imum Logical Unit Number (LUN) supported by the device. This is not equivalent to the number of LU on the device; since units are numbered starting from 0, a device with 5 LUs should report a value of 4, which will be the index of the fifth unit. Optionally, a device with only one LUN may STALL this request instead of returning a value of zero. Bulk-Only Mass Storage Reset This request is used to reset the state of the device and prepare it to receive commands and data. Note that the data toggle bits must not be altered by a RESET command; same for the Halt state of endpoints.

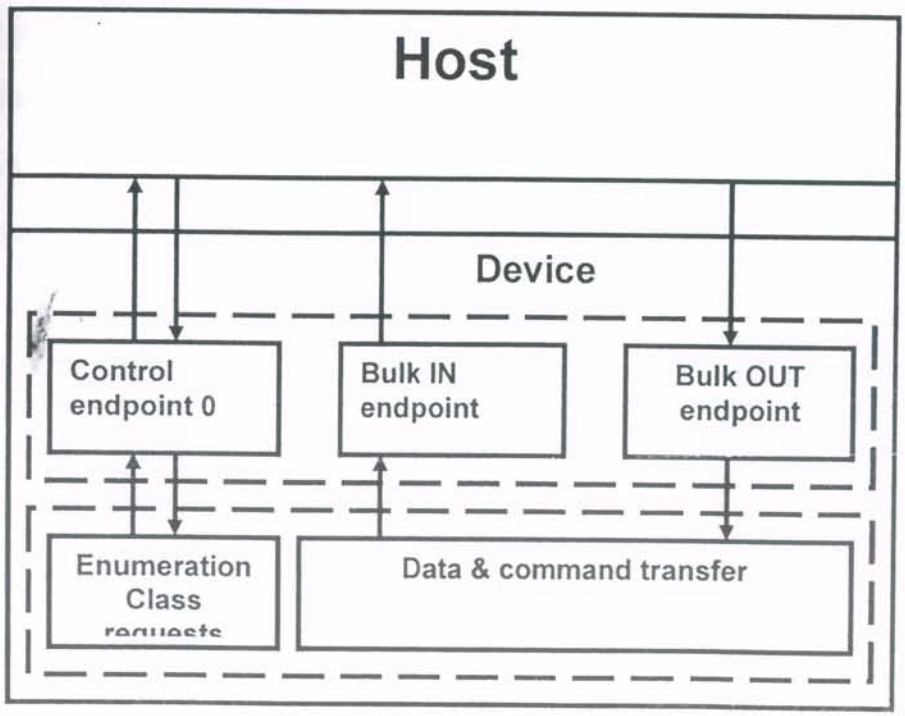


Fig. 4.2: Mass Storage Device Driver Architecture

4.3.2.2 Command/Data/Status

Each MSD transaction is divided into three steps:

- Command stage
- Data stage (optional)
- Status stage

During the command stage, a Command Block Wrapper (CBW) is transmitted by the host to the device. The CBW describes several parameters of the transaction (direction, length, LUN) and carries a variable-length command block. The command block contains data in the format defined by the transfer protocol used by the device. After the device has received and interpreted the command, an optional data stage may take place if the command requires it. During this step, data is transferred either to or from the device depending on the command, in several IN/OUT transfers. Once the data stage is complete, the host issues a final IN request on the Bulk-IN endpoint of the device to request the Command Status Wrapper (CSW). The CSW is used to report correct or incorrect execution of the command, as well as indicating the length of remaining data that has not been transferred. These steps are all performed on the two Bulk endpoints, and do not involve Control endpoint 0 at all.

4.3.2.3 Reset-Recovery

When severe errors occur during command or data transfers , the device must halt both Bulk endpoints and wait for a Reset Recovery procedure. The Reset Recovery sequence goes as follows:

- The host issues a Bulk-Only Mass Storage Reset request
- The host issues two CLEAR-FEATURE requests to unhalt each endpoint

4.3.3 Host Drivers

Almost all operating systems now provide a generic driver for the MSD class. However, the set of supported data transfer protocols may vary. For example, Microsoft Windows does not support the Reduced Block Command set.

4.4 Mass Storage SCSI Disk

This section describes how to implement a USB disk by using the MSD class with the SCSI transparent command set and the AT91 USB Framework. The application uses the internal flash of the chip as its storage medium.[(2)]

4.4.1 Architecture

The AT91 USB Framework application architecture is shown in **Fig.4.3**

4.4.2 Descriptors

There are no class-specific descriptors for a device using the MSD class with the Bulk-only transport protocol. Therefore standard descriptors are used.

(a) Device Descriptor

Since the MSD class code is only specified at the Interface level, the very basic Device descriptor are used.

(b) Configuration Descriptor

Since one interface is required by the MSD specification, this must be specified in the Configuration descriptor. When the Configuration descriptor is requested by the host (by using the GET-DESCRIPTOR command), the device must also send all the related descriptors, i.e., Interface, Endpoint and Class-Specific descriptors. It is convenient to create a single packed structure to hold all this data, for sending everything in one chunk. A Sbot configuration descriptor structure has been declared to that end.

(C) Interface Descriptor

The Interface descriptor must indicate the following features:

- Mass Storage Device class code (08h) in the b-Interface-Class field
- Data transport protocol code in the b-Interface-Subclass field
- Bulk-Only Transport protocol code (50h) in the b-Interface-Protocol field

We are using the SCSI transparent command set. This is the most appropriate setting for a Flash device, given that the RBC command set is not supported by Microsoft Windows.

sequence. If it is, then CLEAR-FEATURE requests to unhalt a Bulk endpoint must be discarded.

(b) Get-Max-LUN

The first request issued by the host right after the enumeration phase is a GET-MAX-LUN request. It enables it to discover how many different logical units the device has; each of these LUNs can then be queried in turn by the host when needed. After the request is received by the device, it should return one byte of data indicating the maximum Logical Unit Number (LUN). It is equal to the number of LUNs used by the device minus one. For example, a device with three LUNs shall return a GET-MAX-LUN value of two. Sending this byte is done by calling the USB-Write method on Control endpoint 0. The data is held in a permanent buffer (since the transfer is asynchronous); A dedicated field is used in the driver structure (S-bot) to store this value. The Mass storage Bulk-Only Transport specification defines strict requirements on the wValue, wIndex and wLength fields. In this case, there is only one interface, so the wIndex field has a value of zero. A request which does not comply to these requirements must be STALLed.

(c) Bulk-Only Mass Storage Reset

The host issues RESET requests to return the MSD driver of the device to its initial state, i.e., ready to receive a new command. However, this request does not impact the USB controller state; in particular, endpoints must not be reset. This means the data toggle bit must not be altered, and Halted endpoint must not be returned to a normal state. After processing the reset, the device must return a Zero-Length Packet (ZLP) to acknowledge the SETUP transfer. Like GET-MAX-LUN, this request is

issued with specific parameters. A request which does not have valid values in its field must be acknowledged with a STALL handshake from the device. The handler for this request must return the state machine to its initial state.

4.4.4 State Machine

(a) Rationale

A state machine is necessary for non-blocking operation of the driver. There are three steps when processing a command:

- Reception of the CBW
- Processing of the command (with data transfers if required)
- Emission of the CSW

Without a state machine, the program execution would be stopped at each step to wait for transfers completion or command processing. For example, reception of a CBW does not always happen immediately (the host does not have to issue commands regularly) and can block the system for a long time.

(b) States

Apart from the three states corresponding to the command processing flow CBW, command processing and CSW), two more can be identified. The reception/emission of CBW/CSW will be broken down into two different states: the first state is used to issue the read/write operation, while the second one waits for the transfer to finish. This is done by monitoring a "transfer complete" flag which is set using a callback function. In addition, some commands can be quite complicated to process: they may require several consecutive data transfers mixed with media access. Each command thus has its own second tier state machine. During execution of a

command, the main state machine remains in the "processing" state, and proceeds to the next one (CSW emission) only when the command is complete.

A single function, named BOT-State-Machine, is provided by the driver. It must be called regularly during the program execution.

(i) **BOT-STATE-READ-CBW** : This state is used to start the reception of a new Command Block Wrapper.

The result code of the function is checked for any error; the USB-STATUS-SUCCESS code indicates that the transfer has been successfully started.

A callback function to invoke when the transfer is complete is provided to the USB-Read method, to update a transfer status structure. This structure indicates the transfer completion, the returned result code and the number of transferred and remaining bytes.

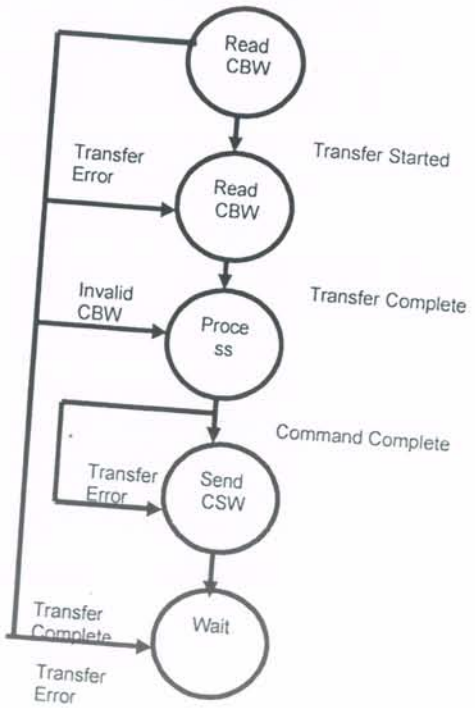


Fig. 4.5: MSD Drive State Machine

(ii) **BOT-STATE-WAIT-CBW** The first step here is to monitor the b-Semaphore field of the transfer status structure this will enable detection of the transfer end. Please note that this field must be declared as volatile in C, or accesses to it might get optimized by the compiler; this can result in endless loops. If the transfer is complete, then the result code is checked to see if there was an error. If the operation is successful, the state machine can proceed to command processing. Otherwise, it returns to the READ-CBW state.

(iii) **BOT-STATE-PROCESS-CBW** Once the CBW has been received, its validity must be checked. A CBW is not valid if:

- It has not been received right after a CSW was sent or a reset occurred or
- It is not exactly 31 bytes long or
- Its signature field is not equal to 43425355h

The state machine prevents the first case from happening, so only the two other cases have to be verified. The number of bytes transferred during a USB-Read operation is passed as an argument to the callback function, if one has been specified. If the CBW is not valid, then the device must immediately halt both Bulk endpoints, to STALL further traffic from the host. In addition, it should stay in this state until a Reset Recovery is performed by the host. This is done by setting the is-Wait-Reset-Recovery flag in the driver structure. Finally, the CSW status is set to report an error, and the state machine is returned to BOT-STATE-READ-CBW. Otherwise, if the CBW is correct, then the command can be processed. The CBW tag must be copied regardless of the validity of the CBW.

(iv) **BOT-STATE-SEND-CSW** This state is similar to BOT-STATE-READ-CBW, ex-cept that a write operation is performed instead of a read and the CSW is sent, not the CBW. The same callback function is used to fill the transfer structure.

(v) **BOT-STATE-WAIT-CSW** Again, this state is very similar to BOT-STATE-WAIT-CBW. The only difference is that the state machine is set to BOT-STATE-READ-CBW regardless of the operation result code:

4.4.5 Media

(a) Architecture

Media access is done using a three-level abstraction, as shown in **Fig. 4.6**

At the bottom level is the specific driver for each media type. In the middle, a structure named S-media is used to hide which specific driver a media instance is using. This enables transparent use of any media driver once it has been initialized. Finally, a LUN abstraction is made over the media structure to allow multiple partitions over one media. This also makes it possible to place the LUN at any address and use any block size. When performing a write or read operation on a LUN, it forwards the operation to the underlying media while translating it to the correct address and length.

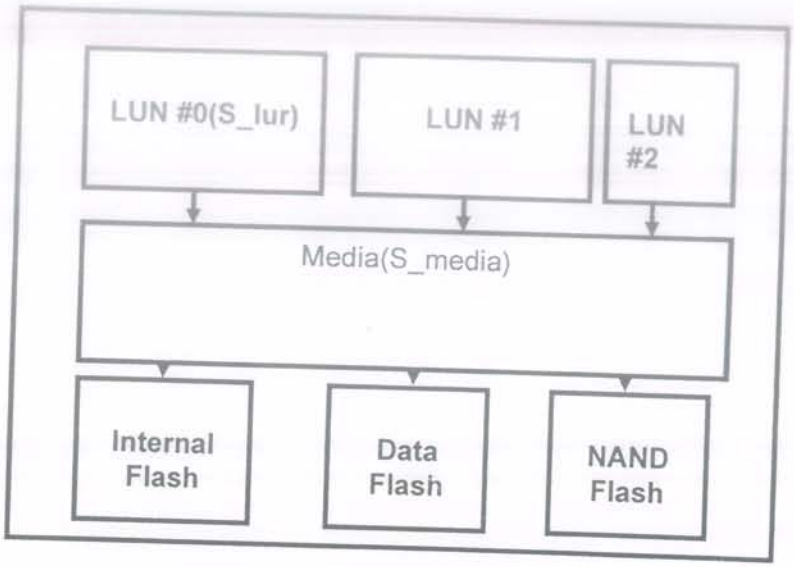


Figure 4.6: Media Architecture

(b) Drivers

Requirements: As required by the S-media structure above, a media driver must provide several functions for:

- Reading data from the media
- Writing data on the media
- Handling interrupts on the media

The last function may be empty if the media does not require interrupts for asynchronous operation, or if synchronous operation produces an acceptable delay. In addition, it should also define a function for initializing a S-media structure with the correct values, as well as perform the necessary step for the media to be useable.

Internal Flash Driver This describes how the internal Flash driver used in the ex-ample software has been implemented. This is done by detailing the content of the five necessary functions, as defined previously.

(i) Initialization Function(FLA-Init): The first step of the initialization function is to setup the values of the S-media structure. The 4 function

pointers are set to the corresponding Flash methods; Flash parameters (base address, size, physical interface) are the ones defined for the chip in the Lib v3. Since Atmel AT91RM9200 chips have only one internal Flash bank, the peripheral interface value is hard-coded; it can however be easily passed as an argument of the initialization function. The transfer status structure is initialized with default values. The next stage is to initialize the media so it is functional. In the case of the internal Flash, there is nothing to do; the only required operation is to configure the Mode register of the Embedded Flash Controller with the correct Microsecond Cycle Number. This value can be inferred given the frequency of the master clock.

(ii) **Read Function (FLA-Read)**: Reading from the internal Flash is as simple as reading from the SRAM, given the address to access. The function should verify that the target address and length are valid and make sure that the media is not busy (i.e., an-other operation is already in progress). If everything is correct, then the specified number of bytes can be read from the Flash into the provided buffer. If a callback has been specified, it is invoked when the transfer finishes.

(iii) **Write Function (FLA-Write):** Per-forming a write operation is a more complex than reading. First of all, since the internal Flash on most AT91SAM chips is single-plane, it cannot be read and written simultaneously. This means the software must actually run from the SRAM while writing the Flash. The simplest method is to copy the entire software to SRAM, set the remap bit in the memory controller and run from there. The actual Write function should first check the parameters. Since the Embedded Flash Controller only supports memory writes of exactly one double-word (4 bytes), any access with an address not aligned on a 4-bytes

boundary or a length not multiple of 4 must be rejected. Of course, the data must not be bigger than the size of the media and the media must be in the ready state. If input parameters are correct, actual writing can begin. The internal Flash is divided into chunks of bytes called pages, which are generally between 64 and 256 bytes in length. A Write operation must be done one page at a time, and the page must be cleared before writing. This can lead to problems if only part of the page is altered; in this case, the remaining data can simply be copied before writing. A second function is used to write one page of data at a time; the Write method starts the whole transfer by writing the first page and initializing the transfer status. Each page write takes some time to complete, so the interrupt on the Flash Ready status bit is used to avoid blocking the system while waiting for the operation.

(iv) **Interrupt Handler Function(FLA-Handler):** This interrupt handler is invoked when a Flash page has been written. It should check the transfer status to see if there is more data to write; if there is, then it calls the page write function again with new parameters. Otherwise, the transfer is finished and the callback function invoked if one has been defined. In any case, the Flash Ready interrupt must be disabled to avoid continuous interruption, and re-enabled when the page write operation starts.

4.4.6 SCSI Commands

The code uses SCSI commands with the MSD class, since this is the most appropriate setting for a Flash device.

SCSI commands use the big-endian format for storing word-and double word-sized data. This means the Most Significant Bit (MSB) is stored

(c) Move to SDRAM location 0x00200000 and flash program the zimage.txt file to location 10080000 with data size of 0x00142000

(d) Set flashprog.txt to SDRAM location 0x00200000Set inird.txt to SDRAM location 0x20900000

(e) Move to SDRAM location 0x00200000 and flash program the file initrd.txt to location 10200000 with data size of 0x001E3000

(f) Set flashprog.txt to SDRAM location 0x00200000

(g) Set moniprog to SDRAM location 0x20900000

(h) Move to SDRAM location 0x00200000 and flash program the file moniprog.txt to location 10000000 with data size of 0x0010AB90

Now the location of various images on the flash is as given in **Fig. 5.2** .

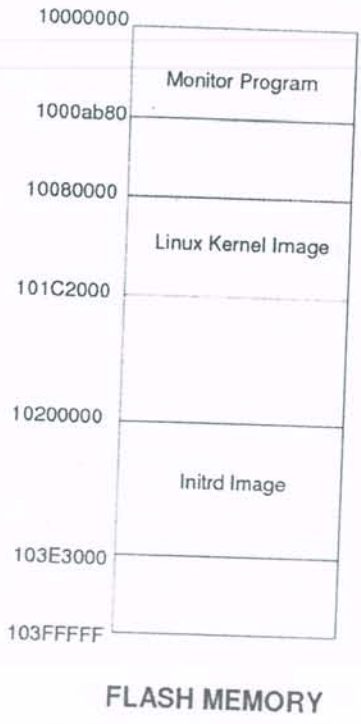


Figure 5.2: Flash Location of Linux image and Initrd image

the Samba server can easily be moved to a proprietary "big iron" Unix mainframe, which can outperform Windows running on a PC many times.

Samba suite revolves around a pair of Unix daemons that provide shared resources—called *shares* or *services*—to SMB clients on the network. These are:

Smbd : A daemon that handles file and printer sharing and provides authentication and authorization for SMB clients.

Nmbd: A daemon that supports NetBIOS Name Service and WINS, which is Microsoft's implementation of a NetBIOS Name Server (NBNS). It also assists with network browsing.

6.2 Procedure for Cross Compiling and Configuring Samba for ARM Platform

(a) Get source Samba-3.0.24 (from debian maybe)

(b) Do following:

```
export LD_LIBARY_PATH=""
export CPPFLAGS=""
export LDFLAGS="-static"
export CC=/usr/local/arm/3.4.1/bin/arm-linux-gcc
export CFLAGS="-march=armv4 -mtune=arm9tdmi"

cd source
```

(c) Then run configure

```
./configure --without-krb5 --without-ldap --without-ads --
prefix=/usr/local/arm/samba
--host=i686 --target=arm-linux --build=arm-linux --disable-cups --without-
swat
```

(d) Then run make
Will show error when linking smbd
Need to do the following manually

i. In the Makefile, append following to LIBS
printing/pcap.o printing/print_generic.o printing/load.o printing/lpq_parse.o
ii. Need to build these manually. ie, do as:
#make printing/pcap.o

.. and so on ..

(e) With this, smbd should link fully. But some other error later will occur. Seems ok. Keep going by giving:
#make -k

(f) Major binaries are now made and in source/bin/.

(j) But need to fix the getpwnam bug (?) as:

Update file lib/util_pw.c

.. at the beginning ...

```
static void str_cpy(char* target, const char* source)
{
        int i=0;

        while(source[i] != 0)
                target[i] = source[i++];
        target[i] = 0;
}
```

.. modify within function >>>struct passwd *getpwnam_alloc(TALLOC_CTX *mem_ctx, const char *name)<<< ...

```
        /*temp = sys_getpwnam(name);

        if (!temp) */
        {
#if 0
                if (errno == ENOMEM) {
                        /* what now? */
                }
#endif
                char *target = (char*)malloc(20*sizeof(char));
                  temp->pw_name = target;
                  str_cpy(temp->pw_name, "root");

                target = (char*)malloc(20*sizeof(char));
                  temp->pw_passwd = target;
                  str_cpy(temp->pw_passwd, "");

                  temp->pw_uid = 0;
                  temp->pw_gid = 0;

                target = (char*)malloc(20*sizeof(char));
                  temp->pw_gecos = target;
                  str_cpy(temp->pw_gecos, "root");

                target = (char*)malloc(20*sizeof(char));
                  temp->pw_dir = target;
```

```
                str_cpy(temp->pw_dir, "/usb");
              target = (char*)malloc(20*sizeof(char));
                temp->pw_shell = target;
                str_cpy(temp->pw_shell, "/bin/busybox");
                /* return NULL; */
            }
```

(k) Now recompile as usual using:

#make -k

(l) Make the board up, mount usb pendrive to /usb. Do

#mkdir /usr/local

#ln -s /usb/samba /usr/local/samba

(m) Make individual directories within /usb/samba (refer Using Samba - O'Reilly page 48)

Make - bin, libexec, share, lib, include, info, man

(n) Copy all binaries in Samba-3.0.24/source/bin (on host) to /usb/samba/bin

(o) Add following smb.con file (Note: The board seems to have a 8+3 file name restriction)

.. snip ...

```
#======= Global Settings==============================
[global]

# workgroup = NT-Domain-Name or Workgroup-Name, eg: LINUX2
  workgroup = iitm

# server string is the equivalent of the NT Description field
  server string = Samba Server

# user level security. See the HOWTO Collection for details.
  security = user

  socket options = TCP_NODELAY

  dns proxy = no

  interfaces = 10.7.2.252/255.255.255.0
  smb passwd file = /usr/local/samba/private/smbpassw

#========== Share Definitions ==============================
[homes]
  comment = Home Directories
  browseable = no
  writable = yes
# A publicly accessible directory, but read only, except for people in
# the "staff" group
[public]
  comment = Public Stuff
```

path = /usb
public = yes
writable = yes

.. snip ..

(p) Add a file smbpassw to /usb/samba/private (TODO: Check if this is really needed)

.. snip ..

root:0:AAD3B435B51404EEAAD3B435B51404EE:31D6CFE0D16AE931B73C59D7E0C089C0:[U]:LCT-0000124C:

.. snip ..

Then run (within bin) ./smbpassw -c ../lib/smb.con -D 10 -a root
And check if it goes through with no "user not found" error. It is ok if it says user root already exists.

(q) NOTE: Logging information for the samba daemon is in /usb/samba/var/log.smb

(r) Run samba on board:
#./smbd --configfile=../lib/smb.con -d 10

(s) Check by running ps
#ps

(t) NOTE: You can check shares by using smbclient (say by running a samba server on the *host* m/c)
#./smbclient -U% -L 10.7.2.253

(u) Connect from Windows and check (go via Search->computer option)

6.3 Conclusion

The binary files generated are stored in the Mass Storing Device(ie Hard Disk). The flash memory on the board is limited to 4MB. Out of which more than 3 MB is being used for Linux Image and monitor program. The Samba binaries (after disabling most of the extra features) comes out to be around 16 MB. Therefore could not be programmed to the flash. Thereby it is stored in the hard disk and is being soft linked to the Linux environment.

CHAPTER 7

Configuring Apache for Web access

The Apache HTTP Server, commonly referred to simply as Apache, is a web server. When first released, Apache was the only viable alternative to the Netscape Communications Corporation web server (currently known as Sun Java System Web Server). It has since evolved to rival other Unix-based web servers in terms of functionality and performance. Since April 1996 Apache has been the most popular HTTP server on the World Wide Web; as of March 2007 Apache serves 58% of all websites.

Apache is developed and maintained by an open community of developers under the auspices of the Apache Software Foundation. The application is available for a wide variety of operating systems including Microsoft Windows, Novell NetWare and Unix-like operating systems such as Linux and Mac OS X. Released under the Apache License, Apache is free and open source software.

Apache is primarily used to serve both static content and dynamic Web pages on the World Wide Web. Many web applications are designed expecting the environment and features that Apache provides.

Apache is the web server component of the popular LAMP web server application stack, alongside Linux, MySQL, and the PHP/Perl/Python programming languages.

Apache is used for many other tasks where content needs to be made available in a secure and reliable way. One example is sharing files from a personal computer over the Internet. A user who has Apache installed on their desktop can put arbitrary files in the Apache's document root which can then be shared.

7.1 Apache webserver has been used for web hosting with Perl module for scripting and controlling various resources (ie. session manager, cleaning and making of user folders etc). "Apache" web server is already configured and it does support Perl scripts. Perl script is used to generate HTML pages. The HTML pages are internally linked to each other. There are three important resources used to host the website and control various resources.

(a) **Template** : The Apache-Template module provides a simple interface to the Template Toolkit from Apache/mod_perl. The Template Toolkit is a fast, powerful and extensible template processing system written in Perl. It implements a general purpose template language which allows to clearly separate application logic, data and presentation elements. This has been used to activate various resources and password authentication.

(b) **Common Gateway Interface (CGI)** : It is a standard for interfacing external applications with information servers, such as HTTP or Web servers. A plain HTML document that the Web daemon **retrieves** is **static**, which means it

exists in a constant state: a text file that doesn't change. A CGI program, on the other hand, is **executed** in real-time, so that it can output **dynamic** information.

For example, when we want to "hook up" our Unix database to the World Wide Web, to allow people from all over the world to query it. Basically, we need to create a CGI program that the Web daemon will execute to transmit information to the database engine, and receive the results back again and display them to the client.

CGI programs reside in a special directory, so that the Web server knows to execute the program rather than just display it to the browser. This directory is usually under direct control of the webmaster, prohibiting the average user from creating CGI programs. Source code for some of the CGI programs is stored in the /cgi-bin directory.

(b) **SAMBA CLIENT PARSER** : Various calls are used by Apache to communicate with Samba server.

7.2 Conclusion

Apache web server has been used for web hosting with Perl module for scripting and controlling various resources. This has been hosted on a different desktop. It communicates with the Samba Server and carries out various functions (ie reading, writing, delete etc. of files and folders). It can be hosted on the internet also.

CHAPTER 8

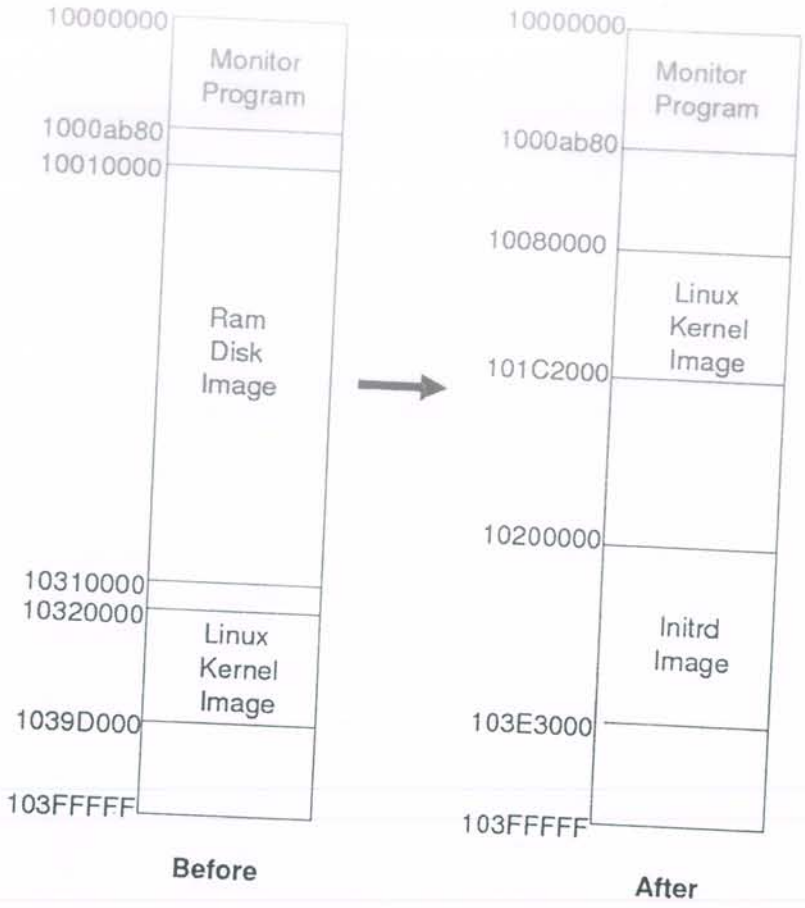
Conclusion

The Network Attached Storage device has been implemented on ARM based microcontroller AT91RM9200. The first step to implement was to enable the detection of Mass Storage Device by the board. This has been done using two methods:

- **As an Application**: Code is written in C. Compiled using the Windows CE/ Visual C (nmake) utility. It is then converted to ASCII form using the UUEncode utility and run on the board as an application. However the application needs to be loaded to the SDRAM location whenever the board boots up. This is done using the serial link between the PC and the AT91RM9200 board. The solution to the problem has been obtained by second method of providing Linux Kernel level support for the USB device.

- **Linux Kernel level support**: In this method the USB code for USB MSD for the said device has been patched with the latest kernel source code. A kernel image is prepared and then ported to the specified Flash location.

After the images have been ported to the Flash the changed flash location diagram is given in **Fig. 8.1**



FLASH MEMORY

Figure 8.1: Flash Memory

The following inferences have been drawn from the work done for developing USB Mass Storage Device Driver:

(a) Instead of running application separately for various resources, it is easier and better to implement the Linux Kernel Level support. Codes can either be written or downloaded for various applications and patched to the Linux source code.

(b) This Linux kernel can be compiled using the cross compiler and an image can be prepared. It can then be ported to the Flash Memory of the AT91RM9200 board. Various arm based mobile phones, PDAs, Digital diaries etc are made in this manner using Linux core. It may further be used to implement various ARM based devices like phones, PDAs etc.

8.1 Implementing Samba Server:

The Samba Server-Client configuration is implemented by cross compiling the Samba Server code for the ARM platform. The binary files generated are stored in the Mass Storing Device (ie Hard Disk) and are being soft linked to the Linux environment. It can therefore be accessed from anywhere in the network and following services are achievable:

- Share one or more directory trees
- Share one or more Distributed filesystem (Dfs) trees
- Assist clients with network browsing
- Authenticate clients logging onto a Windows domain
- Provide or assist with Windows Internet Name Service (WINS) name-server resolution

8.2 Implementing Apache Server: Apache webserver has been used for web hosting with Perl module for scripting and controlling various resources. Perl script is used to generate HTML pages. The HTML pages are internally linked to each other. The web server used controls all password authentication and provides all the resources required by the user.

8.3 Conclusion

The present implementation has been used for Network Attached storage device with web access. It can further be modified to the following.

(a) While compiling Samba for the ARM platform, many resources have been ignored due to the available capacity of the hard disk and some implementation difficulties. Theses resources may further be added as per the requirement.

(b) Password authentication has been provided through the Apache web server. This can be done from the Samba server also.

(c) The Apache server is running on a different machine. This is to avoid any direct threat to the data stored in the board. However putting Apache to the hard disk would involve compiling of the Apache source code for the ARM environment and larger storage space. It will also involve scripting of the Perl code as per ARM environment

REFERENCES

[1] **Atmel Corp**., AT91 USB Framework, lit. num. 6269; Published in.1991 **Website** : *www.atmel.com/dyn/resources/prod_documents/doc6283.pdf.*

[2] **Paul Suhler**,T10, SCSI Block Commands - 3 (SBC-3), Revision 7, September 22, 2006: **Website**: *www.t10.org/sy0605.htm* .

[3] **Ralph O**., T10, SCSI Primary Commands - 4 (SPC-4), Revision 6, July 18, 2006: **Website:** *www.t10.org/ftp/t10/drafts/spc4/spc4r10.pdf.*

[4] **USB Forum**, Mass Storage Class Bulk-Only Transport, Revision 1.0.1999: **Website**: *www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf.*

[5] **USB Forum,** Mass Storage Class Compliance Test Specification, Revision 0.9a.1991: **Website**:*www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf.*

[6] **Andrew Sloss** ARM System Developer's Guide .2005:Published by SciTech Media.,Inc.

[7] **USB Forum,** Mass Storage Class Specification Overview, Revision1.2.2003: **Website**: *www.usb.org/developers/devclass_docs/MSC-compliance-0_9a.pdf.*

[8] **ARM-FORUM** ,**Website**:*www.arm.com.*

[9] **ATMEL-FORUM** ,.2003: **Website** :*www.atmel.com*

[10] **USB Forum** Mass Storage Class Compliance Test Specification, Revision 0.9a.2005: **Website**: *www.usb.org/ developers/devclassdocs*

[11] **USB Forum** Mass Storage Class Specification for USB : **Website**: *www.usb.org/developers/devclass-docs/usb-msc-overview-1.2.pdf.*

[12] **Steve Furber**, ARM System-on-chip Architecture.2005:Published by published by Addison Wesley.

Appendix A

Changes in the MSD Device Driver Application Code

The code for USB Mass Storage Device has been taken from ATMEL website **www.atmel.com /dyn /resource /prod-documents /doc6283.pdf**. This application is written for AT91SAM7S64 board. Some modifications in the code has been done for it to work on AT91RM9200 board. These modifications are below:

The software example provided along with this application note is divided into following files:

- **msd.h**: header file with generic Mass Storage Device definitions
- **bot_driver.h**: header file with definitions for the Bulk-Only Transport driver
- **bot_driver.c**: source file for the BOT driver
- **media.h**: header file with media definitions
- **media.c**: source file for media interrupt handling
- **media_flash.h**: header file for the internal flash media driver
- **media_flash.c**: source file for the internal flash media driver
- **lun.h**: header file for LUN definitions
- **lun.c**: source file for LUN usage
- **sbc.h**: header file with SCSI SBC-3 and SPC-4 definitions
- **sbc_commands.h**: header file for SBC functions definitions
- **sbc_commands.c**: source file for SBC functions implementation
- **bot_example.c**: source file for the BOT example application

Following are the changes done to work on AT91RM9200 board:

(a) **board.h**

```
//------------------------------------------------------------------------------
//      Internal functions
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//! \brief  Sets the correct number of wait states in the flash controller
//------------------------------------------------------------------------------
extern inline void BRD_SetFlashWaitStates()
{
#if defined(AT91C_BASE_FLASH)
  SET(AT91C_BASE_FLASH->FLA_FMR, AT91C_FLASH_WAIT_STATES);
#endif
}
```

changes made:-

```
//------------------------------------------------------------------------------
//      Internal functions
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//! \brief  Sets the correct number of wait states in the flash controller
//------------------------------------------------------------------------------
//extern inline void BRD_SetFlashWaitStates()
//{
//#if defined(AT91C_BASE_FLASH)
//   SET(AT91C_BASE_FLASH->FLA_FMR, AT91C_FLASH_WAIT_STATES);   //Santosh
//#endif
//}
```

(As there is no base flash in AT91RM9200)

(b) **Makefile**:

THUMB_cc has been changed from gcc to armcc compiler.

```
#-------------------------------------------------------------------------------
#       Compilation flags
#-------------------------------------------------------------------------------

CCFLAGS = -apcs /interwork -gtp -Wb+avnglp -D__APCS_INTERWORK\
        $(OPTIMIZATION) -D$(TARGET) -D$(BOARD) -I$(LIB) -I./
```

changes:

```
#-------------------------------------------------------------------------------
#       Compilation flags
#-------------------------------------------------------------------------------

CCFLAGS = --device=DARMATS9 --apcs  /interwork -W -IC:\KeilArm122006\ARM\RV30\INC -
D__APCS_INTERWORK\
        $(OPTIMIZATION) -D$(TARGET) -D$(BOARD) -I$(LIB) -I./
```

(The program has been written to work on Windows CE environment. We are using the Keil armcc compiler along with the nmake utility.)

(c) **AT91RM9200.h**

```
// ****************************************************************************
//             MEMORY MAPPING DEFINITIONS FOR AT91RM9200
// ****************************************************************************
// IROM
#define AT91C_IROM    ((char *)         0x00100000) // Internal ROM base address
#define AT91C_IROM_SIZE       ((unsigned int) 0x00020000) // Internal ROM size in byte (128
Kbytes)

#endif
```

changes:

```
// ****************************************************************************
//             MEMORY MAPPING DEFINITIONS FOR AT91RM9200
// ****************************************************************************
// IROM
//#define AT91C_IROM   ((char *)         0x00100000) // Internal ROM base address
```

```
//#define AT91C_IROM_SIZE     ((unsigned int) 0x00020000) // Internal ROM size in byte (128
Kbytes)

#define AT91C_IFLASH ((char *)          0x00100000) // Internal FLASH base address
#define AT91C_IFLASH_SIZE    ((unsigned int) 0x00020000) // Internal FLASH size in byte (64
Kbytes)
#define AT91C_IFLASH_PAGE_SIZE     ((unsigned int) 128) // Internal FLASH Page Size: 128
bytes
#define AT91C_IFLASH_LOCK_REGION_SIZE  ((unsigned int) 4096) // Internal FLASH Lock
Region Size: 4 Kbytes
#define AT91C_IFLASH_NB_OF_PAGES          ((unsigned int) 512) // Internal FLASH Number
of Pages: 512 bytes
#define AT91C_IFLASH_NB_OF_LOCK_BITS    ((unsigned int) 16) // Internal FLASH Number of
Lock Bi

#endif
```

(These definitions are required for the application to be written on the 16MB flash provided)