Networked Outdoor Media Player

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

In

Electronics & Communication Engineering

(Embedded Systems)

By

Niral S. Soni

(12MECE21)

Electronics & Communication Engineering
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481

May 2014

Networked Outdoor Media Player

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

In

Electronics & Communication Engineering Branch

(Embedded System)

By

Niral S. Soni

(12MECE21)

Under the Guidance of

Project Guide

Prof.Dhaval Shah
Asst. Prof. E & C
Institute of Technology,
Nirma University,Ahmedabad

Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
May 2014

Declaration

This is to certify that

1. The thesis comprises my original work towards the degree of Master of Technology in Embedded Systems at Nirma University and has not been submitted elsewhere for a degree.

2. Due acknowledgement has been made in the text to all other material used.

Niral S. Soni

Certificate

This is to certify that the Major Project entitled **"Networked Outdoor Media Player"** submitted by **Niralkumar Sharadkumar Soni (12MECE21)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge,haven't been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Ahmedabad

Prof.Dhaval G. Shah
Internal Project Guide

Dr. N.P. Gajjar
Program Coordinator

Dr. P.N.Tekwani
Head of EE Dept.

Dr. K. Kotecha
Director, IT

Acknowledgement

I would like to express my gratitude and sincere thanks to Dr. P.N.Tekwani, Head of Electrical Engineering Department, and Dr. N.P.Gajjar, PG Coordinator of M.Tech Embedded Systems program for allowing me to undertake this thesis work and for his guidelines during the review process.

I would first of all like to offer thanks to **Prof.Dhaval Shah** , Guide and Assistant Professor M.Tech. Electronics and Communication, Institute of Technology, Nirma University, Ahmedabad whose keen interest and excellent knowledge base helped me to finalize the topic of the dissertation work. His constant support and interest in the subject equipped me with a great understanding of different aspects of the required architecture for the project work. He has shown keen interest in this dissertation work right from beginning and has been a great motivating factor in outlining the flow of my work.

I am thankful to Nirma University for providing all kind of required resources. I would like to thank The Almighty, my family for supporting and encouraging me in all possible ways. I would also like to thank all my friends who have directly or indirectly helped in making this dissertation work successful.

-Niral S. Soni
(12MECE21)

Abstract

Digital Signs are used in wayfinding,exhibitions, public installations, marketing and outdoor advertising. Digital signage displays use content management systems and digital media distribution systems which can either be run from personal computers and servers or regional/national media hosting providers.

In this thesis, the approach of the media player content management through the networking has been represented. Because of its media and graphics processing capacity (720p @ 30 fps) Beagleboard-XM has been selected for the media player module while Desktop PC has been set up as a content server.The steps of installing linux on the board and software configuration for the networking feature has been briefly described.Theoretical aspects and implementation of the libraries and *make-file* utility which are used in the project are explained.To prevent unauthorized access to the content and to the media player itself AES method of encryption has been described and timing analysis has been carried out to select the best mode of operation from the available modes of operations. The concept of parallel processing , process creation , Socket and socket programming has been described that are used in the implementation of the client-server model.

The client-server model for the content management system has been represented, In which the server process flow and the client process flow are described. A method for content management by comparing and contrasting the content of the server and client at periodic intervals has been described. A method for transferring bulk of the files at a instance from the server to client as per the process flow has been described.Lastly,an attempt has been made to write and test a start up script that automates this whole process on the system boot up.

Acronyms

HDMI High-Definition Multimedia Interface

DVI Digital Visual Interface

OS Operating System

AES Advanced Encryption Standard

DES Data Encryption Standard

POP Package On Package

OMAP Open Multimedia Application Platform

SIMD Single Instruction Multiple Data

OTG ON-The-Go

fPIC Position Independent Code

MPEG Moving Picture Expert Group

HLOS High Level Operating System

Contents

Declaration i

Certificate ii

Acknowledgement iii

Abstract iv

List of Figures ix

1 Introduction 1

1.1 Background 1

1.2 Scope of Work 3

1.3 Outline of Thesis 3

2 Beagleboard-XM Specifications and Configuration 5

2.1 BeagleboardXM overview 5

2.1.1 Processor 5

2.1.2 Memory 9

2.1.3 Power management 10

2.1.4 HS USB 2.0 OTG Port 10

2.1.5 HS USB 2.0 Host Ports 10

2.1.6 DVI-D Connector 11

2.1.7 Onboard USB HUB 11

2.1.8 LCD Header 11

2.1.9 MicroSD Connector 11
2.1.10 User Button 11
2.1.11 Indicators 12
2.1.12 Power Connector 12
2.1.13 JTAG Connector 13
2.1.14 RS232 DB9 Connector 13
2.1.15 Camera Connector 13
2.1.16 MMC3 Expansion Header 14
2.2 Porting Linux Operating System on Beagleboard-XM 14
2.2.1 Install Pre-Configured Image [11] 14
2.2.2 NetInstall Method 15
2.3 System Configuration for Static MAC and IP address 16

3 Basics of Libraries,Makefile and Shell 17
3.1 Importance of Library 17
3.2 Library Naming Conventions 18
3.3 Linux Library Types 18
3.3.1 Shared Library (.a) 18
3.3.2 Dynamically Linked "Shared Object"Libraries: (.so): 19
3.4 Makefile 22
3.5 Shell Scripting 25
3.5.1 Shell Types 26
3.5.2 Writing a Script 26
3.5.3 System run level and init.d scripts 27

4 Media File Encryption 33
4.1 Types of Encryption 33
4.2 Advanced Encryption Standard (AES) 34
4.2.1 Advantages of AES over DES 34
4.2.2 Description of Algorithm 34
4.2.3 AES Modes of Operation and Initialization vector 36
4.2.4 Implementation of Algorithm 37

4.2.5 Timing Analysis 38

5 Process and Socket Programming **39**

5.1 Process ID 39

5.2 Creating Process 40

5.2.1 Calling fork 40

5.2.2 Calling execv 40

5.3 Socket Interprocess Communication 41

5.3.1 System calls 43

6 Implementation of Client-Server model **46**

6.1 Server Process 46

6.2 Client Process 47

7 Conclusion and Future Scope **52**

7.1 Conclusion 52

7.2 Future Scope 53

Bibliography **54**

List of Figures

2.1 Beagleboard-XM 7
2.2 HighLevel Block Diagram of Beagleboard-XM 8

6.1 Server Process Flowchart 48
6.2 Process of sending files 49
6.3 Client Process 50
6.4 Client Process 51

Chapter 1

Introduction

This thesis work is related to project of **"Networked Outdoor Media Player"** which implements various features of standard Digital Media Player and extended special feature of networking of media player for the content management and administration from remote side content server. The project work includes selection of development board used for the media player module that includes microprocessor supporting high definition media content processing(decoding) and peripherals for the networking and storage component for on content storage and OS installation. The project work also describes designing and implementation of robust process flow for automatic content management of the media player module from the content server. ,encryption process for the content (Media files)to avoid unauthorized access of media files and media player. To accommodate all these features Beagleboard-xM is selected to work as a media player module and linux(Ubuntu) has been installed as the operating system of the module.

1.1 Background

Digital signs are a form of electronic display that shows television programming, menus, information, advertising and other messages. Digital signs (frequently utilizing technologies such as LCD, LED, plasma displays, or projected images to display content) can be found in both public and private environments, including retail

stores, hotels, restaurants, and corporate buildings, among st other locations.

Digital sign displays are most commonly controlled by personal computers or servers, through the use of either proprietary software or free software; this approach often allows the operator to avoid large capital outlays for the controller equipment.

Advertising using a digital sign is a form of out-of-home advertising in which video content, advertisements, and/or messages may be displayed on digital signs with a common goal of delivering targeted messages, to specific locations and/or consumers, at specific times. This is often called "digital out of home "or abbreviated as DOOH.

Since digital sign content may be both frequently and easily updated, saving the printing and/or construction costs associated with a static sign, and also because of the interactive abilities available through the accompanying employment of such real-world interfaces as embedded touch screens, movement detection and image capture devices, it has won wide acceptance in the marketplace.

Digital signs rely on a variety of hardware to deliver the content. The components of a typical digital sign installation include one or more display screens, one or more media players, and a content management server. Sometimes two or more of these components are present in a single device but typically there is a display screen, a media player, and a content management server that is connected to the media player over a network. One content management server may support multiple media players and one media player may support multiple screens. Stand-alone digital sign devices combine all three functions in one device and no network connection is needed.

1.2 Scope of Work

The experimental set up is prepared for this dissertation work includes Desktop PC set up as a content server and Beagleboard-xM configured as a media player and LCD monitor set as a display screen. A part of study includes methods of linux(Ubuntu) installation on the board and its configuration for networking.A study work includes learning of the system calls for the process creation and socket programming.AES encryption process and its implementation details has been described. The client-server model for the content management has been represented in which server process flow and client process flow are described.

1.3 Outline of Thesis

The contents of this document have been organized in a logical sequence. A brief description of the contents of each chapter is given below.

Chapter 2 Beagleboard-XM Specifications and Software Configurations: This chapter briefly describes hardware specifications of the Beagleboard-XM,block diagram of the board.Installation steps of the OS,configuration of the OS for the networking features etc are included in this chapter

Chapter 3 Basics of Library,Makefile and Shell Scripting: This chapter represents short tutorial notes on the libraries creation,makefile creation and shell scripting.

Chapter 4 Media file Encryption: This chapter covers theory and implementation details of AES encryption process used in this project work.

Chapter 4 Process and Socket Programming: This chapter describes the method for child process creation,process identification, parallel processing etc. It also describes linux's concept of "Socket ", socket structure, and basic methods of filling up socket and various system calls used in socket programming.

Chapter 5 Implementation of Client-Server model: This chapter describes process flow of the server and client modules,implementation of the server-client model for the file transfer and synchronization between server and client modules.

Chapter 6 Conclusion and Future Scope: This chapter briefly concludes the approach of Networked Outdoor Media player used in this project work and its future scopes.

Chapter 2

Beagleboard-XM Specifications and Configuration

2.1 BeagleboardXM overview

The BeagleBoardXM [7]is designed specifically to address the Open Source Community. It has been equipped with a minimum set of features to allow the user to experience the power of the processor. By utilizing standard interfaces, the BeagleBoardXM is highly extensible to add many features and interfaces.

2.1.1 Processor

The BeagleBoard-xM processor is the DM3730CBP[8] 1GHz version and comes in a .4mm pitch POP package. POP (Package on Package) is a technique where the memory is mounted on top of the processor. For this reason, when looking at the BeagleBoard, labled name DM3730CBP cannot be found.

Processor Overview The DM37x generation of high-performance, applications processors are based on the enhanced device architecture and are integrated on TI's advanced 45-nm process technology. This architecture is designed to provide best in class ARM and Graphics performance while delivering low power consumption.

Table 2.1: Beagleboard-xM Features

	Feature	
Processor	Texas Instruments Cortex A8 1GHz processor	
POP Memory	Micron 4Gb MDDR SDRAM (512MB) 200MHz	
PMIC TPS65950	Power Regulators Audio CODEC Reset USB OTG PHY	
Debug Support	14 Pin JTAG UART	GPIO Pins 3LEDs
Indicators	Power, Power Error PMU	2-User Controllable USB Power
USB Host Ports	SMSC LAN9514 Ethernet HUB 4 FS/LS/HSUp to 500ma per Port if adequate power is supplied	
Ethernet	10/100	From USB HUB
SD/MMC Connector	MicroSD	
User Interface	1-User defined button	Reset Button
Video	DVI-D	S-Video
Power Connector	USB Power	DC Power
Overvoltage Protection	Shutdown @ Over voltage	
Main Expansion Connector	Power (5V & 1.8V) McBSP I2C MMC2	UART McSPI GPIO PWM
2 LCD Connectors	Access to all of the LCD control signals plus I2C	3.3V, 5V, 1.8V
Auxiliary Expansion	MMC3	GPIO,ADC,HDQ

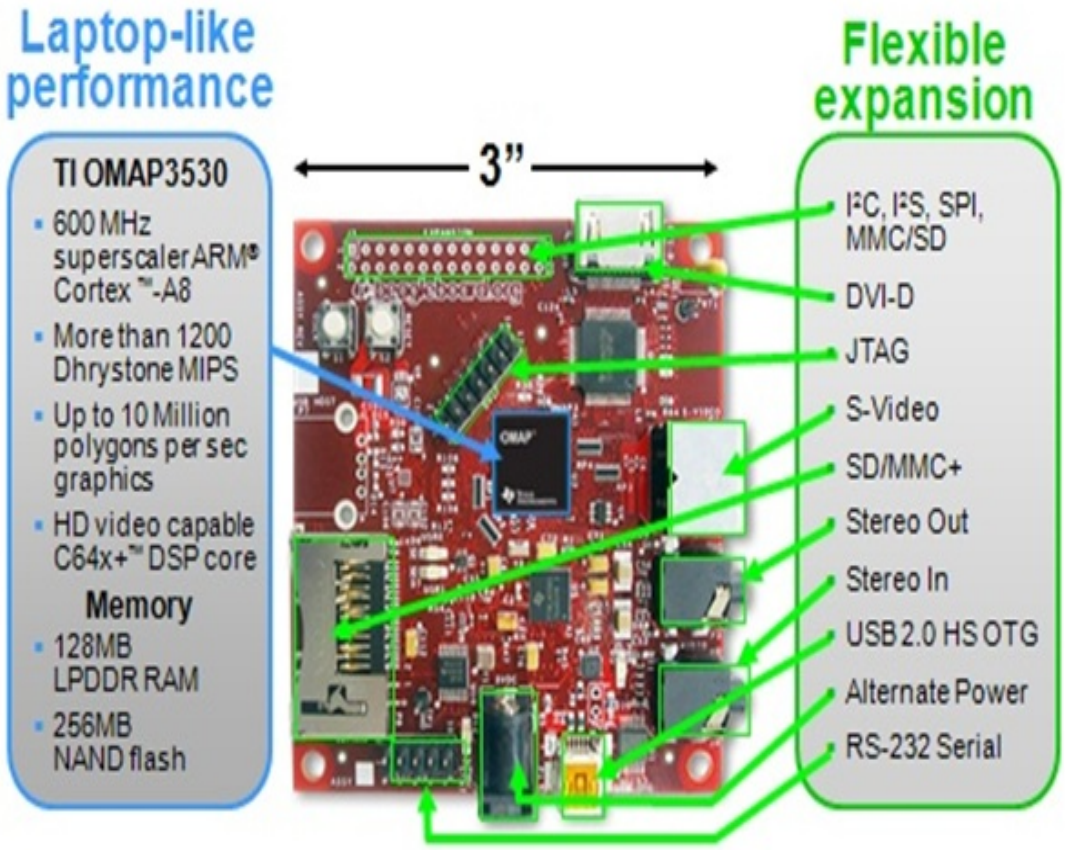

Figure 2.1: Beagleboard-XM

This balance of performance and power allow the device to support the following example applications:

- Portable Data Terminals
- Navigation
- Auto infotainment
- Gaming
- Medical Imaging

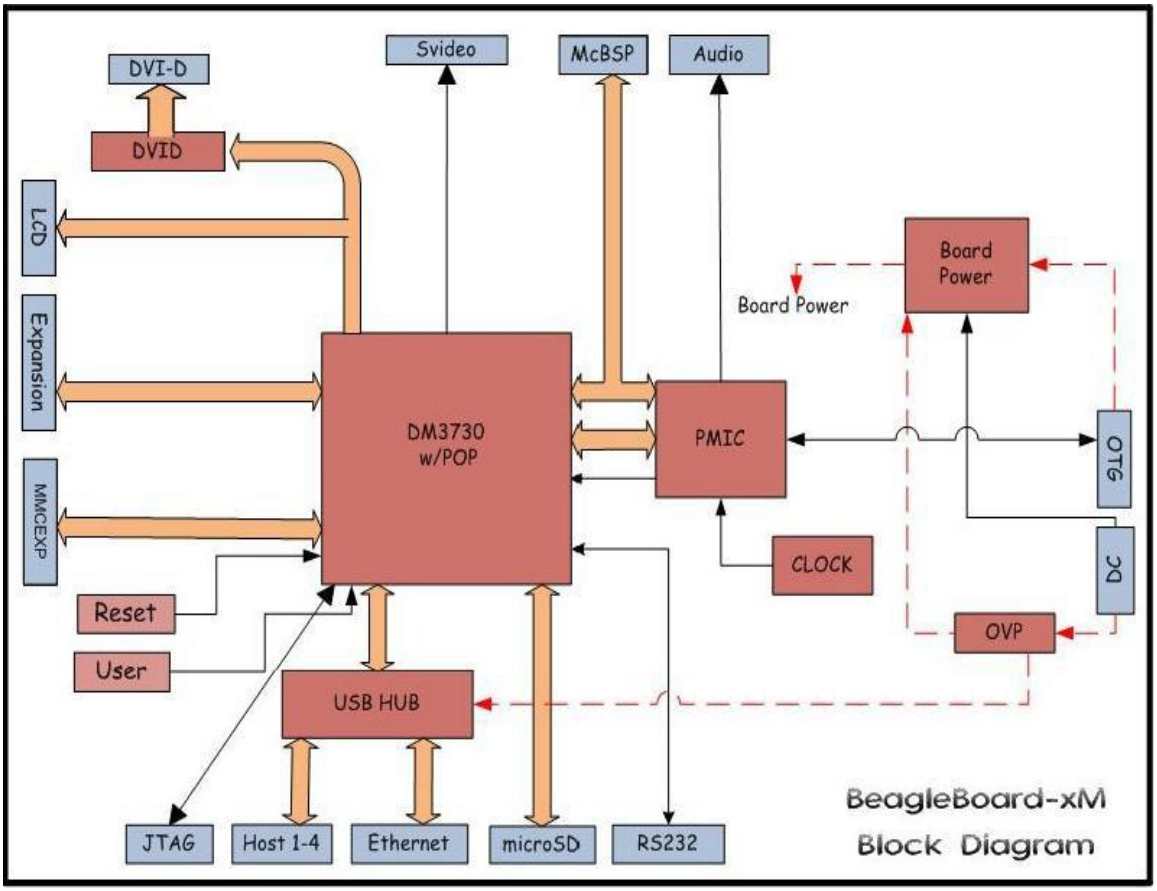

Figure 2.2: HighLevel Block Diagram of Beagleboard-XM

- Single board Computers

The device can support numerous HLOS and RTOS solutions including Linux and Windows Embedded CE which are available directly from TI. Additionally, the device is fully backward compatible with previous Cortex-A8 processors and OMAP processors.

Main Features

- ARM microprocessor (MPU) Subsystem compatible with OMAP-3 Architecture
- High Performance Image, Video, Audio Accelerator Subsystem

- POWER SGX Graphics Accelerator
- Advanced Very-Long-Instruction-Word (VLIW) TMS320C64x+ DSP Core

NEON SIMD for multimedia processing The ARM NEON[9] general-purpose SIMD engine efficiently processes current and future multimedia formats, enhancing the user experience. NEON technology can accelerate multimedia and signal processing algorithms such as video encode/decode, 2D/3D graphics, gaming, audio and speech processing, image processing, telephony, and sound synthesis by at least 3x the performance of ARMv5 and at least 2x the performance of ARMv6 SIMD.

NEON supports the widest range of multimedia codecs

- Many soft codec standards: MPEG-4, H.264, On2 VP6/7/8, Real, AVS
- Ideal solution for normal size "internet "streaming" decode of various formats
- Not just for codecs - also applicable to 2D and 3D graphics and other vector processing
- Off the shelf tools, OS support, and ecosystem support
- NEON will give 60-150% performance boost on complex video codecs
- Individual simple DSP algorithms can show larger performance boost (4x-8x)
- Processor can sleep sooner, resulting in overall dynamic power saving

2.1.2 Memory

There are two possible memory devices used on the xM. The -00 assembly uses the Micron POP memory and the -01 uses the Numonyx POP memory. The key function of the POP memory is to provide: 4Gb MDDR SDRAM x32 (512MB @ 166MHz)

Unlike with earlier versions of the board, no other memory devices are on the BeagleBoard. It is possible however, that additional nonvolatile memory storage can be added to BeagleBoard by:

- Accessing the memory on the uSD card
- Use the USB OTG port and a powered USB hub to drive a USB Thumb drive or hard drive.
- Install a thumb drive into one of the USB ports
- Add a USB to Hard Disk adapter to one of the USB ports

2.1.3 Power management

The TPS65950 is used on the BeagleBoard to provide power with the exception of a 3.3V regulator which is used to provide power to the DVI-D encoder and RS232 driver and an additional 3.3V regulator to power the USB Hub.

2.1.4 HS USB 2.0 OTG Port

The USB OTG port can be used as the primary power source and communication link for the BeagleBoard and derives power from the PC over the USB cable. The client port is limited in most cases to 500mA by the PC. There are instances where the PC or laptop does not supply sufficient current to power the board as it does not provide the full 500mA. Under this mode the USB HUB will now be powered based on the design changes made to the over volt circuitry. Care should be taken not to overload the USB ports as the total power supplied to the ports will not enable full power to all of the USB ports as you can have with the DC power.

2.1.5 HS USB 2.0 Host Ports

On the board are four USB Type A connectors with full LS/FS/HS support. Each port can provide power on/off control and up to 500mA of current at 5V as long as the input DC is at least 3A. The ports will not function unless the board is powered by the DC jack. They cannot be powered via the OTG port.

2.1.6 DVI-D Connector

The BeagleBoard can drive a LCD panel equipped with a DVI-D digital input. This is the standard LCD panel interface of the processor and will support 24b color output. DDC2B (Display Data Channel) or EDID (Enhanced Display ID) support over I2C is provided in order to allow for the identification of the LCD monitor type and the required settings. The BeagleBoard is equipped with a DVI-D interface that uses an HDMI connector that was selected for its small size. It does not support the full HDMI interface and is used to provide the DVI-D interface portion only. The user must use a HDMI to DVI-D cable or adapter to connect to a LCD monitor. This cable or adapter is not provided with the BeagleBoard. A standard HDMI cable can be used when connecting to a monitor with an HDMI connector.

2.1.7 Onboard USB HUB

A new feature of the xM board is the inclusion of an onboard USB 4 port hub with an integrated 10/100 Ethernet as shown in the block diagram.

2.1.8 LCD Header

A pair of 1.27mm pitch 2x10 headers are provided to gain access to the LCD signals. This allows for the creation of LCD boards that will allow adapters to be made to provide the level translation to support different LCD panels.

2.1.9 MicroSD Connector

A single microSD connector is provided as a means for the main non-volatile memory storage on the board. This replaces the 6 in 2 SD/MMC connector found on the BeagleBoard.

2.1.10 User Button

A button is provided on the BeagleBoard to be used as an application button that can be used by SW as needed. As there is no NAND boot option on the board,

this button is no longer needed to force an SD card boot. It is can be used by the UBoot SW to switch between user scripts to allow different boot configurations to be selected as long as that feature is included in the UBoot used. If you press this button on power up, the board will not boot properly

2.1.11 Indicators

There are four green LEDs on the BeagleBoard that can be controlled by the user.

- One on the TPS65950 that is programmed via the I2C interface
- Two on the processor controlled via GPIO pins
- One Power LED that indicates that power is applied and can be turned off via SW
- One to indicate that power is applied to the onboard USB HUB and can be controlled via the SW.

There is also one red LED on the BeagleBoard that provides an indication that the power connected to the board exceeds the voltage range of the board. If this LED ever turns on, please remove the power connector and look for the correct power supply in order to prevent damage to the board.

2.1.12 Power Connector

Power can be supplied via the USB OTG connector for some application that does not require the USB Host ports. A wall supply 5V can be plugged into the DC power jack from full access to all functions of the board. When the wall supply is plugged in, it will remove the power path from the USB connector and will be the power source for the whole board. The power supply is not provided with the BeagleBoard.
When using the USB OTG port in the host mode, the DC supply must be connected as the USB port will be used to provide limited power to the hub at a maximum

of 100mA, so the hub must be powered. The 100mA is not impacted by having a higher amperage supply plugged into the DC power jack. The 100mA is a function of the OTG port itself. Make sure the DC supply is regulated and a clean supply. If the power is over the voltage specification, a RED LED will turn on. This will prevent the power from actually making it to the circuitry on the board and will stay on as long as the power exceeds the voltage specification.

2.1.13 JTAG Connector

A 14 pin JTAG header is provided on the BeagleBoard to facilitate the SW development and debugging of the board by using various JTAG emulators. The interface is at 1.8V on all signals. Only 1.8V Levels are supported.

2.1.14 RS232 DB9 Connector

Support for RS232 via UART3 is provided by DB9 connector on the BeagleBoard for access to an onboard RS232 transceiver. A USB to Serial cable can be plugged directly into the Beagle. Unlike on the original version of the Beagle, a straight through non null modem cable is required. The cable you used on the BeagleBoard will NOT work on the xM version. A standard male to female straight DB9 cable may be used or you can use a USB to serial adapter that will plug directly into the board without the need for any other cables.

2.1.15 Camera Connector

A single connector has been added to the BeagleBoardxM board for the purpose of supporting a camera module. The camera module does not come with the board but can be obtained from Leopard Imaging. The supported resolutions include VGA, 2MP, 3MP, and 5MP camera modules. For proper operation of the cameras, the correct SW drivers are required. This connector is populated on the board and is ready for the camera module to be installed.

2.1.16 MMC3 Expansion Header

New to the BeagleBoard-xM is a 20 pin connector provided to allow access to additional signals including GPIO and the MMC3 port. This connector is populated on the board.

2.2 Porting Linux Operating System on Beagleboard-XM

Linux operating systems that Beagleboard-XM supports are :

- Angstrom
- Ubuntu
- Android
- Archlinux

In this project Ubuntu has been chosen for its ease of network configuration,desktop like GUI and inbuilt GCC compiler.Different methods used for porting OS on the board that includes manual installation,installing pre-configured image and net-install method.In this project later two methods have been used and tested for porting OS on the board.

2.2.1 Install Pre-Configured Image [11]

Get Prebuilt Image:
wget https://rcn-ee.net/deb/rootfs/raring/ubuntu-13.04-console-armhf-2013-11-15.tar.xz

Verify Image
md5sum ubuntu-13.04-console-armhf-2013-11-15.tar.xz 6692e4ae33d62ea94fd1b418d257a514 ubuntu-13.04-console-armhf-2013-11-15.tar.xz

Unpack Image
tar xf ubuntu-13.04-console-armhf-2013-11-15.tar.xz
cd ubuntu-13.04-console-armhf-2013-11-15

To find location of SD card:
sudo ./setup_ sdcard.sh –probe-mmc

Install Script for Beagleboard-XM
sudo ./setup_ sdcard.sh –mmc /dev/sdX –uboot beagle_ xm

Additional Options:
–rootfs ¡ext4 default¿
–swap_ file ¡swap file size in MB's¿

For Basic Framebuffer driven desktop environment:
Ethernet: *sudo ifconfig -a* and *sudo dhclient usb1* or *sudo dhclient eth0*

2.2.2 NetInstall Method

Download netinstall script
git clone git://github.com/RobertCNelson/netinstall.git
cd netinstall

Board Selection
*BeagleBoard xMx - omap3-beagle-xm
Options for Ubuntu Distributions
–distro oneiric (11.10)
–distro precise-armhf (12.04)
–distro quantal-armhf (12.10)

Install Script for ¡board¿ selection
sudo ./mk_ mmc.sh –mmc /dev/sdX –dtb ¡board¿ –distro < distro >

So for beagleboard-xM and Ubuntu 12.10 (quantal-armhf)
sudo ./mk_ mmc.sh –mmc /dev/sdX –uboot beagle _ xm –distro quantal-armhf

other options
–firmware : installs firmware
–serial-mode : debian-installer uses Serial Port

2.3 System Configuration for Static MAC and IP address

[4] Unlike its previous version Beagleboard-XM does not have a flash memory for static MAC address.Each time system assigns random MAC address on system boot up. Since it is required to have static MAC address that attaches to only given static IP address system is configured. In this configuration system file *interfaces* at the location */etc/network* is rewritten as follows.
/etc/network/interfaces:
auto eth0
iface eth0 inet static
address < IP address >
netmask < >
gateway < >
dns-nameservers < >
hwaddress ether < MAC address >

Chapter 3

Basics of Libraries,Makefile and Shell

In this chapter fundamental theoretical aspects and the implementation method of the libraries,makefile utility and shell scripting has been described that are used in the project's development.

3.1 Importance of Library

This methodology, also known as "shared components"or "archive libraries"together multiple compiled object code files into a single file known as a library. Typically C functions/C++ classes and methods which can be shared by more than one application are broken out of the application's source code, compiled and bundled into a library. The C standard libraries and C++ STL are examples of shared components which can be linked with your code. The benefit is that each and every object file need not be stated when linking because the developer can reference the individual library. This simplifies the multiple use and sharing of software components between applications. It also allows application vendors a way to simply release an API to interface with an application. Components which are large can be created for dynamic use, thus the library remain separate from the executable reducing it's size and thus disk space used. The library components are then called by various

applications for use when needed.

In this project work encryption process are carried by creating static library form the available open source code which is described in the next chapter.

3.2 Library Naming Conventions

Libraries are typically names with the prefix "lib". This is true for all the C standard libraries. When linking, the command line reference to the library will not contain the library prefix or suffix.[1]

Thus the following link command:
gcc src-file.c -lm -lpthread

The libraries referenced in this example for inclusion during linking are the math library and the thread library. They are found in /usr/lib/libm.a and /usr/lib/libpthread.a.

3.3 Linux Library Types

There are two Linux C/C++ library types which can be created:

3.3.1 Shared Library (.a)

Library of object code which is linked with, and becomes part of the application.
Steps to generate Library:

1. Compile: *gcc Wall c math.c*
2. Create Library libmath.a : *ar -cvq libmath.c math.o*

3. List files in library : *ar t libmath.a*

4. Linking with library:
   gcc o exec_ name main.c libmath.a
   gcc -o exec_ name main.c -L/path/to/library-directory lmath

3.3.2 Dynamically Linked "Shared Object"Libraries: (.so):

Steps to create shares object file:

1. create object code

2. Create Library

3. Optional: create default version using a symbolic link.

Library creation example:
gcc Wall fPIC c math.c
gcc -shared -Wl,-soname,libmath.so.1 o libmath.so.1.0 math.o
mv libmath.so.1.0 /opt/lib
ln -sf /opt/lib/libmath.so.1.0 /opt/lib/libmath.so.1
ln -sf /opt/lib/libmath.so.1.0 /opt/lib/libmath.so

This creates the library libmath.so.1.0 and symbolic links to it. It is also valid to cascade the linkage:

ln -sf /opt/lib/libcmath.so.1.0 /opt/lib/libmath.so.1
ln -sf /opt/lib/libmath.so.1 /opt/lib/libmath.so
Compiler Options:

- -Wall: include warnings. See man page for warnings specified.

- -fPIC: Compiler directive to output position independent code, a characteristic required by shared libraries. Also see "-fpic".

- -shared: Produce a shared object which can then be linked with other objects to form an executable.
- -Wl,options: Pass options to linker. In this example the options to be passed on to the linker are: "-soname libmath.so.1". The name passed with the "-o" option is passed to gcc.
- Option -o: Output of operation. In this case the name of the shared object to be output will be "libmath.so.1.0"

Library Links:

- The link to /opt/lib/libmath.so allows the naming convention for the compile flag -lctest to work.
- The link to /opt/lib/libmath.so.1 allows the run time binding to work. See dependency below.

Compile main program and link with shared object library:
Compiling for runtime linking with a dynamically linked libctest.so.1.0:
gcc -Wall -I/path/to/include-files -L/path/to/libraries main.c -lmath -o main or
gcc -Wall -L/opt/lib main.c -lmaath -o main
Where the name of the library is libmath.so. The libraries will NOT be included in the executable but will be dynamically linked during runtime execution.

List Dependencies: The shared library dependencies of the executable can be listed with the command: ldd name-of-executable
Example: *ldd main*
Linux-gate.so.1 = > (0xb77c4000)
Libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7605000)
/lib/ld-linux.so.2 (0Xb77c5000)

Setting Library Path :

In order for an executable to find the required libraries to link with during run time, one must configure the system so that the libraries can be found. Methods available:

1. Add library directories to be included during dynamic linking to the file /etc/ld.so.conf
   Sample: /etc/ld.so.conf
   /usr/X11R6/lib
   /usr/lib
   ...
   ...
   /usr/lib/sane
   /usr/lib/mysql
   /opt/lib
   Add the library path to this file and then execute the command (as root) ldconfig to configure the linker run-time bindings. You can use the "-f file-name" flag to reference another configuration file if you are developing for different environments.
   OR

2. Add specified directory to library cache: (as root)
   ldconfig -n /opt/lib
   Where /opt/lib is the directory containing your library libctest.so (When developing and just adding current directory: ldconfig -n . Link with -L.) This will NOT permanently configure the system to include this directory. The information will be lost upon system reboot.
   OR

3. Specify the environment variable LD_ LIBRARY_ PATH to point to the directory paths containing the shared object library. This will specify to the run time loader that the library paths will be used during execution to resolve

dependencies.

Example (bash shell): export LD_ LIBRARY_ PATH=/opt/lib:$LD_ LIBRARY_ PATH

This instructs the run time loader to look in the path described by the environment variable LD_ LIBRARY_ PATH, to resolve shared libraries. This will include the path /opt/lib.

3.4 Makefile

Large projects can contain thousands of lines of code, distributed in multiple source files, written by many developers and arranged in several subdirectories. A project may contain several component divisions. These components may have complex inter-dependencies for example, in order to compile component X, you have to first compile Y; in order to compile Y, you have to first compile Z; and so on. For a large project, when a few changes are made to the source, manually recompiling the entire project each time is tedious, error-prone and time-consuming.[1]

Make is a solution to these problems. It can be used to specify dependencies between components, so that it will compile components in the order required to satisfy dependencies. An important feature is that when a project is recompiled after a few changes, it will recompile only the files which are changed, and any components that are dependent on it. This saves a lot of time. Make is, therefore, an essential tool for a large software project.

Each project needs a Makefile a script that describes the project structure, namely, the source code files, the dependencies between them, compiler arguments, and how to produce the target output (normally, one or more executables). Whenever the make command is executed, the Makefile in the current working directory is interpreted, and the instructions executed to produce the target outputs. The Makefile contains a collection of rules, macros, variable assignments, etc. (Makefile or makefile are both acceptable.)

Writing Makefile:

target: dependency1 dependency2 ...

[TAB] action1

[TAB] action2

.....

Makefile

all: main.o module.o

gcc main.o module.o -o target_ bin

main.o: main.c module.h

gcc -I . -c main.c

module.o: module.c module.h

gcc -I . -c module.c

Clean:

*rm -rf *.o*

rm target_ bin

1. all is a special target that depends on main.o and module.o, and has the command (from the manual steps earlier) to make GCC link the two object files into the final executable binary.

2. main.o is a filename target that depends on main.c and module.h, and has the command to compile main.c to produce main.o.

3. module.o is a filename target that depends on module.c and module.h; it calls GCC to compile the module.c file to produce module.o.

4. clean is a special target that has no dependencies, but specifies the commands to clean the compilation outputs from the project directories.

Makefile Processing[12]: When the make command is executed, it looks for a file named makefile or Makefile in the current directory. It parses the found Makefile,

and constructs a dependency tree. Based on the desired make target specified (or implied) on the command-line, make checks if the dependency files of that target exist. And (for filename targets explained below) if they exist, whether they are newer than the target itself, by comparing file timestamps.

Before executing the action (commands) corresponding to the desired target, its dependencies must be met; when they are not met, the targets corresponding to the unmet dependencies are executed before the given make target, to supply the missing dependencies.

When a target is a filename, make compares the timestamps of the target file and its dependency files. If the dependency filename is another target in the Makefile, make then checks the timestamps of that targets dependencies. It thus winds up recursively checking all the way down the dependency tree, to the source code files, to see if any of the files in the dependency tree are newer than their target filenames. (Of course, if the dependency files dont exist, then make knows it must start executing the make targets from the lowest point in the dependency tree, to create them.)

If make finds that files in the dependency tree are newer than their target, then all the targets in the affected branch of the tree are executed, starting from the lowest, to update the dependency files. When make finally returns from its recursive checking of the tree, it completes the final comparison for the desired make target. If the dependency files are newer than the target (which is usually the case), it runs the command(s) for the desired make target. This process is how make saves time, by executing only commands that need to be executed, based on which of the source files (listed as dependencies) have been updated, and have a newer timestamp than their target.

Now, when a target is not a filename (like all and clean in our Makefile, which we called special targets), make obviously cannot compare timestamps to check whether the targets dependencies are newer. Therefore, such a target is always ex-

ecuted, if specified (or implied) on the command line.

For the execution of each target, make prints the actions while executing them. Note that each of the actions (shell commands written on a line) are executed in a separate sub-shell. If an action changes the shell environment, such a change is restricted to the sub-shell for that action line only. For example, if one action line contains a command like cd newdir, the current directory will be changed only for that line/action; for the next line/action, the current directory will be unchanged.

In above example, targets dependencies are module.o and main.o. Since these files do not exist on our first run of make for this project, make notes that it must execute the targets main.o and module.o. These targets, in turn, produce the main.o and module.o files by executing the corresponding actions/commands. Finally, make executes the command for the target all.

For any changes we can remove bin file by explicitly stating clean after make:
$ make clean
*rm -rf *.o*
rm target_ bin

3.5 Shell Scripting

A Shell is a program that acts as an interface between user and UNIX system. A Shell is an environment in which commands, programs and Shell scripts can be run. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.[2]

3.5.1 Shell Types

In UNIX there are two major types of shells:

1. The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
2. The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows:

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The Bourne shell was the first shell to appear on UNIX systems, thus it is referred to as "the shell".
The Bourne shell is usually installed as /bin/sh on most versions of UNIX. For this reason, it is the shell of choice for writing scripts to use on several different versions of UNIX.
Unlike common language shell just interprets the input rather than compile it so utilities written in shells are faster to run than programs written in particular language compiling , debugging tools. Moreover shell can interprets inputs from many language including C,C++, Perl, Tcl/Tk and Python.

3.5.2 Writing a Script

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by a pound

sign,# , describing the steps. A shell scrip is written in text editor which is can be given extension of .sh However it is not compulsory. Every shell script starts with comment that is

!/bin/sh
This tells the system that program following # ! will be used to execute following script. Scripts are ended with exit code that ensures that script returns with sensible exit code. Writing an exit code is important when script is being called in by another script in which exit code return is used for testing success of the script.

Making Script Executable
Scripts in Unix can only be executed with admin permission. Before executing script it is required to make it executable. Following command can be used for this. $chmod +x first.sh $first.sh

Bash Environment Variables: When a shell script starts, some variables are initialized from values in the environment. These are normally capitalized to distinguish them from user-defined variables in the script which are conventionally lowercase. Some principal environment variables are:

Parameter Variable If script is invoked with parameters, some additional variables are created. Even if no parameters are passed, the preceding environment variable $# still exists but has a value of 0. The parameter variables are listed in the following table.

3.5.3 System run level and init.d scripts

The /etc/init.d directory contains the scripts executed by init at boot time and when the init state (or "runlevel") is changed.
There is symbolic link method for handling these scripts. These scripts are referenced by symbolic links in the /etc/rcn.d directories[6]. When changing runlevels, init looks in the directory /etc/rcn.d for the scripts it should execute, where n is the

Table 3.1: Shell Environment Variables

Environment Variable	Description
$HOME	The home directory of the current user.
$PATH	A colon-separated list of directories to search for commands.
$PS1	A command prompt, frequently $, but in bash user can use some more complex values.
$PS2	A secondary prompt, used when prompting for additional input;usually >.
$IFS	An input field separator; a list of characters that are used to separate words when the shell is reading input, usually space, tab, and newline characters.
$0	The name of the shell script.
$#	The number of parameters passed.
$$	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example /tmp/tmpfile_$$.

Table 3.2: Parameter Varible

Parameter Variable	Description
$1,$2,....	The parameters given to the script
$*	A list of all the parameters, in a single variable, separated by the first character in the environment variable IFS.
$@	A subtle variation on $*; it doesnt use the IFS environment variable, so parameters may be run together if IFS is empty.

runlevel that is being changed to, or S for the boot-up scripts.
The names of the links all have the form Smmscript or Kmmscript where mm is a two-digit number and script is the name of the script (this should be the same as the name of the actual script in /etc/init.d).
When init changes runlevel first the targets of the links whose names start with a K are executed, each with the single argument stop, followed by the scripts prefixed with an S, each with the single argument start. (The links are those in the /etc/rcn.d directory corresponding to the new runlevel.) The K links are responsible for killing services and the S link for starting services upon entering the runlevel.
When init changes runlevel first the targets of the links whose names start with a K are executed, each with the single argument stop, followed by the scripts prefixed with an S, each with the single argument start. (The links are those in the /etc/rcn.d directory corresponding to the new runlevel.) The K links are responsible for killing services and the S link for starting services upon entering the runlevel.
For example, if we are changing from runlevel 2 to runlevel 3, init will first execute all of the K prefixed scripts it finds in /etc/rc3.d, and then all of the S prefixed scripts in that directory. The links starting with K will cause the referred-to file to be executed with an argument of stop, and the S links with an argument of start. The two-digit number mm is used to determine the order in which to run the scripts: low-numbered links have their scripts run first. For example, the K20 scripts will be executed before the K30 scripts. This is used when a certain service must be started before another.

Building a Start-up Script

Packages that include daemons for system services should place scripts in /etc/init.d to start or stop services at boot time or during a change of runlevel. These scripts should be named /etc/init.d/package, and they should accept one argument, saying what to do:

stop :

stop the service,
restart :
stop and restart the service if it's already running, otherwise start the service
reload:
cause the configuration of the service to be reloaded without actually stopping and restarting the service,
force-reload:
cause the configuration to be reloaded if the service supports this, otherwise restart the service.

The start, stop, restart, and force-reload options should be supported by all scripts in /etc/init.d, the reload option is optional.

The init.d scripts must ensure that they will behave sensibly (i.e., returning success and not starting multiple copies of a service) if invoked with start when the service is already running, or with stop when it isn't, and that they don't kill unfortunately-named user processes. The best way to achieve this is usually to use start-stop-daemon with the –oknodo option.

If a service reloads its configuration automatically (as in the case of cron, for example), the reload option of the init.d script should behave as if the configuration has been reloaded successfully.

The /etc/init.d scripts must be treated as configuration files, either (if they are present in the package, that is, in the .deb file) by marking them as conffiles, or, (if they do not exist in the .deb) by managing them correctly in the maintainer scripts. This is important since we want to give the local system administrator the chance to adapt the scripts to the local system, e.g., to disable a service without de-installing the package, or to specify some special command line options when starting a service, while making sure their changes aren't lost during the next package upgrade.

These scripts should not fail obscurely when the configuration files remain but the package has been removed, as configuration files remain on the system after the package has been removed. Only when *dpkg* is executed with the –purge option will configuration files be removed. In particular, as the /etc/init.d/package script itself is usually a conffile, it will remain on the system if the package is removed but not purged. Therefore, you should include a test statement at the top of the script, like this:

test -f program-executed-later-in-script —— exit 0

Script variable should be placed in a file in /etc/default, which typically will have the same base name as the init.d script. This extra file should be sourced by the script when the script runs. It must contain only variable settings and comments in SUSv3 sh format. It may either be a conffile or a configuration file maintained by the package maintainer scripts. See for more details.

The program update-rc.d is provided for package maintainers to arrange for the proper creation and removal of /etc/rcn.d symbolic links, or their functional equivalent if another method is being used.

By default update-rc.d will start services in each of the multi-user state runlevels (2, 3, 4, and 5) and stop them in the halt runlevel (0), the single-user runlevel (1) and the reboot runlevel (6). The system administrator will have the opportunity to customize runlevels by simply adding, moving, or removing the symbolic links in /etc/rcn.d if symbolic links are being used, or by modifying /etc/runlevel.conf if the file-rc method is being used.
To get the default behavior of package, put in users postinst script
update-rc.d package defaults

To remove package from start up

update-rc.d -f package remove

Running a Script

The program invoke-rc.d is provided to make it easier for package maintainers to properly invoke an init script, obeying run level and other locally-defined constraints that might limit a package's right to start, stop and otherwise manage services. This program may be used by maintainers in their packages' scripts. By default, invoke-rc.d will pass any action requests (start, stop, reload, restart...) to the /etc/init.d script, filtering out requests to start or restart a service out of its intended run levels.

Chapter 4

Media File Encryption

As long as the operating system is running on a system without file encryption, access to the files will have to go through OS-controlled user authentication and access control lists. If an attacker gains physical access to the computer, however, this barrier can be easily circumvented. One way would be to remove the disk and put it in another computer with an OS installed that can read the file system, or simply reboot the computer from a boot CD or USB device containing an OS that is suitable to access the local file system. In this project it unauthorized access to media player at the site and media files are prevented by transmitting them in the encryption form and store them in the encrypted form at the display site, where files are decrypted by the program before they are taken out for the display.

4.1 Types of Encryption

1. **Symmetric Key Encryption**

   Symmetric-key algorithms are a class of algorithms for cryptography that use the same cryptographic keys for both encryption of plaintext and decryption of ciphertext.Examples of popular symmetric algorithms include Twofish, Serpent, AES (Rijndael), Blowfish, CAST5, RC4, 3DES, and IDEA.

2. **Public Key Encryption**

   Public-key cryptography, also known as asymmetric cryptography, refers to

a cryptographic algorithm which requires two separate keys, one of which is secret (or private) and one of which is public. Although different, the two parts of this key pair are mathematically linked. The public key is used to encrypt plaintext or to verify a digital signature; whereas the private key is used to decrypt ciphertext or to create a digital signaturePublic-key algorithms are fundamental security ingredients in cryptosystems, applications and protocols. They underpin such Internet standards as Transport Layer Security (TLS), PGP, and RSA

4.2 Advanced Encryption Standard (AES)

4.2.1 Advantages of AES over DES

In this project AES (Advanced Encryption Standard) has been used to encrypt media files which is a symmetric encryption algorithm. AES supersedes DES (Data Encryption Standard) which is also a symmetric-key algorithm.

DES uses 56 bit length key for encryption which is vulnerable to exhaustive search. DES can only process maximum 64-bits block at a time hence it is slower in both hardware and software implementation.

AES is based on a design principle known as a substitution-permutation network, and is fast in both software and hardware. Unlike DES, AES does not use a Feistel network. AES is a variant of Rijndael which has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. By contrast, the Rijndael specification per se is specified with block and key sizes that may be any multiple of 32 bits, both with a minimum of 128 and a maximum of 256 bits.

4.2.2 Description of Algorithm

The key size used for an AES cipher specifies the number of repetitions of transformation rounds that convert the input, called the plaintext, into the final output,

called the ciphertext. The number of cycles of repetition are as follows:

- 10 cycles of repetition for 128-bit keys
- 12 cycles of repetition for 192-bit keys
- 14 cycles of repetition for 256-bit keys

Each round consists of several processing steps, each containing four similar but different stages, including one that depends on the encryption key itself. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key.Details of the processing steps are described below.

1. KeyExpansion-round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.
2. InitialRound-
   (a) *AddRoundkey*-each byte of the state is combined with a block of the round key using bitwise xor.
3. Rounds
   (a) *SubBytes* -a non-linear substitution step where each byte is replaced with another according to a lookup table.
   (b) *ShiftRows*-a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
   (c) *MixColumns*-a mixing operation which operates on the columns of the state, combining the four bytes in each column.
   (d) *AddRoundkey*
4. Final Round(No MixColumns)
   (a) *SubBytes*

(b) *ShiftRows*

(c) *Addroundkey*

4.2.3 AES Modes of Operation and Initialization vector

Mode of Operation is an algorithm that uses block cipher to encrypt the plaintext. A block cipher by itself is only suitable for the secure cryptographic transformation (encryption or decryption) of one fixed-length group of bits called a block.A mode of operation describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block size.Common Modes that are used to cipher whole plaintext are as follows.

1. Electronic Code Book (ECB):The simplest of the encryption modes is the electronic codebook (ECB) mode. The message is divided into blocks, and each block is encrypted separately.

2. In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted.

3. Cipher Feedback: The cipher feedback (CFB) mode, a close relative of CBC, makes a block cipher into a self-synchronizing stream cipher. Operation is very similar; in particular, CFB decryption is almost identical to CBC encryption performed in reverse.

4. Output Feedback: The output feedback (OFB) mode makes a block cipher into a synchronous stream cipher. It generates keystream blocks, which are then XORed with the plaintext blocks to get the ciphertext.

Initialization Vector

Most modes require a unique binary sequence, often called an initialization vector (IV), for each encryption operation. The IV has to be non-repeating and for some modes random as well. The initialization vector is used to ensure distinct ciphertexts are produced even when the same plaintext is encrypted multiple times independently with the same key.

4.2.4 Implementation of Algorithm

Implementation of AES encryption has been done by static library which is created from open source code files written by Brian Gladman.This static library *aes.a*contains following source code files.[10]

aesencrypt.c-The main c source code file for encryption and decryption
aeskey.c-The main c source code file for the key schedule
aestab.c-The main file for AES block tables.
aesmodes.c-This c file defines all AES modes used in this algorithm.

AES Calling Interfaces

The basic AES code keeps its state in a context, there being different contexts for encryption and decryption. These are:
aes_ encrypt _ ctx
aes_ decrypt _ ctx

AES code is initialized with the call
aes _ init(void)

AES encryptio key is set by one of the following calls:
*aes _ encrypt _ key128(const unsigned char *key, aes _ encrypt _ ctx cx[1])*
*aes _ encrypt _ key192(const unsigned char *key, aes _ encrypt _ ctx cx[1])*
*aes _ encrypt _ key256(const unsigned char *key, aes _ encrypt _ ctx cx[1])*

Similarly, the AES decryption key is set by one of:
*aes _ decrypt _ key128(const unsigned char *key, aes _ decrypt _ ctx cx[1])*
*aes _ decrypt _ key192(const unsigned char *key, aes _ decrypt _ ctx cx[1])*
*aes _ decrypt _ key256(const unsigned char *key, aes _ decrypt _ ctx cx[1])*

Table 4.1: AES timing analysis for Beagleboard-XM

	Delay(Sec/ Mb)for the key size in bits					
Mode of Operation	128 Bits		192 bits		256 bits	
OFB	0.1617	0.1600	0.1767	0.1750	0.1868	0.1900
CFB	0.1583	0.1633	0.1800	0.1733	0.1916	0.1950
CBC	0.0583	0.0600	0.0583	0.0600	0.0533	0.0600

The above subroutines return a value of EXIT _ SUCCESS or EXIT _ FAILURE depending on whether the operation succeeded or failed.

Steps of the File Encryption

1. Declare and initialize the encryption Key
2. Open input and output file in read and write mode respectively
3. Pick a random initialization vector and write it in the output file,a file that would save cipher text
4. Set the encryption key for the schedule
5. Read the bytes from the input file into the buffer
6. Apply encryption to the buffer data by set encrypted key and initialization vector
7. Write this encrypted data into the buffer to the output file
8. Repeat the steps 5 to 7 until all of the bytes of the input file are read out
9. Close the input and output files

4.2.5 Timing Analysis

Timing analysis (Delay)has been carried out for different modes of operation and key sizes to find minimum delay with sufficient strength of encryption. The results are shown in the table 4.1.

Chapter 5

Process and Socket Programming

In Linux system each terminal window is probably running a shell; each running shell is another process. When command is executed from the shell, the corresponding program is executed in a new process; the shell process resumes when that process completes.[1]

In Linux most of the process manipulation functions are declared in ¡unistd.h¿.

5.1 Process ID

Each newly created process in the Linux system is assigned 16 bit unique number which is called process ID or pid. Each process has its parent process which is identified by parent process ID or ppid in corresponds to that process. In Linux system are arranged in tree like structure in which init process is at root of the all processes. Process ID of the current process can be obtained by getpid() and parent process ID can be obtained by getppid() function.

5.2 Creating Process

There are two methods of creating process. One method is by using system() function in the program which is inefficient and have security risks. Another method is by using fork() and exec() functions which is discussed here.

5.2.1 Calling fork

When a program calls fork, a duplicate process, called the child process, is created. The parent process continues executing the program from the point that fork was called. The child process, too, executes the same program from the same place. The fork function returns two different values of process ids to the parent process and child process. The return value in the parent process is the process ID of the child. The return value in the child process is zero. Because no process ever has a process ID of zero, this makes it easy for the program whether it is now running as the parent or the child process. Duplicate a programs process using fork

5.2.2 Calling execv

When a program calls an exec function, that process immediately ceases executing that program and begins executing a new program from the beginning, assuming that the exec call doesnt encounter an error.

Within the exec family, there are functions that vary slightly in their capabilities and how they are called.

- Functions that contain the letter p in their names (execvp and execlp) accept a program name and search for a program by that name in the current execution path; functions that dont contain the p must be given the full path of the program to be executed.

- Functions that contain the letter v in their names (execv, execvp, and execve) accept the argument list for the new program as a NULL-terminated array of pointers to strings. Functions that contain the letter l (execl, execlp, and execle) accept the argument list using the C languages varargs mechanism.
- Functions that contain the letter e in their names (execve and execle) accept an additional argument, an array of environment variables.The argument should be a NULL-terminated array of pointers to character strings. Each character string should be of the form VARIABLE=value.

Because exec replaces the calling program with another one, it never returns unless an error occurs.

5.3 Socket Interprocess Communication

Socket is one of the types of interprocess communication. Socket communication allows communication between unrelated process running on two different computers, it can also be used for process running on the same computer by specifying domain namespace as Local. The parameters of the socket are communication style, name space and protocol, that are need to be specified at the time of socket creation.

Communication style is how socket treats transmitted data and number of communicating partners. Data are sent in chunks called packets. There are two approaches in sending packets.

1. Connections oriented- It guarantees delivery of all the packets in order they were sent. If the packets are lost or reordered by problems in the network, the receiver automatically requests their re-transmission from the sender. A connection-style socket is like a telephone call:The addresses of the sender and receiver are fixed at the beginning of the communication when the connection is established.
2. Datagram styles do not guarantee delivery or arrival order. Packets may be lost or reordered in transit due to network errors or other conditions. Each packet

must be labeled with its destination and is not guaranteed to be delivered.The system guarantees only best effort, so packets may disappear or arrive in a different order than shipping.

A datagram-style socket behaves more like postal mail.The sender specifies the receivers address for each individual message.

A Socket name space specifies how socket address are written. In the local name space socket addresses are local file names while on the internet domain socket addresses are made of IP address and port number.
A protocol specifies how data is transmitted. Some protocols are TCP/IP, the primary networking protocols used by the Internet; the AppleTalk network protocol; and the UNIX local communication protocol.
For Socket creation all parameters are defined in struct .Structure is used for service name lock-ups and host name lock-ups.

This structure for handling internet addresses is:

include < netinet /in.h >

struct sockaddr_ in {
short sin_ family; // e.g. AF_ INET
unsigned short sin_ port; // e.g. htons(3490)
struct in _ addr sin_ addr; // see struct in_ addr, below
char sin_ zero[8]; // zero this if you want to
};

struct in_ addr {
unsigned long s_ addr; // load with inet_ aton()
}; For internet domain sin_ family has to be set to AF_ INET while sin_ zero should be set to zero with function memset().

5.3.1 System calls

[5] **getaddrinfo()**

It is used for DNS and service name lock ups and fills out the structure for subsequent use.

Prototype

include < sys/types.h >

include < sys/socket.h >

include < netdb.h >

int getaddrinfo(const char *node, // e.g. "www.example.com" or IP

const char *service, // e.g. "http" or port number

const struct addrinfo *hints,

struct addrinfo **res);

getaddinfo() gives pointer res to a linked-list, node parameter is the host name or ip address to connect Next is the parameter service, which can be a port number, like "80", or the name of a particular service

socket()

Prototype:

include < sys/types.h >

include < sys/socket.h >

int socket(int domain, int type, int protocol);

It returns socket() descriptor which can subsequently used by other system calls.

bind()

Prototype:

include < sys/types.h >

include < sys/socket.h >

int bind(int sockfd, struct sockaddr *my_ addr, socklen_ t addrlen);

bind() system call is used by server program to let remote client connect to the

server. Bind system call binds struct sockaddr_ in that has loaded up and socket descriptor. If server has only one IP address then s_ addr field in struct sockaddr_ in can be filled INADDR_ ANY.

connect()

Prototype:

include < sys/types.h >

include < sys/socket.h >

int connect(int sockfd, const struct sockaddr *serv_ addr, socklen_ t addrlen);

This system call is used to connect client to the remote server. It requires socket descriptor and server address.

listen()

Prototype:

include < sys/socket.h >

int listen(int s, int backlog);

Once socket descriptor is created server can call listen() to listen incoming requests from client side. Backlog parameter shows number of pending connection requests before server start rejecting them.

accept()

Prototype:

include < sys/types.h >

include < sys/socket.h >

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

s : The listen()ing socket descriptor.

addr: This is filled in with the address of the site that's connecting to you.

addrlen: This is filled in with the sizeof() the structure returned in the addr parameter.

send() and recv()

These two function are used for communication over stream sockets or connected datagram socket.
int send(int sockfd, const void *msg, int len, int flags);

Here *msg is the pointer to the data to be sent, len is the length of data to be sent and flag is generally set 0. send() returns number of byte sent out.

recv()
int recv(int sockfd, void *buf, int len, unsigned int flags);

sockfd is the socket descriptor to read from, buf is the buffer to read the information into, len is the maximum length of the buffer, and flags can again be set to 0. recv() returns the number of bytes actually read into the buffer, or -1 on error.

gethostname()
To convert human-readable hostnames, either numbers in standard dot notation (such as 10.0.0.1) or DNS names (such as www.codesourcery.com) into 32-bit IP numbers, gethostbyname() can be used.

Prototype:
include < unistd.h >

int gethostname(char *hostname, size_ t size);

The arguments are simple: hostname is a pointer to an array of chars that will contain the hostname upon the function's return, and size is the length in bytes of the hostname array.

Chapter 6

Implementation of Client-Server model

The process of updating the content of the media player module is the sequence of timing events that are described in this chapter through the client-server model.The important part of this process is the transferring of the bulk of the media files to the client at one instance for the updates as there is no user interaction is used at the media player module by which acknowledgment for the reception of the each files can be sent to the server.

6.1 Server Process

The task of the content server is to maintain content at the client side updated as and when user modifies content at the corresponding server directory.The task of synchronization between the server and client is maintained on the basis of 'timeout' phenomenon,in which client queries server after the each timeout session continuously.

Server process is described by the process flow chart in the figure 6.1. As shown in the flow chart on boot up server connects with the client and checks for its authentication. Server process then checks if it is newly joined client or first time session

then server transfers all content of the directory to the client in response.When server connects to client it waits for the timeout and receives client's media files list.Server matches this list with it content list if there is any difference the case when server content is updated in between the new 'difference' list is created , files in the list are extracted out,encrypted and sent to the server.Again server enters in the timeout session and thus process of updating continues.

Extracting files fro from directory and sending them over the sockets requires to form a list of files. The process of picking out file name from the list one by one and sending it over the socket is described in the figure 6.2 .

6.2 Client Process

Media player module runs client process that queries server process for updates at the fixed intervals.Client process also forks the child process that displays media files in the directory in a loop of the sequence.
As per the client process flow is described in the figure. client process prepares list of files in the directory and sends it to server. In return server send updated list and newly added files to the client. Client process compares server file list with its current directory list and deletes files that are not in the server list to update the content.

The important part of the client process is to recognize the received bytes as belonging to which particular file and store them in it. In this mechanism bytes to be stored are counted and compared with the file size of the file in which it is to be stored. Since there is no attribute such as the *EOF* for the media files the only mechanism to instruct the client to distribute the bytes in the corresponding files is by sending it the predetermined list of the file size for the comparison. This mechanism of receiving files based on their size is described in the figure 6.4.

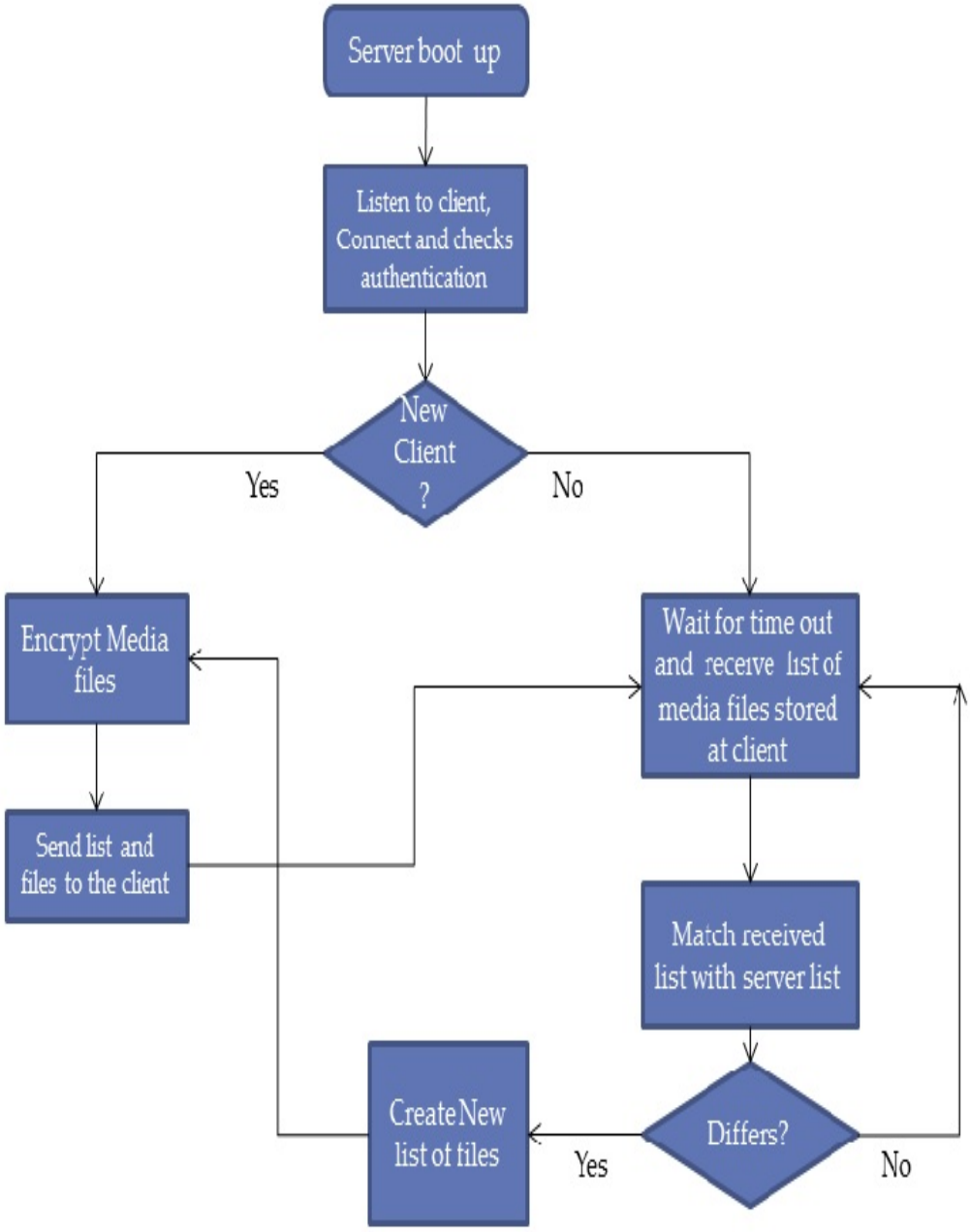

Figure 6.1: Server Process Flowchart

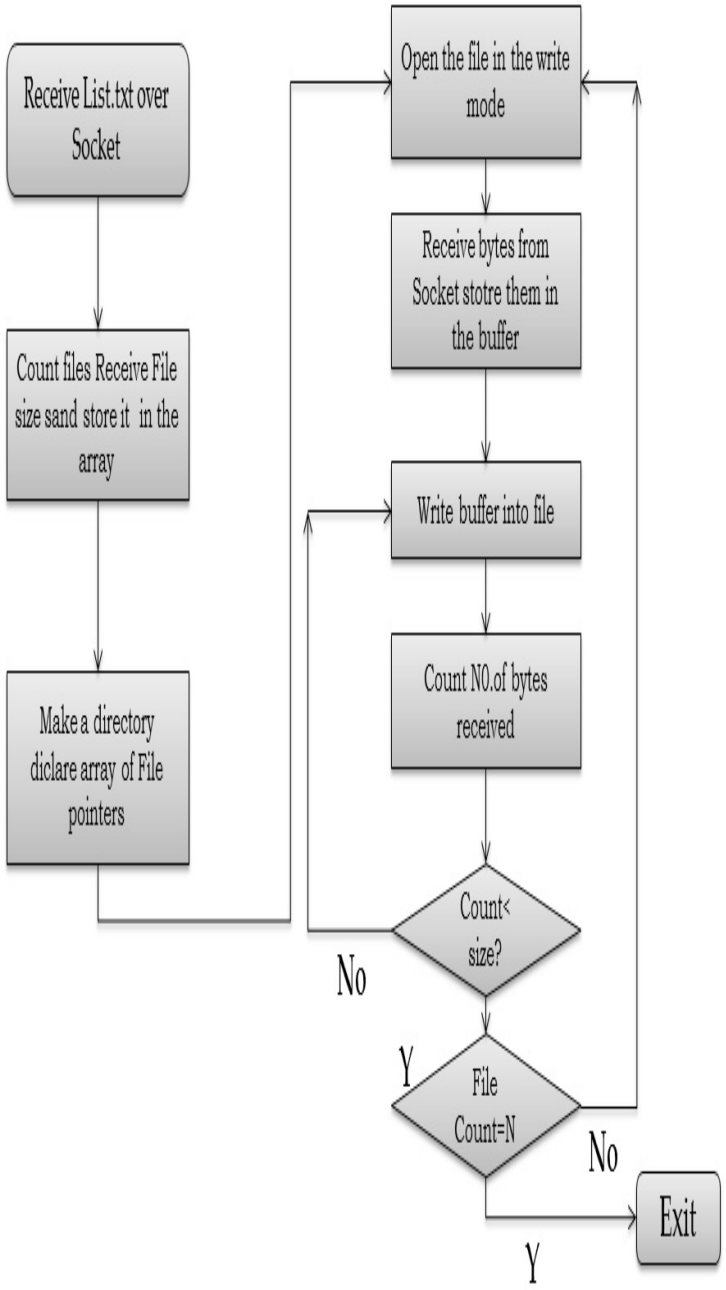

Figure 6.2: Process of sending files

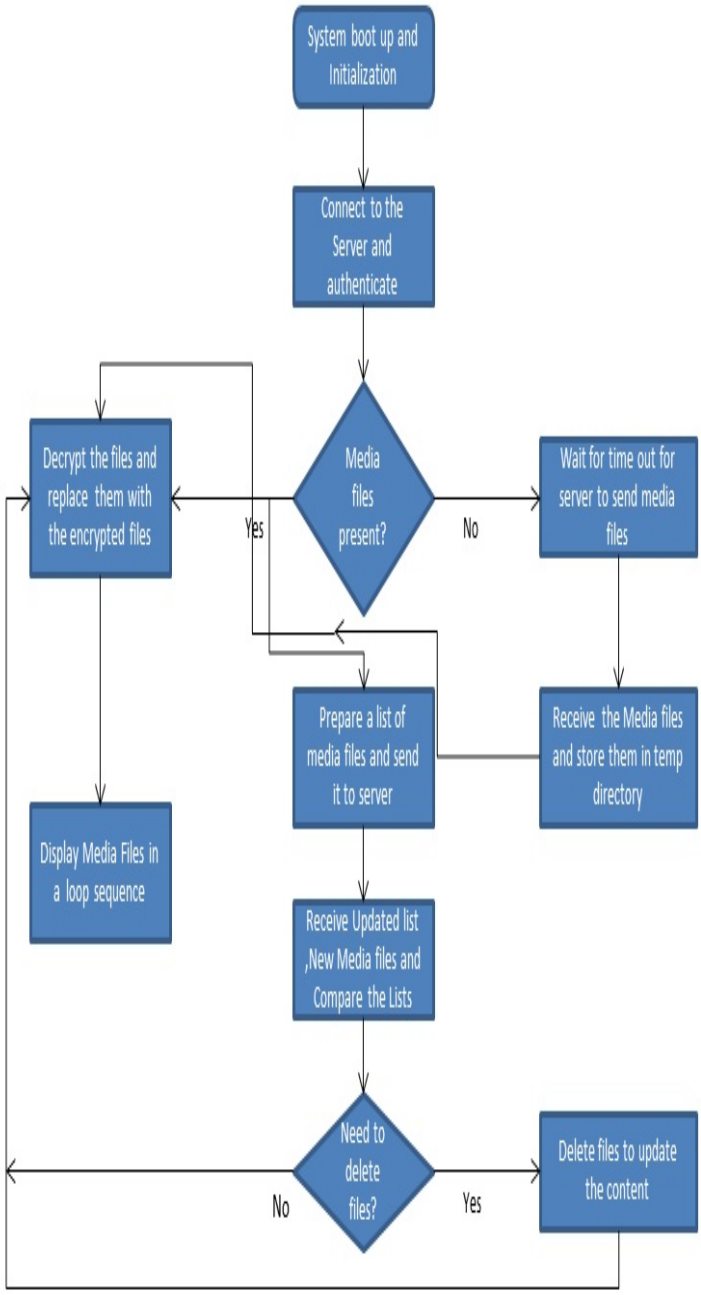

Figure 6.3: Client Process

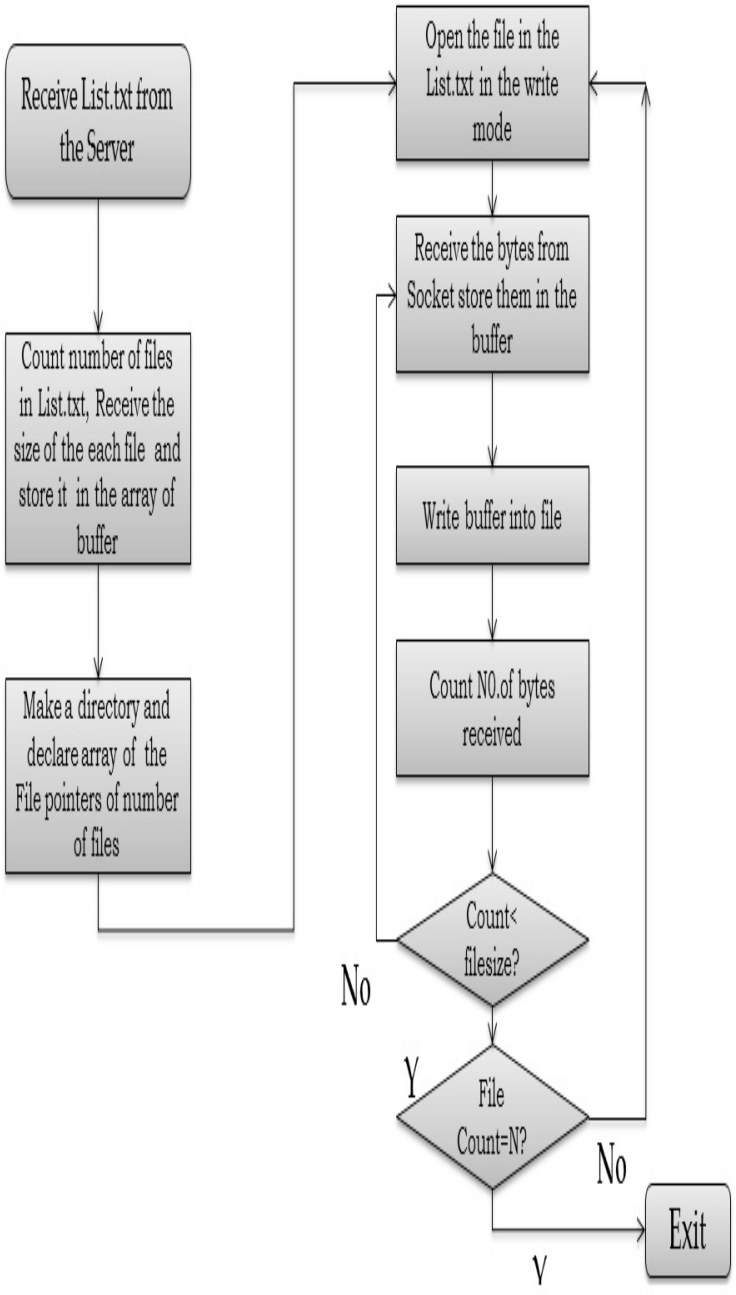

Figure 6.4: Client Process

Chapter 7

Conclusion and Future Scope

7.1 Conclusion

In this dissertation, an approach has been proposed for the content management and display of the digital outdoor media player module. Beagleboard-xM features has been explored and Beagleboard-xM has been tested with different OS that it supports and Ubuntu a popular linux OS has been chosen as the media player OS. OS has been configure for the static MAC and IP address and tested as Beagleboard xM doesnt have a flash memory to store the permanent MAC address. An open source media player software for the display is installed also the plugins for the MPEG-4 decoder has been installed (As Ubuntu doesnt configured with it)and media files of 720p @ 30fsp are tested. . Fundamentals of the libraries and *Makefile* utility has been studied, tested and applied to create the static library (aes.a) and to compile the program files. AES encryption has been studied and implemented for the different modes of operation and key sizes. From the results of the timing analysis mode of operation has been selected and applied for the encryption of the media files. Linux socket structure and system calls has been studied and example program developed and tested for the implementation of client-server model. The process flow for the server client model has been designed and implemented. The complex process of transferring bulk of media files for the periodic content update and synchronization has been designed, implemented and tested. A client program of displaying media

files from the directory has been developed and tested. The method of parallel processing using fork system call has been designed and tested which is applied in the concurrent operation of displaying media files and server interaction at the background. A start up script has been written that automates the process on the system boot up.

7.2 Future Scope

New technologies for digital sign are currently being developed, such as three-dimensional (3D) screens, with or without 3D glasses , 'holographic', displays,water screens and fog screens.Digital sign can interact with mobile phones. Using SMS messaging and Bluetooth, some networks are increasing the interactivity of the audience. SMS systems can be used to post messages on the displays, while Bluetooth allows users to interact directly with what they see on screen. In addition to mobile interactivity, networks are also using technology that integrates social and location-based media interactivity. This technology enables end users to send Twitter and Flickr messages as well as text messages to the displays.

Bibliography

[1] Advanced Linux Programming by Mark Mitchell, Jeffrey Oldham, and Alex Samuel, New Riders Publishing, 2001.

[2] Beginning Linux Programming 3rd Edition by Neil Mathews, Richard Stones,2006.

[3] The C programming Language By Brian W. Kernighan and Dennis M. Ritchie. Published by Prentice-Hall in 1988

[4] Embedded HOWto,"Patch for fixing random MAC address on Beagleboard-xM", http://blog.galemin.com/2010/11/patch-for-fixing-random-mac-address-on-beagleboard-xm/

[5] Beej's Guide to Network Programming,Verson 3.0.15,2012 By Beej Jorgensen

[6] Ian Jackson and Christian Schwarz,"System Run levels and *init.d* scripts, Debian Policy Manual Chapter-9 Operating System, http://www.debian.org/doc/debian-policy/# contents

[7] Beagleboard.org,"Reference Manual Beagleboard-xM"http://beagleboard.org/static/BBxMSRM_ latest.pdf

[8] Texas Instruments,DM3730, http://www.ti.com/product/dm3730

[9] ARM Holdings plc, Cortex-A8 Processor, http://www.arm.com/products/processors/cortex-a/cortex-a8.php

[10] Brian Gladman,"AES and Combined Encryption/Authentication Modes", http://www.gladman.me.uk/

[11] elinux.org, http://elinux.org/BeagleBoardUbuntu

[12] Sarath Lakshman "GNU Make in Detail for Beginners", http://www.linuxforu.com/2012/06/gnu-make-in-detail-for-beginners/