

Kernel & File System Optimization for Embedded VoIP Phone

A Dissertation

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science & Engineering

By

Ragin Shah

(04MCE018)

Guide

Prof. Jaladhi Joshi



Department of Computer Science & Engineering

Institute of Technology

Nirma University of Science and Technology

Ahmedabad - 382481

May, 2006



This is to certify that the Dissertation entitled
Kernel & File System Optimization for Embedded VoIP Phone

Presented by

Ragin Shah

has been accepted toward fulfillment of the requirement
for the degree of
Master of technology in Computer Science & Engineering

Professor in Charge

Head of the Department

CERTIFICATE

This is to certify that the work presented here by **Mr.Ragin Shah** entitled “*Kernel & File System Optimization for Embedded VoIP Phone*” has been carried out at **NirmaLabs** during the period **September 2005 – May 2006** is the bonafide record of the research carried out by him under my guidance and supervision and is up to the standard in respect of the content and presentation for being referred to the examiner. I further certify that the work done by him is his original work and has not been submitted for award of any other diploma or degree.

Prof. Jaladhi Joshi

Date: 28/04/06

ABSTRACT

Voice over IP is an emerging technology that enables voice communication over the Internet using the Internet Protocol. It is the process of breaking up audio or video into small chunks, transmitting those chunks over an IP n/w & reassembling those chunks at the far end so that two people can communicate. It is a cost-saving approach compared to traditional telephony approach. Project VoIP aims to develop a VoIP phone to capitalize this opportunity.

The Flash memory requirements for an Embedded Device are dependent upon the Size of the Operating System. The limited built-in configurability of Linux can lead to size overhead when it is used in embedded system like VoIP phone. The built in configuration options in Linux are not engineered for producing smallest kernel, but rather for enabling the kernel's deployment on a wide range of general-purpose systems. To remove the overhead in kernel & optimization of kernel according to hardware & software requirements for VoIP phone is a major task. VoIP application is a kind of real-time application. The performance of kernel should be improved for an application like VoIP. To improve kernel performance, it should have low scheduler latency. So, this scheduler latency needs to be reduced for VoIP phone. The root file system is needed in VoIP phone to hold the VoIP application & which enables the interaction of the application to the kernel & low-level hardware. The most critical part of an embedded system is boot loader. It is needed to start the operating system.

This Thesis addresses the issues of optimizing the Kernel, reducing Scheduler latency, preparing Boot loader and creating & optimizing the Root File System according to hardware and software requirements of a VoIP phone.

ACKNOWLEDGEMENTS

The project has been long one and many people got associated with it. The work would not have matured without the constant feedback, suggestions and constructive criticism of many people who took some time out of their busy schedule to guide this project; it has been a pleasure knowing them and inspiration learning from them.

This project reflects the help and advice of many people. I would like to thank Dr. Madhu Mehta, Chief Architect, NirmaLabs and Mr.Thyagrajan, CEO, NirmaLabs for their inspiration and advice.

Prof. Jaladhi Joshi, Project Guide, has been a tremendous supporter during the project period & I will always be grateful for his help and patience.

Special thanks to Mr. Madhukar Pai for showing me realities of self directed thinking & for giving technical assistance. I am very thankful to Nirav & Jatin for providing constant encouragement during project.

A very special thanks to my Course Coordinator Dr. S.N.Pradhan, who has supported and guided me in each and every step that has moved. He provided a very good technical help during project period.

TABLE OF CONTENTS

Abstract	i
Acknowledgements	ii
Table of Contents	iii
List of Tables	v
List of Figures	v
Acronyms	vi
Chapter 1: Introduction	1
1.1 Background	2
1.2 Project Definition	3
1.3 Tasks Enlisted	4
Chapter 2: Literature Study	5
2.1 Voice over IP	6
2.2 Embedded Linux	11
2.3 Kernel Optimization	15
2.4 Kernel Performance Improvement	18
2.5 Kernel Modules	21
2.6 File System Optimization	23
Chapter 3: Tools Overview	25
3.1 Hardware Tools	26
3.2 Software Tools	28
Chapter 4: Methodology	29
4.1 Architecture	30
4.2 Implementation Approach	32
4.3 Implementation Challenges	35

Chapter 5: Implementation Details	36
5.1 Creation of Project Workspace.....	37
5.2 Building a Cross-compiler tool-chain	38
5.3 Kernel Configuration & Optimization	42
5.4 Performance Improvement of Kernel	50
5.5 Mounting File System using NFS-mount	52
5.6 Creation & Optimization of Root File System	54
5.7 Making Kernel Module for keypad driver	60
5.8 Miscellaneous Tasks	62
Chapter 6: Results	65
Chapter 7: Summary	68
7.1 Conclusion	69
7.2 Future Work	69
References	70

LIST OF TABLES

No.	Page
1 Linux File System Hierarchy.....	24
2 Project Work Space.....	37
3 Cross-compiler tool-chain contents.....	41
4 Binary Executable Format.....	47

LIST OF FIGURES

No.	Page
1 VoIP Network.....	7
2 Architecture of Generic Linux System.....	12
3 TS-7250 Development board.....	26
4 Hardware Architecture of VoIP phone.....	30
5 Software Architecture of VoIP phone.....	31
6 Module wise work distribution.....	32

ACRONYMS

ADC	Analog to Digital Converter
ATA	Analog Telephone Adaptor
CODEC	Coder – Decoder
CPLD	Complex Programmable Logic Device
DAC	Digital to Analog Converter
DHCP	Dynamic Host Configuration Protocol
DIO	Digital Input Output
DNS	Domain Name System
EEPROM	Electrically Erasable Programmable Read Only Memory
GPOS	General Purpose Operating System
ITU	International Telecom Union
IETF	International Engineering Task Force
JFFS2	Journaling Flash File System-2
LAN	Local Area Network
MTD	Memory Technology Device
MMU	Memory Management Unit
NFS	Network File System
PBX	Private Branch Exchange
PPP	Point to Point Protocol
PSTN	Public Switched Telephone Network
SCSI	Small Computer System Interface
SDRAM	Synchronous Dynamic Random Access Memory
SIP	Session Initiation Protocol
TFTP	Trivial File Transfer Protocol
VLAN	Virtual Local Area Network
VoIP	Voice over Internet Protocol
WAN	Wide Area Network
YAFFS	Yet Another Flash File System



CHAPTER 1

INTRODUCTION

1.1 Background:

Project VoIP has been started in NirmaLabs, Nirma University. Project VoIP has a commercial value and aims at creating an intellectual property. But at the core of this project lies the intension of offering intellectual stimulus to the students who can turn ideas into tangible products that can benefit the entire community. Total eight students joined the VoIP project & I was one of them. Among eight students, six students were working in a software team & two students were working in hardware team. The goal of software team was to develop a VoIP phone on TS-7250 development board and the goal of hardware team was to replicate the TS-7250 board & make a prototype board which would work as a VoIP phone. In short, ultimate goal of Project VoIP team was to develop a full fledged working VoIP phone.

Now, memory would be critical issue for prototype board. The Flash memory requirements for an Embedded Device are dependent upon the Size of the Operating System, VoIP applications based on H.323 protocol, SIP application, etc. So, the goal of software team was to design & implement software in such a way that it best suits to memory requirement of target VoIP phone. Thereby, the main task of s/w team was to optimize the VoIP applications & size of operating system as per the software & hardware requirements of a VoIP phone.

1.2 Project Definition:

The Flash memory requirements for an Embedded Device are dependent upon the Size of the Operating System. The limited built-in configurability of Linux can lead to size overhead when it is used in embedded system like VoIP phone. The built in configuration options in Linux are not engineered for producing smallest kernel, but rather for enabling the kernel's deployment on a wide range of general-purpose systems. The kernel image which is provided by Technologic System (the TS-7250 board manufacturer) occupies 1.8 MB of memory space. This is very large for target VoIP phone's flash memory requirements. To remove the overhead in kernel & optimization of kernel according to hardware & software requirements for VoIP phone is a major task. VoIP application is a kind of real-time application. The performance of kernel should be improved for an application like VoIP. To improve the kernel performance, it should have low scheduler latency. So, this scheduler latency needs to be reduced for VoIP phone. The root file system is needed in VoIP phone to hold the VoIP application & which enables the interaction of the application to the kernel & low-level hardware. The root file system which is provided by Technologic System, occupies 7.6 MB of memory space. This size is beyond the limit of flash memory that we would keep in VoIP phone. So, root file system needs to be optimized for VoIP phone. The critical part of an embedded system is boot loader. It is needed to start the operating system.

As a part of my thesis work, I have been given tasks of optimizing the Kernel, reducing Scheduler Latency, preparing Boot loader and creating & optimizing the Root File System according to hardware and software requirements of a VoIP phone.

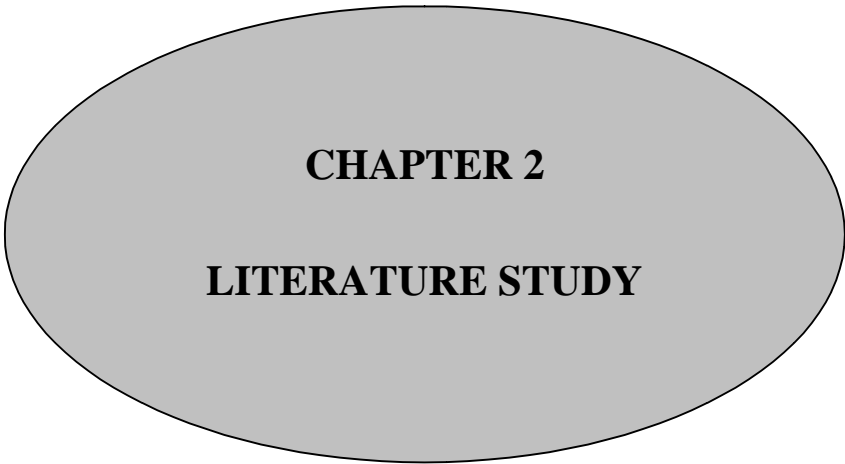
Now, Apart from doing above primary tasks, I had to do some secondary tasks that could help other team members to do their project work. I have been given a task to develop a cross-compiler tool-chain. It is needed to cross-compile the application & kernel so that they can run on TS-7250 development board & target VoIP phone as well. I have been told to setup a NFS-server. Using NFS-server, team members could get rid of copying application to the flash memory each time they want to test. NFS-server also needed to test the root file system that was developed by me.

1.3 Tasks Enlisted:

I have carried out following tasks during my whole project period.

- Built a Cross-compiler tool chain.
- Configured kernel for x86 & ARM architecture.
- Optimized the kernel by size.
- Improved the performance of kernel by reducing scheduler latency.
- Boot the kernel using TFTPBOOT.
- Configured DHCP server.
- Mounted File system using NFS mount.
- Created Root File system.
- Optimized the root file system.
- Made a kernel module for keypad driver.
- Ported boot loader in Flash & in EEPROM
- Provided Single Process Tracing support in TS-7250 board.
- Observed the differences between stock kernel-2.4.26 & Technologic System's kernel-2.4.26ts9.

The detail description of above tasks is given in chapter-5 of this document.



CHAPTER 2
LITERATURE STUDY

2.1 Voice over IP:

Since the telephone was invented in the late 1800s, telephone communication has not changed substantially. Of course, new technologies like digital circuits, DTMF (or, "touch tone"), and caller ID have improved on this invention, but the basic functionality is still the same. Over the years, service providers made a number of changes to improve on the kinds and types of services offered to subscribers, including toll-free numbers, call-return, call forwarding, etc. By and large, users do not know how those services work, but they did know two things: the same old telephone is used and the service provider charges for each and every little incremental service addition introduced. In the 1990s, a number of individuals in research environments, both in educational and corporate institutions, took a serious interest in carrying voice and video over IP networks, especially corporate intranets and the Internet. This technology is commonly referred to today as VoIP. As the Internet would make it possible to interconnect every home and every business with a packet-switched data network, VoIP will have a great future [12].

Voice over IP (VoIP) is an emerging technology that enables voice communication over the Internet using the Internet Protocol. It is the process of breaking up audio or video into small chunks, transmitting those chunks over an IP n/w & reassembling those chunks at the far end so that two people can communicate. VoIP can be used three ways:

- (1) ATA (Analog Telephone Adaptor): With the help of ATA, we can use normal PSTN telephone as a VoIP Phone. We just need to connect a normal telephone into ATA which in turn connected to Internet.
- (2) IP Phone: IP phone is the phone which can be connected to Internet directly. It has VoIP application pre-built in it.
- (3) Computer to Computer: This is also called soft phone. Here necessary VoIP application software is installed in the computer & after connecting computer to Internet world; it will work as a VoIP phone.

2.1.1 Why VoIP?

The cost-effectiveness is initially attractive when looking into VoIP. It is evident that an organization can gain efficiencies by only having to support a single network infrastructure. By using a single packet-switched network, as opposed to having to manage both packet and circuit-switched networks, organizations can realize reduced maintenance and management costs.

This convergence of voice and data networks onto a single IP network also provides some inherent flexibility, in terms of being able to easily add, change or remove nodes (e.g. phones) on the network. Finally, VoIP promises to deliver many nice new features, such as advanced call routing, computer integration, unified messaging, integrated information services, long-distance toll bypass, and encryption. Because of the common network infrastructure, it is also possible to integrate other media services, like video or even electronic white boards etc [6].

Due to the cost-effectiveness, flexibility and promise that leveraging a single IP network offers, it is no wonder that organizations are looking hard at the VoIP technology and trying to figure out how best to use it to their advantage.

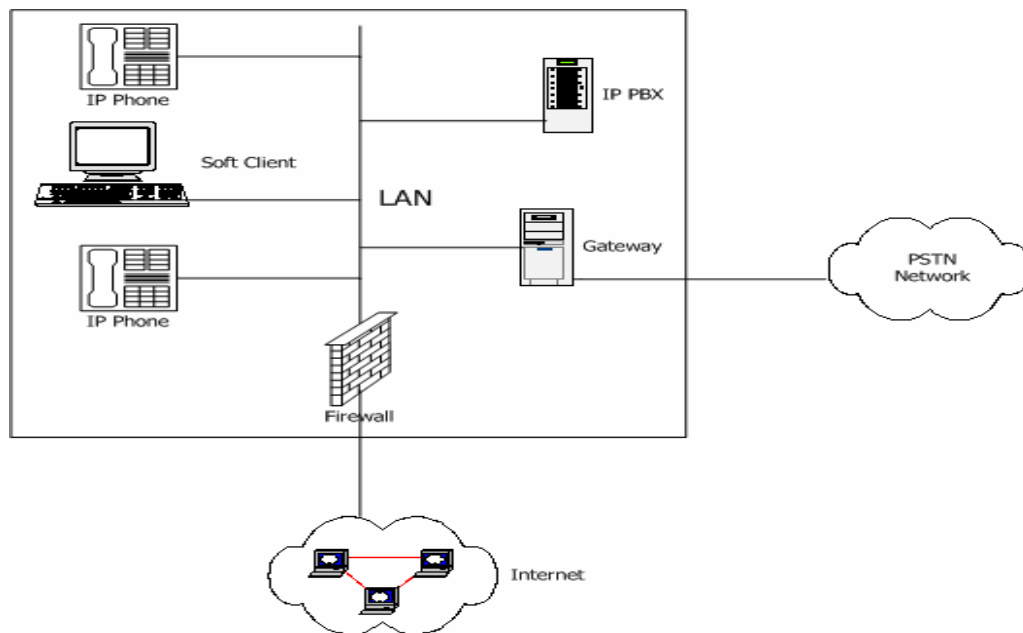


Figure 1 – VoIP network

2.1.2 VoIP Functions:

These are the basic functions of VoIP:

- Signaling
- Database services
- Call connect and disconnect
- CODEC operations

Signaling: Signaling is the way that devices communicate within the network, activating and coordinating the various components needed to complete a call. In a VoIP network, signaling is accomplished by the exchange of IP datagram messages between the VoIP components. The format of these messages may be dictated by any number of standard protocols [6].

Database Services: Database services are a way to locate an endpoint and translate the addressing that two (usually heterogeneous) networks use. A call control database contains these mappings and translations. A PSTN uses phone numbers to identify endpoints. A VoIP network uses an IP address (address abstraction could be accomplished with DNS) and port number to identify an endpoint [6].

Calls connect and disconnect: The connection of a call is made by two endpoints opening a communication session between one another. In the PSTN, the public (or private) switch connects logical (Digital Signal) DS-0 channels through the network to complete the calls. In a VoIP implementation, this connection is a multimedia stream (audio, video, or both) transported in real time. This connection represents the voice or video content being delivered. When a communication is complete, the IP sessions are released and optionally network resources are freed [6].

CODEC operations: Traditional voice communication is analog, while data networking is digital, as a result, the network needs a way to be able to convert the voice into a format that it can transport. Since the PSTN is often analog, this is not necessarily a major function, however, for VoIP, it is necessary for packetizing the voice. The process

of converting analog waveforms to digital information is done with a coder-decoder (CODEC, which is also known as a voice coder-decoder [VOCODER]). There are many ways an analog voice signal can be transformed, all of which are governed by various standards. The most common standards are G.711, G.721, G.729, G.723.1, etc. Each encoding scheme has its own history and merit, along with its particular bandwidth needs [6].

2.1.3 VoIP Components:

There are three major components of a VoIP network:

- Call Processing Server/IP PBX
- User End-Devices
- VoIP Gateways & Gatekeepers

Call Processing Server/ IP PBX: The call processing server, also known as an IP PBX, is the heart of a VoIP phone system, managing all VoIP control connections. VoIP communications require a signaling mechanism for call establishment, known as control traffic, and actual voice traffic, known as voice stream or VoIP payload. VoIP control traffic follows the client-server model, with VoIP terminals, including messaging servers that hold voice-mail messages representing the clients that communicate to the call processing servers. IP PBX also provides conferencing facility, call on hold facility, etc.

User End-devices: The user end devices are actually VoIP phones. VoIP phones can be soft phone (i.e. Computer) or hard phones (i.e. handsets or traditional phone)

VoIP Gateways & Gatekeepers: Gatekeepers are mainly used for call admission, call control & bandwidth management. Gateways are required to establish the connection between two different networks (i.e. between PSTN network & IP network).

2.1.4 VoIP Signaling Protocols:

VoIP signaling protocols are the enablers of the VoIP network. The protocols determine what types of features and functionality are available, as well as how all of the VoIP components interact with one another.

There are a variety of VoIP protocols and implementations, with a wide range of features that are currently deployed. Two major standards that are used for VoIP are H.323 & SIP. H.323 is the ITU standard for establishing VoIP connections, while IETF uses Session Initiation Protocol (SIP) as its standard.

Each of the voice protocols has its own strengths and weaknesses, and each takes a different approach to service delivery. Each of these protocols is successful in different products having a specific market focus [1].

2.2 Embedded Linux:

First question that might arise is that what is Linux? Strictly speaking, Linux refers to the kernel maintained by Linus Torvalds & distributed under the same name through the main repository and various mirror sites. The kernel provides the core system facilities. It may not be the first software to run on the system, as a boot loader may have preceded it, but once it is running, it is never swapped out or removed from control until the system is shut down. In effect, it controls all hardware and provides higher level abstractions such as processes, sockets, and files to the different software running on the system.

Now, second question might arise is that what is Embedded Linux? But before discussing about Embedded Linux, let me discuss about embedded system first. Embedded system is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function or some specific task. In some cases, embedded systems are part of a larger system or product. VoIP phone is one kind of Embedded System. Embedded system differs from traditional computer system in terms of size, performance, memory requirement, timeliness, cost, etc. Now, talking about Embedded Linux, there is no such thing like embedded version of kernel distributed by Linus Torvald. In Embedded Linux, kernel is configured as per specific hardware of Embedded System for a specific application. Embedded Linux kernel often include some optimization not found in the main kernel tree and are patched for support for some debugging tools such as kernel debuggers. Since Linux systems are made up of many components, let us take a look at the overall architecture of a generic Linux system (Fig-2).

Talking from lowest layer -Hardware, Linux requires at least a 32-bit CPU containing a memory management unit (MMU). Second, a sufficient amount of RAM must be available to accommodate the system. Third, minimal I/O capabilities are required if any development is to be carried out on the target with reasonable debugging facilities. This is also very important for any later troubleshooting in the field. Finally, the kernel must be able to load and/or access a root files system through some form of permanent or networked storage. Although the modified version of Linux called uClinux does run on

some CPUs that aren't equipped with MMU, The development of applications for Linux on such processors differs.

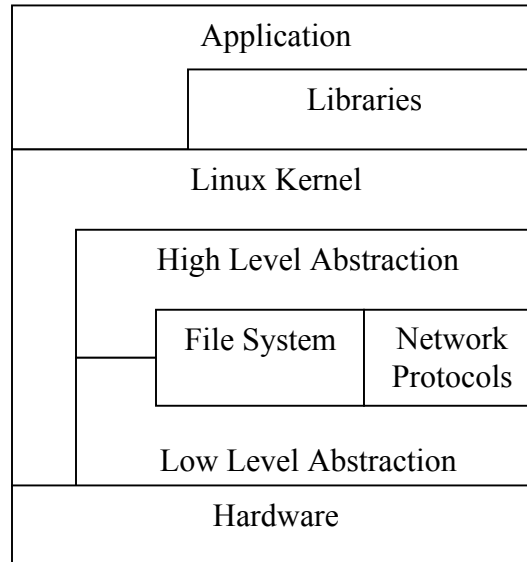


Figure 2- Architecture of a generic Linux system

Immediately above the hardware sits the kernel. The kernel is the core component of the operating system. Its purpose is to manage the hardware in a coherent manner while providing familiar high-level abstractions to user-level software. As with other Unix-like kernels, Linux drives devices, manages I/O accesses, controls process scheduling, enforces memory sharing, handles the distribution of signals, and tends to other administrative tasks. It is expected that applications using the APIs provided by a kernel will be portable among the various architectures supported by this kernel with little or no changes. This is usually the case with Linux, as can be seen by the body of applications uniformly available on all architectures supported by Linux. Within the kernel, two broad categories of layered services provide the functionality required by applications.

The low-level interfaces are specific to the hardware configuration on which the kernel runs and provide for the direct control of hardware resources using a hardware-independent API. That is, handling registers or memory pages will be done differently on an Intel system and on an ARM system, but will be accessible using a common API

to higher-level components of the kernel, with some rare exceptions. Typically, low-level services will handle CPU-specific operations, architecture specific memory operations, and basic interfaces to devices. Above the low-level services provided by the kernel, higher-level components provide the abstractions common to all UNIX systems, including processes, files, sockets, and signals. Since the low-level APIs provided by the kernel are common among different architectures, the code implementing the higher-level abstractions is almost constant regardless of the underlying architecture. There are some rare exceptions, as stated above, where the higher-level kernel code will include special cases or different functions for certain architectures.

Between these two levels of abstraction, the kernel sometimes needs what could be called interpretation components to understand and interact with structured data coming from or going to certain devices. File system types and networking protocols are prime examples of sources of structured data the kernel needs to understand and interact with to provide access to data going to and coming from these sources. Disk devices have been and still are the main storage media for computerized data. Yet disk devices, and all other storage devices for that matter, themselves contain little structure. Their content may be addressable by referencing the appropriate sector of a cylinder on a certain disk, but this level of organization is quite insufficient to accommodate the ever changing content of files and directories. File level access is achieved using a special organization of the data on the disk where file and directory information is stored in a particular fashion so that it can be recognized when it is read again. This is what file systems are all about. To accommodate these existing file systems and the new ones being developed, the kernel has a number of file system engines that can recognize a particular disk structure and retrieve or add files and directories from this structure. The engines all provide the same API to the upper layers of the kernel so that accesses to the various file systems are identical even though accesses to the lower-layer services vary according to the structure of the file system. The API provided to the virtual file system layer of the kernel by, for instance, the FAT file system and the ext2 file system is identical, but the operations both will conduct on the block device driver will differ according to the respective structures used by FAT and ext2 to store data on disk. During its normal operation, the kernel requires at least one properly structured file system, the root file

system. It is from this file system that the kernel loads the first application to run on the system. It also relies on this file system for future operations such as module loading and providing each process with a working directory. The root file system may either be stored and operated on from a real hardware storage device or loaded into RAM during system startup and operated on from there. As we'll see later, the former is becoming much more popular than the latter with the advent of facilities such as the JFFS2 file system. Yet the services exported by the kernel are often unfit to be used directly by applications. Instead, applications rely on libraries to provide familiar APIs and abstract services that interact with the kernel on the application's behalf to obtain the desired functionality. The main library used by most Linux applications is the GNU C library. For embedded Linux systems, substitutes to this library can be used. The major substitutes are uClibc, diet libc, etc. Libraries are typically linked dynamically with applications. That is, they are not part of the application's binary, but are rather loaded into the application's memory space during application startup. This allows many applications to use the same instance of a library instead of each having its own copy. The C library found on a the system's file system, for instance, is loaded only once in the system RAM, and this same copy is shared among all applications using this library. But note that in some situations in embedded systems, static linking, whereby libraries are part of the application's binary, is preferred to dynamic linking. When only part of a library is used by one or two applications, for example, static linking will help avoid having to store the entire library on the embedded system's storage device [2].

2.3 Kernel Optimization:

During project period, I have worked on kernel optimization. So, in this section I will discuss the basic theory of Linux kernel & kernel optimization.

2.3.1 Linux Kernel:

Kernel is the innermost portion of the operating system. The kernel is sometimes refers to as the supervisor or core of the operating system. Typical components of kernel are interrupt handlers to service interrupt handlers, a scheduler to share processor time among multiple processes, a memory management system to manage process address spaces, and system services such as networking and inter-process communication. Applications running on the system communicate with the kernel via system call. An application typically calls functions in a library- for example, the C library-that in turn relies on the system call interface to instruct the kernel to carry out tasks on their behalf. When application executes a system call, it is said that the kernel is executing on behalf of the application. Furthermore, the application is said to be executing a system call in kernel-space and the kernel is running in process context. Kernel also manages the system's hardware. When hardware wants to communicate with the system, it issues an interrupt that asynchronously interrupts the kernel. Interrupts generally are associated with a number. The kernel uses the number to execute a specific interrupt handler to process and respond to the interrupt [3].

In Linux, we can generalize that the process is doing one of three things at any given moment:

- In kernel-space, in process context, executing on behalf of a specific process.
- In kernel-space, in interrupt context not associated with a process, handling an interrupt.
- In user-space, executing user code in a process.

Linux kernels come in two flavors: stable or development. Stable kernels are production-level releases suitable for widespread deployment. Development kernels, on the other hand, undergo rapid change where anything goes. Linux kernel distinguishes between stable and development kernels with a simple naming scheme. For example, I

was using kernel-2.4.26 during my project. Here, 2 represents major release, 4 represents minor release & 26 represents revision. The minor number determines whether the kernel is a stable or development kernel. Even minor number is for stable version where as odd minor number is for development kernel [5].

2.3.2 Kernel Optimization:

Embedded System developer always tries to reduce the code footprint which is going to be embedded in the hardware. Size of software in embedded system is major concern. Embedded system developer always tries to remove redundant code in system. The redundant code is an existing program slice but is never reached, i.e. there is no control flow path to it from other parts of the system. Another case of redundant code is computing a value which contributes nothing, because the result is never used or is used but with no benefit to the system. The presence of redundant code in the Linux kernel may result from logical errors due to alterations in its control flow or from significant changes in the assumptions or environment of the program. Thus, the code that is redundant can be eliminated without causing any ill effects to the system.

Linux is a kind of GPOS that is designed to support a variety of popular functions; nevertheless, an embedded system is designed for specific purposes and for individual use only. According to this principle, when we use a GPOS (such as Linux) as an embedded system, there will probably be some redundant code. This redundant code will either be never executed (unreachable code) or contribute nothing (dead code) to the targeted embedded system. Therefore, for an embedded system, it is beneficial to eliminate redundant code. A typical approach to eliminating redundant code from the Linux kernel and adapt it to an embedded system is through manual processing of an existing Linux kernel. The developer scans the kernel source code first, and then, identifies and manually eliminates the redundant code line by line. The developer can control the code size and functionality of the Linux kernel. This is probably the most useful approach to getting the smallest possible Linux kernel for a specific embedded system. However, it is not so easy for a general user, and will take lots of time to do the job. Furthermore, it should be redone again when a new version of Linux kernel is released, or when the applications or the embedded platform is changed. Developer

should carefully decide which part of kernel code has to be eliminated every time the kernel customization is performed. Another drawback of this approach is that it is challenging to debug when system crashes.

The kernel of Unix-like systems is monolithic operating systems. Linux consists of a number of procedures which cooperatively perform jobs by calling each other. In short, the Linux kernel is a non-fixed structure. The monolithic property of this kind, huge and modifiable, is different from a typical program that has a fixed hierarchy. Hence, it becomes more complicated for a designer to predict the Linux kernel in advance. The first challenge of customizing a Linux kernel is to precisely understand its structure [7].

There might be several other issues related with customizing the Linux kernel:

- Most modern operating systems are constructed using a layer structure (see Fig. 2) and Linux is an example. Therefore, the Linux kernel serves as a medium layer. The Linux kernel, together with other layers such as applications, library and device drivers could also be reusable.
- The Kernel is a modularized and very large component. Abstracting the kernel in lines of code is not efficient. Procedures seem to be a suitable granularity and abstraction level.
- There is no rooted procedure (a rooted procedure is similar to the main (), which is a root of a C program) for the Linux kernel. Hence, the calling relations among the kernel's procedures are intricate.

According to research papers that I studied, there are many ways to optimize the kernel. First thing they are suggesting is that construct kernel's call graph. Then, by observing the kernel's call graph removes the system calls from the kernel source code which are not used by an application. Second thing they suggest is that identify the needed hardware devices for an embedded system. And according to hardware configuration, remove device initialization code for unneeded devices. Also, remove the device driver code of unneeded devices from kernel source code by properly configuring kernel. Third thing they suggest that identify unnecessary exception handlers in the kernel & remove it.

2.4 Kernel Performance Improvement:

Linux was originally designed as non-preemptive kernel and is not well suited for real-time applications, since processes may spend several milliseconds in Kernel mode while handling interrupts. This behavior is not acceptable for applications that require predictable and low response times.

For VoIP application, to improve the performance, Kernel should have low response time to I/O events. What is kernel response time?

Kernel response Time: It is the amount of time that elapses from when the interrupt is asserted to when the thread that issued the I/O request runs.

- Interrupt latency: Time between physical interrupt signal asserted & the interrupt service routine running.
- Interrupt handler duration: Amount of time spent routine that handles the interrupt.
- Scheduler latency: Time between interrupt service routine completing & scheduler function being run.
- Scheduler duration: Time spent in scheduler to decide what thread should run next & context switch to it.

Among all above four, major concern is scheduler latency. The reason is that the scheduler latency is in terms of mill second where as other three are in terms of micro second. So, for VoIP application, to have a good performance, scheduler latency should be minimized.

To reduce scheduler latency two mechanisms are used. First apply Preemption patch & secondly apply low-latency patch.

Preemption Patch:

The basic idea behind the preemption patch is to create opportunities for the scheduler to be run more often and minimize the time between the occurrence of an event and the running of the schedule() kernel function. The preemption patches do this by modifying the spinlock macros and the interrupt return code so that if it is safe to preempt the current process and a rescheduling request is pending, the scheduler is called. The preemption patch adds a variable to the task structure (the structure that maintains state

for each thread) named `preempt_count`. The `preempt_count` field is modified by the macros `preempt_disable()`, `preempt_enable()` and `preempt_enable_no_resched()`. The `preempt_disable()` macro increments the `preempt_count` variable, while the `preempt_enable()` macros decrement it. The `preempt_enable()` macro checks for a reschedule request by testing the value of `need_resched` and if it is true and the `preempt_count` variable is zero, calls the function `preempt_schedule()`. This function marks the fact that a preemption schedule has occurred by adding a large value to the `preempt_count` variable, calls the kernel `schedule()` function, and then subtracts the value from `preempt_count`. The scheduler has been modified to check `preempt_count` for this active flag and so short-circuit some logic in the scheduler that is redundant when being called from the preemption routine. The macro `spin_lock()` was modified to first call `preempt_disable()`, then actually manipulate the spinlock variable. The macro `spin_unlock()` was modified to manipulate the lock variable and then call `preempt_enable()`, and the macro `spin_trylock()` was modified to first call `preempt_disable()` and then call `preempt_enable()` if the lock was not acquired. In addition to checking for preemption opportunities when releasing a spinlock, the preemption patches also modify the interrupt return path code. This is assembly language code in the architecture specific directory of the kernel source that makes the same test done by `preempt_enable()` and calls the `preempt_schedule()` routine if conditions are right. The effect of the preemption patch modifications is to reduce the amount of time between when a wakeup occurs and sets the `need_resched` flag and when the scheduler may be run. Each time a spinlock is released or an interrupt routine returns, there is an opportunity to reschedule [9].

Low-Latency Patches:

The idea is to find places that iterate over large data structures and figure out how to introduce a call to the scheduler if the loop has gone over a certain threshold and a scheduling pass is needed (indicated by `need_resched` being set). This is done by dropping a spinlock, scheduling and then reacquiring the spinlock.

The low latency patches are a simple concept, but not so simple to implement. Finding and fixing blocks of code that contribute to high scheduler latency is a time intensive debugging task. Given the dynamic nature of the Linux kernel, the job of finding and

fixing high latency points in kernel code could be a full-time job. So, how does one find a high latency block of code? One tool is rtc-debug patch. This patch modifies the real time clock driver to look for scheduler latencies greater than a specified threshold and when it finds, it writes it to the system log file. Examining the syslog file and looking at the routines that show up the most leads to the long latency code blocks, patch replaces that code with efficient code which has minimum scheduler latency [18].

2.5 Kernel Modules:

Definition:

Kernel modules are dynamically loaded kernel function such as a file system or a device driver. Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system.

A module runs in the so-called *kernel space*, whereas applications run in *user space*.

Advantage:

Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.

Kernel modules Versus Application:

Whereas an application performs a single task from beginning to end, a module registers itself in order to serve future requests, and its “main” function terminates immediately. In other words, the task of the function *init_module* (the module’s entry point) is to prepare for later invocation of the module’s functions; it’s as though the module were saying, “Here I am, and this is what I can do.” The second entry point of a module, *cleanup_module*, gets invoked just before the module is unloaded. It should tell the kernel, “I’m not there anymore; don’t ask me to do anything else.” The ability to unload a module is one of the features of modularization that we’ll most appreciate, because it helps cut down development time; we can test successive versions of your new driver without going through the lengthy shutdown/reboot cycle each time.

Adding & Removing Kernel Modules:

We can add or remove kernel modules using ‘insmod’ or ‘rmmod’ command.

‘insmod’ installs a loadable module in the running kernel. ‘insmod’ tries to link a module into the running kernel by resolving all symbols from the kernel's exported symbol table.

Example:

```
insmod keypad.o
```

This will load module keypad into the running kernel by linking the functions to the kernel symbol table (Symbol table consist of the functions and global variable available to modules. Symbol tables are like the libraries available to the user space programs). We can see what modules are already loaded into the kernel by running 'lsmod', which gets its information by reading the file /proc/modules.

'rmmod' unloads loadable modules from the running kernel. 'rmmod' tries to unload a set of modules from the kernel, with the restriction that they are not in use and that they are not referred to by other modules [11].

Compiling Kernel Modules:

Kernel modules need to be compiled with certain gcc options to make them work.

Following gcc options need to be used.

- `-O2`: The kernel makes extensive use of inline functions, so modules must be compiled with the optimization flag turned on. Without optimization, some of the assembler macros calls will be mistaken by the compiler for function calls. This will cause loading the module to fail, since insmod won't find those functions in the kernel.
- `-W -Wall`: A programming mistake can take our system down. We should always turn on compiler warnings, and this applies to all our compiling endeavors, not just module compilation.
- `-isystem /lib/modules/`uname -r`/build/include`: We must use the kernel headers of the kernel we're compiling against. Using the default `/usr/include/linux` won't work.
- `-D__KERNEL__`: Defining this symbol tells the header files that the code will be run in kernel mode, not as a user process.
- `-DMODULE`: This symbol tells the header files to give the appropriate definitions for a kernel module [11].

2.6 File System Optimization:

A file system is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk. The word is also used to refer to a partition or disk that is used to store the files or the type of the file system.

Most UNIX file system types have a similar general structure, although the exact details vary quite a bit. The central concepts are super block, inode, data block, directory block, and indirection block. The super block contains information about the file system as a whole, such as its size (the exact information here depends on the file system). An inode contains all information about a file, except its name. The name is stored in the directory, together with the number of the inode. A directory entry consists of a filename and the number of the inode which represents the file. The inode contains the numbers of several data blocks, which are used to store the data in the file. There is space for a few data block numbers in the inode, however, and if more are needed, more space for pointers to the data blocks is allocated dynamically. These dynamically allocated blocks are indirect blocks; the name indicates that in order to find the data block, one has to find its number in the indirect block first. Like UNIX, Linux chooses to have a single hierarchical directory structure. Everything starts from the root directory, represented by /, and then expands into subdirectories instead of having so-called 'drives'. Such a file system is called a hierarchical structure and is managed by the programs themselves (program directories), not by the operating system [10]. On the other hand, Linux sorts directories descending from the root directory / according to their importance to the boot process. In Linux, root file system contains specific directory structure. Table-1 shows the Linux file system hierarchy.

Directory	Usage
/bin	Essential command binaries
/boot	Static files of the boot loader
/dev	Device files
/etc	Host-specific system configuration
/lib	Essential shared libraries & kernel modules
/media	Mount point for removable media
/mnt	Mount point for mounting file system
/opt	Add-on application software packages
/sbin	Essential system binaries
/tmp	Temporary files
/usr	User's directory
/var	Variable data

Table 1 - Linux File System Hierarchy

Now, In Embedded Systems, file system has to be optimized according to embedded application requirement. Generally embedded systems are not multi-user systems. So, In Embedded system '/home', '/opt' & '/root' directory is not required which is actually for multi-user environment. Also, variable data don't required to store in embedded system, so '/var' directory can be omitted. Size of file system can further be optimized by keeping the library files which is required by an application. Size of library files can be reduced by using uClibc. By keeping only required command binaries & system binaries in '/bin' & '/sbin' directory respectively. To further optimize the file system, the system startup scripts can be re-written according to system requirement. Now, file system has to be converted in appropriate file system type before porting it onto flash. There are many formats like CRAMFS, JFFS2, ROMFS, ext2 are used now a days. But, in embedded system, JFSS2 is widely used because it has very good features like 'power down reliability', 'persistency', 'write capable' etc. compared to other file system types.



CHAPTER 3

TOOLS OVERVIEW

3.1 Hardware Tools:

TS-7250 Board:

TS-7250 is a compact, full featured Single Board Computer (SBC) based on the Cirrus9302 ARM9 CPU. The EP9302 features an advanced 200 MHz ARM920T processor design with a memory management unit (MMU) that allows support for high level operating system such as Linux, Windows CE and other embedded operating systems. The ARM920T's 32-bit architecture, with a five-stage pipeline, delivers very impressive performance at very low power [13].



Figure 3 - TS-7250 board

The TS-7250 board has following features:

- 200 MHz ARM9 processor with MMU
- 2 standard serial ports with 16 Byte FIFO
- Watchdog timer unit
- 32 MB of High Speed SDRAM
- 32 MB Flash disk used for RedBoot boot-loader, Linux kernel and root file system
- USB Flash drive supported
- 10/100 Ethernet interface - auto sense, LED indicators
- 2 USB host ports
- 20 DIO lines - plus one output capable of switching 1 Amp at 30 V
- 5 channel 12-bit A/D converter
- PC/104 8/16 bit bus
- SPI bus header
- Dimensions are 3.8 " x 4.5 " (PC/104 mounting holes)
- Power requirements are 5V DC @ 400mA
- Operating Temperature Range: Fanless -20° to +70°C
- Optional Battery Backed Real Time Clock
- Optional RS-485 support on COM2 (full or half duplex) with fully automatic TX enable control
- Optional 8 channel 12-bit A/D converter with selectable input ranges
- Optional on-board temperature sensor

3.2 Software Tools:

Operating System:

Host Operating System: Linux based Debian Operating system (version- sarge 3.1) is host operating system. Debian Operating system has a very good support for the tools required for embedded system development. It is also very user friendly. So, I have chosen Debian operating system as my host operating system. I have used kernel-2.4.27 version in the Debian OS.

Target Operating System: I have used Embedded Linux as a target operating system. I have used kernel-2.4.26 (which is supplied by Technologic Systems) as a target operating system kernel.

Development language:

I have used C language & Linux shell script for development.

Cross-Compiler tool chain:

As the host computer & target system has different architecture, all applications that are developed on host system, must be cross-compiled so that they can run on target ARM processor. To cross-compile the application, Cross-compiler tool chain is needed. I have developed my own cross-compiler tool chain using gcc version-2.95.3 & glibc-2.3.1.

Terminal Emulator:

The most common way to communicate with an embedded system is to use a terminal emulation program on the host to communicate through a serial port with the target. I have used minicom as a terminal emulator.



CHAPTER 4

METHODOLOGY

4.1 Architecture:

Project VoIP aims to develop a VoIP phone. First task in developing VoIP phone is to make the TS-7250 development board working as IP phone. Then second task is to replicate the TS-7250 board in some extent & make our own prototype board and make it work as IP phone.

Let us first look at hardware architecture of IP phone that we are developing.

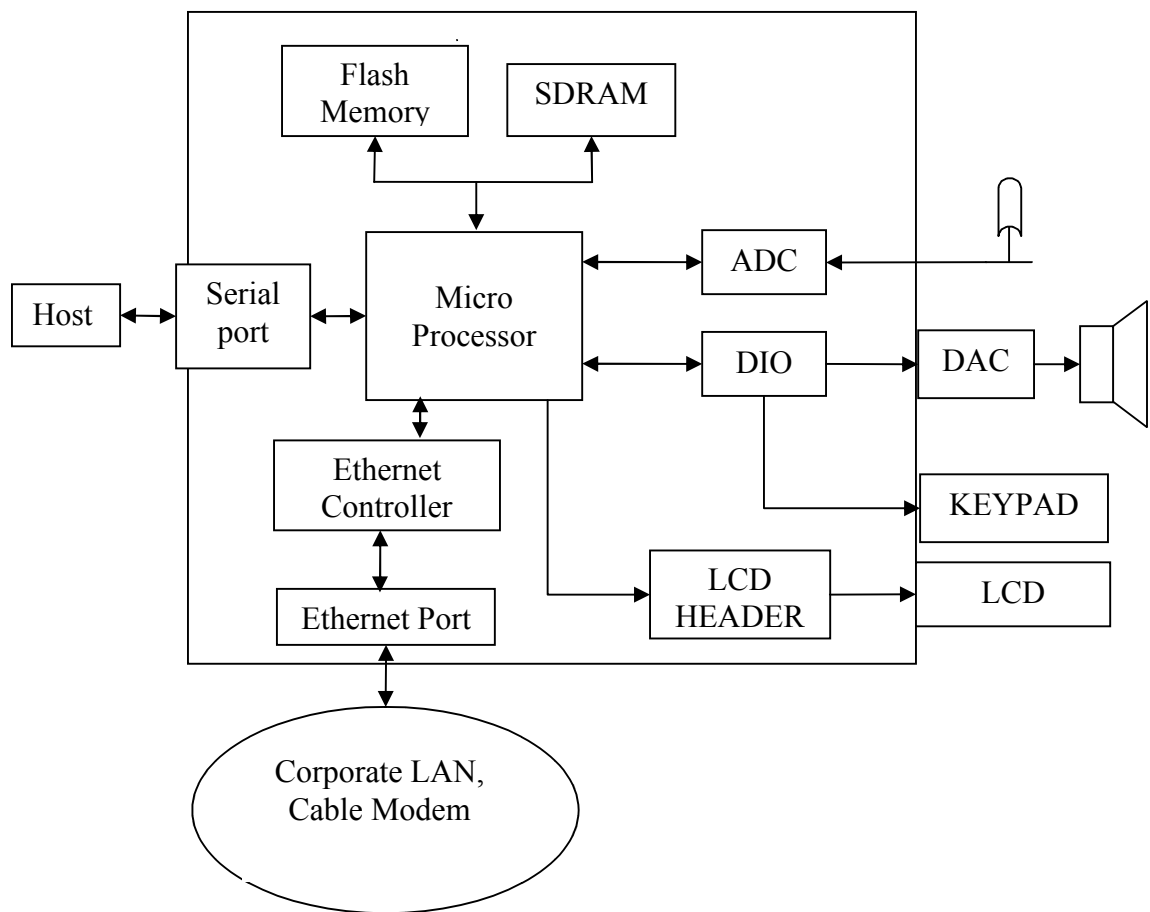


Fig -4 - Hardware Architecture of VoIP phone

As shown in the figure-4, central part of IP phone is the microprocessor. In Project VoIP, it is ARM9 processor. Flash is used to store the operating system & VoIP application. Ethernet controller is used to connect the IP phone to the IP network. Keypad & LCD is used to input & output the data respectively. Keypad is connected to processor via DIO (Digital Input Output) pins. Serial port is used for testing & debugging purpose. It will not be kept in final version of IP phone that is to be developed. ADC (Analog to Digital converter) is used to give voice input to the phone. Actually, ADC is connected to microphone. ADC takes the analog input from microphone & converts it to digital format & sends it to VoIP application. Similarly, DAC is connected to speaker. It takes digital voice from VoIP application & converts it to analog format & sends it to speaker for audio output.

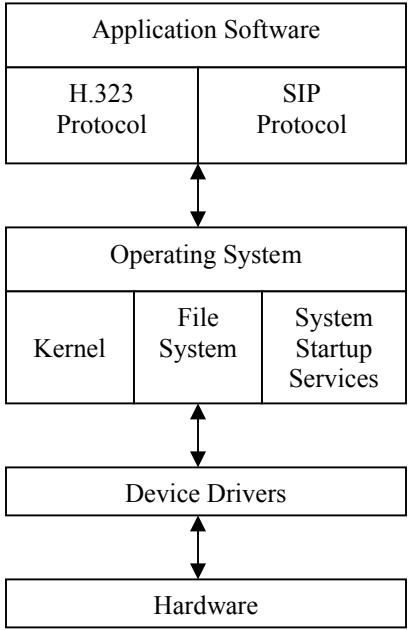


Fig-5 - Software Architecture for VoIP phone

As shown in figure, there are four main blocks. This thesis work is concentrated on second block of fig-5, which is related to operating system.

4.2 Implementation Approach:

When Project VoIP was started, I have been given tasks that have to be done during whole project period. I have organized work in module wise manner which is shown in fig-6.

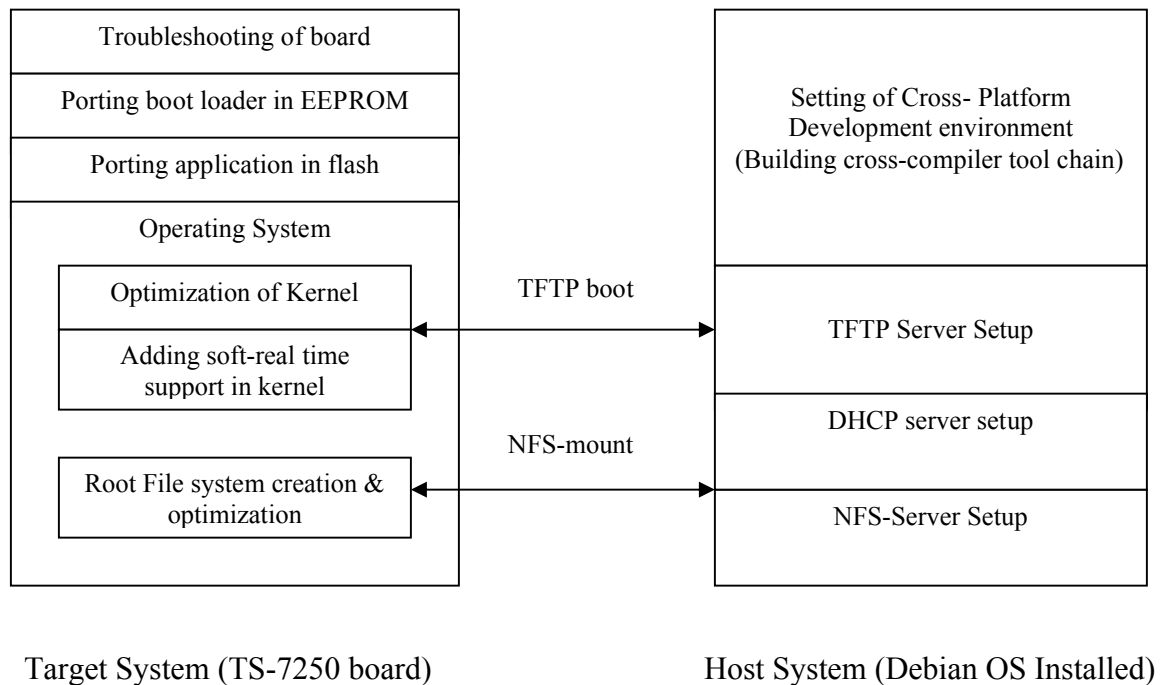


Fig-6 - module wise work distribution

In the starting phase of project, necessary setups have to be done in host environment. Cross-platform development is a major issue in any type of embedded system development. So, I started my work with building a cross-compiler tool chain. Once it is done, the next task was how to access the development board on host computer. For that, I have installed & configured terminal emulator (e.g. minicom) on host computer. I set appropriate parameters (i.e. baud rate=115200, parity=none, data bits=8 & stop bit=1) to have a proper access of development board on host computer via serial port. Now, progressing towards actual project aim, first thing was to optimize the kernel. Why it is

needed to optimize the kernel? As a VoIP-project team, our actual task was to develop a prototype board which is exact replica of TS-7250 board & which will work as IP phone. Memory would be major concern in prototype board. So, it is needed to reduce the size of kernel & file system as much as possible. To optimize the kernel, first thing that has to be done is to study the hardware details of development board. Then configuration & cross-compilation of the kernel according to hardware specification has been done. Removal of unnecessary drivers from the kernel source is being carried out next. Then I have used appropriate compiler optimization & stripping techniques to optimize the kernel. Then, to further optimize the kernel, I removed 'printk' statements from kernel source code. This is my final step towards optimizing the kernel by size. The detail description of each & every step of kernel optimization is given in chapter-5 in this document. Now, after optimizing the kernel, the next question that arises was how to test whether the kernel is working or not? For that, booting method like TFTPBOOT is used. TFTPBOOT is a method, by which the development board can be booted using the kernel resided in host computer provided that the kernel which is resided on host pc is cross-compiled & configured according to hardware specification of development board. So, to test the kernel, I have setup the TFTP server on host computer & boot the development board using TFTP boot.

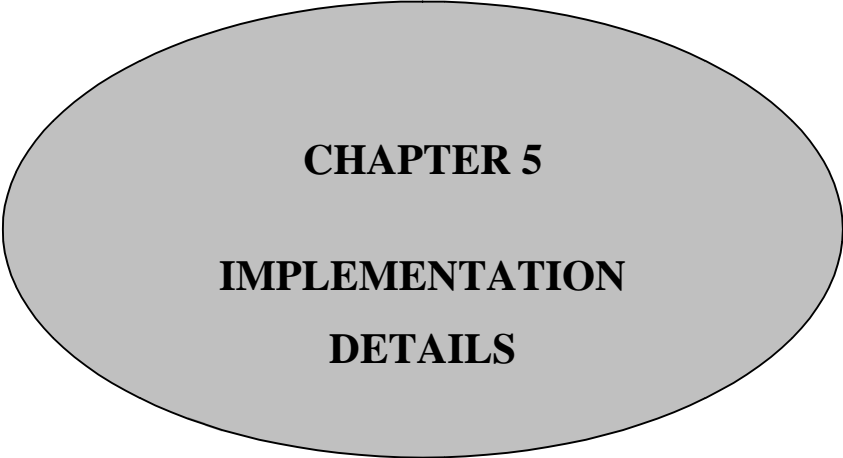
Now, second task in project journey was to improve the performance of kernel. By default, Linux kernel is not designed for real-time application. I have not provided any hard real-time support in kernel. Hard real-time support can be provided by giving higher priority to a specific task so that kernel runs that specific task more frequently & thereby reducing overall latency. Apart from hard real-time support, to have real-time support, soft-real time support can also be provided. So, I tried to provide soft-real time support in the kernel. Provision of soft-real time support means to reduce the scheduler latency in the kernel. To reduce the scheduler latency, there are many patches (i.e. preemption patch, low-latency patch, rtc-debug patch, etc) available. But, these patches are not compatible with ARM architecture & the kernel-2.4.26 is used in TS-7250 board & target VoIP phone as well. So, I have made these patches compatible with ARM architecture & kernel-2.4.26. Therby, soft-real time support is added in the kernel.

After finishing kernel stuff, next task was to concentrate on root file system.

Optimization of the file system has to be done. For that, I have created root file system & optimized it. Step-wise description of file system optimization is given in the chapter-5 of this document. After creating and optimizing the file system, next question that arise in mind was that how to test the file system? Using NFS mount it is possible to mount the file system resided in host computer. To NFS-mount the file system, first thing is to setup DHCP server & NFS server. So, setup of DHCP server & NFS server has been done. The detail step-wise description of NFS-mounting is given in chapter-5 in this document. This way, following above approach, project work has been finished.

4.3 Implementation Challenges:

- The first challenge of customizing a Linux kernel is to precisely understand its structure.
 - Most modern operating systems are constructed using a layer structure and Linux is an example. Therefore, the Linux kernel serves as a medium layer. The Linux kernel, together with other layers such as applications, library and device drivers could also be reusable.
 - The Kernel is a modularized and very large component. Abstracting the kernel in lines of code is not efficient. Procedures seem to be a suitable granularity and abstraction level.
 - There is no rooted procedure (a rooted procedure is similar to the main (), which is a root of a C program) for the Linux kernel. Hence, the calling relations among the kernel's procedures are intricate
- Writing a boot loader in a flash was challenging task for me. During project period, we faced problem with development board. Boot loader in EEPROM got corrupted. The boot loader is available but it is just for serial booting. So, I had to make some changes in boot loader so that it works for FLASH booting.
- There is a file system for flash available on the internet. But, directly optimization of that file system is not possible because of the library linking of some file. So, challenge for me was to create own file system & optimize it.
- While creating a file system, whatever modules I have installed, they need to be checked for module dependencies. But, by default in host system the utility 'depmod' checks the module dependencies for host computer. Finding depmod.pl script for cross-compiled modules & configuring it was a major challenge.



CHAPTER 5
IMPLEMENTATION
DETAILS

In this section, I have described the tasks that have been carried out during project period.

5.1 Creation of Project Workspace:

First & most important thing for any project is to create a workspace on host computer. To develop & customize software for target VoIP phone, various software packages & project components needs to be organized in a comprehensive and easy-to-use directory structure. Table-2 shows a directory structure that I used on host machine.

Directory	Content
bootldr	The boot loader for target
build-tools	The packages & directories needed to build the cross-platform development tool chain
debug	The debugging tools for target.
images	The binary images of boot loader, the kernel & the root file system to be used on the target
kernel	Kernel source code
rootfs	The root file system
sysapps	The system applications required for our target
tools	The complete cross-platform development tool chain

Table-2 – Project Work Space

The above directory structure is created /home/ragin/voip-project directory on host system. Then, it has been needed to set various paths for certain utilities to build-up. The following path settings have been kept:

```
export PROJECT=voip-project
export PRJROOT=/home/ragin/voip-project
export TARGET=arm-linux
export PREFIX=/home/ragin/voip-project/tools
export TARGET_PREFIX=/home/ragin/voip-project/tools/arm-linux
export PATH=/home/ragin/voip-project/tools/bin:${PATH}
```

5.2 Building a Cross-Compiler tool-chain:

The tool chain is needed to cross-develop applications for any target. As far as voip-project is concerned, target processor is ARM, whereas host processor on which application was developed is Intel processor. So, It has been needed to cross-compile each and every program that was going to be run on target platform. Cross-compiler tool-chain is required to cross-compile the program which is developed on one platform & is going to be run on different platform.

Cross-compiler tool-chain includes: binary utilities, C compiler, C library glibc & Kernel Header. I used Kernel 2.4.26, binutils 2.10.1, gcc 2.95.3 glibc 2.1.3 & Patches.

Steps to build the tool-chain:

- Kernel Header setup
- Binary utilities setup
- Bootstrap compiler setup
- C library setup
- Full compiler setup

Note: Here compiler seems to be built twice because C++ requires glibc support. So, First compiler is built with support for C only. Then glibc is setup & then full compiler with c++ support is built.

The detailed description of above steps is as follows:

[1] Preparing build-tools directory:

```
$ cd /home/ragin/voip-project/build-tools
$ mkdir build-binutils build-boot-gcc build-glibc build-gcc
```

[2] Kernel Headers setup:

```
Then, configuration of the kernel is required to build kernel headers.
$ cd linux-2.4.26
$ make ARCH=arm CROSS_COMPILE=arm-linux menuconfig
```

Here during the kernel configuration, the most important thing that has to set is the target processor & system type. Just setting the processor & system type is usually enough to generate the appropriate headers for the tool-chain build.

Create 'include' directory required for tool-chain & copy the kernel headers to it.

```
$ mkdir -p ${TARGET_PREFIX}/include
$ cp -r include/linux/ ${TARGET_PREFIX}/include
$ cp -r include/asm-arm/ ${TARGET_PREFIX}/include/asm
$ cp -r include/asm-generic/ ${TARGET_PREFIX}/include
```

[3] Binutils setup:

The binutils package includes the utilities most often used to manipulate binary object files. This package contains utilities like assembler as, linker ld, stripping utility strip etc. To build binutils package, following commands needs to be run.

```
$ cd build-binutils
```

To configure the package:

```
$ ../binutils-2.10.1/configure --target=$TARGET --prefix=$PREFIX
```

Here target & prefix option is used to specify target & installation directory respectively.

To build & install the package:

```
$ make
$ make install
```

[4] Bootstrap compiler setup:

Here, it is required to setup bootstrap compiler first, which supports only C language. Later, once C library has been compiled, gcc is recompiled to provide full C++ support. Here compiler has to be built twice because C++ requires glibc support. So, First compiler is built with support for C only. Then glibc is setup & then full compiler with C++ support is built. The following commands need to be run to setup bootstrap compiler.

```
$ cd build-boot-gcc
$ ../gcc-2.95.3/configure --target=$TARGET --prefix=${PREFIX} \
> --without-headers --with-newlib --enable-languages=c
```

At this stage, there are no system header files for the target (they would be available once the gcc is built),It has been needed to use the '--without-headers' option. It is also needed to use the '--with-newlib' option to tell the configuration utility not to use glibc, since it has not yet been compiled for target. The '--enable-languages' option tells the

configuration script which programming languages the resulting compiler has to support.

Then, building & installation of the bootstrap compiler has been carried out using following commands.

```
$ make all-gcc
$ make install-gcc
```

[5] C library setup:

C library has been setup using following commands.

```
$ cd build-glibc
$ CC=arm-linux-gcc ../glibc-2.2.3/configure --host=$TARGET \
> prefix="/usr" --with-headers= ${TARGET_PREFIX}/include
```

Here '--with-headers' option is used to tell the configuration script where to find the kernel headers that has been set up earlier.

Then, to build & install C library following commands need to be run.

```
$ make
$ make install_root=${TARGET_PREFIX} prefix="" "install
```

The last step that has been carried out to finalize glibc's installation: the configuration of the libc.so file. This file is used during the linking of applications to the C library. It contains references to the various libraries needed for real linking. The installation carried out by 'make install' assumes that the library is being installed on a root file system and hence uses absolute pathnames in the libc.so link script to reference the libraries. Since I have installed the C library in a nonstandard directory, Modification of the link script was necessary so that the linker would use appropriate libraries.

In its original form, libc.so looks like this:

```
/* GNU ld script */
GROUP (/lib/libc.so.6 /lib/libc_noshared.a)
```

I modified that script so that it works properly. This is the modified one.

```
/* GNU ld script */
GROUP (libc.so.6 libc_noshared.a)
```

Thus, by removing the references to an absolute path, I was forcing the linker to use the libraries found within the same directory as the libc.so script.

[6] Full compiler setup:

The last step that has been carried out was to setup full compiler which supports both C & C++ languages.

```
$ cd ${PRJROOT}/build-tools/build-gcc
$ ../gcc-2.95.3/configure --target=$TARGET --prefix=${PREFIX} \
> enable languages=c, c++
$ make all
$ make install
```

After following above steps, Cross-compiler tool chain has been created successfully. The cross-compiler utilities that were installed in \${PREFIX}/bin directory are as follows.

Utility	Usage
arm-linux-as	The GNU assembler
arm-linux-ld	The GNU linker
arm-linux-ar	Creates & manipulates archive content
arm-linux-readelf	Display information about the content of object files
arm-linux-size	Lists the size of sections within an object file
arm-linux-strip	Strip symbols from object files
arm-linux-gcc	Cross-compile the programs written in C language
arm-linux-g++	Cross-compile the programs written in C++ language

Table-3 – Cross-compiler tool-chain contents

5.3 Kernel Configuration & Optimization:

5.3.1 Kernel Configuration for x86 Architecture:

As per project definition, it has been needed to optimize the kernel for ARM architecture. But before going towards cross-platform optimization of kernel, I did configuration of kernel for x86 architecture. At the beginning of the project, I was new at kernel stuff. So, very first thing I did was tried to understand the configuration mechanism of kernel on x86 architecture. I configured the kernel for x86 architecture first. This configuration of kernel for x86 architecture also helped me at the later stage of project when I was trying to provide soft-real time support in the kernel.

These are the steps that I followed for kernel configuration for x86 architecture.

- First, install the required packages & kernel sources.

```
$ cd /usr/src
```

```
$ apt-get install gcc libc6-dev tk8.3 libncurses5-dev fakeroot
```

```
$ apt-get install kernel-package kernel-source-2.4.27
```

- Then untar the kernel source.

```
bash:/usr/src$ tar -xjvf kernel-source-2.4.27.tar.bz2
```

- Create a symlink

```
bash:/usr/src$ ln -s kernel-source-2.4.27 linux
```

- Check the current minimal requirements

For that the 'ver_linux' script has to be run. It compares the current minimal requirement as per /Documentation/Changes file & shows which packages are to be upgrade or replace.

```
$ ./ver_linux
```

-Configure the kernel.

To configure the kernel we can use 'make config' or 'make xconfig' or 'make menuconfig' command. I used

```
$ make menuconfig
```

Above step shows the 'ncurses' based menu for configuration. I enabled or disabled various options during these steps as per the H/W specification of the computer that I was using.

-Building the kernel image.

To build the kernel image 'make-kpkg', a script which automates & replaces the sequence "make dep; make clean; make bzImage; make modules" is used. With the 'make-kpkg' command, I used the '--append-to-version' option to append my own kernel version to the original kernel version.

```
$ make-kpkg -- append-to-version=.101205
```

-Making the kernel image.

```
$ make-kpkg clean
```

```
$ make-kpkg --append-to-version=.101205 kernel_image
```

-Installing the kernel-image package.

```
$ dpkg -i kernel-image-2.4.27.101205_10.00.Custom_i386.deb
```

- Removing the symlink.

```
$ rm linux
```

- Removing the old kernel.

```
$ dpkg -P kernel-image-2.4.27
```

- Reboot computer.

This way, configuration of the kernel for x86 architecture has been done.

5.3.2 Kernel Configuration for ARM architecture:

To configure the kernel for ARM architecture, first kernel needs to be cross-compiled. In Project VoIP, development board is TS-7250 board. The TS-7250 board manufacturer provided the kernel source of kernel-2.4.26. I had to use this kernel only because the TS-7250 board has CPLD programmed chip which is proprietary of Technologic Systems. The board manufacturer has made changes in the kernel source code according to this CPLD chip & onboard timer.

To configure the kernel for ARM architecture, following steps have been followed:

- Downloaded the kernel source file 'linux24-ts9-ksrc.tar.gz'
- Uncompressed the kernel source file.
- Edited top-level Makefile in kernel source code to Cross-compile the kernel.

Following variables in Makefile have been setup:

- ARM=arm
- CROSS_COMPILE='path of tool chain'
- To automate the kernel configuration according to TS-7250 board:
\$ make ts7250_config
- To configure the kernel:
\$ make menuconfig
- Then to check the dependencies:
\$ make dep
- Finally, to make kernel executable:
\$ make vmlinux

After following the above steps, the kernel image 'vmlinux' is built. The resultant kernel was of size **1.8MB**.

5.3.3 Kernel Optimization for ARM architecture:

After configuring the kernel as mentioned above, the kernel size was 1.8 MB. Now, in final prototype board for VoIP phone, memory would be major concern. So, kernel needs to be optimized. The steps that I followed to optimize the kernel are as follows:

Step 1: Removing unnecessary H/W support & driver support that is not required for target H/W:

I studied the details of Ts-7250 hardware. I noted down the H/W support & Driver support that is required for target hardware & for VoIP application. Now, kernel source code contains all driver support & all type of H/W support because it is designed for general purpose. But, as far as our project is concerned, all supports is not required. We require the supports that are specific to TS-7250 board & VoIP application.

The supports that I removed from kernel during kernel configuration are as follows:

- Support for hot-pluggable devices.
- Sysctl support.
 - This support is required if it is needed to change the kernel parameters when kernel is running.

- Compressed boot loader in ROM/flash option.
 - This option is required if it is needed to keep compressed boot loader in FLASH/ROM. Compressed boot loader in FLASH is not required for target VoIP phone.
- Parallel port support.
 - This support is required if it is needed to connect parallel port to our device. This is not required for VoIP phone.
- Loop back device support.
 - This support is required if we want to create a file system on block device (like cdrom, floppy etc.) & mount it.
- 802.1Q VLAN support.
- PPP (point-to-point protocol) support.
- SCSI support.
 - This support is required if it is needed to connect SCSI hard-disk, SCSI cdrom, etc. to the computer
- IrDA subsystem support.
 - It is required if we want to connect any blue-tooth devices to the board.
- Wire-less LAN support.
- Input core support (like Mouse support)
- Dis-contiguous memory support.
- File System support
 - DOS FAT fs support.
 - MSDOS fs support.
 - VFAT fs support.
 - ‘/proc’ file system support
 - This is virtual file system. This helps to view and modify the kernel & system parameters at run time. It is not required such type of facility in VoIP phone.
 - ‘/dev/pts’ file system support
 - This is related to virtual file system.

- Second extended file system(ext2) support
 - We want to have a YAFFS (Yet Another Flash File System –similar to journaling flash file system) or JFFS2 in flash. Ext2 file system is not required in target VoIP phone.
- Network File System (NFS) support
- USB support.
 - USB port is not required in target VoIP phone.
- Bluetooth support.
- Plug and Play support
- RAID support
 - This is required if it is needed to connect Redundant Array of Multiple disk to computer.
- WAN support
- TS-7kv serial port support
- Support for more than 4 serial ports
- Initial RAM Disk(initrd) support
 - Initial RAM disk is a RAM disk that is loaded by boot loader & it is mounted as root before the normal boot procedure. It is used load modules needed to mount the “real” root file system. In VoIP phone, file system is going to be mounted from flash.
- Kernel Hacking Support
 - Verbose fault handling support.

After doing this, when I cross-compiled the kernel the resultant kernel was of size **1624KB**.

Step-2: Using appropriate compiler flags & stripping options.

I used appropriate compiler flags 'Os', 'fdata-sections', 'ffunction-sections'. I edited the top-level Makefile in kernel source code & set the compiler flags following way.

- HOST_CFLAGS= -Os -fdata-sections -ffunction-sections

-ffunction-sections was originally designed to allow for the linker to reorder functions to improve icache & tlb behavior. It turns out the same technology (putting each function into its own section) greatly simplifies the garbage collection of unused functions. Similarly, 'fdata-sections' option places each function or data item into its own section in the output file if the target supports arbitrary sections. Both of the above option reduces the size of the object file.

Then, I used arm-linux-strip utility from the cross-compiler tool-chain that I built. Actually, any binary file of ELF format contains following sections.

Section	Description
.text	Executable code
.bss	Un-initialized data
.data	Initialized data
.stab (Symbolic table)	Contains debugging information
.stabstr	Contains debugging information
.shstrtab	Section header string table
.symtab	Contains debugging information
.strtab	Contains debugging information

Table-4 – Binary Executable format

The 'arm-linux-strip' utility of cross-compiler tool-chain removes '.stab, .stabstr, .symtab, .strtab' sections. Also, '.shstrtab' would be changed because it shrinks in size since there are fewer sections in binary now.

When I applied 'arm-linux-strip' to kernel executable, the kernel size is reduced to **1284 KB**.

Step-3: Removed unnecessary 'printk' statements from kernel source code.

There are around 50000 lines of 'printk' statements in kernel source code. These statements are for debugging purpose. These statements are not required for VoIP application. So, I comment out the most of 'printk' statements in kernel source code. For that I have written one shell script.

The content of shell script is as follows:

```
#!/bin/sh
cd /home/ragin/printk_remove/linux24/$1
for fl in *.c; do
mv $fl $fl.old
sed 's_printk_//printk_' $fl.old > $fl
rm -f $fl.old
done
```

After running this script, when I compiled the kernel, it showed error. This is because the above script comments the printk statements which are of one line. In many files of kernel source code, printk statements split between two or more lines. So, I had to manually correct those errors by manually comment out those printk statements.

After commenting out printk statements & compiling the kernel, the total size of the kernel was reduced by around 130 KB.

Step-4: Used compressed image of kernel.

As explained in section-5.3.3, I configured the kernel. But this time I used 'make bzImage' instead of 'make vmlinux' while making of kernel image.

'make bzImage' command make compressed image of kernel. The resultant kernel image 'vmlinux' would be resided in '/linux/arch/arm/boot/compressed' directory in kernel source directory. The resultant kernel image was of size **448KB**.

5.3.4 Booting of Kernel using 'TFTPBOOT'.

As I explained above that I configured & optimized the kernel for ARM architecture. The big question came at that time is 'How would I test the kernel?' Using 'TFTPBOOT', it is possible to test the working of kernel resided in host pc.

'tftp' server has been downloaded in host pc. Then, configuration of 'tftp' server is being carried out. As a result development board was able to boot using kernel resided in host pc. The steps for configuring 'tftp' server are as follows:

[1] Download & install 'tftp' server on host platform

```
$ apt-get install tftp
```

```
$ apt-get install tftpd
```

[2] Configure the 'tftp' server.

- Edit the file /etc/inetd.conf by adding the line:

```
'tftp dgram udp wait root /usr/sbin/tcpd
```

```
/usr/sbin/in.tftpd /home/ragin/voip-project/images/'
```

[3] Restart the system services.

```
$ inetd restart.
```

Now, steps to boot the development board from the kernel resided in host pc using 'tftpboot' are as follows.

-Put cross-compiled kernel image 'vmlinux' in '/home/ragin/voip-project/images/' directory in host pc.

-Restart the development board.

-Press 'Ctrl + C' just after restarting the dev. board to get control of boot loader.

-Now, on RedBoot (i.e. boot loader) prompt, type following to load kernel from host computer into memory of dev. board.

```
$ RedBoot> load -b 0x00218000 -h 192.168.0.25 vmlinux
```

Here, 192.168.0.25 is the IP address of host pc.

-Then to execute the kernel, type following on boot loader prompt.

```
$ RedBoot> go 0x00218000
```

5.4 Performance Improvement of Kernel:

The performance of kernel has been improved by adding soft-real time support. For VoIP application, to improve the performance; Kernel should have low response time to I/O events. To get low response time from kernel, kernel scheduler latency should be as minimum as possible. Reducing kernel scheduler latency means provision of soft real time support in the kernel.

To reduce the scheduler latency, there are many patches like low-latency patch, preempt-kernel patch, rtc-debug patch are available. But, the main problem is that whatever patches available for reducing scheduler latency are not for kernel-2.4.26 kernel & ARM architecture. So, I have modified patches & source code accordingly to apply it successfully.

To apply low-latency patch -> array.c, mmap.c & Makefile in kernel source code & patch have been modified.

To apply rtc-debug patch -> rtc.c, traps.c in kernel source code & patch have been modified.

To apply timepeg patch -> Makefile, traps.c, sched.h, threads.h in kernel source code & patch has been modified.

To apply preempt-kernel patches -> Makefile in kernel source code has been modified.

Procedure:

I have first applied preempt-kernel patch with necessary modifications as described earlier. To have effect of this patch, enabled 'Preempt Kernel' option under 'Processor types & features' menu during configuration of kernel.

Then, low-latency patch with necessary modifications is being applied. To have effect of this patch, enabled 'Low-latency scheduling' option under 'Processor types & features' menu during configuration of kernel. Then rtc-debug patch with necessary modifications is being applied.

Now, to observe the kernel performance, I have applied timepeg patch with necessary modifications. To have effect of this patch, enabled 'Timepeg instrumentation' option under 'Kernel Hacking' menu during configuration of Kernel.

To test the performance of kernel, I replicated above steps for x86 architecture. Then, the performance of kernel for x86 architecture has been tested. For this, download 'tpt' tool & build it & install it on pc. Now, I wanted to measure how long kernel takes to schedule a request from network device driver. I have modified 'asmlinkage long sys_sendto' system call in 'socket.c' file in kernel source code & put '**TIMEPEG ("sys_sendto");**' sentence at the beginning of system call.

Then, I have also modified network device driver file '8139too.c'. I put '**TIMEPEG('rtl8139_start_xmit');**' sentence in 'rtl8139_start_xmit' function in 8139too.c file.

Then I configured the kernel & installed it onto pc. Then I run 'tpt' from pc. This tool measures the time duration between two **TIMEPEG** sentences in kernel source code.

Now, one can enable or disable low-latency support manually. To enable low-latency support command is: 'echo 1 > /proc/kernel/sys/lowlatency' & to disable low-latency support command is: 'echo 0 > /proc/kernel/sys/lowlatency'.

After running 'tpt', I observed the kernel performance & noticed that kernel scheduler latency is minimized by great extent. I have shown the results in the chapter-6 in this document.

5.5 Mounting File System using NFS-mount:

In VoIP project, there was a need to test the various applications like SIP, H.323 & device driver frequently. Sometimes, the application was big enough such that it can't be possible to copy it to flash & test it. Also every time copying the application to the flash is very tedious job. Also, for testing own created file system, NFS-mount is required.

So, what I did was that I mounted file system using NFS mount. Using NFS-mount it is possible to use file system that is resided on host pc. (Note: Here the file system which resides on host pc is same as the file system resides on flash.) After doing NFS mount, kernel would mount the file system from host pc instead of mounting file system from flash. Now, to mount the file system via NFS mount, configuration of DHCP server is also required.

The steps that have been followed to do the NFS mount are as follows:

- Download & install 'nfs-kernel-server'
- Edit the following line in '/etc/exports' file.

```
'/home/ragin/rootfs 192.168.0.0/255.255.255.0 (rw,no_root_squash)'
```

Here, 'rw' argument is used to give client machine to give read & write access. By default, any file request made by user root on the host computer is treated as if it is made by user nobody on the server. If no_root_squash is selected, then root on the client machine will have the same level of access to the files on the system as root on the server.

- Put the file system to be mounted in '/home/ragin/rootfs' directory.
- Edit the file '/home/ragin/rootfs/etc/fstab' & add the following line in it.

```
'192.168.0.25:/home/ragin/voip-project/rootfs/ / nfs exec,dev,suid 1 1'
```

- Then edit the file '/etc/hosts.allow' & give permission to client machine to access the network file system.
- Then, start the nfs-server.

```
$ /etc/init.d/nfs-kernel-server restart.
```

- Start the necessary daemons like- 'mountd, nfsd, portmapper'.

- Now, install & configure the DHCP server to assign IP address to the dev-board while mounting via nfs-mount. The steps to install & configure DHCP server are as follows:

-Download & install DHCP server

```
$ apt-get install dhcp
```

-Add the following lines into '/etc/dhcpd.conf' file.

```
' allow booting;
  allow bootp;
  option routers 192.168.0.1;
  option subnet-mask 255.255.255.0;
  option domain-name "voipdomain";
  option domain-name-servers 192.168.0.1;
  default-lease-time 30000;
  max-lease-time 30000;
  subnet 192.168.0.0 netmask 255.255.255.0 {
    range 192.168.0.50 192.168.0.51;
    host voip-03 {
      hardware ethernet 00:50:ba:a9:06:ac;
      fixed-address 192.168.0.25;
    }
  }'
```

- Then, start dhcp server.

```
$ dhcpd start
```

Now, restart the dev. board & get command of boot loader & load kernel using tftpboot & then load file system using nfs-mount.

```
$ RedBoot> load -b 0x00218000 -h 192.168.0.25 vmlinux
```

```
$ RedBoot> exec -c "console=ttyAM0, 115200 ip=dhcp nfsroot=
192.168.0.25:/home/ragin/voip-project/rootfs/"
```

Note: 192.168.0.25 is the IP address of host computer & 192.168.0.50 is the IP address of TS-7250 board.

5.6 Creation & Optimization of Root File System:

5.6.1 Root File System Creation:

In final prototype board for VoIP phone, memory would be major concern. So, file system with minimal functionalities has to be created. The way I created File System, is a very tedious job. These are the steps that have been followed to create file system:

- (1) Build GNU tool-chain for Cross-Compilation.
- (2) Use C Library Alternatives like uClibc.
- (3) Build & install Kernel & Kernel modules.
- (4) Build basic root file system structure.
- (5) Finalize the root file system setup & prepare it for NFS mount.

[1] Build GNU tool-chain for Cross-Compilation:

I already explained about how to create GNU tool-chain in section-5.2.

[2] uClibc setup:

As memory is a major concern in project VoIP & it was required to minimize the size of file system. So, I have used uClibc in file system. I downloaded ucLibc0.9.16. Then I have configured & cross-compiled it as per target H/W specifications & VoIP application. The main configuration menu includes the following submenus:

- Target Architecture Features and Options
- General Library Settings
- String and Stdio Support
- Library Installation Options

In 'Target Architecture Features and Options' menu, it is required to enable various options as per the target H/W specification.

In 'General Library Settings' menu, I set appropriate directory path settings like Linux kernel header, Shared library loader path, uClibc development environment directory, uClibc dev environment system directory, uClibc dev environment tool directory. I set it following way.

Linux kernel header location:

```
KERNEL_SOURCE=/home/ragin/voip-project/kernel/linux-2.4.27
```

Shared library loader path:

```
SHARED_LIB_LOADER_PATH=/lib
```

uClibc development environment directory:

```
DEVEL_PREFIX=/home/ragin/voip-project/tools/uclibc
```

uClibc development environment system directory:

```
SYSTEM_DEVEL_PREFIX=$(DEVEL_PREFIX)
```

uClibc development tool directory:

```
DEVEL_TOOL_PREFIX=$(DEVEL_PREFIX)/usr
```

Then, I configured uClibc & removed unnecessary functionality which is not required. Finally, I cross-compiled it & installed it in the `/${PREFIX}/tools` directory of the project work space. Then I copied all uClibc's components to the target's root file system using following commands:

```
$ cd ${PREFIX}/uclibc/lib
$ cp *.*.so ${PRJROOT}/rootfs/lib
$ cp -d *.so.[*0-9] ${PRJROOT}/rootfs/lib
```

[3] Build & install Kernel and Kernel modules:

I have already described the process for building the kernel. Here I will explain the method of creating & installing kernel modules. Move into the directory where the kernel source resides & type the following.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- \
> INSTALL_MOD_PATH=/home/ragin/voip-project/images/ \
> modules-2.4.27 modules_install
```

Once it is done copying the modules, kernel tries to build the module dependencies needed for the module utilities during runtime. Since 'depmod', the utility that builds the module dependencies, is not designed to deal with cross-compiled modules, it will fail. To overcome this problem, I downloaded the 'depmod.pl' script from busybox source from the internet & used it following way.


```

$ depmod.pl -k ./vmlinux -F ./System.map \
> -b /home/ragin/voip-project/images/modules-2.4.27/lib/modules > \
> /home/ragin/voip-project/images/modules-2.4.27/lib/modules \
> 2.4.27/modules.dep

```

This way, the modules have been installed in target root file system.

[4] Build basic Root File System hierarchy:

The basic root file system contains following directory structure.

<u>Directory</u>	<u>Content</u>
bin	Essential user command binaries
boot	Static files used by the boot loader
dev	Devices & other special files
etc	System configuration files, including startup files
home	User home directories
lib	C Library and Kernel modules
opt	Add-on s/w packages
proc	Virtual file system for kernel & process information
root	Root user's home directory
sbin	System administration binaries
usr	Application & documents useful to users

Now, as far as Embedded system development is concerned, one can omit '/home, /opt, /root' directories as multi-user environment is not required in embedded system. '/var' directory can be omitted as it is not required the variable data to be stored. The remaining directories '/bin, /dev, /etc, /lib, /proc, /sbin, & /usr' are essential. So I have created the above directories.

```

$ cd rootfs
$ rootfs> mkdir bin dev etc lib proc sbin usr

```

I have also created '/usr' directory structure.

```

$ rootfs> mkdir usr/bin usr/lib usr/sbin

```

Then, I copied the library files needed, from uClibc tool-chain that has been built, into the '/lib' directory of root file system. Then, I have copied kernel modules which are

created earlier into root file system structure. (i.e. copied it into '/lib' directory).

Now, talking about '/dev' directory, I have not created any devices in '/dev' directory. Instead, '**devfs**' utility has been used. I have cross-compiled the kernel with 'devfs' support enabled. Also, during configuration of busybox, I included 'devfsd' utility. Now, when root file system is mounted, kernel automatically creates devices in '/dev' directory.

In any file system, command binaries are needed. For that busybox-1.0 has been downloaded & configuration of busybox has been carried out.

```
$busybox-1.0> make TARGET_ARCH=arm CROSS=arm-linux-  
PREFIX=/home/ragin/rootfs all install
```

I enabled all necessary command binaries & system administration binaries to include it in '/bin' & '/sbin' directory. Above command will create '/bin' & '/sbin' directory with command binaries inside it. Finally, talking about most important directory -'/etc'. I have created two files 'inittab' & 'fstab' and one directory named 'init.d' which contains 'rcS' script. To mount root file system, kernel first calls init which will call '/etc/inittab' file. The content of my inittab file is:

```
::sysinit:/etc/init.d/rcS  
::respawn:/sbin/getty 115200 ttyAM0  
::restart:/sbin/init  
::shutdown:/bin/umount -a -r
```

First line tells to initialize the system via rcS script. Whenever system gets restarted, it will call /sbin/init which will call inittab file. When system gets shut down, this file will unmount the system using /bin/umount command.

The most important thing about my inittab file is removal of run level support, multi-user support. I kept it very simple.

Now, when system starts, inittab file calls '/etc/init.d/rcS'.

The content of my rcS script is as follows:

```
#!/bin/sh
PATH='/bin:/sbin:/usr/bin:/usr/sbin'
export PATH
/sbin/devfsd /dev
/sbin/ifconfig eth0 192.168.0.50
/bin/SIP_APPLICATION
```

Here 192.168.0.50 is the IP address of TS-7250 board. Above file creates the devices in '/dev' directory using 'devfsd' command. Then it will configure the IP address for the system. Then, without starting anything else it directly starts application (i.e. SIP_Application).

Finally, file '/etc/fstab' has been created to mount root file system using nfs-mount.

I kept the following content in fstab file:

```
'192.168.0.25:/home/ragin/voip-project/rootfs/ / nfs exec, dev, suid 1 1'
```

[5] Finalizing the File System setup & do NFS-Mount:

After doing all above steps, the root file system has been created successfully.

After finalizing the root file system setup, restart the development board & press 'Ctrl + c' to get command of boot loader. Then type following on RedBoot prompt:

```
$RedBoot> load -b 0x00218000 -h 192.168.0.25 vmlinux
$RedBoot> exec -c "console=ttyAM0, 115200 ip=dhcp \
nfsroot=192.168.0.25:/home/ragin/voip-project/rootfs"
```

After doing the above steps, dev. board mounts the root file system & directly starts application that is what required in target VoIP application.

5.6.2 Optimization of Root File System:

The file system that Technologic Systems has provided is of 7.6 MB. Also, the file system which I have created occupies 6.7 MB. It is too large for target VoIP phone. So, optimization of the file system according to VoIP application requirement has to be carried out. In the optimized root file system I have just kept '/bin, /dev, /sbin, /etc, /lib' directory.

In '/bin' directory I have kept the command utilities which is actually required for VoIP phone. The command utilities I kept in '/bin' directory are 'busybox, echo, ln, mknod, mount, sh, umount'.

Similarly, in '/sbin' directory I kept 'devfsd, getty, halt, init, ifconfig, klogd, syslogd, modprobe, poweroff, reboot, rmmmod' command utility. As a result of configuring the busybox as per application requirements, above command utilities in '/bin' & in '/sbin' directories, have been created.

In '/lib' directory which has been created, I have just kept kernel modules. I have not included any library files. The reason for not including any library files, is that I have configured busybox with statically linked to C library. So, no command utility required the library files to be present on target file system. So, I omitted the library files & this step saves approx.**2.4 MB** of space. The content of '/etc/' directory is same above (i.e initab, rcS & fstab file).

Then, I have cross-compiled the kernel with unnecessary H/W support removed & built kernel modules with only necessary support included.

After doing all above steps, the optimized version of root file system occupies **881KB** of space. Now, file system has to be converted in appropriate file system type, so that it can be directly ported onto flash. I have converted file system in JFFS2 format. For that, MTD utilities has been downloaded & configured according to host computer configuration. These are the steps for configuration of MTD utilities:

- Down load MTD utilities in 'build-tools' directory.

```
$ ${PRJROOT}/build-tools/mtd-util/configure --with-kernel=/usr/src/linux
```

```
$ make
```

```
$ make prefix=${PREFIX} install
```

Here, 'with-kernel' option is used to specify the location of kernel source of host computer. Now, after installing MTD utilities, it can be used to convert the root file system in JFFS2 format using following command.

```
$ mkfs.jffs2 -r ${PRJROOT}/rootfs/ -o ${PRJROOT}/images/rootfs.img
```

This way, the file system with JFFS2 format has been created which occupies **484 KB**.

5.7 Making Kernel Module for Keypad Driver:

The kernel module for keypad driver has been created. For that, first task is to register the device driver to the kernel. . This is synonymous with assigning it a major number during the module's initialization. The *major number* of a device normally identifies the class of devices that the driver can manage. I did this by using the `register_chrdev` function, defined by `linux/fs.h`.

```
'int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);'
```

Where unsigned 'int major' is the major number to request, 'const char *name' is the name of the device as it'll appear in `/proc/devices` and 'struct file_operations *fops' is a pointer to the `file_operations` table for keypad driver. A negative return value means the registration failed. Note that I have not passed the minor number to `register_chrdev`. That's because the kernel doesn't care about the minor number; only driver uses it.

Now the question is, how do we get a major number without hijacking one that's already in use? The easiest way would be to look through `Documentation/devices.txt` and pick an unused one. That's a bad way of doing things because we'll never be sure if the number you picked will be assigned later. The answer is if we pass a major number of 0 to 'register_chrdev', the return value will be the dynamically allocated major number. The downside is that we can't make a device file in advance, since we don't know what the major number will be. There are a couple of ways to do this. First, the driver itself can print the newly assigned number and we can make the device file by hand. Second, the newly registered device will have an entry in `/proc/devices`, and we can either make the device file by hand or write a shell script to read the file in and make the device file. The third method is we can have our driver make the device file using the `mknod` system call after a successful registration and remove during the call to `cleanup_module`.

Registration is done typically in the `init_module` method.

I have written the `INSTALL` script which does the job of extracting the major number and `mknod`'ing the device file. It looks like this:

```
#!/bin/sh
module="keypad"
device="keypad"
mode="777"
major=`awk "\$2==\"$module\" {print \$1}" /proc/devices`
mknod /dev/${device} c $major 0
# give appropriate group/permissions
chmod $mode /dev/${device}
```

Un-registration of a device is done with the following function:

```
unregister_chrdev (Major_Number, DEVICE_NAME);
```

This typically has a place in the `cleanup_module` method.

Then, I built the kernel-2.4.26 for which keypad module has to be made. I set the ‘-isystem’ path to point to the new kernel include directory in keypad’s Makefile. (Note: Kernel source directory `linux24` contains the source and headers for the on-board kernel. So, the path has been set to point to it since the module has to be build for the on-board kernel). Then, I set appropriate compiler option in keypad’s Makefile & compiled it. The `module_init()` & `module_exit()` function has been created in keypad file. Then, finally module has been inserted in the kernel using ‘`insmod`’.

5.8 Miscellaneous Tasks:

5.8.1 Ported boot loader in FLASH & in EEPROM:

I have ported boot loader in FLASH & in EEPROM. The boot loader had to be ported in EEPROM & FLASH to recover from one problem. But, this procedure will also help when actual porting of boot loader in the target prototype board (which will be developed by other members of VoIP-project team) is done.

We faced the problem with development board. The problem was that the 'boot.bin' file in EEPROM got corrupted. Development board was not able to boot. To recover from this problem, the following steps have been carried out:

- Apply jumper 1(JP1) on board for serial booting.
- Download the TS version of ECOS source from site :
`ftp://ftp.embeddedarm.com/ecos-src.tar.gz`
- Untar it to a dir, cd there and applied the eeprom patch:

```
$ tar xzvf ecos-src.tar.gz
$ mv ecos ecos_ts7250_eeprom
$ cd ecos_ts7250_eeprom
$ patch -p1 < redboot_ts7250_eeprom.diff
```

- Set ecos repository environment variable path:

```
ECOS_REPOSITORY='/path to ecos_ts7250_eeprom'/packages
```

- Edited packages/hal/arm/arm9/ts7250/current/cdl/

hal_arm_arm9_ts7250.cdl & set cross- compiler tool-chain path in it.

- Create a build directory and build the new redboot.bin file. & Download the ecosconfig utility in host linux-PC.

```
$ cd build
$ ecosconfig new ts7250 redboot
$ ecosconfig import '/path to ecos_ts7250_eeprom'/packages/
hal/arm/arm9/ts7250/current/misc/redboot_ROMRAM_ts7250.ecm
$ ecosconfig tree
$ make
```

- Resulting redboot image was: build/install/bin/redboot.bin
- Copy this file to the serial_blaster directory.
- In serial blaster directory type:

```
$ make
```

By doing this, 'boot.bin' file has been created. Then by running serial blaster program it has been copied in internal buffer for EEPROM & 'redboot.bin' file has been copied into flash memory.

Now, serial booting is possible. But, still development board didn't boot itself once power goes down. For that, 'boot.bin' file has to be written in EEPROM To write the 'boot.bin' file in EEPROM I have modified the **boot.S** file & followed the same steps given above to generate boot.bin file.

To write the boot.bin file in EEPROM, type following on RedBoot prompt:

```
$RedBoot> load -r -b 0x00218000 -h 192.168.0.25 boot.bin
$RedBoot> eeprom_write -b 0x00218000 -o 0 -l 2048
```

5.8.2 Finding differences between stock kernel-2.4.26 & kernel-2.4.26ts9:

Some of VoIP-project team members are developing H/W prototype board which is similar to TS-7250 board. This H/W prototype board contains the H/W components required by VoIP phone. This H/W prototype board is somewhat different from TS-7250 board like- H/W prototype board doesn't contain CPLD chip which is proprietary of Technologic System. Also, H/W prototype board uses processor's clock where as TS-7250 board uses the clock that is resided on board. Also, some components like FLASH, SDRAM etc. would be different from TS-7250 board components.

Now, board manufacturer has modified the original stock kernel source 2.4.26 & they made changes according to CPLD programming & onboard clock. Now, for the H/W prototype board, the kernel-2.4.26ts9 wouldn't work because of some changes would be made in H/W prototype board. So, it is required to study the both kernel source (the original stock kernel 2.4.26 & the kernel which is provided by TS-7250 board manufacturer) & observe the differences between the two kernels. According to those differences, it is required to make changes in kernel source so that it will work on H/W prototype board.

5.8.3 Provided Single Process tracing support.

I was curious about knowing the interaction of kernel with VoIP application. Also, I wanted to know which system calls are required by VoIP application. So, I provided Single Process Tracing support in TS-7250 board.

Single Process Tracing allows monitoring the interaction between application & kernel. I used 'strace' utility. Strace uses ptrace () system call to intercept all system calls made by an application. To provide the support of 'strace', I did the following steps.

- Download 'strace' source.
- \$ cd /home/ragin/strace4.4
- \$ CC=/usr/local/opt/croscotool/arm-linux/gcc-3.3.4-glibc-3.2/bin/arm-linux-gcc ./configure --host=arm-linux
- \$ make LDLIBS="-static" -W1 --start-group -lc -lnss_files -lnss_dns \-lresolve -W1 --end-group"
- Copy 'strace' utility in Target root file system.
\$ cp strace /home/ragin/rootfs/usr/bin/



CHAPTER 6

RESULTS

- Kernel Optimization:

When I have compiled the kernel-2.4.26 without configuring & optimizing it as per VoIP application, the default size was **1.8 MB**. After the application of various optimization techniques, the kernel is reduced to 448 KB.

- After configuring the kernel according to H/W & S/W specification of VoIP phone and removing unnecessary drivers, the total kernel image size became **1624 KB**.
- After applying appropriate compiler optimization techniques & stripping techniques, the total kernel image size became **1284 KB**.
- After removing most of 'printk' statements from kernel source code, the kernel size became **1154 KB**.
- After compressing the kernel image, the total kernel image size became **448 KB**.

- File System Optimization:

I have reduced the size of root file system from 7.6 MB to 881 KB.

- The file system which is provided by Technologic System occupies **7.6 MB** of memory space.
- The file system which I have created without applying any optimization Techniques, occupies **6.7 MB** of memory space.
- After using uClibc instead of Glibc, the root file system size reduced to **4.3 MB**.
- After optimizing the file system as per the VoIP phone configuration, the root file system size became **881 KB**.
- After converting file system in JFFS2 format, the file system size reduced to **484 KB**.

- Kernel Performance Improvement:

I have reduced the scheduler latency of kernel-2.4.26 for ARM architecture. But, I could not able to measure the scheduler latency of kernel for ARM architecture as there is no tool available for ARM architecture which measures the scheduler latency by doing kernel level instrumentation.

So, I have reduced the scheduler latency of kernel for x86 architecture in a similar way that I did for ARM architecture. After reducing the scheduler latency, I measured the scheduler latency for the kernel with low-latency support disabled and the kernel with low-latency support enabled. The results are given in following table.

<u>Kernel</u>	Low-latency support Disabled	Low-latency support Enabled
Maximum latency	439	40
Minimum latency	4.35	4.65
Average latency	1808.42	46.69
Total latency	15,753.33	1124.34

Note: All measurements in table are in mill seconds



CHAPTER 7

SUMMARY

7.1 Conclusion:

It's been a great experience for me while working in Project VoIP. During project, I got in-depth understanding of embedded system development process. Also, the tasks that I have performed would be very useful for VoIP phone development. The optimization of the kernel according to h/w & s/w configuration for VoIP phone is done as part of this thesis work. Also, task of improving kernel performance by providing Soft Real Time support was carried out during thesis work. The root file system has been created & optimized to occupy very less memory space. Both kernel & file system which have been optimized can be directly ported to the VoIP phone prototype board. The boot loader for the target VoIP phone is ready & can be directly ported to the EEPROM & flash memory. Apart from these tasks, the Project also involved troubleshooting the development board & solving technical problems related to the development board.

7.2 Scope of Future work:

The kernel can be further optimized as per the specific requirements of VoIP phone. Also, the kernel & file system which have been optimized can be tested & ported to the hardware prototype board which will work as a VoIP phone & which will be made by other team members soon. In future, the kernel can be made real-time by providing hard real-time support.

- **REFERENCES:**

- [1] Taking Charge of your VoIP Project; By John Walker; Cisco Press Edition
- [2] Building Embedded Linux Systems; By Karim Yaghmour; O'Reilly
- [3] Linux Kernel Programming; By U Kunitz, C Schroter; Pearson Education
- [4] Embedded Linux- Hardware, Software and Interfacing; By
Craig Hollabaugh, Second Edition; Peorson Education.
- [5] Linux Kernel Development; By Robert Love; SAMS, Developer Lib. Series
- [6] VoIP-101: Understanding VoIP networks; By Stefan Brunner, Akhlaq A. Ali
- [7] An application oriented Linux Kernel Customization for Embedded System;
By Chi- Tai Lee, Jim- Min Lin
- [8] System-wide Compaction and Specialization of the Linux Kernel;
By Bruno De Bus, Ludo Van Put
- [9] Linux scheduler latency, by Clark Williams, Red Hat, Inc, march-2002
- [10] Linux File System Hierarchy: version 0.65, by Bin Nguyen
- [11] Linux Kernel Module Programming guide, by Peter Jay Salzman, Ori Pomerats
- [12] http://www.packetizer.com/voip/papers/voip/voip_introduction.html
- [13] <http://www.embeddedarm.com>
- [14] <http://www.arm.linux.org.uk>.
- [15] <http://www.aleph1.co.uk/armlinux/book/book1.html>
- [16] <http://www.linuxdoc.org/HOWTO/NFS-HOWTO/index.html>
- [17] <http://www.kernelnewbies.org>
- [18] <http://www.linuxdevices.com/files/articles>
- [19] [ftp:// ftp.embeddedarm.com/ecos-src.tar.gz](ftp://ftp.embeddedarm.com/ecos-src.tar.gz)
- [20] <http://www.uclibc.org>
- [21] http://www.tldp.org/LDP/intro-linux/html/sect_04_02.html
- [22] <http://www.arm.cirrus.com>