

EMBEDDED SYSTEMS INITIALIZATION PROFILING AND CUSTOMIZATION

Submitted in partial fulfilment of the requirements for the award of
**Bachelor of Engineering Degree in
Electronics and Communication Engineering**

BY

SHRUTHI PRABAKARAN (38130210)



**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

SCHOOL OF ELECTRONICS

SATHYABAMA

**INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)**

Accredited with Grade "A" by NAAC

JEPPIAAR NAGAR, RAJIV GANDHI SALAI, CHENNAI – 600119, TAMILNADU

MARCH-2022



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited with “A” grade by NAAC | 12B Status by UGC |
Approved by AICTE

Jeppiaar Nagar, Rajiv Gandhi Salai, Chennai – 600 119
www.sathyabama.ac.in

DEPARTMENT OF ELECTRONICS AND COMMUNICATIONS ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of **SHRUTHI PRABAKARAN (38130210)** who carried out the project entitled “**EMBEDDED SYSTEMS INITIALIZATION PROFILING AND CUSTOMIZATION**” under the guidance of **DR. A. ARANGANATHAN. M. Tech., Ph/D.,** during the period NOV 2021 to APRIL 2022.

INTERNAL GUIDE

Dr.A.ARANGANATHAN. M.Tech., Ph.D.,

EXTERNAL GUIDE

**Mr.SILAMBARASAN
RAJENDIRAN,
TECHNICAL ARCHITECT
Mr. PRADEEP KUMAR K,
TECHNICAL LEAD
ERS-CET-TECH
EXCELLENCE-EPL-HCL
Technologies**

Head of the Department

Dr.T. RAVI, ME., Ph.D.

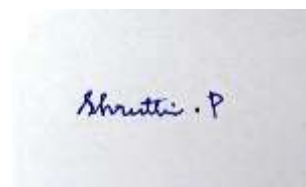
Submitted for Viva voce Examination held on

Internal Examiner

External Examiner

DECLARATION

I **SHRUTHI PRABAKARAN (38130210)** hereby declare that the Project Report entitled **“EMBEDDED SYSTEMS INITIALIZATION PROFILING AND CUSTOMIZATION”** done by me under the guidance of **Dr. A. ARANGANATHAN. M.Tech., Ph.D.,** (Internal Guide) and **Mr. SILAMBARASAN RAJENDIRAN, Technical Architect** and **Mr. PRADEEP KUMAR, TECHNICAL LEAD, ERS-CET-TECH EXCELLENCE-EPL-HCL TECHNOLOGIES** (External Guide) at **HCL TECHNOLOGIES Ltd, No. 602/3 Elcot Economic Zone, Sholinganallur Village, Medavakkam High Road, Sholinganallur, Chennai - 600119** is submitted in partial fulfilment of the requirements for the award of Bachelor of Engineering degree in Electronics and Communications Engineering.



DATE: 16/03/2022

PLACE: CHENNAI

SHRUTHI PRABAKARAN (38130210)

SIGNATURE OF THE CANDIDATE

ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to the Board of Management of **SATHYABAMA** for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to **Dr.N. NANDITHA M.E., Ph.D.**, Dean, School of Electronics and Communication and **Dr.T. RAVI, M.E., Ph.D.** Heads of the Department of Electronics and Communication Engineering for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide of **Dr. A. ARANGANATHAN. M.Tech., Ph.D.**, for his valuable guidance, suggestions, and constant encouragement that paved the way for the successful completion of my project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the **Department of Electronics and Communication Engineering** who were helpful in many ways for the completion of the project.

ABSTRACT

In Embedded Systems, there are stringent requirements about certain aspects such as fast and accurate performance, power consumption, memory accesses etc. Even with high-end processors and GPUs, faster booting is not achieved, especially when open operating systems like Linux are used. Since Linux operating systems are large in size, the boot time tends to be too slow. The primary goal of this project is to calculate the time taken from booting to the execution of application code, when an ARM architecture system is switched ON. The time that is obtained as the result is used to optimize a particular part of a code which provides performance benefits to applications. This process of obtaining data (time, in this project) to improve the performance of an Embedded System is known as Profiling.

High-end devices like security cameras, consumer electronics, and medical devices are expected to boot faster for a given requirement. A normal embedded Linux system takes around 30 to 60s to boot. Hence, the user needs to wait at least more than 30 seconds to utilize the system which can lead to loss of video capture and significant clues. It is particularly important for devices in the medical field to have a faster booting time since they could be used as life supporting systems. These requirements in the market result in a problem statement of Linux Boot time.

TABLE OF CONTENTS

CHAPTER NO	TITLE	PAGE NO
	ABSTRACT	v
	LIST OF TABLES	ix
	LIST OF FIGURES	x
	LIST OF ABBREVIATIONS	xi
1	INTRODUCTION	1
1.1	EMBEDDED SYSTEM	1
1.1.1	Working of an Embedded System	1
1.1.2	Basic Structure of an Embedded System	2
1.2	MICROCONTROLLER	5
1.2.1	Interrupt Controller	6
1.3	MEMORY ORGANIZATION IN MICROCONTROLLER	7
1.3.1	Flash Memory	7
1.3.2	Types of Flash Memory	7
1.4	DIFFERENCE BETWEEN EEPROM AND FLASH	7
1.5	SPI FLASH	8
1.6	FLASH MEMORY LAYOUT	8
1.7	MICROPROCESSOR	10
1.8	TYPES OF MEMORY IN EMBEDDED SYSTEMS	11
1.8.1	Volatile Memory	11
1.8.2	Non-Volatile Memory	11
2.	LITERATURE SURVEY	12
3.	EXISTING SYSTEM	14
3.1	EXISTING SYSTEM	14

CHAPTER NO	TITLE	PAGE NO
3.2	DRAWBACKS	14
4.	PROPOSED SYSTEM	15
4.1	iMX51 BOARD	15
4.2	EMBEDDED LINUX	16
4.3	SYSTEM WIDE BOOT TIME MEASUREMENT	17
4.3.1	Grab Serial	17
4.3.2	Uptime	18
4.4	KERNEL INVOCATION TIME MEASUREMENT	18
4.4.1	KFT (Kernel Function Time)	18
4.4.2	PRINTK	19
4.5	USER SPACE TIME MEASUREMENT	20
4.5.1	Boot Chart	20
4.5.2	Strace	21
4.6	BOOTLOADER	21
4.6.1	Application of Bootloader	21
4.7	COMMUNICATION PROTOCOLS	21
4.7.1	<i>SPI Protocol</i>	21
4.7.2	UART Protocol	23
4.8	INTERRUPT VECTOR TABLE	25
4.8.1	Interrupts and Exceptions in Arm MCU	25
4.8.2	Interrupt Vector Table Store in Microcontroller Memory	26
4.8.3	Role of Interrupt Vector Table in Interrupt Processing	27
4.9	EXCEPTIONS	27
4.9.1	Interrupts	28
4.9.2	ABORT	28

CHAPTER NO	TITLE	PAGE NO
4.10	EXCEPTION HANDLERS	28
	4.10.1 IRQ (Interrupt Request)	29
	4.10.2 FIQ (Fast Interrupt)	29
4.11	PIPELINING	30
4.12	BRANCHING	30
4.13	BRANCH PREDICTOR	31
4.14	CACHE MEMORY	31
	4.14.1 Types of Cache Memory	31
4.15	BOOTING PROCESS	32
4.16	RESET HANDLER	33
4.17	BOOT SEQUENCE	34
4.18	START UP FILE	35
4.19	START UP CODE	35
4.20	BOOT ROM	39
4.21	BOOT ROM CODE ARCHITECTURE	40
4.22	BOOTING PROCEDURE	41
4.23	BOOTING FLOW DIAGRAM	42
5.	RESULTS AND DISCUSSIONS	43
5.1	RESULT	43
6.	CONCLUSION AND FUTURE SCOPE	45
6.1	CONCLUSION	45
6.2	FUTURE SCOPE	45
	REFERENCES	46

LIST OF TABLES

TABLE NO	TITLE	PAGE NO
4.1	Interrupt Vector Table	32

LIST OF FIGURES

FIGURE NO	TITLE	PAGE NO
1.1	Embedded Systems Block Diagram	2
1.2	Architecture of Microcontroller	6
1.3	Memory Segments in Flash Memory	8
1.4	Architecture of Microprocessor	10
4.1	iMX51 Board	15
4.2	Linux System Architecture	16
4.3	SPI Protocol Block Diagram	23
4.4	UART Block Diagram	24
4.5	Pipelining Diagram	30
4.6	Reset Handler	33
4.7	Boot ROM	39
4.8	Boot ROM code architecture	40
4.9	Boot ROM code booting process	41
4.10	Booting Flow Diagram	42
5.1	Boot time of Normal Embedded System	43
5.2	Boot time before optimization	44
5.3	Boot time with Video after optimization	44

LIST OF ABBREVIATIONS

ADCs	- Analog-to-Digital Converters
ALU	- Arithmetic Logical Unit
ASIC	- Application-Specific Integrated Circuits
CPU	- Central Processing Unit
DSP	- Digital Signal Processors
FIQ	- Fast Interrupt
FPGA	- Field-Programmable Gate Arrays
IRQ	- Interrupt Request
ISR	- Interrupt Service Routine
KFT	- Kernel Function Time
MISO	- Master In Slave Out
MOSI	- Master Out Slave In
POR	- Power on Reset
RAM	- Random Access Memory
SCK	- Serial Clock
SPI	- Serial Peripheral Interface
SS	- Slave Select
UART	- Universal Asynchronous Receiver Transmitt

CHAPTER 1

INTRODUCTION

1.1 EMBEDDED SYSTEM

An embedded system is a microcontroller or microprocessor-based system which consists of hardware and software. Embedded systems work on their own or a part of a large system to perform specific tasks. An integrated circuit is present at the core of the system, which performs computing for real-time operations.

From a single microcontroller to a suite of processors with associated peripherals and networks, and from no user interface to complicated graphical user interfaces, there are many different levels of complexity. An embedded system's complexities vary greatly depending on the tasks for which it is intended to perform.

Digital watches and microwaves are examples of embedded system applications, as are hybrid automobiles and avionics. Embedded systems account for up to 98 percent of all microprocessors produced.

1.1.1 *Working of an Embedded System*

Microcontrollers or digital signal processors (DSP), application-specific integrated circuits (ASIC), field-programmable gate arrays (FPGA), GPU technology, and gate arrays are responsible for the working of embedded systems. These processing systems include components that handle electric and/or mechanical interfaces.

The software that is responsible for providing instructions to embedded systems, known as firmware, is stored in read-only memory or flash memory chips and it runs on restricted computer hardware resources. Input and output devices are connected to embedded systems through input/output peripherals.

1.1.2 Basic Structure of an Embedded System

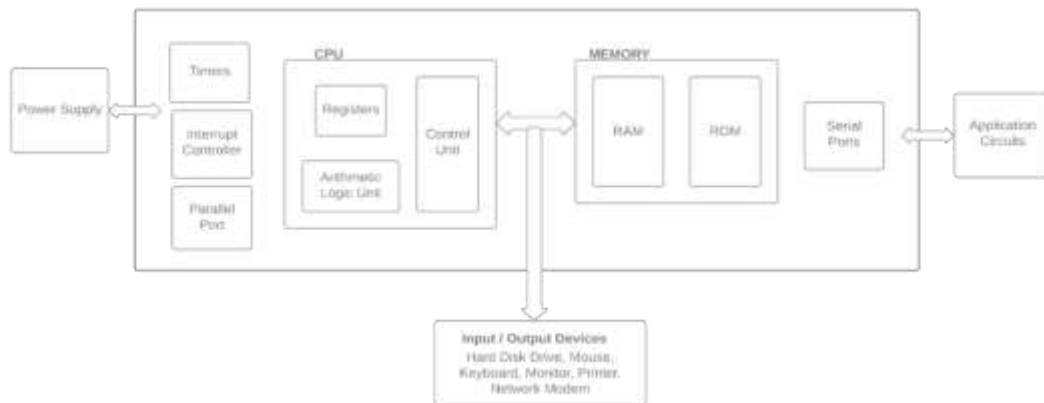


Fig. 1.1. Embedded Systems Block Diagram

CPU

A microprocessor is also known as a CPU (Central Processing Unit) is usually present on a single integrated circuit. A microprocessor is usually present on a single integrated circuit. The terms microprocessor and CPU are often interchangeable. Intel Pentium series, Freescale/IBM PowerPC, MIPS, ARM, and the Sun SPARC are the most commonly used microprocessors nowadays. A processor performs the following operations: write data to memory or write data to an I/O device, read data from memory or read data from an I/O device, read instructions from memory, and internal manipulation of data within the processor.

The processor has an internal storage system called registers. The number of registers is limited in the processor and they are used to store data/operands that are immediately required by the CPU.

ALU

The Arithmetic Logic Unit (ALU) is responsible for performing arithmetic and logical operations of data in the processor. The data flow between the registers and ALU are controlled by the instructions that are read and executed by the processor. The arithmetic operations performed by the ALU via the ALU's control inputs are also controlled by the instructions.

The processor provides instructions to the ALU to perform certain operations (usually addition, subtraction, NOT, AND, OR, XOR, shift left/right, or rotate left/right) on one or more values. These values which are known as operands are fetched from a register, or one value from a register and the other from a memory location. The result obtained from the operation is stored in a register or a memory location.

Control unit

Control Unit is a component present inside the central processing unit (CPU) which controls the other peripherals of a system. It is responsible for sending out instructions in the form of signals to the memory, ALU and input/output devices. It fetches these instructions from main memory and transfers them to the CPU registers and based on the contents in the register, the control unit sends out control signals that monitor the execution of instructions.

Registers

Registers are a form of temporary memory location which are used to store and transfer data and instructions that are needed immediately by the CPU. The registers used by the CPU are often known as processor registers. These registers are used for manipulating data and for holding memory addresses. The register that holds the memory location is used for calculating the address of the next instruction after the present instruction's execution is completed.

RAM

Random Access Memory (RAM) is a type of memory that is used to store temporary data. The "working memory" of a computer system is known as RAM. It's where the processor can write data for temporary storage with ease. When the system loses power, the contents of RAM are normally lost. Before the system shuts down, any data in RAM must be written to some type of permanent storage.

ROM

ROM stands for Read-Only Memory. ROMs are non-volatile memory, requiring no power to retain their contents. They are generally slower than RAM, and considerably slower than fast static RAM. The primary purpose of ROM within a system is to hold the code (and sometimes data) that needs to be present at power-up.

Input / Output controllers

The input/output controller is a device that connects a computer or hardware device to an input or output device. The input/output controller is usually found on the motherboard of a computer.

In addition, an I/O controller can enable the computer to use additional input or output devices. It might also be an internal add-on component that replaces a motherboard's failing I/O controller.

I/O Devices

Other devices besides memory can be found in the processor's address space. The CPU uses these input/output devices (I/O devices, also known as peripherals) to connect with the outside world. Serial controllers, which connect with keyboards, mouse, modems, and other peripherals; parallel I/O devices, which operate an external subsystem; and disk-drive controllers, video and audio controllers, and network interfaces are just a few examples.

Timers

Timers are critical components in embedded systems; their responsibility is to maintain operations on time, whether it's managing the blinking rate of LEDs, controlling the sampling rate of ADCs, or simply delaying source code. When we need to count events, we link an external event source (for example, event: rising edge or falling edge) to the timer/counter module's input pin. The value of the counter increases when an external event happens. The number of external events is represented by the value of the counter.

When we intend to create time delays, we connect an external oscillator to the timer/counter module's input pin. The value of the counter increases and the content of the counter register indicates time that is passed at each tick (rising or falling edge) of the clock (external oscillator).

Power Supply

All embedded systems require a significant amount of power. Nothing can function without the use of power. Several power choices are available depending on the type of application. If the system does not need to be portable, for example, it can be powered straight from the wall through AC adaptors. They typically offer an output voltage of +3.3V to +12V DC and a current of a few amperes.

An embedded system is made up of many separate components that may run on a range of various power sources. However, some components, such as Analog-to-Digital Converters (ADCs), require a steady voltage source to deliver a precise output since they require a reference voltage to convert the analogue signal to digital count. This is accomplished through the employment of a voltage regulator. Its duty is to convert a variety of DC input voltages to a fixed output voltage. A voltage regulator also reduces power supply noise and provides some protection for the embedded system against harm caused by variable input voltages. In the end, integrating a voltage regulator in your design is always a wise decision.

1.2 MICROCONTROLLER

A microcontroller is a computer that is contained within a single integrated circuit and is allocated to perform a single task and for running a single application. Memory, programmable input/output peripherals, and CPU are all included. Microcontrollers are primarily employed in embedded applications and are widely used in automated electronic devices such as mobile phones, cameras, microwave ovens, and washing machines.

External Interrupt

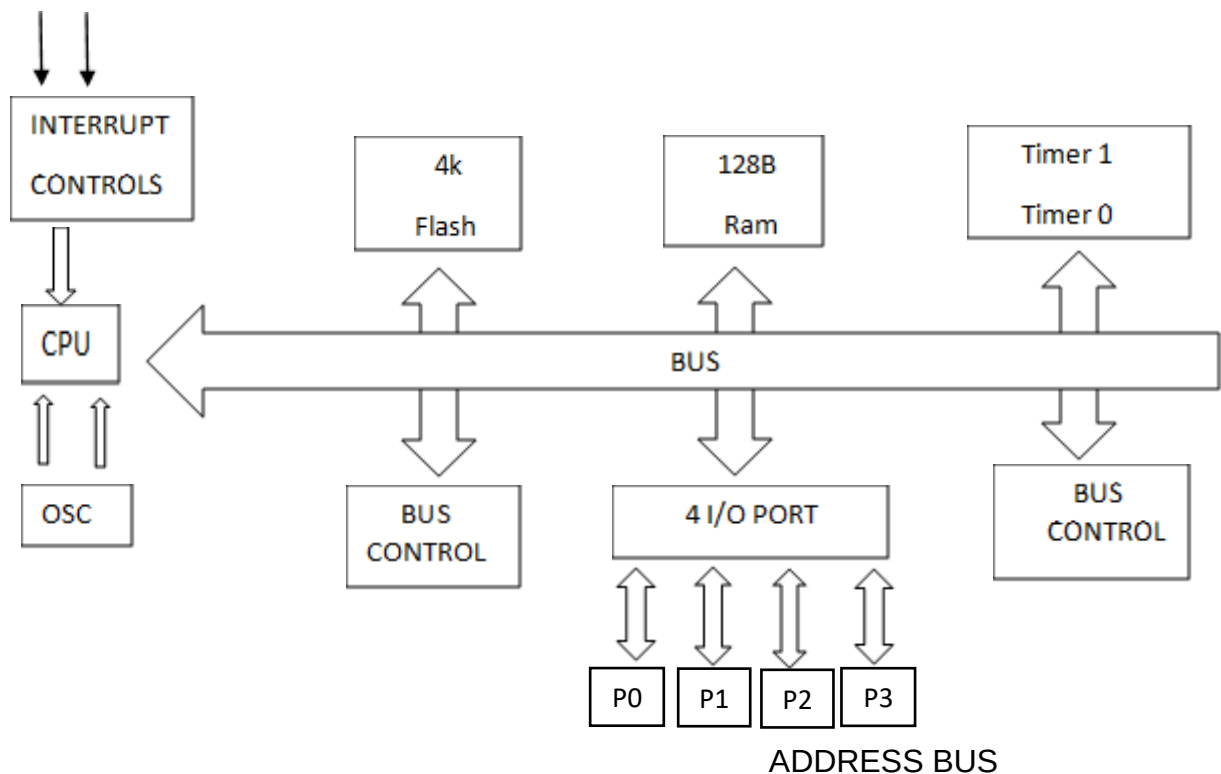


Fig 1.2: Architecture of Microcontroller

1.2.1 Interrupt Controller

An interrupt controller is a device that manages interrupts from a variety of peripheral devices on a single chip. These devices are usually connected via a bus subsystem controlled by the CPU or, in certain circumstances, by on-chip Bus Masters.

The CPU gets "interrupt requests" from the interrupt controller. A changing input on a change notification pin or the elapsed time on one of the timers can both create an interrupt request (IRQ). If the CPU accepts the request, it will pause whatever it is doing and move to an interrupt service routine (ISR), which is a software function. When the ISR is completed, program control is returned to the process that was executing before that process was halted.

1.3 MEMORY ORGANIZATION IN MICROCONTROLLER

1.3.1 *Flash Memory*

In embedded systems, flash memory is commonly utilised to store data and code. It's a non-volatile storage media, which means it can save data even if the power source is turned off. Flash memory can be wiped and reprogrammed electrically, and it erases data in blocks and rewrites data at the byte level. Flash memory is widely utilised in systems that update data regularly, such as USB flash drives and SD cards.

1.3.2 *Types of Flash Memory*

NOR Flash

NOR Flash is designed for random access, which allows it to read data in any order without having to follow a set of storage units. Each of NOR Flash's memory cells are linked in parallel, according to its design. This lets individual memory cells be accessed by the system.

NAND Flash

NAND Flash, on the other hand, is designed for high-density data storage and does not provide random access. NAND Flash cells are linked in a string to the bit line, generally eight memory transistors at a time.

In conclusion, NAND-based Flash memory is best suited for large-capacity data storage, whereas NOR-based Flash memory is best suited for code storage and execution in limited capacities.

1.4 DIFFERENCE BETWEEN EEPROM AND FLASH

EEPROM, or electrically erasable programmable read-only memory, is a kind of flash memory. The reading, writing, and erasing operations of stored data are among the various distinctions between EEPROM and Flash memory. For

example, EEPROM can read, write, and erase data at the byte level, but Flash memory can only delete data at the block level.

1.5 SPI FLASH

SPI (Serial Peripheral Interface) NAND Flash provides a low-cost, high-density non-volatile memory storage option for embedded systems. SPI flash is a flash module that is connected to a system through SPI. By using low-cost embedded controller standards, SPI connections may be relatively fast (133MHz). Some also offer 2-bit and 4-bit data buses, allowing for even faster transfer rates than a pure serial interface.

1.6 FLASH MEMORY LAYOUT

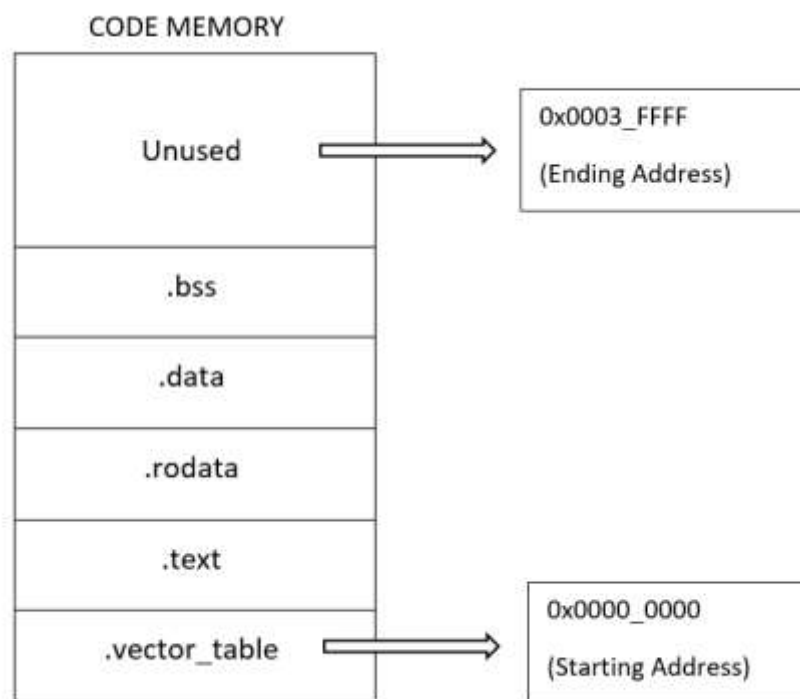


Fig 1.3: Memory Segments in Flash Memory

The interrupt vector table, code segment, initialised data segment, uninitialized data segment, constant data segment, and other application code segments are all stored in the flash memory of microcontrollers. The embedded system construction process uses a linker script file that specifies where each of

these segments will be stored in flash memory. The linker script determines where these various code segments will be stored in actual flash memory. The target embedded device determines the amount of flash memory and the starting/ending.

Interrupt Vector Table Segment

The addresses that correspond to different interrupt and exception procedures, such as the reset handler, are stored in this sub-segment of code memory.

Text Segment

The major application code, such as program instructions, interrupt service routines, user-defined functions, microcontroller start-up code, and standard library functions, are included in the text segment. It contains the actual microcontroller application code that we have developed, in simple terms. This segment is restricted to read-only access. This signifies that the CPU can only read instructions from this section and cannot write.

Rodata Segment

The .rodata portion of our program holds the constant types of data that we created in our program, such as strings, constant variables, and so on. .const is another name for this section. Once specified during program compilation, the constant data cannot be altered during program execution. This will throw an exception.

Data Segment

The .data section consists of global, global static, and local static variables of program data.

BSS Segment

Uninitialized global, global static, and local static variable data are contained in .bss segment.

Thus, the compiled sections of program are mapped to the flash memory of the microcontroller.

RAM/Data Memory Layout

The .data section, .bss section, stack, and heap are all parts of the data memory. To begin with, when a microcontroller powers up, the data and. bss sections (described in the previous section) copy to RAM memory. The heap and stack memory areas are also assigned to the data memory's stack and heap sections. All of these hardware initialization tasks are handled by a start-up file.

1.7 MICROPROCESSOR

A microprocessor is a microcomputer's controlling unit, manufactured on a tiny chip that can conduct ALU (Arithmetic Logical Unit) operations and communicate with other devices linked to it. An ALU, register array, and control unit make up a microprocessor. The ALU performs arithmetic and logical operations on data from memory or an input device. The register array is made up of registers with letters such as B, C, D, E, H, L, and accumulator. The control unit is in charge of the computer's data and instruction flow.

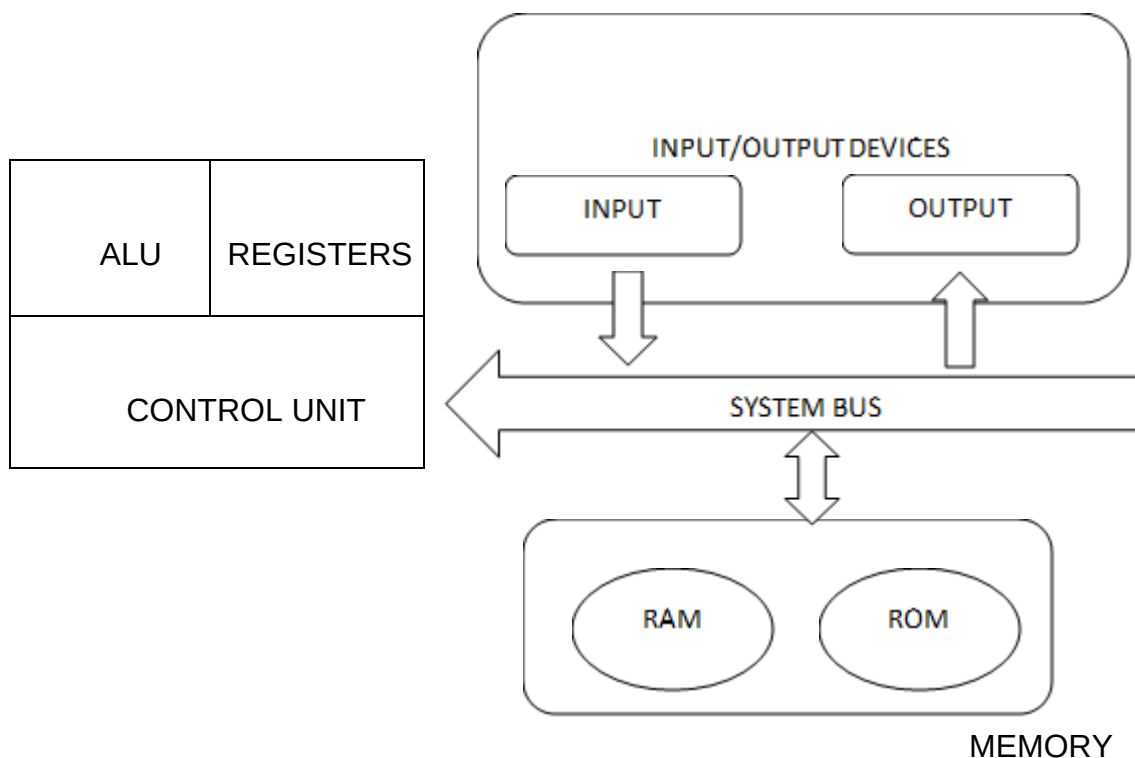


Fig 1.4: Architecture of Microprocessor

1.8 TYPES OF MEMORY IN EMBEDDED SYSTEMS

Memory in Embedded systems can be broadly categorized into two types, Volatiles and Non-Volatile.

1.8.1 *Volatile Memory*

It's the memory hardware that gets and saves data quickly. Temporary memory is another name for it. The data in the volatile memory is retained for as long as the system is operational, but once the system is shut off, the data is immediately erased. Volatile memory includes things like RAM (Random Access Memory) and cache memory. Data retrieval and storage is quick and cost-effective here.

1.8.2 *Non-Volatile Memory*

It is a sort of memory in which data or information is retained even after the power is turned off. The most prevalent type of non-volatile memory is ROM (Read Only Memory). When compared to volatile memory, it is less cost-effective and slower to acquire and store data, but it can store a larger amount of data. Non-volatile memory is used to store information that must be kept for a long period of time. The storage capacity of a system is greatly influenced by non-volatile memory.

CHAPTER 2

LITERATURE SURVEY

C. Profentzas, M. Günes, Y. Nikolakopoulos, O. Landsiedel and M. Almgren [1] With the rising use of the Internet of Things (IoT), it is more important than ever to emphasise overall system security. The Internet connectivity of the IoT exposes isolated internal device structures to cyber-attacks through the Internet, bringing up new attack pathways and vulnerabilities. During the boot process, verifying the operating system and application software is a common technique to ensure system integrity. Residential IoT devices, on the other hand, should be ready to use immediately after being turned on, whereas safety-critical IoT devices have limited boot-up times. As a result, a device's boot time may have an influence on its utility. This study looks at the performance trade-offs of secure boot for medium-scale embedded devices like the Beaglebone and Raspberry Pi. For the software-based option, secure boot adds just 4% to the entire startup time. Additionally, the additional cryptographic hardware storage extends startup time by 36%.

F. -J. Streit [2] In today's embedded systems, comprehensive security requirements across the product life cycle have become an increasingly important access-to-market criterion. On the other hand, these security standards rely on mandatory immunity assumptions such the integrity and validity of an initial system configuration loaded from Non-Volatile Memory (NVM). This is especially true for FPGA-based Programmable System-on-Chip (PSoC) architectures, where object codes and configuration data often surpass the capacity of a safe boot ROM. An attacker might try to control the system by changing the content of the NVM device in this situation.

S. Weng and Y. Lin [3] The purpose of this project is to reduce system startup time and improve I/O multiplexing performance for the image player in embedded Linux systems. The startup time is reduced by bypassing certain unnecessary routines. When it comes to I/O multiplexing, the select, poll, and epoll

system calls are contrasted. The findings imply that using epoll I/O multiplexing in the creation of an image player can result in better performance.

T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente and M. Bertogna [4] One of the main expected restrictions of today's multi-core embedded systems is competition for shared memory resources. Memory hierarchy congestion can be reduced by using partitioning and software-driven memory resource allocation algorithms. Unfortunately, many of the techniques utilised thus far may have unintended repercussions when implemented on latest-generation, high-performance embedded systems. This research presents a framework of software-based memory access determinism restoration solutions for high-performance embedded devices.

Y. Kang, J. Chen and B. Li [5] Bootloaders must be relocated and updated as part of embedded system development. Due to the bootloader's sophisticated architecture and considerable hardware requirement, the cost spent on development becomes unnecessary. The macro definition file is utilised to achieve online updating, and the modular design of the code and custom instructions are used in this study to reduce the transplant effort. To boost reliability, create a backup for the flash memory area at the same time.

Using this bootloader on the general-purpose platform PowerPC speeds up the migration process by more than 60% as compared to U-Boot. The bootloader can be started from the backup area if it has a success rate of 92.5 percent or above.

CHAPTER 3

EXISTING SYSTEM

3.1 EXISTING SYSTEM

Devices that operate fast are the need of the hour in today's world. Fast booting of a device is not only integral for applications that need immediate use but also for improving the user experience and usability. Linux kernel based operating systems are used in most of the consumer electronic devices and the users of these devices expect them to be used quickly after being turned on. Hence, it is important to measure the time taken for booting and this data can be used to make changes to optimise the start up time of devices. After measuring the initial time, the unnecessary applications and services which unknowingly take time to boot and take up the system resources are removed.

It is important that while the boot time is optimized, the functionalities that are already present in the system should not be compromised, instead it should aid the system to improve the booting process. There are various different ways to optimize the booting process: disabling unnecessary features that the bootloader may have, using an uncompressed kernel, eliminating unnecessary kernel options etc.

3.2 DRAWBACKS

In order to reduce boot time, only an individual component cannot be held accountable, the entire system should be designed according to the required boot time of a system. The time it takes for an application to be usable after the booting of a system should be taken into consideration while reducing boot time.

CHAPTER 4

PROPOSED SYSTEM

4.1 iMX51 BOARD

The primary goal of this project is to measure the boot time of a Linux operating system present on an iMX51 board which can be seen below Fig: 4.1



Fig: 4.1 iMX51 Board

The iMX51 board uses an ARM Cortex™-A8 core, which runs at speeds up to 800 MHz, is included in the i.MX51 processors. DDR2 and mobile DDR DRAM clock frequencies of up to 200 MHz are supported by iMX51. At the heart of the i.MX51 processors is a degree of power management that allows the device's broad array of multimedia functions and peripherals to consume as little power as possible in both active and low-power modes. Smart Speed Technology enables the designer to create a feature-rich device that consumes significantly less power than is customary in the industry.

The i.MX51 processors include hardware-enabled security capabilities that enable safe e-commerce, digital rights management (DRM), information encryption, secure boot, and secure software downloads, as the need for sophisticated security for mobile devices grows.

4.2 EMBEDDED LINUX

Embedded Linux is a Linux kernel intended specifically for embedded devices. Android, the popular smartphone operating system, is an example of embedded Linux that has been tailored for smartphones. A kernel, in simple terms, is the part of an operating system that handles the computer's functions and hardware, particularly the memory and CPU. To put it another way, the kernel is the software-hardware interaction.

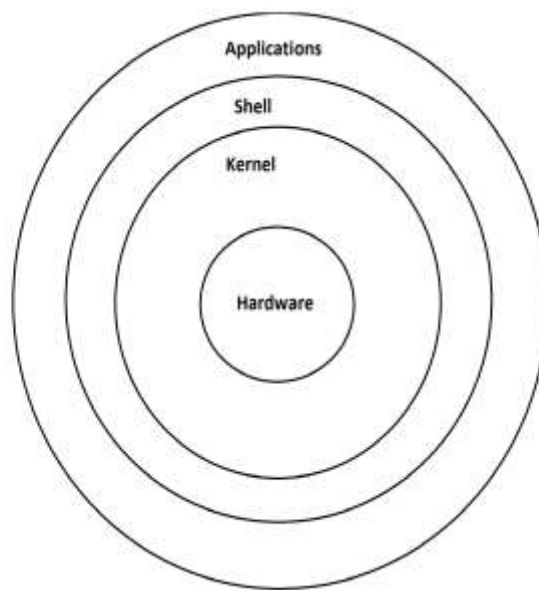


Fig 4.2 Linux System Architecture

Linux has evolved to support a wide range of CPU architectures, including x86, ARM, and PowerPC, among others. So, regardless of the hardware that one uses, there's a strong possibility that a Linux kernel already exists for it. Linux includes almost all of the programming languages and facilities required for embedded system development. So, the user is not limited to any certain software when using Linux.

Since Linux is an open-source kernel, the user may benefit from the efforts of thousands of people all over the globe. They can get their own apps up and running much more easily and fast by altering or directly implementing existing packages.

4.3 SYSTEM WIDE BOOT TIME MEASUREMENT

The boot time of the complete booting process is measured in the system wide time measurement step. The two techniques that come under system wide boot time are discussed below.

- Grab Serial
- Uptime

4.3.1 *Grab Serial*

Grab serial is a software that gathers and displays serial port data. This tool's main duty is to gather output messages sent on a serial terminal from a Linux target system and save them on the host machine. Grab serial appends a time stamp to each line received from the chosen serial port and prints it on the serial console output. It is possible to report the time of low-level events throughout the booting process by doing so.

It is possible to provide parameters that are utilised on the Linux node, as well as port settings, using the capture serial command line options. If no option is specified, the default setup on serial port /dev/ttyS0 at 115200 baud with "8, None, and 1" will be used (8N1). The software continues continuously, reading data from the serial port and displaying it in the chosen form. It gives you the time of each line you get.

The relative time duration between lines may be traced using the following script 'show delta' from the kernel 'scripts' directory. By following the steps outlined below, data may be saved to a file and presented.

```
Grabserial -e 20 >/tmp/bootlog.txt; linux_src/scripts/show_delta /tmp/bootlog.txt
```

4.3.2 Uptime

The most straightforward method is to use Uptime. Uptime displays information such as the current time, the length of time the system has been running, the total number of users who have signed in, and system load averages for the last 1, 5 and 15 minutes on a single line. It displays the system's total time stamp.

It works by adding the command `echo -n "uptime:"; cat /proc/uptime` to `/sbin/init` or `rc.local`.

4.4 KERNEL INVOCATION TIME MEASUREMENT

During kernel invocation, the kernel measurement tools calculate the amount of time spent in various kernel functions and subroutines. These tools assist us in doing a comprehensive study of the most time-consuming kernel functions and subroutines that are necessary to reduce startup time. The following are the most important tools for calculating kernel invocation time:

- KFT
- Printk – Times

4.4.1 KFT (Kernel Function Time)

KFT is a kernel function tracing tool that shows how long kernel functions and subroutines take. It assists us in doing our analysis in order to identify the operations that take the most time. It makes advantage of the gcc compiler's "-finstrument-functions" option to add measurement to each callout and exit. Dynamic trace configuration and static run mode configuration are the two modes of operation for KFT.

KFT can start and stop the trace automatically, handle interrupts as part of the trace, and filter the data using various screening criteria. On completion of the trace, KFT trace data may be generated from `/proc/kft-data`. In unfiltered mode, it may capture a stream of data and assist us in debugging and optimising the code.

The overhead caused by KFT is its worst flaw. KFT uses the gcc compiler's "finstrumentfunctions" functionality to monitor kernel time for function 37's entrance and exit. When trace is turned off, the overhead is incurred during kernel execution.

That is the primary reason why it is disabled by default in the board. This overhead proved to be expensive in time-sensitive procedures.

4.4.2 *PRINTK*

Printk is a simple software that has been built to detect the region of the Linux kernel that may be improved to reduce boot time. To obtain extra timing information, the patch is applied to the regular Printk procedure. This functionality is already included in the standard Linux version 2.6.11 distribution.

We can use Printk to observe how long Boot procedures take. When turning on Printk messages, this approach is used to discover lengthy delays in kernel starting. As soon as the Printk messages start, the time data in seconds appears. This software displays the difference in timing between two routines or the time length related to a given message.

Printk messages can be initiated by booting Linux kernel 2.6.11 and entering the command line option 'time' or Printk='1'. Printk data is gathered and stored to dmesg >/tmp/bootup printks using the dmesg option on the command line. Show_ delta may extract information from Printk output and display the time interval between Printk messages using a utility application.

The simplicity and robustness of Printk approaches are its key features. To achieve boot time measurement, no other application, code, or subroutine is required. On the other hand, there were just a few disadvantages to using Printk. Due to the restricted size of the kernel buffer, the Printk message option 'dmesg' does not display all kernel messages. Each time the printk message was sent, it added a few bytes to the buffer, ultimately filling it up.

4.5 USER SPACE TIME MEASUREMENT

The overall time spent from the start of the init procedure to the state when the system is operational is depicted by user space time measurement. To assess the impacts of boot reduction strategies, user space time measurement tools can effectively measure the boot time. Different techniques such as Application XIP and optimization of RC scripts may be used to lower user space boot time. Boot chart and Strace are two user space measuring tools covered here.

4.5.1 *Boot Chart*

A boot chart is a tool for examining the boot process while the user space is being initialised. It's a tool for analysing performance and visualising the GNU Linux boot process. During the boot process, the boot chart captures statistics on resource use and process information. Later, it is artistically represented as a PNG chart using SVG encoding.

Shell scripts are used in the kernel's init phase by the boot chart. This script collects process information, CPU start, disc usage, and data from the /proc file system in the background.

The disadvantages of using a boot chart are that it adds a minor amount of time to the system startup time since an additional script is required to collect information during the boot process. It works well in systems with plenty of RAM and processing power, but it fails miserably in systems with limited resources, such as a small embedded device.

In terms of scalability, the boot chart performs poorly. Because of the enormous number of system calls, file input/output overhead is significant, and data collecting is the most time-consuming stage. While logging data to output files, many operations must be completed.

4.5.2 *Strace*

Strace is a debugging, diagnostic, and teaching tool. It's used to gather timing data at the process level. It influences how long an application takes to start

up. It can count the number of system calls and the amount of time spent on each one. It can track which actions, such as numerous file opens, are causing major problems contains the sample result acquired from Strace.

Strace's key benefit is that it can operate in parallel with processes, trace all system calls to the kernel, and even offer extensive details in the event of memory failure or a mistake while accessing a file.

4.6 BOOTLOADER

Bootloader is a program that initializes the system to basic-level and loads kernel. Basic initialization is a part of kernel loading i.e., it initializes the parts of system which are needed to load the kernel. In short, bootloader performs all activities to create execution environment for kernel. In controller, bootloaders job is to load and execute main application, receive and program application code to flash memory i.e., firmware upgrade.

4.6.1 *Application of Bootloader*

After power-on or reset only single CPU core and static memory can be used. Other resources such as DRAM, NAND flash, mmc controllers needs to be configured and initialized separately and hence, these interfaces are not accessible. Static memory available on SOC is very limited and hence, it is not possible to load kernel code into it.

4.7 COMMUNICATION PROTOCOLS

4.7.1 SPI Protocol

Serial Peripheral Interface, which is commonly known as S-P-I or “spy” is one of the most popular interface specifications used in embedded systems. Since its introduction in the late 1980’s by Motorola, the SPI protocol has been widely used for short distance communication in embedded systems.

Through the SPI protocol, devices communicate with each other using master-slave architecture. Although multiple slave devices can be supported by SPI,

the number of master devices is limited to one. SPI is known as four-wire serial bus because it consists of four signals:

- Master out slave in (MOSI)
- Master in slave out (MISO)
- Serial clock (SCK),
- Slave select (SS)

In full duplex mode, data rates over 1Mbps can be achieved - this is one of the main advantages of the SPI bus. Compared to I2C, SPI also supports using simple hardware interfacing and provides a higher throughput.

However, the SPI protocol also has some drawbacks – the lack of error-checking mechanism and slave acknowledgment feature are some of the major disadvantages.

Applications of SPI Protocol

- Apply for communication with temperature, pressure sensors, and video game controllers.
- Interface with LCDs and SD cards.
- Communicate with FLASH and EEPROM memory.
- Read data from a real time clock.

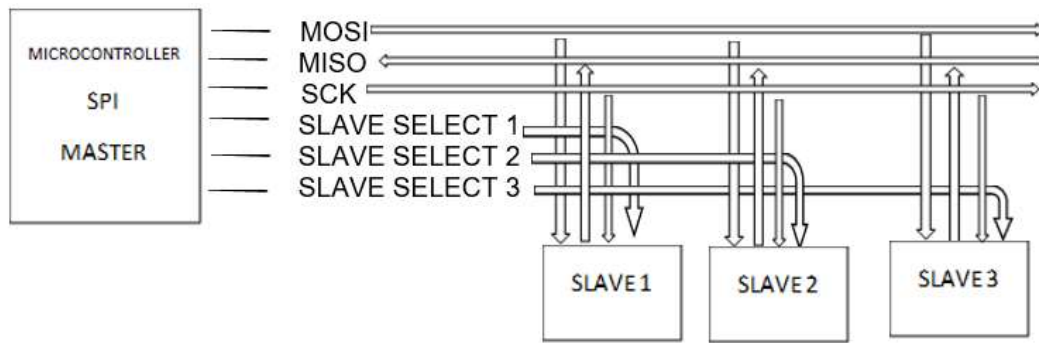


Fig 4.3: SPI Protocol Block Diagram

4.7.2 UART Protocol

The UART, or universal asynchronous receiver-transmitter, is one of the most used device-to-device communication protocols. When properly configured, the UART can work with many different types of serial protocols that involve transmitting and receiving serial data. In serial communication, data is transferred bit by bit using a single line or wire. Depending on the application and system requirements, serial communications need less circuitry and wires, which reduces the cost of implementation.

Despite being a widely used method of hardware communication protocol, it is not fully optimized all the time. Proper implementation of frame protocol is commonly disregarded when using the UART module inside the microcontroller.

Protocols like SPI (serial peripheral interface) and USB (Universal Serial Bus) are used for fast communication. When the high-speed data transfer is not required UART is used. It is a cheap communication device with a single transmitter/receiver. It requires a single wire for transmitting the data and another wire for receiving.

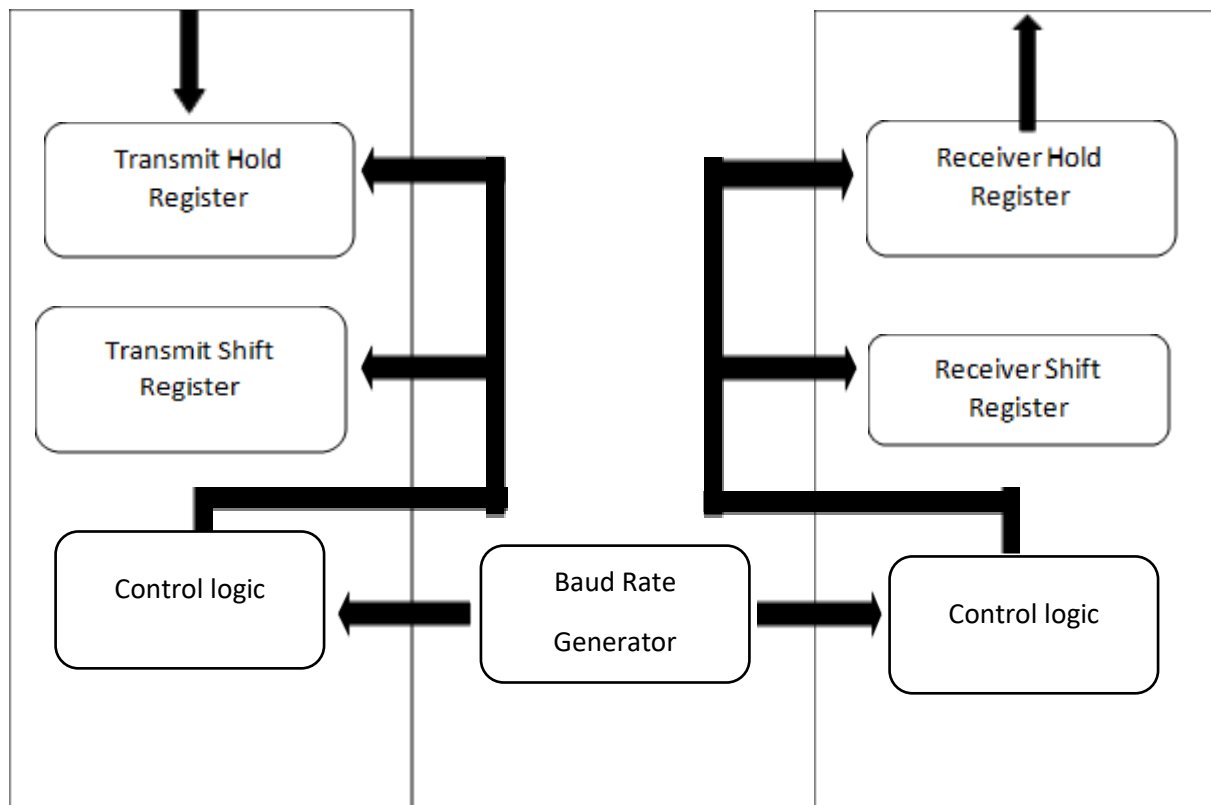


Fig 4.4: UART Block Diagram

The Universal Asynchronous Receiver Transmitter (UART) block diagram has two main components. They are the receiver and transmitter. These two components are coupled with a baud rate generator. This is used mainly for speed generation when the receiver and transmitter section have to receive or transmit data.

The receiver section consists of shift register, control logic and a receive hold register. Likewise, transmitter section also has a shift register, control logic and a transmit hold register. The transmitter hold register contains the data to be transmitted. The shift registers in the two components move the data bits left or right till the data transmission or receive operation is completed.

A write or read logic is used to indicate when the read and write operation must be done. The baud rate generator is used to generate speeds ranging from

110 bps to 230400 bps. The micro-controllers typically use a baud rate of 9600 bps to 115200 bps.

Advantages of UART

- Full duplex transmission of data can be achieved with only two wires.
- Clock or timing signals are not required.
- Less power consumption.
- It has parity bit for is ensured for error checking.

Disadvantages of UART

- The data frame is limited by size.
- Data transfer speed is low in comparison with parallel communication.
- Appropriate rules must be agreed upon by both parties in communication.
- It doesn't support several slave or master systems.

4.8 INTERRUPT VECTOR TABLE

The interrupt vector table is a table of memory addresses of interrupt/exception handler routines. In other words, it defines where the code of a particular interrupt/exception routine is located in microcontroller memory.

4.8.1 Interrupts and Exceptions in Arm MCU

The vector table contains the address of the ISR routines of all interrupts and exceptions that the microcontroller supports.

If you check the datasheet of ARM Cortex M4 microcontroller, it has 15 system exceptions and 138 peripheral interrupts. Peripheral interrupts are also defined as simple exceptions in literature. The main difference between system

exceptions and interrupts is that system exceptions are generated by the CPU and peripheral interrupts are generated by different peripheral modules available in ARM cortex based microcontrollers.

4.8.2 *Interrupt Vector Table Store in Microcontroller Memory*

There are types of memory in microcontrollers such as code memory or data memory. The interrupt vector table stores in flash/code memory of ARM Cortex M4 microcontrollers.

The interrupt vector table stores at the starting addresses of code memory (starting from 0x0000_0000). Because some microcontroller has 154 total exceptions (including system and simple exceptions), Therefore, the vector table contains 154 entries. The first two entries of the vector table are the initial value of the stack pointer and the address of the reset handler function. Because whenever a microcontroller resets, it performs hardware initialization steps. Each memory address contains the address of exception handlers. For example, the address 0x0000_003C contains the address location of the systick timer interrupt handler.

One more point to note here is the exception number. The exception number is used by ARM Cortex M CPU to index into the location of the interrupt vector table.

The vector table and interrupt service routines/exception handlers are defined inside the start-up file of a microcontroller. The start-up file and a linker script file define the way to store the interrupt vector table at the starting 256 locations of the microcontroller's code memory

Because Some microcontrollers support 154 exceptions and interrupts. Hence, IVT stores at first 154 words of ROM or code memory. One extra location is used to store the starting address of the main stack pointer.

4.8.3 *Role of Interrupt Vector Table in Interrupt Processing*

ARM Cortex-M CPU has two modes of operation such as thread mode and exception. In normal execution, CPU runs in thread mode. But when an interrupt occurs the CPU transfers from thread mode to exception mode. In exception mode, nested interrupt vector controller manages all interrupt and exception requests.

When an interrupt x occurs, the interrupt request will be sent to NVIC. If NVIC accepts the exception/interrupt request x , the next step of NVIC to find the starting address of the interrupt service routine or exception handler. The starting address of the respective ISR or exception handler is stored inside the interrupt vector table. Then NVIC uses exception number x to calculate the address of the exception by looking up the interrupt vector table and use the content of that memory address (which is an address of the respective exception handler) to execute the exception handler.

Program counter will be loaded with the address of the exception handler and the CPU starts to execute the exception routine.

INTERRUPT VECTOR TABLE usually stores at the starting addresses of flash or code memory.

4.9 EXCEPTIONS

Exceptions occur when the normal flow of a program must temporarily halt, for example, to service an interrupt from a peripheral. Before attempting to handle an exception, the processor preserves the critical parts of the current processor state so that the original program can resume when the handler routine has finished.

The two primary exceptions are:

- Interrupts
- Aborts

4.9.1 *Interrupts*

The interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention. It alerts the processor to a high-priority process requiring interruption of the current working process. In I/O devices one of the bus control lines is dedicated for this purpose and is called the Interrupt Service Routine (ISR).

4.9.2 *ABORT*

Of the exceptions that an ARM-based processor can raise, two abort exceptions signal that the current memory access cannot be successfully completed.

- **Data Abort:** The first one, data-abort exception, has the second-highest priority, just after reset. This exception conveys that the data access transaction was unsuccessful.
- **Pre fetch Abort:** The second is the pre fetch-abort exception, which has the second-lowest priority, just one notch above the software interrupts. This abort is invoked when the processor is unable to fetch an instruction from memory.

4.10 EXCEPTION HANDLERS

An exception handler is code that stipulates what a program will do when an anomalous event disrupts the normal flow of that program's instructions.

The two types of exception handlers are:

- IRQ (Interrupt Request)
- FIQ (Fast Interrupt)

4.10.1 *IRQ (Interrupt Request)*

An interrupt request (IRQ) is a signal sent from a device to a processor indicating that in order to process a request, attention is required. A hardware IRQ is induced by a hardware peripheral or device request, whereas a software IRQ is induced by a software instruction. Both result in processor status savings, and revert to serving the IRQ using an interrupt handler routine.

The following are characteristics of IRQ:

- IRQ is disabled when FIQ is working.
- IRQ has no dedicated registers.
- It consists of many instructions.
- IRQ is slower than FIQ.

4.10.2 *FIQ (Fast Interrupt)*

Fast Interrupt Requests (FIQs) are a specialized type of Interrupt Request, a standard technique used in computer CPUs to deal with events which need to be processed as they occur such as receiving data from a network card, or keyboard or mouse actions. FIQs are specific to the ARM architecture, which supports two types of interrupts; FIQs for fast, low-latency interrupt handling, and standard interrupt requests (IRQ).

An FIQ takes priority over an IRQ in an ARM system. Also, only one FIQ source at a time is supported. This helps reduce interrupt latency as the interrupt service routine can be executed directly without determining the source of the interrupt.

The following are characteristics of FIQ:

- In ARM processor, registers R0 to R14 will be used for FIQ. These registers are called bank registers.

- The execution of FIQ will be faster since these registers are closer to the CPU.
- It consists of less instructions.

4.11 PIPELINING

Pipelining is the technique of fetching an instruction whilst the prior one is being decoded and the one before that is being executed.

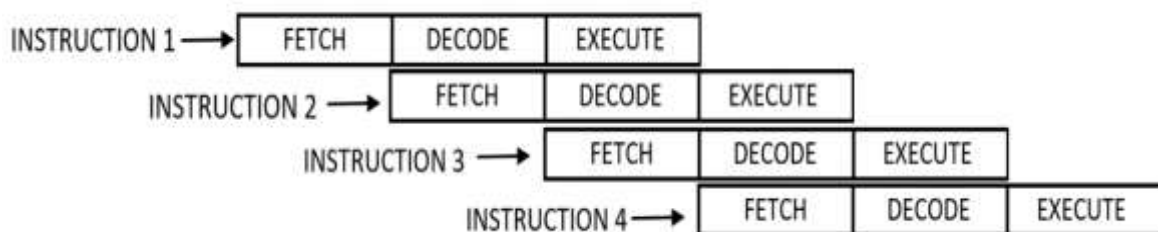


Fig 4.5: Pipelining Diagram

When Pipeline 1 is doing a 'Fetch', Pipeline 2 is doing a 'Decode' on the previous instruction and Pipeline 3 is executing the instruction before that. The three pipelines in the diagram are running at maximum efficiency because they are always full. This is an example of parallel processing.

4.12 BRANCHING

A branch in a computer program is an instruction that tells a computer to begin executing different instructions rather than simply executing the instructions in order. This disrupts the flow of code execution.

Mechanically, a branch instruction can change the program counter of a CPU. The program counter stores the memory address of the next instruction to be executed. Therefore, a branch can cause the CPU to begin fetching its instructions from a different sequence of memory cells.

4.13 BRANCH PREDICTOR

In computer architecture, a branch predictor is a digital circuit that tries to guess which way a branch will go before it goes to a sub function. It is an important component of modern CPU architectures, such as the x86.

4.14 CACHE MEMORY

Cache memory is a high-speed memory, which is small in size but faster than the main memory (RAM). The CPU can access it more quickly than the primary memory. So, it is used to synchronize with high-speed CPU and to improve its performance.

Cache memory can only be accessed by CPU. It can be a reserved part of the main memory or a storage device outside the CPU. It holds the data and programs which are frequently used by the CPU. So, it makes sure that the data is instantly available for CPU whenever the CPU needs this data. In other words, if the CPU finds the required data or instructions in the cache memory, it doesn't need to access the primary memory (RAM). Thus, by acting as a buffer between RAM and CPU, it speeds up the system performance.

4.14.1 *Types of Cache Memory*

The types of cache memory are:

- L1
- L2
- L3
- **L1:** L1 cache or primary cache, is extremely fast but relatively small, and is usually embedded in the processor chip as CPU cache.
- **L2:** L2 cache, or secondary cache, is often more capacious than L1. L2 cache may be embedded on the CPU, or it can be on a separate chip

and have a high-speed alternative system bus connecting the cache and CPU. That way it doesn't get slowed by traffic on the main system bus.

L3: It is a specialized memory developed to improve the performance of L1 and L2. L1 or L2 can be significantly faster than L3, though L3 is usually double the speed of DRAM. With multicore processors, each core can have dedicated L1 and L2 cache, but they can share an L3 cache. If an L3 cache references an instruction, it is usually elevated to a higher level of cache.

4.15 BOOTING PROCESS

Table 4.1: Interrupt Vector Table

VECTOR	OFFSET	PRIORITY
Initial SP Value	0x0000	
Reset	0x0004	The highest
Non-Maskable Interrupt (NMI)	0x0008	-2
...	...	...
...	...	...

1. Power On Reset
2. Execute boot loader in boot ROM
3. Processor read BOOT pins to determine the booting mode. For example, if BOOT0 is 0, Main Flash memory is mapped into Alias space from address 0x00000000
4. Next processor fetches the first word from memory space. As the beginning of the memory space contains the vector table, and the first word in the vector table is the initial value for the Main Stack Pointer (MSP). Processor set up the MSP after that.

Processor continues read the next word in vector table, now is reset vector, which contains the Reset_Handler () function address. Reset_Handler () function address is moved into Program Counter (PC) and processor starts running Reset_Handler () Inside Reset_Handler (), System_Init () function will be run first, follow is our application (main ()).

4.16 RESET HANDLER

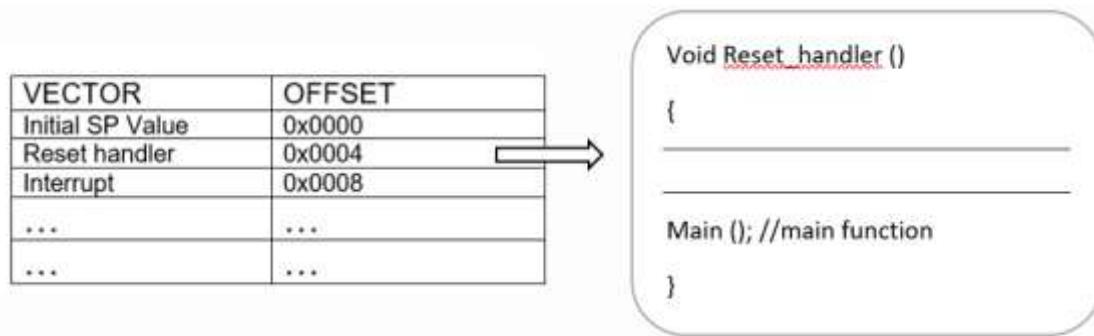


Fig 4.6: Reset Handler

Reset Handler Function

Reset vector or reset handler function performs hardware initialization such as:

1. Disable All Interrupts

Firstly, it disables all interrupts so that microcontroller cannot be able to execute any other interrupt service routine such as watchdog time, etc

When you upload your code to a microcontroller, the binary image of code gets stored in flash memory. Because flash is a non-volatile memory. Therefore, code stores there permanently even without power. During the microcontroller booting process, some data which is stored in flash memory is moved to the RAM.

This is because RAM works faster than flash memory and hence it helps to reduce the latency of data accesses from memory.

2. Initialize Data Segment

If you remember the layout of a running C program then you must remember that the initialized global, static global and static local variables store in the .data section of memory. Reset handler assign memory region to .data section and moves the content of the initialized data section from flash to RAM memory region of the microcontroller.

3. Initialize the .BSS Segment

In our embedded program, we use both initialized and uninitialized static and global variables. The uninitialized static, global static, and local static variables store in the .bss segment of memory. The .bss section also copies from flash to RAM and zero all uninitialized variables.

4. Initialize Stack Segment

The stack segment is a sub-segment of RAM memory. Reset handler reserves the stack space in data memory according to pre-compiled instructions given in the linker file. The stack memory region is used to hold temporary variables, input arguments, and return addresses of functions.

The last two steps are straight forward.

5. Enable Interrupts

6. Call the Main Function

4.17 BOOT SEQUENCE

The boot process begins at **Power on Reset (POR)** where the hardware reset logic forces the ARM core to begin execution starting from the on-chip Boot ROM. Processor/controller starts the execution of instructions from the address which is called reset vector. Jump instruction is there at the reset vector, which points to the start of Boot ROM.

4.18 START UP FILE

A startup file is a piece of code written in assembly or C language that executes before the main () function of our embedded application. It performs various initialization steps by setting up the hardware of the microcontroller so that the user application can run. Therefore, a startup file always runs before the main () code of our embedded application.

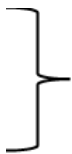
The startup file performs various initializations and contains code for interrupt vector routines. As we have seen in the last tutorial when a microcontroller resets, it executes a reset handler (interrupt service routine) and performs hardware initialization before executing the main code. The startup file consists of this reset handler function and hardware initialization code.

The boot code contains three files:

- START_UP FILE
- MAIN.C FILE
- LINKER SCRIPT(.ld) FILE

4.19 START UP CODE

```
#include<stdint.h>
#include<stdio.h>
-----
-----
.-----
```



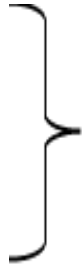
These are the library files used for start-up code

```
void reset_handler (); // declaration of reset handler function
```

```
extern void main (); // extern keyword is used since main function is in another file
```

void default_handler (); // Declaration of default handler function. This function handler is used in the case when interrupts and exceptions occur.

```
extern uint32_t _etext;
extern uint32_t _sdata;
extern uint32_t _edata;
extern uint32_t _sbss;
extern uint32_t _ebss;
```



These variables are defined in the linker script. They are addresses of segments that are created when the final program is compiled and used by the reset handler.

```
#define STACK_TOP 0x20017FFF
```

/* The flow of vector table is below:

ADDRESS 0x00:

0:	TOP_OF_STACK <- 32-bit address
1:	RESET_HANDLER <- 32-bit function address
2-16:	ARM_EXCEPTION_HANDLERS <- 32-bit function addresses
17-X:	INTERRUPT_HANDLERS <- 32-bit function addresses

*/

Vector Table

```
uint32_t _vectors[ ] __attribute__ ((section(".vectors"))) = {
(uint32_t) STACK_TOP,      //    stack top
(uint32_t)&reset_handler,  //    reset handler
(uint32_t)&default_handler, //    NMI_Handler
(uint32_t)&default_handler, //    HardFault_Handler
(uint32_t)&default_handler, //    MemManage_Handler
(uint32_t)&default_handler, //    BusFault_Handler
```


/*We first want to copy the values in .data, .bss to RAM, as that is where we told our code to look for them in the linker script*/

```
main ();  
}
```

```
void default_handler ()  
{  
    while (1)  
    {  
  
    }  
}
```

Main Function Code

```
void main()  
{  
    while (1)  
    {  
        //if code is present in flash it will directly execute  
        //if the code is not present in the flash then code is downloaded from the host  
        to the flash using communication protocol like UART, SPI, I2C etc., then It's taken  
        from flash to ram, after it will execute  
    }  
}
```

Linker script (.ld):

MEMORY:

```
{  
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x000FFFFF  
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00017FFF  
}
```

SECTION:

```
{  
    Text. Section à FLASH  
    Data. Section à RAM  
    Bss. Section à RAM  
}
```

4.20 BOOT ROM

Boot ROM program is provided by SOC manufacturer. Boot ROM program executes from SRAM as DRAM configurations are not known at the time of manufacturing hence, Boot ROM code cannot initialize the DRAM. As part of start-up Boot ROM code configures the PLL (phase locked loop) and sets clock and watchdog timer. So that it can escape from dead-loops while booting.

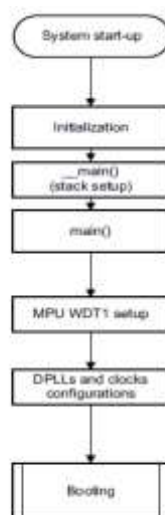


Fig 4.7: Boot ROM

4.21 BOOT ROM CODE ARCHITECTURE

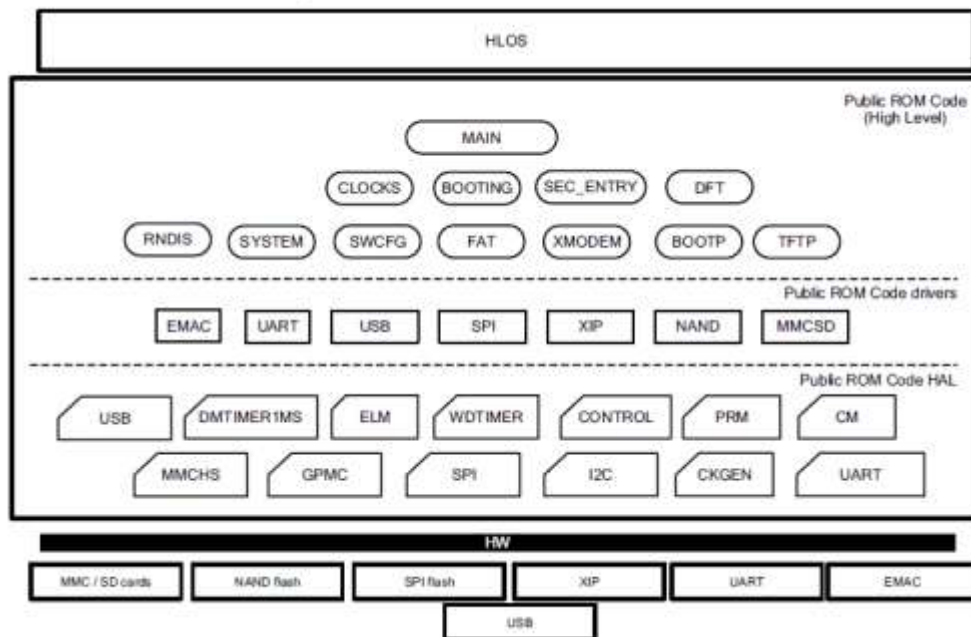


Fig 4.8: Boot ROM code architecture

As seen in the above figure, ROM code is capable of initializing drivers and protocols such as UART, SPI, USB, FAT etc so that external devices/protocols can be interfaced with SOC and Secondary Program Loader can be loaded into the SRAM.

ROM Code can perform following booting procedures:

- **Memory Booting:** Booting the device by starting code stored on permanent memories like flash-memory or memory cards. This process is usually performed after either device cold or warm reset. Memory booting is performed when the memory is of type NOR, NAND, MMC or SPI-EEPROM.
- **Peripheral Booting:** Booting the device by downloading the executable code over a communication interface like UART, USB or Ethernet. This process is intended for flashing a device. Peripheral

booting is performed when device is of type – USB, Ethernet or UART.

4.22 BOOTING PROCEDURE

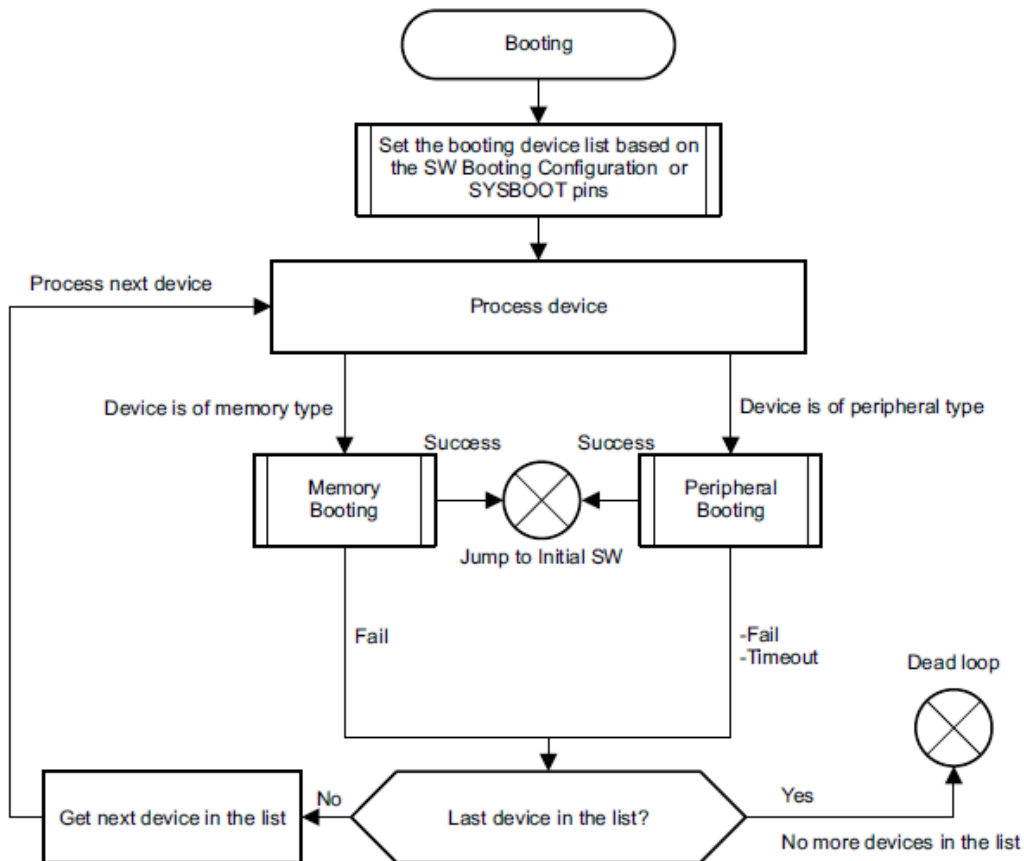


Fig 4.9: Boot ROM code booting process

ROM code generates the device list based on SYS BOOT configuration pins read value. Once, the booting procedure is finalized i.e., depending on memory or peripheral device, ROM code loads the SPL program into SRAM. If it fails to find the device or to validate and run the SPL program then system will get into the dead loop and watchdog timer gets triggered and resets the SOC.

4.23 BOOTING FLOW DIAGRAM

After the bootloader initializes the CPU, hardware, clocks, timers etc., the control is handed over to the main function. In the main function, the boot pins are chosen before entering the while loop (boot pins are configured according to boot mode table). If the boot pins are chosen for the application code to be sent to the RAM, then the application code is executed directly, which leads to fast execution. If the boot mode is selected for the application code to be taken from flash, then the code is moved from flash to RAM and the application code is then executed. This process is slower compared to the direct execution of code from RAM.

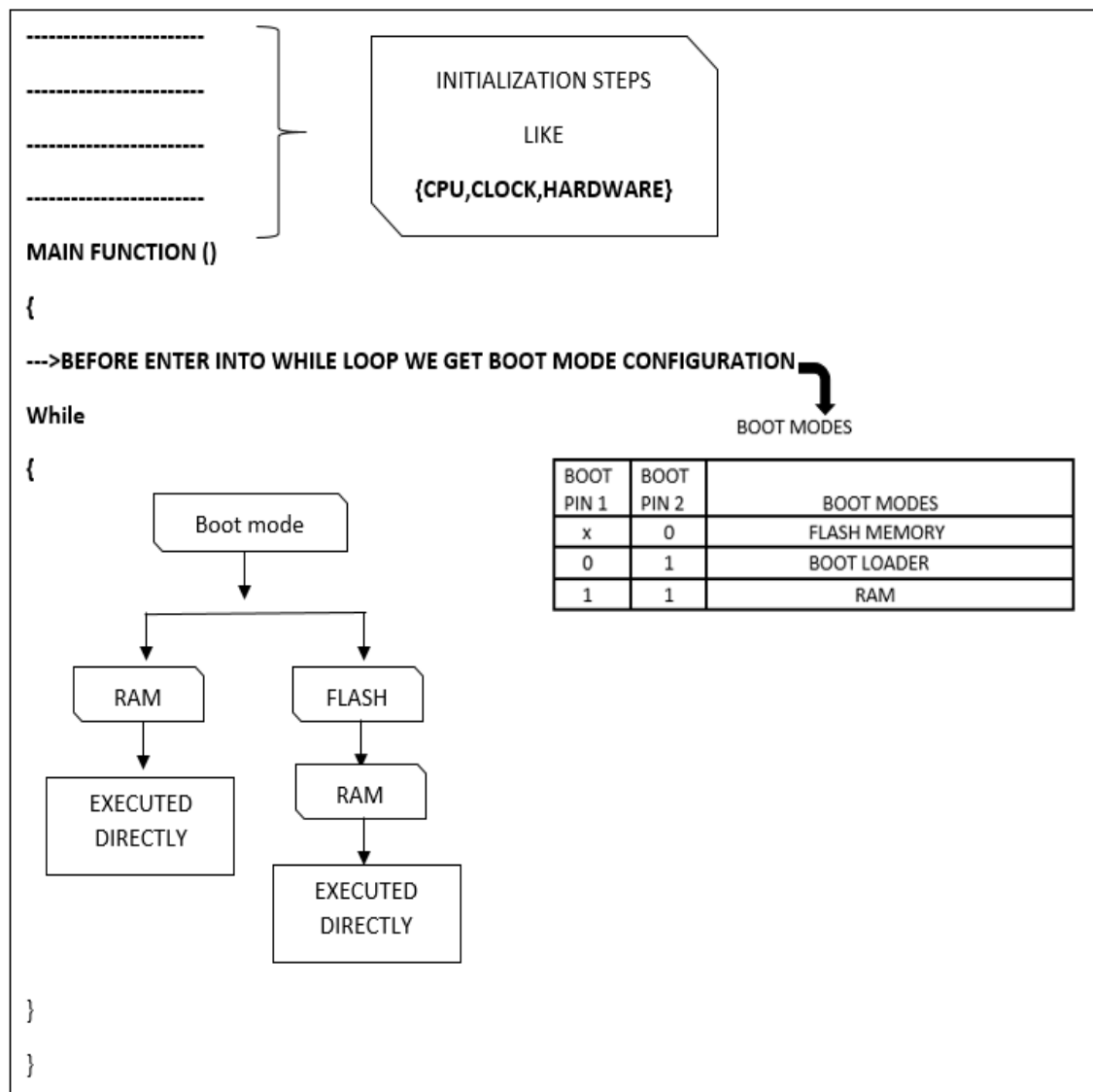


Fig 4.10: Booting Flow Diagram

CHAPTER 5

RESULTS AND DISCUSSIONS

5.1 RESULT

The performance of the microcontroller can be improved in terms of CPU throughput and faster execution by profiling. Here, profiling was done by calculating the time taken for execution of booting to main function of an iMX51 board when it is turned ON. The boot time was calculated using an oscilloscope which was tapped to the power line and serial port TX line.

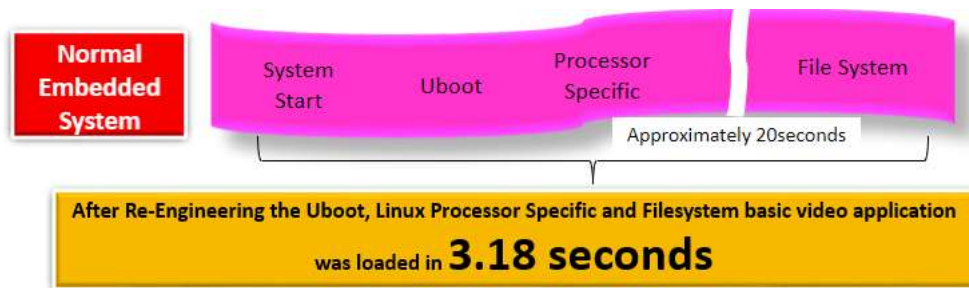


Fig 5.1 Boot time of Normal Embedded System

In fig 5.1 it can be seen that the boot that for an embedded system with a Linux operating system it takes 20 seconds for booting which is very slow for devices that need to be used as soon as they're powered on. Hence, the booting process has to be in order for an embedded device to be used immediately.

For a Linux based embedded system for which a video application (for a camera device) has to be displayed immediately after the boot time, the booting time is much less when compared to the start up time of a normal embedded system, this is done by removing unnecessary functions and device drivers through trial and error, removing unnecessary printout messages, optimizing system kernel by removing the USB and network support etc.

Boot time of 2.779 seconds with Shell		
1	Power On reset	0
2	Configure MMU, MMC driver and Fat read the Ulmage	0.255721
3	Starting Kernel	0.000093
4	Jump into kernel, at debug statement " Un compressing kernel"	0.007867
5	Uncompressing kernel	0.276104
6	Low level Board Init	0.144131
7	Linux Kernel Init	1.19597
8	Driver initialization	0.275976
9	File system	0.624011
10	Total Boot time	2.779873

Fig 5.2 Boot time before optimization

It can be inferred from fig 5.2 that after optimization, the boot time till the execution of shell has been significantly reduced (less than two seconds).

Boot time of 3.185 seconds with Video (Live video displayed on the LCD 800x480 at 30fps).		
1	Power On reset	0
2	Configure MMU, MMC driver and Fat read the Ulmage	0.255721
3	Starting Kernel	0.000093
4	Jump into kernel, at debug statement " Un compressing kernel"	0.007867
5	Uncompressing kernel	0.276104
6	Low level Board Init	0.144131
7	Linux Kernel Init	1.19597
8	Driver initialization	0.581976
9	File system	0.724011
10	Total Boot time	3.185873

Fig 5.3 Boot time with Video after optimization

There was a requirement for displaying a video application (for a camera device) after the booting process. Due to the changes made for reducing the boot time, the display of video application immediately after the booting process takes one a few milli seconds. Hence, the device can be used almost instantaneously after power on.

CHAPTER 6

CONCLUSION AND FUTURE SCOPE

6.1 CONCLUSION

In this project, the startup time for an iMX51 board using a Linux operating system was measured by connecting an oscilloscope to the power line and serial transmission line. This data can be used to reduce the boot up time and make a device ready for use almost immediately after said device is switched on. It was found that by removing various unnecessary functions that are not needed for a particular device, the booting time can be improved significantly.

There are various ways in which boot time optimization can be done effectively. However, its important to note that the boot loader that is resident on a particular microcontroller or microprocessor cannot be changed as it is hardwired by the manufacturer. The bootloader also contributes to system boot time but changed cannot be made in it.

Before optimizing the boot time of a device, it is crucial to understand the architecture and the booting sequence of that device in order to efficiently improve its performance.

6.2 FUTURE SCOPE

Currently there are various areas in which research is carried out for reduction of boot time without minimizing the functionality of a device. The scope of making the booting process faster is high in the future as fast starting technologies are needed everywhere around us in today's world. The study of boot time will be useful for Android based car navigation, entertainment systems, medical devices etc. for which fast boot is vital.

REFERENCES

- [1] <https://microcontrollerslab.com/microcontroller-booting-process-reset-sequence/>
- [2] <https://microcontrollerslab.com/microcontroller-memory-organization-types-memory-segments/>
- [3] <https://microcontrollerslab.com/microcontrollers-startup-file-arm-cortex-m4-mcu/>
- [4] <https://www.embedded.com/building-bare-metal-arm-systems-with-gnu-part-2/>
- [5] <https://www.javatpoint.com/architecture-of-linux#:~:text=The%20Linux%20operating%20system's%20architecture,actions%20of%20the%20Linux%20OS.>
- [6] Kang, Y., Chen, J. and Li, B. "Generic Bootloader Architecture Based on Automatic Update Mechanism," 2018 IEEE 3rd International Conference on Signal and Image Processing (ICSIP), 2018, pp. 586-590.
- [7] Kloda, T., Solieri, M., Mancuso, R., Capodieci, N., Valente, P. and Bertogna, M. "Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems," 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2019, pp. 1-14.
- [8] Profentzas, C., Günes, M., Nikolakopoulos, Y., Landsiedel, O. and Almgren, M. "Performance of Secure Boot in Embedded Systems," 2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS), 2019, pp. 198-204.
- [9] Streitm, F.J. *et al.*, "Secure Boot from Non-Volatile Memory for Programmable SoC Architectures," 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2020, pp. 102-110.
- [10] Weng, S. and Lin, Y., "Achieving Fast Boot Time and Efficient I/O Multiplexing for Image Grabber in Embedded Linux System," 2018 International Symposium on Computer, Consumer and Control (IS3C), 2018, pp. 158-160.