MARRI LAXMAN REDDY
INSTITUTE OF TECHNOLOGY AND MANAGEMENT
(AN AUTONOMOUS INSTITUTION)
(Approved by AICTE, New Delhi & Affiliated to JNTUH, Hyderabad)
Accredited by NBA and NAAC with 'A' Grade & Recognized Under Section2(f) & 12(B)of the UGC act,1956

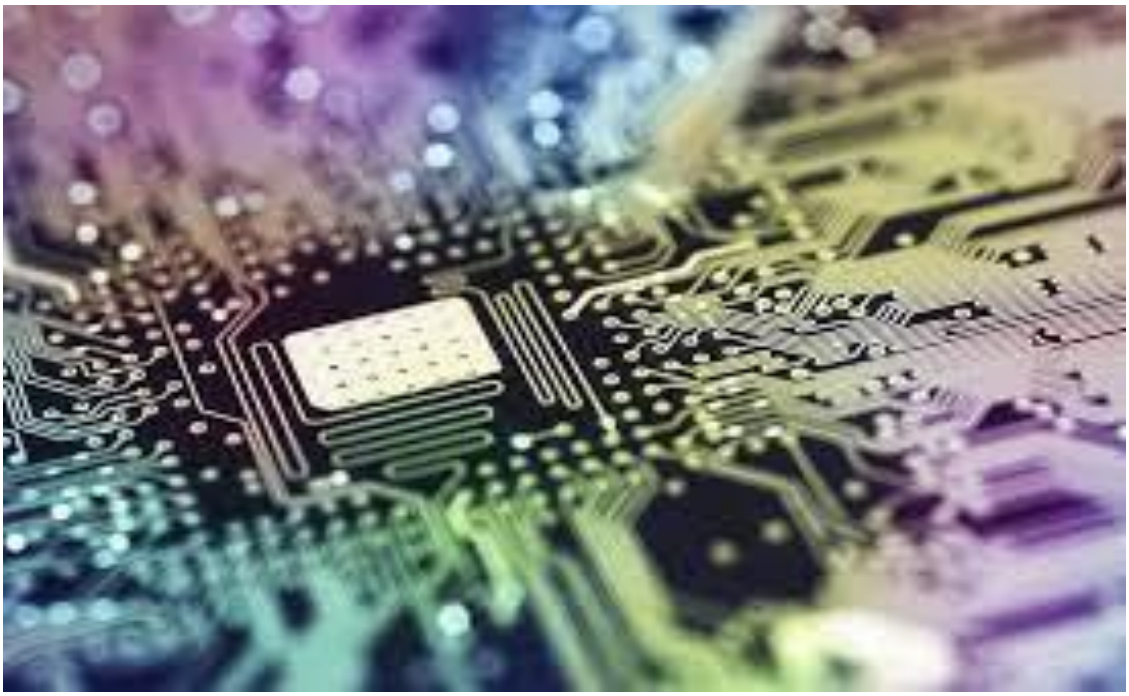
Department of Electronics & Communication Engineering

(242553)

ARM MICROCONTROLLERS LAB

M. TECH. - I YEAR- II SEMESTER (ECE)

R24 (MLRS) REGULATION



A.Y : 2024 - 2025

INDEX

MARRI LAXMAN REDDY
INSTITUTE OF TECHNOLOGY AND MANAGEMENT
(AN AUTONOMOUS INSTITUTION)
(Approved by AICTE, New Delhi & Affiliated to JNTUH, Hyderabad)
Accredited by NBA and NAAC with 'A' Grade & Recognized Under Section2(f) & 12(B)of the UGC act,1956

CERTIFICATE

This is to certify that this manual is a bonafide record of practical work in the ***ARM Microcontrollers lab*** in I **Semester of I -year M. Tech Sem II (ECE) Programme** during the academic year **2024-2025.** This book is prepared by **Dr. N Srinivas (Associate Professor), Mrs. R Babitha (Assistant Professor), Mrs. B Manjula (Assistant Professor),** Department of Electronics and Communication Engineering.

LAB I/C **Head of the Department**

PREFACE

It is one of the core areas of ECE and constitutes the largest applications in use today. Communication has entered into every part of today's world. This laboratory is intended to make students understand the use of ARM Microcontrollers and is designed to help students understand the basic principles of design techniques as well as giving them the insight on design, simulation and hardware implementation of circuits. The main aim is to provide hands-on experience to the students so that they are able to put theoretical concepts to practice. The content of this course consists of two parts, 'simulation' and 'hardwired'. Computer simulation is stressed upon as it is a key analysis tool of engineering design. "CORTEX-M3 development boards and using GNU tool chain" is used for simulation and synthesis of experiments. Students will carry out design experiments as a part of the experiments list provided in this lab manual. Students will be given a specific design problem, which after completion they will verify using the simulation software or hardwired implementation.

By,
Dr. N Srinivas (Associate Professor),
Mrs. B Manjula (Assistant Professor),
Mrs. R. Babitha (Assistant Professor),

ACKNOWLEDGEMENT

It was really a good experience, working with ***ARM Microcontrollers Laboratory*.** First, we would like to thank Dr.N.Srinivas, Assoc. Professor, HOD of Department of Electronics and Communication Engineering, Marri Laxman Reddy Institute of technology & Management for his concern and giving the technical support in preparing the document.

We are deeply indebted and gratefully acknowledge the constant support and valuable patronage of Dr.Ravi Prasad, Dean, Marri Laxman Reddy Institute of technology & Management for giving us this wonderful opportunity for preparing the Digital Control Systems Laboratory manual.

We express our hearty thanks to Dr. R. Murali Prasad, Principal, Marri Laxman Reddy Institute of technology & Management, for timely corrections and scholarly guidance.

At last, but not the least I would like to thanks the entire ECE Department faculty those who had inspired and helped us to achieve our goal.

By,
Dr. N Srinivas (Associate Professor),
Mrs. B Manjula (Assistant Professor),
Mrs. R Babitha (Assistant Professor),

GENERAL INSTRUCTIONS

1. Students should report to the concerned labs as per the timetable schedule.
2. Students who turn up late to the labs will in no case be permitted to perform the experiment scheduled for the day.

3. After completion of the experiment, certification of the concerned staff in-charge in the observation book is necessary.

4. Students should bring a notebook of about 100 pages and should enter the readings/observations into the notebook while performing the experiment.

5. The record of observations along with the detailed experimental procedure of the experiment.
6. Performed in the immediate last session should be submitted and certified by the staff member in-charge.

7.. Not more than one student is permitted to perform the experiment on a setup.
8. When the experiment is completed, students should disconnect the setup made by them, and should return all the components/instruments taken for the purpose.

9. Any damage of the equipment or burnout of components will be viewed seriously by putting penalty.

10. Students should be present in the labs for the total scheduled duration.
11. Students are required to prepare thoroughly to perform the experiment before coming to Laboratory.

12. Procedure sheets/data sheets provided to the student's should be maintained neatly and to be returned after the experiment.

SAFETY PRECAUTIONS

1. No horseplay or running is allowed in the labs.
2. No bare feet or open sandals are permitted.
3. Before energizing any equipment, check whether anyone is in a position to be injured by your actions.
4. Read the appropriate equipment instruction manual sections or consult with your instructor.
5. Before applying power or connecting unfamiliar equipment or instruments into any circuits.
6. Position all equipment on benches in a safe and stable manner.
7. Do not make circuit connections by hand while circuits are energized. This is especially.
8. Dangerous with high voltage and current circuits.

Vision of the Institute

To be a globally recognized institution that fosters innovation, excellence, and leadership in education, research, and technology development, empowering students to create sustainable solutions for the advancement of society.

Mission of the Institute

To foster a transformative learning environment that empowers students to excel in engineering, innovation, and leadership.
To produce skilled, ethical, and socially responsible engineers who contribute to sustainable technological advancements and address global challenges.
To shape future leaders through cutting-edge research, industry collaboration, and community engagement.

Quality Policy

The management is committed in assuring quality service to all its stakeholders, students, parents, alumni, employees, employers, and the community.
Our commitment and dedication are built into our policy of continual quality improvement by establishing and implementing mechanisms and modalities ensuring accountability at all levels, transparency in procedures, and access to information and actions.

Department of Electronics and Communication Engineering
Vision of the Department

To provide quality technical education in Electronics and Communication Engineering through research, innovation, striving for global recognition in specified domain, leadership, and sustainable societal solutions.

Mission of the Department

- To create a transformative learning environment that empowers students in electronics and communication engineering, fostering excellence in technical skills and leadership.
- To drive innovation through research, deliver a transformative education grounded in ethical principles, and nurture the development of professionals
- To cultivate strong industry partnerships, and engaging actively with the community for societal and technological progress.

Program educational Objectives (PEOs)

PEO 1: Have Successful career in Industry

Graduates will excel in the Electronics and Communication industry with a strong foundation in technical expertise, continuous learning, and innovation.

PEO 2: Show Excellence in higher studies/Research

Graduates will excel in higher studies and research in Electronics and Communication Engineering (ECE) through a combination of rigorous academic dedication, cutting-edge innovation, and a deep understanding of emerging technologies.

PEO 3: Show Good Competency towards Entrepreneurship

Graduates will have *t*o show good competency towards entrepreneurship in the field of Electronics and Communication Engineering, one must demonstrate an in-depth understanding of emerging technologies, market trends, and the ability to innovate within this rapidly evolving industry.

Program Specific Outcomes (PSOs)

1. Analyze and design analog & digital circuits or systems for a given specification and function.
2. Implement functional blocks of hardware-software co-designs for signal processing and communication applications.

Department of Electronics and Communication Engineering

Program Outcomes (POs)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and underst and ing of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

1. Analyze and design analog & digital circuits or systems for a given specification and function.
2. Implement functional blocks of hardware-software co-designs for signal processing and communication applications.

COURSE STRUCTURE

Level	Credits	Periods/Week	Prerequisites
PG	2	3	Entire subject of ARM MICROCONTROLLERS

Evaluation Scheme:

MID (Internal Lab) Semester Test	40 marks
End Semester Lab external Examination	60marks

The end semester examination shall be conducted with an external examiner internal examiner.

The external examiner shall be appointed by the principal / Chief Controller of examinations

Course Objectives:

- The ability to code and utilize tool sets in CortexM3.
- Know the real time alterations using PLL modules.
- Learn controlling intensity of an LED using PWM.
- Understand the UART Echo test.
- Learn good LED design techniques per current industrial practices.
- Learn design techniques of sample sound using microphone and display.

Course Outcomes:

At the end of the laboratory work, students will be able to

- Describe develop prototype codes on CORTEXM3.
- Design Circuits in arm micro controllers.
- Utilize tool set for developing applications based on ARM processor core SOC.
- Develop prototype codes using commonly available on and off chip peripherals on CortexM3 development boards.

- Verify analog reading on potentiometer connected to ADC channel.
- Describe system reset using watchdog timer.
- Synthesize temperature indication on RGB LED.
- Implement sleep modes by putting core in sleep modes.

CO1: To ability to code and utilize tool sets in CortexM3.

CO2: Write real times alterations using PLL modules.

CO3: Learn LED design techniques.

CO4: To design techniques of sample sound using microphone and display.

CO5: To Understand the UART Echo test.

Course Outcomes (CO's)–Program Outcomes (PO's)Mapping

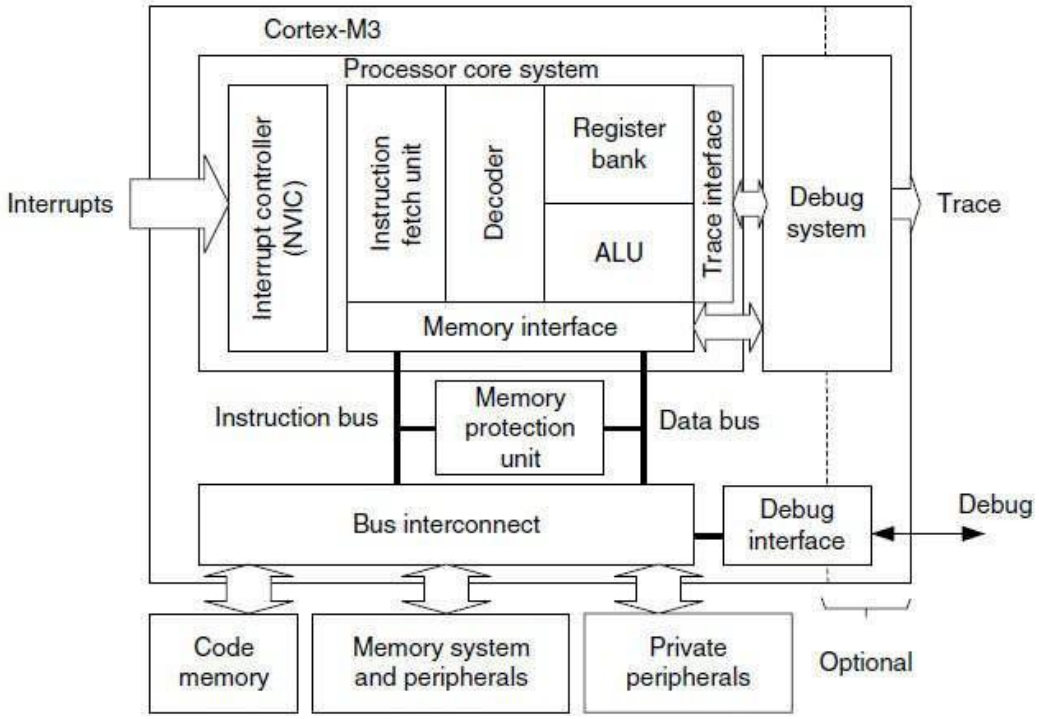
CO's $P_{O's}$	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	3	-	-	3	-	-	-	-	-	-	-
CO2	3	3	-	-	3	-	-	-	-	-	-	-
CO3	3	3	-	-	3	-	-	-	-	-	-	-
CO4	3	3	-	-	3	-	-	-	-	-	-	-
CO5	3	3	-	-	3	-	-	-	-	-	-	-

- **Simple-1** **Moderate-2** **Hig**

INTRODUCTION TO ARM Cortex M3 PROCESSOR

1.1 Introduction

The ARM Cortex-M3 is a general purpose 32-bit microprocessor, which offers high performance and very low power n consumption. The Cortex-M3 offers many new features, including a Thumb- 2 instruction set, low interrupt latency, hardware divide, interruptible/continuable multiple load and store instructions, automatic state save and restore for interrupts, tightly integrated interrupt controller with Wake-up Interrupt Controller and multiple core buses capable of simultaneous accesses. Pipeline techniques are employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory. The processor has a Harvard architecture, which means that it has a separate instruction bus and data bus. This allows instructions and data accesses to take place at the same time, and as a result of this, the performance of the processor increases because data accesses do not affect the instruction pipeline. This feature results in multiple bus interfaces on Cortex-M3, each with optimized usage and the ability to be used simultaneously. However, the instruction and data buses share the same memory space (a unified memory system). In other words, you cannot get 8 GB of memory space just because you have separate bus interfaces. A simplified block diagram of the Cortex-m3 architecture is shown below



It is worthwhile highlighting that the Cortex-M3 processor is not the first ARM processor to be used to create generic micro controllers. The venerable ARM7 processor has been very successful in this market, The Cortex-M3 processor builds on the success of the ARM7processor to deliver devices that are

significantly easier to program and debug and yet deliver a higher processing capability.

1.2 Background of ARMarchitecture

ARM was formed in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer,Acorn Computer Group, and VLSI Technology. In 1991, ARM introduced the ARM6 processor family, and VLSI became the initial licensee. Subsequently, additional companies, including Texas Instruments, NEC, Sharp, and ST Microelectronics, licensed the ARM processor designs, extending the applications of ARM processors into mobile phones, computer hard disks, personal digital assistants (PDAs), home entertainment systems, and many other consumer products. Nowadays, ARM partners ship in excess of 2 billion ARM processors each year. Unlike many semiconductor companies, ARM does not manufacture processors or sell the chips directly. Instead, ARM licenses the processor designs to business partners, including a majority of the world‟s leading semiconductor companies. Based on the ARM low-cost and power-efficient processor designs, these partners create their processors, micro controllers, and system-on- chip solutions. This business model is commonly called intellectual property (IP) licensing.

1.3 Architecture versions

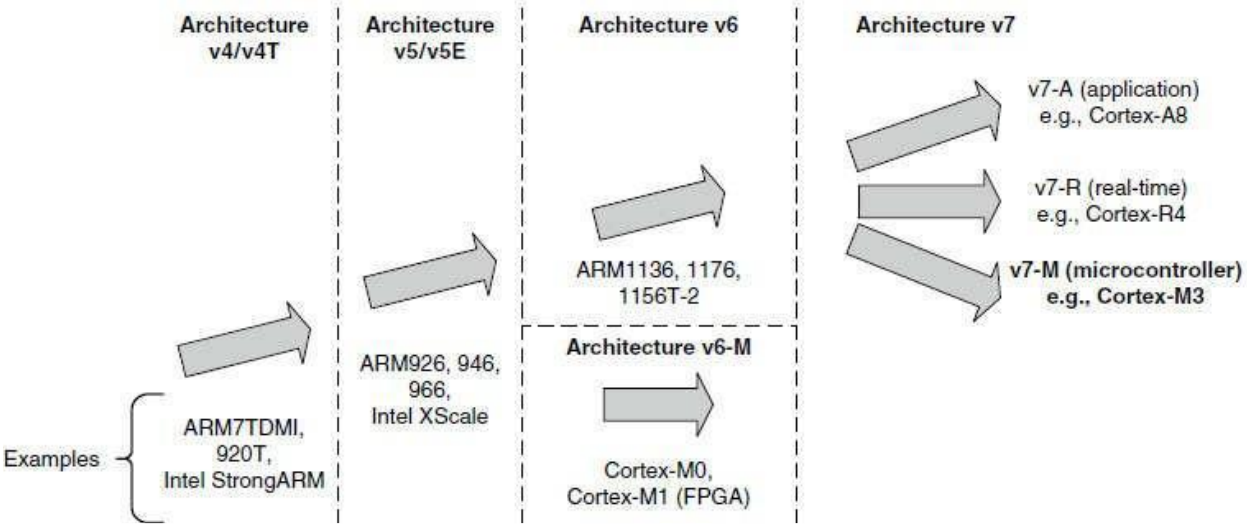
Over the years, ARM has continued to develop new processors and system blocks. These include the popular ARM7TDMI processor and, more recently, the ARM1176TZ (F)-S processor, which is used in high-end applications such as smart phones. The evolution of features and enhancements to the processors over time has led to successive versions of the ARM architecture. Note that architecture version numbers are independent from processor names. For example, the ARM7TDMI processor is based on the ARMv4T architecture (the *T* is for *Thumb* instruction mode support). The ARMv5E architecture was introduced with the ARM9E processor families, including the ARM926E-S and ARM946E-S processors. This architecture added “Enhanced” Digital Signal Processing (DSP) instructions for multimedia applications. With the arrival of the ARM11 processor family, the architecture was extended to the ARMv6. New features in this architecture included memory system features and Single Instruction–Multiple Data (SIMD) instructions. Processors based on the ARMv6 architecture include the ARM1136J (F)-S, the ARM1156T2 (F)- S, and the ARM1176JZ (F)-S. Over the past several years, ARM extended its product portfolio by diversifying its CPU development, which resulted in the architecture version7 or v7. In this version, the architecture design is divided into three profiles:

- The ***A profile*** is designed for high-performance open application platforms.
- The ***R profile*** is designed for high-end embedded systems in which real-time performance is needed.
- The***Mprofile***isdesignedfordeeplyembeddedmicrocontroller-typesystems. Bit more details on these profiles

***A Profile (ARMv7-A)*:** Application processors which are designed to handle complex applications such as high-end embedded operating systems (OSs) (e.g., Symbian, Linux, and Windows Embedded). These processors requiring the highest processing power, virtual memory system support with memory management units (MMUs), and, optionally, enhanced Java support and a secure program execution environment. Example products include high-end mobile phones and electronic wallets for financial transactions.

***R Profile (ARMv7-R)*:** Real-time, high-performance processors targeted primarily at the higher end of the real-time market, those applications, such as high-end breaking systems and hard drive controllers.

***M Profile (ARMv7-M)*:** Processors targeting low-cost applications in which processing efficiency is important and cost, power consumption, low interrupt latency, and ease of use are critical,aswellasindustrialcontrolapplications,includingreal-timecontrolsystems.TheCortex processor families are the first products developed on architecture v7, and the Cortex- M3 processor is based on one profile of the v7 architecture, called ARM v7-M, an architecture specification for micro controller products. Below figure shows the development stages of ARM versions



1.4 Instruction Set Development

Enhancement and extension of instruction sets used by the ARM processors has been one of the key driving forces of the architecture's evolution. Historically (since ARM7TDMI), two different instruction sets are supported on the ARM processor: the ARM instructions that are 32 bits and Thumb instructions that are 16 bits. During program execution, the processor can be dynamically switched between the ARM state and the Thumb state to use either one of the instruction sets. The Thumb instruction set provides only a subset of the ARM instructions, but it can provide higher code density. It is useful for products with tight memory requirements.

The Thumb-2 Technology and Instruction Set Architecture

The Thumb-2 technology extended the Thumb Instruction Set Architecture (ISA) into a highly efficient and powerful instruction set that delivers significant benefits in terms of ease of use, code size, and performance. The extended instruction set in Thumb-2 is a super set of the previous 16-bit Thumb instruction set, with additional 16-bit instructions alongside 32-bit instructions. It allows more complex operations to be carried out in the Thumb state, thus allowing higher efficiency by reducing the number of states switching between ARM state and Thumb state. Focused on small memory system devices such as micro controllers and reducing the size of the processor, the Cortex-M3 supports only the Thumb-2 (and traditional Thumb) instruction set. Instead of using ARM instructions for some operations, as in traditional ARM processors,

it uses the Thumb-2 instruction set for all operations. As a result, the Cortex-M3 processor is not backward compatible with traditional ARM processors.
Nevertheless, the Cortex-M3 processor can execute almost all the 16-bit Thumb instructions, including all 16-bit Thumb instructions supported on ARM7 family processors, making application porting easy. With support for both 16-bit and 32-bit instructions in the Thumb-2 instruction set, there is no need to switch the processor between Thumb state (16-bit instructions) and ARM state (32-bit instructions). For example, in ARM7 or ARM9 family processors, you might need to switch to ARM state if you want to carry out complex calculations or a large number of conditional operations and good performance is needed, whereas in the Cortex-M3 processor, you can mix 32-bit instructions with 16-bit instructions without switching state, getting high code density and high performance with no extra complexity. The Thumb-2 instruction set is a very important feature of the ARMv7 architecture. Compared with the instructions supported on ARM7 family processors (ARMv4T architecture), the Cortex- M3 processor instruction set has a large number of new features. For the first time, hardware divide instruction is available on an ARM processor, and a number of multiply instructions are also available on the Cortex-M3 processor to improve data-crunching performance. The Cortex- M3 processor also supports unaligned data accesses, a feature previously available only in high- end processors.

1.5 Advantages of Cortex M3processors

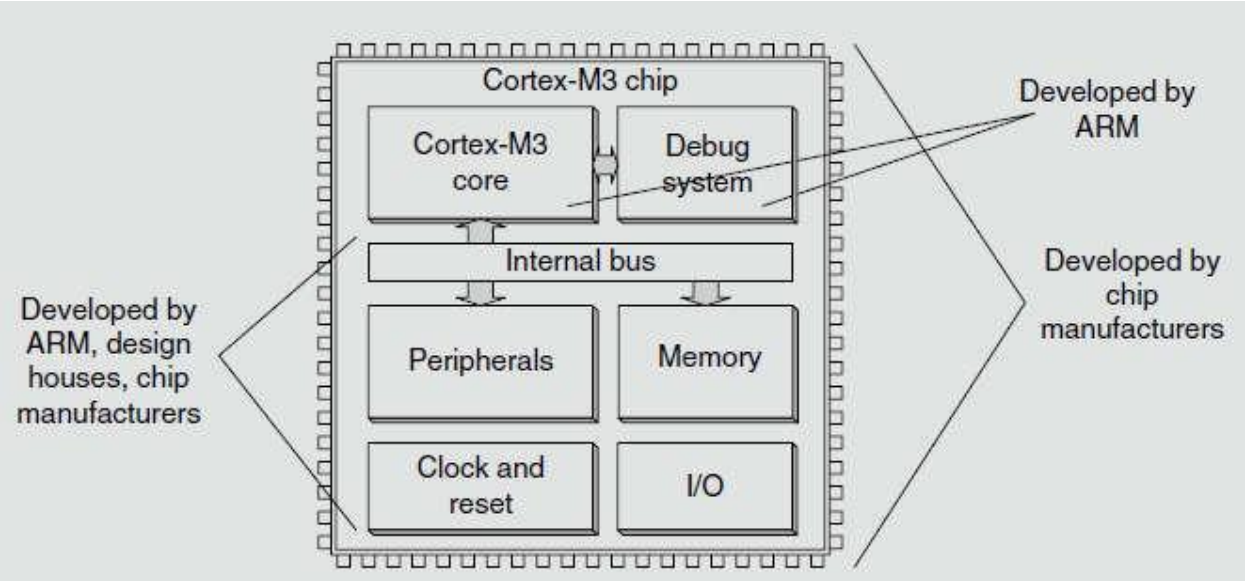
The Cortex-M3 addresses the requirements for the 32-bit embedded processor market in the following ways: ***Greater performance efficiency*:** allowing more work to be done without increasing the frequency or power requirements ***Low power consumption*:** enabling longer battery life, especially critical in portable products including wireless networking applications ***Enhanced determinism*:** guaranteeing that critical tasks and interrupts are serviced as quickly as possible and in a known number of cycles ***Improved code density*:** ensuring that code fits in even the smallest memoryfootprints ***Easeofuse*:**providingeasierprogramminganddebuggingforthegrowingnumberof8-bitand 16-bit users migrating to 32bits ***Lower cost solutions*:** reducing 32-bit-based system costs close to those of legacy 8-bit and 16-bit devices and enabling low-end, 32-bit micro controllers to be priced at less than US$1 for the first time ***Wide choice of development tools*:** from low-cost or free compilers to full-featured development suites from many development tool vendors Cost savings can be achieved by improving the amount of code reuse across all systems. Because Cortex-M3 processor-based micro controllers can be easily programmed using the C language and are based on a well-established architecture, application code can be ported and reused easily, and reducing development time and testing costs.

1.6 Applications of Cortex M3processors

***Low-cost micro controllers*:** The Cortex-M3 processor is ideally suited for low-cost micro controllers, which are commonly used in consumer products, from toys to electrical appliances. It is a highly competitive market due to the many well-known 8-bit and 16-bit micro controller products on the market. Its lower power, high performance, and ease-of-use advantages enable embedded developers to migrate to 32-bit systems and develop products with the ARM architecture. ***Automotive*:** Another ideal application for the Cortex-M3 processor is in the automotive industry. The Cortex-M3 processor has very high-performance efficiency and low interrupt latency, allowing it to be used in real-time systems. The Cortex-M3 processor supports up to 240 external vectored interrupts, with a built-in interrupt controller with nested interrupt supports and an optional MPU, making it ideal for highly integrated and cost-sensitive automotive applications. ***Data communications*:** The processor's low power and high efficiency, coupled with instructions in Thumb-2 for bit-field manipulation, make the Cortex-M3 ideal for many communications applications, such as Bluetooth and ZigBee.

***Industrial control*:** In industrial control applications, simplicity, fast response, and reliability are key factors. Again, the Cortex-M3 processors interrupt feature, low interrupt latency, and enhanced fault-handling features make it a strong candidate in this area. ***Consumer products*:** In many consumer products, a high-performance microprocessor (or several of them) is used. The Cortex-M3 processor, being a small processor, is highly efficient and low in power and supports an MPU enabling complex software to execute while providing robust memory protection.

1.7 TheCortex-M3ProcessorversusCortex-M3-BasedMicro Controllers



The Cortex-M3 processor is the central processing unit (CPU) of a micro controller chip. In addition, a number of other components are required for the whole Cortex-M3 processor-based micro controller. After chip manufacturers license the Cortex-M3 processor, they can put the Cortex-M3 processor in their silicon designs, adding memory, peripherals, input/output (I/O), and other features. Cortex-M3 processor-based chips from different manufacturers will have different memory sizes, types, peripherals, and features.

2. INTRODUCTION TO MICRO CONTROLLERLPC1768

2.1 Architectural Overview

The LPC1768FBD100 is an ARM Cortex-M3 based micro controller for embedded applications requiring a high level of integration and low power dissipation. The ARM Cortex-M3 is a next generation core that offers system enhancements such as modernized debug features and a higher level of support block integration. LPC1768 operate up to 100 MHz CPU frequency. The peripheral complement of the LPC1768 includes up to 512 kilo bytes of flash memory, up to 64KB of data memory, Ethernet MAC, a USB interface that can be configured as either Host, Device, or OTG, 8 channel general purpose DMA controller, 4 UARTs, 2 CAN channels, 2 SSP controllers, SPI interface, 3 I2C interfaces, 2-input plus 2-output I2S interface, 8 channel 12-bit ADC, 10-bit DAC, motor control PWM, Quadrature Encoder interface, 4 general purpose timers, 6-output general purpose PWM, ultra-low power RTC with separate battery supply, and up to 70 general purpose I/O pins. The LPC1768 use a multi layer AHB(Advanced High Performance Bus) matrix to connect the ARM Cortex-M3 buses and other bus masters to peripherals in a flexible manner that optimizes performance by allowing peripherals that are on different slaves ports of the matrix to be accessed simultaneously by different bus masters.

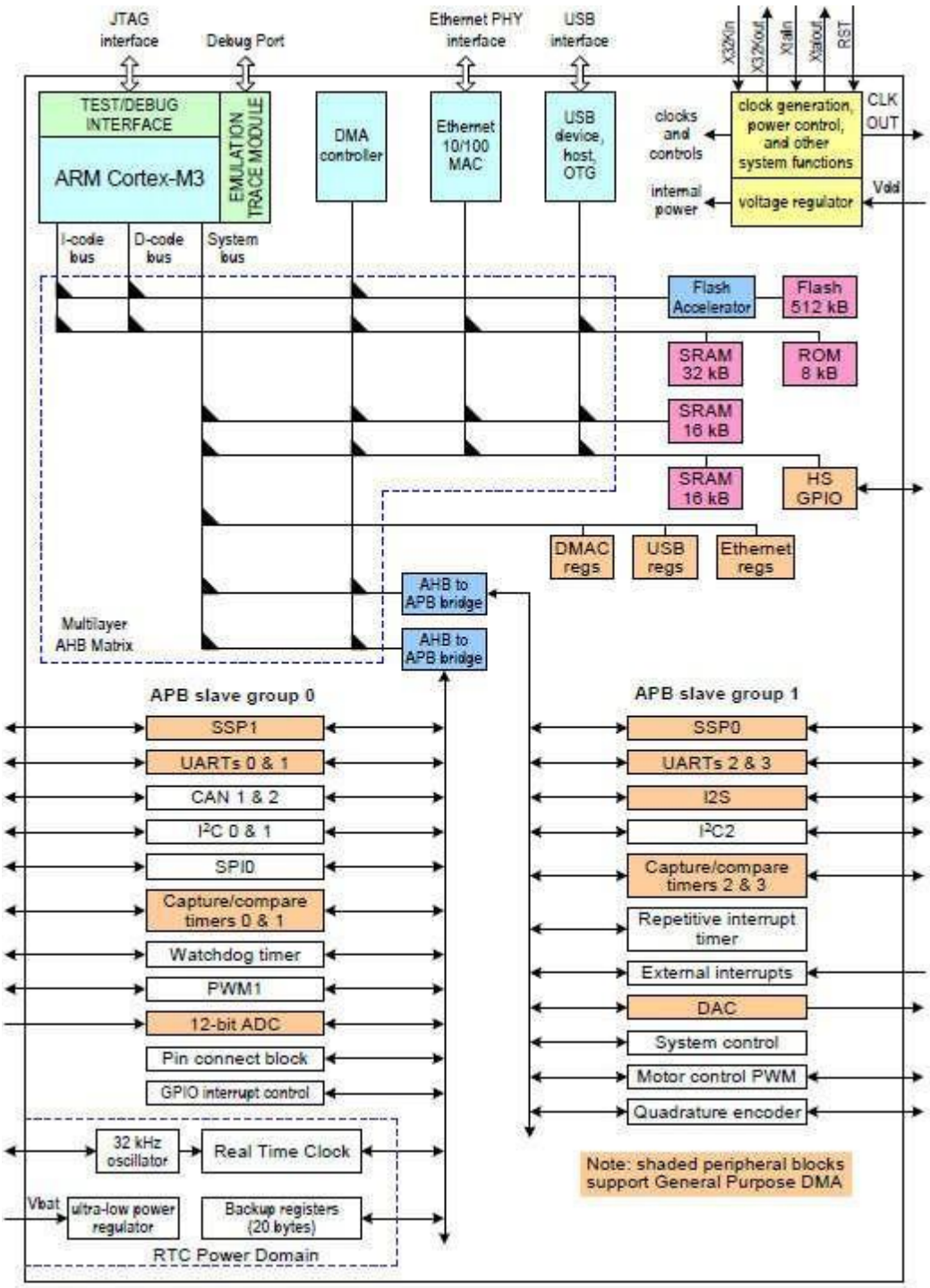
On-chip flash memory system

The LPC1768 contains up to 512 KB of on-chip flash memory. A flash memory accelerator maximizes performance for use with the two fast AHB Lite buses. This memory may be used for both code and data storage. Programming of the flash memory may be accomplished in several ways. It may be programmed In System via the serial port. The application program may also erase and/or program the flash while the application is running, allowing a great degree of flexibility for data storage field firmware upgrades, etc.

On-chip Static RAM

The LPC1768 contains up to 64 KB of on-chip static RAM memory. Up to 32 KB of SRAM, accessible by the CPU and all three DMA controllers are on a higher-speed bus. Devices containing more than 32 KB SRAM have two additional 16 KB SRAM blocks, each situated on separate slave ports on the AHB multilayer matrix. This architecture allows the possibility for CPU and DMA accesses to be separated in such a way that there are few or no delays for the bus masters.

2.2 Block Diagram ofLPC1768FBD10



2.3 A brief description of the blocks:

Nested vector interrupt controller The NVIC is an integral part of the Cortex-M3. The tight coupling to the CPU allows for low interrupt latency and efficient processing of late arriving interrupts.

Features

- Controls system exceptions and peripheral interrupts
- In the LPC1768, the NVIC supports 33 vectored interrupts
- 32 programmable interrupt priority levels, with hardware priority level masking
- Relocatable vector table
- Non-Maskable Interrupt (NMI)
- Software interrupt generation

Interrupt sources

Each peripheral device has one interrupt line connected to the NVIC but may have several interrupt flags. Individual interrupt flags may also represent more than one interrupt source. Any pin on Port 0 and Port 2 (total of 42 pins) regardless of the selected function, can be programmed to generate an interrupt on a rising edge, a falling edge, or both.

General purpose DMA controller

The GPDMA (General Purpose Direct Memory Access) is an AMBA AHB (Advanced Micro controller Bus Architecture Advance high performance bus) compliant peripheral allowing selected peripherals to have DMA support. The GPDMA enables peripheral-to-memory, memory-to-peripheral, peripheral-to-peripheral and memory-to-memory transactions. The source and destination areas can each be either a memory region or a peripheral, and can be accessed through the AHB master. The GPDMA controller allows data transfers between the USB and Ethernet controllers and the various on- chip SRAM areas. The supported APB peripherals are SSP0/1, all UARTs,theI2S-businterface, theADC,andtheDAC.TwomatchsignalsforeachtimercanbeusedtotriggerDMAtransfers.

Function Configuration block

Theselectedpinsofthemicrocontrollertohavemorethanonefunction.Configurationregisters control the multiplexers to allow connection between the pin and the on-chip peripherals. Peripheralsshouldbeconnectedtotheappropriatepinspriortobeingactivatedandpriortoany related interrupt(s) being enabled. Activity of any enabled peripheral function that is not mapped to a related pin should be considered undefined. Most pins can also be configured as open-drain outputs or to have a pull- up, pull-down, or no resistor enabled.

Fast general-purpose parallel I/O

Device pins that are not connected to a specific peripheral function are controlled by the GPIO registers. Pins may be dynamically configured as inputs or outputs. Separate registers allow setting or clearing any number of outputs simultaneously. The value of the output register may be read back as well as the current state of the port pins.

USB interface

TheUniversalSerialBus(USB)isa4-wirebusthatsupportscommunicationbetweenahostand oneormore(upto127)peripherals.ThehostcontrollerallocatestheUSBbandwidthto attached devices through a token-based protocol. The bus supports hot plugging and dynamic configuration of the devices. All transactions are initiated by the host controller. The USB interface includes a device, Host, and OTG controller with on-chip PHY for device and Host functions. The OTG switching protocol is supported through the use of an external controller. **USB device controller** enables 12 Mbit/s data exchange with a USB Host controller. It consists of a register interface, serial interface engine, endpoint buffer memory, and a DMA controller. The serial interface engine decodes the USB data stream and writes data to the appropriate endpoint buffer. The status of a completed USB transfer or error condition is indicated via status registers. An interrupt is also generated if enabled. When enabled, the DMA controller transfers data between the endpoint buffer and the on-chip SRAM.

12-bit ADC

The LPC1768 contain a single 12-bit successive approximation ADC with eight channels and DMA support.

10-bit DAC

The DAC allows to generate a variable analog output. The maximum output value of the DAC is VREFP.

UART's

The LPC1768 contain four UART's. In addition to standard transmit and receive data lines, UART1 also provides a full modem control handshake interface and support for RS-485/9-bit mode allowing both software address detection and automatic address detection using 9-bit mode. The UART's include a fractional baud rate generator. Standard baud rates such as 115200 Baud can be achieved with any crystal frequency above 2 MHz

SPI serial I/O controller

The LPC1768 contain one SPI controller. SPI is a full duplex serial interface designed to handle multiple masters and slaves connected to a given bus. Only a single master and a single slave can communicate on the interface during a given data transfer. During a data transfer the master always sends 8 bits to 16 bits of data to the slave, and the slave always sends 8 bits to 16 bits of data to the master.

SSP serial I/O controller

The LPC1768 contain two SSP controllers. The SSP controller is capable of operation on a SPI, 4-wire SSI, or Micro wire bus. It can interact with multiple masters and slaves on the bus. Only a single master and a single slave can communicate on the bus during a given data transfer. The SSP supports full duplex transfers, with frames of 4 bits to 16 bits of data flowing from the master to the slave and from the slave to the master. In practice, often only one of these data flows carries meaningful data.

I2C-bus serial I/O controllers

The LPC1768 each contain three I2C-bus controllers. The I2C-bus is bidirectional for inter-IC control using only two wires: a Serial Clock line (SCL) and a Serial DAta line (SDA). Each device is recognized by a unique address and can operate as either a receiver-only device or a transmitter with the capability to both receive and send information (such as memory).

Transmitters and/or receivers can operate in either master or slave mode, depending on whether the chip has to initiate a data transfer or is only addressed. The I2C is a multi-master bus and can be controlled by more than one bus master connected to it. General purpose 32-bit timers/external event counters The LPC1768 include four 32-bit timer/counters. The timer/counter is designed to count cycles of the system derived clock or an externally-supplied clock. It can optionally generate interrupts, generate timed DMA requests, or perform other actions at specified timer values, based on four match registers. Each timer/counter also includes two capture inputs to trap the timer value when an input signal transitions, optionally generating an interrupt.

Pulse width modulator

The PWM is based on the standard Timer block and inherits all of its features, although only the PWM function is pinned out on the LPC1768. The Timer is designed to count cycles of the system derived clock and optionally switch pins, generate interrupts or perform other actions when specified timer values occur, based on seven match registers. The PWM function is in addition to these features, and is based on match register events.

Watchdog timer

The purpose of the watchdog is to reset the micro controller within a reasonable amount of time if it enters an erroneous state. When enabled, the watchdog will generate a system reset if the userprogramfailsto„feed"(orreload)thewatchdogwithinapredeterminedamountoftime.

RTC and backup registers

The RTC is a set of counters for measuring time when system power is on, and optionally when it is off. The RTC on the LPC1768 is designed to have extremely low power consumption, i.e. less than 1 uA. The RTC will typically run from the main chip power supply, conserving battery power while the rest of the device is powered up. When operating from a battery, the RTC will continue working down to 2.1 V. Battery power can be provided from a standard 3 V Lithium button cell. An ultra-low power 32 kHz oscillator will provide a 1 Hz clock to the time counting portion of the RTC, moving most of the power consumption out of the time counting function.

Clocking and Power Control Crystal oscillators

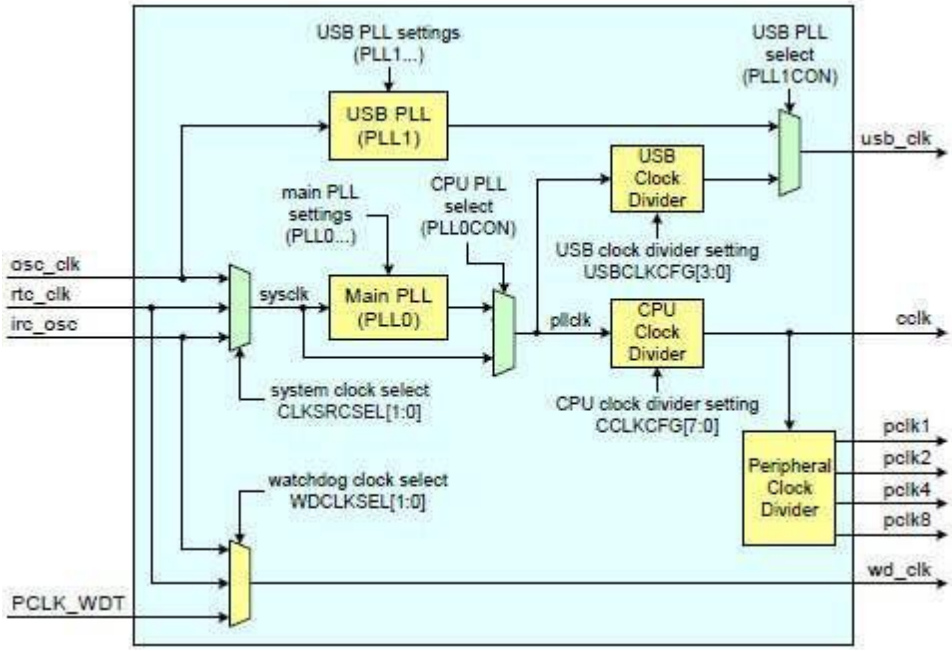
The LPC1768 include three independent oscillators. These are the main oscillator, the IRC oscillator, and the RTC oscillator. Each oscillator can be used for more than one purpose as required in a particular application. Any of the three clock sources can be chosen by software to drive the main PLL and ultimately the CPU. Following reset, the LPC1768 will operate from the Internal RC oscillator until switched by software. This allows systems to operate without any external crystal and the boot loader code to operate at a known frequency.

Power control

The LPC1768 support a variety of power control features. There are four special modes of processor power reduction: Sleep mode, Deep-sleep mode, Power-down mode, and Deep power-down mode. The CPU clock rate may also be controlled as needed by changing clock sources, reconfiguring PLL values, and/or altering the CPU clock divider value. This allows a trade-off of power versus processing speed based on application requirements. In addition, Peripheral Power Control allows shutting down the clocks to individual on-chip peripherals, allowing fine tuning of power consumption by eliminating all dynamic power use in any peripherals that are not required for the application. Each of the peripherals has its own clock divider which provides even better power control. Integrated PMU (Power Management Unit) automatically adjust

internal regulators to minimize power consumption during Sleep, Deep sleep, Power-down, and Deep power- down modes. The LPC1768 also implement a separate power domain to allow turning off power to the bulk of the device while maintaining operation of the RTC and a small set of registers for storing data during any of the power-down modes.

Clock generation block diagram for LPC1768 is shown below



System Control Reset

Reset has four sources on the LPC1768: the RESET pin, the Watchdog reset, power-on reset (POR),andtheBrown-OutDetection(BOD)circuit.TheRESETpinisaSchmitttriggerinputpin. Assertion of chip Reset by any source, once the operating voltage attains a usable level, causes the RSTOUT pin to go LOW. Once reset is de-asserted, or, in case of a BOD- triggered reset, once the voltage rises above the BOD threshold, the RSTOUT pin goes HIGH. In other words RSTOUT is high when the controller is in its active state.

Emulation and debugging

Debug and trace functions are integrated into the ARM Cortex-M3. Serial wire debug and trace functions are supported in addition to a standard JTAG debug and parallel trace functions. The ARM Cortex-M3 is configured to support up to eight breakpoints and four watch points.

Note: For further details on Controller blocks refer the User manual of LPC176x/5x – UM10360 available at www.nxp.com

3. TECHNICAL SPECIFICATIONS ofLPC1768

3.1 Specifications ofLPC1768:

 ARM Cortex-M3 processor runs up to 100 MHz frequency.

 ARM Cortex-M3 built-in Nested Vectored Interrupt Controller(NVIC).

 Up to 512kB on-chip flash program memory with In-System Programming (ISP) and In- Application Programming (IAP) capabilities. The combination of an enhanced flash memory accelerator and location of the flash memory on the CPU local code/data bus provides high code performance from flash.

 Up to 64kB on-chip SRAM includes:

- Up to 32kB of SRAM on the CPU with local code/data bus for high-performance CPU access.
- Up to two 16kB SRAM blocks with separate access paths for higher throughput. These SRAM blocks may be used for Ethernet, USB, and DMA memory, as well as for general purpose instruction and data storage.

 Eight channel General Purpose DMA controller (GPDMA) on the AHB multilayer matrix that can be used with the SSP, I2S, UART, the Analog-to-Digital and Digital-to-Analog converterperipherals,timermatchsignals,GPIO,andformemory-to-memorytransfers.

 Serial interfaces:

- Ethernet MAC with RMII interface and dedicated DMA controller.
- USB 2.0 full-speed controller that can be configured for either device, Host, or OTG operation with an on-chip PHY for device and Host functions and a dedicated DMA controller.
- Four UART's with fractional baud rate generation, internal FIFO, IrDA, and DMA support. One UART has modem control I/O and RS-485/EIA-485support.
- Two-channel CAN controller.
- Two SSP controllers with FIFO and multi-protocol capabilities. The SSP interfaces can be used with the GPDMA controller.
- SPI controller with synchronous, serial, full duplex communication and programmable data length. SPI is included as a legacy peripheral and can be used instead ofSSP0.
- Three enhanced I2C-bus interfaces, one with an open-drain output supporting the full I2C specification and Fast mode plus with data rates of 1Mbit/s, two with standard port pins. Enhancements include multiple address recognition and monitor mode.
- I2S (Inter-IC Sound) interface for digital audio input or output, with fractional rate control. The I2S interface can be used with the GPDMA. The I2S interface supports 3- wire data transmit and receive or 4-wire combined transmit and receive connections, as well as master clock output.

 Other peripherals:

- 70 General Purpose I/O (GPIO) pins with configurable pull-up/down resistors, open drain mode, and repeater mode. All GPIOs are located on an AHB bus for fast access, and support Cortex-M3 bit-banding. GPIOs can be accessed by the General Purpose DMA Controller. Any pin of ports 0 and 2 can be used to generate an interrupt.

- 12-bit Analog-to-Digital Converter (ADC) with input multiplexing among eight pins, conversion rates up to 200 kHz, and multiple result registers. The 12-bit ADC can be used with the GPDMA controller.
- 10-bit Digital-to-Analog Converter (DAC) with dedicated conversion timer and DMA support.

- Four general purpose timers/counters, with a total of eight capture inputs and ten compare outputs. Each timer block has an external count input. Specific timer events can be selected to generate DMA requests.
- One motor control PWM with support for three-phase motor control.
- Quadrature encoder interface that can monitor one external quadrature encoder.

- One standard PWM/timer block with external count input.
- Real-Time Clock (RTC) with a separate power domain. The RTC is clocked by a dedicated RTC oscillator. The RTC block includes 20 bytes of battery-powered backup registers, allowing system status to be stored when the rest of the chip is powered off.
Battery power can be supplied from a standard 3 V Lithium button cell. The RTC will continue working when the battery voltage drops to as low as 2.1 V. An RTC interrupt can wake up the CPU from any reduced power mode.
- Watchdog Timer (WDT). The WDT can be clocked from the internal RC oscillator, the RTC oscillator, or the APB clock.

- Cortex-M3 system tick timer, including an external clock input option. - Repetitive interrupt timer provides programmable and repeating timed interrupts.
 Standard JTAG test/debug interface as well as Serial Wire Debug and Serial Wire Trace Port options.

 Emulation trace module supports real-time trace.

 Four reduced power modes: Sleep, Deep-sleep, Power-down, and Deep power-down.

 Single3.3Vpowersupply(2.4Vto3.6V). Temperature rangeof-40°Cto85 °C.

 Fourexternalinterruptinputsconfigurableasedge/levelsensitive.AllpinsonPORT0and PORT2 can be used as edge sensitive interrupt sources.

 Non Maskable Interrupt (NMI)input.

 Clock output function that can reflect the main oscillator clock, IRC clock, RTC clock, CPU clock, or the USB clock.

 The Wake-up Interrupt Controller (WIC) allows the CPU to automatically wake up from any priority interrupt that can occur while the clocks are stopped in deep sleep, Power- down, and Deep power-down modes

 Processor wake-up from Power-down mode via any interrupt able to operate during Power-down mode (includes external interrupts, RTC interrupt, USB activity, Ethernet wake-up interrupt, CAN bus activity, PORT0/2 pin interrupt, and NMI).

 Each peripheral has its own clock divider for further power savings.

 Brownout detect with separate threshold for interrupt and forced reset.

 On-chip Power-On Reset (POR).

 On-chip crystal oscillator with an operating range of 1 MHz to 25MHz.

 4 MHz internal RC oscillator trimmed to 1% accuracy that can optionally be used as a system clock.

 Anon-chip PLL allows CPU operation upto the maximum CPUratewithouttheneedfora high-frequency crystal. May be run from the main oscillator, the internal RC oscillator, or the RTCoscillator.

 A second, dedicated PLL may be used for the USB interface in order to allow added flexibility for the Main PLL settings.

 Versatile pin function selection feature allows many possibilities for using on-chip peripheral functions.

3.2 SPECIFICATIONS OFALS-SDA-ARMCTXM3-06

 LPC1768 is **ARM Cortex M3** based micro controller with
 512KB flash memory and 64KB SRAM In-System Programming (ISP) and In- Application Programming (IAP)capabilities.
 Single 3.3 V power supply (2.4 V to 3.6V).
 70 General Purpose I/O (GPIO) pins with configurable pull-up/down resistors, open drain mode, and repeater mode.
 12-bit Analog-to-Digital Converter (ADC) and up to 8 analog channels.
 10-bit Digital-to-Analog Converter (DAC) with dedicated conversion timer.
 Four general purpose timers/counters, with a total of eight capture inputs and ten compare outputs.
 Four UART's with fractional baud rate generation, internal FIFO, IrDA.
 SPI controller with synchronous, serial, full duplex communication.
 Three enhanced I2C-businterfaces
 Four reduced power modes: Sleep, Deep-sleep, Power-down, and Deep power- down.
 Real-Time Clock (RTC) with a separate power domain.
 Standard JTAG test/debug interface as well as Serial Wire Debug.
 Four external interrupt inputs configurable as edge/level sensitive.
 12MHz Crystal allows easy communication setup
 Oneonboardvoltageregulatorforgenerating3.3V. Input to this will be from External

+5V DC Power supply through a 9-pin DSUB connector
 Piggy Back module containing **LPC1768**controller
 Standard JTAG connector with ARM 2×10 pin layout for programming/debugging with ARM-JTAG
 Reset push-button for resetting the controller
 One RS232 interface circuit with 9 pin DSUB connector: this is used by the Boot loader program, to program **LPC1768** Flash memory without external Programmer
 DC motor interface with direction and speed control
 Stepper motor interface with direction and speed control
 16×2 alphanumeric LCD Display
 On chip ADC interface circuit usingAD0.5(P1.31)
 8-bit DAC interface
 4x4 Key-Matrix connected to the port lines of the controller
 One External interrupt circuit with LED indication
 Two-digit multiplexed 7-segment display interface
 Interface circuit for on board Buzzer, Relay and Led indication controlled through push button.
 SPI Interface: 2 channel ADC IC with POT and Temperature sensor
 I2C Interface: NVROMIC
 Standard 26-pin FRC connectors to connect to **on-board interface** or some of **ALS standard External Interfaces.**

 A number of software examples in „C-language“ to illustrate the functioning of the interfaces. The software examples are compiled using an evaluation version of KEIL4 „C“ compiler for ARM.
 Compact elegant plastic enclosure
 Optional USB to Serial interface (RS232) cable.

Blink an LED with software delay, delay generated using the Sys Ticktimer	EXPT. NO: 1
	DATE:

AIM: To Blink an LED with software delay, delay generated using the Sys Ticktimer

SOFTWARE:

CORTEX-M3 development boards and using GNU tool chain.

PROGRAM:

For a Cortex-M3 microcontroller, the SysTick timer works similarly to Cortex-M4, but with a slightly different register configuration. Below is an example code for blinking an LED with a software delay generated using the SysTick timer on a Cortex-M3 microcontroller, assuming you're using the STM32 series microcontroller:

```
#include "stm32f10x.h"
void SysTick_Handler(void) {
    // This function is called whenever SysTick timer overflows
}

void delay_ms(uint32_t milliseconds) {
    // Initialize SysTick Timer
    SysTick->LOAD = SystemCoreClock / 1000 - 1; // Configure SysTick to overflow every 1ms
    SysTick->VAL = 0; // Reset the SysTick counter
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk;

// Enable SysTick
    for (uint32_t i = 0; i < milliseconds; ++i) {
        // Wait until the count flag is set
        while (!(SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk));
    }
}
```

```
int main() {
  // Enable GPIO Clock
  RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;

  // Configure PC13 as Output
  GPIOC->CRH |= GPIO_CRH_MODE13_0; // Output mode, max speed 10 MHz
  GPIOC->CRH &= ~GPIO_CRH_CNF13; // General purpose output push-pull

  while(1) {
    // Turn on LED
    GPIOC->BSRR |= GPIO_BSRR_BS13;
    delay_ms(1000); // Delay for 1 second

    // Turn off LED
    GPIOC->BSRR |= GPIO_BSRR_BR13;
    delay_ms(1000); // Delay for 1 second
  }
}
```

In this code:

- **SysTick_Handler** is the interrupt service routine (ISR) for the SysTick timer. It's called whenever the SysTick timer overflows.
- **Delay_ms** function generates a delay in milliseconds using the SysTick timer. It configures the SysTick timer to overflow every 1ms and then waits until the desired delay has passed.
- In the **main** function, PC13 pin (assuming your LED is connected to this pin) is configured as an output. Then, the LED is turned on and off with a delay of 1 second each using the **delay_ms** function.

Make sure to configure your microcontroller and toolchain properly before using this code. Also, adjust the code according to your microcontroller's datasheet and the pin connected to the LED.

RESULT: Blink an LED with software delay, delay generated using the Sys Ticktimer, hence verified.

System clock real time alteration using the PLL modules.	EXPT. NO: 2
	DATE:

AIM: To System clock real time alteration using the PLL modules.

SOFTWARE: CORTEX-M3 development boards and using GNU tool chain.

Altering the system clock in real-time using the PLL (Phase-Locked Loop) module on a Cortex-M3 microcontroller is a bit complex and usually not recommended to be done at runtime due to the potential risks involved in changing the system clock while the system is running. However, it's possible to configure and enable PLL for clock scaling before starting the main system operation. Here's an example code to demonstrate how to configure the PLL on a Cortex-M3 microcontroller, assuming you're using an STM32 series microcontroller:

PROGRAM :

```
#include "stm32f10x.h"

void SystemInit(void) {
    // Configure PLL
    RCC->CFGR &= ~(RCC_CFGR_PLLSRC | RCC_CFGR_PLLXTPRE | RCC_CFGR_PLLMULL); // Clear PLL source, PLLXTPRE and PLLMULL bits
    RCC->CFGR |= (RCC_CFGR_PLLSRC_HSE | RCC_CFGR_PLLMULL9); // Configure PLL source as HSE, PLL multiplier as 9

    // Enable HSE
    RCC->CR |= RCC_CR_HSEON;
    while(!(RCC->CR & RCC_CR_HSERDY)); // Wait until HSE is ready

    // Enable PLL
    RCC->CR |= RCC_CR_PLLON;
    while(!(RCC->CR & RCC_CR_PLLRDY)); // Wait until PLL is ready
```

```
// Select PLL as system clock source
    RCC->CFGR &= ~RCC_CFGR_SW;
    RCC->CFGR |= RCC_CFGR_SW_PLL;
    while((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL); // Wait until PLL becomes
     system clock
}

int main() {
    // Now, the system clock is configured with PLL
    // Your main application code goes here

    while(1) {
        // Your application code
    }
}
```

In this code:

- The 'SystemInit( )' function is called by the startup code before **main()** to initialize the system clock. It configures the PLL to use an external oscillator (HSE) as the PLL source and set the PLL multiplier to 9.
- The PLL is enabled, and the system waits until the PLL is ready.
- After the PLL is ready, it's selected as the system clock source.
- 'main( )' function is where your application code would normally start executing after the system clock is configured.

Remember, changing the system clock configuration should be done with caution and according to the microcontroller's datasheet. It's usually safer to configure the PLL during system initialization rather than altering it at runtime.

RESULT : System clock real time alteration using the PLL modules is verified.

Control intensity of an LED using PWM implementing in software and hardware.	EXPT. NO: 3
	DATE:

PROGRAM :

Controlling the intensity of an LED using PWM (Pulse Width Modulation) can be achieved both in software and hardware on a Cortex-M3 microcontroller. Here's an example demonstrating both methods:

Software PWM :

Software PWM involves toggling the LED at a frequency higher than the human eye can detect, and varying the duty cycle to control the perceived intensity. Here's a basic implementation:

```
#include "stm32f10x.h"

#define LED_PIN GPIO_Pin_13 // Assuming LED is connected to GPIO Pin 13
#define PWM_FREQ 1000 // PWM frequency in Hz

void delay_us(uint32_t microseconds) {
    // Assuming 72 MHz clock
    microseconds *= 72; // Convert microseconds to clock cycles at 72 MHz
    while(microseconds--) {
        asm("nop"); // No operation, consumes one cycle
    }
}
void set_pwm_intensity(uint8_t duty_cycle) {
    if (duty_cycle <= 100) {
        GPIOC->BSRR = LED_PIN; // Set pin high
        delay_us(duty_cycle * 10); // Adjust the delay based on duty cycle
        GPIOC->BRR = LED_PIN; // Set pin low
        delay_us((100 - duty_cycle) * 10); // Adjust the delay based on duty cycle
    }
}
```

```
int main() {
    // Initialize GPIO
    RCC->APB2ENR |= RCC_APB2ENR_IOPCEN; // Enable GPIOC clock
    GPIOC->CRH |= GPIO_CRH_MODE13; // Output mode, max speed 10 MHz

    // PWM loop
    while(1) {
        for (uint8_t i = 0; i <= 100; ++i) {
            set_pwm_intensity(i); // Increase intensity
        }
        for (uint8_t i = 100; i > 0; --i) {
            set_pwm_intensity(i); // Decrease intensity
        }
    }
}
```

Hardware PWM: Some Cortex-M3 microcontrollers have hardware support for PWM, often through specialized timers. Here's how you can configure hardware PWM using a timer:

Program:

```
#include "stm32f10x.h"

void Timer3_PWM_Init(void) {
    // Enable Timer 3 clock
    RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;

    // Configure Timer 3 for PWM mode
    TIM3->CCMR1 = TIM_CCMR1_OC1M_1 | TIM_CCMR1_OC1M_2; // PWM mode 1
    TIM3->CCMR1 |= TIM_CCMR1_OC1PE; // Output Compare 1 preload enable
    TIM3->CCER |= TIM_CCER_CC1E; // Enable output for channel 1
    TIM3->ARR = 99; // PWM period
    TIM3->CR1 |= TIM_CR1_CEN; // Enable Timer 3
```

```
}

void set_pwm_intensity_hw(uint8_t duty_cycle) {
   TIM3->CCR1 = (duty_cycle * TIM3->ARR) / 100; // Set duty cycle
}


int main() {
   // Initialize Timer 3 for PWM
   Timer3_PWM_Init();

   // PWM loop
   while(1) {
      for (uint8_t i = 0; i <= 100; ++i) {
         set_pwm_intensity_hw(i); // Increase intensity
         delay_ms(10); // Delay for a short time
      }
      for (uint8_t i = 100; i > 0; --i) {
         set_pwm_intensity_hw(i); // Decrease intensity
         delay_ms(10); // Delay for a short time
      }
   }
}
```

In both cases, adjust the GPIO pin and PWM frequency (if needed) according to your hardware configuration. Also, ensure the **delay_ms** function is properly implemented or replaced with an appropriate delay function.

RESULT:

Control intensity of an LED using PWM implementing in software and hardware is verified.

Control an LED using switch by polling method, by interrupt method and flash the LED once every five switch presses.	EXPT. NO: 4
	DATE:

AIM: To Control an LED using switch by polling method, by interrupt method and flash the ` LED once every five switch presses.

SOFTWARE: CORTEX-M3 development boards and using GNU tool chain.

Program:

To control an LED using a switch and flash the LED once every five switch presses using a CORTEX-M3 microcontroller, you can implement both polling and interrupt-driven methods. Below is an example code in

C using CMSIS (Cortex Microcontroller Software Interface Standard) for ARM Cortex-M devices. This example assumes that you are using a development board with a Cortex-M3 microcontroller, such as the STM32

series.

```
#include "stm32f10x.h" // Include the appropriate header file for your microcontroller

#define LED_PIN       GPIO_Pin_13
#define LED_PORT      GPIOC
#define SWITCH_PIN    GPIO_Pin_0
#define SWITCH_PORT   GPIOA

volatile int switch_press_count = 0;

void GPIO_Configuration(void) {
   GPIO_InitTypeDef GPIO_InitStructure;

   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOC, ENABLE);
   // Configure LED pin as push-pull output
   GPIO_InitStructure.GPIO_Pin = LED_PIN;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
```

```
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(LED_PORT, &GPIO_InitStructure);

    // Configure switch pin as input with pull-up
    GPIO_InitStructure.GPIO_Pin = SWITCH_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init(SWITCH_PORT, &GPIO_InitStructure);
}

void EXTI_Configuration(void) {
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    // Enable AFIO clock
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);

    // Configure EXTI line for the switch
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);

    EXTI_InitStructure.EXTI_Line = EXTI_Line0;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);

    // Enable and set EXTI0 Interrupt to the lowest priority
    NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x0F;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

```
void delay(int n) {
    int i;
    for (i = 0; i < n; i++);
}

void EXTI0_IRQHandler(void) {
    if (EXTI_GetITStatus(EXTI_Line0) != RESET) {
        EXTI_ClearITPendingBit(EXTI_Line0);
        switch_press_count++;
    }
}

int main(void) {
    GPIO_Configuration();
    EXTI_Configuration();

    while (1) {
        // Polling method
        if (GPIO_ReadInputDataBit(SWITCH_PORT, SWITCH_PIN) == RESET) {
            delay(10000); // Debounce
            if (GPIO_ReadInputDataBit(SWITCH_PORT, SWITCH_PIN) == RESET) {
                switch_press_count++;
                while (GPIO_ReadInputDataBit(SWITCH_PORT, SWITCH_PIN) == RESET);
            }
        }
    // Flash LED once every five switch presses
        if (switch_press_count % 5 == 0) {
            GPIO_SetBits(LED_PORT, LED_PIN);
            delay(1000000);
            GPIO_ResetBits(LED_PORT, LED_PIN);
```

```
        }
    }
}
```

This code sets up an LED connected to pin 13 of port C and a switch connected to pin 0 of port A. The switch press count is incremented both by polling and interrupt-driven methods. The LED flashes once every five switch presses. Make sure to adjust the delay functions to suit your microcontroller's clock speed.

Additionally, you may need to modify the code slightly to fit your specific microcontroller and development board.

RESULT: Control an LED using switch by polling method, by interrupt method and flash the LED once every five switch presses is verified.

UART Echo Test.	EXPT. NO: 5
	DATE:

AIM: To perform UART Echo Test.

SOFTWARE: CORTEX-M3 development boards and using GNU tool chain.

PROGRAM:

To perform a UART Echo Test using a Cortex-M3 microcontroller, you'll need to follow these general steps:

Initialize UART: Configure the UART module of your Cortex-M3 microcontroller. This involves setting up the baud rate, data format (number of data bits, parity, and stop bits), and enabling the UART peripheral.

Setup GPIO Pins: Configure GPIO pins for UART communication. You'll need to assign specific pins for UART transmit (TX) and receive (RX) lines.

Interrupt or Polling: Decide whether you want to use interrupt-driven or polling-based UART communication. Interrupts are generally more efficient as they allow the CPU to perform other tasks while waiting for UART events.

Receive Data: Implement code to receive data from the UART receive buffer. This can be done either by polling the UART status register for received data or by handling UART receive interrupts.

Echo Data: Once data is received, simply send it back by writing it to the UART transmit buffer.

Repeat: Continuously loop through the receive and echo process to create a UART echo functionality.

Here's a very basic example in pseudo code:

```
// Initialize UART
UART_Init();

while (1) {
   // Receive data
   if (UART_DataAvailable()) {
      char received_char = UART_Read();
```

```
// Echo data
        UART_Write(received_char);
    }
}
```

This pseudo code assumes that you have functions like **UART_Init()**, **UART_DataAvailable()**, **UART_Read()**, and **UART_Write()** implemented according to your microcontroller's specific UART peripheral and GPIO configurations.

Remember to consult your microcontroller's datasheet and reference manual for detailed information on UART initialization, GPIO configuration, and interrupt handling. Additionally, consider any specific requirements or constraints of your development environment or platform.

RESULT: UART Echo Test is verified.

Take analog readings on rotation of rotatory potentiometer connected to an ADC channel.	EXPT. NO: 6 DATE:

AIM: To Take analog readings on rotation of rotatory potentiometer connected to an ADC channel.

SOFTWARE: CORTEX-M3 development boards and using GNU tool chain.

PROGRAM:

To read analog values from a potentiometer connected to an ADC channel using a Cortex-M3 microcontroller, you can follow these general steps:
Initialize ADC: Configure the ADC module of your Cortex-M3 microcontroller. This typically involves setting up clock sources, configuring resolution, and selecting the input channel.
Configure GPIO: Set up the GPIO pins to which the potentiometer is connected. Configure them as analog inputs.
Start Conversion: Start the ADC conversion process. This usually involves triggering the ADC to begin converting analog signals.
Read Data: Wait for the conversion to complete and then read the converted data from the ADC data registers.
Scale and Process Data: If necessary, scale the raw ADC value to match the voltage range of your potentiometer and process it as needed.

Here's a basic example code using CMSIS (Cortex Microcontroller Software Interface Standard) for a Cortex-M3 microcontroller:

```
#include "stm32f10x.h" // Include the appropriate header for your microcontroller

void ADC_Configuration(void) {
  ADC_InitTypeDef ADC_InitStructure;
  GPIO_InitTypeDef GPIO_InitStructure;

  // Enable ADC1 and GPIOC clock
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_GPIOC, ENABLE);

  // Configure ADC1 Channel11 pin as analog input
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
  GPIO_Init(GPIOC, &GPIO_InitStructure);
```

```
// ADC1 configuration
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfChannel = 1;
    ADC_Init(ADC1, &ADC_InitStructure);

    // Enable ADC1
    ADC_Cmd(ADC1, ENABLE);

    // ADC1 regular channel11 configuration
    ADC_RegularChannelConfig(ADC1, ADC_Channel_11, 1, ADC_SampleTime_55Cycles5);
}

uint16_t ADC_Read(void) {
    // Start ADC1 Software Conversion
    ADC_SoftwareStartConvCmd(ADC1, ENABLE);

    // Wait until conversion completion
    while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);

    // Get the conversion value
    return ADC_GetConversionValue(ADC1);
}

int main(void) {
    uint16_t adc_value;

    ADC_Configuration();

    while(1) {
        adc_value = ADC_Read();
        // Process adc_value as needed
    }
}
```

Note that this example assumes you're using an STM32 microcontroller and the STM32 Standard Peripheral Library. Make sure to adapt the code according to your specific microcontroller and development environment. Additionally, consult your microcontroller's datasheet and reference manual for detailed information on ADC configuration and usage.

RESULT: Take analog readings on rotation of rotatory potentiometer connected to an ADC channel is verified.

Temperature indication on an RGB LED.	EXPT. NO: 7
	DATE:

AIM: To indicate temperature on an RGB LED.

SOFTWARE: CORTEX-M3 development boards and using GNU tool chain.

PROGRAM:

To indicate temperature on an RGB LED using a Cortex-M3 microcontroller, you'll need to follow these general steps:

Connect the RGB LED: Wire the RGB LED to the GPIO pins of the Cortex-M3 microcontroller. Typically, each color (Red, Green, Blue) will be connected to a separate pin.

Read Temperature: Interface a temperature sensor with the Cortex-M3 to read the current temperature. Common temperature sensors include LM35, DS18B20, or TMP36.

Convert Temperature to RGB Values: Define a mapping between temperature ranges and corresponding RGB color values. For example, cooler temperatures might correspond to blue, moderate temperatures to green, and warmer temperatures to red.

Update LED Color: Based on the temperature reading, calculate the appropriate RGB values and update the GPIO pins connected to the RGB LED accordingly.

Repeat: Continuously read the temperature sensor and update the LED color accordingly in a loop.

```
while True:
    temperature = read_temperature_sensor()  # Read temperature from sensor
    rgb_values = map_temperature_to_rgb(temperature)  # Map temperature to RGB values

    # Update LED color
    set_gpio_pin(red_pin, rgb_values[0])
    set_gpio_pin(green_pin, rgb_values[1])
    set_gpio_pin(blue_pin, rgb_values[2])

    delay(1000)  # Delay for 1 second before reading temperature again
```

In actual implementation, you'll need to replace the placeholder functions (**read_temperature_sensor()**, **map_temperature_to_rgb()**, **set_gpio_pin()**, **delay()**) with the appropriate functions provided by your microcontroller's SDK or libraries.

Ensure that you configure the GPIO pins for output and properly handle any scaling of the temperature values to fit within the range of the RGB LED color values (usually 0-255 for each color).

RESULT: Temperature indication on an RGB LED is verified.

Mimic light intensity sensed by the light sensor by varying the blinking rate of an LED.	EXPT. NO: 8
	DATE:

AIM: To Mimic light intensity sensed by the light sensor by varying the blinking rate of an LED.

SOFTWARE: CORTEX-M3 development boards and using GNU tool chain.

PROGRAM:

To mimic light intensity sensed by the light sensor by varying the blinking rate of an LED using a CORTEX-M3 microcontroller, you can follow these steps:

Initialize peripherals: Set up the GPIO pins for the LED and the light sensor, and any necessary peripherals (such as ADC for analog input from the light sensor).
Read light sensor input: Continuously read the input from the light sensor to determine the current light intensity. If you're using an analog light sensor, you'd use the ADC to convert the analog signal to a digital value.
Calculate LED blinking rate: Map the light intensity value obtained from the sensor to an appropriate blinking rate for the LED. For example, higher light intensity might correspond to a faster blinking rate, and lower light intensity might correspond to a slower blinking rate.
Control LED blinking: Adjust the blinking rate of the LED based on the calculated value. You can achieve this by toggling the GPIO pin connected to the LED at the desired rate.

```
#include "stm32f10x.h" // Include the appropriate header file for your Cortex-M3 MCU

// Function to initialize GPIO pins
void GPIO_Init(void) {
    // Enable clock for GPIO port connected to LED
    RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;
    // Configure PC13 as output push-pull (LED)
    GPIOC->CRH &= ~(GPIO_CRH_CNF13 | GPIO_CRH_MODE13);
    GPIOC->CRH |= GPIO_CRH_MODE13_0;
}

// Function to initialize ADC
void ADC_Init(void) {
    // Enable clock for ADC1
    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;




    // ADC configuration
```

```
    ADC1->CR2 |= ADC_CR2_ADON; // Turn on ADC
    ADC1->CR2 |= ADC_CR2_CAL;  // Start calibration
    while (ADC1->CR2 & ADC_CR2_CAL); // Wait for calibration to finish
}

// Function to read ADC value from light sensor
uint16_t ADC_Read(void) {
    ADC1->SQR3 = 0; // Select channel 0 for ADC conversion (change this according to your setup)
    ADC1->CR2 |= ADC_CR2_ADON; // Start conversion
    while (!(ADC1->SR & ADC_SR_EOC)); // Wait for conversion to finish
    return ADC1->DR; // Return converted value
}

int main(void) {
    GPIO_Init(); // Initialize GPIO for LED
    ADC_Init(); // Initialize ADC for light sensor

    uint16_t light_intensity;

    while (1) {
        // Read light intensity from sensor
        light_intensity = ADC_Read();

        // Map light intensity to LED blinking rate
        // Example mapping: higher intensity, faster blinking
        uint32_t delay = 5000 / light_intensity; // Adjust this formula based on your requirements

        // Toggle LED at calculated blinking rate
        GPIOC->ODR ^= GPIO_ODR_ODR13; // Toggle LED pin state
        for (volatile uint32_t i = 0; i < delay; i++); // Delay
    }
}
```

This code is just a basic example. You may need to adapt it to fit your specific hardware setup and requirements. Make sure to consult the datasheets and reference manuals of your microcontroller and peripherals for accurate configuration and usage details.

RESULT: Mimic light intensity sensed by the light sensor by varying the blinking rate of an LED is verified.

Evaluate the various sleep modes by putting core in sleep and deep sleep modes.	EXPT. NO: 9
	DATE:

AIM: To evaluate the various sleep modes by putting core in sleep and deep sleep modes.

SOFTWARE: CORTEX-M3 development boards and using GNU tool chain.

PROGRAM:

Cortex-M3 core into sleep and deep sleep modes using ARM Cortex-M CMSIS (Cortex Microcontroller Software Interface Standard) and CMSIS-Core library.

```
#include "stm32f10x.h"  // Include the CMSIS-Core header file for your specific Cortex-M3 MCU

int main(void) {
   // Initialize your peripherals and system configuration here

   while (1) {
      // Your application code

      // Enter Sleep Mode
      __WFI();  // Wait For Interrupt instruction

      // After waking up from sleep mode, continue executing from here

      // Enter Deep Sleep Mode
      SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;  // Set SLEEPDEEP bit in System Control Register
      __WFI();  // Wait For Interrupt instruction

      // After waking up from deep sleep mode, continue executing from here
   }
}
```

__WFI() is a CMSIS intrinsic function that puts the processor into sleep mode and waits for the next interrupt to wake it up.

SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk; sets the SLEEPDEEP bit in the System Control Register (SCR). This bit determines whether the processor enters sleep or deep sleep mode when the WFI instruction is executed. Setting this bit enables deep sleep mode.

After setting the SLEEPDEEP bit, calling **__WFI()** puts the processor into deep sleep mode, waiting for an interrupt to wake it up.

Make sure to replace **stm32f10x.h** with the appropriate header file for your specific Cortex-M3 microcontroller, and configure your project settings accordingly.

Keep in mind that when the processor wakes up from sleep or deep sleep mode, it resumes execution from the next instruction after the **__WFI()** call. So, any necessary setup or initialization code should be executed before the **__WFI()** call to ensure proper operation after waking up.

RESULT: Evaluate the various sleep modes by putting core in sleep and deep sleep modes is verified.

System reset using watchdog timer in case something goes wrong.	EXPT. NO: 10
	DATE:

AIM: To reset system using watchdog timer in case something goes wrong.

SOFTWARE: CORTEX-M3 development boards and using GNU tool chain.

PROGRAM:
In a Cortex-M3-based system, the watchdog timer (WDT) can be utilized as a safety mechanism to reset the system in case of malfunctions or faults. Here's a basic outline of how you might implement this:

Initialization: Configure the watchdog timer during system initialization. This involves setting the timeout period, enabling the watchdog, and configuring any other necessary settings.

Watchdog Feed: Regularly "feed" or "kick" the watchdog timer from your main application loop or critical sections of your code. This involves writing to the watchdog timer's register to reset its countdown timer. If this is not done within the specified timeout period, the watchdog will trigger a reset.

Error Handling: Implement error detection mechanisms in your code. If a critical error or fault is detected, such as a system hang or unexpected behavior, intentionally stop feeding the watchdog timer to trigger a reset.

Reset Handling: Upon reset, your system should go through its initialization routine again to ensure a clean startup.

```
// Initialize watchdog timer with a timeout period
void init_watchdog() {
   // Configure watchdog timeout period
   // Enable watchdog
}

// Feed the watchdog timer to prevent reset
void feed_watchdog() {
   // Write to watchdog timer register to reset countdown
}

// Main application loop
int main() {
   // Initialize system
   init_watchdog();
```

```
while (1) {
      // Main application code

      // Feed the watchdog timer regularly
      feed_watchdog();

      // Check for errors and handle them appropriately
      if (error_detected) {
         // Stop feeding the watchdog to trigger reset
         while (1) {
            // Wait for reset to occur
         }
      }
   }
   return 0;
}
```

This approach ensures that your system will reset itself if it gets stuck or encounters unexpected behavior, providing a safety mechanism to prevent prolonged downtime or system failure.

RESULT : **System reset using watchdog timer in case something goes wrong is verified.**

Sample sound using a microphone and display sound levels on LEDs.	EXPT. NO: 11
	DATE:

AIM: To produce sample sound using a microphone and display sound levels on LEDs.

SOFTWARE: CORTEX-M3 development boards and using GNU tool chain.

PROGRAM:

To sample sound using a microphone and display sound levels on LEDs with a Cortex-M3 microcontroller, you'll typically need to interface with an analog-to-digital converter (ADC) to convert the microphone's analog output into digital values that you can process. Then, based on the sampled sound levels, you can control LEDs to display the sound intensity.

Below is a basic example written for STM32 microcontrollers using the HAL (Hardware Abstraction Layer) library:

```
#include "stm32f1xx_hal.h"

#define NUM_LEDS 8

ADC_HandleTypeDef hadc1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_ADC1_Init();

  HAL_ADC_Start(&hadc1);

  uint16_t adc_value;
  uint16_t adc_threshold = 2000; // Adjust this threshold according to your setup
  uint32_t led_mask = 0x01;

  while (1)
  {
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
    adc_value = HAL_ADC_GetValue(&hadc1);
```

```
// Display sound level using LEDs
    for (int i = 0; i < NUM_LEDS; i++)
    {
      if (adc_value > adc_threshold * (i + 1))
      {
        HAL_GPIO_WritePin(GPIOB, led_mask << i, GPIO_PIN_SET);
      }
      else
      {
        HAL_GPIO_WritePin(GPIOB, led_mask << i, GPIO_PIN_RESET);
      }
    }
  }
}

void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI_DIV2;
  RCC_OscInitStruct.PLL.PLLMul = RCC_PLL_MUL9;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK
                              | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
  {
    Error_Handler();
  }
}

static void MX_ADC1_Init(void)
{
  ADC_ChannelConfTypeDef sConfig = {0};
```

```
 hadc1.Instance = ADC1;
  hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
  hadc1.Init.ContinuousConvMode = ENABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }
  sConfig.Channel = ADC_CHANNEL_0; // Adjust this channel according to your setup
  sConfig.Rank = ADC_REGULAR_RANK_1;
  sConfig.SamplingTime = ADC_SAMPLETIME_239CYCLES_5;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
}

static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  __HAL_RCC_GPIOB_CLK_ENABLE();

  GPIO_InitStruct.Pin = GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_10 | GPIO_PIN_11 |
                        GPIO_PIN_12 | GPIO_PIN_13 | GPIO_PIN_14 | GPIO_PIN_15;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}

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

This code configures the ADC to sample an analog signal from a microphone connected to a specific channel. It then reads the ADC value in a loop and adjusts the state of LEDs based on the sampled sound level. You may need to adjust the ADC channel, GPIO pins, ADC sampling time, and LED behavior according to your hardware setup and requirements. Additionally, you might need to apply signal conditioning techniques such as filtering or amplification to improve the accuracy of sound level detection.

RESULT: Sample sound using a microphone and display sound levels on LEDs is verifie