INTERNSHIP REPORT

ON

EMBEDDED SYSTEMS

Submitted in partial fulfillment of the requirements

for the award of Degree of

BACHELOR OF TECHNOLOGY IN

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted By

MANGALI MADHU SUDHAN – 21MG1A04A5

Under the esteemed guidance of

Mrs. M. SREEDEVI, M.Tech.
Sr. Assistant Professor
Internship Coordinator

Department of Electronics and Communication Engineering

SREE VAHINI INSTITUTE OF SCIENCE AND TECHNOLOGY
TIRUVURU, NTR Dist. 521235

Affiliated to JNTU, Kakinada

(2025)

CERTIFICATE

This is to certify that the **Internship** work entitled “**EMBEDDED SYSTEMS”** done by **MANGALI MADHU SUDHAN (21MG1A04A5)** department of Electronics and Communication Engineering, is a record bonafide work carried out by him. This **Internship** is done as a partial fulfillment of obtaining Bachelor of Technology Degree to be awarded by JNTU, Kakinada.

The matter embodied in this **Internship** report has not been submitted to any other university for the award of any other degree.

Mrs. M. SREEDEVI	**Dr. R. SRIDEVI**
Sr. Assistant professor	**Professor& HOD**
Internship Coordinator	

External Examiner

ACKNOWLEDGEMENT

Great acknowledgement is expressed to SVIST Management members whose guidance is invaluable to complete this **Internship** in time.

I take this opportunity to place on record my heartiest thanks to **Dr. R. NAGENDRA BABU**, **Principal of our college**, for his guidance and co-operation during our course of study.

I express my deep sense of gratitude to **Dr. R. SRIDEVI,** **Professor& HOD** for her motivation and inspiration provided during the **Internship** work.

I express my sincere gratitude towards **Internship** Coordinator, **Mrs. M. SREEDEVI,** **Sr.Assistant Professor**, for her advice, support and guidance throughout this **Internship.**

Finally, I would like to acknowledge my deep sense of gratitude to all well-wishers and friends who helped us directly or indirectly to complete this work.

MANGALI MADHU SUDHAN - 21MG1A04A5

DECLARATION

I here by declare that the work presented in this **Internship** titled **“EMBEDDED SYSTEM”** is submitted towards completion of B.Tech **Internship** in **ELECTRONIC & COMMUNICATION ENGINEERUNG** at **SREE VAHINI INSTITUTE OF SCIENCE AND TECHNOLOGY**, Tiruvuru,NTR Dist. Its is an authentic record of my original work pursed under the guidance of **Mrs.M. SREEDEVI, Sr. Assistant Professor.**

I have not submitted the matter embodied in this **Internship** for the award of any other degree.

MANGALI MADHU SUDHAN - 21MG1A04A5

CONTENTS

List of Figures:

CHAPTER-1:

INTRODUCTION

1.1 EMBEDDED SYSTEMS

Embedded systems are specialized computing systems designed to perform dedicated functions within larger mechanical or electrical systems. Unlike general-purpose computers, they are optimized for specific tasks, often with real-time constraints, and are embedded as part of a complete device. These systems typically combine hardware (microcontrollers/microprocessors, sensors, actuators) and software (firmware) to control physical processes, from simple home appliances like microwave ovens to complex industrial machines and automotive systems. Their efficiency, reliability, and low power consumption make them indispensable in modern technology.

A key characteristic of embedded systems is their real-time operation, where tasks must be completed within strict time deadlines. They often use microcontrollers (e.g., 8051, ARM Cortex-M, PIC) or microprocessors (e.g., Raspberry Pi) as their core processing units, depending on the complexity of the application. These systems interact with the physical world through peripherals like ADCs (Analog-to-Digital Converters), PWM (Pulse Width Modulation) modules, and communication interfaces (UART, SPI, I2C). For example, a smart thermostat uses temperature sensors (input) and relays (output) to maintain room temperature, all controlled by embedded firmware.

The applications of embedded systems span nearly every industry, including consumer electronics (smartphones, wearables), automotive (engine control, ADAS), healthcare (medical devices), and industrial automation (PLC, robotics). With the rise of the Internet of Things (IoT), embedded systems now often include wireless connectivity (Wi-Fi, Bluetooth, LoRa) for remote monitoring and control. As technology advances, embedded systems are becoming more powerful, energy-efficient, and secure, driving innovation in areas like edge computing, AI at the edge, and autonomous systems. Their ability to merge hardware and software for tailored solutions ensures they remain at the heart of modern technological progress.

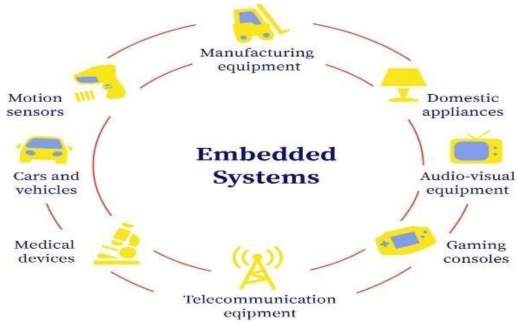


Fig1: Introduction to Embedded Systems

History of Embedded Systems

The origins of embedded systems date back to the 1940s–1960s, when early computers were used for specialized military and aerospace applications. One of the first recognizable embedded systems was

NASA's Apollo Guidance Computer (AGC), developed in the 1960s to control spacecraft navigation during the

Apollo missions. These early systems were primitive by today's standards, relying on bulky hardware and lowlevel programming, but they laid the foundation for modern embedded computing.

The 1970s–1990s marked a turning point with the invention of microprocessors and microcontrollers. The Intel 4004 (1971) and later the Intel 8051 (1980) enabled compact, cost-effective embedded solutions, revolutionizing industries like automotive, consumer electronics, and industrial automation. During this period, real-time operating systems (RTOS) such as VxWorks emerged, allowing more complex and reliable embedded applications. By the 1990s, embedded systems were everywhere—from microwave ovens to anti-lock braking systems in cars—thanks to advancements in semiconductor technology.

From the 2000s onward, embedded systems evolved rapidly with the rise of System-on-Chip (SoC) designs, Linux-based embedded platforms (e.g., Raspberry Pi), and the Internet of Things (IoT). Today, modern embedded systems integrate AI, wireless connectivity, and energy-efficient processing, powering innovations like smart homes, autonomous vehicles, and wearable health monitors. As technology progresses, embedded systems continue to push the boundaries of what’s possible, blending hardware and software to create smarter, more connected devices.

Generations of Embedded Systems

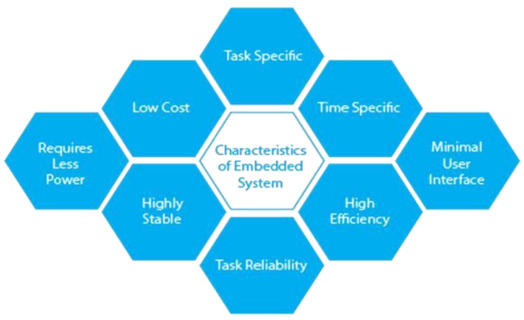
The evolution of embedded systems can be categorized into distinct generations, each marked by technological breakthroughs and expanding applications. The first generation (1970s–1980s) was defined by the introduction of microprocessors like the Intel 4004 and 8-bit microcontrollers such as the Intel 8051. These early systems were rudimentary, with limited processing power and memory, but they enabled basic automation in industrial machines, calculators, and early automotive controls. Programming was done primarily in assembly language, and real-time performance was a key focus for applications like traffic light controllers and digital watches.

The second generation (1990s–early 2000s) saw the rise of 16-bit and 32-bit microcontrollers (e.g., ARM7, PIC18) and the adoption of Real-Time Operating Systems (RTOS) like VxWorks and QNX. This era brought enhanced processing capabilities, allowing embedded systems to handle more complex tasks in telecommunications, automotive ECUs, and consumer electronics. C became the dominant programming language, replacing assembly for most applications. The introduction of flash memory also improved firmware updates, making systems more adaptable and easier to maintain.

In the third generation (mid-2000s–2010s), embedded systems became more interconnected and powerful, driven by System-on-Chip (SoC) designs and Linux-based platforms (e.g., Raspberry Pi). Wireless communication (Wi-Fi, Bluetooth, Zigbee) enabled IoT applications, transforming industries with smart devices, wearables, and home automation. This period also saw the integration of graphical user interfaces

(GUIs) and touchscreens, expanding usability in medical devices, infotainment systems, and industrial HMIs. Energy efficiency and miniaturization became critical as battery-powered and portable devices grew in popularity.

Today, we are in the fourth generation (2020s and beyond), where embedded systems are merging with AI, edge



computing, and 5G connectivity. Modern microcontrollers (e.g., ESP32, STM32H7) support machine learning at the edge, enabling real-time decision-making in autonomous robots, smart cameras, and predictive maintenance systems. RISC-V architecture is gaining traction as an open-source alternative, while security and low-power design remain top priorities. The future points toward quantum computing integration, self-healing systems, and even more seamless human-machine interactions, ensuring embedded technology continues to drive innovation across all sectors.

1.2 CHARACTERICSTICS OF EMBEDDED SYSTEMS

Embedded systems are designed to perform a specific task, in contrast with general-purpose computers designed for multiple tasks. Some have real-time performance constraints that must be met, for reasons such as safety and usability; others may have low or no performance requirements, allowing the system hardware to be simplified to reduce costs.

Embedded systems are not always standalone devices. Many embedded systems are a small part within a larger device that serves a more general purpose. For example, the Gibson Robot Guitar features an embedded system for tuning the strings, but the overall purpose of the Robot Guitar is to play music. Similarly, an embedded system in an automobile provides a specific function as a subsystem of the car itself.

The program instructions written for embedded systems are referred to as firmware, and are stored in read-only memory or flash memory chips. They run with limited computer hardware resources: little memory, small or non-existent keyboard or screen.

Here are some of the important characteristics of embedded systems:

Fig 2: Characteristics Of Embedded Systems

- **Performs specific tasks:** Embedded systems are designed to perform specific tasks or functions. They are optimized for the particular task they are intended to perform, which makes them more efficient and reliable.

- **Low Cost:** Embedded systems are typically designed to be cost-effective. This is because they are often used in large volumes, and the cost per unit must be low to make the product economically viable.
- **Time Specific:** Embedded systems must operate within a specific time frame. This is important in applications such as industrial control systems, where timing is critical for safety and efficiency.

- **Low Power:** Embedded systems are designed to operate with minimal power consumption. This is important for applications where the system needs to operate for extended periods on battery power or where power consumption needs to be minimized to reduce operating costs.

- **Highly Stable:** Embedded systems are typically designed to be stable and reliable. They are often used in applications where failure is not an option, such as in medical devices or aviation.
- **High Reliability:** Embedded systems are designed to operate reliably and consistently over long periods.

This is important in applications where downtime can be costly or dangerous.

Architecture of Embedded System

Here are some common components of an embedded systems architecture:

- **Processor**: The processor is the heart of the embedded system and is responsible for executing instructions and controlling the system's functions. The processor can be a microcontroller, microprocessor, or digital signal processor (DSP).
- **Memory**: Embedded systems require memory to store program code and data. There are two types of memory used in embedded systems: RAM (Random Access Memory) and ROM (Read-Only Memory). RAM is used to store temporary data, while ROM is used to store permanent data and program code.
- **Input/Output (I/O) Interfaces**: Embedded systems require interfaces to interact with the external environment. These interfaces can be digital, analog, or both. Digital interfaces are used to interact with devices that use binary signals (0 or 1) such as switches or sensors, while analog interfaces are used to interact with devices that use continuous signals such as temperature sensors.
- **Communication Interfaces:** Embedded systems may require communication interfaces to exchange data with other devices. These interfaces can be wired or wireless, such as Ethernet, USB, SPI, I2C, Bluetooth, or Wi-Fi.
- **Power Management:** Embedded systems may require power management techniques to minimize power consumption and extend battery life. These techniques can include sleep modes, power gating, or dynamic voltage scaling.
- **Operating System:** An operating system can be used in embedded systems to provide a user interface, manage resources, and control system functions. Common operating systems used in embedded systems include Linux, Windows CE, and Free RTOS.

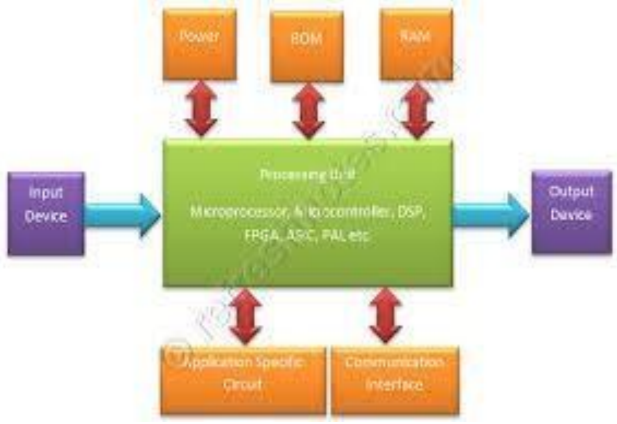

Power
ROM
RAM
Input Device
Processing Unit
Microprocessor, Microcontroller, DSP, FPGA, ASIC, PAL etc
Output Device
Application Specific Circuit
Communication Interface

Fig 3: Architecture of Embedded System

Types of Embedded Systems

- **Microcontroller-based Embedded Systems**: These are embedded systems that use a microcontroller as the main processing unit. They are commonly used in small-scale applications such as household appliances, toys, and automotive systems.
- **Real-time Embedded Systems**: These are embedded systems that are designed to respond to external events or input signals within a specified time frame. They are commonly used in critical applications such as aerospace, defense, and medical devices.
- **Networked Embedded Systems**: These are embedded systems that are connected to a network and can communicate with other devices. They are commonly used in applications such as home automation, building automation, and industrial control systems.
- **Mobile Embedded Systems:** These are embedded systems that are designed for use in mobile devices such as smartphones, tablets, and portable gaming devices. They are optimized for low power consumption and high performance.
- **Programmable Logic Controller (PLC) Systems:** These are embedded systems that are commonly used in industrial automation applications to control machinery and processes.
- **Digital Signal Processing (DSP) Systems:** These are embedded systems that are optimized for processing signals such as audio and video. They are commonly used in applications such as telecommunications, multimedia, and image processing.

Advantages and Disadvantages of Embedded Systems

- **Improved System Performance:** Embedded systems are designed to perform specific tasks and are optimized for efficiency and performance. They can deliver faster response times and higher accuracy compared to general-purpose computing systems.
- **Lower Power Consumption:** Embedded systems are optimized for low power consumption, making them ideal for battery-powered devices or systems that need to operate in remote or inaccessible locations.

- **Increased Reliability:** Embedded systems are designed to be reliable and stable, with minimal downtime or errors. They can operate in harsh environments and withstand temperature, humidity, and other external factors.

- **Cost-Effective:** Embedded systems are often less expensive compared to general-purpose computing systems. They require fewer hardware components and are optimized for specific tasks, reducing the overall system cost.
- **Compact Size:** Embedded systems are designed to be small and compact, making them ideal for applications where space is limited, such as in cars, aircraft, and medical devices.
- **Customizable:** Embedded systems can be customized to meet specific application requirements, allowing for greater flexibility and functionality.
- **Improved Security:** Embedded systems can be designed with built-in security features, such as encryption and authentication, to protect against cyber attacks and unauthorized access.
- **Limited Functionality:** Embedded systems are designed to perform specific tasks and are often limited in their functionality. They may not be suitable for applications that require more complex or varied tasks.
- **Limited Upgradability:** Embedded systems are often designed with limited upgradability, which can be a disadvantage in applications where future upgrades or modifications may be required.
- **Limited Connectivity:** Some embedded systems may have limited connectivity options, which can limit their ability to communicate with other devices or systems.

1.3 REAL TIME EMBEDDED SYSTEMS

A real time embedded system is defined as, a system which gives a required o/p in a particular time. These types of embedded systems follow the time deadlines for completion of a task. Real time embedded systems are classified into two types such as soft and hard real time systems. Further this Real-Time Embedded System is divided into two type‘s i.e.

- **Soft Real Time Embedded Systems**
- **Hard Real-Time Embedded Systems**

In the simplest form, real-time systems can be defined as those systems that respond to external events in a timely fashion, the response time is guaranteed.

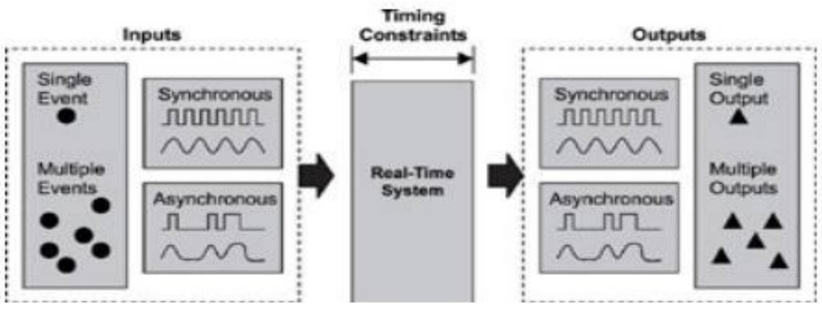


Fig4: Real Time Embedded Systems

A good way to understand the relationship between real-time systems and embedded systems is to view them as two intersecting circles. It can be seen that not all embedded systems exhibit real-time behaviors nor are all real-time systems embedded. However, the two systems are not mutually exclusive, and the area in which they overlap creates the combination of systems known as real-time embedded systems

Hard Real-Time Systems

A hard real-time system is a real-time system that must meet its deadlines with a near-zero degree of flexibility. The deadlines must be met, or catastrophes occur. The cost of such catastrophe is extremely high and can involve human lives. The computation results obtained after the deadline have either a zero-level of usefulness or have a high rate of depreciation as time moves further from the missed deadline before the system produces a response.

Soft Real-Time Systems

A soft real-time system is a real-time system that must meet its deadlines but with a degree of flexibility. The deadlines can contain varying levels of tolerance, average timing deadlines, and even statistical distribution of response times with different degrees of acceptability. In a soft realtime system, a missed deadline does not result in system failure, but costs can rise in proportion to the delay, depending on the application.

1.4 BARE METAL PROGRAMING

When developing embedded systems that are to be real-time capable, one of the first and most important questions is whether the applications should run under a real-time operating system (RTOS) or whether a baremetal solution should be developed. Bare-metal programming is generally understood to mean that an application is written directly on the hardware without using an external programming interface, i.e. an operating system. Applications access here directly hardware registers of microcontrollers. Here one helps oneself with approaches such as endless loops, which execute tasks with fixed computing time. This sequential execution is only deviated from when an interrupt event occurs. This bare-metal development approach for embedded systems is therefore also known as super-loop.

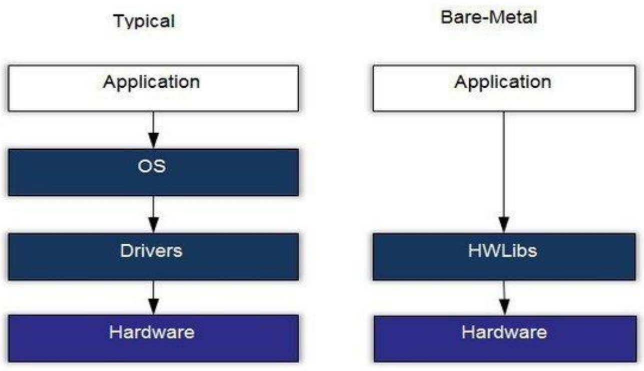


Fig5: Bare Metal Programing

Bare-metal, it is usually considered as a low-level method of programming that is specific to the hardware only and is often used for optimizing software and applications for an electronics hardware or the creation of basic tools which would be used on a new system to do things such as bypassing the BIOS or operating system interface etc.

Bare metal programming mostly involves two programming languages, C and Assembly. Although Assembly language is not used now, but developers often need to have knowledge of assembly language even if they code in C for a hardware.

Moat microcontrollers are programmed in bare metal as they have very little memory and are not capable of running an OS, most of them cannot run more than a single program.

The main advantage of coding embedded systems bare metal is its execution speed, simplicity, reduced power consumption and most importantly, price.

Benefits of Bare Metal Programming

1. Maximum Resource Utilization:

Bare Metal Programming allows for optimal utilization of system resources since it eliminates the overhead associated with operating systems. This leads to faster execution times and reduced memory footprints.

2. Predictable Performance:

With no operating system in the way, developers can achieve deterministic and predictable performance. This is crucial in applications where timing and responsiveness are paramount.

3. Reduced Complexity:

Bare Metal Programming simplifies the development process by eliminating the need to navigate through layers of operating system abstraction. This reduction in complexity streamlines the development cycle and facilitates faster time-to-market.

Drawbacks of Bare Metal Programming

1. Limited Abstraction:

While the direct interaction with hardware provides control, it also means dealing with the complexities of hardware-specific details. This lack of abstraction can make the development process more challenging.

2. Portability Challenges:

Bare Metal Programming may face challenges in portability, as the code is closely tied to the hardware architecture. Adapting the code to different platforms might require substantial modifications.

3. Lack of Standardization:

Unlike higher-level programming with standardized libraries and APIs, Bare Metal Programming lacks such standardization. This can make code less reusable and may lead to increased development effort.

1.5 INTRODUCTION TO MICROPROCESSOR & MICROCONTROLLER

A microprocessor is the central processing unit (CPU) of a computer, integrated onto a single semiconductor chip. It performs arithmetic, logic, and control operations, acting as the "brain" of electronic systems. Modern microprocessors (e.g., Intel Core, ARM Cortex) power devices ranging from PCs to smartphones, leveraging advancements like multi-core architectures and nanometer-scale fabrication. Unlike microcontrollers, microprocessors require external memory and peripherals to function, making them flexible but less selfcontained.

A microcomputer is a complete computing system built around a microprocessor, including memory (RAM/ROM), input/output interfaces, and storage. Early examples like the Altair 8800 (1975) and Apple I pioneered personal computing, while modern variants include embedded single-board computers (e.g., Raspberry Pi). Microcomputers balance processing power and compact size, serving as development platforms, industrial controllers, or educational tools. Their modular design allows customization through expansion buses (USB, PCIe).

The key difference lies in integration: microprocessors are standalone CPUs, whereas microcomputers are functional systems. Microcontrollers (e.g., Arduino's ATmega) further integrate CPU, memory, and peripherals on one chip for embedded applications. Together, these technologies drive the evolution of computing—from supercomputers to IoT devices—by scaling performance and efficiency.

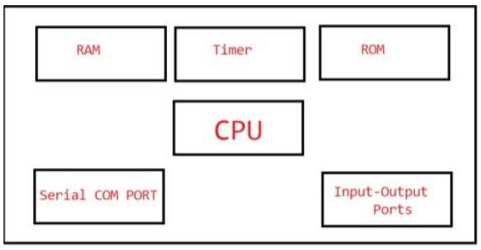


Fig6: Basic Architecture of MicroController system

MICROPROCESSOR ARCHITECTURE AND ITS OPERATION

Microprocessor architecture refers to the design and organization of the components within a microprocessor, which is the central processing unit (CPU) of a computer. The architecture defines how the microprocessor processes data, communicates with other components, and executes instructions. Understanding microprocessor architecture is crucial for designing efficient computing systems and optimizing performance.

Components of Microprocessor Architecture

1. **Arithmetic Logic Unit (ALU)**:

• The ALU performs arithmetic operations (addition, subtraction, multiplication, division) and logical operations (AND, OR, NOT, etc.). It is a critical component for executing instructions.

2. **Control Unit (CU)**:

• The control unit orchestrates the operations of the microprocessor by directing the flow of data between the ALU, registers, and memory. It interprets instructions from the program and generates control signals to execute them.

3. **Registers**:

• Registers are small, fast storage locations within the CPU used to hold temporary data and instructions. Common types of registers include:

General-purpose registers:

Used for a variety of functions.

Special-purpose registers:

Such as the program counter (PC), which keeps track of the address of the next instruction to execute, and the stack pointer (SP), which points to the top of the stack in memory.

4. **Cache Memory**:

• Cache memory is a small, high-speed storage area located close to the CPU. It stores frequently accessed data and instructions to speed up processing by reducing the time it takes to access data from the main memory (RAM).

5. **Bus Interface**:

• The bus interface connects the microprocessor to other components of the computer, such as memory and input/output devices. It includes data buses, address buses, and control buses that facilitate communication.

6. **Instruction Set Architecture (ISA)**:

• The ISA defines the set of instructions that the microprocessor can execute, including the format of the instructions, addressing modes, and data types. It serves as the interface between software and hardware.

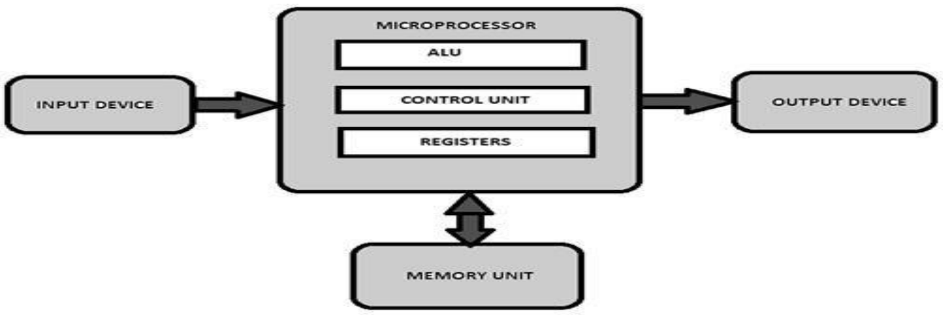

MICROPROCESSOR
ALU
CONTROL UNIT
REGISTERS
INPUT DEVICE
OUTPUT DEVICE
MEMORY UNIT

Fig 7: Basic Architecture of Microprocessor

Microprocessor Operation

The operation of a microprocessor can be broken down into several key stages:

1. **Fetch**:

• The control unit fetches the next instruction from memory using the program counter to determine the address. The instruction is then loaded into the instruction register.

2. **Decode**:

• The fetched instruction is decoded to determine what action is required. The control unit interprets the instruction and generates the necessary control signals.

3. **Execute**:

• The ALU performs the required operation based on the decoded instruction. This may involve arithmetic calculations, logical operations, or data movement.

4. **Memory Access**:

• If the instruction requires data from memory (e.g., loading data into a register), the microprocessor accesses the memory to read or write data.

5. **Write Back**:

• The results of the execution are written back to the appropriate registers or memory locations.

6. **Update Program Counter**:

• The program counter is updated to point to the next instruction, and the cycle repeats.

Types of Microprocessor Architectures

1. **CISC (Complex Instruction Set Computing)**:

• CISC architectures have a large set of instructions, allowing complex operations to be executed in a single instruction. Examples include x86 architecture.

2. **RISC (Reduced Instruction Set Computing)**:

• RISC architectures use a smaller set of simple instructions, which can be executed in a single clock cycle.

This design aims for higher performance through pipelining and parallelism. Examples include ARM and MIPS architectures.

3. **VLIW (Very Long Instruction Word)**:

• VLIW architectures allow multiple operations to be encoded in a single instruction word, enabling parallel execution of instructions.

4. **Superscalar**:

• Superscalar architectures can issue multiple instructions per clock cycle, allowing for greater instructionlevel parallelism.

1.6 TYPES OF PROCESSORS

An embedded processor is specifically designed to handle the needs of an embedded system. It is like a computer chip that has been embedded in various machines. Microprocessors and microcontrollers are important parts of many modern electronic devices. These parts are small chips called integrated circuits (ICs) used in things like computers, laptops, washing machines, air conditioners, and many other gadgets. Both microprocessors and microcontrollers help automate tasks, but their roles are different. In this blog, we highlight how many types of embedded microprocessors and the major difference between microprocessors and microcontrollers.

Different Types of Embedded Processors

There are different types of Embedded Processors, some of them are as follows:

1. General-Purpose Processors (GPPs)

General-purpose processors, like Intel Core and AMD Ryzen, are designed for versatility in PCs, servers, and workstations. They support complex operating systems (Windows, Linux) and handle multitasking efficiently. These processors excel in performance but consume more power than specialized chips, making them unsuitable for battery-operated embedded systems.

2. Microcontrollers (MCUs)

Microcontrollers integrate a CPU, memory, and peripherals (ADC, PWM, UART) on a single chip. Examples include **ARM Cortex-M** (STM32), **AVR** (Arduino), and **PIC**. They are optimized for low-power, real-time control in embedded systems like wearables, automotive ECUs, and smart appliances. Their simplicity and costeffectiveness make them ideal for mass-produced electronics.

3. Digital Signal Processors (DSPs)

DSPs (e.g., Texas Instruments TMS320) specialize in high-speed mathematical operations for signal processing.

They are used in audio processing (noise cancellation), telecommunications (modems), and image/video processing (medical imaging). DSPs feature **MAC units** (Multiply-Accumulate) for efficient FFT and filter algorithms.

4. Application-Specific Integrated Circuits (ASICs)

ASICs are custom-designed for a single application (e.g., Bitcoin mining, AI accelerators). They offer unmatched performance and power efficiency for their specific task but lack flexibility. High design costs make them viable only for large-scale production (e.g., smartphone SoCs like Apple's A-series).

5. Field-Programmable Gate Arrays (FPGAs)

FPGAs (e.g., Xilinx, Intel/Altera) are reconfigurable chips that emulate custom hardware logic. They bridge the gap between software (GPPs) and hardware (ASICs), used in prototyping, military systems, and 5G infrastructure. Their parallel processing suits real-time systems but requires HDL programming (VHDL/Verilog).

6. Graphics Processing Units (GPUs)

Originally for rendering graphics, GPUs (NVIDIA, AMD) now accelerate parallel tasks like AI (CUDA cores), scientific simulations, and cryptocurrency mining. They outperform CPUs in **SIMD (Single Instruction, Multiple Data)** workloads but consume significant power.

7. Quantum Processors

An emerging technology, quantum processors (IBM Q, Google Sycamore) use qubits for exponential speedup in cryptography, optimization, and drug discovery. While not yet mainstream, they promise breakthroughs in solving classically intractable problems.

Choosing The Right Microprocessor

- Microprocessors are like the older cousins of microcontrollers. While microcontrollers pack many essential features such as memory directly into the chip, microprocessors do not. Instead, they rely on external components for additional features, making them better suited for more complex computing tasks.
- When choosing a microprocessor, your decision will depend on what your embedded system needs to do. Here are some key questions to guide you:
- Will the system be battery-operated?
- Does the design need to be future-proof?

- What interface is needed?
- What peripherals are needed?
- What processing tasks will the system perform?

CHAPTER-2

8051 MICROCONTROLLER

2.1 ARCHITECTURE OF 8051

The memory is an important part of the 8051 Microcontroller Architecture (for that matter, any Microcontroller).

So, it is important for us to understand the 8051 Microcontroller Memory Organization i.e. how memory is organized, how the processor accesses each memory and how to interface external memory with 8051 Microcontroller.

Before going in to the details of the 8051 Microcontroller Memory Organization, we will first see a little bit about the Computer Architecture and then proceed with memory organization.

Types of Computer Architecture Basically, Microprocessors or Microcontrollers are classified based on the two types of Computer Architecture: Von Neumann Architecture and Harvard Architecture.

Harvard Architecture

Harvard Architecture, in contrast to Von Neumann Architecture, uses separate memory for Instruction (Program) and Data.

Since the Instruction Memory and Data Memory are separate in a Harvard Architecture, their signal paths i.e. buses are also different and hence, the CPU can access both Instructions and Data at the same time.

Almost all Microcontrollers, including 8051 Microcontroller implement Harvard Architecture.

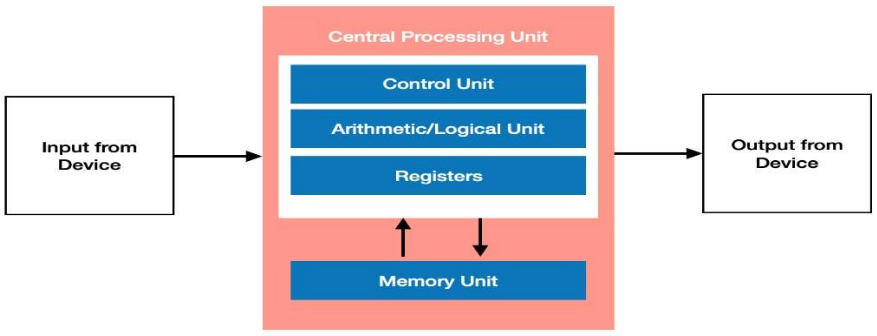


Fig8: Harvard Architecture

Von Neumann Architecture

Von Neumann Architecture or Princeton Architecture is a Computer Architecture, where the Program i.e. the Instructions and the Data are stored in a single memory.

Since the Instruction Memory and the Data Memory are the same, the Processor or CPU cannot access both Instructions and Data at the same time as they use a single bus.

This type of architecture has severe limitations to the performance of the system as it creates a bottleneck while accessing the memory.

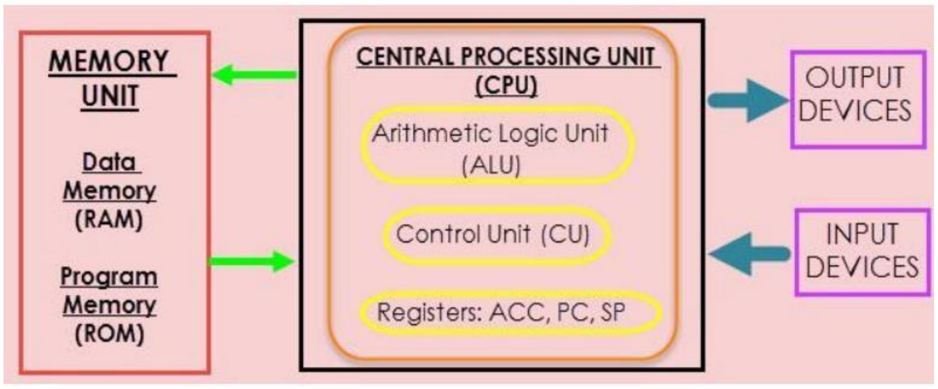


Fig9: Von Neumann Architecture

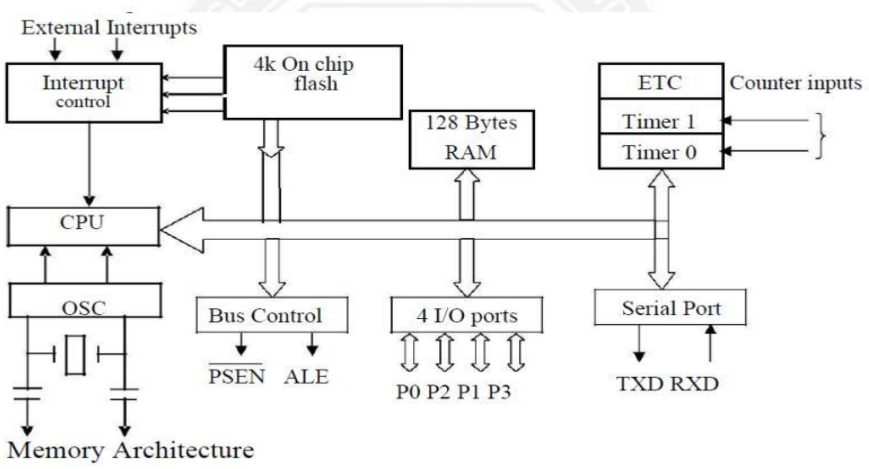


Fig 10:Block Diagram Of Architecture Of 8051

Features of 8051:

- 8-bit ALU, Accumulator, 8-bit Registers and 8-bit data bus; hence it is an 8-bit microcontroller 16-bit program counter
- 8-bit Program Status Word(PSW)
- 8-bit Stack Pointer
- Internal RAM of128bytes
- On chip ROM is4KB
- Special Function Registers (SFRs) of 128bytes
- 32 I/O pins arranged as four 8-bit ports (P0 -P3)
- Two 16-bit timer/counters : T0 andT1
- Two external and three internal vectored interrupts

- Full duplex UART (serial port)

2.2 CLASSIFICATION OF 8051

The 8051 microcontroller, developed by Intel in 1980, is classified based on its architecture, memory configuration, and instruction set. It is an 8-bit microcontroller, meaning it processes data in 8-bit chunks, making it suitable for simple embedded systems. The 8051 follows a Harvard architecture, which separates program memory and data memory, allowing for faster execution. Its core features include a built-in CPU, RAM, ROM, timers, and I/O ports, making it versatile for various applications like automation, consumer electronics, and industrial control systems.

Another way to classify the 8051 is by its memory variants. The original 8051 had 4KB of ROM and 128 bytes of RAM, but modern derivatives come with different memory configurations, such as the 8031 (no ROM), 8052 (extended RAM and ROM), and 8751 (EPROM instead of ROM). Flash memory-based versions like the AT89C51 from Atmel have also become popular due to their reprogrammability. These variations allow developers to choose the right microcontroller based on cost, memory requirements, and application complexity.

Lastly, the 8051 family can be classified based on its instruction set and enhancements. The standard 8051 has 111 instructions, including arithmetic, logical, and branching operations. Enhanced versions, like the DS89C4x0 from Maxim Integrated, offer higher clock speeds, additional peripherals (UART, ADC, PWM), and lower power consumption. Some modern 8051-compatible microcontrollers also include advanced features like insystem programming (ISP) and real-time operating system (RTOS) support, ensuring their relevance in contemporary embedded systems despite newer architectures like ARM and AVR.

2.3 8051 PIN CONFIGURATION

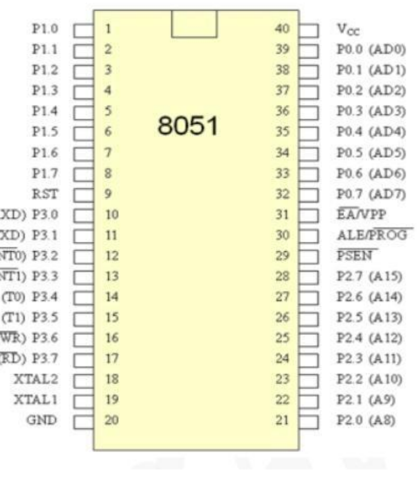


Fig 11: 8051 Pin Configuration

- **Pins 1 to 8** − these pins are known as Port 1. This port doesn't serve any other functions. It is internally pulled up, bi-directional I/O port.
- **Pin 9** − It is a RESET pin, which is used to reset the microcontroller to its initial values.

- **Pins 10 to 17** − These pins are known as Port 3. This port serves some alternate functions like interrupts, timer input, control signals, serial communication signals RxD and TxD, etc.

- **Pins 18 & 19** − These pins are used for interfacing an external crystal to get the system clock.

- **Pin 20**:- Titled as Vss – it symbolizes ground (0 V) association.
- **Pins 21 to 28** − These pins are known as Port 2. It serves as I/O port. Higher order address bus signals are also multiplexed using this port.
- **Pin 29** − This is PSEN pin which stands for Program Store Enable. It is used to read a signal from the external program memory.
- **Pin 30** − This is EA pin which stands for External Access input. It is used to enable/disable the external memory interfacing.
- **Pin 31** − This is ALE pin which stands for Address Latch Enable. It is used to demultiplex the address-data signal of port.
- **Pins 32 to 39** − These pins are known as Port 0. It serves as I/O port. Lower order address and data bus signals are multiplexed using this port.
- **Pin 40** − This pin is used to provide power supply to the circuit.
- 8051 microcontrollers have 4 I/O ports each of 8-bit, which can be configured as input or output. Hence, total 32 input/output pins allow the microcontroller to be connected with the peripheral devices.
- Pin configuration, i.e. the pin can be configured as 1 for input and 0 for output as per the logic state.
- Input Configuration: If any pin of this port is configured as an input, then it acts as if it "floats", i.e. the input has unlimited input resistance and in-determined potential.
- Output Configuration: When the pin is configured as an output, then it acts as an "open drain". By applying logic 0 to a port bit, the appropriate pin will be connected to ground (0V), and applying logic 1, the external output will keep on "floating". In order to apply logic 1 (5V) on this output pin, it is necessary to build an external pullup resistor.

CHAPTER-3

PROGRAMMING LANGUAGE

3.1 EMBEDDED C PROGRAMMING (BASIC)

Embedded C is a specialized version of the C programming language used for developing microcontroller-based applications. Unlike general-purpose C, Embedded C focuses on hardware-level operations, such as register manipulation, bitwise operations, and direct memory access. It is widely used in programming microcontrollers like the 8051, ARM, AVR, and PIC, where efficiency, speed, and low resource consumption are critical. Embedded C programs are typically written to interact with sensors, actuators, and communication modules, making it essential for IoT, automotive systems, and industrial automation.

Key Features of Embedded C

Embedded C includes features like bit manipulation, memory-mapped I/O, and direct hardware control, which are not commonly used in standard C. Since embedded systems have limited resources (RAM, ROM, and processing power), Embedded C emphasizes optimization and efficiency. It also relies heavily on pointers and structures to interact with hardware registers. Additionally, Embedded C often uses interrupts for real-time event handling, ensuring timely responses to external triggers like button presses or sensor inputs.

Basic Structure of an Embedded C Program

A typical Embedded C program consists of:

1. Header Files (e.g., #include <reg51.h> for 8051) – Provide microcontroller-specific definitions.
2. Configuration Bits – Set up clock speed, I/O modes, and peripheral settings.
3. Main Function – The entry point where initialization and the main loop execute.
4. Hardware Initialization – Configuring GPIO, timers, UART, etc.
5. Infinite Loop – Continuously runs the application logic (e.g., reading sensors, controlling motors).

Common Embedded C Operations

Some fundamental operations in Embedded C include:

- GPIO Control: Setting pins as input/output (e.g., P1 = 0xFF; for 8051).
- Timers & Interrupts: Used for delays, PWM, and real-time tasks.
- Analog-to-Digital Conversion (ADC): Reading sensor values.
- Serial Communication (UART, SPI, I2C): Sending/receiving data.

- Bitwise Operations: Used for efficient register manipulation (e.g., P0 |= (1 << 3); to set a bit).

Challenges & Best Practices

Embedded C programming comes with challenges like limited debugging tools, hardware dependency, and realtime constraints. Best practices include:

- Writing modular code for reusability.
- Optimizing memory usage by avoiding dynamic allocation.
- Using watchdog timers to recover from crashes.
- Testing on actual hardware since simulators may not replicate real-world behavior.

By mastering Embedded C, developers can build efficient, reliable embedded systems for diverse applications.

Example

```
#include <reg51.h>  // Header file for 8051 registers   void
delay(unsigned int time);  // Delay function declaration
void main() {      while(1) {
       P1 = 0x00;  // Turn ON LED (assuming active-low)
delay(1000);
       P1 = 0xFF;  // Turn OFF LED
delay(1000);
   }
}
void delay(unsigned int time) {
unsigned int i, j;      for(i = 0; i
< time; i++)          for(j = 0; j <
1275; j++);
}
```

3.2 DATA TYPES IN EMBEDDED C

Embedded C uses standard data types to define variables, but with a stronger emphasis on memory efficiency and hardware interaction compared to general-purpose programming. The most common data types include char (8-bit), int (16-bit), short (16-bit), long (32-bit), and float (32-bit floating-point). Since embedded systems often have limited RAM (e.g., 2KB in an 8051), choosing the right data type is crucial to optimize performance. For example, using an int instead of a long when only small numbers are needed saves memory and processing time.

Embedded C often employs qualifiers like unsigned, signed, volatile, and const to enhance control over variables.

An unsigned char (range: 0 to 255) is frequently used for sensor readings or loop counters, while signed char (-128 to 127) is useful for negative values. The volatile keyword tells the compiler that a variable(e.g.,a hardware register) can change unexpectedly, preventing optimization errors. Meanwhile, const ensures a variable remains read-only, saving flash memory when storing fixed values like lookup tables.

Bit-Specific and Register-Level Data Types

In low-level embedded programming, bit manipulation is common, and some compilers support bit-sized data types like bit (1-bit) or sbit (single-bit access in 8051) for direct port control. For memory-mapped registers, Embedded C uses special data types like uint8_t, uint16_t, and uint32_t (from <stdint.h>) to guarantee exact bit-widths across platforms. This is critical when interfacing with peripherals (e.g., ADC, UART) where register sizes must match exactly.

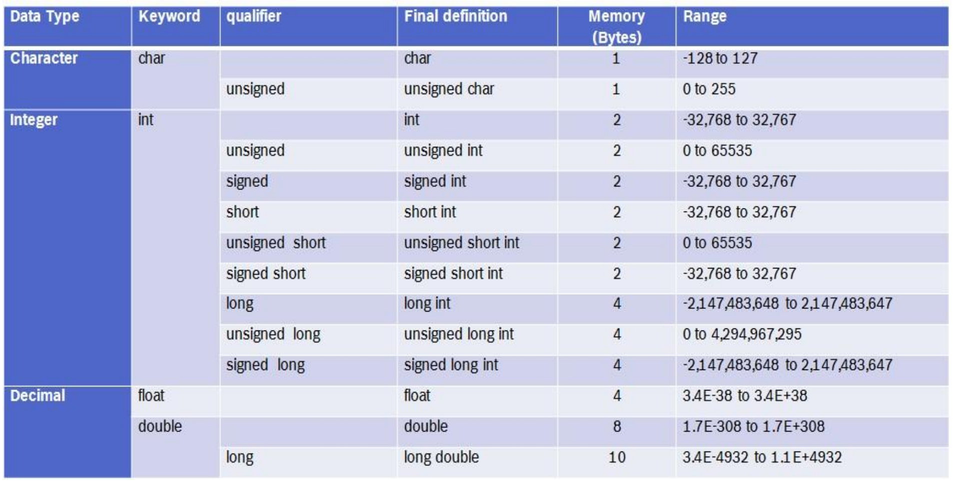
Floating-Point and Custom Data Types

While float and double are available, they are avoided in resource-constrained systems due to high computational overhead. Instead, fixed-point arithmetic or integer scaling is often used for decimal operations.

Embedded C also allows custom data types via typedef, such as: typedef

uint8_t sensor_data_t; // Custom type for sensor readings

This improves code readability and portability. By carefully selecting data types, embedded developers balance precision, speed, and memory usage, ensuring efficient firmware execution on microcontrollers.



Data Type	Keyword	qualifier	Final definition	Memory (Bytes)	Range
Character	char		char	1	-128 to 127
		unsigned	unsigned char	1	0 to 255
Integer	int		int	2	-32,768 to 32,767
		unsigned	unsigned int	2	0 to 65535
		signed	signed int	2	-32,768 to 32,767
		short	short int	2	-32,768 to 32,767
		unsigned short	unsigned short int	2	0 to 65535
		signed short	signed short int	2	-32,768 to 32,767
		long	long int	4	-2,147,483,648 to 2,147,483,647
		unsigned long	unsigned long int	4	0 to 4,294,967,295
		signed long	signed long int	4	-2,147,483,648 to 2,147,483,647
Decimal	float		float	4	3.4E-38 to 3.4E+38
	double		double	8	1.7E-308 to 1.7E+308
		long	long double	10	3.4E-4932 to 1.1E+4932

Fig 12: Data Types in Embedded C

3.3 EMBEDDED SYSTEM C OPERATORS & CONDITIONAL STRUCTURES

Operators in Embedded C

Embedded C uses standard C operators for arithmetic, logic, and bit manipulation, but with a stronger focus on efficiency and hardware control. Key operators include:

✞ Arithmetic Operators (+, -, *, /, %)

Used for calculations, but avoid floating-point (/) in resource-constrained systems.

Example: pwm_duty = (adc_value * 100) / 1023; (Scaling ADC to percentage).

✟ Bitwise Operators (&, |, ^, ~, <<, >>)

Critical for register manipulation (e.g., setting GPIO pins).

Example: PORTB |= (1 << PB0); (Set PB0 high without affecting other bits).

✟ Logical Operators (&&, ||, !)

Used in conditionals for sensor checks or state machines.

Example: if (temp > 30 && fan_on == false) start_fan();

✞ Relational Operators (==, !=, >, <, >=, <=) Compare sensor values or flags.

Example: if (adc_value >= 512) led_on();

✞ Assignment Operators (=, +=, &=, etc.)

Shorthand for efficient code (e.g., counter += 1; instead of counter = counter + 1

✞ **Conditional Structures in Embedded C**

Conditionals control program flow based on hardware states or sensor inputs. Key structures: if / else

if / else

Basic decision-making for sensor thresholds or errors. if

```
(button_pressed) { led_on();
} else {
led_off();
}
```

✞ switch-case

Efficient for multi-state systems (e.g., menu navigation, FSM).

```
switch (system_state) { case IDLE:    sleep mode (); break; case
ACTIVE:  read sensors (); break; default:      error handler ();
break;
}
```

✞ Ternary Operator (? :)

Compact conditional assignments (useful for register tweaks).

```
Pwm duty = (adc_value > 512) ? 100 : 50;  // Set 100% or 50% duty cycle.
```

✞ Loops with Conditionals (while, for)

Poll sensors or wait for hardware flags (with timeouts to avoid deadlocks).

```
while (!(UCSRA & (1 << RXC))) {  // Wait for UART data if
(timeout++ > 1000) break;  // Prevent infinite loop }
```

3.4 EMBEDDED C PROGRAMMING (DELAYS)

Embedded C programming often requires precise timing control, and delays are essential for tasks like debouncing switches, generating PWM signals, or synchronizing communication protocols. Unlike generalpurpose programming, embedded systems lack an operating system to manage time, so delays must be implemented manually using hardware timers or software loops. In 8051 microcontrollers, delays can be created using simple for or while loops that waste CPU cycles, but this approach is inefficient and blocks other operations. For more accurate and non-blocking delays, hardware timers are preferred, as they allow the CPU to perform other tasks while waiting.

Software-Based Delays in Embedded C

In embedded systems, delays are often implemented using software loops that consume CPU cycles to create precise timing. A simple delay can be generated using nested for or while loops, where the loop count determines the delay duration. For example, on an 8-bit microcontroller like the 8051, a 1ms delay can be approximated by calibrating loop iterations based on the clock frequency (e.g., 12MHz). However, software delays are blocking—they halt all other operations, making them unsuitable for multitasking systems. They also vary with compiler optimizations and clock speed, requiring manual tuning for accuracy.

Hardware Timer Delays for Precision

For more accurate and non-blocking delays, hardware timers are preferred. Microcontrollers like the 8051, ARM Cortex-M, or AVR have built-in timers that can be configured to trigger interrupts after a set period. For instance, a 10ms delay can be achieved by initializing a 16-bit timer, setting its overflow value, and enabling an interrupt. When the timer overflows, the Interrupt Service Routine (ISR) executes, allowing the main program to continue running. This approach is efficient for real-time systems where precise timing is critical, such as PWM generation or sensor polling.

Hybrid and Advanced Delay Techniques

In complex embedded applications, hybrid delay methods combine hardware timers and RTOS (Real-Time Operating System) features. An RTOS can provide task-scheduled delays using vTaskDelay() (in FreeRTOS), allowing other tasks to run during the wait. For ultra-low-power systems, sleep modes with watchdog timer wake-ups minimize energy consumption during delays. The choice of delay method depends on factors like timing accuracy, CPU load, and power constraints, with hardware timers being the most reliable for timecritical operations while software loops remain useful for simple, short delays in bare-metal firmware.

3.5 PERIPHERALS AND PROGRAMMING IN EMBEDDED SYSTEMS (OVERVIEW)

Embedded systems rely on peripherals—hardware components connected to a microcontroller—to interact with the external world. Common peripherals include GPIO (General-Purpose Input/Output), timers, UART (Universal Asynchronous Receiver-Transmitter), ADC (Analog-to-Digital Converter), PWM (Pulse Width Modulation), and I2C/SPI communication interfaces. These peripherals allow the microcontroller to read sensors, control actuators, communicate with other devices, and perform timing operations. Programming these peripherals involves configuring registers, handling interrupts, and ensuring efficient data flow between the CPU and external hardware.

GPIO and Basic I/O Programming

The most fundamental peripheral is GPIO, which allows the microcontroller to read digital inputs (like buttons) and control outputs (like LEDs). In embedded C, GPIO pins are configured as input or output by setting direction registers (e.g., P1 = 0xFF for output in 8051). Writing to a port (e.g., P0 = 0x01) turns pins on/off, while reading (e.g., if (P2 & 0x80)) checks input states. Proper initialization and pull-up/pull-down resistor configurations are essential to avoid floating inputs and ensure reliable operation.

Timers are critical for generating delays, measuring pulse widths, and scheduling tasks. Microcontrollers like the 8051 have built-in timers (Timer 0, Timer 1) that can operate in different modes (8-bit auto-reload, 16-bit, etc.). Programming timers involves setting control registers (like TMOD in 8051), loading initial values (TH0, TL0), and enabling interrupts if needed. For example, a 1ms delay can be created by configuring a timer to overflow at precise intervals and triggering an ISR (Interrupt Service Routine) to handle the event.

Communication Interfaces: UART, I2C, and SPI

Serial communication peripherals enable microcontrollers to exchange data with other devices. UART is used for asynchronous communication (e.g., sending data to a PC via USB-TTL). I2C and SPI are synchronous protocols for interfacing with sensors, EEPROMs, and displays. Programming these involves initializing baud rates (for UART), configuring clock signals (for SPI), and handling start/stop conditions (for I2C). Libraries or direct register manipulation are used to send/receive data bytes efficiently.

Analog and PWM Control with ADC and PWM Modules

Many embedded systems require analog signal processing, which is handled by ADCs. Microcontrollers read analog voltages (e.g., from a temperature sensor) and convert them to digital values. PWM peripherals generate variable-duty-cycle signals to control motor speed, LED brightness, or servo positions. Programming ADCs involves setting reference voltages and reading conversion results, while PWM requires configuring frequency and duty cycle registers.

Interrupts and Real-Time Event Handling

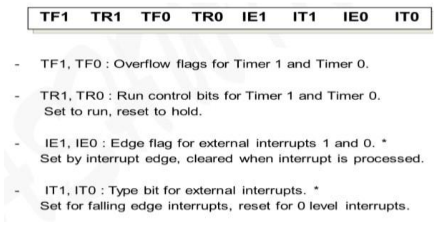
Peripherals often use interrupts to notify the CPU of events (e.g., a button press or UART data reception).

Interrupt programming involves enabling specific interrupts (like EX0 for external interrupts in 8051), writing ISRs, and prioritizing tasks. Well-managed interrupts improve system responsiveness, allowing the CPU to handle multiple peripherals efficiently without constant polling.

In summary, peripheral programming in embedded systems involves configuring hardware modules, managing data flow, and optimizing performance through interrupts and efficient coding techniques. Mastery of these concepts is essential for developing robust and responsive embedded applications.

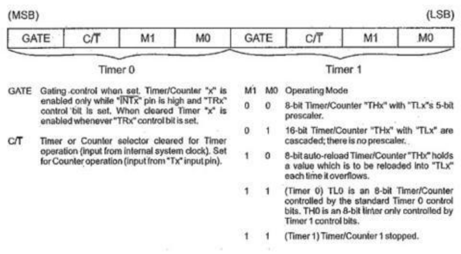
3.6 8051 TIMERS

TCON & TMOD



TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

- TF1, TF0 : Overflow flags for Timer 1 and Timer 0.
- TR1, TR0 : Run control bits for Timer 1 and Timer 0.
  Set to run, reset to hold.
- IE1, IE0 : Edge flag for external interrupts 1 and 0. *
  Set by interrupt edge, cleared when interrupt is processed.
- IT1, IT0 : Type bit for external interrupts. *
  Set for falling edge interrupts, reset for 0 level interrupts.



GATE	Gating control when set. Timer/Counter "x" is enabled only while "$\overline{\text{INTx}}$" pin is high and "TRx" control bit is set. When cleared Timer "x" is enabled whenever "TRx" control bit is set.
C/T	Timer or Counter selector cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin).

M1	M0	Operating Mode
0	0	8-bit Timer/Counter "THx" with "TLx"s 5-bit prescaler.
0	1	16-bit Timer/Counter "THx" with "TLx" are cascaded; there is no prescaler.
1	0	8-bit auto-reload Timer/Counter "THx" holds a value which is to be reloaded into "TLx" each time it overflows.
1	1	(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits. TH0 is an 8-bit timer only controlled by Timer 1 control bits.
1	1	(Timer 1) Timer/Counter 1 stopped.

Fig 13: 8051 Timers Modes

MODES OF OPERATION:

- **MODE 0 :**

- Mode 0 is exactly same like mode 1 except that it is a 13-bit timer instead of 16-bit. The 13- bit counter can hold values between 0000 to 1FFFH in TH-TL.
- Therefore, when the timer reaches its maximum of 1FFH, it rolls over to 0000, and TF is raised.

- **MODE 1:**

- It is a 16-bit timer; therefore it allows values from 0000 to FFFFH to be loaded into the timer's registers TL and TH.
- After TH and TL are loaded with a 16-bit initial value, the timer must be started.
- We can do it by "SETB TR0" for timer 0 and "SETB TR1" for timer 1.
- After the timer is started, it starts count up until it reaches its limit of FFFFH.
- When it rolls over from FFFF to 0000H, it sets high a flag bit called TFx (timer flag).
- This timer flag can be monitored.
- When this timer flag is raised, one option would be stop the timer with the instructions "CLR TR0" or CLR TR1 for timer 0 and timer 1 respectively.
- Again, it must be noted that each timer flag TF0 for timer 0 and TF1 for timer1.
- After the timer reaches its limit and rolls over, in order to repeat the process the registers TH and TL must be reloaded with the original value and TF must be reset to 0

- **MODE 2:**

- It is an 8 bit timer that allows only values of 00 to FFH to be loaded into the timer's register TH.
- After THx is loaded with 8 bit value, the 8051 gives a copy of it to TLx.
- Then the timer must be started.

- It is done by the instruction “SETB TR0” for timer 0 and “SETB TR1” for timer1.
- This is like mode 1.
- After timer is started, it starts to count up by incrementing the TLx register.
- It counts up until it reaches its limit of FFH.

- When it rolls over from FFH to 00, it sets high the TFx (timer flag).

- If we are using timer 0, TF0 goes high; if using TF1 then TF1 is raised.

3.7 SERIAL COMMUNICATION PROGRAMMING

Serial communication is a fundamental feature of the 8051 microcontroller, allowing data transfer between devices using protocols like UART (Universal Asynchronous Receiver-Transmitter). The 8051 has a built-in serial port (SBUF register) that supports full-duplex communication, meaning it can transmit and receive data simultaneously. Serial communication is widely used in applications such as PC interfacing, wireless modules (Bluetooth, Wi-Fi), and sensor data logging. Unlike parallel communication, which requires multiple wires, serial communication uses just two wires (TX & RX), making it efficient for long-distance data transfer.

UART Configuration and Registers

The 8051's serial communication is controlled by the SCON (Serial Control) register and uses Timer 1 for baud rate generation. Key bits in SCON include:

- SM0, SM1 – Set the operation mode (8-bit UART, 9-bit UART, or variable baud rate).
- REN – Enables reception of data.
- TI (Transmit Interrupt) – Set when data transmission completes.

Programming UART Transmission

To send data via UART, the microcontroller writes a byte to the SBUF (Serial Buffer) register, which automatically transmits it. The TI flag must be cleared before sending new data. Below is an example of sending a string over UART:

```
#include <reg51.h>  void UART_Init() {
   TMOD = 0x20;    // Timer 1, Mode 2 (auto-reload)
   TH1 = 0xFD;     // 9600 baud @ 11.0592 MHz
   TR1 = 1;        // Start Timer 1
   SCON = 0x50;    // 8-bit UART, REN enabled
}
void UART_Send(char data) {
   SBUF = data;   // Load data into buffer      while
(!TI);    // Wait for transmission
```

```
TI = 0;        // Clear transmit flag
```

```
}   void main()
```

```
{
   UART_Init();
   UART_Send('A'); // Send character 'A'     while
(1);
}
```

Programming UART Reception

Receiving data requires polling the **RI flag** or using **interrupts**. When data arrives, it is stored in **SBUF**, and the **RI flag** is set. The following code reads incoming data and echoes it back:

```
#include <reg51.h>    void
UART_Init() {
   TMOD = 0x20;    // Timer 1, Mode 2
   TH1 = 0xFD;     // 9600 baud
   TR1 = 1;
   SCON = 0x50;    // Enable receiver
}
char UART_Receive() {
   while (!RI);    // Wait for data       RI =
0;        // Clear receive flag       return
SBUF;   // Return received data
}   void main() {       UART_Init();
while (1) {           char data =
UART_Receive();
      UART_Send(data); // Echo back
}
```

}

Interrupt-Driven UART Communication

For efficient serial communication, **interrupts** can be used instead of polling. The **ES (Enable Serial Interrupt)** bit in **IE (Interrupt Enable)** register triggers an interrupt when **TI or RI** is set. Example:

```
#include    <reg51.h>        void
UART_ISR() interrupt 4 {
    if (RI) {       // If data received
        RI = 0;
        SBUF = SBUF; // Echo back
    }       if (TI) TI = 0; // Clear transmit
flag
}   void main()
{
    TMOD = 0x20;
    TH1 = 0xFD;
    TR1 = 1;
    SCON = 0x50;
    EA = 1;        // Enable global interrupts       ES = 1;
// Enable serial interrupt       while (1);     // Let
interrupt handle communication   }
```

Applications of Serial Communication

- **PC Debugging** – Sending sensor data to a terminal (e.g., PuTTY).
- **Wireless Modules** – Interfacing **Bluetooth (HC-05)** and **Wi-Fi (ESP8266)**.
- **Embedded Networking** – Multi-microcontroller communication.
- **GPS & GSM Systems** – Receiving location/SMS data.

CHAPTER-4

4.1 INTERRUPTS

○ The Microcontroller can serve several devices. ○ The Interrupt is the method to indicate the microcontroller by sending an interrupt signal. ○ After receiving an interrupt, the microcontroller interrupts whatever it is doing and serves the device. ○ The program associated with the interrupt is called the interrupt service routine (ISR). ○ When an interrupt is invoked, the microcontroller runs the interrupt service routine. ○ For every interrupt, there is a fixed location set aside to hold the addresses of ISRs

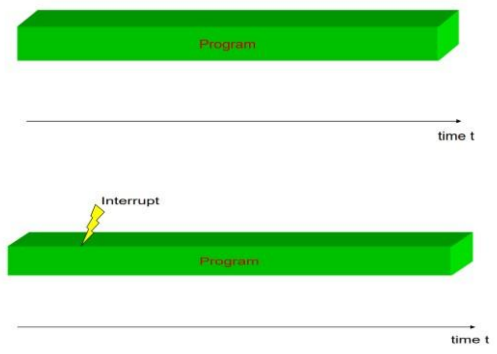


Fig 14:Interrupt Handling

4.2 SENSORS & ACTUATORS

Sensors in Embedded Systems

Sensors are critical components in embedded systems that detect physical or environmental changes and convert them into electrical signals. Common sensors include temperature sensors (LM35, DHT11), motion detectors (PIR), proximity sensors (IR, ultrasonic), and pressure sensors (BMP180). These devices provide real-time data to microcontrollers (like the 8051 or Arduino), enabling systems to monitor conditions such as heat, light, motion, or humidity. Sensors typically interface with ADCs (Analog-to-Digital Converters) for analog signals or GPIO pins for digital signals, allowing the microcontroller to process and respond to sensor inputs.

Actuators in Embedded Systems

Actuators are output devices that convert electrical signals into physical action, allowing embedded systems to interact with the real world. Examples include motors (DC, stepper, servo), relays, solenoids, and LEDs. Actuators are controlled using PWM (Pulse Width Modulation) for speed/dimming, H-bridge circuits for motor direction, or simple GPIO toggling for on/off control. For instance, a microcontroller can drive a servo motor to adjust a robotic arm or switch a relay to control high-power appliances. Proper driver circuits (like transistors or motor drivers) are often needed to interface actuators safely.

Sensor-Actuator Interaction

In embedded systems, sensors and actuators often work together in closed-loop feedback systems. For example, a temperature sensor (LM35) may feed data to a microcontroller, which then adjusts a cooling fan (actuator) via PWM to maintain a set temperature. Similarly, an ultrasonic sensor can detect obstacles, prompting a motor to stop or change direction in an autonomous robot. This interaction relies on real-time data processing, control algorithms (like PID), and efficient firmware to ensure accurate and responsive system behavior.

Interfacing Sensors & Actuators with Microcontrollers

Sensors and actuators connect to microcontrollers through GPIO, ADC, PWM, or communication protocols (I2C, SPI, UART). Analog sensors (e.g., potentiometers) require an ADC to convert signals, while digital sensors (e.g., DHT11) use predefined communication protocols. Actuators like stepper motors may need dedicated driver ICs (e.g., ULN2003), and relays often require optocouplers for isolation. Firmware must handle sensor calibration, noise filtering, and actuator control timing to ensure reliability.

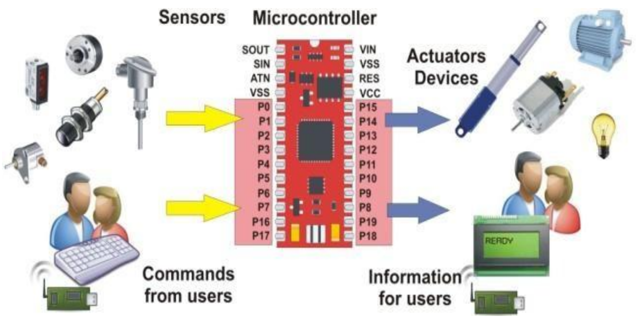


Fig 15: Sensors & Actuators

4.3 INTERFACING DC MOTOR WITH 8051 MICROCONTROLLER

Interfacing a DC motor with an 8051 microcontroller requires an H-bridge driver (like L293D or L298N) to control the motor's direction and speed, as the 8051's GPIO pins cannot supply sufficient current directly. The motor is connected to the H-bridge's output pins, while the 8051's GPIO pins control the driver's input pins to set direction (forward/reverse). For speed control, a PWM (Pulse Width Modulation) signal is generated using the 8051's timers and applied to the H-bridge's enable pin, allowing variable motor speeds by adjusting the duty cycle. Additionally, flyback diodes are used across the motor terminals to protect the circuit from voltage spikes caused by inductive loads. This setup enables precise motor control in applications like robotics, conveyor systems, and automated machinery.

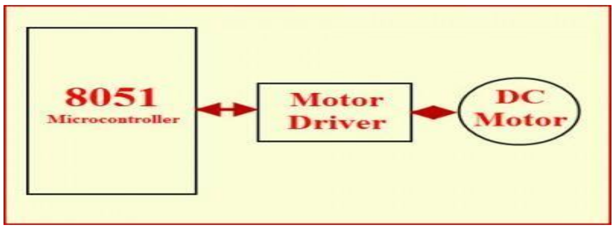


Fig 15: Interfacing DC Motor with 8051 Microcontroller

Here, interfacing 8051 with DC motor requires a motor driver.

There are various types of driver ICs among which L293D is typically used for interfacing DC motor with 8051.

L293 is an IC with 16 pins.

4.4 EMBEDDED SYSTEM DESIGN (PROTEUS INTRODUCTION)

Proteus is a powerful simulation and PCB design tool widely used in embedded system development, offering a virtual environment to model and test microcontroller-based circuits (like 8051, Arduino, and ARM) before hardware implementation. Its ISIS (Intelligent Schematic Input System) allows users to design schematics with components such as sensors, motors, and displays, while VSM (Virtual System Modelling) enables real-time firmware simulation by integrating embedded C/assembly code. With features like debugging, logic analyser, and peripheral emulation, Proteus helps validate system behaviour, detect errors early, and optimize designs, making it essential for students and engineers developing IoT, robotics, and automation projects efficiently.

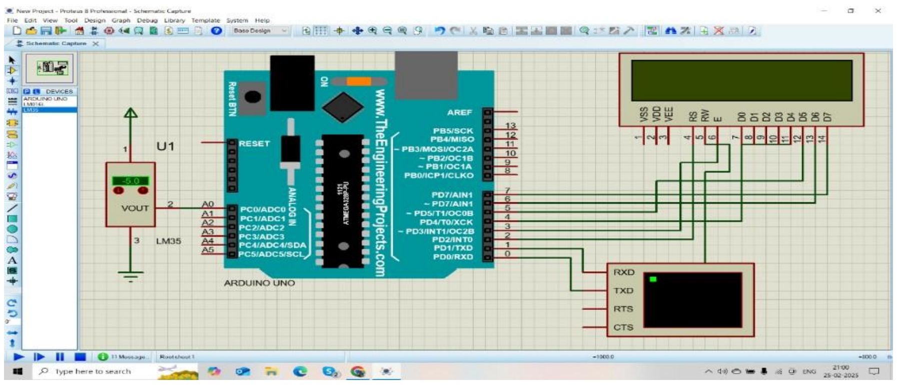


Fig 16: Interface Of Proteus Software

CHAPTER-5

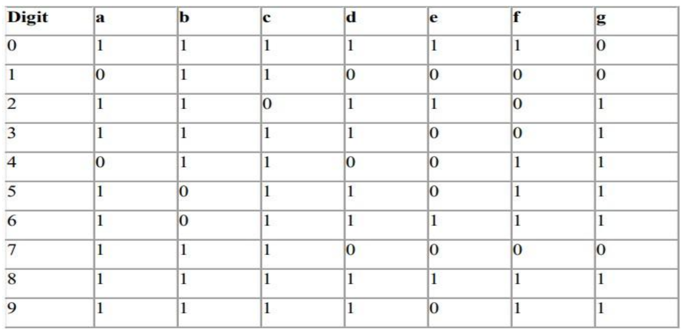
5.1 INTERFACE A SEVEN SEGMENT LED DISPLAY TO AN 8051 MICROCONTROLLER

- 7 segment LED display is very popular and it can display digits from 0 to 9 and quite a few characters like A, b, C, ., H, E, e, F, n, o,t,u,y, etc.
- Knowledge about how to interface a seven-segment display to a micro controller is very essential in designing embedded systems.
- A seven-segment display consists of seven LEDs arranged in the form of a squarish'8′ slightly inclined to the right and a single LED as the dot character.
- Different characters can be displayed by selectively glowing the required LED segments.
- Seven segment displays are of two types, common cathode and common anode.
- In common cathode type , the cathode of all LEDs are tied together to a single terminal which is usually labeled as 'com' and the anode of all LEDs are left alone as individual pins labeled as a, b, c, d, e, f, g & h (or dot) .
- In common anode type, the anode of all LEDs are tied together as a single terminal and cathodes are left alone as individual pins.

Digit drive pattern:

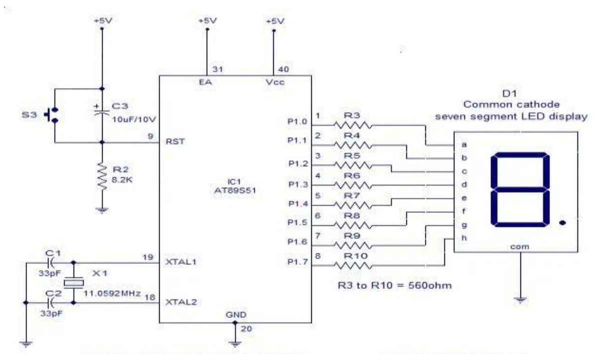
Digit drive pattern of a seven segment LED display is simply the different logic combinations of its terminals 'a' to 'h' in order to display different digits and characters.

The common digit drive patterns (0 to 9) of a seven segment display are shown in the table below.



Digit	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Fig 17: seven segment Truth Table



Interfacing 7 segment display to 8051 www.circuitstoday.com

Fig 18: Block Digarm of Seven Segment

Code Source:

```
#include<reg51.h> void delay(); unsigned char

cmd[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x67}; void main() {

unsigned char i;  while(1)  {   for(i=0;i<10;i++)

  {

   P2=cmd[i];

delay();

  }

 } } void

delay()

{  unsigned int

i=90000;  while(i--);
```

}

5.2 INTRODUCTION TO INTERNET OF THINGS (IOT)

The Internet of Things (IoT) refers to a network of interconnected physical devices embedded with sensors, software, and communication technologies that enable them to collect, exchange, and act on data. These devices, ranging from smart home appliances to industrial machines, leverage internet connectivity to communicate with each other, cloud platforms, or user interfaces, creating intelligent systems that enhance automation, efficiency, and decision-making. IoT transforms everyday objects into smart, data-driven tools, enabling applications like remote monitoring, predictive maintenance, and real-time analytics across industries such as healthcare, agriculture, and smart cities.

At its core, IoT relies on embedded systems, wireless protocols (Wi-Fi, Bluetooth, LoRa, Zigbee), and cloud computing to process and store vast amounts of data. Microcontrollers (e.g., ESP8266, Arduino) and singleboard computers (e.g., Raspberry Pi) serve as the brains of IoT devices, while sensors and actuators bridge the digital and physical worlds. With advancements in AI/ML and edge computing, IoT systems are becoming smarter, capable of localized data processing and faster responses. As IoT continues to evolve, it promises to revolutionize industries, improve resource management, and create a more connected and sustainable future.

5.3 INTRODUCTION TO RASPBERRY-PI

The Raspberry Pi is a versatile, credit-card-sized single-board computer (SBC) developed by the Raspberry Pi Foundation to promote affordable computing and STEM education. Unlike traditional microcontrollers, it runs full-fledged operating systems like Raspbian (Linux) and supports programming languages such as Python, C++, and Java, making it ideal for both beginners and advanced users. With built-in GPIO pins, USB ports, HDMI output, and wireless connectivity (Wi-Fi/Bluetooth), the Raspberry Pi serves as a powerful tool for projects ranging from home automation and robotics to media centers and IoT gateways. Its low cost, high accessibility, and strong community support have made it a favorite among hobbyists, educators, and engineers.

The Raspberry Pi family includes multiple models, such as the Pi 4, Pi Zero, and the compact Pi Pico (RP2040 microcontroller). The Pi 4, with its quad-core processor, up to 8GB RAM, and 4K video output, functions as a desktop replacement or server, while the Pi Zero offers a minimalist design for lightweight embedded projects. The Pi Pico, equipped with programmable GPIO and low-power capabilities, bridges the gap between microcontrollers and SBCs. Whether used for DIY electronics, AI prototyping, or network-attached storage (NAS), the Raspberry Pi's flexibility and scalability make it a cornerstone of modern embedded computing and innovation.

Fig 19: Raspberry-Pi

5.4 INTRODUCTION TO ARDUINO

Arduino is an open-source electronics platform designed to simplify embedded systems development for beginners and professionals alike. At its core, Arduino consists of programmable microcontroller boards (like the Uno, Nano, and Mega) and an easy-to-use Integrated Development Environment (IDE) based on Wiring, a simplified version of C++. Known for its plug-and-play functionality, Arduino boards feature built-in analog and digital I/O pins that interact with sensors, motors, LEDs, and other peripherals, making them ideal for prototyping IoT devices, robotics, and automation projects. The platform's simplicity, extensive library support, and active global community have made it a go-to choice for education, DIY electronics, and rapid prototyping.

Beyond hardware, Arduino's ecosystem includes shields (add-on modules) for Wi-Fi, motor control, and displays, as well as cloud-based tools like Arduino IoT Cloud for remote monitoring. Boards like the Arduino Uno (ATmega328P) are perfect for beginners, while the Arduino Due (ARM Cortex-M3) caters to advanced applications. With compatibility across operating systems and support for third-party clones, Arduino democratizes electronics by lowering barriers to entry. Whether used in smart agriculture, wearable tech, or interactive art, Arduino's versatility continues to drive innovation in the maker movement and beyond.

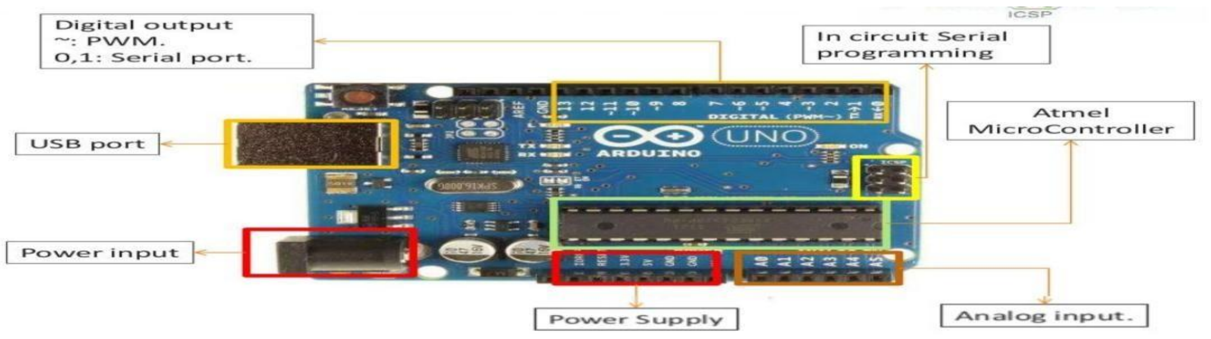
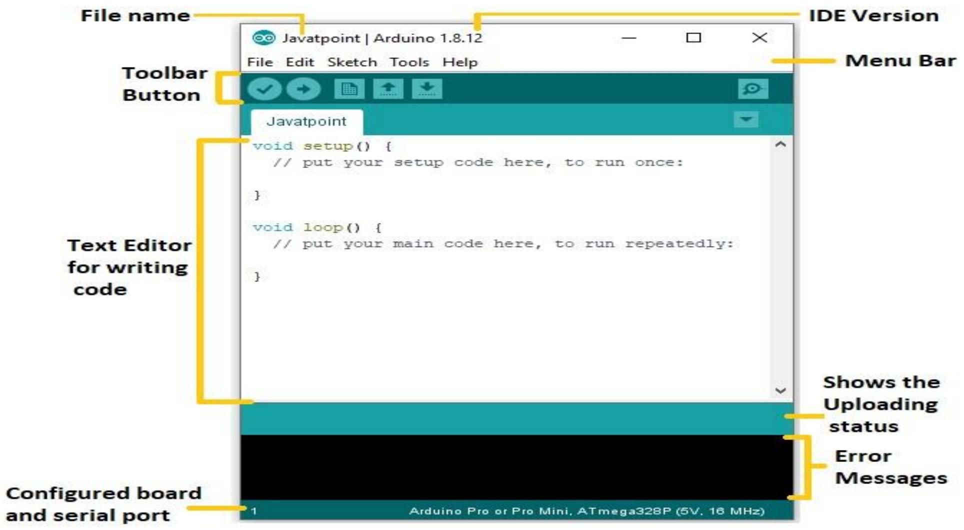


Fig 20: Arduino Board

The Arduino IDE (Integrated Development Environment) is a user-friendly, cross-platform software tool designed to write, compile, and upload code to Arduino boards effortlessly. Built on a simplified version of C/C++, it features a straightforward interface with essential functions like syntax highlighting, auto-formatting, and a serial monitor for debugging. The IDE includes a vast collection of built-in libraries for sensors, actuators, and communication protocols (I2C, SPI, UART), enabling rapid prototyping without complex configurations. Its "one-click upload" process and compatibility with Windows, macOS, and Linux make it accessible for beginners, while advanced users benefit with support for third-party boards (ESP32, STM32) via board managers. As the backbone of Arduino programming, the IDE bridges simplicity and functionality, empowering makers, students, and engineers to bring their embedded projects to life quickly.


File name
IDE Version
Javatpoint | Arduino 1.8.12
File Edit Sketch Tools Help
Menu Bar
Toolbar Button
Javatpoint
Text Editor for writing code
Shows the Uploading status
Error Messages
Configured board and serial port
1
Arduino Pro or Pro Mini, ATmega328P (5V, 16 MHz)

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Fig 21: Arduino IDE Interface

CHAPTER-6

TASK WEEK-1

Task-1

Aim: To measure the temperature reading from the sensor using Arduino.

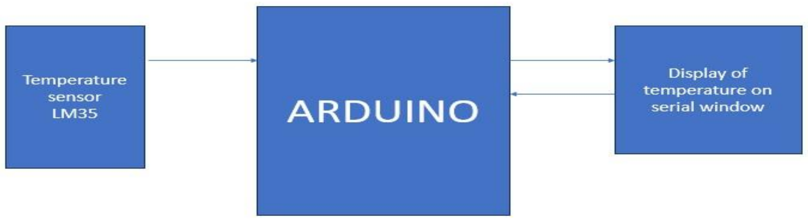
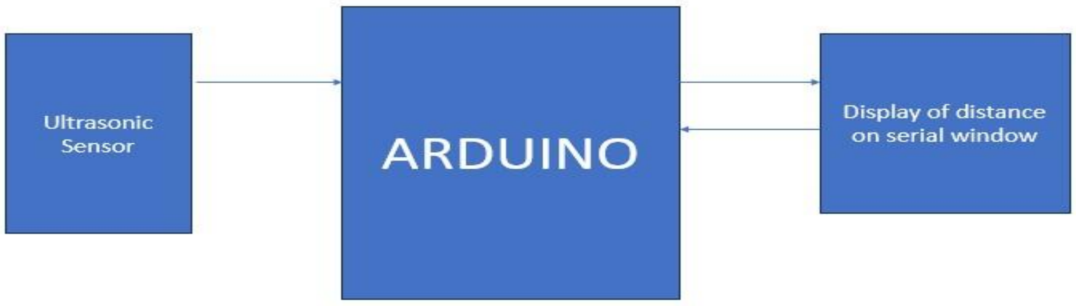


Fig 21: Block diagram of temperature reading from the sensor using Arduino

Code Source:
```
const int lm35Pin = A0;  // LM35 output
connected to A0 void setup() {
  Serial.begin(9600);     // Initialize serial communication
} void loop() {   int rawValue = analogRead(lm35Pin);  // Read analog value (0-
1023)   float voltage = (rawValue * 5.0) / 1023.0;  // Convert to voltage (0-5V)   float
temperature = voltage * 100.0;  // LM35 outputs 10mV/°C (e.g., 0.5V = 50°C)
  Serial.print("Temperature: ");
  Serial.print(temperature);
Serial.println(" °C");   delay(1000);  //
Update every second
}
```

Task2

AIM: To measure the distance of an object using Ultrasonic sensor with Arduino



Ultrasonic Sensor:

->Ultrasonic sensor can measure the distance of an object by using sound waves.

->This sensor is a very popular sensor used in many applications where measuring distance or sensing objects are required.

->This sensor has two eyes like projects in the front which forms the ultrasonic trasmitter and receiver.

->The ultrasonic transmitter transmits an ultrasonic wave, this wave travels in air and when it gets objected by any material it gets reflected back toward the sensor, this reflected wave is observed by the ultrasonic receiver module.

-> We are using HC-SR04 ultrasonic sensor.

-> This ultrasonic sensor ia a 4 pin module, whose pins names are VCC, Trigger, Echo and GND respectively.

->This sensor works with the simple formula that Distance=Speed X Time.

How it Works:

1. The trigger is sent a short pulse(10us) to start the measurement.
2. The sensor emits an ultrasonic sound wave that travels through the air.
3. If the sound wave encounters an object, it bounces back towards the sensor.
4. The echo pin goes HIGH as soon as the sound wave is reflected back to the sensor.
5. By measuring the time the echo pin remains HIGH, you can calculate the distance to the object using the speed of suong in air.

Formula-> Distance=(Timex Speed)/2 where time is the duartion the Echo pin stay HIGH the speed of suond in air is approximately 343meters pr second(or 0.0343cm/us).

Project code:

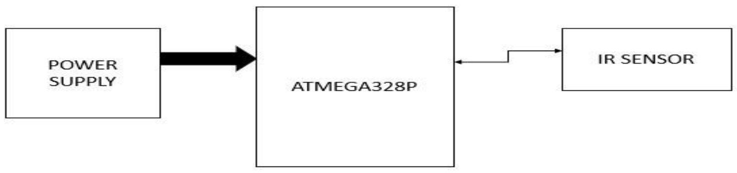
```
int echo=6; int
trigger=7; int
var=0; int d;
void setup()
{
 Serial.begin(9600);  pinMode(echo,INPUT);
pinMode(trigger,OUTPUT);
}         void
loop()
{  digitalWrite(trigger,LOW);//ensures the trigger pin is low to start
delayMicroseconds(10);// delay to make sure the sensor resets
digitalWrite(trigger,HIGH);// sending HIGH pulse to activate the sensor
delayMicroseconds(10);// again ensures a short pulse
digitalWrite(trigger,LOW);// turns the trigger pin back off
 var=pulseIn(echo,HIGH);//waits for the seosor echo signal and returns the time in microseconds it takes for
the pulse to return
 d=(0.034*var)/2;//calculates the distance the speed of suond in air is approximately 0.034cm per microsecond,
the division by 2 accounts for round trip distance.
 Serial.print("object detected at distance=");
Serial.println(d);  delay(1000);// to make all
resets
}
```

CHAPTER-2

TASK WEEK-2

Task-1

7.1Aim: The main aim of this project is to detect the object/obstacle by IR sensor with Atmega3289 microcontroller.



The ATmega328P is a microcontroller from the ATmega family of microcontrollers, which are produced by Microchip Technology (formerly Atmel).

It is widely used in embedded systems and is particularly popular for DIY electronics, especially in Arduino-based projects.

The ATmega328P is a popular, versatile microcontroller that is easy to use and widely available.

Its combination of I/O pins, memory, power efficiency, and available communication interfaces makes it suitable for many applications in embedded systems.

Code Source: LCD:16x2

lcd

control pins:

RS: register select(for comm=0, for dat=1)

RW:read/write(read=1, write=0) EN:enable(high to

low pulse) data pins(D0 to D7)-8 bit: connecting

PORTD(8 bit) wap to diplay "hai" on lcd

```
#include<avr/io.h>
```

```
void com(char x)
```

```
{
 PORTB=~(1<<0);//rs=0
 PORTD=x;//sending com to data pins
 PORTB|=(1<<1);// en=1
 _delay_ms(100);//delay
 PORTB&=~(1<<1);//en=0
 _delay_ms(100);
}   void   dat(char
y)
{
 PORTB=(1<<0);//rs=1
 PORTD=y;
 PORTB|=(1<<1);
 _delay_ms(100);
 PORTB&=~(1<<1);
 _delay_ms(100);
} int
main() {
  int i;
  DDRD=0xff;//lcd data pins connected to PORTD and as output
  DDRB=(1<<0)|(1<<1);//RS-PB0, EN-PB1(RW=write =0=GND
  char b[]="Skill Dezire";   while(1)
  {
    com(0x38);
com(0x01);
com(0x80);     com(0x0e);

for(i=0;b[i]!='\0';i++)

    {      dat(b[i]);

delay ms(50);

    }

    com(0x01);

  }

}
```

7.2 Task4: Analog to Digital Converter ADC of ATmega328P

Steps to write the program:

1. Select the input channel and reference voltage
2. Select the prescale value
3. Enabling the ADC
4. Start the conversion
5. Wait for the conversion is completed
6. Read the output: read ADC(number int 1023 then convert to ASCII 1023 to ‘1023’)
7. Display the output on LCD(use 8 bit) Programming the ADC of ATmega328P:

int g;

*Direction of LCD pins

*Commands to the LCD

1. Select the input channel: select A0

2. Select the reference voltage: as AVcc

3. Pre-scaling the frequency: 128 value

4. Enable the ADC

5. Start the conversion

6. Waiting for the conversion to be completed: -> read the ADIF bit while((ADCSRA & (1<<4))==0);

7. Read the output:

g=ADC;

8. Convert int value to Ascii.

-> g=1023-----------------------------> a[]="1023"

9. Display the read value on LCD (character).

Code Source: device operation using switches: switch1-PB1(input) switch2-PB2(input) device1-PB3(output) device2-PB4(output)

lcd:

RS-PC0

RW-GND

EN-PC1 DATA-PORTD

```
#include<avr/io.h>
void com(char x) {
 PORTC=~(1<<0);
 PORTD=x;
 PORTC|=(1<<1);
```

```
_delay_ms(50);
```

```
 PORTC&=~(1<<1);
 _delay_ms(50);
} void dat(char
y) {
  PORTC=(1<<0);
  PORTD=y;
  PORTC|=(1<<1);
  _delay_ms(50);
  PORTC&=~(1<<1);
  _delay_ms(50);
} void str(char
*st)
{
  int j;   for(j=0;st[j]!='\0';j++)
  {
dat(st[j]);
    _delay_ms(50);
  } } int
main() {
  DDRD=0xff;
  DDRB&=~((1<<1)|(1<<2));//DDRB=(0<<1)|(0<<2);
  DDRB=(1<<3)|(1<<4);
```

```
DDRC=(1<<0)|(1<<1);
```

```
com(0x38);   com(0x01);

com(0x80);   com(0x0e);

char a[]="device operation";

char b[]="using switches";

char c[]="light on ";   char

d[]="fan on ";   char

e[]="light off ";   char

f[]="fan off ";   com(0x01);

  str(a);

  com(0xc0);

  str(b);

  com(0x01);  while(1)

  {

    if((PINB&(1<<1))==0)

    {

      PORTB=(1<<3);

PORTB&=~(1<<4);

      str(c);

    }

    if((PINB&(1<<2))==0)

    {
```

```
PORTB=(1<<4);       PORTB&=~(1<<3);
```

```
    str(d);
   }
   if((PINB&(1<<1))!=0)
   {
    PORTB&=~(1<<3);
    str(e);
   }
   if((PINB&(1<<2))!=0)
   {
    PORTB&=~(1<<4);
    str(f);
   }
   com(0x01);
  }
}
```

7.3 COMPLETION OF INTERNSHIP:

The learner's journey is structured in such a way that after completion of the live & Record Session and I have Successfully my **Embedded Systems Internship** on the **Skilldzire.** I received below as :

Link: https://learn.skilldzire.com/s/courses/6787b695fa5a1a12d8d38dbf/take

RESULT:

STEP1: SUBMISSSION

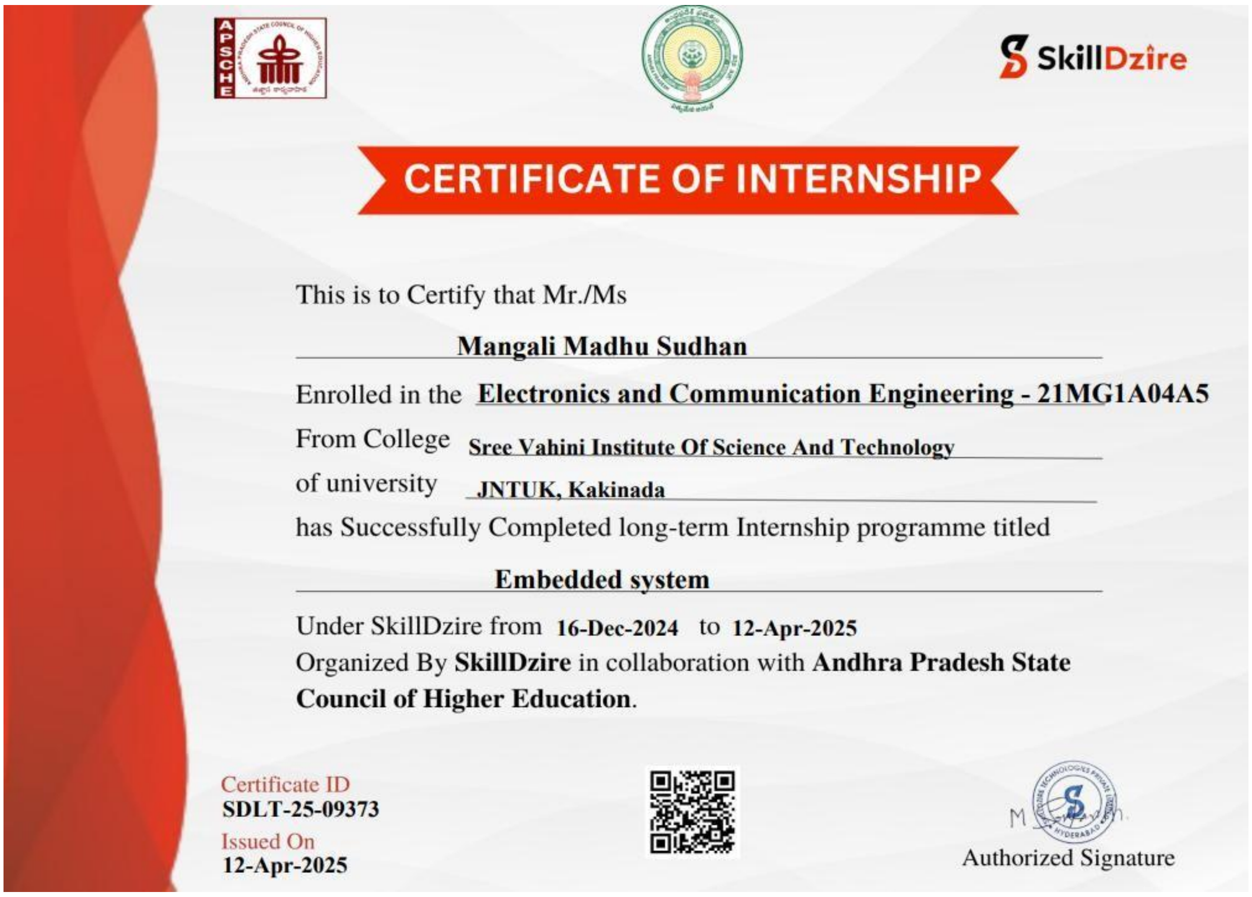
After completion of above all videos or modules, completion status by Skilldzire platform track the progress and assist according.

STEP2: VERIFICATION

Here the team will verify the completion videos of Embedded Systems on my LMS portal, which have been assigned for me. This process will take approximately 15-25days.

STEP3: CERTIFICATION GENERATION

After Successful Verification of my LMS Portal by our team. The team will generate the event partification Certificate for my Successful Completion of the **Embedded System Long Term Internship.**



APSCHE

SkillDzire

CERTIFICATE OF INTERNSHIP

This is to Certify that Mr./Ms

Mangali Madhu Sudhan

Enrolled in the **Electronics and Communication Engineering - 21MG1A04A5**

From College **Sree Vahini Institute Of Science And Technology**

of university **JNTUK, Kakinada**

has Successfully Completed long-term Internship programme titled

Embedded system

Under SkillDzire from **16-Dec-2024** to **12-Apr-2025**

Organized By **SkillDzire** in collaboration with **Andhra Pradesh State Council of Higher Education**.

Certificate ID
SDLT-25-09373

Issued On
12-Apr-2025

Authorized Signature

COMPLETION OF CERTIFICATION