Course Handout :

Embedded Systems and Telecommunications

Semester: 2
Teaching Unit: UED 1.2
Subject 1: Embedded Systems and Telecommunications
Total Hours: 22h30 (Lecture: 1h30)
Credits: 1
Coefficient: 1

Course Objectives:

This subject aims to provide basic knowledge in a field that integrates two autonomous systems: an electronic and computing system known as an embedded system. It will also allow students to understand the different stages involved in designing an embedded system.

Recommended Prerequisite Knowledge:
Microprocessor.

Author:

Dr. Mohammed Zakarya BABA-AHMED

Foreword

"Today's science is tomorrow's technology." – Edward Teller

This course handout, titled *"Embedded Systems and Telecommunications,"* is designed to provide a solid foundation for Master's students in the ST program by addressing both fundamental and advanced concepts related to embedded systems. It aims to support students in their educational journey, preparing them for the technical and scientific challenges of modern systems that intersect electronics, computing, and telecommunications.

Embedded systems are at the heart of numerous modern technological innovations, including connected devices, intelligent transportation systems, and medical equipment. This module covers essential topics such as hardware and software architectures, real-time constraints, design methodologies, and the security of embedded systems.

To promote a thorough understanding and practical application of the concepts, this handout includes concrete examples, case studies, and key references to encourage autonomy and deeper exploration of the subject matter.

We hope this material will serve as a valuable resource for students and contribute to strengthening their expertise in embedded systems while sparking their curiosity and interest in this fascinating and ever-evolving field.

Table of Contents

Notations and Abbreviations

A: Accumulator

ABS: Anti-lock Braking System

ADC: Analogue-Digital Converter

AES: Advanced Encryption Standard

AGC: Apollo Guidance Computer

API: Automate Programmable Industriel

ARM: Advanced RISC Machine

ASIC: Application-Specific Integrated Circuit

ASIP: Application-Specific Instruction set Processor

AVR: Automatic Voltage Regulator

CISC: Complex Instruction Set Computer

CoAP: Constrained Application Protocol

CPU: Central Processing Unit

CRC: Cyclic Redundancy Check

DAC: Digital- Analogue Converter

DEMA: Differential Electromagnetic Analysis

Di: Deadline Absolu

DM: Deadline Monotonic

DMA: Direct Memory Access

DoS: Denial of Service

DPA: Differential Power Analysis

DRAM: Dynamic Random Access Memory

DRM: Digital Rights Management

DSP: Digital Signal Processors

DTLS: Datagram Transport Layer Security

DVFS: Dynamic Voltage and Frequency Scaling

eCAP: evoked Compound Action Potential

ECC: Elliptic Curve Cryptography

ECU: Electronic Control Unit

EDF: Earliest Deadline First

EEPROM: Electrically Erasable Programmable Read-Only Memory

EMAC: Ethernet Media Access Controller

EMIF: External Memory InterFace

EPROM: Erasable Programmable Read-Only Memory

eQEP: enhanced Quadrature Encoder Pulse

FPGA: Field Programmable Gate Arrays

GPIO: General Purpose Input/Output

GPP: General Purpose Processor

GPS: Global Positioning System

HMI: Human-Machine Interface

HTTP: HyperText Transfer Protocol

I2C: Inter-Integrated Circuit

I2S: Inter-IC Sound

ICD: In-Circuit Debugger

IDE: Integrated Development Environment

IoT: Internet of Things

IP: Internet Protocol

IPC: Inter-Process Communication

IPsec: Internet Protocol security

IR: Instruction Register

ISDN: Integrated Services Digital Network

ISR: Interrupt Service Routine

JTAG: Joint Test Action Group

LED: Light-Emitting Diode

LLF: Least Laxity First

LoRaWAN: Long Range Wide Area Network

MAC: Media Access Control

MEMS: Micro-ElectroMechanical Systems

MIPS: Million Instructions Per Second

MPC: Model Predictive Control

MQTT: Message Queuing Telemetry Transport

MSP: Main Stack Pointer

NASA: National Aeronautics and Space Administration

OS: Operating System

OTA: Over-The-Air

PC: Personal Computer

PC: Program Counter

PCB: Printed Circuit Boards

PDA: Personal Digital Assistant

PDM: Pulse Density Modulation

PIC: Programmable Interface Controller

PID: Proportional-Integral-Derivative controller

PLC: Programmable Logic Controller

PWM: Pulse-Width Modulation

QoS: Quality of service

RAM: Random Access Memory

RISC: Reduced Instruction Set Computer

RM: Rate Monotonic

ROM: Read Only Memory

RS232: Recommended Standard 232

RSA: Rivest–Shamir–Adleman

RTC: Real-Time Communication

RTOS: Real-Time Operating System

SCL: Serial CLock line

SD: Secure Digital

SDA: Serial DAta line

SDMMC: Secure Digital/ Multi Media Card interface

SEMA: Simple Electromagnetic Analysis

SGI: Silicon Graphics International corp

SHA: Secure Hash Algorithms

SNMP: Simple Network Management Protocol

SoC: System-on-a-Chip

SOI: Silicon-On-Insulator

SOS: Silicon-On-Sapphire

SP: Stack Pointer

SPA: Simple Power Analysis

SPARC: Scalable Processor Architecture

SPI: Serial Peripheral Interface

SRAM: Static Random Access Memory

SSD: Solid-State Drive

TCP: Transmission Control Protocol

TDM: Time Division Multiplexing

TLS: Transport Layer Security

UART: Universal Asynchronous Receiver/Transmitter

UDP: User Datagram Protocol

USB: Universal Serial Bus

VCC: Common Collector Voltage

VHDL: Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language

Wi-Fi: Wireless Fidelity

WPA 3: Wi-Fi Protected Access 3

INTRODUCTION

Embedded Systems have become a cornerstone of modern technologies, playing a key role in numerous fields such as automotive, aerospace, connected devices, and telecommunications. These systems, which integrate electronic, software, and hardware components, are distinguished by their ability to perform specific tasks autonomously and in real-time, often within constrained environments.

This course handout, titled "Embedded Systems and Telecommunications," is primarily designed for Master's students in the "Telecommunication Systems" program. Its goal is to offer a thorough understanding of both foundational and advanced principles essential for the design and analysis of contemporary embedded systems. Through its five chapters, this material covers diverse topics ranging from the basics of embedded systems to advanced design methodologies, real-time system specifics, processor architectures, and security aspects.

The first chapter introduces embedded systems by detailing their functionalities, history, and hardware and software specificities. Students will explore concrete examples and typical architectures, enabling them to grasp various facets of these complex systems.

The second chapter focuses on real-time embedded systems, a critical domain for applications requiring rapid and reliable responses. This chapter explores memory and concurrency management, introduces fundamental Linux concepts for embedded systems, and examines the architecture and functioning of real-time embedded systems.

In the third chapter, the focus shifts to the architectures of embedded processors. Key architectural concepts, such as dedicated operating systems, special-purpose processors, and pipeline operations, are explained in detail. Students will also delve into communication mechanisms, associated protocols, and real-world examples of architectures.

The fourth chapter presents a comprehensive methodology for designing embedded systems. It discusses design environments, lifecycle stages, and the tools needed for developing these systems, complemented by practical examples to illustrate key principles.

Finally, the fifth chapter is dedicated to the security of embedded systems. It examines hardware and software vulnerabilities and strategies to secure communications and protect critical system data.

Through a balanced combination of theory and practice, this course handout aspires to equip students with the skills necessary to understand, design, and secure modern embedded systems.

Chapter I

Introduction to Embedded Systems

I.1 Features

I.1.1 What is an Embedded System?

You only need to look at your surroundings to find the answer:

- ✓ You are awakened in the morning by your alarm clock;
- ✓ You program your coffee machine to brew coffee;
- ✓ You turn on the TV and use your remote control;
- ✓ You drive your car, and the onboard computer alerts you if you haven't fastened your seatbelt.

All these are embedded systems. The list could go on, enumerating countless embedded systems encountered unknowingly throughout the day. In short, embedded systems surround us, omnipresent and ready to serve us. We interact with dozens of them daily without realizing it.

They are everywhere—discreet, efficient, and purpose-driven. Already omnipresent, they are bound to become even more so. Embedded systems are packed with varying degrees of complex electronics and sophisticated computing capabilities.

I.1.2 Definition

Embedded systems are systems with predefined functions, generally characterized by low power consumption. They are compact, cost-effective, and have limited computational and storage capacities (unlike public AI systems handling large-scale data) [1].

Typically, they lack standard input/output interfaces like a keyboard or a computer monitor. The hardware and software are deeply integrated, with embedded software being "hidden" within the hardware. Unlike a traditional PC environment, hardware and software in embedded systems are not easily distinguishable.

Embedded systems generally contain one or more microprocessors designed to execute a set of programs defined during the design phase. These programs are stored in memory. Embedded systems often have real-time constraints, and their resources are typically limited [2].

I.1.3 Role of Embedded Systems

Embedded systems play a crucial role across various fields, including aerospace, space exploration, automotive, industrial applications, and computing.

The term embedded computing refers to the software aspects integrated into equipment not primarily intended for computing purposes. The combination of software and hardware integrated within a device constitutes an embedded system.

There were 15 billion embedded devices in 2018 and 80 billion in 2020. The staggering growth in the number of connected devices demonstrates that the embedded systems industry continues to develop new technologies and opportunities [3].

In just two decades, embedded systems have ushered in a new industrial era, leveraging the miniaturization of electronic chips to make everyday objects powerful, intelligent, and interconnected. With this evolution, the Internet of Things (IoT) promises to be an even greater revolution than mobile technology.

As a key player in this digital and societal revolution, the embedded systems industry spans all sectors of the economy. (For instance, one-third of an aircraft's manufacturing cost is related to embedded systems.) This industry has created a growing demand for highly qualified professionals who can manage the diverse constraints of these complex systems and develop innovative solutions.

I.2 Historical Overview

The development of embedded systems is closely tied to advances in electronics, computing, and telecommunications. Below are the key historical milestones marking their emergence and evolution:

I.2.1 The Early Days (1940s–1950s)

The first systems that could be considered "embedded" appeared during World War II.

Key Example: The first guided weapon system, the "Guided Missile A-4 (V-2 rocket)," developed by Germany, used an analog computer to adjust its trajectory mid-flight.

These systems were rudimentary, bulky, and relied on analog circuits and mechanical relays.

I.2.2 The Birth of Modern Embedded Systems (1960s)

With the invention of the transistor (1947) and the rise of integrated circuits, embedded systems became more compact and reliable.

1961: The first modern embedded system appeared: the Apollo Guidance Computer (AGC), used in NASA's lunar missions. The AGC incorporated specific software and digital circuits.

Processors began being designed for specific applications, paving the way for modern embedded systems.

I.2.3 Microprocessors and the 1970s Revolution

The invention of microprocessors, such as the Intel 4004 (1971), marked a decisive turning point. These processors made embedded systems more affordable and accessible.

The first microcontrollers emerged, combining a processor, memory, and input/output interfaces on a single chip (e.g., Intel 8048 in 1976).

Industrial applications grew rapidly, particularly in automotive (engine management systems) and consumer electronics.

I.2.4 The Rise of Real-Time Systems (1980s)

The 1980s saw the widespread adoption of embedded systems in critical environments requiring real-time responses, such as aerospace, military, and industrial control.

Real-Time Operating Systems (RTOS) emerged, enabling more efficient task management in these environments.

Key Examples: ABS braking systems in automobiles and the first mobile phones incorporating embedded systems.

I.2.5 Miniaturization and Democratization (1990s)

With advancements in microelectronics, embedded systems became even more compact and affordable.

Applications multiplied in consumer electronics: gaming consoles, digital cameras, MP3 players, and more.

The GPS system emerged as a widely used technology in embedded systems, revolutionizing sectors like transportation and navigation.

I.2.6 The Era of Connectivity and the Internet of Things (2000s)

The integration of embedded systems into connected networks marked the beginning of the Internet of Things (IoT) [4].

Wireless communication technologies (Wi-Fi, Bluetooth, Zigbee) became essential components of embedded systems.

Applications diversified with the advent of smartphones, connected devices (smartwatches, thermostats, etc.), and home automation systems.

I.2.7 Intelligent Embedded Systems (2010s to Today)

With advances in artificial intelligence and machine learning, embedded systems have become intelligent and autonomous.

Key Examples: Autonomous vehicles, drones, intelligent medical devices.

Modern hardware platforms, such as Raspberry Pi, Arduino, and FPGA (Field Programmable Gate Arrays), have made design and prototyping accessible to engineers and researchers.

Embedded systems play a central role in cutting-edge technologies such as 5G, embedded neural networks, and edge computing.

I.3 Specific Features of an Embedded System

Embedded systems are distinguished by several essential characteristics that reflect their design and application in various contexts. Here are the main specific features:

I.3.1 Dedicated Functionality

An embedded system is designed to perform one or more specific tasks, unlike general-purpose computer systems [5]. For example, a cruise control system in a car is designed solely to regulate and maintain the vehicle's speed.

I.3.2 Autonomy and Integration

These systems are often autonomous and integrated into a larger device. They operate independently without requiring constant interaction from the user.

I.3.3 Resource Constraints

Embedded systems typically operate with limited resources:

- Memory: RAM and ROM are often minimized to reduce costs.
- Processor: Computing power is optimized for the specific task.
- Energy: Systems are often designed to be energy-efficient, especially for battery-powered device

I.3.4 Reactivity and Real-Time Constraints

Many embedded systems are deployed in critical applications that require real-time responses with strict deadlines. For example, ABS braking systems must react within a few milliseconds to ensure safety [6].

I.3.5 Reliability and Robustness

Embedded Systems are designed to operate reliably, often in harsh environments with physical constraints such as extreme temperatures, humidity, or vibrations. Their robustness is crucial in ensuring consistent functionality under challenging conditions.

I.3.6 Compact Size and Low Cost

Embedded systems are engineered to be compact, lightweight, and cost-effective, making them suitable for integration into mass-market products.

I.3.7 Hardware-Software Interaction

Embedded systems seamlessly combine hardware components (processors, sensors, actuators) with software elements (firmware, embedded operating systems). This interaction is meticulously optimized to enhance performance and energy efficiency.

I.3.8 Connectivity and Interoperability

In the era of connected devices (IoT), many embedded systems incorporate communication technologies like Wi-Fi, Bluetooth, and Zigbee. These technologies enable devices to interact with other systems or transmit data to remote platforms, fostering interconnectivity and interoperability.

I.3.9 Security and Dependability

Modern embedded systems, especially those used in critical sectors such as healthcare and automotive, integrate advanced security mechanisms. These safeguards protect against cyberattacks while ensuring dependable operations in sensitive applications.

These characteristics make embedded systems indispensable in modern technology, where efficiency, reliability, and performance are paramount.

I.4 Hardware and Software Aspects

An embedded system is a fusion of computer hardware and software, either fixed or programmable, designed to perform specific functions or a set of tasks within a larger system.

Thus, an embedded system consists of hardware and software working together. As shown in Figure I.1.

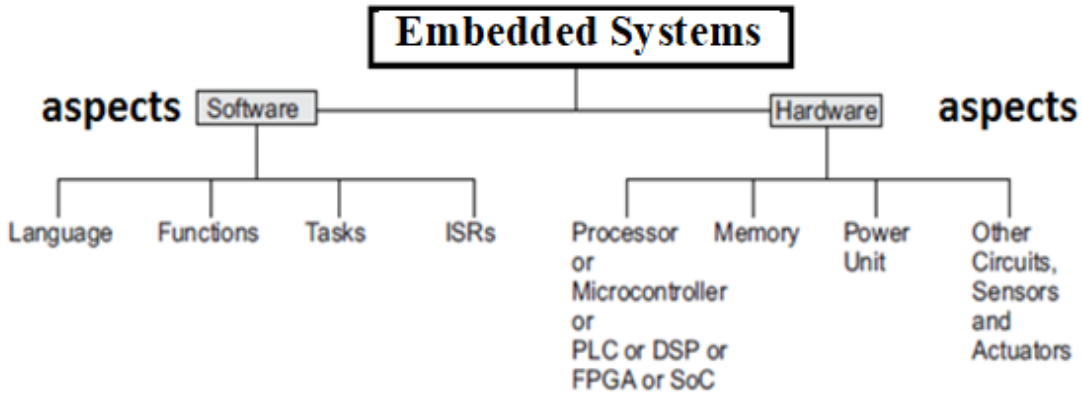

Figure I.1. *Hardware and Software Aspects of Embedded Systems*

I.4.1 Software Aspects

- The software aspects of embedded systems are organized around several key elements:
- Language

  - Embedded systems are programmed using low-level languages (such as assembly) or high-level languages (C, C++, Python, Ada, etc.), tailored to the specific constraints of these systems.
  - These languages help optimize the use of limited resources (memory, processor).

- Functions

  - Functions are reusable blocks of code that perform specific tasks.
  - They simplify the software structure and allow for better modularity.

- Tasks

  - Tasks define processes or subprograms in multitasking or real-time systems.
  - These tasks can be scheduled and executed based on their priority using a Real-Time Operating System (RTOS).

- ISR (Interrupt Service Routines)

  - ISRs are interrupt routines triggered by hardware or software events (such as a signal from a sensor).
  - They enable a quick and efficient response to critical events.

I.4.2 Hardware Aspects

The hardware aspects of embedded systems cover the physical components required to execute the system's functions:

- Processor or Processing Unit

    - Microcontroller: Contains a processor, memory, and integrated I/O, ideal for simple applications.
    - Microprocessor: Provides high computational power but requires external components (memory, interfaces).
    - PLC, DSP, FPGA, SoC:

        - PLC (Programmable Logic Controller): Used in industrial systems.
        - DSP (Digital Signal Processor): Specialized in rapid signal processing.
        - FPGA (Field-Programmable Gate Array): Reconfigurable for specific applications [7].
        - SoC (System on Chip): Integrates multiple components (processor, memory, interfaces) on a single chip.

- Memory
  - ROM: Stores the program or firmware.
  - RAM: Temporary memory used for execution.
  - EEPROM/Flash: Allows for saving modifiable data.

❖ Power Unit

- Ensures the energy supply to the system, either through a battery or an external source.
- Voltage regulators and energy converters ensure a stable power supply.

❖ Other Circuits, Sensors, and Actuators

- Sensors: Measure physical quantities (temperature, pressure, light).
- Actuators: Convert electrical signals into mechanical or physical actions (motors, relays).
- Auxiliary Circuits: ADC/DAC converters, RTC clocks, communication circuits (UART, I2C, SPI, etc.)

I.4.3 Relationship Between Hardware and Software

The synergy between hardware and software is essential for embedded systems:

- The software controls the hardware's functionalities by interacting directly with sensors and actuators.
- The hardware executes the instructions provided by the software to meet specific needs while adhering to real-time and energy consumption constraints.

I.5 Functional Description and Architecture of Embedded Systems

An embedded system can be defined as: *An Autonomous Electronic and Computing System, specialized in a specific task* [8].

This system is a combination of two parts: Electronic System + Computing System, as shown in Figure I.2.

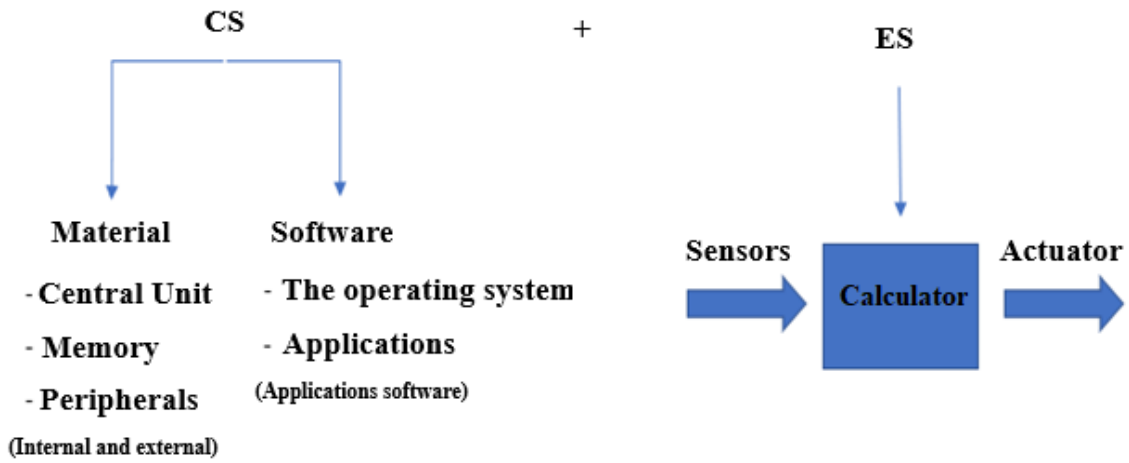

Figure I.2. *Functional description of embedded systems.*

I.5.1 Electronic System (EC)

The electronic system operates on low-current systems (current used to carry information rather than energy).

The electronic system consists of a calculator, which is essentially an electronic component or device that performs calculations. Its inputs are connected to one or more sensors, and its outputs are connected to one or more actuators.

Sensors: Sensors capture the physical signal to be observed and transform it into a signal usable by the calculator.

Actuators: Actuators receive energy to perform an action (work) that modifies the state or behavior of a system.

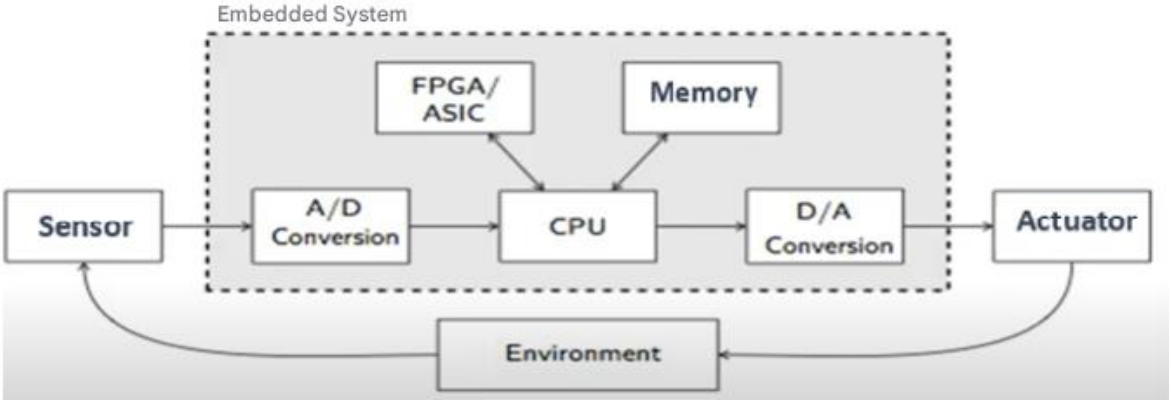

Figure I.3. *Electronic System of Embedded Systems.*

As shown in Figure I.3, The environment where the signal is captured by the sensors will transmit it to an A/D converter (Analog-to-Digital Converter) for digitization so that the signal can be presented digitally to the CPU (Central Processing Unit).

The CPU operates on low-current systems and performs low-level input/output functionalities.

The CPU can interact with memory and/or FPGA/ASIC circuits.

The signal output from the CPU is reconverted into an analog signal (D/A conversion) so that it can be transmitted to the actuators of the embedded system.

I.5.2 Computing System (CS)

The computing system consists of two complementary parts: hardware and software.

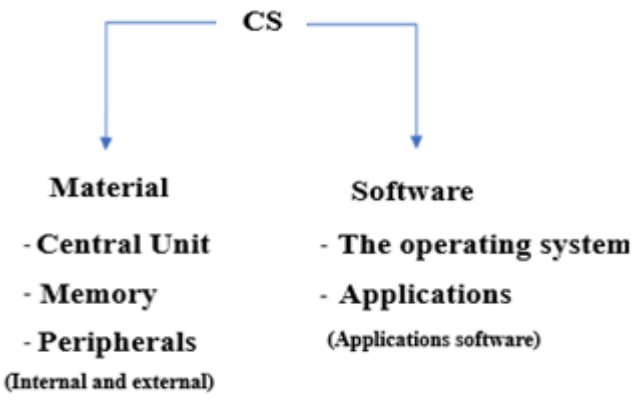

Figure I.4. *Computing System of Embedded Systems.*

❖ Hardware Side

The hardware of embedded systems is based on specific components, selected and optimized to meet particular requirements such as performance, energy consumption, compact size, and robustness. The main essential hardware elements are shown in Figure I.5.

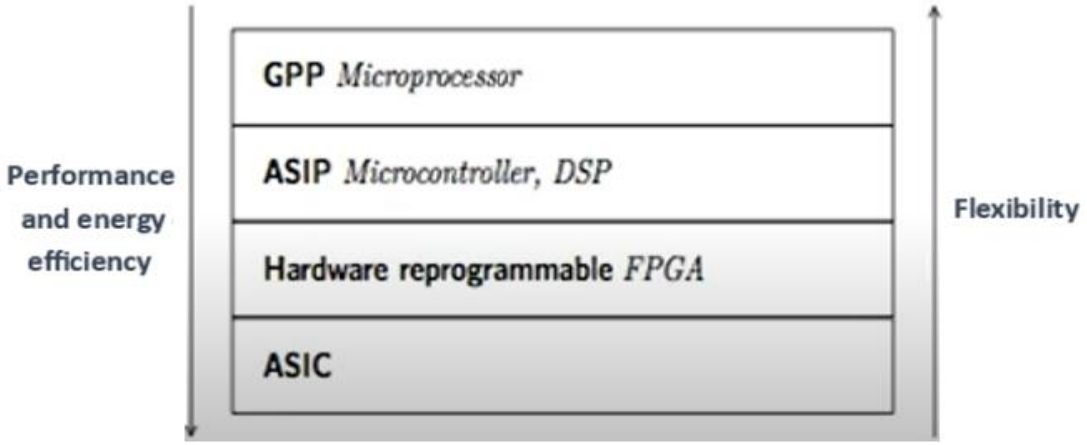

Figure I.5. *Computing System: Hardware Side*.

1. Processing Unit or Processor

The processor is the heart of the embedded system, executing software instructions to ensure overall functionality. Different types of processors can be used depending on the system's constraints and objectives [10]:

- General Purpose Processor (GPP):
  - Versatile processors used in complex embedded applications. They efficiently execute a wide variety of tasks.
  - Example: Intel x86, ARM Cortex-A.

- Microcontroller (MCU):
  - A compact solution combining processor, memory, and peripherals on a single chip. Ideal for systems requiring simple control with limited resources.
  - Example: STM32, PIC, AVR.

- Application-Specific Instruction set Processor (ASIP):
  - A programmable processor optimized for specific tasks, offering a balance between flexibility and performance.
  - Used in fields such as cryptography, multimedia processing, or communications [9].

- Digital Signal Processor (DSP):
  - Designed for fast and efficient digital signal processing, such as audio, video, or communication signals.
  - Example: Texas Instruments TMS320, Qualcomm Hexagon.

- Field Programmable Gate Array (FPGA):
  - A reconfigurable platform capable of massive parallel computation or custom hardware architectures. Used in critical applications like aerospace, telecommunications, or robotics.

- Example: Xilinx, Intel (formerly Altera).

- Application-Specific Integrated Circuit (ASIC):
  - An integrated circuit designed specifically for a particular task or application. It offers optimal performance and low power consumption at the cost of flexibility [11].
  - Example: ASICs for cryptocurrency or medical systems.

2. Memory

Embedded systems use various types of memory to meet specific requirements:

- ROM (Read-Only Memory):
  - Stores the firmware and main program of the system.
  - Example: Flash, EEPROM.
- RAM (Random Access Memory):
  - Temporarily stores data during program execution.
  - Common types include SRAM (Static) and DRAM (Dynamic).
- Non-volatile Memory:
  - Used to store data even without power.
  - Example: NAND Flash for logs or user files.

3. Communication Interfaces

Embedded systems communicate with other devices through suitable hardware interfaces:

- Local Interfaces: UART, SPI, I2C for connecting sensors and actuators.
- Network Interfaces: Ethernet, Wi-Fi, Bluetooth for remote connectivity.
- Industrial Buses: CAN, LIN for automotive and industrial applications.

4. Input/Output Circuits

These circuits allow the embedded system to interact with its environment:

- ADC/DAC Converters: Convert signals between analog and digital forms.
- GPIO Ports (General Purpose Input/Output): Connected to LEDs, buttons, relays, etc.

5. Sensors and Actuators

- Sensors:
  - Measure physical quantities such as temperature, acceleration, or light.
  - Example: MEMS sensors, infrared sensors.

- Actuators:
  - Translate electrical signals into physical actions (motors, valves).

6. Power Unit

Embedded systems require stable and reliable power supplies:

- Rechargeable or non-rechargeable batteries.

- Voltage regulators to stabilize the power supply for sensitive components.

7. Specific Circuits

- Embedded systems require stable and reliable power supplies:
- Rechargeable or non-rechargeable batteries.
- Voltage regulators to stabilize the power supply for sensitive components.

❖ Software Side

In programming, an embedded system equipped with a processor must have a codebase that performs specific tasks. This code needs to be stored in RAM or ROM and be accessible by the CPU to execute instructions sequentially.

- Machine-Level Programming: An embedded system can be directly programmed in machine language using assembly language, which is translated with the help of an assembler.
- High-Level Programming: Alternatively, it can be programmed in a high-level language, which is compiled using a compiler to produce executable code for the system.

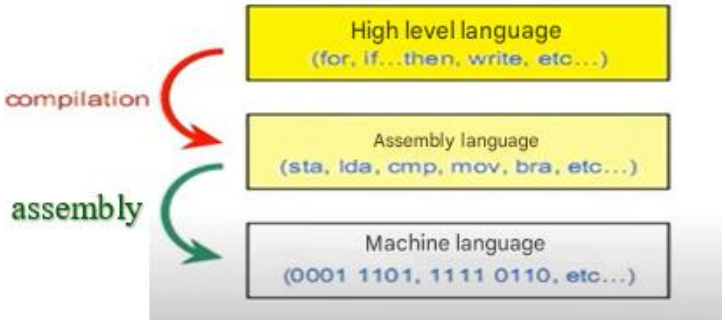

Figure I.6. *Software Side of Embedded Systems.*

Programming Language:

The most suitable programming language for embedded systems, which remains low-level and closely linked to hardware, is the C language. Its efficiency and control over hardware make it ideal for resource-constrained environments.

Operating System (OS):

The operating system in an embedded system is essentially a program executed on the hardware [12]. It acts as an intermediary between the hardware and user applications, simplifying the programming process for embedded systems. By providing abstraction, the OS facilitates hardware management and ensures smoother execution of applications.

I.6 Examples of Embedded Systems

Embedded systems are electronic systems integrated into specific devices to perform one or more dedicated tasks. Below are some examples categorized by application domains:

I.6.1 Automotive

- Anti-lock Braking Systems (ABS): Manage braking to prevent wheel lockup.
- Airbag Systems: Deploy airbags in the event of an accident.
- Engine Control Units (ECU): Optimize engine performance and fuel consumption.
- GPS Navigation Systems: Provide real-time routes and navigation information.

I.6.2 Consumer Electronics

- Smart TVs: Offer advanced features like streaming and voice commands.
- Smartwatches: Track physical activity and provide communication functionalities.
- Digital Cameras: Manage image capture and digital processing.

I.6.3 Healthcare

- Pacemakers: Regulate patients' heart rhythms.
- Insulin Pumps: Deliver precise doses of insulin to diabetics.
- Medical Imaging Equipment: Includes devices like MRI scanners and ultrasound machines.

I.6.4 Aerospace and Space

- Autopilot Systems: Maintain an aircraft's trajectory and stability.
- Telemetry Systems: Monitor and communicate satellite parameters.

I.6.5 Industry

- Programmable Logic Controllers (PLC): Manage manufacturing processes.
- SCADA Systems: Supervise critical infrastructures such as power grids.

I.6.6 Home Automation (Domotics)

- Smart Thermostats: Regulate home temperature based on user preferences.
- Automated Lighting Systems: Adjust lighting based on ambient brightness or predefined schedules.
- Connected Locks: Enable remote, secure access control.

I.6.7 Telecommunications

- Routers and Modems: Provide network connectivity.
- Antenna Management Systems: Optimize signal reception and transmission.

I.6.8 Defense and Security

- Military Drones: Use embedded systems for navigation and autonomous missions.
- Surveillance Radars: Detect and track moving objects.

These systems are often optimized for energy efficiency, reliability, and performance, with dedicated hardware and software tailored to their specific tasks.

Chapter II

Embedded and Real-Time Systems

II.1 Introduction

Time is becoming increasingly crucial in defining services and applications, especially in areas such as industrial computing and telecommunications. High communication link speeds impose processing time constraints, such as managing access to multiple heterogeneous information streams that need synchronization—particularly in multimedia applications.

The introduction of a new service often includes defining the Quality of Service (QoS), which provides insights into the expected temporal behavior of the service (e.g., connection establishment time, request response time).

Achieving global temporal properties stems from the composition of individual behaviors. Time management involves several phases:

- ✓ Enhancing processing performance (greater computational power).
- ✓ Developing lightweight mechanisms (reducing wasted time on non-application-related processes).
- ✓ Innovating methods for predicting temporal behaviors.

II.1.1 Definition of a Real-Time System

A real-time system must always deliver correct responses within predefined deadlines. Failure to meet these deadlines results in system malfunctions. Hence, in a real-time system, not only the result but also the time of delivery is crucial.

In other words, a system is considered real-time if the correctness of its applications depends not only on the output but also on the time when the output is produced [13]. A response delivered after its deadline is invalid, even if its content is correct.

II.2 Memory Management

Memory management refers to the process of controlling and coordinating a system's memory. It spans the hardware, the operating system (OS), and applications:

- ✓ Hardware Level: Involves physical components that store data, such as RAM chips, cache memory, and flash storage (e.g., SSDs).
- ✓ Operating System Level: Includes allocating and reallocating memory blocks to individual programs as user demands change.
- ✓ Application Level: Ensures the availability of sufficient memory for the objects and data structures of each running program at any given time.

When a program requests a block of memory, a part of the memory manager known as the allocator assigns this block to the program.

When a program no longer needs data stored in previously allocated memory blocks, those blocks become available for reallocation. This task can be performed manually (by the programmer) or automatically (by the memory manager).

II.2.1 Types of Memory in Embedded Systems

a. Volatile Memory

- RAM (Random Access Memory): Used to store temporary data during program execution.
  - Common types: SRAM (Static RAM), DRAM (Dynamic RAM).

- Cache: High-speed memory used to reduce access time for frequently used data

b. Non-Volatile Memory

- ROM (Read-Only Memory): Stores firmware or permanent instructions for the computer. Contents are written during manufacturing and cannot be easily modified.
- Flash Memory: Electrically erasable and reprogrammable memory used in USB drives, SSDs, and memory cards.
- Hard Disk Drive (HDD): Magnetic storage device with large capacity. Common in desktops and laptops for storing operating systems, software, and user data.
- Solid-State Drive (SSD): Uses flash memory for faster access speed and reliability compared to HDDs. Widely used in modern systems for improved performance.
- EEPROM (Electrically Erasable Programmable ROM): A type of ROM that can be reprogrammed and erased electronically. Used in embedded systems and microcontrollers.

II.2.2 Memory Management Challenges in Embedded Systems

- Size Constraints: Available memory is often limited.
- Energy Constraints: Memory operations must be efficient to minimize energy consumption.
- Determinism: Real-time embedded systems require predictable memory access times.
- Fragmentation: Memory must be allocated efficiently to avoid unusable spaces.

II.2.3 Memory Management Techniques

a. Static Allocation

- Memory is allocated at compile-time, with fixed variable sizes.
- Advantages: Simple, fast, and predictable.
- Disadvantage: Lacks flexibility.

b. Dynamic Allocation

- Memory is allocated at runtime as needed.
- Uses allocators such as malloc ( ) and free ( ) in C language.
- Risks: Fragmentation, memory leaks, and unpredictability.

c. Segmentation Management

- Divides memory into dedicated segments: stack, heap, and static data.
- Example:
  - Stack: For function calls and local variables.
  - Heap: For dynamic allocation.
  - Global Data: For global and static variables.

II.3 Concurrency Management

In many cases, it is necessary to execute multiple instructions simultaneously (e.g., in FPGA systems) for several reasons, including:

- ✓ Reducing Latency: For example, if a program controls keyboards or slow peripherals, the entire system should not be blocked while waiting for infrequent events.
- ✓ Utilizing Multiple Processors: Certain machines are equipped with multiple processing units, which need to be efficiently programmed.
- ✓ Handling Multiple Tasks: An embedded system must be capable of managing the concurrent execution of multiple tasks.

To manage concurrency, fundamental concepts must be defined, such as processes, semaphores, and conditions.

II.3.1 Processes

A process is an instance of a program in execution. It is a fundamental unit of execution in a multitasking operating system [12].

a. Key Characteristics

- Memory Space: Each process has its own memory space, including code, data, and stack.
- Isolation: Processes do not directly share their data, ensuring their independence.
- Communication: Inter-process communication (IPC) is required to exchange data between processes.

b. Types of Processes

- Lightweight Processes (Threads): Share the same memory space but perform concurrent tasks.
- Independent Processes: Operate in isolation and require explicit mechanisms to interact.

c. Example

In an embedded system, one process might handle network communication while another manages sensor data.

II.3.2 Semaphores

A semaphore is a variable or structure used to control concurrent access to a shared resource. It helps synchronize processes or threads.

a. Types of Semaphores

Binary Semaphore:

- Has two states: 0 (unavailable) and 1 (available).
- Similar to a mutex.

Counting Semaphore:
- Manages simultaneous access to a resource with multiple available instances.
- Example: Managing access to a queue with multiple available slots.

b. Key Functions

- Wait ( ) (P): Decreases the value of the semaphore. If the resource is not available (value = 0), the process waits.
- Signal ( ) (V): Increases the value of the semaphore to indicate that a resource has been released.

b. Example

A binary semaphore can be used to protect access to a sensor in an embedded system:

1. A task gains exclusive access by calling wait ( ).
2. Once access is complete, it releases the sensor by calling signal ( ).

II.3.3 Conditions (Condition Variables)

Condition variables allow a thread to suspend execution until a specific condition is met. They are used in conjunction with mutexes to synchronize threads.

a. Key Functions

1. Wait (mutex):
   - The thread goes into a waiting state and releases the associated lock (mutex).
   - It will be awakened when another thread signals the condition.
2. Signal ( ):
   - Wakes up one thread waiting on the condition.
3. Broadcast ( ):
   - Wakes up all threads waiting on the condition.

b. Example

In a producer-consumer system:

- The producer generates data and uses signal ( ) to notify the consumer that the data is ready.
- The consumer waits using wait ( ) until data becomes available.

Concept	**Main Function**	**Example of Application**
Process	Isolated unit of execution.	Parallel management of sensors and networks.
Semaphores	Control access to shared resources.	Exclusive access to a device.
Conditions	Synchronization on specific events.	Coordination between producers and consumers.

Tableau II.1 *Summary of Fundamental Concepts for Concurrency Management*

II.4 Linux for Embedded Systems

Linux is a popular choice for embedded systems due to its flexibility, stability, and large user and developer base. Below is a simplified explanation of Linux in the context of embedded systems:

- ✓ Linux Kernel: The Linux kernel is the core of the operating system. It manages hardware resources such as the processor, memory, devices, etc. For embedded systems, the kernel is often customized to meet the specific needs of the device. Manufacturers can configure the Linux kernel to include only the required features, reducing the image size and optimizing performance.
- ✓ Device Drivers: Linux supports a wide range of hardware devices through its device driver system. For embedded systems, manufacturers often need to develop or adapt specific drivers for their hardware to ensure compatibility with Linux.
- ✓ Power Management: Embedded systems often have strict power consumption constraints. Linux provides power management mechanisms to optimize system energy usage, such as hibernation, reducing processor frequency, etc.
- ✓ File Systems: Linux supports various file systems, allowing developers to use the one best suited for their application's needs. For embedded systems, lightweight and optimized file systems can be used to save storage space and improve performance.
- ✓ Security: Linux offers several security features to protect embedded systems from potential threats. This includes features such as permission management and kernel security mechanisms.
- ✓ Development Tools: Linux has a wide range of development tools, including compilers, debuggers, profilers, etc., which can be used to develop, debug, and optimize applications for embedded systems.

II.4.1 Advantages

1. Open Source: Free, modifiable, with a large developer community.
2. Portability: Compatible with numerous architectures (ARM, x86, MIPS, etc.).
3. Feature-Rich: Supports networking, file systems, hardware drivers, and more.
4. Real-Time Support: Extensions like PREEMPT-RT or Xenomai can add real-time capabilities.
5. Mature Ecosystem: Includes numerous tools, libraries, and frameworks to accelerate development.

II.4.2 Disadvantages

1. Memory Size: Linux requires more memory compared to a dedicated RTOS.
2. Complexity: It may be too resource-intensive for highly constrained systems.
3. Boot Time: Startup time can be long, although optimizations are possible.

II.5 Presentation of Embedded Real-Time Systems

Before discussing embedded real-time systems, embedded systems can be classified according to several types [16]:

- ✓ Transformational System: A computing activity where the system reads its data and inputs during startup, provides its outputs, and then terminates.
- ✓ Interactive System: A system that interacts almost continuously with its environment, even after initialization. The system's reaction is determined by received events and its current state (based

on past events and reactions). The interaction rate is determined by the system, not the environment.

- ✓ Reactive or Real-Time System: A system that interacts continuously with its environment, including after initialization. The system's reaction is determined by received events and its current state (based on past events and reactions). However, the rate of interaction is determined by the environment, not the system.

A real-time system is not simply a "fast" system, but rather a system that meets time constraints. These time constraints depend on the application and the environment, whereas speed depends on the technology used, such as the processor.

Real-time systems can be classified into three categories [17]:

II.5.1 Hard Real-Time

A Hard Real-Time system refers to a system where events that are processed too late or lost result in catastrophic consequences for the system's operation (loss of critical information, crashes, etc.).

Systems with strict time constraints only tolerate rigid time management to maintain the integrity of the service provided.

In a hard real-time system, a delay in obtaining the result renders it useless, as shown in Figure II.1.

Figure II.1. *Hard Real-Time Systems* [18].

- ✓ In a hard real-time system, time constraints are extremely strict.
- ✓ Each task must be completed within a fixed and precise deadline [19].
- ✓ These systems are often used in applications where safety and reliability are critical, such as in aircraft control systems or anti-lock braking systems in automobiles.

Example of hard real-time systems: Air traffic control (Control Tower), missile guidance systems.

II.5.2 Firm Real-Time

A Firm Real-Time system refers to a flexible real-time system where occasional deadline misses may occur. When events are processed too late or lost, the consequences are significant enough to affect the system's proper functioning. No guaranteed average percentage of CPU time usage is ensured.

Firm real-time systems have strict time constraints but allow for some tolerance to occasional delays. In a firm real-time system, a delay is useless once the deadline has passed [20], as shown in Fig II.2.

Figure II.2. *Firm Real-Time Systems* [18].

- ✓ The majority of tasks must be completed within precise deadlines, but a few rare delays can be tolerated without causing significant problems.
- ✓ Delays may be inconvenient but are not critical.
- ✓ Real-time multimedia processing systems, such as video streaming, can be examples of firm real-time applications.

Example of firm real-time systems: Stock market transactions, video projection (loss of a few image frames).

II.5.3 Soft Real-Time

A Soft Real-Time system refers to a system where events that are processed too late or lost do not have catastrophic consequences for the system's operation. Only an average percentage of CPU time usage is guaranteed.

Soft real-time systems have flexible time constraints and can tolerate variations in data processing. This is often referred to as Quality of Service (QoS) [21].

Soft real-time systems have time constraints, but they are much less strict than in hard or firm real-time systems.

In a soft real-time system, a delay in obtaining the result is not dramatic but gradually reduces its relevance, as shown in Figure II.3.

Figure II.3. *Soft Real-Time Systems [18].*

- ✓ Tasks generally need to be completed within a reasonable time frame, but occasional delays are acceptable and do not critically affect the system.
- ✓ System performance may decrease if deadlines are frequently missed, but this does not necessarily lead to a system failure.
- ✓ Systems for scheduling online meetings or real-time data processing applications in social networks can be examples of soft real-time applications.

Example of soft real-time systems: Data acquisition systems for display, embedded software in mobile phones, iPods.

II.6 Structure and Operation of Embedded Real-Time Systems

II.6.1 Task Management

A task in an embedded system is a unit of work that must be executed to perform a specific function or operation within the system. In embedded systems, tasks are typically programs or functions designed to run autonomously, i.e., without user intervention or the involvement of other processes in the system.

Tasks in embedded systems are often divided into two categories: real-time tasks and non-real-time tasks. Real-time tasks have strict time requirements and must be executed within precise deadlines to ensure the system operates correctly. Non-real-time tasks have less stringent time requirements and can be executed more flexibly [22].

Task management in embedded systems is a crucial aspect to ensure the system functions correctly. Embedded systems often use task scheduling techniques to allocate resources effectively and ensure that real-time tasks are executed on time.

Task scheduling techniques include preemptive and cooperative scheduling, as well as the use of different scheduling algorithms to manage task priorities based on their requirements.

II.6.2 Cooperative Scheduling

In a system with cooperative scheduling (also called "cooperative multitasking"), each task is responsible for voluntarily yielding control to other tasks at regular intervals.

Cooperative tasks must explicitly cooperate by releasing the processor after completing their work or when they reach a synchronization point.

Cooperative scheduling is often used in systems where tasks are relatively simple and well-behaved, and developers can trust other tasks to cooperate correctly.

However, a major disadvantage of cooperative scheduling is that a single poorly written or blocked task can monopolize the processor, causing a system slowdown or freeze.

II.6.3 Preemptive Scheduling

In a system with preemptive scheduling, the operating system can interrupt the execution of a task at any time to run another task with higher priority.

Preemptive scheduling is commonly used in real-time and multitasking systems where it is important to ensure that critical tasks are executed within strict deadlines.

A common example of preemptive scheduling is priority scheduling, where tasks are assigned priority levels, and higher-priority tasks can interrupt lower-priority tasks [23][24].

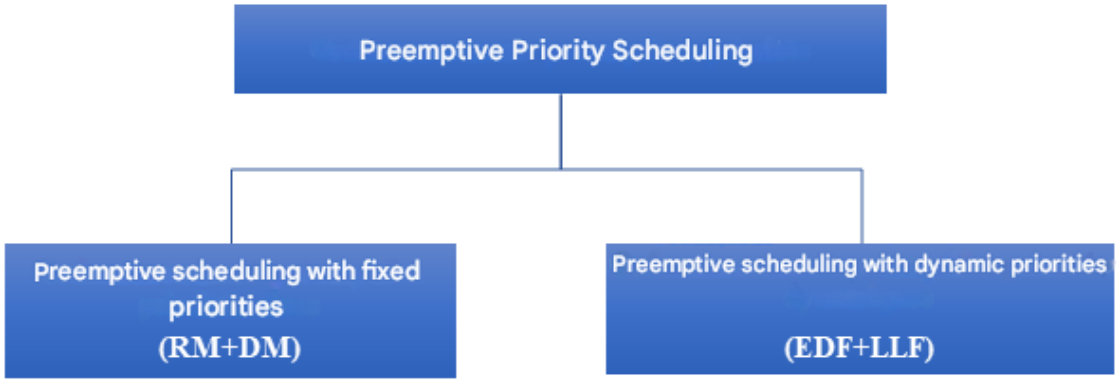

Figure II.4. *Preemptive Priority Scheduling.*

II.6.3.1 Fixed Priority Preemptive Scheduling (RM+DM)

Fixed priority preemptive scheduling is a technique used in operating systems to schedule and order tasks based on their priority. This method uses two different scheduling algorithms:

- ✓ The first is called Rate Monotonic (RM), which assigns priorities to tasks based on their period or frequency. This means that tasks with the shortest periods have the highest priority.
- ✓ The second algorithm is called Deadline Monotonic (DM), which assigns priorities based on the task's deadline. This means that tasks with the closest deadlines have the highest priority.

In fixed priority preemptive scheduling (RM+DM), the operating system combines these two algorithms to assign priorities to tasks based on their period and deadline. Therefore, tasks with short periods and close deadlines are given the highest priority.

Fixed priority preemptive scheduling (RM+DM) is used in real-time operating systems to ensure that the most critical tasks are executed first and that the deadlines of tasks are met. However, this method can lead to inefficient use of system resources if priorities are poorly chosen and can cause delays if tasks are not properly planned.

- ✓ Fixed Priority Scheduling (Rate Monotonic - RM)

- ❖ Principle: A task is prioritized the more its activation period (T_i) is small.

- ❖ Each task τ_i is described by the following parameters:

1. Execution Time (C_i):
    - the processor time required to fully execute the task.
2. Period (T_i):
    - A fixed interval between two consecutive activations of the task.
3. Deadline (D_i):
    - The maximum time allowed to complete the task after its activation.
    - In RM, D_i is often equal to T_i (strict deadline).
4. Activation Time
    - The instant when the task is ready to be executed for the first time.

❖ Basic Assumptions in RM (Rate Monotonic)

1. Tasks are periodic and independent.
2. Deadlines (D_i) are equal to periods (T_i).
3. The execution time (C_i) of each task is constant and known.
4. Tasks are executed on a single processor.
5. The system supports preemptive scheduling.
6. No task blocks the execution of another by shared resources [25]

❖ Feasibility Analysis (Processor Utilization)

For the RM scheduling to guarantee that deadlines are met, the sum of the processor utilizations (U) of the tasks must satisfy the following condition:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot \left(2^{\frac{1}{n}} - 1\right) \qquad (II-1)$$

- n: the number of tasks.
- $2^{\frac{1}{n}} - 1$: is a theoretical upper bound for n tasks.

For n→∞, this limit tends to approximately **69,3 %** (or ln (2)).

❖ Example of RM Scheduling:

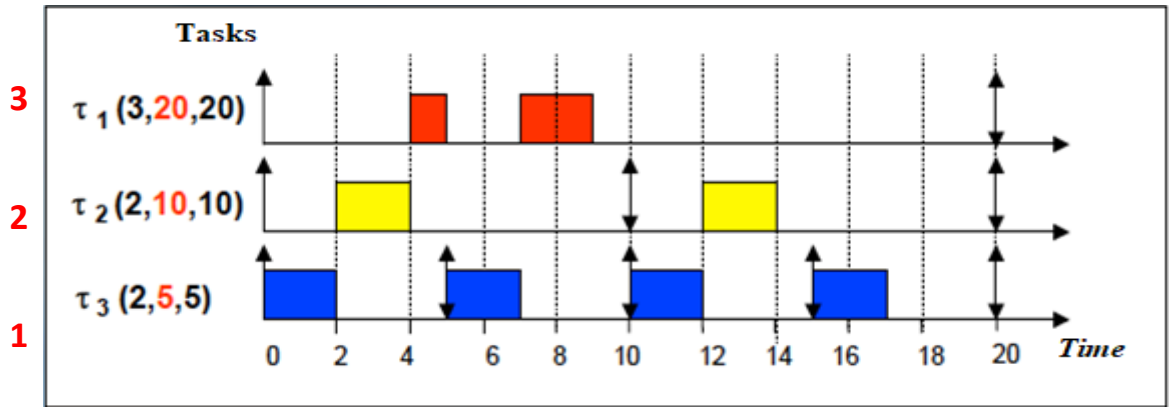

Figure II.5. *RM Scheduling.*

Consider Three Tasks:

1. τ_1 : C_1=3, T_1=20
2. τ_2 : C_2=2, T_2=10
3. τ_3 : C_3=2, T_3=5

Priorities (based on T_i) :

- τ_1 (lowest priority)
- τ_2
- τ_3 (highest priority)

Processor Utilization Calculation:

$$n \cdot \left(2^{\frac{1}{n}} - 1\right) = 3 \cdot \left(2^{\frac{1}{3}} - 1\right) = 0{,}779$$

$$U = \frac{c_1}{T_1} + \frac{c_2}{T_2} + \frac{c_3}{T_3} = \frac{3}{20} + \frac{2}{10} + \frac{2}{5} = 0{,}15 + 0{,}2 + 0{,}4 = 0{,}75$$

- Here, U < 0.779 for n=3 so the scheduling guarantees that all deadlines will be respected.

❖ Advantages of RM Scheduling:

- Simplicity: Static priorities, easy to implement.
- Clear mathematical analysis: Simple utilization condition to check feasibility.
- Predictability: Critical tasks (with shorter periods) are always executed first.

❖ Limitations of RM Scheduling:

- Processor utilization constraint: Does not guarantee optimal processor usage for high loads (when U>69.3%).
- Strict assumptions: Difficult to handle aperiodic tasks or dependencies between tasks.
- Preemption penalty: Frequent interrupts can introduce context-switching overhead.

❖ Applications of RM Scheduling:

- Real-time embedded control: Examples include Anti-lock Braking Systems (ABS) or flight control systems.
- Critical monitoring systems: Such as medical equipment or factory control systems.
- Scheduling in environments where periodic tasks predominate.

The RM Algorithm remains a fundamental reference in managing periodic real-time systems due to its simplicity and predictability.

✓ Fixed-Priority Scheduling (Static) - Deadline Monotonic (DM)

❖ Principle**:** A task has higher priority the smaller its critical deadline (D_i) is.
❖ The deadline is the latest time by which a task must be completed.
❖ Deadline is also called the due date or critical deadline [22].
❖ Basic Assumptions in DM:
- Tasks are independent.
- The deadlines (D_i) are less than or equal to the periods ($D_i \leq T_i$).

- The execution time (C_i) of each task is constant and known.
- The tasks are executed on a single processor.
- The system supports preemptive scheduling.

❖ Feasibility Analysis:

For DM scheduling to guarantee respect of deadlines, the sum of processor utilizations (U) must satisfy the following condition:

Feasibility Condition:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \qquad (II-2)$$

Necessary Condition:

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n \cdot \left(2^{\frac{1}{n}} - 1\right) \qquad (II-3)$$

This condition is identical to the one for RM. However, DM is more efficient when D_i<T_i, as RM does not consider deadlines that are independent of periods.

❖ Example of DM Scheduling

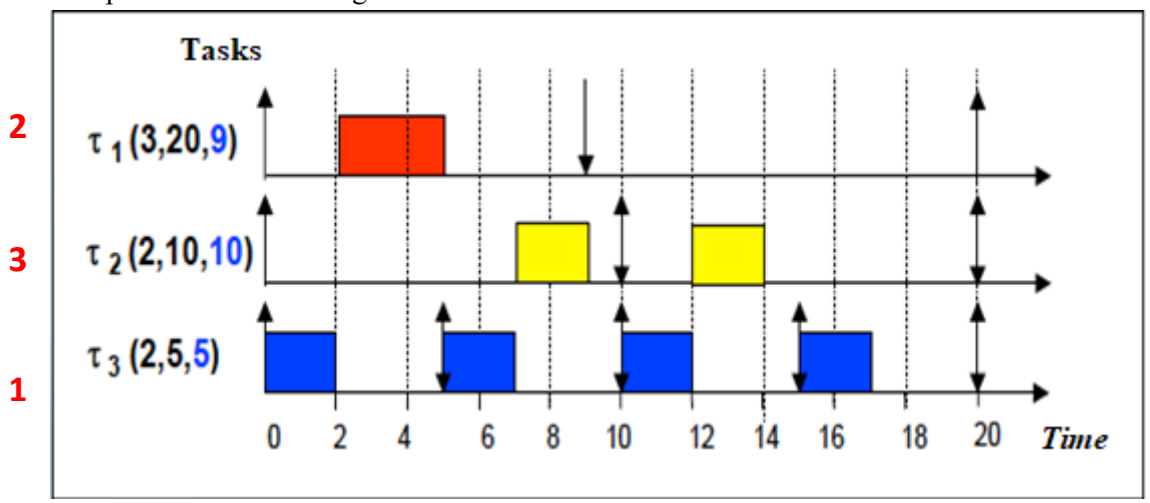

Figure II.6. *DM Scheduling.*

Consider three tasks:

2. τ_1: C_1=3, T_1=20, D_1=9
3. τ_2: C_2=2, T_2=10, D_2=10
1. τ_3: C_3=2, T_3=5, D_3=5

Priorities (based on T_i):

- τ_1
- τ_2 (lowest priority)
- τ_3 (highest priority)

Calculation of Processor Utilization:

$$n \cdot \left(2^{\frac{1}{n}} - 1\right) = 3 \cdot \left(2^{\frac{1}{3}} - 1\right) = 0{,}779$$
$$U = \frac{c_1}{D_1} + \frac{c_2}{D_2} + \frac{c_3}{D_3} = \frac{3}{9} + \frac{2}{10} + \frac{2}{5} = 0{,}33 + 0{,}2 + 0{,}4 = 1{,}03$$

- Here, U > 0.779 for n=3, so the scheduling may not guarantee that all deadlines are met.

❖ Comparison with RM

- In RM, priorities would have been assigned according to T_i : $\boldsymbol{\tau}_3 > \boldsymbol{\tau}_1 > \boldsymbol{\tau}_2$
- DM better reflects the actual constraints when deadlines differ from periods.

❖ Advantages of DM

1. Increased Flexibility:
   - DM is more suitable for systems where deadlines differ from periods ($D_i < T_i$).
2. Optimization of Critical Deadlines:
   - Tasks with stricter timing constraints are prioritized.
3. Simplicity of Implementation:
   - Priorities are fixed and pre-determined.

❖ Limitations of DM

1. Strict Assumptions:
   - DM assumes that all tasks are independent and that deadlines are known in advance.
2. High Concurrency:
   - Frequent preemptions due to high-priority tasks can result in context-switching overhead.
3. Feasibility Conditions:
   - Tasks are not always schedulable if processor utilization exceeds the theoretical limit ($n \cdot (2^{1/n} - 1)$).

❖ Applications

- Aerospace: Flight control systems where some deadlines are more critical than periods.
- Medical Systems: Prioritization of vital tasks with strict deadlines.
- Robotics: Systems where certain actions must be completed before specific deadlines.

II.6.3.2 Preemptive Scheduling with Dynamic Priorities (EDF + LLF)

Preemptive scheduling with dynamic priorities is a scheduling technique used in operating systems to determine the order of task execution in a multitasking system. This technique uses dynamic priorities to give precedence to tasks with shorter deadlines or tasks that require more processor time.

✓ The EDF+LLF scheduling combines the advantages of both techniques by using:
✓ EDF scheduling for tasks whose deadline is shorter than their laxity.
✓ LLF scheduling for tasks whose deadline is longer than their laxity.

This technique ensures that tasks are executed in an order that respects the deadlines for critical tasks, while minimizing the risks of deadline overruns for other tasks.

EDF Scheduling (Earliest Deadline First) is a preemptive dynamic priority technique that uses task deadlines to determine their execution order. According to this technique, the task with the nearest deadline is given the highest priority [16]. EDF scheduling ensures that tasks will be executed in an order that respects their deadlines.

LLF Scheduling (Least Laxity First) is another preemptive dynamic priority technique that uses the concept of laxity to determine the execution order. Laxity is defined as the difference between the time remaining before the deadline and the execution time required to complete the task [26]. According to this technique, the task with the least laxity is given the highest priority [27]. LLF scheduling ensures that tasks are executed in an order that minimizes the risk of missing deadlines.

✓ Preemptive Scheduling with Dynamic Priorities: Earliest Deadline First (EDF)

❖ Principle: At every moment, the highest-priority task is the one whose absolute deadline d_i is the closest.

1. Priority based on absolute deadline (d_i):

- A specific time when the task must be completed

$$d_i = t_{activation} + D_i \qquad (II-5)$$

  - At any given moment, the task with the closest absolute deadline is executed first [28].
  - The priority of tasks is dynamically recalculated when a new task becomes ready or when a currently executing task finishes.

2. Preemptive scheduling:

- A task that becomes ready with a closer deadline can preempt a task that is currently executing.

❖ Feasibility Conditions

EDF is optimal for single-processor systems, provided that the total processor utilization (U) is less than or equal to 100%.

Feasibility Condition:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \qquad (II-2)$$

Necessary Condition:

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \qquad (II-4)$$

- If $U \leq 1$, EDF guarantees that all tasks will meet their deadlines
- If $U > 1$, some tasks may not be executed on time, regardless of the algorithm.

❖ Example of EDF Scheduling

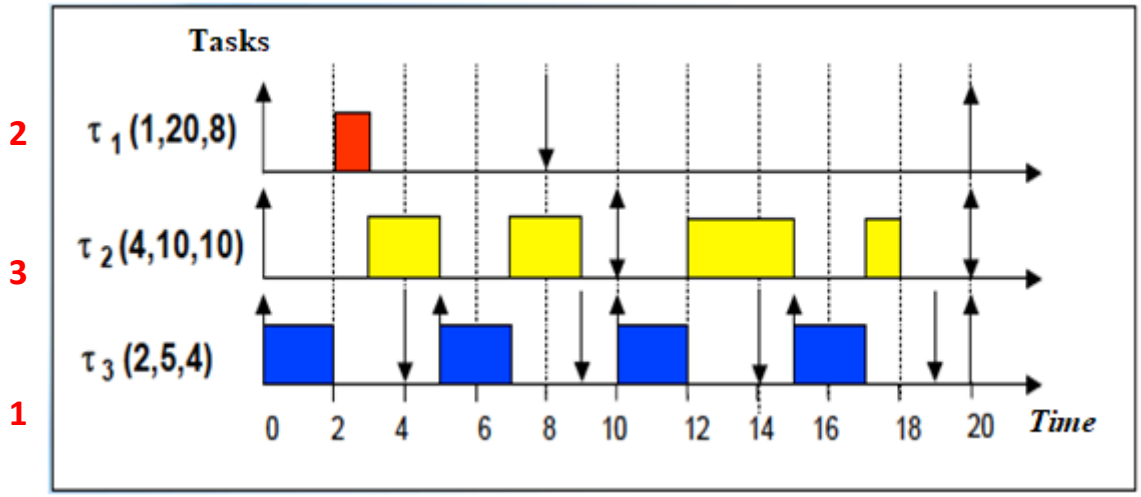

Figure II.7. *EDF Scheduling.*

Consider three periodic tasks:

2. τ_1: C_1=1, T_1=20, D_1=8
3. τ_2: C_2=4, T_2=10, D_2=10
1. τ_3: C_3=2, T_3=5, D_3=4

Priorities (based on T_i):

- τ_1
- τ_2 (lowest priority)
- τ_3 (highest priority)

Processor Utilization Calculation:

$$U = \frac{c_1}{D_1} + \frac{c_2}{D_2} + \frac{c_3}{D_3} = \frac{1}{8} + \frac{4}{10} + \frac{2}{4} = 0{,}125 + 0{,}4 + 0{,}5 = 1{,}025$$

- Here, U < 1 for n=3, so the scheduling may not guarantee that all deadlines will be met.

❖ Advantages of EDF:

1. Optimality on a single processor: EDF can use up to 100% of processor utilization, unlike RM or DM, which are limited to approximately 69.3% for a large number of tasks.

2. Adaptability: EDF adapts to tasks with varying and unforeseen deadlines.

3. Conceptual Simplicity: The rule "execute the task with the closest deadline" is easy to understand and implement.

❖ Limitations of EDF:

1. Implementation Complexity: It requires dynamic priority management, which can incur real-time overhead.
2. Sensitivity to Overloads: If $U > 1$, the system may become unstable, and significant deadlines may be missed.
3. Preemption Costs: Preemptive scheduling can lead to context-switching and resource overheads.

❖ Applications of EDF

1. Industrial Control Systems: Prioritizing critical tasks with approaching deadlines.

2. Critical Embedded Systems: Examples: Airbag systems, autopilot systems.

3. Real-Time Multimedia: Audio/video playback where strict deadlines must be respected.

✓ Dynamic Priority Scheduling: Least Laxity First (LLF)

❖ Principle: Least Laxity First (LLF) is a dynamic scheduling algorithm used in real-time systems. It assigns priorities based on a task's laxity, which is the "free" time remaining before the task misses its deadline.

1. Laxity Calculation: The laxity (L_i) of a task τ_i is defined as [1-3]:

$$L_i = d_i - t - C_i \qquad (II-6)$$

Where:

- d_i: Absolute deadline of the task.
- t: Current time.
- Ci: Remaining execution time for completing the task.

Interpretation: The laxity represents the available time after completing the task execution to meet its deadline.

2. Priority Rule:
   - The task with the least laxity (i.e., the least "free" time) is executed first.
   - A task with zero or negative laxity is critical and must be executed immediately to meet its deadline.
3. Preemptive Scheduling:
   - If the laxity of a task becomes smaller than that of the currently executing task, the processor immediately switches to this task.

❖ Task Characteristics in LLF

Each task τ_i is characterized by [1-3]:

1. Execution time (C_i): Time required to complete the task.
2. Relative deadline (D_i): The maximum time allowed to complete the task after its activation.
3. Absolute deadline ($d_i = t_{activation} + D_i$): The specific moment when the task must be completed.
4. Period (T_i) (for periodic tasks): Fixed interval between two successive activations.

❖ Example of LLF Scheduling

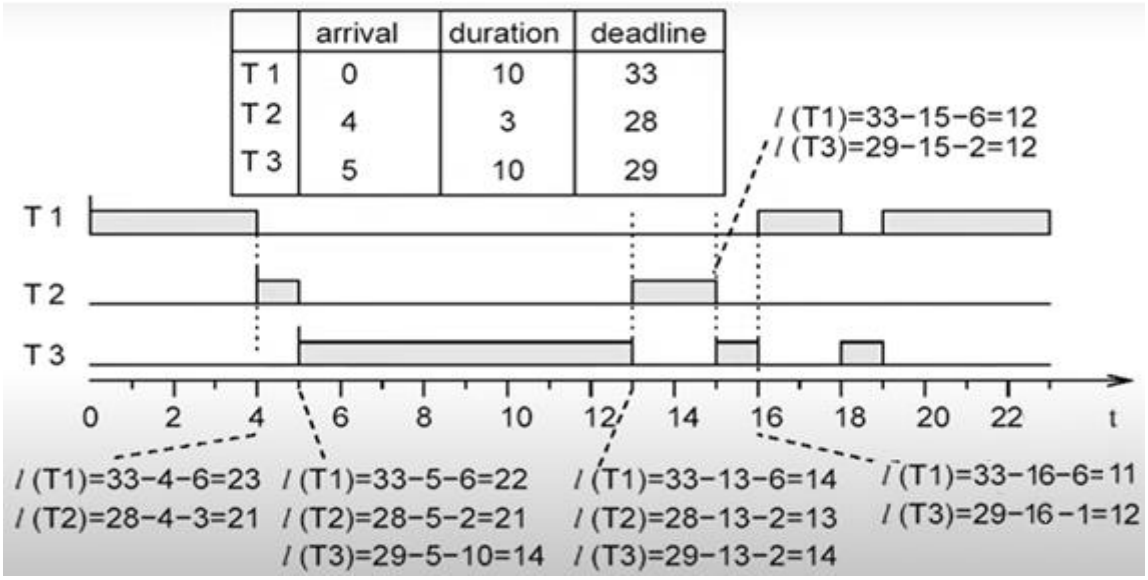

Figure II.8. *LLF Scheduling.*

Consider three periodic tasks:

1. τ_1 : C_1=10, à t=0, D_1=33
2. τ_2 : C_2=3, à t=4, D_2=28.
3. τ_3 : C_3=10, à t=5, D_3=29.

Scheduling Simulation:

1. Initialization (t=0):
   - Only taskτ1 is ready, with its initial laxity until (t=4)

2. At t=4: task τ_2 becomes ready
   1. L_1=33−4−6=23.
   2. L_2=28−4−3=21.
   3. task τ_2 (with minimal laxity) is executed first.

3. At t=5: task τ3 also becomes ready
   1. L_1=33−5−6=22.
   2. L_2=28−5−2=21.
   3. L_3=29-5-10=14.
   4. task τ_3 (with minimal laxity) is executed first.
4. At t=13:
   1. L_1=33−13−6=14.
   2. L_2=28−13−2=13.
   3. L_3=29-13-2=14.
   4. task τ_2 (with minimal laxity) is executed first.
   5. τ_2 finishes at t=15.
5. At t=15:
   1. L_1=33−15−6=12.
   2. L_3=29-15-2=12.
   3. task τ_3 is executed first (equal laxity, so task with smaller deadline is executed).
6. At t=19:
   1. τ_3 finishes at t=19.
   2. τ_1 finishes at t=23.

❖ Advantages of LLF

1. Precision in deadline management:
    - LLF is particularly effective in avoiding deadline overruns because it prioritizes the most urgent tasks in terms of remaining time.
2. Optimality on a single processor:
    - Like EDF, LLF can guarantee deadline compliance if total processor utilization (U) is less than or equal to 100%.
3. Flexibility:
    - LLF can handle tasks with irregular or non-periodic deadlines.

❖ Limitations of LLF

1. Computation Overhead:
    - Frequent recalculation of laxity at each time unit or event (new task ready, task completion) can be costly in terms of processor time.

2. Frequent Preemptions:
    - Since laxity can change rapidly, LLF often causes frequent task preemptions, increasing context-switching costs.

3. Sensitivity to Estimation Errors:
    - If execution times (C_i) are poorly estimated, calculated laxities may be incorrect, compromising scheduling.

❖ Applications of LLF

1. Critical Real-Time Systems:
    - Examples: Air traffic control, management of sensitive industrial processes.
2. Systems with Strict Deadlines:
    - LLF ensures that the most urgent tasks receive the necessary attention.
3. Complex Embedded Systems:
    - When precise control over priorities is needed for systems with highly variable deadlines.

II.6.3.3 Comparison Between Different Preemptive Scheduling Systems

The table II.2 below compares the two preemptive scheduling systems. EDF/LLF is often preferred for systems requiring maximum resource utilization and adaptability, while RM/DM are simpler to implement for moderate loads.

Criterion	EDF/LLF	RM/DM
Priorities	Dynamic	Static
Max Utilization (U)	100%	Around 69.3% (for RM)
System Overhead	Higher (dynamic management)	Lower (fixed priorities)
Flexibility	High	Limited
Complexity	Medium to High	Low

Tableau II.2 *Comparison Between RM/DM and EDF/LLF*

- ✓ RM/DM: Simple, efficient, but limited in processor utilization. Ideal for embedded systems with fixed constraints.
- ✓ EDF: Flexible and optimal on a single processor if the load is manageable, but sensitive to overloads.
- ✓ LLF: Provides precise deadline management, but at the cost of frequent preemptions and increased complexity.

Scenario	**Recommended Algorithm**
Periodic tasks with fixed constraints	RM or DM
Non-periodic tasks or varying deadlines	EDF
Highly critical tasks with strict deadlines	LLF
Predictable and stable system load	RM or DM
Variable or unpredictable system load	EDF
Maximized processor utilization	EDF

Tableau II.3 *Comparison of Real-Time Scheduling Algorithms*

Chapter III

Embedded Processor Architecture

III.1 Key Architectural Concepts

An embedded computer system differs from general-purpose systems in that it is typically dedicated to performing a specific task, such as engine control in a car. In contrast, general-purpose computers handle multiple functions, including word processing, web browsing, and gaming.

Embedded systems generally rely on low-power microprocessors or microcontrollers, where the software component is partially or fully embedded in hardware, often stored in non-volatile memory such as ROM, EPROM, EEPROM, or FLASH. This is commonly referred to as firmware.

The processor architecture, or the arrangement of a computer's elements, is optimized to suit the specific task of the embedded system. In these systems, the microprocessor plays a central role by performing computations and transferring data between memory locations.

Embedded systems can range from simple devices, such as toy controllers or toasters, to complex systems like factory management units. Each level of complexity corresponds to different processor models: simple systems might use 8-bit microprocessors, while more advanced ones employ 64-bit processors. The number of bits in a processor affects its speed, cost, and data-handling capacity.

The architecture of embedded systems is built upon fundamental concepts that define their structure, operation, and interaction with their environment. The key concepts include:

III.1.1 Hardware Architecture

Main Components

- ✓ Microprocessor/Microcontroller: The core of the system, responsible for executing instructions. Examples include ARM Cortex, AVR, and PIC.

Memory Units:

- ✓ ROM: For storing firmware.
- ✓ RAM: Volatile memory used for processing data.
- ✓ EEPROM/Flash: Non-volatile memory used for configurations or updates.

Communication Interfaces:

- ✓ Communication buses (I2C, SPI, UART, CAN).
- ✓ Network interfaces (Ethernet, Wi-Fi, Bluetooth).

Input/Output Peripherals:

- ✓ Sensors (input) and actuators (output).
- ✓ User interfaces (buttons, screens, LEDs).

Common Architectures:

- ✓ Von Neumann: Shared memory for instructions and data.

The Von Neumann architecture was one of the earliest architectures in which both the program and data are stored in the same memory and accessed via the same bus. Consequently, the CPU performs one operation at a time, as it fetches instructions from memory.

- The memory contains both data and instructions.
- The central processing unit (CPU) fetches instructions from the memory.
- A set of registers assists the CPU:
  - Program Counter (PC): Keeps track of the address of the next instruction.
  - Instruction Register (IR): Holds the instruction currently being executed.
  - Stack Pointer (SP): Points to the top of the stack.
  - General-purpose registers: For example, the accumulator (A).

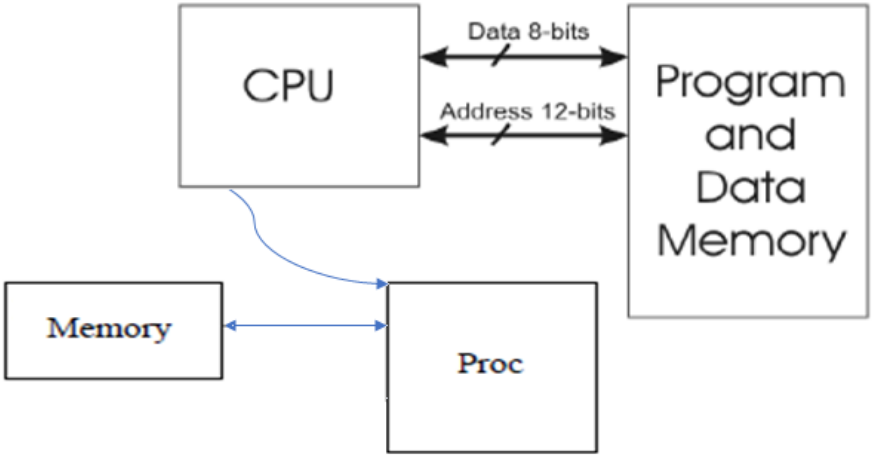

Figure III.1. *Von Neumann Architecture.*

The Program Counter (PC) is a special register in the processor that contains the address of the next instruction to be executed. With each clock cycle, the PC is incremented to point to the next instruction in the program sequence.

The Instruction Register (IR) is a processor register that holds the instruction currently being executed. The IR is loaded from memory using the address provided by the PC. The IR is then decoded to determine the operation to perform and the associated data [29].

The Stack Pointer (SP) is a register that points to the stack location in memory. The stack is a memory data structure used for temporarily storing data or return addresses for subroutines.

General-Purpose Registers (Accumulator: A) are temporary storage locations for data used in arithmetic or logical operations performed by the processor. The accumulator typically holds the result of an operation or serves as an operand for subsequent operations. Besides the accumulator, there may be several other general-purpose registers in the processor to store temporary data during program execution.

- ✓ Harvard: Memory Separation for Improved Performance.

The Harvard architecture is a modification of the Von Neumann architecture. It separates memory into two distinct sections: one for accessing program code and another for accessing data. Each memory section has its own dedicated bus. This separation allows simultaneous retrieval of both program instructions and data.

- The operational speed is higher compared to Von Neumann.
- Data and instructions are stored in separate memory spaces.
- Allows two simultaneous memory accesses:
  - Commonly used in most digital signal processors (DSPs).
  - Provides higher bandwidth.
  - Ensures more predictable performance.

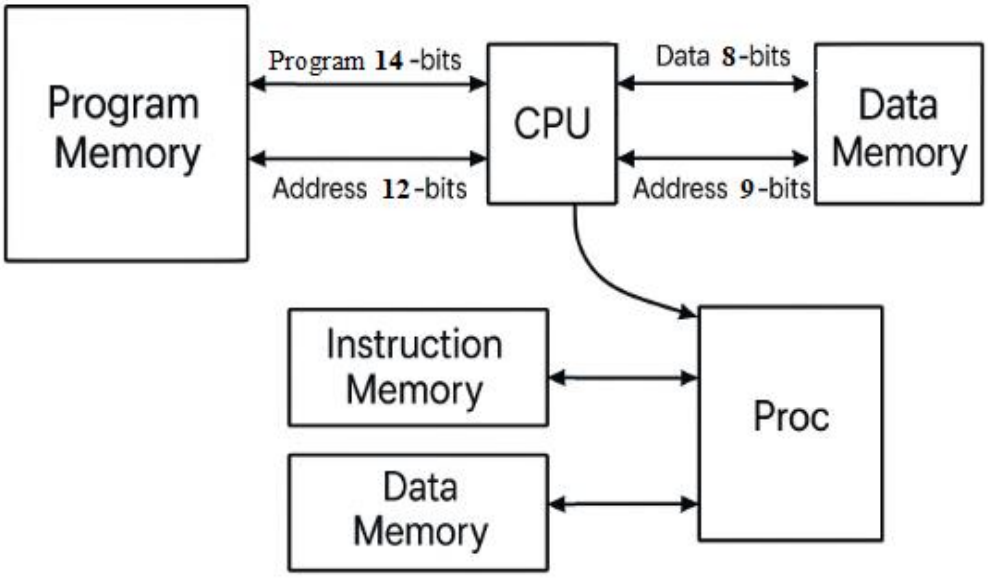

Figure III.2. *Harvard Architecture*.

The Harvard architecture separates memory into distinct sections for program instructions and data. Each has a dedicated bus, enabling simultaneous access to both, which enhances performance.

III.1.2 Software Architecture

Software Levels:

- ✓ Firmware:
  - Low-level software directly embedded in hardware.
  - Manages peripherals, initialization, and interrupt routines.

- ✓ Real-Time Operating Systems (RTOS):
  - Enable the management of concurrent tasks and priorities.
  - Examples: FreeRTOS, VxWorks, Zephyr.

- ✓ Applications:
  - Implement specific system functionalities (e.g., motor control, data processing).

Design Paradigms:

- ✓ Super Loop:
  - A simple design ideal for low-complexity systems.
- ✓ RTOS:

- Multitasking management for complex systems requiring real-time constraints.

III.*1.3 Internal and External Communication*

- ✓ Internal Communication:
  - Between hardware modules via buses (e.g., I2C, SPI).
  - Direct Memory Access (DMA) for fast transfers.

- ✓ External Communication:
  - Wireless: Wi-Fi, Bluetooth, Zigbee, LoRa.
  - Wired: Ethernet, RS232, CAN.

III.1.4 Energy Management

- ✓ Energy Optimization:
  - Low-power modes (Sleep, Deep Sleep).
  - Dynamic Voltage and Frequency Scaling (DVFS).

- ✓ Energy Sources:
  - Battery, solar power, grid power.

- ✓ Energy Monitoring:
  - Consumption tracking to extend battery life.

III.1.5 Real-Time and Scheduling

- ✓ Deterministic Delays:
  - Systems must respond to events within defined time limits.

- ✓ Scheduling Algorithms:
  - Fixed priorities: Rate-Monotonic (RM), Deadline-Monotonic (DM).
  - Dynamic priorities: Earliest Deadline First (EDF), Least Laxity First (LLF).

III.1.6 Fault Tolerance and Security

- ✓ Hardware and Software Redundancy:
  - Examples include redundant processors or error recovery mechanisms (e.g., watchdog timers).

- ✓ Security Mechanisms:
  - Authentication, data encryption, intrusion protection.

III.1.7 Human-Machine Interface (HMI)

- ✓ Simple Systems: LEDs, segment displays, buttons.
- ✓ Advanced Systems: Touchscreens, voice interfaces, mobile applications.

III.1.8 Modular Design

- ✓ Modularity:

- *Division into independent functional modules (e.g., communication, processing, energy management).*

✓ Reusability:
- *Use of standardized components and software to accelerate development.*

III.1.9 Design Constraints

✓ Size: Miniaturization to integrate systems into confined environments.
✓ Energy Consumption: Reduction to extend battery life.
✓ Reliability: Continuous operation without failures.
✓ Cost: Optimization to remain competitive.

III.1.10 Security and Cyber Resilience

✓ Data Protection:
- Encryption mechanisms (e.g., AES, RSA).
- Authentication to restrict unauthorized access.

✓ Secure Updates:
- Over-The-Air (OTA) updates to keep the system up to date while ensuring integrity [30].

III.2 Operating Systems for Embedded Systems

An embedded operating system is specifically designed to be installed on an embedded system. It is tailored to manage the unique requirements of the embedded system [31].

An operating system (OS) acts as an intermediary between the application software and the hardware. When a program needs to access a hardware resource, it simply sends the necessary information to the operating system, which then relays it to the appropriate device through its driver [32].

Without drivers, each program would need to recognize and handle communication with every type of peripheral device individually.

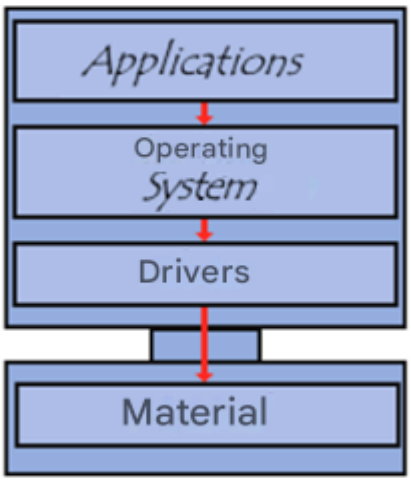

Figure III.3. *Operating Systems for Embedded Systems.*

An embedded operating system must be significantly more robust than a PC operating system. Embedded OSs require exceptional reliability and high performance.

While PC operating systems are designed with specific human-machine interfaces (e.g., screen-keyboard-mouse), embedded operating systems may feature specialized interfaces (e.g., numeric keypads, smartphone screens) or lack an interface altogether (e.g., credit cards, SIM cards). In such cases, an intermediary device (e.g., kiosks, phones) is used for interaction.

- Academic Examples:
  - ExoKernel, SPIN, Think.
  - Choices, OSKit, Coyote, PURE, 2K.
- Commercial Examples:
  - VxWorks, QNX, Windows CE, Windows XP Embedded.
  - JavaOS, MMLite, icWORKSHOP, Pebble.
  - Industrial-Grade Open Source: Linux, RTLinux (Real-Time Linux), eCos.

- VxWorks: VxWorks is a multitasking real-time operating system (RTOS) used by NASA in space missions, including the Discovery program, Mars Pathfinder, Stardust, Messenger, and Dawn. It is primarily employed in research and industry (aerospace, automotive, transportation, telecommunications) and in various enterprise communication systems like Intel, PowerPC, ARM, and MIPS. The languages commonly used with VxWorks are C, C++, and Ada.

- Windows CE: Windows CE is optimized for devices with limited storage capacity. The kernel can run with less than one megabyte of RAM and features a 32-bit multitasking architecture. Systems are often built without storage disks. Windows CE meets the definition of a real-time operating system. Examples of devices include PDAs, phones, fixed gaming consoles, web tablets, interactive TVs, and automated kiosks (e.g., ticketing machines). A notable example is the Xbox operating system [33].

- Linux: Since the 2000s, the Linux kernel has been used on hardware ranging from mobile phones to supercomputers. A prime example is Android, powering more than 80% of smartphones.
  - The Linux kernel supports most embedded systems, whether civil or military (e.g., set-top boxes, robots, aerospace, drones).
  - Targets: 16-bit and 32-bit CPUs, such as Motorola 683xx, Motorola ColdFire, Intel i960, ARM7TDMI, Altera NIOS, etc.

III.3 Specific-Purpose Processors and General-Purpose Processors

A processor is a digital system designed to process data through a sequence of simple steps. It must acquire data, process it, and produce a result.

III.3.1 General-Purpose Processors

General-purpose processors are characterized by a set of instructions and are software-programmable [34]. They can perform any task without requiring changes to their structure or interconnections. These processors are mass-produced and commonly referred to as microprocessors or central processing units (CPUs).

Examples: Microcontrollers, Intel Core processors, AMD FX-8000 processors, ARM, MIPS, TMS320, etc.

- ✓ Instruction Sets

An instruction set, also known as "Instruction Set Architecture" (ISA), refers to the set of instructions a processor can execute. Each instruction in the set is a specific command that the processor can perform to complete a particular task.

Instruction sets typically include:

- ✓ Arithmetic operations: e.g., addition and subtraction.
- ✓ Logical operations: e.g., value comparison and bit shifting.
- ✓ Memory operations: e.g., reading and writing.
- ✓ Control instructions: e.g., conditional and unconditional jumps.

The choice of an instruction set significantly impacts the performance and efficiency of a processor. Processors with simpler and smaller instruction sets can be faster and more energy-efficient, while those with larger instruction sets are more flexible and capable of handling a broader range of tasks.

The Instruction Set Architecture (ISA) is critical because:

- It defines the basic instructions executed by the CPU.
- It balances the CPU's hardware complexity and the ease of expressing required operations.
- It is often represented symbolically (e.g., MSP - Main Stack Pointer, encoded in 16 bits):
  - MOV R5, @R8; Comment: R5 [R8]
  - lab: ADD R4, R5; R4 R5+R4
- There are two main categories of instruction sets:
  - CISC: Complex Instruction Set Computer
  - RISC: Reduce Instruction Set Computer

- ✓ CISC: Complex Instruction Set Computer

CISC (Complex Instruction Set Computing) refers to a processor architecture where a single instruction can perform multiple operations. CISC processors typically feature a rich and complex instruction set capable of executing intricate tasks in a single clock cycle. These processors are often used in personal computers and servers.

To simplify and speed up programming, CISC computers support a large number of instructions. These instructions can perform complex operations, resulting in compact but complex code. One instruction may retrieve one or more operands, perform one or more operations, and thus vary in execution time. Examples of CISC processors: DEC VAX, Motorola 68000, Intel x86/Pentium, IBM System/360.

Historically, as memory was expensive, designers favored a dense instruction set to minimize memory usage.

Key characteristics of CISC:

- A single instruction can represent multiple elementary operations.
  - Example: Loading data, performing arithmetic, and storing results.
  - Example: Computing linear interpolation for multiple values in memory.
- Execution speed is enhanced using complex hardware mechanisms.
- Instructions exhibit wide variations in size and execution time.

- Produces compact code but is challenging to generate efficiently.

✓ RISC (Reduced Instruction Set Computer)

RISC (Reduced Instruction Set Computing) is a processor architecture that employs a simpler and smaller set of instructions compared to CISC processors. Each RISC instruction performs a single operation but can be executed much faster than CISC instructions. RISC processors are commonly used in embedded systems and mobile devices. [9]

Key Features of RISC:

- Small, simple instructions, all of the same size, with nearly identical execution times.
- No complex instructions.
- Pipelining (3 to 7 pipeline stages per instruction) significantly accelerates execution and increases clock speed.
- Simpler code generation, though it tends to be less compact.
- Reduced hardware complexity, leading to lower costs.
- Most modern microprocessors adopt this paradigm, including SPARC, MIPS, ARM, PowerPC, and others.

In the early 1980s, processor designers focused on minimizing the size and complexity of instruction sets. Simple, uniform instructions enable faster execution and reduced hardware requirements. This made RISC processors more cost-effective and efficient.

RISC processors generally offer:

- Faster performance.
- Higher energy efficiency.
- A trade-off with less flexibility compared to CISC processors.

Over time, the distinctions between RISC and CISC have blurred, as modern processors integrate features of both architectures to harness their respective advantages.
Examples of RISC Architectures:

- MIPS (Microprocessor without Interlocked Pipeline Stages)
  - Initially designed for powerful workstations (e.g., Silicon Graphics International Corp.).
  - Later adopted in gaming consoles like the Nintendo N64.
  - Wide-ranging family:
    - High-performance versions (e.g., MIPS 20Kc, 64-bit).
    - Compact versions (e.g., SmartMIPS, 32-bit for smart cards).
- ARM (Advanced RISC Machines, formerly Acorn)
  - Among the most popular 32-bit embedded processors, especially in mobile phones.
  - Known for low power consumption.
  - Its successor, StrongARM, was commercialized by Intel under the name XScale.
- SuperH (SH) (Hitachi)
  - Widely used in Sega consoles and Personal Digital Assistants (PDAs).

- PowerPC
  - Equally utilized in embedded systems and desktop computers.
- SPARC (Scalable Processor Architecture)
  - Used in UNIX systems by Sun Microsystems and other manufacturers.

III.3.2 Application-Specific Processors

These are hardware-programmable processors designed to meet a specific need. They are less complex and more efficient than general-purpose processors but often require significant effort to design.

Example: DSP, RS232 transmitter, etc.

✓ DSP: Digital Signal Processing

A Digital Signal Processor (DSP) is a specialized processor designed to handle digital signals, such as audio, video, or sensor data.

Unlike a general-purpose processor, a DSP is optimized to quickly perform mathematical and logical operations on digital signals. DSPs are widely used in various fields, including telecommunications, digital audio, computer vision, biomedical signal processing, navigation, and more.

DSPs can be programmed with specific algorithms to carry out complex signal processing tasks such as noise reduction, audio and video compression, speech recognition, and pattern detection [35].

They can also be combined with sensors to deliver real-time signal processing systems for control and monitoring applications. Many embedded systems, such as mobile phones and portable media players, rely on DSP-based signal processing applications.

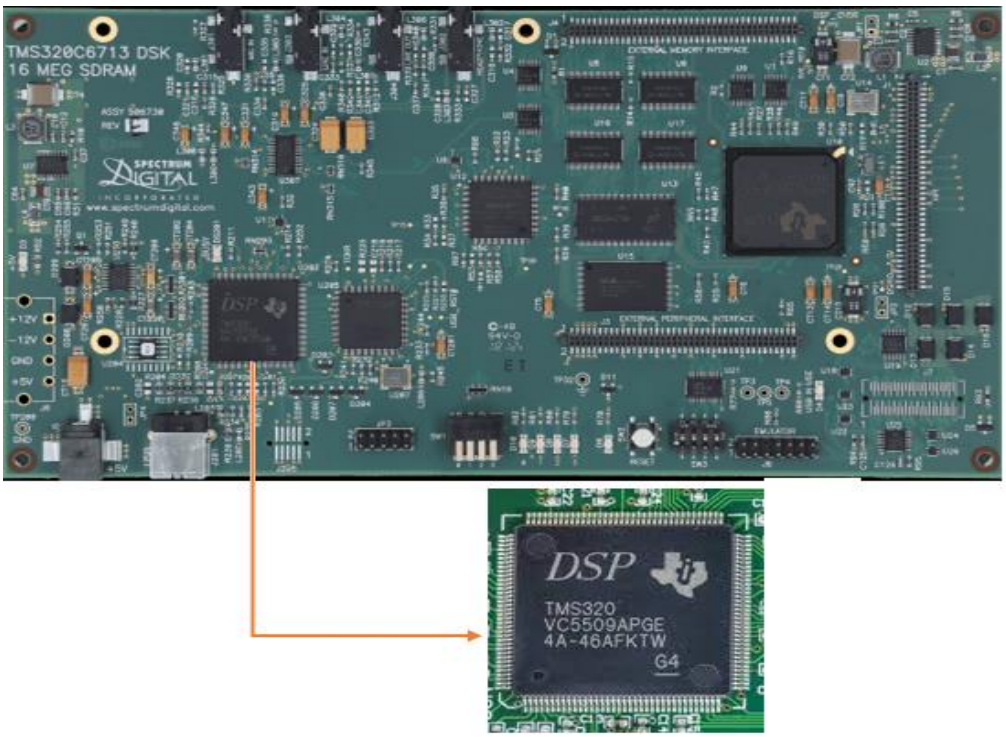

Figure III.4. *DSP Kit (DSK) Starter Kit TMS320C6713.*

III.4 Pipeline Operation

Pipeline operation (or "processing pipeline") is a technique used in processor design to enhance performance. It involves dividing the processing of an instruction into multiple distinct stages, which can be executed concurrently.

A pipeline is a component within a processor where the execution of instructions is split into several stages. The processor can begin executing a new instruction without waiting for the previous one to complete. This pipeline architecture speeds up instruction execution. Each stage in a pipeline is called a "pipeline stage," and the total number of stages determines the "pipeline depth."

A typical pipeline is divided into five main stages:

- ✓ Fetch Stage: The instruction is read and retrieved from memory.
- ✓ Decode Stage: The instruction is decoded to identify the operation to be performed.
- ✓ Execute Stage: The operation is performed.
- ✓ Memory Stage: Data is transferred to or from memory.
- ✓ Write-Back Stage: The results are written to the appropriate register.

Each stage is performed by a specialized hardware unit known as a "functional unit." Instructions are processed simultaneously by different functional units, optimizing processing time and improving the processor's performance. However, pipelines can also introduce challenges, such as data dependencies and control conflicts, which can slow down processing. Therefore, designing an efficient and optimized pipeline is crucial for each processor.

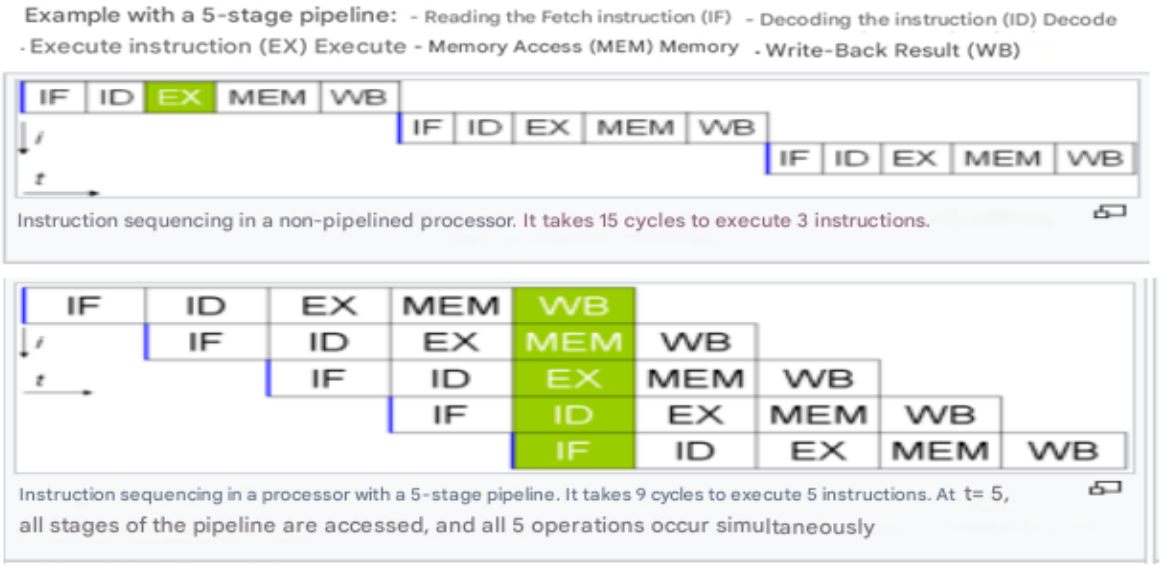

Figure III.5. *Operation with and without Pipelining.*

- ✓ With Pipelining: in a pipelined processor, multiple instructions are processed simultaneously at different stages of the pipeline. For example, while one instruction is being executed, another can be decoded, and yet another fetched. This parallelism significantly increases the throughput and overall performance of the processor.

- ✓ Without Pipelining: in a non-pipelined processor, each instruction is processed sequentially—fetch, decode, execute, memory, and write-back occur one after the other. The processor must wait for one instruction to fully complete before starting the next, resulting in slower performance.

III.5 Memory Hierarchy

One of the primary roles of an embedded systems designer is to ensure that everything necessary is as close to the CPU as possible, following the principle of locality. This leads to the adoption of techniques such as:

- ✓ Using Direct Memory Access (DMA) techniques.
- ✓ Implementing architectural techniques to optimize performance.

III.5.1 Levels of Memory Hierarchy

- ✓ On-Chip Memory (Internal Memory): This is the first level of memory hierarchy and is composed of the processor's registers. It stores temporary and intermediate variables. This type of memory is the fastest and most expensive.

- ✓ Cache Memory System: This is the second level of the hierarchy. While it is still fast and expensive, it is less costly than the first level. Cache memory is used to store instructions and data close to the CPU for quick access, especially for frequently or recently used data. Cache systems typically use SRAM technology.

- ✓ Off-Chip Memory (External Memory): This is the third level of the hierarchy. It is slower and less expensive than the other types of memory. External memory is usually used for storing unused data and instructions, serving as long-term storage. Accessing information from this memory requires additional "handshaking" and control, making it slower compared to the other levels.

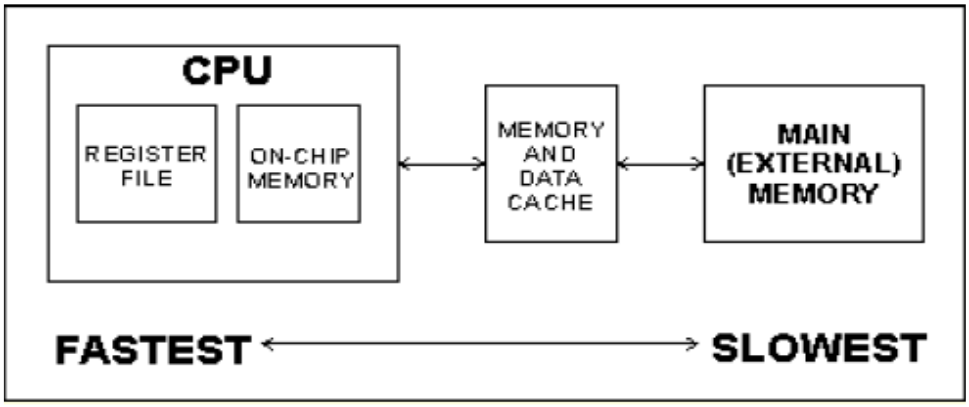

Figure III.6. *Memory Hierarchy Levels in Embedded Systems.*

III.6 Peripherals and Interfaces

Peripherals expand the functionality of a system. They can be categorized into two types:

- ✓ Peripherals for External Communication:
    - Connect external components such as memory, oscillators, clocks, or external interfaces (USB, Ethernet, CAN, SPI, etc.).
- ✓ Peripherals for Data Processing Acceleration:
    - Examples include graphics cards for video algorithm acceleration.

From the processor's perspective, peripherals are accessed via interfaces, making them "addressable". This allows software, through the processor, to configure and use them with simple read/write memory accesses.

Examples: Hard drives, displays, keyboards, mice, sound cards, LEDs, etc. Peripherals communicate using protocols and are connected via buses.

III.6.1 UART (Universal Asynchronous Receiver/Transmitter) Bus

A serial communication bus that uses two full-duplex data wires (TX and RX) for data transfer [32].

Operates asynchronously (no global clock signal), making it simple and cost-effective but slower and less precise than other buses.

III.6.2 USB (Universal Serial Bus)

- ✓ A synchronous serial communication bus that uses four data wires (D+, D-, VCC, GND).
- ✓ Features a global clock signal for synchronized data transfer.
- ✓ Faster and more precise than UART and capable of supplying power to connected devices.
- ✓ Roles in communication:
  - Host: Directs traffic and regularly queries USB devices.
  - Device: Responds to host requests.

III.6.3 I2C (Inter-Integrated Circuit) Bus

- ✓ A synchronous serial communication bus using two wires (SDA and SCL).
- ✓ Designed for low-speed communication, often used to connect sensors and devices to microcontrollers.
- ✓ Commonly used for internal communication between on-board devices, such as EEPROM memory, DAC/ADC converters, etc.
- ✓ Follows a master-slave communication model initiated by the processor.

III.6.4 SPI (Serial Peripheral Interface)

- ✓ Similar to I2C but faster and uses four wires (MOSI, MISO, SCK, SS).
- ✓ Supports synchronous communication with a global clock signal.
- ✓ Often used for high-speed connections, such as memory, LCDs, SD cards, video processing, and telecommunications.
- ✓ Features:
  - Three data wires + one slave-select wire per connected circuit.
  - Master-slave communication initiated by the processor, enabling simultaneous bidirectional data transfer.

III.6.5 Comparison of Different Buses

Table III.1 provides a concise summary and comparison of the different buses used in embedded systems.

Bus	Communication Type	Number of Wires	Maximum Speed	Synchronization
UART	Asynchronous Serial	2	Up to 115.2 Kbps	Asynchronous
USB	Synchronous Serial	4	Up to 10 Gbps	Synchronous
I2C	Synchronous Serial	2	Up to 3.4 Mbps	Synchronous
SPI	Synchronous Serial	4	Up to 80 Mbps	Synchronous

Tableau III.1 *Overview and Comparison of Embedded System Buses*

Summary:

- UART: Simple and cost-effective, but slower and less precise than other buses.

- USB: Faster and more precise than UART, with the added capability of powering connected devices.
- I2C: Designed for low-speed communication, often used to connect sensors and peripherals to microcontrollers.
- SPI: Faster than both UART and I2C, commonly used to connect memories, LCDs, and SD cards to microcontrollers.

III.7 Communication Mechanisms and Associated Protocols

Communication between the embedded processor and peripherals occurs via buses. This communication can be:

- Serial: Transmitting one bit at a time.
- Parallel: Transmitting multiple bits simultaneously.

Additionally, communication can be:

- Synchronous: The sender and receiver are synchronized to exchange data simultaneously (e.g., Internet network connections).
- Asynchronous: Transmission and reception occur at distinct times, separated by a delay (e.g., fax and email).

Full Duplex vs. Half Duplex:

- Half Duplex: Uses a single wire to allow bidirectional communication, but only one direction at a time.
- Full Duplex: Enables simultaneous bidirectional communication.

Synchronous processors are valued for both their high processing speeds and energy efficiency, making them particularly appealing for industries designing autonomous embedded systems (e.g., mobile phones, probes).

III.7.1 Communication Protocols

Serial Port: The serial port is the most important communication port in an embedded system, traditionally based on the RS232 standard. On modern computers, it has been replaced by USB emulators.

Definition: A communication protocol specifies a set of rules for a particular type of communication.

III.7.2 Concept

Communication involves transmitting data. However, unless the participants attribute meaning to the transmitted data, it remains raw data and does not qualify as information. For effective communication, participants must:

1. Share a common language.
2. Follow minimal rules for sending and receiving data.

Role of Protocols: Protocols ensure that these conditions are met, allowing smooth communication between devices.

III.7.3 Example of Use

Consider a session using X Window over an ISDN line. Since ISDN charges are time-based, the low-level ISDN session can be disconnected after a few seconds of inactivity while maintaining the high-level TCP/IP connection.

When a TCP/IP message is sent, the ISDN driver re-establishes the connection within two seconds, creating the illusion of a continuous link while minimizing costs. For TCP/IP, the connection appears uninterrupted.

Key Components:

- ISDN (Integrated Services Digital Network): A digital network for voice, video, and data.
- TCP (Transmission Control Protocol) and IP (Internet Protocol): Core protocols for data transfer over the Internet.
- TCP/IP Suite: The set of protocols used for Internet data transfer.

III.8 Example of Architecture

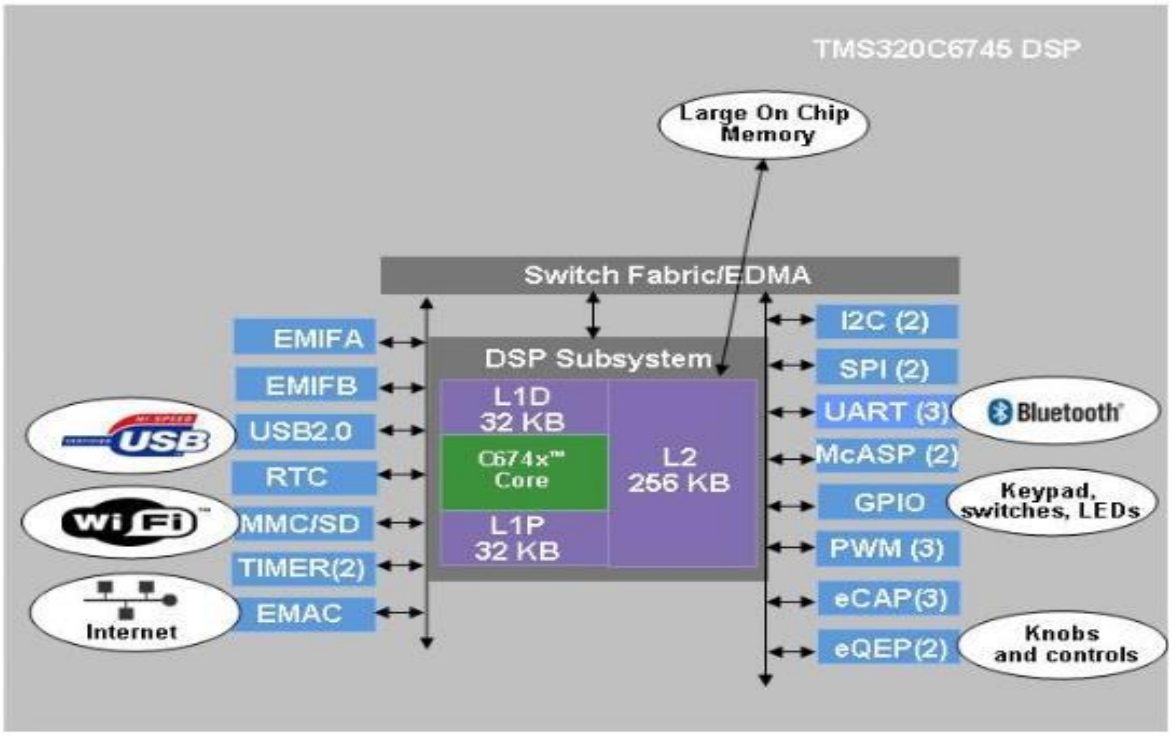

Figure III.7. *Example of Embedded Processor Architecture.*

Figure III.7 illustrates an example of architecture for embedded processors, showcasing the various surrounding components:

- ✓ Asynchronous External Memory Interface (EMIF): Provides an interface to external asynchronous memory.
- ✓ USB Synchronous External Memory Interface: Used for synchronous communication with external memory.
- ✓ RTC (Real-Time Clock): Provides switching network capabilities.

- ✓ SDMMC Controller: Enables communication between the microcontroller and multimedia cards or SD memory cards.
- ✓ Timer Interface: Allows users to pause parts of their code to specify a region of interest. The Timer class interface supports start and stop operations.
- ✓ Ethernet Media Access Controller (EMAC): Offers an efficient interface between the device's main processor and the network [36].
- ✓ McASP (Multichannel Audio Serial Port): Typically interfaces with devices using Time-Division Multiplexing (TDM), often configured as Inter-IC Sound (I2S) for audio applications.
- ✓ GPIO (General-Purpose Input/Output): A standard interface for connecting microcontrollers to other electronic devices.
- ✓ Pulse Width Modulation (PWM): A method for controlling the average power delivered by an electrical signal. Common applications include controlling LEDs, fans, or vibrators in mobile phones. Fixed-purpose PWMs do not require Linux PWM API implementation.
- ✓ eCAP Block: Allows configuration of the enhanced capture peripheral. It records time intervals between well-defined logical transitions, enabling tracking of up to four sequential event intervals.
- ✓ eQEP Block: Configures the enhanced quadrature encoder pulse, often used for precise motion tracking and motor control.

This architecture provides a comprehensive set of interfaces and components for efficient communication, control, and processing in embedded systems

Chapter IV

Methodology for Designing Embedded Systems

IV.1 Design Environments

Design environments for embedded systems consist of ecosystems that bring together tools, platforms, and methodologies to develop, test, and deploy embedded systems. Below is a detailed overview of their characteristics and classification.

Figure IV.1 illustrates a general architecture of a complex embedded system (Embedded System Z), which integrates multiple embedded subsystems (X and Y).

- Sensors and Actuators (S&C): Embedded subsystems X and Y receive input data from sensors that measure physical variables (e.g., temperature, pressure) and use actuators to execute commands.
- Communication Interfaces (CI): Subsystems communicate with each other and external systems via communication interfaces. These can include protocols such as UART, SPI, or CAN, as well as wireless interfaces like Bluetooth or Wi-Fi.
- Main System (Z): The main system coordinates the various subsystems, integrating sensor data and managing centralized communications.

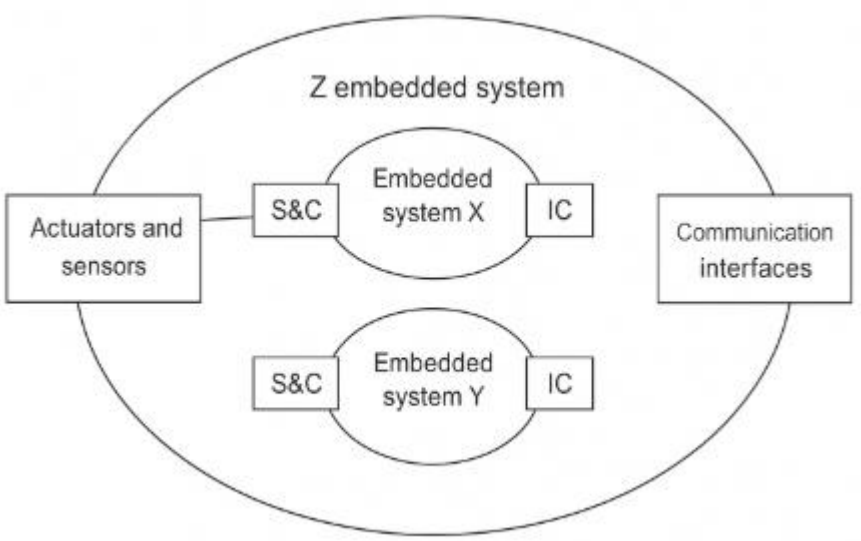

Figure IV.1. *Embedded system and its environment* [37]

IV.1.1 Definition and Role of Design Environments

An embedded system design environment encompasses the software and hardware tools used for:

- Specifying, modeling, and simulating systems.
- Implementing hardware and software solutions.
- Testing, validating, and deploying systems.

These environments are tailored to the specific constraints of the system being designed, such as real-time requirements, energy consumption, robustness, and cost [1].

.

Design environments for embedded systems include all the tools, methods, and platforms used to develop, test, and validate embedded systems. These environments comprise:

- ✓ Programming Languages: C, C++, Python, VHDL, Verilog, etc.
- ✓ Integrated Development Environments (IDEs): Eclipse, Keil, MPLAB, and more.
- ✓ Simulation and Modeling Tools: MATLAB/Simulink, LabVIEW.
- ✓ Project and Configuration Management Tools: Git, SVN, Jira.
- ✓ Development Boards and Emulators: Arduino, Raspberry Pi, STM32, JTAG, and others.

IV.1.2 Choosing Environments Based on Constraints

✓ Categories of Environments

❖ Software Environments: These are tailored for designing embedded applications and performing software simulations. They include:

- Integrated Development Environments (IDEs):
  - Eclipse, Keil µVision, MPLAB X, Arduino IDE.
- Specific Programming Languages:
  - C, C++, Python, Rust for embedded applications.
  - VHDL, Verilog for hardware design.
- Simulation and Modeling Tools:
  - MATLAB/Simulink: For signal processing and control.
  - ModelSim: For digital circuit simulation.
  - LabVIEW: For interface design and prototyping.

❖ Hardware Environments: These focus on physical prototyping and validation on real-world platforms. They include:

- Development Boards:
  - Arduino, Raspberry Pi, BeagleBone, STM32.
- Emulators and Debuggers:
  - JTAG, SWD, In-Circuit Debugger (ICD).

- Hardware Simulators:
  - FPGA (Field Programmable Gate Array) for hardware prototyping.

❖ Mixed Environments: These integrate software and hardware simulation for real-time testing. They combine tools for complete design and validation, including:

- Hardware/Software Co-Design Simulation:
  - Enables testing the interaction between hardware and software.
  - Tools: CoWare, SystemC.

- Real-Time Environments:
  - Real-Time Operating Systems (RTOS) such as FreeRTOS, Zephyr, and VxWorks.

✓ Criteria for Selection: The choice of a design environment depends on several factors:

❖ Technical Constraints:
- Complexity of the system to be developed.
- Real-time requirements, energy consumption, and size constraints.

❖ Ease of Use:
- Learning curve associated with the tools.
- Availability of documentation and community support.

❖ Cost and Licensing:
- Open-source environments (e.g., Eclipse, FreeRTOS) vs. commercial solutions (e.g., MATLAB, VxWorks).

❖ Portability and Scalability:
- Ability to migrate between hardware platforms.
- Integration of third-party tools for specific needs.

IV.2 Lifecycle and Development Stages of an Embedded System

IV.2.1 Lifecycle Definition

The lifecycle of an embedded system refers to the series of stages involved in its design, development, deployment, and maintenance.

Figure IV.2 presents the various stages of an embedded system's lifecycle as a sequential diagram. The development process occurs partially in parallel for hardware and software aspects but includes several common phases to ensure coordination between the two.

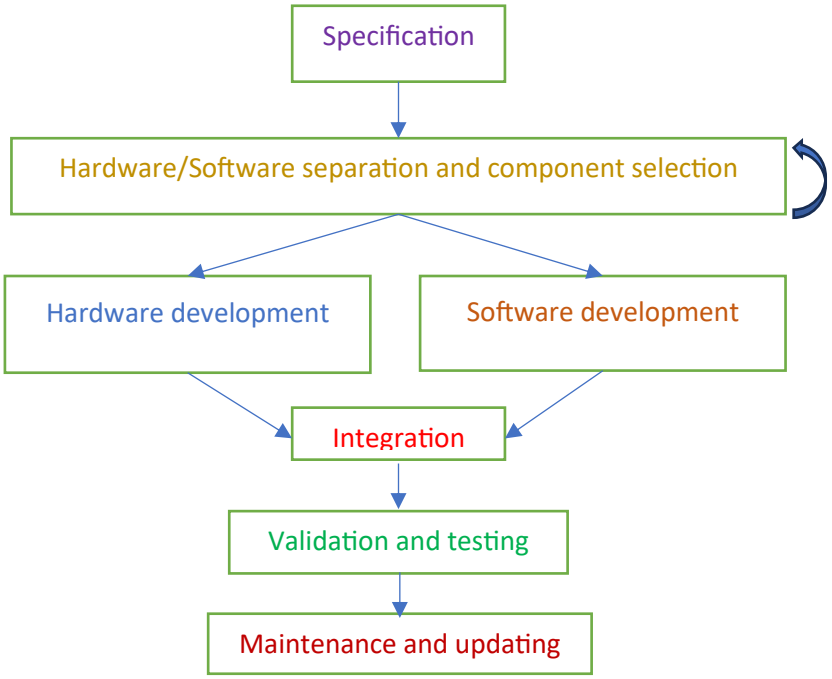

Figure IV.2. *The development cycle of an embedded system involves parallel processes for hardware and software, with shared phases for synchronization* [38].

IV.2.2 Key Stages

✓ Specification

This is the initial stage of development. It involves clearly defining the system's objectives, requirements, and constraints, including:

- Drafting a detailed specification document that outlines the required functionalities.
- Identifying constraints such as energy consumption, real-time deadlines, or cost.

Importance: This phase is critical to prevent misunderstandings that could negatively impact subsequent design stages.

✓ Hardware/Software Partitioning and Component Selection

At this stage, the system is divided into two primary components:

- Hardware: Includes microcontrollers, sensors, actuators, etc.
- Software: Comprises algorithms, drivers, and user interfaces.

The selection of hardware and software components is vital to meet the requirements defined in the specification phase.

Example: Choosing a microcontroller with sufficient memory to handle complex algorithms.

✓ Hardware Development

In this phase, the hardware components are designed and integrated:

- Electronic Circuit Design: Creation of schematics and Printed Circuit Boards (PCBs).
- Component Assembly: Integration of sensors, microcontrollers, and communication modules.
- Prototyping: Fabrication of prototypes for preliminary testing.

Objective: To ensure that the hardware meets performance and robustness requirements [39].

✓ Software Development

Software development occurs in parallel with hardware development and includes:

- Embedded Programming: Developing algorithms and software for the microcontroller.
- Driver Implementation: Writing drivers to interface hardware and software.
- Simulation and Validation: Using tools like MATLAB or Simulink to validate software behavior.

Expected Outcome: A functional software application that is fully compatible with the hardware.

✓ Integration

This is a critical phase where hardware and software are combined into a complete system.It involves:

- Interaction Testing: Verifying that software and hardware modules communicate correctly.
- Conflict Resolution: Debugging integration-related issues.

Objective: Ensure the system operates as intended in its entirety.

✓ Validation and Testing

This phase ensures that the system meets the initial specifications:

- Unit Testing: Verifying individual components.
- Integration Testing: Validating the entire system.
- Real-World Testing: Simulating real-world scenarios to assess robustness and performance.

Importance: To guarantee quality and reliability before mass production.

✓ Maintenance and Updates

After deployment, it is crucial to maintain the system, which includes:

- Continuous Monitoring: Evaluating performance during operation.
- Bug Fixes: Identifying and resolving issues.
- Updates: Adding new features or improving performance.

Example: Updating the firmware of an IoT system to introduce a new feature.

IV.3 Control and Regulation Systems

Control and regulation systems play a central role in embedded systems, especially in fields such as automotive, aerospace, and industry. They ensure the supervision and control of various subsystems to maintain optimal operation [40].

IV.3.1 Fundamental Concepts

✓ Control

Control refers to the actions taken by an embedded system to influence a physical process or machine. It relies on algorithms, sensors for measurements, and actuators to implement decisions.

Example: Adjusting the speed of a motor to maintain a specified setpoint.

✓ Regulation

Regulation is a type of control in which the system automatically adjusts a measured variable to keep it at a target value (setpoint), despite disturbances.

Example: Regulating the temperature of an industrial furnace by adjusting the heating element.

✓ Architecture of Control and Regulation Systems

Control and regulation systems typically consist of the following components:

- Sensors: Measure physical quantities (temperature, speed, pressure, etc.).
- Actuators: Apply commands (motors, valves, etc.).
- Microcontroller/Embedded Processor: Hosts the control and regulation algorithms.
- Communication Interfaces: Facilitate data exchange between modules (I2C, SPI, CAN, etc.).
- Control Software: Contains regulation algorithms (e.g., PID, fuzzy logic).

✓ Types of Control
- Open-Loop Control
  - Principle: The system applies a command without verifying whether the expected result is achieved.
  - Advantages: Simple and fast.
  - Disadvantages: Less accurate, sensitive to disturbances.
  - Example: Operating a motor at a fixed speed without measuring its actual speed.

- Closed-Loop Control (Feedback Control)
  - Principle: A measured variable is compared to a setpoint, and the resulting error is used to adjust the action.
  - Advantages: More accurate and robust against disturbances.
  - Example: Regulating a motor's speed using a speed sensor.

✓ Regulation Algorithms
- PID Controller (Proportional, Integral, Derivative)
  - Operation: Adjusts the output based on three terms:
    - Proportional (P): Reacts to the current error.
    - Integral (I): Corrects past errors.
    - Derivative (D): Anticipates future errors.
  - Applications: Temperature control, speed regulation, and position control.

- Model Predictive Control (MPC)
  - Principle: Uses a mathematical model to predict the system's future behavior and adjusts control actions accordingly.
  - Advantages: Optimal for complex systems.
  - Applications: Advanced industrial processes.

- Adaptive Control
  - Principle: Automatically adjusts control parameters in response to changes in the system or environment.
  - Applications: Avionics, drones, and robotic systems.

✓ Practical Applications

- Automotive
  - Control: Managing engine speed using a PID controller.
  - Regulation: Controlling the temperature in the air conditioning system.

- Industry
  - Control: Operating robotic arms for assembly tasks.
  - Regulation: Maintaining pressure in a hydraulic system.

- Home Automation
  - Control: Activating lights based on detected presence.
  - Regulation: Regulating ambient temperature through a thermostat.

- Aeronautics
  - Control: Automatic stabilization of drones.
  - Regulation: Controlling altitude using barometric sensors.

✓ Diagram of a Control and Regulation System

A typical system can be represented by the following diagram:

- Sensor: Measures the quantity of interest.
- Microcontroller: Compares the measurement to the setpoint, calculates the error, and applies the algorithm (PID, etc.).
- Actuator: Executes the command.
- Process: Reacts to the command.
- Feedback Loop: The sensor measures again and continuously adjusts the system.

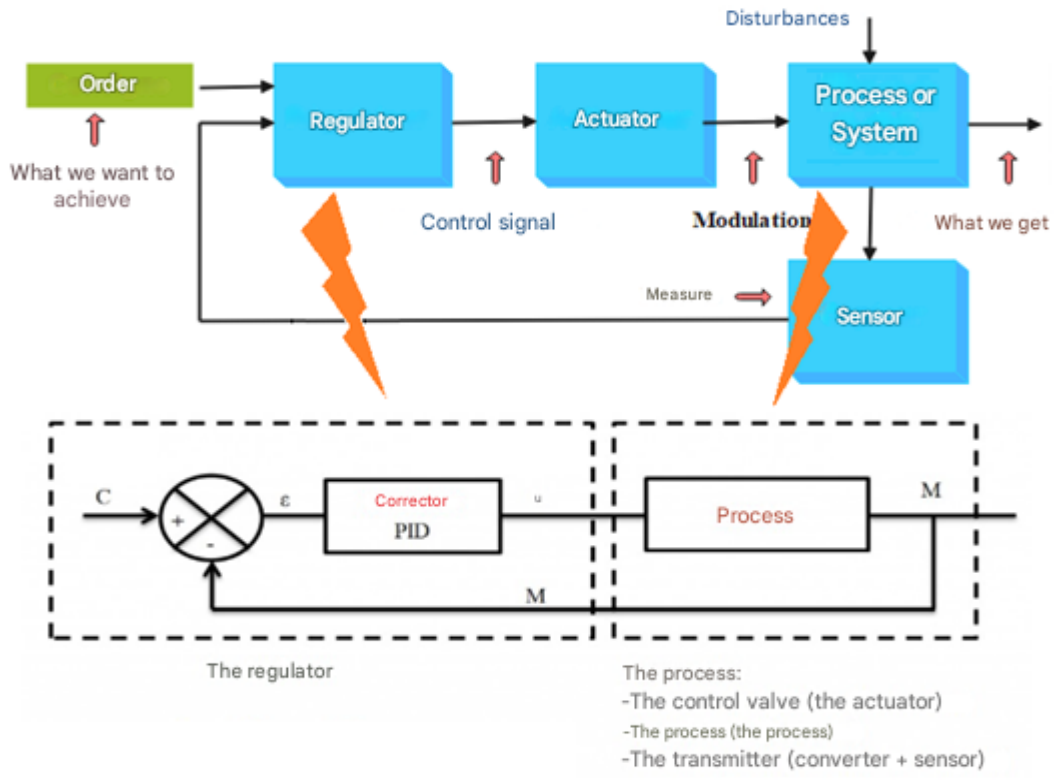

Figure IV.3. *Control and Regulation System*

✓ Development Tools and Platforms

- Software Tools: MATLAB/Simulink, LabVIEW.
- Development Boards: Arduino, STM32, Raspberry Pi.
- Communication Protocols: I2C, CAN, UART.

IV.4 Design Examples

IV.4.1 Case Study 1: Embedded Temperature Monitoring System

✓ Objective: Develop an embedded system capable of measuring and regulating temperature in a given environment.

✓ Process:

- Specifications: Define the temperature range and required accuracy.
- Design: Utilize a temperature sensor (DS18B20) and a microcontroller (STM32).
- Implementation: Program in C to acquire data and control a heating device.
- Testing: Validate performance under different scenarios.

Figure IV.4. *Climate Monitoring in Agricultural Greenhouses.*

IV.4.2 Case Study 2: Line-Following Mobile Robot

✓ Objective: Build a robot capable of following a line drawn on the ground in real time.

✓ Process:

- Specifications: Define speed and tracking precision requirements.
- Design: Use infrared sensors, an Arduino microcontroller, and DC motors.
- Implementation: Program in C++ to analyze sensor data and adjust motor controls.
- Testing: Validate the robot's performance on various paths.

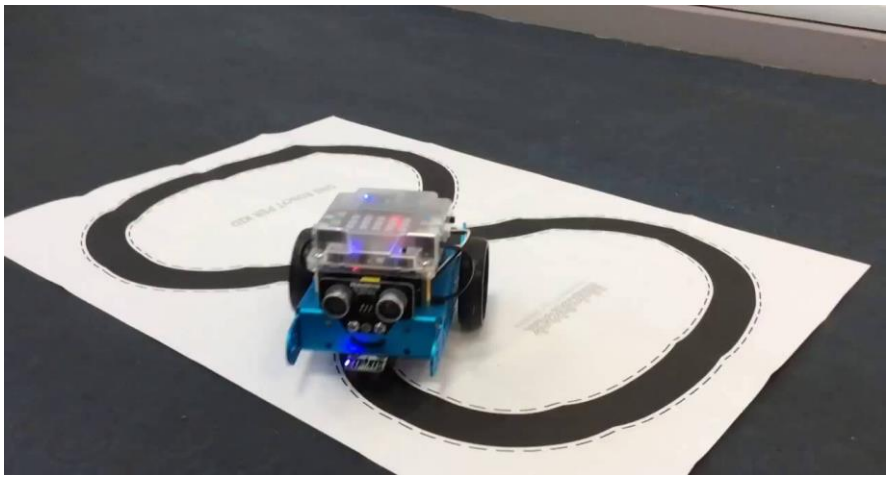

Figure IV.5. *Line-Following Mobile Robot.*

IV.4.3 Case Study 3: Heart Rate Monitoring System Design

- ✓ Objective: Develop a wearable device for real-time heart rate monitoring.
- ✓ Process:

  - Specifications: Define the frequency range and battery life requirements.
  - Design: Use a PPG (PhotoPlethysmoGram) sensor and an ESP32 microcontroller for Bluetooth data transmission.
  - Implementation: Program in Python for data processing.
  - Testing: Conduct clinical validation with human subjects.

Figure IV.6. *Smartwatch with Heart Rate Monitoring Feature*

IV.4.4 Case Study 4: Energy Management System for IoT

- ✓ Objective: Design an embedded system optimized for energy management in IoT devices to extend their battery life while ensuring efficient performance.
- ✓ Process:

  - Specifications:
    - Identify the energy requirements of the devices.
    - Define possible energy sources (batteries, solar panels, energy harvesting).
    - Establish maximum consumption criteria.

- Design:
  - Use low-power microcontrollers (e.g., ESP32, STM32L).
  - Integrate energy management modules such as efficient DC-DC converters.
  - Implement sleep and deep-sleep state management algorithms.
- Implementation:
  - Program in C/C++ to optimize processor energy consumption.
  - Monitor and control battery charging and discharging.
  - Integrate intelligent sensors to reduce unnecessary sampling.
- Testing:
  - Simulate consumption scenarios under real-world conditions.
  - Validate energy performance across various IoT devices.
  - Measure battery life under different operational modes.
- Optimization:
  - Adjust parameters to maximize battery life.
  - Update algorithms to adapt to evolving requirements.

✓ Examples of Applications:

- Wireless sensor devices in smart homes.
- Low-power GPS tracking systems.
- Agricultural monitoring using solar-powered sensors.

Figure IV.7. *AI-Based Energy Management System for IoT Devices.*

Chapter V

Embedded Systems Security

V.1 Introduction

Embedded systems, now ubiquitous in computing infrastructures, are integrated into online services and interact with databases. However, these systems must adhere to stringent constraints related to their environment or specific functionalities, making any vulnerability unacceptable. Consequently, cybersecurity becomes a critical concern in the design and development of embedded applications and systems.

In a context where software plays a central role across all sectors, verification methods are progressively becoming essential tools to ensure the reliability and security of computing systems, particularly programs. The field of embedded systems follows this trend. Their security-focused nature demands a rigorous verification of security properties. Moreover, the financial and technical consequences of a significant error, replicated on a large scale (as seen with smart cards), further emphasize the importance of ensuring the reliability and correctness of embedded programs.

V.2 Security Objectives

Digital security pursues three fundamental goals: ensuring confidentiality, maintaining integrity, and ensuring the availability of resources, information, and systems.

V.2.1 Confidentiality

Confidentiality refers to ensuring that only authorized entities (people, machines, or software) can access the resources and information intended for them. It aims to prevent sensitive data from being disclosed to unauthorized users. Attacks targeting confidentiality seek to steal or exfiltrate sensitive information.

To protect confidentiality in an embedded system, several mechanisms are implemented:

- ✓ Access control to the system's information and resources.
- ✓ Restricting communications within the system to authorized entities only.
- ✓ Implementation techniques include:

    - Strict management of access to the embedded system.
    - Regulation of access to the system’s internal resources.

V.2.2 Availability

Availability aims to ensure that the system, its resources, and its information are accessible and operational when needed. This includes preventing resource saturation, protecting against failures, and denying access to unauthorized users.

Attacks targeting availability aim to make the system unusable or unavailable, thereby compromising its functionality.

V.2.3 Integrity

Integrity ensures that resources and information remain intact, without being modified, altered, or destroyed by unauthorized entities. Attacks targeting integrity aim to manipulate, add, or delete data or resources. The goal is to preserve the reliability of the information by ensuring that no alterations have been made.

✓ Types of alterations:

- Data: Unauthorized modifications, deletions, or additions.
- Programs: Unauthorized manipulation or changes to software.

To protect the integrity of data, it is essential to strictly control write and modification operations [41]. Verification mechanisms can detect potential alterations by comparing data with reference values, such as:

✓ Redundancy mechanisms.
✓ CRC codes (Cyclic Redundancy Check).
✓ Digital signatures.

These measures can be implemented through hardware or software solutions.

Consequences of Alteration: A modified program, whether due to a deliberate attack or a simple programming error, can have significant impacts, such as:

✓ Changes to application data.
✓ Modifications to critical systems, such as authorization tables or operating system programs.
✓ Activation of unintended services.

Thus, ensuring the integrity of an embedded system is crucial to prevent malfunctions or exploitable vulnerabilities.

V.3 Hardware and Software Vulnerabilities

Embedded systems connected via IP are increasingly exposed to network attacks. These threats are already targeting various devices, such as routers or connected printers, and could just as easily extend to home automation networks or even Internet-connected vehicles.

From the design phase, it becomes imperative to integrate security measures tailored to embedded systems. However, this approach is still somewhat marginal in the current practices of designers.

The security of embedded systems primarily relies on three essential aspects:

✓ Hardware: Protecting physical components from manipulation or malicious intrusions.
✓ Embedded Software: Ensuring the reliability and resilience of programs against cyberattacks.
✓ External Communications: Ensuring that exchanges with the external environment are protected from unauthorized interceptions and alterations.

V.3.1 Attacks on Embedded Systems

Attacks against embedded systems primarily target their non-functional properties, including:

- ✓ Confidentiality: The main goal is to access sensitive or private information without authorization.
- ✓ Integrity: These attacks aim to modify, delete, or add information without authorization, thereby compromising the reliability of the data or systems.
- ✓ Availability: The goal is to overload the device with a large number of requests, rendering it unusable or unavailable to legitimate users.

The literature provides various definitions of the concept of malicious code. It is often described as any code fragment added, modified, or removed from software with the intent to intentionally cause malfunctions or disrupt the normal operation of the system [42].

Other approaches expand this definition by considering malicious code as an element that disrupts or enables a third-party program to alter the integrity of data, the information flow, the control, or the overall operation of a system.

Attacks on systems can be grouped into two broad categories:

- Physical and Side-Channel Attacks
- Logical Attacks

- ✓ Physical and Side-Channel Attacks: These attacks exploit the specific implementation details or physical properties of the system [43]. They are typically divided into two types:

  - Invasive Attacks: These attacks aim to directly access the device to observe, manipulate, or interfere with its internal components. They often require expensive infrastructure, making them complex to implement. Examples include:
    - Micro-probing
    - Reverse engineering of circuits

  - Non-Invasive Attacks: These attacks exploit the physical characteristics of the system without needing direct access. They include:
    - Synchronization analysis
    - Fault induction
    - Electromagnetic analysis
    - Power consumption analysis

Side-channel attacks are particularly dangerous because they reveal sensitive information without physically compromising the device.

Logical Attacks are relatively easy to execute, particularly in environments capable of downloading and running applications. They exploit vulnerabilities or bugs in the overall system architecture (hardware/software) as well as in the design of cryptographic algorithms or security protocols.

In practice, attackers often combine several techniques to achieve their objectives. For example, on a smartphone, possible malicious actions may include:

- Making the phone unusable.
- Modifying its behavior.

- Accessing sensitive data.
- Making voice or data calls without the user's knowledge.
- Bypassing digital rights management (DRM) policies.

✓ Software Attacks

These attacks represent a major threat to embedded systems, particularly those that allow for the downloading and execution of applications. Unlike physical or side-channel attacks, they generally require simple and easily accessible infrastructure [44].

Such attacks are often carried out by malicious agents such as:

- Viruses
- Worms
- Trojans

They exploit vulnerabilities in the operating system or applications, gaining access to internal systems and disrupting their functioning. The impacts can include:

- Manipulation of critical data or processes (attacks on integrity).
- Theft of personal or confidential information (attacks on confidentiality).
- Denial of access to system resources (attacks on availability).

This type of attack is particularly concerning in embedded systems due to their limited resources and often tight connections with other devices.

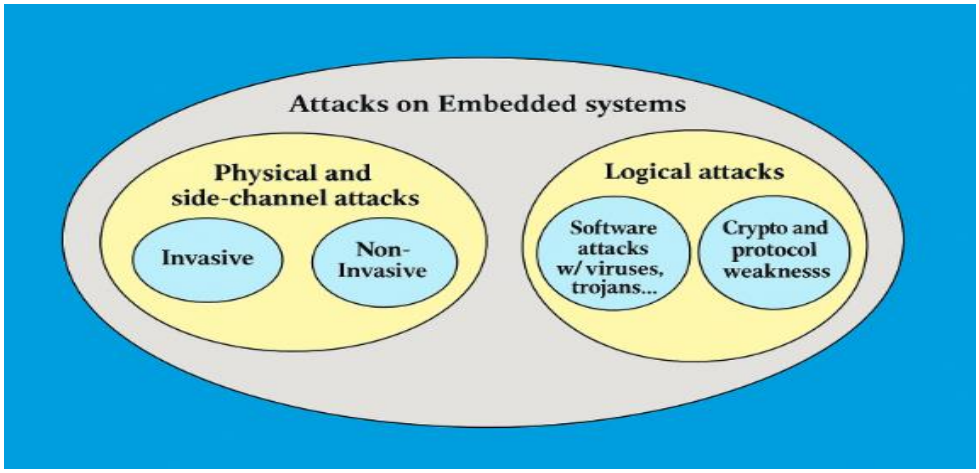

Figure V.1. *Various Types of Attacks on Embedded Systems*

V.3.2 Hardware Vulnerabilities

✓ Physical Attacks

- De-packaging and Physical Attacks on Components: De-packaging involves accessing various levels of an electronic component to analyze or interact with its internal elements. This step allows for the reconstruction of the component's internal architecture, including:
  - Memories.
  - Data bus.
  - Address bus.

- Once de-packaging is performed, techniques such as probing (using probes to physically access the component) can be employed to directly interact with its internal elements.

❖ Complexity of Physical Attacks

Physical attacks vary in terms of complexity and generally require specialized equipment. These attacks are often destructive and serve as a preliminary step for more sophisticated types of attacks.

❖ Countermeasures to Protect Against Physical Attacks

To minimize the risks associated with these attacks, several strategies can be implemented:

- Reinforced packaging to protect the component.
- Specific silicon design:
    - Use of multi-level architectures.
    - Incorporation of random memory structures.
    - Implementation of encrypted communication buses to secure internal exchanges.

These approaches aim to complicate unauthorized access to critical components and protect the sensitive data they contain.

✓ Side-Channel Attacks

Side-channel attacks exploit the physical characteristics of an electronic component to extract sensitive information or disrupt its operation. These attacks are classified into several categories:

❖ Current Attacks: The execution of a program within a component requires power. Variations in current consumption are directly related to the operations performed, which can be exploited to discover sensitive data.
    - Power Analysis Attacks:
        - SPA (Simple Power Analysis): Direct analysis of current fluctuations.
        - DPA (Differential Power Analysis): Uses statistical techniques to correlate power consumption with cryptographic information

❖ Countermeasures:
    - Introduce noise into the power signals or timing.
    - Use a constant or chosen execution flow to mask variations.
    - Hide sensitive information.

✓ Timing Attacks

These attacks exploit variations in the execution time of operations to extract information, such as cryptographic keys.

- The timing variations are influenced by:

- The data being processed.
- The hardware used.
- The software implementation.

- Example: During an exponentiation operation, differences in computation can reveal details about the encryption key.

✓ Fault Injection Attacks

- These attacks disrupt the normal functioning of a system by forcing errors or modifying specific values [44].
- Common methods include:
  - Modifying values in data areas or programs via probing.
  - Disturbing the values of the program counter (PC).
  - Varying the clock or voltage.
  - Using laser beams or heavy ions to induce faults.

❖ Countermeasures:
  - Use of hardened technologies such as SOS/SOI (Silicon-On-Sapphire/Silicon-On-Insulator).

✓ Electromagnetic Attacks
- These attacks analyze the electromagnetic radiation emitted by electronic components. Originally used to study CRT screens, they have now extended to embedded systems.
- Methods similar to current attacks include:
  - SEMA (Simple Electromagnetic Analysis): Direct observation of emissions.
  - DEMA (Differential Electromagnetic Analysis): Statistical analysis of electromagnetic signals.

❖ Countermeasures:
  - Use of a Faraday cage to block electromagnetic emissions.

These attacks represent a growing threat to the security of embedded systems, especially those incorporating cryptographic functions. Innovative solutions must be implemented from the design phase to address them [45].

V.3.3 Software Vulnerabilities

✓ Software Attacks Exploiting Software Vulnerabilities

Although embedded systems are optimized for their specific functions, they remain exposed to various vulnerabilities, often exploited by crackers. These vulnerabilities arise from:

- Hardware design errors that introduce exploitable flaws.
- Software vulnerabilities, such as:

- Backdoors: Hidden access points integrated by programmers for testing purposes, sometimes forgotten in final versions.
- Failure to comply with common standards, leading to inconsistencies.
- Poor programming practices, such as memory overflows (buffer overflow, stack overflow) or the absence of input parameter validation.
- Vulnerable HTTP servers, often designed to be lightweight and therefore less secure.

✓ Examples of Attacks

Certain attacks targeting devices such as routers or network printers include:

- Modification of the IP address via SNMP protocol, exploiting default community strings such as "write".
- Sending specific UDP datagrams to force the closure of a socket port.
- Weak or nonexistent authentication, facilitating unauthorized access.
- Clear-text passwords accessible via SNMP.
- Resetting via SNMP, exposing systems to unintended interruptions.
- Writing long strings via SNMP, causing crashes or unexpected behavior in the device.

✓ Perspectives and Limitations of Exploits

Despite these vulnerabilities, exploits targeting embedded systems remain rarer compared to those affecting traditional systems (PCs, routers, etc.), due to several factors:

- Limited documentation on the internal workings (hardware and software) of embedded systems, making it difficult for attackers to access this information [46].
- Traditional attacks such as shellcode or buffer overflows, common on conventional systems, are nearly nonexistent on embedded systems, except in environments incorporating operating systems like Linux.
- The consequences of attacks on embedded systems often result in a crash or restart, which may be less critical than other types of system compromises.

Although embedded systems seem less targeted than traditional systems, their security should not be overlooked. Developers must implement robust practices, such as input validation, strengthened authentication, and adherence to programming standards, to minimize risks.

V.4 Communication Security

Secure communications are essential in embedded systems, especially when they interact with external networks or other devices. The following are the key aspects to consider:

V.4.1 Communication Security Objectives

✓ Confidentiality: Prevent unauthorized access to exchanged information. Communications must be encrypted to avoid interception by malicious third parties.
✓ Integrity: Ensure that the data exchanged is not modified or altered during transmission.
✓ Authentication: Verify the identity of communicating parties to prevent identity spoofing attacks.

- ✓ Availability: Ensure that communication channels remain operational, even in the event of an attack attempt.

V.4.2 Types of Communication Vulnerabilities

- ✓ Data interception (Man-in-the-Middle attack): An attacker intercepts messages to extract or modify their content.
- ✓ Injection of malicious messages: A third party injects commands or data into the communication flow.
- ✓ Denial of Service (DoS): Saturation of communication channels, rendering services unavailable.
- ✓ Message replay: Intercepted messages are reused to simulate legitimate communication.

V.4.3 Countermeasures and Security Techniques

- ✓ Data Encryption
    - Use of robust cryptographic protocols such as TLS, IPsec, or DTLS to ensure the confidentiality and integrity of transmitted data.
    - Symmetric encryption (e.g., AES) or asymmetric encryption (e.g., RSA, ECC) depending on resource constraints.
- ✓ Authentication
    - Use of digital certificates to authenticate entities.
    - Secure key management protocols to prevent key compromise.
- ✓ Message Integrity
    - Implementation of Message Authentication Codes (MACs) to detect alterations.
    - Data hashing (e.g., SHA-256 or higher) to ensure integrity.
- ✓ Physical Security
    - Limiting physical access to hardware to prevent local attacks, such as cable sniffing or electromagnetic interference.
- ✓ Protection Against Replay Attacks
    - Use of sequence numbers or time tokens to prevent message replay.

V.4.4 Application Example: Security in a Home Automation Network

- Context: An embedded system in a home automation network (e.g., smart thermostats, security cameras) communicates via Wi-Fi or ZigBee.
- Potential Vulnerabilities: Interception of commands, data tampering, or device control takeover.
- Solutions:
    - Data encryption through WPA3 for Wi-Fi.
    - Mutual authentication between devices and the central server.
    - Intrusion detection to identify suspicious activities.

V.4.5 Challenges and Opportunities

- Embedded systems, being resource-constrained, may face additional limitations when implementing robust security mechanisms.
- The adoption of secure communication standards tailored for embedded systems, such as MQTT-S, CoAP, or LoRaWAN, facilitates enhanced security while adhering to energy and power constraints.

V.5 Security of Embedded Systems in General

Embedded systems, also referred to as embedded software, are autonomous entities designed to perform specific, sometimes critical, tasks without requiring human intervention. These systems interact directly with their environment, whether physical or computational.

They must meet stringent functional requirements that influence their design, robustness, and ability to execute tasks with limited resources. These constraints often include strict timing requirements and energy consumption limitations. Their behavior must be meticulously controlled, ensuring a high level of security and safety throughout their lifecycle.

These systems must not only be protected against potential threats but also guarantee optimal operation, both in their internal processes and in their interactions with the external environment. As autonomous systems, they must demonstrate resilience to unforeseen events, necessitating robust security and safety properties.

To ensure this robustness, it is essential to implement a local protection layer based on mathematical, physical, and security models. This "shell" enables the system to :

- ✓ Perceive: Observe, detect, locate, and diagnose anomalies.
- ✓ Analyze: Assess unforeseen events, dangers, and random occurrences.
- ✓ React: Adapt its behavior by correcting, tolerating, or maintaining stability in the face of deviations from the nominal state. If necessary, the system can degrade or self-terminate to prevent more severe consequences.

V.5.1 Phases of Security and Safety

The security and safety of embedded systems are developed through several key stages:

- ✓ System Specification: This involves mastering the complexity and architecture of the system. A clear separation between the autonomous module and its infrastructure is critical, as is managing energy consumption and communication flows. Proven methodologies are applied at this stage to specify security and safety aspects.
- ✓ System Design: This relies on advanced techniques such as modeling, abstraction, and methods based on mathematical approaches.
- ✓ Implementation: This phase requires rigorous resource management and limited communication with peripherals to conserve resources.
- ✓ Validation: The system undergoes formal verification, simulations, and thorough testing. Security and safety assessment are conducted to ensure compliance.

The design of embedded systems must incorporate a variety of disciplines, including mathematics, computer science, electronics, and architecture. It must also comply with interoperability standards while considering economic constraints to ensure a viable and efficient deployment.

V.6 Some Critical Embedded System Accidents

V.6.1 The Ariane 501 Launch Failure

On June 4, 1996, the maiden flight of the Ariane 5 launcher ended in failure. Approximately 40 seconds after ignition, the rocket broke apart, resulting in its immediate self-destruction (see John Rushby's message on comp.risks). On June 13, 1996, an independent group of scientists led by J. L. Lions was tasked with investigating the cause of this failure. The report of their investigation was made public on July 19, 1998.

```
Newsgroups: comp.risks
Subject: Ariane 5 failure
John Rushby <RUSHBY@csl.sri.com>
Wed 5 Jun 96 14:54:47-PDT

>From cnn's web page www.cnn.com:

Faulty computer blamed in Ariane rocket failure

Experts studying the moments before the Ariane-5 rocket explosion say
faulty computer software may be to blame for the rocket veering off
course. Apparently, the rocket was misfed information that made it think
it was not following the right path. The rocket then changed direction,
causing the upper part to began to break apart.
```

Figure V.2. *Initial announcement suspecting a software issue with Ariane 5.*

The software error originated from a module that had been designed and verified for the Ariane 4, but had not been verified for Ariane 5. This module contained a function that was only useful when the rocket was on its launch platform and should have been deactivated afterward. However, this function was still active during the flight, and it was this very function that caused the loss of Ariane 5.

Figure V.3. *The accident during the inaugural flight of the Ariane 5 rocket.*

The conclusions of the report on the Ariane V failure largely focus on the methodology used for the qualification and validation of the computer systems, which were deemed insufficient.

V.6.2 The Therac-25

The Therac-25 is a medical device with a user-friendly interface designed for cancer treatment. It emits either electron beams (for superficial tissues) or photon beams (very high energy for internal tissues).

- ✓ 1976: First prototype used in a hospital setting.
- ✓ 1982: First sale.
- ✓ July 1985: Similar incident with a tumor in the pelvis, patient died 5 months later. The analysis concluded a failure of micro-switches.
- ✓ December 1985, March 1986: Two new incidents, one fatal. The analysis concluded the presence of electric shocks due to a short circuit.
- ✓ April 1986: A new incident, one additional fatality, but the second fatality was that of an operator who could replicate the error. The company acknowledged the software fault. The error was corrected.
- ✓ January 1987: A final incident, one last fatality. New software errors were discovered, leading to the cessation of Therac-25 operations.

Figure V.4. *The Therac-25 medical device*

The assessment of this failure can be summarized as follows:

- 6 confirmed accidents.
- 4 fatalities.
- 2 years to identify the software issue and withdraw the machines.

V.6.3 Phobos 1

When it comes to space, almost all quantities considered are multiplied by an "astronomical" factor. Distances, speeds, costs... This is especially true since it sometimes takes very little to cause a mission to fail. On Saturday, September 10, 1988, the Russian control center for the Phobos 1 mission to Mars relocated from Crimea to the outskirts of Moscow. A 20 to 30-page program was sent to Phobos 1 to make adjustments related to this change.

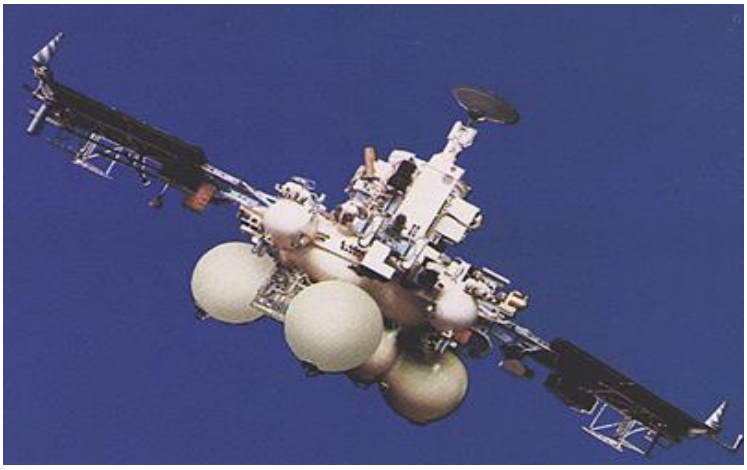

Figure V.5. The Phobos 1 satellite

After receiving the program, the probe, instead of acknowledging the changes, reoriented its solar panels away from the Sun. Due to communication latency, by the time the control center received the acknowledgment message from the probe regarding the solar panel movement, and the control center's response requesting immediate correction of the panel positioning reached the probe, it was too late. The probe had exhausted its batteries and was no longer responding.

The total cost of this mission was the equivalent of one billion dollars.

CONCLUSION

Embedded systems represent a cornerstone of modern electronics and digital intelligence. These specialized computing systems, designed to perform dedicated functions within larger systems, are omnipresent in today's world—from simple household appliances to complex medical equipment, automotive control units, industrial automation, and intelligent communication devices. Understanding their structure, functionality, and constraints is essential for engineers and researchers involved in designing tomorrow's smart systems.

This course has aimed to build a solid foundation in embedded system concepts by covering a broad range of topics, including the architecture of embedded devices, classification of embedded systems, differences between microcontrollers and microprocessors, and the integration of software and hardware. Emphasis was placed on the application-specific nature of embedded systems, their limited resources, and their real-time operational requirements, all of which distinguish them from general-purpose computing systems.

In addition to exploring the theoretical underpinnings, the course has provided insight into practical aspects of embedded system design. Topics such as input/output interfacing, memory organization, power optimization, and the role of real-time operating systems (RTOS) were discussed in detail. Students were also introduced to the scheduling of real-time tasks, interrupt handling mechanisms, and system-level considerations essential for building reliable and time-sensitive embedded applications.

One of the critical aspects highlighted throughout the course is the growing importance of embedded systems in the context of the Internet of Things (IoT), where billions of devices must operate efficiently, securely, and autonomously. As such, this field not only demands a strong grasp of electronic and computer engineering principles but also a forward-looking mindset capable of adapting to rapidly evolving technologies.

By completing this course, students are expected to have developed the analytical skills and technical proficiency necessary to engage with embedded system projects in both academic and industrial contexts. They are now better prepared to contribute to the design and development of intelligent, efficient, and reliable systems that form the backbone of next-generation technologies.

In conclusion, as embedded systems continue to expand into every aspect of human activity, the knowledge acquired through this course serves as a vital stepping stone toward innovation, research, and impactful technological advancement in an increasingly connected world.

References

[1] RODRÍGUEZ J.P., Thermal-aware schedulability analysis for modern real-time computing systems, 2024.

[2] EDWARD LEE H.G., SCHÄTZ B.R.B., Model-based engineering of embedded real-time systems, 2010.

[3] AHMAD I., RANKA S. (eds.), Handbook of Energy-Aware and Green Computing, Volume 1, CRC Press, 2012.

[4] HASSAN G., Democratizing The Internet Of Things Through Platform Virtualization (Doctoral dissertation, Queen's University (Canada)), 2022.

[5] QIU M., LI J., Real-time embedded systems: optimization, synthesis, and networking, CRC Press, 2011.

[6] LEE I., LEUNG J.Y., SON S.H. (eds.), Handbook of real-time and embedded systems, CRC Press, 2007.

[7] STOJILOVIĆ M., RASMUSSEN K., REGAZZONI F., TAHOORI M.B., TESSIER R., "A Visionary Look at the Security of Reconfigurable Cloud Computing", Proceedings of the IEEE, 2023.

[8] ALSHEIKHY A.A., High Performance Embedded Systems, University of Connecticut, 2016.

[9] DAWOUD S., PEPLOW R., Digital system design-use of microcontroller, Taylor & Francis, 2010.

[10] ELVANY HUGUE M.M., STOTTS P.D., "Guaranteed task deadlines for fault-tolerant workloads with conditional branches", Real-Time Systems, vol. 3, no. 3, 1991, pp. 275-305.

[11] SHRIVASTAV V., Towards High-Speed Networking in the Post-Moore Era, Cornell University, 2020.

[12] MEZA J.R., Hand held workstation: a guide to embedding the Linux kernel, California Polytechnic State University, 1996.

[13] ROY A., Improving Energy Efficiency and Quality-of-Control Metrics in Reliable Multiprocessor Real-Time Systems (Doctoral dissertation, George Mason University), 2021.

[14] CHAKRAVARTHI V.S., KOTESHWAR S.R., System on Chip (SOC) Architecture: A Practical Approach, Springer Nature, 2023.

[15] AZAR N.H., Designing Industrial Pick and Place Robotic Arm with AVR Microcontroller (Master's thesis, Khazar University (Azerbaijan)), 2024.

[16] SUYYAGH A., Towards energy-efficient real-time computing in embedded systems, McGill University (Canada), 2019.

[17] NAIA N.P.D., Real-Time Linux and Hardware Accelerated Systems on QEMU (Master's thesis, Universidade do Minho (Portugal)), 2015.

[18] COTTET F., GROLLEAU E., Systèmes temps réel de contrôle-commande : conception et implémentation, Dunod, 2005.

[19] POTTEIGER B., A Moving Target Defense Approach Towards Security and Resilience in Cyber-Physical Systems (Doctoral dissertation, Vanderbilt University), 2019.

[20] CEDENO W., LAPLANTE P.A., "An overview of real-time operating systems", JALA: Journal of the Association for Laboratory Automation, vol. 12, no. 1, 2007, pp. 40-45.

[21] PIRRI ARDIZZONE M.F., MENTUCCIA I., STORRI S., "Ambient Intelligence: Impact on Embedded System Design", Ambient Intelligence: Impact on Embedded System Design, Springer, 2003, pp. 131-158.

[22] CHANG H.P., CHANG R.I., SHIH W.K., CHANG R.C., "GSR: A global seek-optimizing real-time disk-scheduling algorithm", Journal of Systems and Software, vol. 80, no. 2, 2007, pp. 198-215.

[23] ZURAWSKI R., Embedded Systems Handbook, CRC Press, 2003.

[24] OBERMAISSER R., “Time-triggered communication”, in ZURAWSKI R. (ed.), Embedded Systems Handbook, CRC Press, 2017, pp. 14-1.

[25] LEUNG J.Y., Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Chapman and Hall/CRC, 2004.

[26] GANSSLE J., Embedded Systems Dictionary, CRC Press, 2003.

[27] CHEN F., KALOGERAKI V., “A Soft Real-Time Agent-Based Peer-to-Peer Architecture”, in Intelligent Systems Design and Applications, Springer, 2003, pp. 493-502.

[28] PLOEREDERER E., GARRIDO J., SANDÉN B.I., GHORBEL A., AMOR N.B., JALLOULI M., COVER I.B., “ADA USER”, Ada User Journal, vol. 38, no. 2, 2017.

[29] SULLIVAN M.L., Pediatric Oral Health Literacy Among Caregivers of Young Children (Doctoral dissertation, Old Dominion University), 2024.

[30] VUPPALAPATI C., Democratization of Artificial Intelligence for the Future of Humanity, CRC Press, 2021.

[31] ISHAK M.K., HWAN O.J., JIASHEN T., MAT ISA N.A., “Automated Compilation Test System for Embedded System”, Makara Journal of Technology, vol. 22, no. 3, 2019, p. 1.

[32] YI Q., PUXIANG X., ZHU T., The Design and Implementation of the RT-thread Operating System, Auerbach Publications, 2020.

[33] HAN C.C., LIN K.J., HOU C.J., “Distance-Constrained Scheduling and its Applications to Real-Time Systems”, IEEE Transactions on Computers, vol. 45, no. 7, 1996, pp. 814-826.

[34] CZAJKOWSKI K., DEMIR A.K., KESSELMAN C., THIEBAUX M., “Practical Resource Management for Grid-Based Visual Exploration”, in Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing, IEEE, 2001, pp. 416-423.

[35] PORTUGAL R., Media Gateway Utilizando um GPU (Master's thesis, Universidade de Aveiro (Portugal)), 2012.

[36] ZHANG F., WU Q.Z., REN G.Q., “A Real-Time Capture and Transport System for High-Resolution Measure Image”, in 2010 International Conference on Intelligent Computation Technology and Automation, vol. 1, IEEE, 2010, pp. 306-309.

[37] MESMAN B., et al., “Embedded Systems Roadmap”, Vision on Technology for the Future of PROGRESS, STW Technology Foundation IPROGRESS, 2002.

[38] COMBÉFIS S., « Programmation de systèmes embarqués : Introduction aux systèmes embarqués », ECAM Brussels Engineering School, 2017.

[39] GUPTA A., CHANDRA A.K., LUKSCH P., Real-Time and Distributed Real-Time Systems: Theory and Applications, CRC Press, 2016.

[40] HUSSAIN I., Improving the Analysis Techniques for Mixed-Criticality Multicore Systems and Controller Area Networks by Leveraging Periodic Load Patterns (Doctoral dissertation, Universidade do Porto (Portugal)), 2024.

[41] HERRMANN D.S., A Practical Guide to Security Engineering and Information Assurance, Auerbach Publications, 2001.

[42] SOUSA P., HACC-V: Hardware-Assisted Cryptographic Coprocessor for RISC-V (Master's thesis, Universidade do Minho (Portugal)), 2022.

[43] RAVI S., RAGHUNATHAN A., KOCHER P., HATTANGADY S., “Security in Embedded Systems: Design Challenges”, ACM Transactions on Embedded Computing Systems (TECS), vol. 3, no. 3, 2004, pp. 461-491.

[44] RAVI S., RAGHUNATHAN A., CHAKRADHAR S., “Tamper Resistance Mechanisms for Secure Embedded Systems”, in Proceedings of the 17th International Conference on VLSI Design, IEEE, 2004, pp. 605-611.

[45] ZHANG X., Confidentiality and Privacy-Preserving: Intertwining Deep Learning and Side-Channel Analysis (Doctoral dissertation, Northeastern University), 2024.

[46] LIM H., YU H., SUH T., “Using Virtual Platform in Embedded System Education”, Computer Applications in Engineering Education, vol. 20, no. 2, 2012, pp. 346-355.