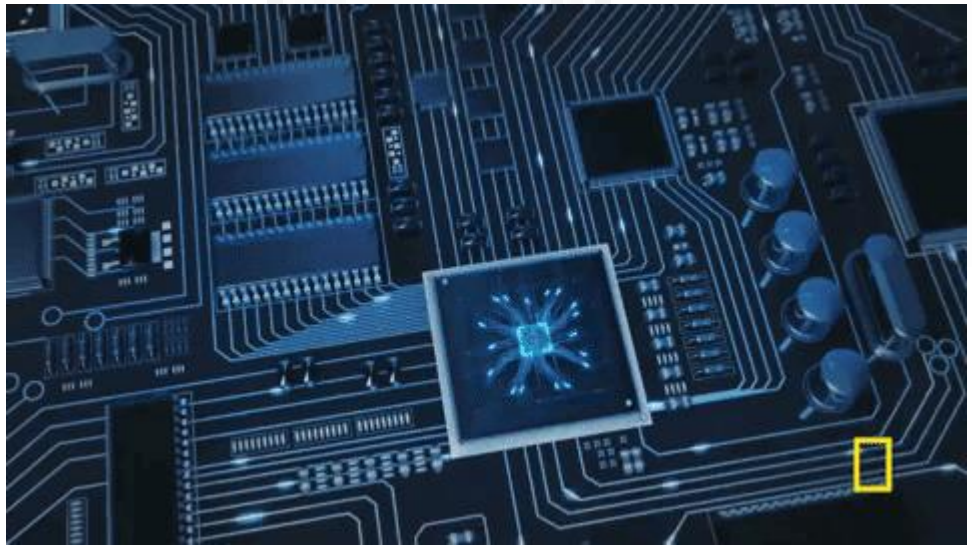
ATME College of Engineering

Department of EEE
Emitting Elite Energy

BEE613B
Embedded Systems Design
Module-4: Embedded Firmware Design and Development



Presented by,

Mr.Shreeshayana R

Assistant Professor

Electrical and Electronics Engineering

ATME College of Engineering, Mysuru

Contents

- Embedded Firmware Design Approaches,
- Embedded Firmware Development Languages (Chapter 9 – Text 1: 9.1, 9.2)
- Embedded System Development Environments: Types of files generated on cross compilation (only explanation – programming codes need not be dealt),
- Disassemble/decompliler,
- Simulators,
- Emulators and Debugging (Chapter 13 – Text 1: 13.2, 13.3,13.4)

Introduction

Embedded systems are purpose-built computing systems designed to perform dedicated functions within a larger system.

At the heart of every embedded product lies its **firmware**—the software that directly controls the hardware.

The efficiency, responsiveness, and reliability of an embedded system are largely influenced by how its firmware is designed.

Introduction

Embedded Firmware

Embedded firmware is the flash memory chip that stores specialized software running in a chip in an **embedded** device to control its functions.

Introduction

- Embedded firmware refers to the **control algorithm** (Program Instructions) and the configuration settings that an **embedded system developer** dumps into the code(program) memory of the embedded system.
- Responsible for controlling the various peripherals of the first embedded hardware.
- Firmware is considered as the master brain of the embedded system.
- In case of firmware corruption the firmware should be reloaded to bring back the embedded product to normal functioning.

Introduction

Essential Skills for Embedded Developers

To excel in embedded systems, consider developing proficiency in the following areas:

- **Programming Languages**: C, C++, and Embedded C
- **Microcontrollers & Processors**: Experience with ARM Cortex, AVR, PIC, or 8051
- **Real-Time Operating Systems (RTOS)**: Knowledge of FreeRTOS, VxWorks, or Embedded Linux
- **Hardware Interfaces**: Familiarity with UART, SPI, I2C, and GPIO
- **Debugging Tools**: Proficiency with JTAG, oscilloscopes, and logic analyzers
- **Version Control Systems**: Experience with Git
- **Software Development Life Cycle (SDLC)**: Understanding of development methodologies
- **Scripting Languages**: Basic knowledge of Python or Bash for automation

Introduction

Tip for Freshers:

- Target companies offering training or internships.
- Learn tools like **Keil, MPLAB, STM32Cube**, and languages like **C/C++, Python**.
- Basic understanding of **RTOS, I2C/SPI/UART protocols**, and **microcontrollers (8051, ARM Cortex, etc.)** is essential.

Tips for Aspiring Embedded Developers

- **Build a Strong Portfolio**: Work on personal projects involving microcontrollers like Arduino or Raspberry Pi.
- **Certifications**: Consider certifications in Embedded Systems or RTOS to enhance your resume.
- **Networking**: Join local tech meetups or online forums related to embedded systems.
- **Stay Updated**: Keep abreast of the latest trends and technologies in embedded systems.

Introduction

Experience Level	Typical Salary Range (INR/Year)	Notes
Fresher (0–1 year)	₹3.0 LPA – ₹6.0 LPA	Depends on company (startups vs MNCs), skillset (C, C++, RTOS, etc.)
1–3 years	₹5.0 LPA – ₹9.0 LPA	Strong knowledge of microcontrollers, firmware, communication protocols
3–6 years	₹8.0 LPA – ₹15.0 LPA	Added responsibilities like design, debugging, team collaboration
6–10 years	₹14.0 LPA – ₹25.0 LPA	Involves project ownership, system architecture, and client interaction
10+ years	₹25.0 LPA – ₹45.0+ LPA	Often in roles like Tech Lead, Embedded Systems Architect, or Manager

Firmware design for embedded systems can vary significantly depending on several factors, such as:

- **Complexity of functions** to be executed
- **Response time or real-time constraints**
- **Cost and resource availability**
- **Type of microcontroller or processor used**

Depending on these requirements, two primary approaches are widely used in embedded firmware development:

1.Conventional Procedural Based Firmware Design, also referred to as the **Super Loop Model**

2.Embedded Operating System (OS) Based Design, which can use either a **General-Purpose OS (GPOS)** or a **Real-Time Operating System (RTOS)**

Super Loop Based Firmware Design (Conventional Procedural Model)

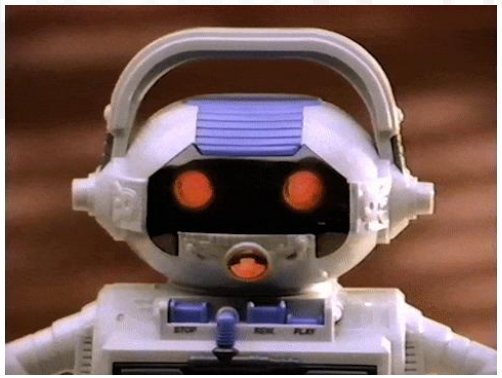
Also known as: Bare-metal programming, Infinite loop model

Key Features:

- No **Operating System required**
- Simple and low-cost implementation
- Tasks are executed **sequentially in a hardcoded**, **infinite loop (while(1))**
- Suitable for **non-time-critical** applications
- Priority and order of tasks are fixed
- Failure in one task affects the whole system

Application Use Cases:

- Electronic toys
- Basic card readers
- Simple display systems
- Entry-level embedded products using 8051 or AVR controllers



⚠ Limitations:

- No dynamic task scheduling
- No real-time guarantees
- Difficult to scale for complex systems
- Risk of system hang-ups (needs Watchdog Timer)
- Limited flexibility in responding to external asynchronous events (e.g., key press)

Basic structure of a **Super Loop Based Firmware Design** in C

```
void main()
{
   Configurations();
   Initializations();
   while (1)
{
      Task_1();
      Task_2();
      ...
      Task_n();
   }
}
```

Execution Flow:

1.Initialization of hardware (memory, I/O, peripherals)
2.Start Task 1 → Task 2 → … → Task N
3.Loop back to Task 1 and repeat (infinite loop)

Practical example of a super loop firmware structure for a **temperature monitoring system using an LM35 sensor and displaying it on an LCD**:

```
#include <lcd.h>
#include <adc.h>

void Configurations()
{
   // Configure clock, ports
   ADC_Config();    // Set up ADC for LM35
   LCD_Init();      // Initialize the LCD
}

void Initializations() {
   LCD_Clear();
   LCD_SetCursor(0, 0);
   LCD_Print("Temp Monitor");
   _delay_ms(1000);
}
```

```
void Task_ReadTemperature()
{
    int adc_value = ADC_Read(0); // Read from channel 0
    float temperature = (adc_value * 5.0 * 100.0) / 1024; //
Convert to °C
}


void Task_DisplayTemperature(float temp) {
    char buffer[16];
    sprintf(buffer, "Temp: %.2f C", temp);
    LCD_SetCursor(1, 0);
    LCD_Print(buffer);
}
```

```
void main()
{
    float current_temp;

    Configurations();
    Initializations();

    while (1) {
        current_temp = ADC_Read(0) * 5.0 * 100.0 / 1024;
        Task_DisplayTemperature(current_temp);
        _delay_ms(1000); // 1 second delay
    }
}
```

Embedded OS-Based Firmware Design

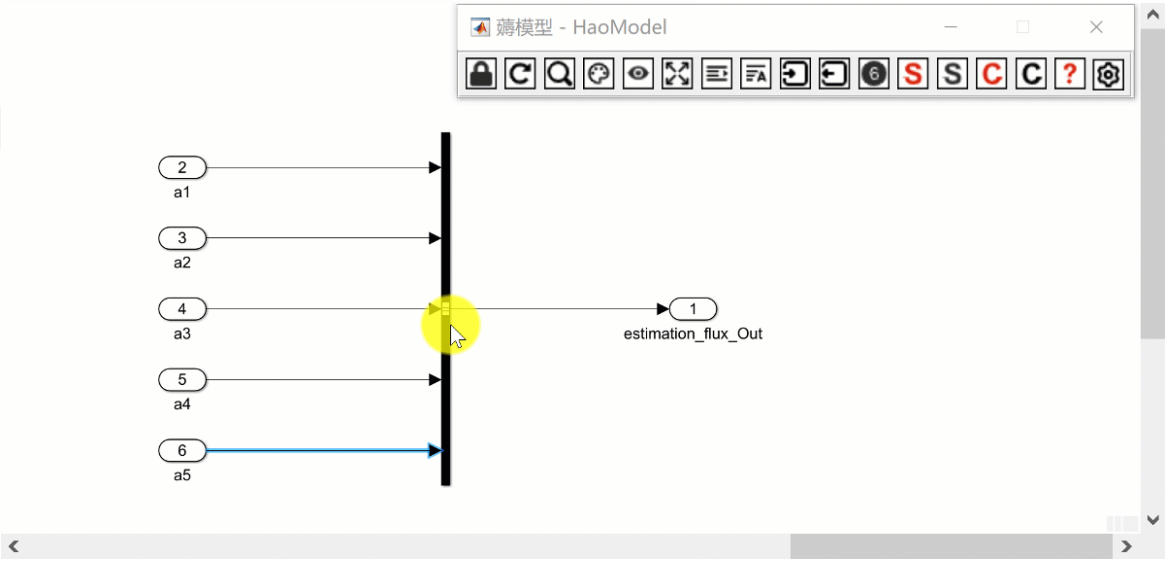
Types:

- **GPOS** (General Purpose OS): e.g., Windows XP Embedded, Linux
- **RTOS** (Real-Time OS): e.g., VxWorks, µC/OS-II, FreeRTOS, ThreadX, Embedded Linux

What Is VxWorks?

FreeRTOS™ - FreeRTOS™

Azure RTOS Is Now Eclipse ThreadX | The Eclipse Foundation

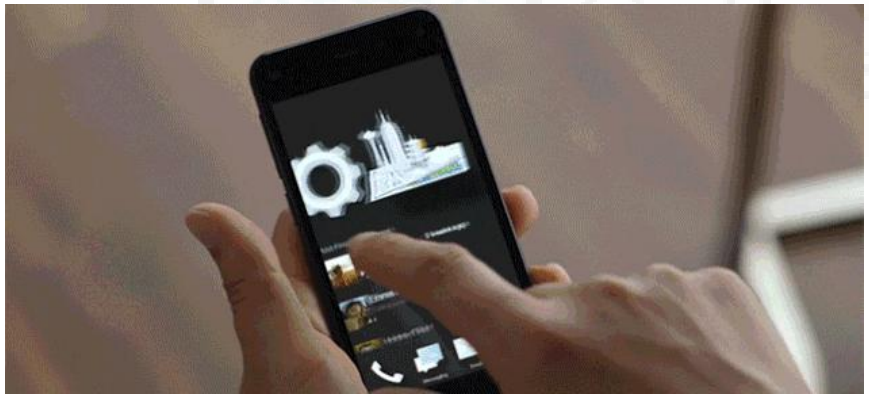


✅ Key Features:

- **OS handles task scheduling**, inter-task communication, and timing
- Suitable for **time-critical** or **real-time systems**
- Preemptive multitasking
- APIs support hardware abstraction
- Modular and scalable

⚙️ Components in RTOS:

- Real-Time Kernel
- Scheduler
- Task/Thread management
- Inter-task communication (e.g., Semaphores, Queues)
- Timers and memory management



⚠️ Limitations:

- Higher memory and processing overhead
- Requires deeper knowledge of OS internals
- More complex debugging

🧰 Application Use Cases:

- Smartphones and PDAs
- Industrial automation systems
- Medical equipment
- POS terminals
- Automotive applications

Application: Smart Home Automation

Objective:

•Control light based on LDR

•Read temperature using DHT11

•Send data to WiFi (ESP8266) using UART

This system uses an RTOS approach to handle multiple tasks **concurrently**

Why RTOS Here?

•Light control happens **independently** of temperature reading.

•Each task runs **periodically**, improving performance.

•Ideal for **multi-sensor IoT nodes**.

```
#include <FreeRTOS.h>
#include <task.h>
#include "ldr.h"
#include "dht11.h"
#include "uart.h"
#include "relay.h"

int ldr_value = 0;
float temperature = 0;
float humidity = 0;
```

```
// Task 1: Light control based on LDR
void Task_LightControl(void
*pvParameters) {
    while (1) {
        ldr_value = LDR_Read();
        if (ldr_value < 300)
{
            Relay_On();  // It's dark
        } else {
            Relay_Off(); // It's bright
        }
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

```
// Task 2: Read Temperature & Humidity
void Task_ReadDHT11(void *pvParameters) {
    while (1) {
        DHT11_Read(&temperature, &humidity);
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

// Task 3: Send data to ESP8266 over UART
void Task_SendToWiFi(void *pvParameters)
{
    char buffer[64];
    while (1) {
        sprintf(buffer, "Temp:%.2f,Hum:%.2f,LDR:%d\n", temperature, humidity, ldr_value);
        UART_SendString(buffer);
        vTaskDelay(pdMS_TO_TICKS(3000));
    }
}
```

```
void main() {
    LDR_Init();
    DHT11_Init();
    UART_Init(9600);
    Relay_Init();

    xTaskCreate(Task_LightControl, "Light", 128, NULL, 2, NULL);
    xTaskCreate(Task_ReadDHT11, "DHT11", 128, NULL, 1, NULL);
    xTaskCreate(Task_SendToWiFi, "WiFi", 128, NULL, 1, NULL);

    vTaskStartScheduler();
}
```

Embedded Firmware Development Languages

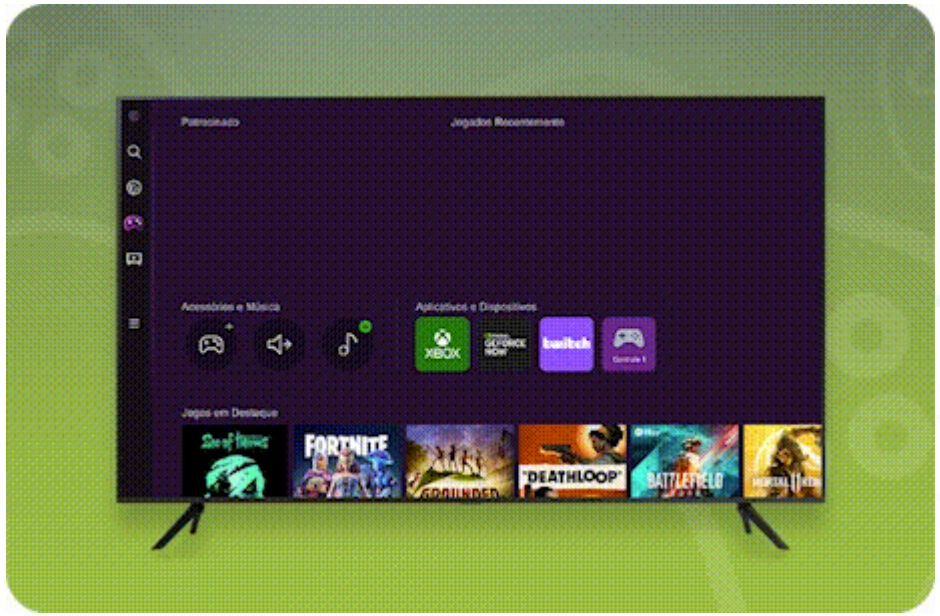
Embedded firmware development languages involve **creating software** that is **built into hardware devices**.
This software is responsible for **controlling the device's functions** and ensuring that it performs its intended tasks correctly.

Importance of Embedded Firmware Development

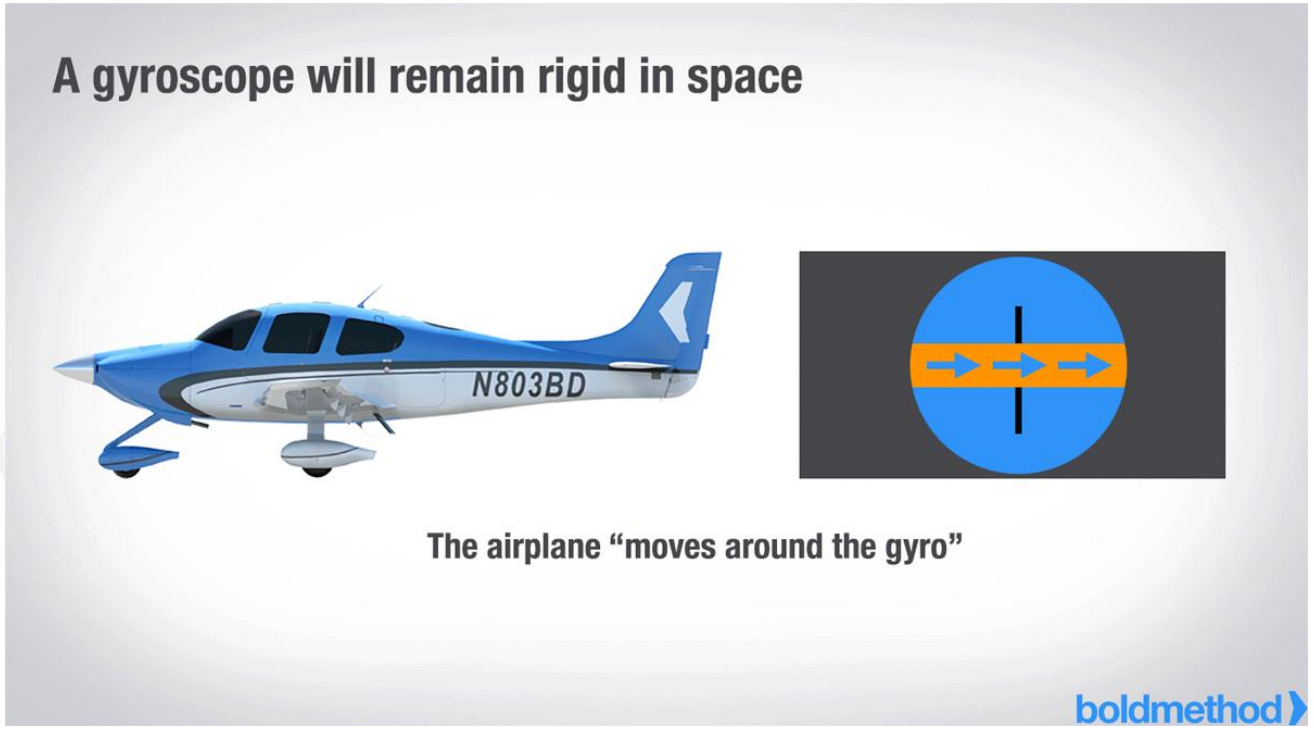
Consumer Electronics: Embedded firmware powers numerous consumer devices, including **smartphones, smart TVs, wearables, and home automation systems**. It ensures smooth operation, connectivity, and user-friendly interfaces.

Automotive and Transportation: Modern vehicles heavily rely on embedded firmware to control engine management, navigation systems, driver-assistance features, and infotainment systems. Firmware ensures the safety, efficiency, and comfort of passengers.

Importance of Embedded Firmware Development



Importance of Embedded Firmware Development

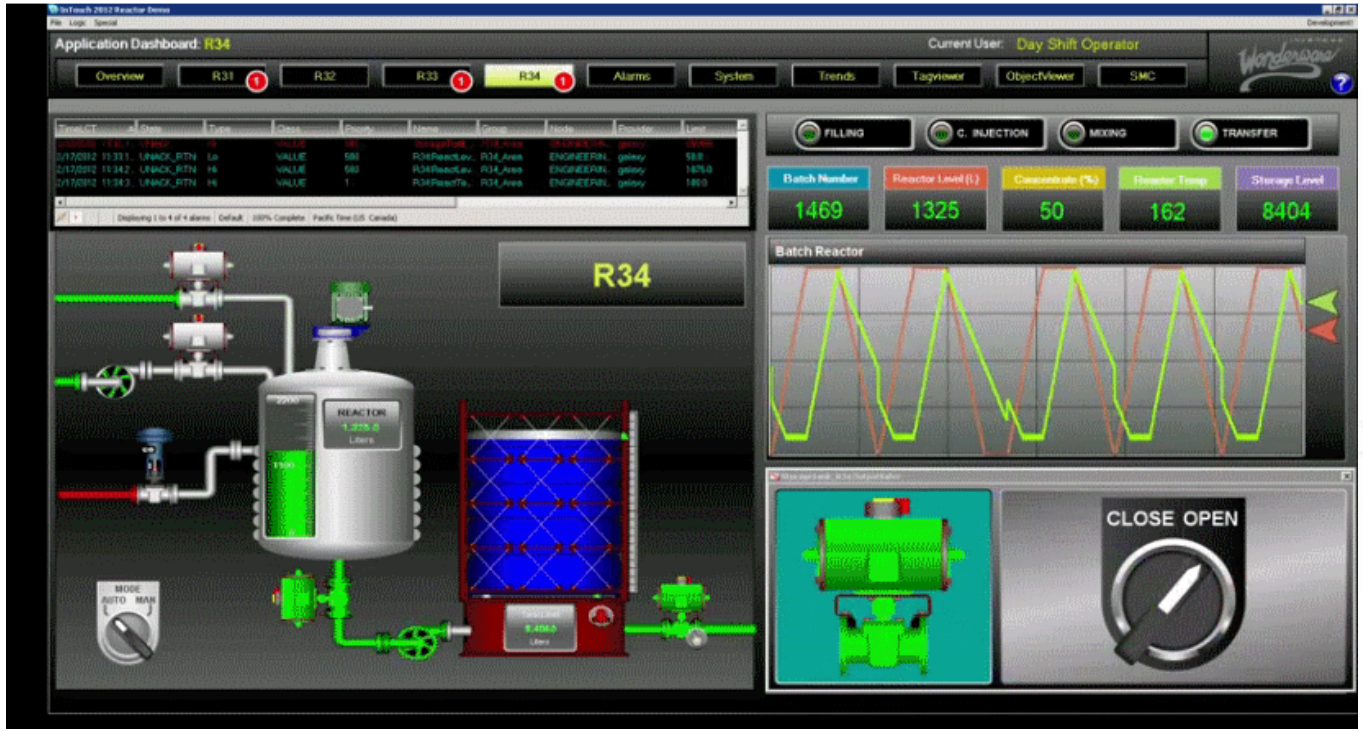


Importance of Embedded Firmware Development

Industrial Automation: Embedded firmware is integral to industrial control systems, including **programmable logic controllers (PLCs) and supervisory control and data acquisition (SCADA) systems**. It enables real-time control and monitoring of industrial processes.

Internet of Things (IoT): In the IoT ecosystem, embedded firmware connects and manages networked devices, facilitating data collection, analysis, and remote control. It enables seamless communication and interoperability between devices.

Importance of Embedded Firmware Development



Importance of Embedded Firmware Development

Medical Devices: Firmware, or embedded software, is essential in medical devices: it ensures that measurements are accurate and controls function properly.

Aerospace and Defense: Embedded firmware is critical for avionics systems, unmanned aerial vehicles (UAVs), and defense applications. It provides reliable control, navigation, and communication capabilities.

Firmware in embedded systems can be developed using:

Low-Level Languages (e.g., **Assembly**) – Specific to the target processor or controller.

- **High-Level Languages** (e.g., **C, C++, Java**) – Independent of processor/controller.
- **Combination of Both** – Common in real-world systems for performance and ease.

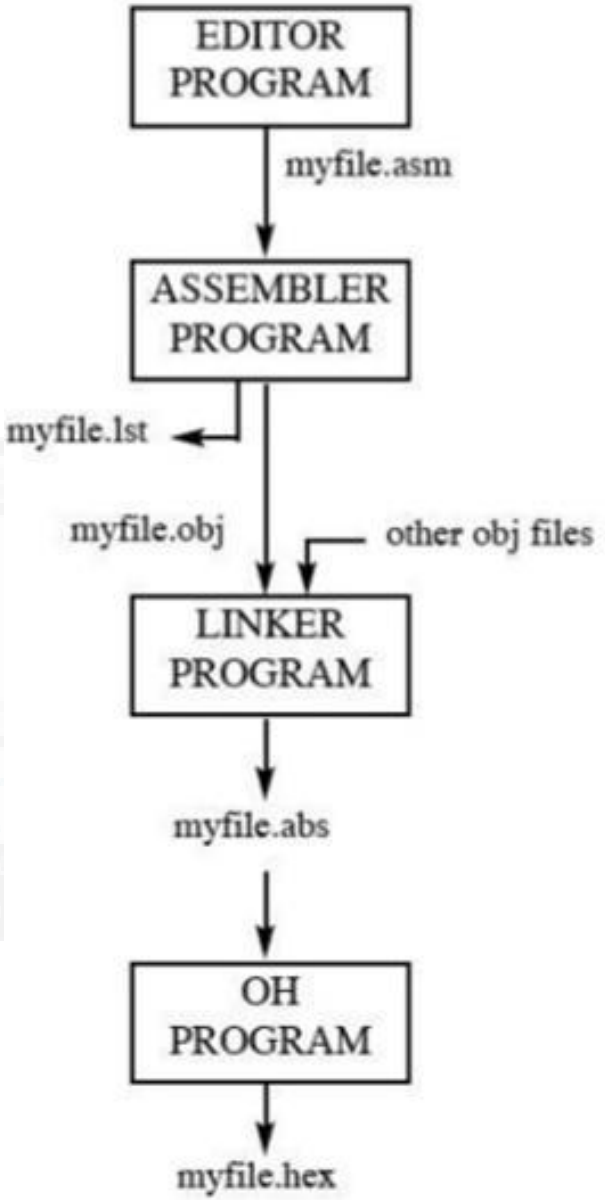
Low-Level Languages (e.g., Assembly)

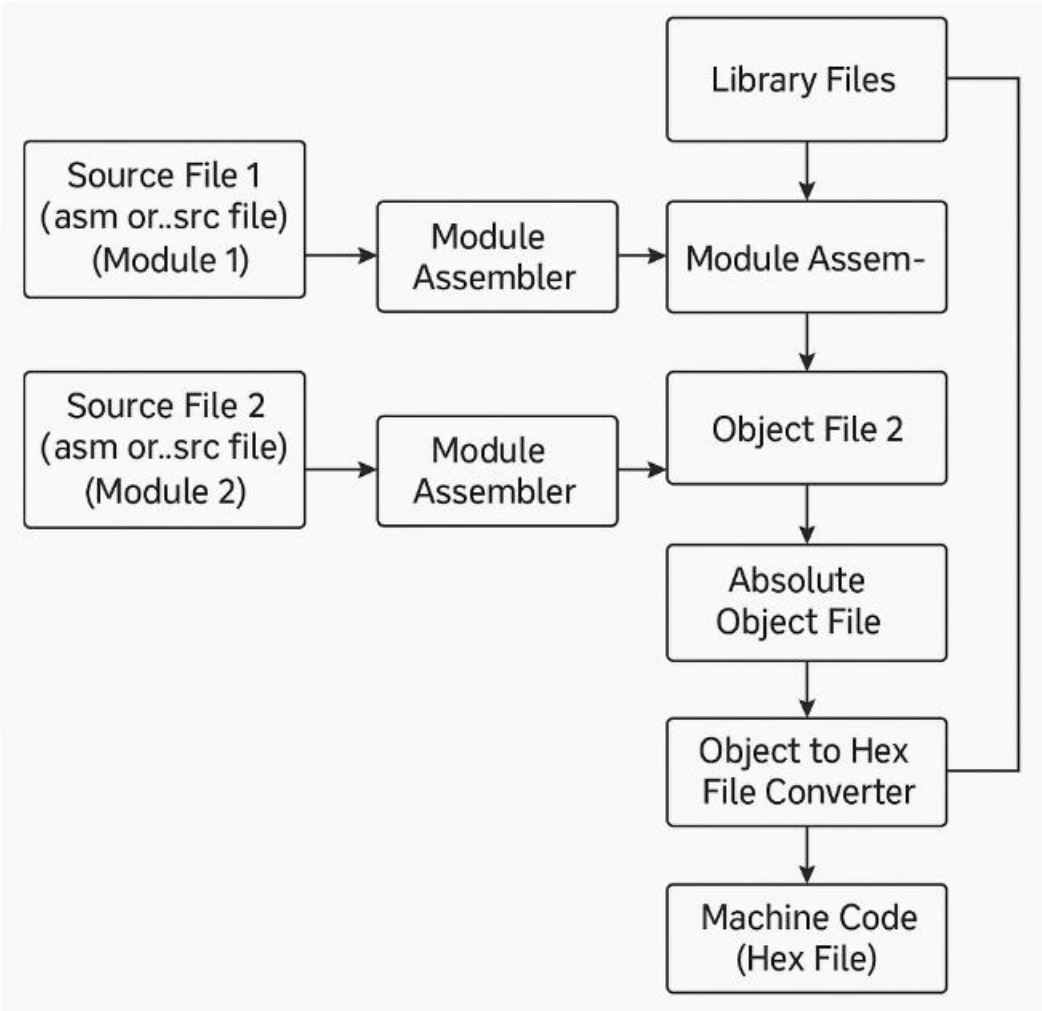
What is Assembly Language?

- A **human-readable** representation of **machine language**.
- Uses **mnemonics** to represent **binary opcodes**.
- **Processor-specific**, unlike high-level languages.
- Converts to machine code via an **assembler**.

Historical Context

- Early programs (e.g., 1990s console games like **NBA Jam**) were mostly written in assembly.
- Still used in **system-level programming**, such as:
  - **Device drivers**
  - **OS-hardware interfaces**


EDITOR PROGRAM
myfile.asm
ASSEMBLER PROGRAM
myfile.lst
myfile.obj
other obj files
LINKER PROGRAM
myfile.abs
OH PROGRAM
myfile.hex


Library Files
Source File 1 (asm or..src file) (Module 1)
Module Assembler
Module Assem-
Source File 2 (asm or..src file) (Module 2)
Module Assembler
Object File 2
Absolute Object File
Object to Hex File Converter
Machine Code (Hex File)

Instruction Format

Each Assembly line usually contains:

LABEL OPCODE OPERAND ;COMMENTS

Example: MOV A, #30 ; Move immediate value 30 to Accumulator

- **MOV** = Opcode
- **A** = Register
- **#30** = Operand

Machine Code Representation:

01110100 00011110

- First byte: Opcode
- Second byte: Operand

Implicit Operand Example:

- Machine Code: 00000100

Instruction Types

Type	Example	Description
Single operand	MOV A, #30	One operand (value)
Implicit	INC A	Operand is assumed
Dual operand	LJMP 16-bit addr	Two operands (target address split into 2 bytes)

Labels

•Identifiers for memory locations, jump points, subroutines.
•Syntax: Must start with a valid character, can include numbers and underscore, end with a colon (:).

Why use Labels?

•Avoids hardcoding addresses.
•Easier debugging and code management.
•Automatically handled by assembler.

Sample Subroutine

```
//Subroutine for Delay
DELAY:
   MOV R0, #255    ; Load R0
RELAX:
   DJNZ R1, RELAX  ; Decrement R1, jump if not zero
   RET             ; Return
```

Usage in Main Program:

```
LCALL DELAY       ; Call DELAY subroutine
```

Assembler Directives

- Not instructions for the processor but for the assembler.
- Example: ORG 0100H – Tells the assembler to start placing instructions from memory address 0100H.

⚠️ Caution:

Avoid manually assigning addresses. Use **labels** instead to prevent memory overwrite errors and maintain code flexibility.

Merits of Assembly Language

- **Faster** execution
- **More control** over hardware
- **Compact** code (optimized memory use)

❌ Demerits

- **Processor-dependent** (not portable)
- **Complex and error-prone**
- **Long development time**

High Level Language Based Development (e.g., C, C++)

Why Use High-Level Languages:

•Assembly language is time-consuming, non-portable, and requires deep processor knowledge.

•High-level languages like **C** and **C++** are easier to use and widely supported with cross-compilers.

Development Flow:

•Code is written in .c or .cpp files using text editors or IDEs.

•**Cross-compilers** convert high-level code to processor-specific machine code (e.g., Keil's C51 for 8051 microcontrollers).

•The rest of the development process is similar to assembly language development.

Advantages:

•**Reduced Development Time:** Minimal hardware knowledge needed; faster learning curve.

•**Developer Independence:** Universal syntax; easier team collaboration and maintenance.

•**Portability:** Code can be reused across different processors with minimal changes.

Limitations:

- **Less Optimization:** Some compilers may generate inefficient or bulky machine code.
- **Lower Efficiency in Critical Tasks:** Not ideal for timing-critical, low-level hardware control.
- **Higher Cost:** Development tools for high-level languages are more expensive than those for assembly.

Mixing Assembly and High-Level Language

In embedded development, it is often necessary to **combine Assembly Language and High-Level Languages (like C)**. This is especially important when performance, precision timing, or hardware-specific features are required.

Mixing Assembly with C

Used when:

- You write most of your code in C but need **critical functions in Assembly** for speed or direct hardware control.
- **Cross-compilers lack support** for some low-level tasks (e.g., ISRs).

Steps (Keil C51 Example):

1. Write a C function with parameters and return values.
2. Add #pragma SRC to the top of your C file.
3. Compile it → Generates a .SRC file (not .OBJ).
4. Rename .SRC to .A51.
5. Insert your custom Assembly code into the shell function.

Mixing C with Assembly

Used when:

- You have an **existing Assembly program** and want to call some routines written in C.
- Certain operations (e.g., 16-bit multiplication) are easier in C than in Assembly.
- You need to use **built-in libraries** (e.g., graphics, string operations).

Parameter Passing (Keil C51):

- C functions pass arguments via **registers** or **fixed memory**:
  - char → R7, R6, R5
  - int → (R7,R6), (R5,R4), (R3,R2)
- If >3 arguments: extra ones go to fixed memory.
- Return Values:
  - char → R7
  - int → R7 and R6

Calling C from Assembly:

LCALL _Cfunction

_ prefix = parameters passed through registers.
Without _ = parameters passed via memory.

Inline Assembly in C

This embeds assembly **directly into C code** using special directives.

Syntax (Keil C51):

```
#pragma asm
    MOV A, #13H
#pragma endasm
```

Used for:

- Avoiding call overhead
- Executing quick, time-sensitive instructions inline

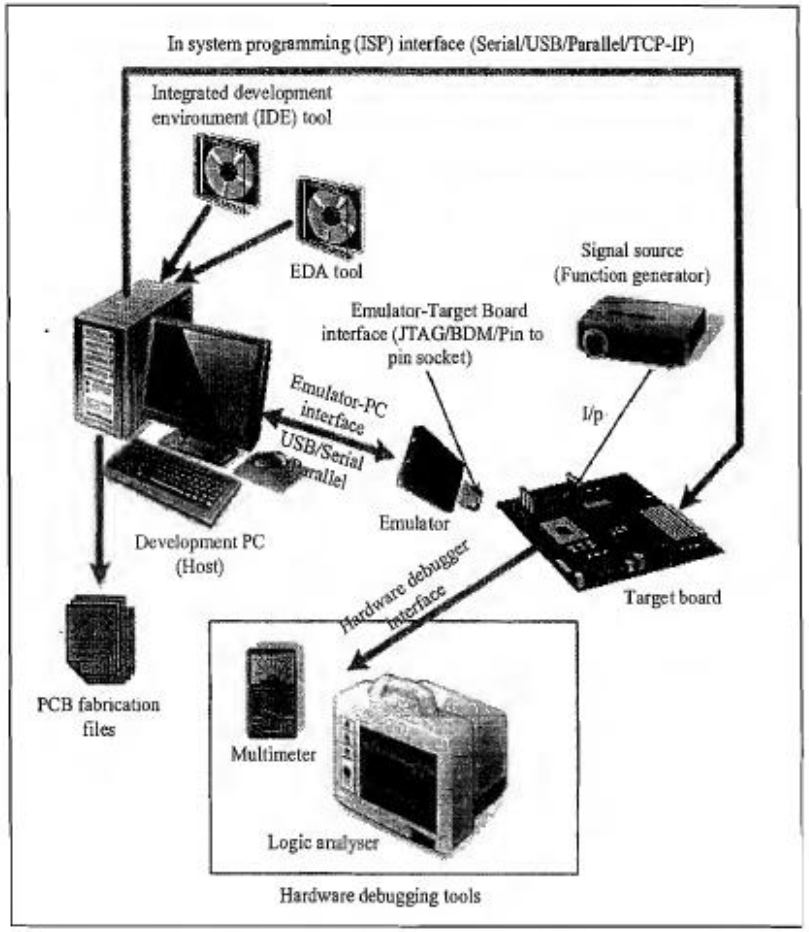
Firmware Programming Languages

There are several Firmware programming languages used in embedded firmware development, each with its own strengths and weaknesses. The most popular languages used in embedded firmware development are **C, C++, Assembly, and Rust.**

Embedded System Development Environments: Types of files generated on cross compilation

A typical setup includes:

- A **Host PC (Development Computer)** as the core,
- **IDE tools** for firmware development and debugging,
- **EDA tools** for hardware design,
- **Emulator hardware** for debugging,
- **Signal sources** (e.g., function generators) to simulate inputs,
- **Debugging tools** like digital CROs, multimeters, and logic analyzers,
- And the **target hardware**.



IDE and EDA tools are chosen based on project needs and come as installable packages—freeware, licensed, or evaluation versions. Licensed tools offer full features, while trial versions may be limited in function or usage duration.

Keil µVision3 - 8051 Project Setup (Quick Steps)

1.Open Keil µVision3

2.Create New Project

- Project > New Project... → Name and save it

3.Select Microcontroller

- Example: **Atmel > AT89C51**

4.Create Source File

- File > New → Write code → Save as .c file

5.Add File to Project

- Right-click **Source Group 1** > Add Files > Select .c file

6.Build Project

- Project > Build Target (or press **F7**)

7.Save All Files

- File > Save All

Cross Compilation vs Cross Assembling

•**Cross Compilation**:
Converts **high-level language (e.g., Embedded C)** to **target machine code** using a cross-compiler on a different processor (e.g., x86 PC to 8051).

•**Cross Assembling**:
Converts **Assembly code** to **machine code** using a **cross-assembler**.

Files Generated During Cross Compilation/Assembling

File Type	Extension	Purpose
List File	.lst	Detailed log of compilation: source code + assembly + errors + symbols
Preprocessor Output	Varies	Shows output after handling macros, #include, #if etc.
Object File	.obj	Intermediate code (not executable); input for the linker
Map File	.map	Linker output showing memory allocation and linking details
Hex File	.hex	Final machine code; uploaded to microcontroller (e.g., Intel HEX format)

Key Points about Each File

- **List File (.LST)**

•Shows **source + generated assembly + symbol table**
•Helps in **debugging**

- **Preprocessor Output File**

•Contains code after processing directives like #define, #ifdef
•Useful for **macro debugging**

- **Object File (.OBJ)**

•Contains re-locatable code, symbol tables, references
•Needs **linking** before execution

- **Map File (.MAP)**

•Shows **memory layout**, input modules, and **final address allocation**
•Helps in understanding code placement in memory

- **Hex File (.HEX)**

•Final **machine-readable** file
•Upload to microcontroller
•Common formats: **Intel HEX**, **Motorola HEX**

Disassembler/Decompiler

A **disassembler** converts machine code into processor-specific **assembly code**, while a **decompiler** translates it into a **high-level language**.

These tools perform the reverse of compilers and assemblers, aiding **reverse engineering** in embedded product development to understand proprietary technologies or detect malicious code.

Though they can't recreate exact source code (e.g., comments or symbolic names), they provide a close approximation. Disassemblers and decompilers are available as both **freeware** and **commercial tools**.

Examples of Tools:

- **Disassemblers**: IDA Pro, Ghidra (also has decompiler), objdump.
- **Decompilers**: Ghidra, RetDec, Hex-Rays Decompiler.

Aspect	Disassembler	Decompiler
Function	Converts machine code to **assembly code**	Converts machine code to **high-level source code**
Output Language	Low-level, processor-specific instructions	High-level language (e.g., C/C++)
Use Case	Analyzing system calls, instruction-level debugging	Understanding program logic, reverse engineering
Application	Embedded firmware analysis, malware detection	Code recovery, software audits
Limitations	No comments, variable names, or functions retained	Output may be imprecise, logic may be misinterpreted

Simulators in Embedded Systems

Simulators are software tools that mimic target hardware for firmware debugging, without needing physical devices.

Advantages:

- **No hardware required** – useful for early development.
- **Simulate I/O** – allows testing input/output behavior.
- **Test abnormal conditions** – helps study firmware response to unexpected inputs

Key Features:

- **Hardware Emulation:** Simulate microcontrollers, sensors, memory, and peripherals.
- **Code Debugging:** Step-through execution, breakpoints, and memory inspection.
- **Cost-Effective:** No need for physical prototypes during early development.
- **Time-Saving:** Speeds up the development and testing cycle.
- **Safe Testing:** Ideal for testing edge cases or potentially destructive operations.

Examples of Popular Simulators:

- **Proteus** – Simulates microcontrollers and entire circuits.
- **Keil uVision Simulator** – For ARM-based microcontroller development.
- **AVR Simulator IDE** – For Atmel AVR series microcontrollers.
- **QEMU** – A generic and open-source machine emulator and virtualizer.

Limitations:

- **Not real-time** – lacks timing accuracy of real hardware.
- **Deviation from actual behavior** – results may differ from real-world execution.

1. Emulators

Definition:
An **emulator** is a software or hardware that mimics another system's behavior, allowing one system to run software designed for another. It creates a virtual environment to test and run applications without the actual target hardware.

Applications:
- **Mobile App Development**: Android Emulator, iOS Simulator
- **Embedded Systems**: AVR emulator, Proteus, Keil µVision
- **Game Consoles**: PCSX2 (PlayStation), Dolphin (GameCube/Wii)

Advantages:

- No need for physical hardware
- Faster iteration and testing
- Helps test across multiple environments/platforms

2. Debugging

Definition:

Debugging is the process of finding and fixing bugs (errors) in software or hardware design.

Tools Used:

- **Software Debuggers**: GDB (GNU Debugger), Visual Studio Debugger, LLDB
- **Embedded System Debuggers**: JTAG, ST-Link, Atmel-ICE
- **IDEs**: Arduino IDE, Keil, MPLAB X with built-in debugging tools

Key Debugging Techniques:

- **Breakpoints**: Pause code execution at a specific line
- **Watch Variables**: Monitor values of variables in real-time
- **Step Execution**: Step into/over functions line by line
- **Logging**: Use print statements or serial monitors

Task	Emulator Role	Debugger Role
Run software without hardware	Creates a virtual device	Not applicable
Test hardware-specific code	Simulates hardware peripherals	Step through code to verify logic
Catch and fix errors	May simulate faulty scenarios	Helps pinpoint bugs precisely

ATME College of Engineering

Department of EEE
Emitting Elite Energy

Thank You