NYABIHU DISTRICT

BIGOGWE TSS

EMBEDDED SYSTEM FIRMWARE DEVELOPMENT

CLASS:LEVEL CSA

2025-2026

PREPARED BY NIBISHAKA ERIC /TRAINER IN CSA
Phone :0783434006/nibishakae@gmail.com

EMBEDDED SYSTEM FIRMWARE DEVELOPMENT

Learning Outcome 1: Identify Firmware Requirement Learning Outcome

2: Design Firmware Architecture Learning Outcome

3: Implement Firmware System Design Learning Outcome

4: Deploy Firmware

Learning Outcome 1: Identify Firmware Requirement

1.1. Collection of Data for Firmware Development

1.2. Analysing Collected Data

1.3. Extraction Of Firmware Requirements

1.1. Collection of Data for Firmware Development

1.1 .1 Firmware is a special type of software that is **permanently programmed** into a hardware device to control how it works.

Firmware is stored on ROM(For old embedded) or flash memory for New embedded hardware

1.1.2 Different between firmware and software

Aspect	Firmware	Software
Definition	A special type of software embedded into hardware during manufacturing.	General term for programs and applications that run on computers and devices.
Storage	Permanently stored in hardware (e.g., Flash/ROM chips).	Stored on drives (HDD, SSD, optical disks) and loaded into RAM when used.
Purpose	Controls hardware functions and ensures devices operate correctly.	Provides specific functionalities for users (e.g., OS, apps, games).
Flexibility	More stable, hardware-specific, not updated often.	Easily updated, modified, or replaced by users/developers.
Examples	BIOS/UEFI, printer firmware, washing machine controller.	Operating systems, word processors, web browsers, games.

1.1.3 Types of Firmware Data

1.BIOS (Basic Input/Output System)

-Essential for computer booting.

-Initializes hardware and provides interface between OS and hardware.

2.UEFI (Unified Extensible Firmware Interface)

-Modern replacement for BIOS.

-Supports larger drives, faster boot, and has a graphical interface.

3.Bootloader

-Small program that loads the operating system into memory during boot.

-Stored in firmware and initiates the OS.

4.Device Firmware

-Firmware for specific hardware like hard drives, graphics cards, printers.

-Provides low-level control and ensures proper hardware-OS communication.

5.Embedded Firmware

-Found in microcontrollers of appliances, cars, industrial machines.

-Tailored specifically for the device's function.

6.Peripheral Firmware

-For peripherals like keyboards, mice, monitors.

-Enables advanced features and customization (e.g., gaming mouse settings).

7.Network Device Firmware

-Found in routers, modems, and network devices.

-Controls networking, data transmission, security, and configurations.

8.System Management Firmware

-Used in servers and enterprise systems.

-Allows remote management, diagnostics, and error logging.

8.Security Firmware

-Focused on cybersecurity.

-Handles encryption, secure boot, and protection against unauthorized access.

9.Software-defined Firmware

-Can be updated or modified via software updates.

-Allows bug fixes, performance improvements, and new features without replacing hardware.

1.1.4 Types of Firmware

-Low-Level Firmware:

-Built-in part of the hardware.

-Stored on ROM or read-only chips; cannot be rewritten or updated.

-High-Level Firmware:More complex and can be updated.

-Stored on flash memory chips in computers.

-Subsystem Firmware:Part of embedded systems; operates like high-level firmware.

Difference in table list

Type of Firmware	Definition / Key Features	Storage / Update	Example / Use
Low-Level Firmware	Built-in, essential part of hardware. Controls very basic functions of the device.	Stored on ROM or read-only chips; **cannot be updated**.	BIOS in old computers, microcontroller boot ROM.
High-Level Firmware	More complex, provides advanced control. Interfaces more closely with the operating system or user applications.	Stored on flash memory; **can be updated**.	Modern BIOS/UEFI, router firmware, SSD controller firmware.
Subsystem Firmware	Firmware for specific parts (subsystems) of a device. Works like high-level firmware but controls individual components.	Stored on flash or embedded memory; **can be updated**.	CPU microcode, LCD controller firmware, network card firmware.

1.1.5 Key Components of Firmware Architecture

-**Bootloader**:Starts the device when you turn it on

-**Operating system** : Core brain of the system that manages all resources

- **Device Drivers**:Translates between software and hardware parts to Allows the system to "talk" to different components.

-**Hardware Abstraction Layer (HAL)**:Creates a standard way for software to access hardware

-**APIs (Application Programming Interfaces)**:Rules that let different software parts communicate

1.1.6 Applications of Firmware

1**.Consumer Electronics**

-Smartphones: Controls camera, sensors, communication

-TVs: Manages display, audio, smart features

2.Computing Devices

-Personal Computers: BIOS/UEFI starts up the computer

-Printers/Scanners: Controls printing/scanning operations

3.Embedded Systems

-Cars: Engine control units manage engine performance

-Medical Devices: Controls pumps, monitors, imaging equipment

-Industrial Systems: Manages factory automation equipment

4.Networking Equipment

-Routers/Switches: Manages internet traffic and security

-Network Cards: Controls computer-to-network communication

5.Home Appliances

-Refrigerators/Washing Machines: Controls operations and sensors

-Smart Thermostats: Manages temperature and energy efficiency

6.Wearable Devices

-Fitness Trackers/Smartwatches: Monitors health, connects to phones

7.Gaming Systems

-Game Consoles: Controls hardware and manages updates

8.Communication Devices

-VoIP Phones: Handles calls and audio processing

-Modems: Manages internet data transmission

9.Aerospace and Defense

-Aircraft Systems: Navigation, communication, flight control

-Military Equipment: Radar, communication systems

10.Internet of Things (IoT)

-Smart Home: Light bulbs, security cameras, door locks

-Connected Sensors: Data collection and wireless communication

1.1.7.Firmware Development Process

Step 1: Define System Specifications:Write down exactly what the system needs to accomplish

Step 2: Develop Hardware:Choose or design the physical components (chips, sensors, etc.)

Step 3: Develop Software/Firmware: Write the actual code using development tools

Step 4: Test and Validate: Test thoroughly on real hardware, fix bugs, finalize**.**

Step 5: Firmware deployment: After the firmware has been tested and verified, it is ready to be deployed on the target device.

Step 6:Maintenance and updates: Even after the firmware has been deployed, it is important to maintain and update it over time as needed.

2.Data collection methods: Data collection methods are the techniques or processes used to gather information or data for analysis or research purposes.

2.1: Categories of data collection methods

1.Primary Data Collection Methods: Information collected directly from original sources for a specific research purpose.

Ex: -Surveys and Questionnaires

Method: Asking structured questions to individuals or groups.

-**Interviews:** Collect qualitative data through direct conversations

-**Observation**: Directly watching and recording behaviors or events

-**Experiments**:Test hypotheses under specific, controlled conditions

2.Secondary Data Collection Methods: Using existing information that was previously collected by others.

Sources: Books, journals, databases, records, reports.

Indicative content 1.2: Analysing collected data

1.2.1 Firmware & Embedded System Requirements

1.2.1.1 Firmware requirement

1. 2.1.1.1 Hardware Requirements

1.Processing Power :How fast the hardware can execute instructions.

2.Power Design :Choose efficient, stable, and reliable power systems:

Here you consider the following:

-Correct power supply & voltage regulation.

-Energy efficiency (important for battery devices).

-Dynamic power management (use power only when needed).

-Battery management (if battery-powered).

-Cooling (fans/heat sinks).

-Use low-power components.

-Monitor power usage.

3.**Communication & Interfaces:**How the system talks to sensors, peripherals, users.

-Sensor/Data Interfaces: I2C, SPI, UART, ADC.

-Networking**:** TCP/IP, Wi-Fi, Bluetooth, Zigbee, LoRa.

-Serial/USB: Data transfer to host systems.

-Inter-Processor Communication**:** Shared memory, message passing.

-Storage Interfaces**:** SD cards, Flash chips.

-HMI: Buttons, touchscreens, displays.

-Industrial Protocols**:** CAN, Modbus, Profibus.

-Security Interfaces**:** TLS/SSL, encryption.

-Custom Interfaces: Device-specific protocols.

4.Storage Capacity

What to consider:

-Compression

-Backups.

-Scalability (future growth).

-Hardware storage limits.

Types of Storage used to store firmware :

-ROM → Permanent (bootloader, startup code).

-Flash → Firmware, configs (rewritable, non-volatile).

-EEPROM → Small data (settings, calibration).

-SRAM → Fast temporary memory (variables, cache).

-DRAM → Larger temporary memory (main memory).

5. Environmental Considerations

What to consider :

-Temperature & humidity tolerance.

-EMI/EMC (electrical noise resistance).

-Vibration/shock resistance.

-Dust/water resistance (IP rating).

-Chemical & radiation exposure.

-Physical size & weight limits.

-RF interference (for wireless systems).

1.2.1.2.Embedded System Requirements

a) Functional requirements: Clearly define what the system must do.

- **Data Acquisition** – Ability to read data from sensors (temperature, pressure, motion, etc.).

- **Control Operations** – Execution of control algorithms (e.g., motor control, power regulation).

- **Input/Output Handling** – Manage analog and digital I/O (e.g., LEDs, buttons, displays).

- **Communication** : Support protocols like UART, SPI, I2C, CAN, Bluetooth, Wi-Fi, or Ethernet.

- **User Interface** : Provide interaction through buttons, touchscreens, LEDs, or displays.

- **Data Storage** :Save data locally (EEPROM, Flash, SD card) or send to external storage/cloud.

- **System Initialization** :Boot and initialize hardware properly (firmware startup routines).

- **Error Detection & Handling** : Detect faults, handle errors, and trigger recovery processes.

- **Real-Time Operation** – Respond to events within strict timing constraints (e.g., automotive ABS system).

- **Security Functions** – Provide secure boot, authentication, and data encryption (if required).

-Use case studies -Choose the case study where to collect data for your project .Such as Temperature monitoring, Smart lighting control, smart irrigation control, Motion detection and surveillance , Alcohol detection for route traffic ,House security for smoke detection ,Water flow monitoring ….

b)Non functional requirements: How it should perform (speed, reliability, scalability, security, usability).

These describe **quality attributes and constraints** of the system:

-Performance – Define speed, latency, and throughput of operations.

-Reliability & Stability – System must run consistently without crashes or unexpected behavior.

-Power Efficiency – Low power consumption, especially for battery-powered systems.

-Scalability – Ability to handle more features, sensors, or users in the future.

-Maintainability – Easy to update firmware and fix bugs.

-Security – Protect against unauthorized access, data breaches, or firmware tampering.

-Usability – Provide intuitive interaction for end-users.

-Portability – System should work across different environments or platforms with minimal changes.

-Environmental Tolerance – Operate under conditions like extreme temperature, humidity, vibration, or dust.

-Compliance – Must follow industry standards, certifications, and safety regulations such as ISO 26262 and IEC6730.

-Memory constraint

-Response time

1.2.1.3 The role of firmware engineer /developer and end user

Here's how end-users and engineers play different but complementary roles:

End-Users: •

Roles:

- Describe desired tasks and features.

- Report bugs and performance issues.

-Provide feedback on ease of use and intuitiveness.

Engineers:

Roles:

-Translate user needs into technical specifications.

-Identify software limitations and dependencies.

- Consider security and safety implications.

Similarities:

-Both want a well-functioning device.

- Both provide valuable insights during the requirements gathering process.

1.2.1.4 System feasibility for an embedded system

Examination of different aspects that can affect the development of an embedded systems

1.Technical Feasibility:

Evaluates the technical aspects of implementing the firmware, considering the available technology, expertise, and compatibility with existing systems.

2.Economical Feasibility

This examines the financial aspects of the firmware development project, assessing the costs and potential benefits.

3.Legal
This addresses compliance with legal requirements, standards, and regulations relevant to the firmware and its application.

4.Operational Feasibility
Here assesses the practicality and effectiveness of implementing the firmware within the operational environment.

5.Scheduling Feasibility
This examines the time constraints and deadlines associated with the firmware development project.

Lo 2: Firmware Architecture Design

Firmware development architecture refers to the overall structure and organization of software that runs on embedded systems.

2.1 Selection of Tools, materials, and equipment

2.1.1. Development Tools

2.1.1.1 Software Tools

IDEs (Integrated Development Environments)

Purpose: Help programmers write code,debug code ,and compile code efficiently

Examples: Eclipse, Keil uVision, Proteus,Arduino IDE

2.1.1.2.Version Control Systems

Purpose: Manage source code changes over time.

Examples: Git, SVN (Subversion)

2.1.1.3 Modeling and Design Tools

Purpose: Create visual representations of firmware structure

Examples:

-**Microsoft Visio** :for flowcharts and diagrams

-Lucidchart: Online collaborative diagramming tool for system diagrams and firmware workflows.

-**Draw.io (diagrams.net)**:Free tool for creating flowcharts, state diagrams, and architecture diagrams.

-**PlantUML**:Text-based tool for UML diagrams (class, sequence, state machine, etc.).

- **Dia**:Open-source tool for creating flowcharts, network diagrams, and ER diagrams.

2.1.1.4 Debugging Tools

Purpose: Find and fix code issues

Examples:

-JTAG Debuggers (Segger J-Link, ST-Link)

-Logic Analyzers and Oscilloscopes

2.1.1.5 Simulation Tools

Purpose: Test firmware without physical hardware

Examples: QEMU, Proteus

2.1.1.6 RTOS (Real-Time Operating System)

Purpose: Manage resources with predictable timing

Examples: FreeRTOS

2.1.1.7 Documentation Tools

Purpose: Generate and manage project documentation

Examples: Doxygen, Markdown Editors

2.1.1.8 Code Analysis Tools

Purpose: Analyze code quality and find issues

Examples: Coverity, CodeSonar, PC-lint, Unity testing framework

Hardware Description Languages (HDL)

Purpose: Describe digital circuit structure and behavior

Examples: VHDL, Verilog

2.1.1.9 Collaboration Tools

Purpose: Team communication and project management

Examples:

Communication: Slack, Microsoft Teams, Discord

Project Management: JIRA, Trello

Code Review: Gerrit

2.1.2 Materials Required

Materials used to build an embedded hardware .

-Breadboard and Jumper Wires: Circuit prototyping

-Sensors and Actuators: Physical world interaction (temperature sensors, motors, LEDs)

-Communication Modules: Device connectivity (Wi-Fi, Bluetooth, GSM modules)

-Microcontroller/Processor: Central processing unit (Atmel AVR, ARM Cortex-M, PIC)

-Source Code: Written in C, C++, or Assembly

-Documentation: Specifications and user manuals

2.1.3 Equipment Needed

-Computer: Development workstation (laptop/desktop)

-Power Supply: Electrical power for circuits (bench supply, battery pack)

-Oscilloscope: Signal measurement and analysis

-Logic Analyzer: Digital signal capture and analysis

-Multimeter: Voltage, current, resistance measurement

-Development Boards: Integrated platforms (Arduino, Raspberry Pi)

Different boards used to build embedded hardware

Board	Main Feature	Best For
Arduino Uno	Very simple, lots of tutorials, many add-on shields	Absolute beginners, small sensor projects
Arduino Mega	Same as Uno but with more pins and memory	Robotics, projects needing many connections
Raspberry Pi	A small computer (runs Linux, has HDMI/USB/Wi-Fi)	Learning programming, IoT, multimedia
ESP32 / ESP8266	Built-in Wi-Fi + Bluetooth, very cheap	IoT, wireless devices, smart home
STM32 Nucleo	Powerful ARM Cortex-M, real-time performance	Advanced embedded control, robotics
BeagleBone Black	Linux board with many I/Os	Industrial systems, robotics
TI LaunchPad	Very low power, supported by TI tools	Low-power applications, RTOS learning
BBC micro:bit	Easy for kids, has built-in LEDs & sensors	Education, beginner coding

PIC Boards	Use Microchip PIC microcontrollers	Industrial learning, microcontroller fundamentals

2.2 Preparation of drawing environment

2.2.1 Installation of a drawing Tool

In **firmware development**, a **drawing tool** is a **software application used to create diagrams, models, and visual representations** of the firmware system

2.2.1.1. Installation Process

Choose Tool: Select based on requirements

Check System Requirements: Verify compatibility

Download Software: From official website

Run Installer: Follow installation wizard

Customize Installation: Select needed features

Accept License: Agree to terms

Choose Location: Specify install directory

Complete Installation: Finish setup process

Activate Tool: Enter license key if required

Check Updates: Install latest versions

Explore Documentation: Learn tool features

2.2.1.2 Popular Drawing Tools

Tool	Advantages	Disadvantages
Lucidchart	-Cloud-based collaborative, -Easy to use - Many templates	-Paid for full features -Relies on internet, -Limited offline functionality
PlantUML	-Text-based UML diagrams -Easy version control, -Integrates with IDEs	-Requires learning syntax -Not visually intuitive, -Limited diagram styling
Draw.io (diagrams.net)	-Free -Simple to use , -Online and offline desktop versions	-Limited advanced features, -Interface can be basic for Complex diagrams

	-Integrates with cloud storage	
KiCad	-Free -Open-source, -Good for PCB design, -Cross-platform	-Steeper learning curve -Fewer advanced features, -Manual library setup
Altium Designer	-Professional-grade, - Powerful features, -Extensive libraries	-Very expensive, -High system requirements, Complex interface for beginners
OmniGraffle	Easy-to-use, visually appealing diagrams, good for macOS/iOS	Apple-only, paid software, limited collaboration features
Dia	Free and open-source, Lightweight	-Outdated interface, -Fewer templates and features, -Limited support/updates
Microsoft Visio	-Rich diagramming features, -Integrates with MS Office	-Paid software, -Can be complex, -Windows-only for full features

2.2.1.3 Installation of each drawing tool

Tool	Installation Steps
Lucidchart	1. Go to lucidchart.com 2. Sign up using email or Google 3. Start creating diagrams in browser (optional desktop app)
PlantUML	1. Download PlantUML jar from plantuml.com 2. Install Java Runtime Environment (JRE) 3. Run via command line or integrate into IDE
Draw.io (diagrams.net)	1. Go to diagrams.net 2. Choose storage option (Google Drive, local, etc.) 3. Start creating diagrams (optional desktop version download)
KiCad	1. Go to kicad.org/download 2. Select OS and download installer 3. Run installer and follow prompts 4. Launch KiCad
Altium Designer	1. Visit altium.com 2. Download trial or installer 3. Run installer and follow prompts 4. Activate license and launch
OmniGraffle	1. Go to omnigroup.com/omnigraffle 2. Download trial or App Store version 3. Run installer and launch tool

Dia	1. Go to sourceforge.net/projects/dia-installer 2. Download installer for your OS 3. Run installer and launch Dia

✓ 2.3 Identification of Symbols and Notation

1.Flowcharts:

Process: Represented by a rectangle with rounded corners.

Decision: Shown as a diamond shape, indicating a decision point in the flow.

Start/End: Represented by an oval shape, indicating the beginning or end of a process.

Input/Output: Shown as a parallelogram, indicating input or output operations.

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectagle represents a process
	Decision	A diamond indicates a decision

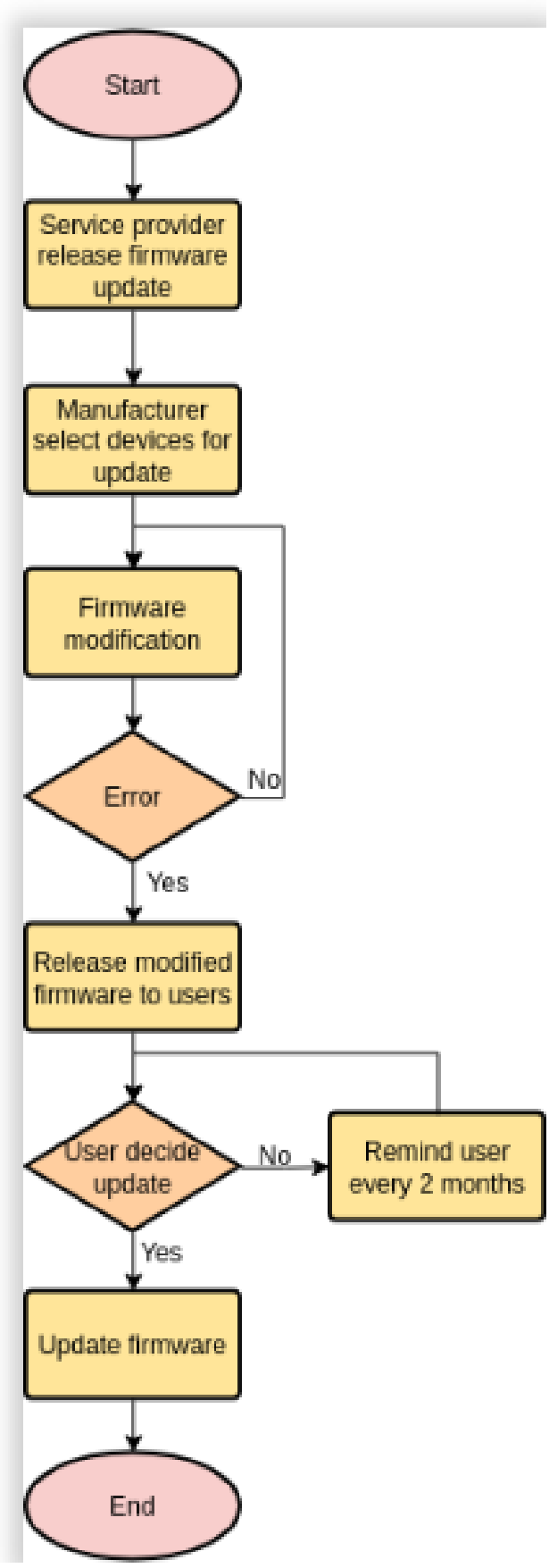

Start
Service provider release firmware update
Manufacturer select devices for update
Firmware modification
Error
No
Yes
Release modified firmware to users
User decide update
No
Remind user every 2 months
Yes
Update firmware
End

Diagram	Purpose / Use	Symbols / Annotations
Flowchart	Shows program logic, step-by-step process	Ovals (Start/End), Rectangles (Process), Diamonds (Decision), Arrows (Flow)
Block Diagram/Hardware block diagram	Represents system components and data flow	Rectangles (modules/components), Arrows (data/control flow), Circles (inputs/outputs)
UML Class Diagram	Shows object-oriented design, classes, relationships	Rectangles (classes), Lines (associations), Arrows (inheritance), Attributes & Methods inside rectangles
UML Sequence Diagram	Shows interaction between objects over time	Lifelines (dashed lines), Rectangles (activation), Arrows (messages), Notes for explanation
State Machine / State Diagram	Represents system states and transitions	Circles/Ovals (states), Arrows (transitions), Labels on arrows (events/conditions)
Entity-Relationship Diagram (ERD)	Shows data storage and relationships	Rectangles (entities), Diamonds (relationships), Lines connecting entities, Attributes inside entities
Timing Diagram	Shows signal changes over time in hardware/software	Horizontal lines (signals), Vertical markers (time events), Labels for signal state (high/low)
Circuit / Schematic Diagram	Represents hardware connections for firmware	Resistors, capacitors, ICs, wires, power supply, LEDs (standard electrical symbols)
Data Flow Diagram (DFD)	Shows flow of information within system	Circles or ovals (process), Arrows (data flow), Rectangles (external entities), Open-ended rectangles (data stores)
Use Case Diagram (UML)	Represents user interactions with system	Ovals (use cases), Stick figures (actors), Lines connecting actors to use cases, Notes for details
Activity Diagram (UML)	Models workflow or operations	Rounded rectangles (activities), Diamonds (decision), Arrows (control flow), Bars (start/end of parallel activities)

2.4 Gathering Complete Hardware Information before developing a firmware .

collecting all the technical details about the target hardware platform where the firmware will run.

Hardware components for embedded hardware

-Microcontroller

-Microprocessor

-Memory

-I/O

Here are the steps typically followed to gather microcontroller information:

1. **Identify the Microcontroller**
   Find the exact model/part number of the MCU being used.
2. **Review Datasheets**
   Get technical details: pinout, electrical specs, memory maps, and peripherals.
3. **Check Reference Manuals**
   Study in-depth functional and programming details of the MCU.
4. **Use Technical Guides & Application Notes**
   Learn best practices and manufacturer-recommended implementations.
5. **Explore Online Resources**
   Visit the manufacturer's site, libraries, and support forums.
6. **Consult Development Tool Documentation**
   Understand IDE setup, toolchains, compilers, and programmer/debugger use.
7. **List Peripheral Modules**
   Identify timers, UART, SPI, I²C, GPIO, ADC/DAC, etc., and learn their configs.
8. **Understand Memory Architecture**
   Know the types, sizes, and addressing of Flash, RAM, and EEPROM.
9. **Review Clocking & Power**
   Study clock sources, frequency ranges, and power management options.
10. **Programming & Debug Interfaces**
    Learn how to connect via JTAG, SWD, or other debug/program ports.
11. **Check Compliance/Certifications**
    Identify standards (e.g., safety, EMC) relevant to the MCU/project.

Certainly! Here's a table outlining some key differences between a microcontroller (MCU) and a microprocessor (MPU):

Aspect	**Microcontroller (MCU)**	**Microprocessor (MPU)**
Integrated Components	Typically contains CPU, memory, I/O	CPU only, requires external components such as RAM

Aspect	Microcontroller (MCU)	Microprocessor (MPU)
Purpose	Used in embedded systems and devices	Found in general-purpose computing devices
Complexity	Generally less complex	More complex
Cost	Often lower cost	May be higher cost depending on components
Power Consumption	Lower power consumption	Higher power consumption in some cases
Performance	Lower performance capabilities	Higher performance capabilities
I/O Integration	Integrated I/O peripherals	External peripherals required
Footprint(size of cpu occupy on the board).**Size**	Compact, smaller footprint	Larger footprint due to external components

Gathering for most hardware components used in embedded hardware for firmware development

Hardware Component	Steps to Gather Information
Microcontroller (MCU)	1. Identify part number/model 2. Review datasheet & reference manual 3. Check CPU architecture, clock, instruction set 4. Review memory (Flash, RAM, EEPROM) sizes & map 5. Study peripherals (UART, SPI, I²C, timers, ADC, GPIO) 6. Check power specs (voltage, consumption) 7. Confirm programming/debug interfaces (JTAG, SWD, ISP)
Memory (External)	1. Identify type (Flash, EEPROM, SRAM, SD) 2. Check capacity & speed 3. Review interface (SPI, I²C, parallel) 4. Study addressing & timings

	5. Note endurance (write cycles) & retention
Sensors (Temp, Motion, etc.)	1. Identify sensor model/type 2. Review datasheet: range, accuracy, resolution 3. Check communication interface (Analog, I²C, SPI, UART) 4. Review sampling rate & response time 5. Check calibration needs 6. Review power & environmental conditions
Actuators (Motors, Relays, LEDs, etc.)	1. Identify type & electrical specs (V/I) 2. Check control method (PWM, GPIO, driver IC) 3. Review response time & load capacity 4. Check protection requirements (e.g., flyback diodes) 5. Review lifetime & reliability
Power Supply / Battery	1. Identify voltage requirements for all components 2. Check power source type (battery, adapter, USB, solar) 3. Review current capacity & efficiency 4. Study regulators (LDO, DC-DC) 5. Consider backup/low-power modes
Communication Modules (Wi-Fi, BT, ZigBee, etc.)	1. Identify module model 2. Review interface (UART, SPI, USB) 3. Check supported protocols & data rates 4. Study voltage/current requirements 5. Review range & antenna needs 6. Check driver/library support
Display / User Interface	1. Identify display model & controller IC 2. Review resolution, color depth, refresh rate 3. Check interface protocol (SPI, I²C, parallel) 4. Verify driver/library availability 5. Study power & backlight requirements
Connectors & I/O Interfaces	1. Identify connector types (USB, RS232, HDMI, custom) 2. Check pinout & voltage levels 3. Review supported standards 4. Study ESD protection & shielding needs

2.5 .Drawing templates for firmware development

A drawing template is a **standardized diagram or schematic format** used to represent different parts of the firmware system and its relationship with hardware, software layers, and external components.

2.5.1 :Roles of drawing templates

- **Standardize Documentation** – Provides a consistent format for diagrams across the project.

- **Visualize Architecture** – Shows the structure of firmware, hardware, and their interactions.

- **Clarify Module Interactions** – Helps understand how different firmware modules communicate.

- **Guide Implementation** – Serves as a reference for coding, debugging, and testing.

- **Facilitate Communication** – Makes it easier for developers, testers, and stakeholders to understand the system.

- **Support Maintenance** – Helps future updates or modifications by documenting design decisions.

2.5.2 Steps to Create Drawing Templates for Hardware Components

-Identify Components – List key hardware components (MCUs, sensors, actuators, interfaces).

-Select Drawing Tool – Choose suitable software (e.g., KiCad, Altium, Inkscape).

-Create Template – Design a layout with placeholders for components.

-Define Symbols – Assign clear, standardized symbols for each component.

-Labeling & Documentation – Add names, part numbers, specifications, and pin labels.

-Connection Lines & Wiring – Represent signal, power, and control lines; indicate flow direction.

-Grouping & Organization – Logically group related components or subsystems.

-Annotations & Notes – Include explanations for functionality, constraints, or requirements.

-Create Multiple Views – Use layers or views for control, power, or subsystem perspectives.

-Review & Iterate – Validate accuracy with team/stakeholders and refine.

-Save as Templates – Store as reusable templates for future projects.

2.5.3 Applying Color Scheme and Styling to designed drawing step

Steps to Apply Color Scheme and Styling on Drawings

-Open the Project / Drawing – Launch your drawing tool and open the diagram or schematic.

-Access the Editor – Navigate to the editor where components and connections are displayed.

-Set Background Color – Adjust the workspace or canvas background for better visibility.

-Apply Component Colors – Assign distinct colors to different components or modules for clarity.

-Customize Component Symbols – Modify shapes, colors, and styles of symbols to make them easily recognizable.

-Style Connections – Use different colors or line styles to differentiate signal types (data, control, power).

-Consistent Styling – Maintain consistent fonts, colors, and line widths throughout the drawing.

-Review and Adjust – Check the visual clarity of the diagram and adjust colors or styles if needed.

2.6 Drawing Firmware architecture diagrams

2.6.1 Definition

2.6.2:Steps for firmware architecture diagrams

-**Identify Key Components:** List main firmware modules and functions.

- **Define Relationships:** Determine how components interact and communicate.

- **Choose a Tool:** Select a suitable drawing tool (e.g., Visio, Draw.io, Lucidchart).

- **Create a New Diagram:** Start a new architecture diagram layout.

- **Represent Components:** Use shapes to depict modules and external systems.

- **Label Components:** Clearly name and describe each component.

- **Draw Connections:** Add arrows or lines to show data flow and communication.

- **Group Components:** Organize related modules into subsystems or layers.

- **Highlight Interfaces:** Emphasize key interfaces (UART, SPI, I2C, etc.).

- **Indicate Data Flow:** Show direction and paths of critical data.

- **Include External Dependencies:** Add sensors or external hardware connections.

- **Add Annotations:** Provide short notes for context or design rationale.

- **Review and Iterate:** Refine the diagram with team feedback.

- **Document Constraints:** Note assumptions or limitations affecting design.

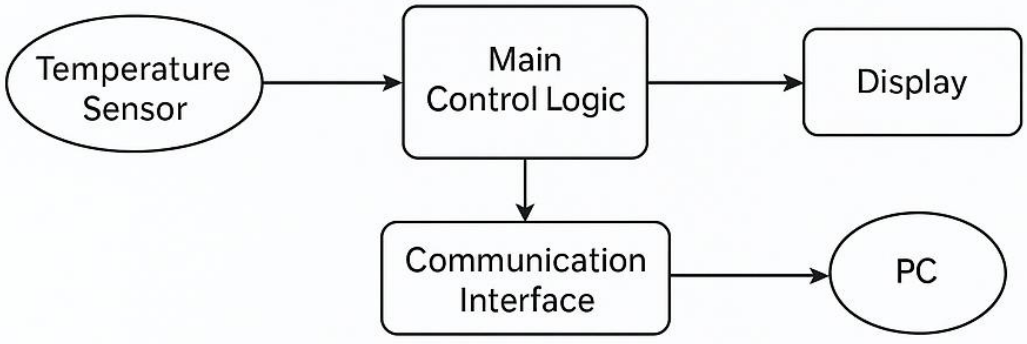
- **Save and Share:** Export the final diagram for distribution or documentation

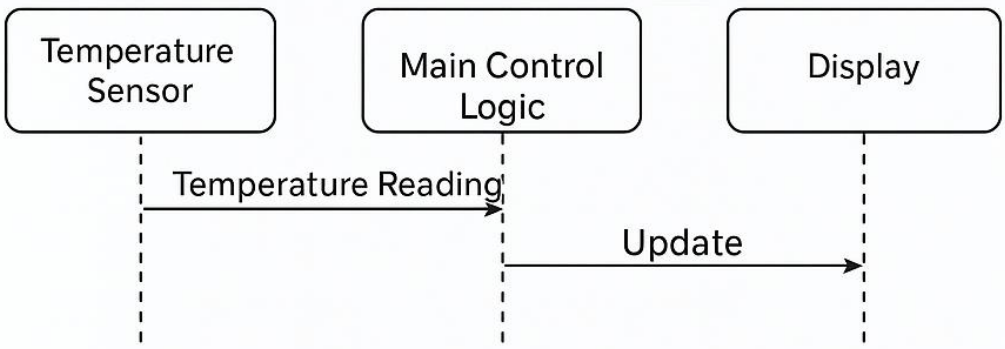
Example for temperature monitoring

STEPS TO DRAW FIRMWARE ARCHITECTURE DIAGRAMS

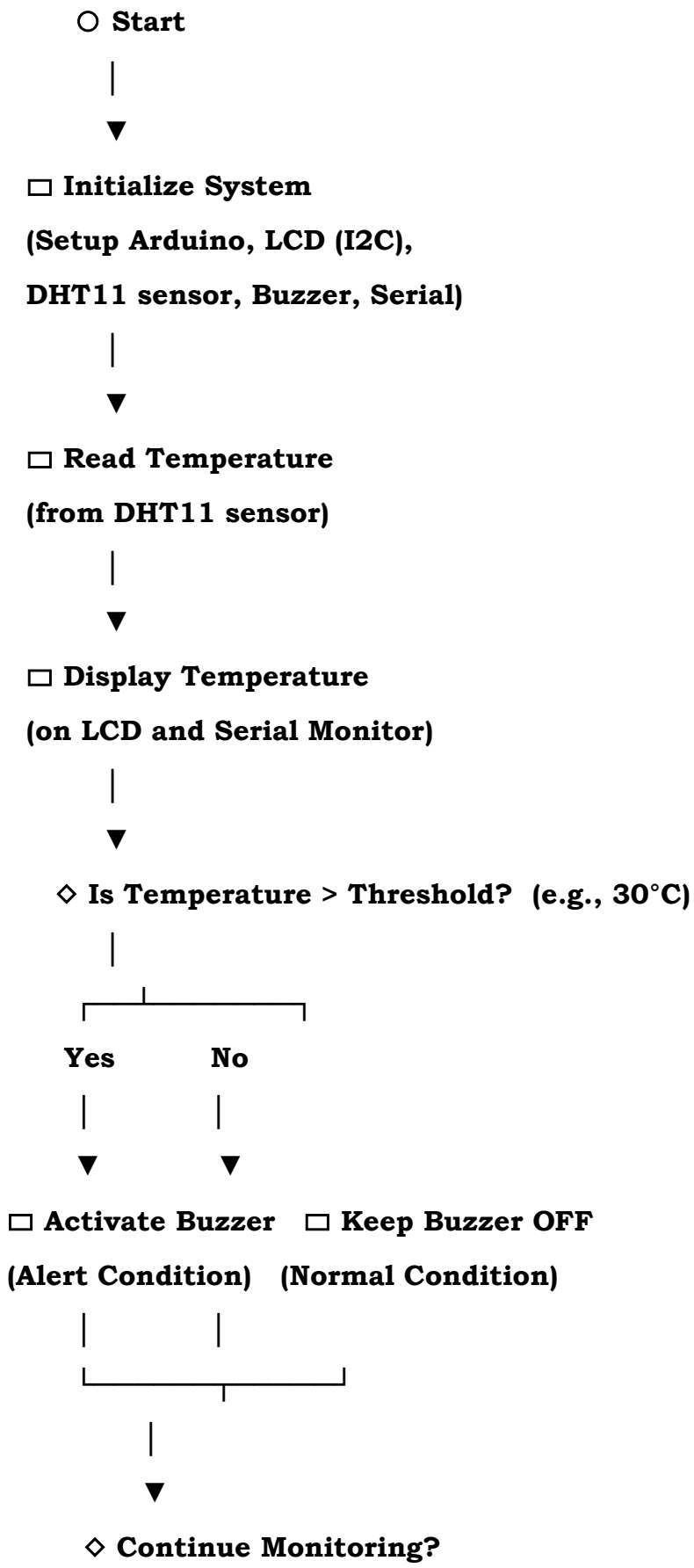
Identify Key Components	Define Relationships	Choose a Tool
Create a New Diagram	Represent Components	Indicate Data Flow
Label Components	Draw Connections	Group Components
Include External Dependencies	Add Annotations	Review and Iterate

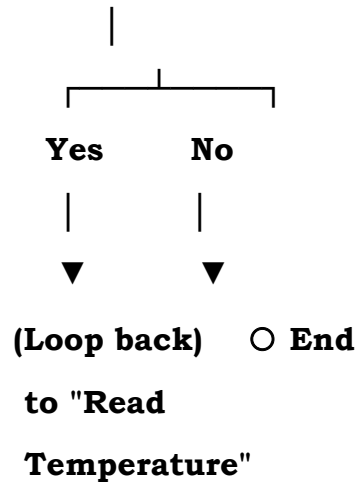
EXAMPLE: TEMPERATURE MONITORING





Follow chart diagram for Temperature diagram





Use the following symbols

Symbol	Meaning
○	Start / End
▭	Process / Action
◇	Decision / Condition
→	Flow direction

2.7.Identification of firmware module

2.7.1 Definition

A firmware module **is a** part of the firmware code **that manages a** specific hardware component **or performs a** specific function **within the system.**

In firmware development, a module is like a small, specialized team within a bigger team.

Each module has a specific job or task to make sure everything works well together.

2.7.2 Examples in table

Module Name	Main Job	Hardware Managed
Sensor Module	Reads data from sensors	Temperature sensor (e.g., DHT11)
Display Module	Shows information to the user	LCD (I2C Display)
Control Module	Makes decisions or controls outputs	Arduino internal logic or MCU
Buzzer Module	Activates alarm	Buzzer
Communication Module	Sends data to PC or other devices	Serial/USB/UART interface

2.7.3 Common methods to identify firmware modules:

1. File Signatures and Headers:

-File Extensions: Check for common extensions like .bin, .hex, .rom, or .fw.

-Magic Numbers: Look for specific byte sequences that indicate firmware formats (e.g., 0x55AA for U-Boot).

-File Headers: Examine headers for information like firmware name, version, target device, and checksum.

2. Use Binary Analysis Tools:

-Binwalk: use Binwalk software to Scans firmware images for embedded files and file systems.

-Firmware Analysis Toolkit (FAT): Collection of tools for firmware analysis, including header identification.

-IDA Pro, Ghidra: use those software (Disassemblers and debuggers) can help you to identify code sections and modules.

3. Documentation and Resources:

-Vendor Documentation: Refer to manufacturer datasheets, manuals, or firmware update notes.

-Online Resources: Search forums, communities, or repositories like GitHub for information on specific firmware types.

4. Reverse Engineering Techniques:

-Disassembly: Analyze firmware code to understand its structure and modules.

-Pattern Matching: Look for common code patterns or strings that might indicate module boundaries.

2.7.4: Steps for identifying firmware module

- **Understand the Jobs:**
Identify all the main tasks your firmware needs to perform (e.g., sensing, controlling, communicating).

- **Break Into Smaller Parts:**
Divide big tasks into smaller, manageable parts—each part becomes a **module**.

- **Reuse When Possible:**
Create reusable modules that can be used in other projects (like reusable tools).

- **Plan Communication:**
Define how modules will **communicate** and share data with each other.

- **Group Similar Tasks:**
Combine related functions into one module (e.g., a "Control Module" for all control operations).

- **Keep It Simple:**
Each module should focus on **one main job** to make the system easier to understand and maintain.

- **Handle External Interfaces:**
Include modules that interact with external devices such as **sensors, displays, or actuators**.

- **Draw a Diagram:**
Use diagrams to visually show the **relationship** between modules.

- **Define Limits:**
Note constraints for each module (e.g., memory, timing, power usage).

- **Document Each Module:**
Clearly describe what each module does and how it interacts with others.

- **Name Clearly:**
Give every module a **descriptive name** that reflects its purpose.

- **Ensure Collaboration:**
Make sure all modules **work together** to achieve the overall firmware goal

2.7.1 Creating firmware modules

2.7.1.0 Firmware architecture diagram types

1.Firmware Architecture Diagram:

Purpose: Illustrates the overall structure and organization of the firmware system, including the main components, modules, and their interactions.

Elements: Modules, components, interfaces, and their relationships.

2.Flowchart:

Purpose: Represents the flow of control or data within a specific process or algorithm.

Elements: Process steps, decision points, input/output, and flow arrows.

3.State Diagram:

Purpose: Illustrates the different states that a system or a part of the firmware can be in and the transitions between these states.

Elements: States, transitions, events, and actions.

4.Sequence Diagram:

Purpose: Shows the interactions and order of messages between different components or modules over time.

Elements: Lifelines (objects), messages, activations, and time.

5.Use Case Diagram:

Purpose: Describes the different ways users (actors) interact with the firmware system, showcasing its functionalities.

Elements: Actors, use cases, and relationships.

6.Class Diagram:

Purpose: Represents the static structure of the firmware system by illustrating classes, their attributes, methods, and relationships.

Elements: Classes, attributes, methods, associations, and inheritance.

7.Component Diagram:

Purpose: Shows the organization and dependencies among components in the firmware system.

Elements: Components, interfaces, dependencies, and relationships.

8.Deployment Diagram:

Purpose: Illustrates how software components are deployed on hardware nodes or devices.

Elements: Nodes (hardware), components, and relationships.

9.Data Flow Diagram (DFD):

Purpose: Describes the flow of data within the firmware system, depicting processes, data stores, and data flows.

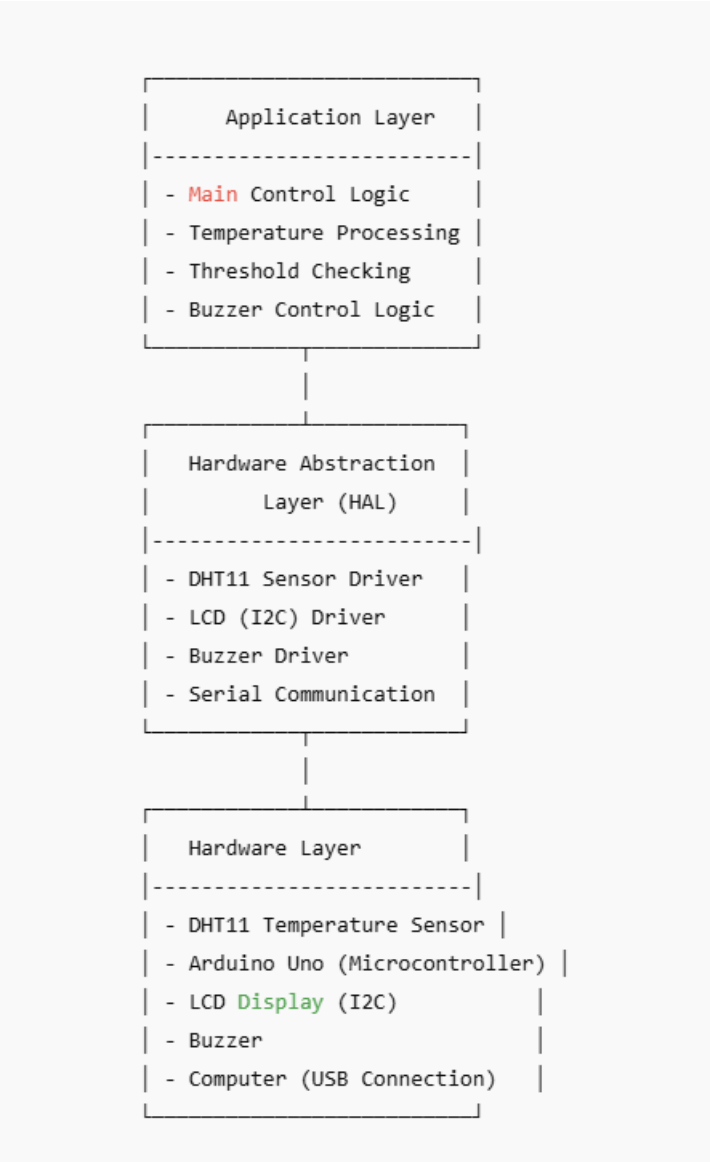
Elements: Processes, data stores, data flows, and external entities.

10.Activity Diagram:

Purpose: Represents the flow of activities or actions within a specific process or use case.

Elements: Activities, transitions, decisions, and forks/joins.

2.7.1.1 Create Functional flow diagram (flow chart diagram) for firmware (temperature monitoring system)



Layer	Responsibilities	Example in Your Project
Hardware Layer	Physical devices	DHT11 sensor, LCD (I2C), Buzzer, Arduino board

Hardware Abstraction Layer (HAL)	Low-level drivers that handle direct communication with hardware	DHT.h library for sensor, LiquidCrystal_I2C.h for LCD, pin control for buzzer
Application Layer	Main logic, system behavior, decision making	Reading temperature, displaying it, checking thresholds, turning buzzer ON/OFF, looping continuously

2.7.1.2 Create Hardware Block Diagram

A hardware block diagram visually represents the major hardware components and their connections within a system.

2.7.1.2.1. Steps to Create a Hardware Block(hardware block) Diagram

1. **Identify Key Components**
   - List main hardware parts (e.g., microcontroller, sensors, actuators, power, communication interfaces).
2. **Define Component Functions**
   - Describe what each component does in the system.
3. **Group by Functionality**
   - Organize components by role (e.g., sensing, control, communication, power).
4. **Determine Interconnections**
   - Identify how components connect (signal lines, communication buses, power lines).
5. **Select Drawing Tool**
   - Use tools like Draw.io, Lucidchart, or Visio to create the diagram.
6. **Create New Diagram**
   - Open a new project specifically for the hardware block diagram.
7. **Represent Components as Blocks**
   - Draw each component as a labeled rectangle or block.
8. **Connect Blocks with Lines**
   - Use arrows or lines to show data, control, or power flow between components.
9. **Use Standard Symbols**
   - Apply standard hardware symbols for clarity (sensor, MCU, display, etc.).
10. **Add Labels & Annotations**
    - Include details like signal names, voltages, or communication types.
11. **Show Power Distribution**
    - Indicate how power is supplied to each component.

12. **Organize Layout**
    - Arrange blocks clearly—group related components and ensure easy readability.
13. **Include External Interfaces**
    - Show any external devices (e.g., computer, network connections).
14. **Review & Refine**
    - Check accuracy, gather feedback, and make improvements.
15. **Save & Share**
    - Export the final diagram (PDF/PNG) for documentation and collaboration.

2.7.1.2.2 Structure for hardware block diagram (Temperature monitoring embedded system)

Hardware Components for Temperature Monitoring System:

-Temperature Sensor → Reads temperature (e.g., LM35, DHT11, DHT12,DS18B20)

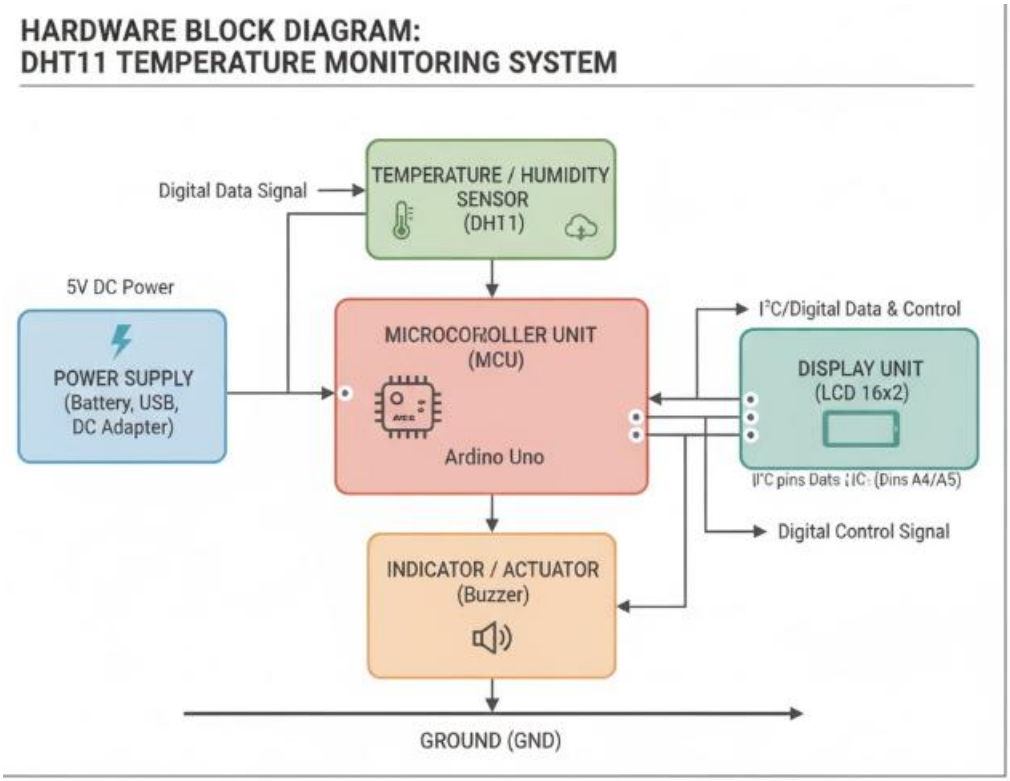
-Microcontroller (MCU) → Processes sensor data (e.g., Arduino, STM32)

-Power Supply → Powers MCU and sensors (Battery or Adapter)

-Display Unit → Shows temperature (e.g., LCD, 7-segment, OLED)

-Communication Module (optional) → Sends data remotely (e.g., Wi-Fi, Bluetooth)

-Actuator/Control (optional) → Turns on/off cooling/heating devices



2.Components Block Diagram: Basic Smart Irrigation System

Category	Component	Function	Data/Control Flow
Power Supply	DC Adapter / Battery	Provides electrical power to all components.	Supplies power to the **Arduino Uno** and **Water Pump** (via the Relay).
Input Sensor	**Soil Moisture Sensor**	Measures the level of moisture in the soil.	Outputs an **Analog Signal** to the **Arduino Uno**.
Controller	**Arduino Uno (MCU)**	**Reads the sensor**, runs the "learning" logic, and determines when to water and what to display.	Reads **Analog Input** from the sensor; sends **Digital Control** signals to the Relay, LCD, and Buzzer.
Actuator Control	**Relay Module**	An electrically controlled switch that allows the low-power Arduino to turn the high-power **Water Pump** ON or OFF.	Receives a **Digital Control Signal** from the Arduino.
Actuator	**Water Pump**	Pumps water to the plants.	Controlled by the **Relay**.
Output Display	**LCD** (16x2)	Displays the current soil moisture	Receives **Data Signal** from the Arduino.

		percentage and system status ("Watering," "Dry," "Idle").	
Output Indicator	**Buzzer**	Provides audible alerts (e.g., when watering starts or finishes).	Receives a **Digital Control Signal** from the Arduino.

2.7.2 Represent Firmware Modules

In **firmware development**, a **firmware module** is a self-contained part of the firmware that performs a specific function or task within the system.

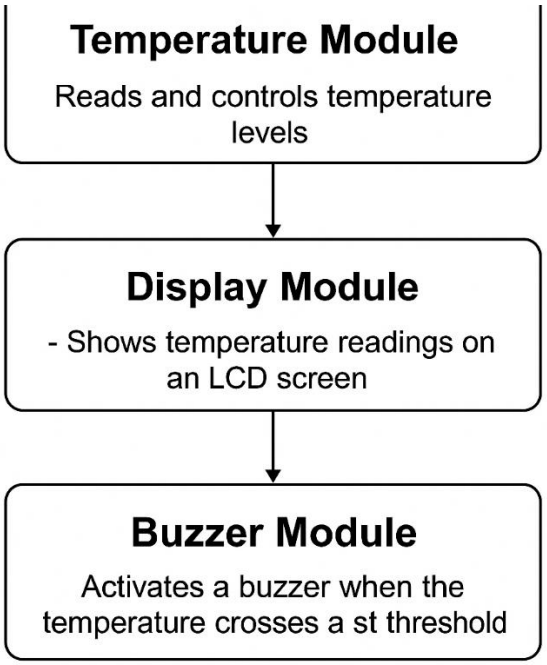
To **"represent a firmware module"** means to **visually or conceptually show a specific part of the firmware system**—its function, inputs, outputs, and how it interacts with other modules.

Example of firmware module are:

Temperature module :control temperature in temperature monitoring

Display module : allow temperature to be displayed on LCD

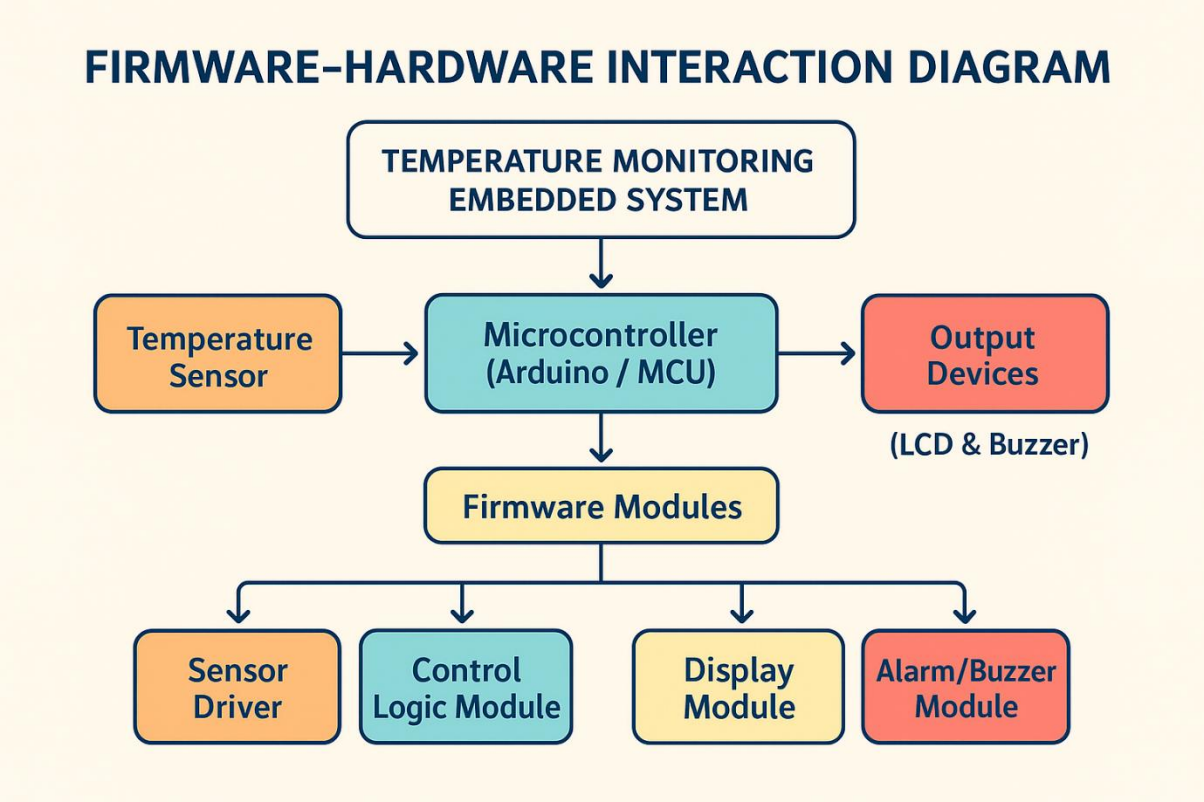
Buzzer module : allow buzzer to beep when threshold get to max value



2.7.3 Hardware Interaction

Creating a firmware hardware interaction diagram involves illustrating how the firmware components interact with hardware elements.

Firmware hardware interaction for temperature monitoring



1.Sensor Driver or module : Include header for sensor, definition for sensor as type of sensor and soon

2.Control module : In the **Firmware–Hardware Interaction Diagram**, the **Control Logic Module** (or simply **Control Module**) is the **"decision-making" part** of the firmware.

It is the firmware section that:

-Processes data coming from the sensor (temperature readings).

-Compares the temperature value to a set **threshold** (for example, 30°C).

-Decides what action to take next — for example:

- If temperature < 30°C → keep normal operation.
- If temperature ≥ 30°C → activate buzzer and display "High Temp".

3.Display module : code for allow temperature value to be displayed in the LCD Display

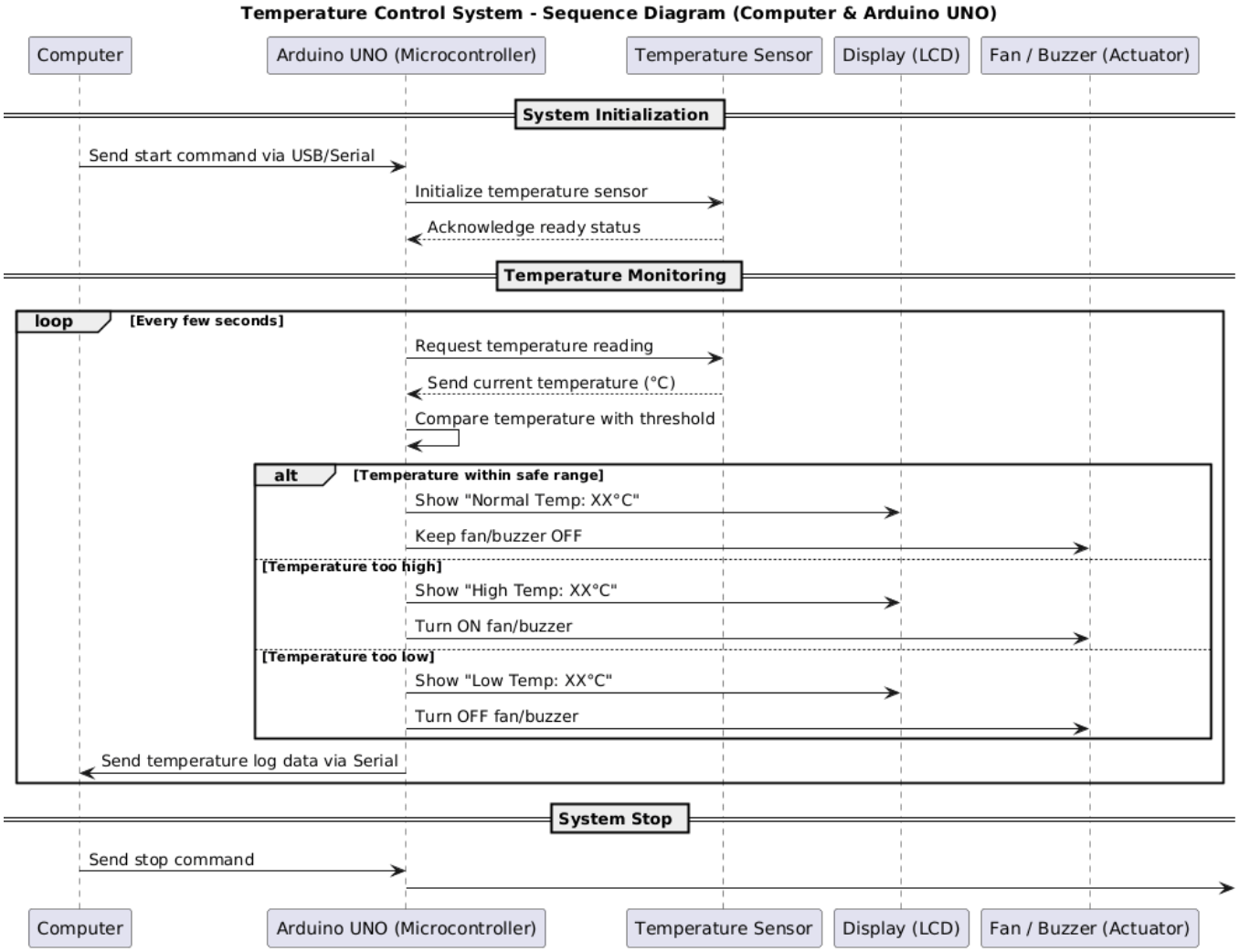
4.Buzzer Module: Code for output high or Low (ON/OFF)

Difference between the control module and Error handling module

Module Type	Purpose	Example
Control Logic Module	Normal decision-making (system behavior)	Turns buzzer ON when temperature too high
Error Handling Module	Fault detection & safe response	Shows “Sensor Error” if sensor disconnected

2.7.4 Data Flow and Communication (Sequence Diagram)

Creating a sequence diagram involves illustrating the interactions and messages exchanged between different components or objects over a period of time.



2.7.3 Documentation of firmware architecture

Steps to Create Firmware Architecture Documentation

1. **Understand the Architecture:**
   Study the firmware’s components, modules, interactions, and dependencies.
2. **Define the Scope:**
   Decide what aspects to document and the level of detail needed.
3. **Select Tools:**
   Choose documentation tools such as Word, Markdown, or LaTeX.

4. **Create Overview:** Provide a high-level description of the firmware's purpose, goals, and system context.
5. **Add Architecture Diagrams:** Include block, module interaction, and hardware diagrams to visualize the structure.
6. **Detail Key Components:** Describe main modules, their functions, and relationships.
7. **Document Interfaces:** Specify internal and external interfaces, data formats, and communication protocols.
8. **Explain Data Flow:** Show how data moves, is processed, and stored in the system.
9. **Specify Hardware Requirements:** Include details about microcontrollers, memory, and peripherals.
10. **Describe Error Handling:** Explain error detection, logging, and recovery mechanisms.
11. **Consider Security:** Document authentication, authorization, and encryption methods.
12. **List Dependencies:** Record software and hardware dependencies.
13. **Include Future Enhancements:** Suggest possible improvements or updates.
14. **Maintain Revision History:** Track changes and updates to the documentation.
15. **Review and Update Regularly:** Ensure the documentation stays current and accurate.
16. **Share and Collaborate:** Make it accessible to all stakeholders for feedback and coordination.

Here are key elements commonly found in firmware architecture documentation:

-Introduction – Purpose and goals of the project.

- **Overview** – Brief summary of how the system works and what problem it solves.

- **Architecture Diagrams** – Simple drawings showing how components connect.

- **Main Components** – List and explain each part's function.

- **Interfaces & Data Flow** – Show how data moves between parts.

- **Hardware & Software Requirements** – List all tools and components used.

- **Error Handling & Security** – Explain how errors are managed or prevented.

- **Future Improvements** – Suggest ways to enhance the project later.

- **Revision History** – Record changes or updates made.

Learning outcome 3: Implement firmware system design

3.1 Identification of programming Languages used for firmware development

- ✓ **Types**

1.Python

Python is an understood high-level programming language for general-purpose programming.

is a computer programming language often used to build websites and software, automate tasks, and conduct data analysis.

Advantages	Disadvantages
Extensive libraries	Speed limitations.
Improved productivity.	Undeveloped data base access layer.
Free and open source.	Design restrictions.

2. C

C is a general-purpose programming language created by Dennis Ritchie at the Bell Laboratories in 1972.

It is a very popular language, despite being old.

The main reason for its popularity is because it is a fundamental language in the field of computer science

Advantages	Disadvantages
C programming language is a building block for many other currently known languages	It doesn't support object oriented programming Such as inheritance ,encapsulation ,polymorphism etc.
It has the ability to extend itself.	It does not offer data security
C programming language is easy to learn	It does not support reusability of source code.

3. C++

is an object-oriented programming (OOP) language that is viewed by many as the best language for creating large-scale applications.

C++ is a superset of the C language.

Advantages	Disadvantages
Portability allows developing programs irrespective of Hardware	When C++ used for web applications complex and difficult to debug
C++ is an object oriented embedded language.	C++ can't support garbage collection
It allows moving the program development for one Platform to another platform.	It has no security.

4. C#

C#(pronounced as c-sharp) is a general-purpose high-level programming language supporting multiple paradigms.

C# encompasses static typing, strong typing,

lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines.

+

Advantages	Disadvantages
The .net class library will allows for rapid prototype development ,it will do a ton of things for you	C# is less flexible than C++.C# depends greatly on .NET Frame works.
Automatic garbage collection	C# is slower to run.
Strong memory backup	.NET application needs a window platform to execute.

5. **Rust**

- **Advantages**: Strong memory safety guarantees, performance close to C/C++, good for systems program ming.
- **Disadvantages**: Still evolving, fewer libraries and tools compared to more established languages.

✓ ***Comparison of Languages for Firmware Development:***

-Efficiency: Assembly and C/C++ are highly efficient due to their close-to-hardware nature, while Python and other high-level languages might sacrifice speed for ease of development.
-Portability: C/C++ are relatively portable across different architectures, while Assembly is very architecture-specific.
-Memory Safety: Languages like Rust offer stronger memory safety, reducing the risk of certain bugs and vulnerabilities compared to C/C++.
-Development Time: Higher-level languages like Python might allow for faster development due to their ease of use, but may sacrifice performance.
-Community & Support: C/C++ have a large community and extensive libraries/tools, while newer languages like Rust might have a smaller ecosystem.

-Resource Constraints: For memory or resource-constrained environments, Assembly or C/C++ might be preferred due to their lower-level control.

3.2 Preparation of Development Environment for Firmware Development

3.2.1. Integrated Development Environment (IDE)

Definition:
An IDE is software that provides tools for **writing, testing, debugging, and managing firmware or embedded software**.

3.2.2 Types of IDEs:

-Microcontroller-Specific IDEs – Tailored to a specific microcontroller (e.g., Keil uVision).

-Cross-Platform IDEs – Support multiple architectures (e.g., PlatformIO, Visual Studio Code with PlatformIO).

-General-Purpose IDEs with Embedded Support – Extensions for embedded development (e.g., Eclipse CDT, Visual Studio with VisualGDB).

-Safety-Critical System IDEs – For high-reliability applications (e.g., Ada).

-Lightweight IDEs – For rapid prototyping (e.g., Arduino IDE).

-RTOS IDEs – For real-time systems (e.g., SEGGER Embedded Studio).

-IoT Development IDEs – Tailored for IoT platforms (e.g., Arduino IDE, PlatformIO).

-Machine Learning Firmware IDEs – For embedded AI/ML applications.

3.2.3 Key Features of an IDE

-Code Editor: Helps write and edit code efficiently.

-Compiler/Build Tools: Converts code into executable form.

-Debugger: Detects and fixes code errors.

-Project Management: Organizes code and settings.

-Simulator/Emulator: Tests code without hardware.

-Version Control Integration: Tracks changes over time.

-Hardware Debugging Support: Tests on real devices.

-Peripheral Configuration Tools: Simplifies hardware setup.

-Library Management: Reuse existing code easily.

Interactive Development: Instant code testing.

-Cross-Platform Support: Runs on Windows, Mac, Linux.

-Extensions & Plugins: Adds extra functionality.

-Documentation Integration: Access guides within IDE.

-Collaboration Tools: Supports team development.

-Performance Profiling: Optimizes code speed and efficiency.

3.2.4. Installation of IDE

-Download: Get the installer from the official website.

-Install: Run the installer and follow instructions.

-Configure: Adjust settings, add plugins, and set up tools like compilers and debuggers.

3.2.4.1 Example – Arduino IDE Installation (Windows):

-Download from Arduino website.

-Run the installer and follow the wizard.

-Install USB drivers if needed.

-Finish installation and run Arduino IDE.

3.2.4.2 Examples of Arduino board

- Atmel AVR

- Arduino

-AVR

-ATX Mega

-ARM

-8051

-ATmega 328P

3.2.4.3 Comparison of Arduino and atmel studio

Here is a simplified comparison of **Arduino IDE** and **Atmel Studio**:

Feature	Arduino IDE	Atmel Studio
Purpose	Simple environment for beginners.	Advanced environment for professionals.

Audience	Hobbyists and beginners.	Engineers and advanced users.
Supported Boards	Mainly Arduino boards.	AVR, ARM microcontrollers
Ease of Use	Very easy to use.	More complex, steeper learning curve.
Programming Language	Simplified C/C++.	Full C/C++ with advanced features.
Debugging	Basic, with Serial Monitor.	Full debugging support (breakpoints, etc.).
Platform Support	Windows, macOS, Linux.	Mostly Windows.
Project Management	Simple projects.	Advanced project handling.
Library Management	Easy to install libraries.	Manual library inclusion.
Optimization	Limited control.	Full control over hardware.
Use Cases	Small projects, DIY electronics(**Do It Yourself electronics**,).	Professional, complex embedded systems.

3.2.5. Configuration of IDE

-Setting Preferences: Customize editor themes, fonts, and other display options.
-Adding Plugins: Install additional plugins or extensions to enhance functionality.
-Setting up Tools: Configure build tools, compilers, debuggers, and version control systems.
-Project Configuration: Create or import projects, set project-specific settings, and manage dependencies.

3.3 Communication Protocols in Firmware Development

A communication protocol is a set of rules that governs how data is exchanged between components or devices in an embedded system.

Communication:
It is the process of transferring information between devices or system components.

3.3.1 Importance of Communication Protocols

-Interoperability: Ensures different devices and components can work together.

-Data Integrity: Detects and corrects errors to maintain accurate data.

-Efficient Data Exchange: Optimizes resource use and reduces communication overhead.

-System Coordination: Enables synchronized operation of devices.

-Scalability: Supports adding new components while maintaining compatibility.

-Error Handling: Prevents data corruption and ensures reliable communication.

-Hardware Abstraction: Hides hardware details, simplifying development.

-Security: Protects sensitive data from unauthorized access.

-Compatibility: Allows devices from different sources to work together.

-Predictable Timing: Ensures timely data transmission, important for real-time systems.

-Debugging and Testing: Simplifies troubleshooting and issue resolution.

-Standards Compliance: Meets regulatory and industry requirements.

3.3.2 Types, feature, advantages, disadvantages ,applications of communication protocols

Protocol	Definition	Characteristics	Advantages	Disadvantages	Applications
I2C	Serial, synchronous communication protocol for multiple devices on a single bus.	-Two wires (SDA, SCL), -Multi-master/slave, -Low speed -Provide addressing	Simple wiring, supports multiple devices, widely used, low pin count.	-Relatively slow, -Llimited distance.	Multiple Sensors connection (temperature, humidity,
SPI	Serial, synchronous communica	-Four wires (MOSI, MISO, SCLK, CS),	-High speed, -Precise control,	-More wiring than I2C;	-SD cards, -ADC/DAC modules converter

	tion protocol for high-speed data exchange between microcontroller and peripherals.	-Full-duplex, -Fast, -Multiple slaves via CS lines. **-MISO (Master In Slave Out):**Carries data from the slave device **to the master**. **-MOSI (Master Out Slave In):**Carries data from the master **to the slave device**. **-SCLK (Serial Clock):**Clock signal generated by the master to synchronize data transfer. **-CS – Chip Select** (sometimes called **Slave Select, SS**) Activates a specific slave device for communication; **active**	-Full-duplex, -Suitable for high-data-rate devices.	-Complex with many devices,.	

		low in most cases.			
UART	Serial, asynchronous point-to-point communication protocol.	Two wires (TX, RX), no clock line, configurable baud rate, full-duplex.	Simple to implement, widely supported, ideal for debugging and serial links.	Point-to-point only, limited distance, no built-in error checking.	-PC communication, -Bluetooth/ Wi-Fi modules, -GPS, inter-microcontroller communication.
USB	Serial, host-slave communication protocol for connecting devices to PCs or hosts.	High speed, standardized connectors, plug-and-play, can supply power, host-controlled.	High speed, standardized, supports multiple devices via hubs, plug-and-play.	Requires drivers, more complex to implement, slightly higher resource use.	-PC interfacing, -Firmware updates, -USB storage, -Keyboards, -IoT boards, cameras.

3.3.3. Scenarios for each communication protocols

Scenario	Strengths	Weaknesses	Most Suitable Protocol & Justification
1. Connecting a microcontroller to multiple sensors over a short distance	I2C: Supports multiple devices on the same bus; simple wiring (2 wires).	I2C: Relatively low speed; needs pull-up resistors.	**I2C** – Ideal for multiple sensors due to multi-device addressing and minimal wiring.
2. Debugging firmware by connecting to a PC for serial data transfer	UART: Simple to implement; widely supported by PCs; low overhead.	UART: Point-to-point only; limited distance; no error correction by default.	**UART** – Best for direct serial communication with PC; easy to set up for debugging.
3. Interfacing a microcontroller with a high-speed display module	SPI: High-speed data transfer; full-duplex; can drive multiple	SPI: Requires more wires than I2C; complex	**SPI** – Provides the speed needed for high-

	devices with separate CS lines.	wiring for many devices.	performance display updates.
4. Storing sensor data on an external EEPROM	I2C: Multi-device support; simple connections; widespread support.	I2C: Slower than SPI; not ideal for very large, frequent data transfers.	**I2C** – EEPROMs are usually I2C-based; simple wiring suffices.
5. Communicating between two microcontrollers	UART: Simple, low-cost; full-duplex.	UART: Limited to one-to-one communication; requires matching settings.	**UART** – Perfect for point-to-point microcontroller communication.
6. Reading data from an SD card in a data logger	SPI: Fast; handles large data transfers efficiently.	SPI: Extra wires and chip select management needed.	**SPI** – Provides high-speed data transfer needed for logging large files.
7. Connecting a microcontroller to multiple low-speed sensors (temperature, humidity)	I2C: Supports multiple sensors on one bus; minimal wiring.	I2C: Not suitable for high-speed requirements.	**I2C** – Efficient and simple for multiple low-speed sensors.
8. Wireless module communication (e.g., Bluetooth)	UART: Widely supported; easy to interface.	UART: Limited distance; no multi-device support.	**UART** – Standard for many Bluetooth or Wi-Fi modules; easy integration.
9. Controlling multiple LED drivers simultaneously	SPI: Fast; allows multiple devices with individual CS lines.	SPI: Extra wiring for multiple devices.	**SPI** – High speed and precise control for multiple LEDs.
10. Connecting a microcontroller to a PC for firmware updates	USB (not UART/I2C/SPI in this case): High speed; plug-and-play; power supply.	USB: Slightly complex to implement on microcontroller; needs drivers.	**USB** – Ideal for firmware updates and PC interfacing; fast and standardized.

3.3.4: Effectiveness of each communication protocol

Protocol	Speed	Number of Devices	Complexity	Data Integrity	Typical Use Case	Highlight / Notes
UART	Moderate (up to 1 Mbps)	2 (point-to-point)	Very Low	Low–Moderate	Debugging, GPS modules	☑ Simple, easy to implement
SPI	Very High (tens of Mbps)	Limited by CS lines (4–8 typical)	Moderate	Moderate	Flash memory, displays, sensors	☑ Fastest speed
I2C	Moderate (100 kbps–3.4 Mbps)	Many (up to 127 devices)	Moderate	Moderate–High	Multi-sensor systems, EEPROMs	☑ Best for multi-device buses
USB	Very High (480 Mbps USB2, 5 Gbps USB3)	Many (127 via hubs)	High	High	PC communication, storage, cameras	☑ Highest data integrity, reliable

Best Highlights:

-Speed: SPI

-Number of devices: I2C / USB

-Simplicity: UART

-Data integrity: USB

-Multi-device embedded networks: I2C

3.4 Identification of default segments of data memory

data memory" refers to the portion of a computer's memory that is used for storing and managing data during the execution of a program.

In firmware development, **data memory** is the portion of a microcontroller's memory used to store and manage program data during execution. It is typically divided into **default segments**.

3.3.4.2: Types of data memory

- **.data** – Stores initialized global and static variables.

- **.bss** – Allocates memory for uninitialized global and static variables.

-**Heap** – Used for dynamic memory allocation, usually for global or dynamically created variables.

- **Stack** – Used for local variables and function call management, with static memory allocation.

Differ from stack and heap

Feature	Stack	Heap
Memory Allocation	Static	Dynamic
Data Structure	Linear	Hierarchical (tree-like)
Usage	Local variables, function calls	Global variables, dynamic memory
Size	Limited, fixed by OS	Flexible, decided by programmer
Memory Management	Automatic	Manual
Contiguity	Contiguous blocks	Random allocation
Access Speed	Fast	Slower
Implementation	Array, linked list, dynamic memory	Array, tree
Common Issues	Stack overflow due to fixed size	Memory fragmentation
Variable Scope	Fixed	Can vary

3.3.5 Performance of memory management in embedded C

3.3.5.1 Static memory allocation:
Static memory allocation is the process of reserving a fixed amount of memory for variables or data structures **at compile time**, before the program runs.

3.3.5.2 Characteristics:

-Memory is allocated during compilation.

-Size of memory is fixed and known in advance.

-Variables have predetermined size and location in memory.

-Memory management is automatic (handled by the compiler).

3.3.5.3 Advantages vs disadvantages of static memory allocation

Advantages	Disadvantages
- Fast memory access (allocated at compile time) - Predictable memory usage - Simple to implement and managed automatically by compiler	- Inflexible size (cannot change at runtime) - May waste memory if allocated more than needed - Not suitable for dynamic or variable-sized data

3.3.5.3 Example

3.3.5.4 Example for Arduino code for temperature monitoring on static memory allocation

```
#include <DHT.h>

#include <Wire.h>

#include <LiquidCrystal_I2C.h>


// Sensor and LCD configuration

#define DHTPIN 2

#define DHTTYPE DHT22

#define LCD_ADDRESS 0x27

#define LCD_COLUMNS 16

#define LCD_ROWS 2


DHT dht(DHTPIN, DHTTYPE);

LiquidCrystal_I2C lcd(LCD_ADDRESS, LCD_COLUMNS, LCD_ROWS);


// Static memory allocation for temperature readings

const int NUM_READINGS = 5;

float temperatures[NUM_READINGS];
```

```
float humidities[NUM_READINGS];

void setup() {
  Serial.begin(9600);
  dht.begin();

  lcd.init();
  lcd.backlight();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Initializing...");
  delay(2000);
}

void loop() {
  // Read temperature and humidity and store in static arrays
  for (int i = 0; i < NUM_READINGS; i++) {
    float temp = dht.readTemperature();   // Celsius
    float hum = dht.readHumidity();

    if (isnan(temp) || isnan(hum)) {
      Serial.println("Failed to read from DHT sensor!");
      lcd.clear();
      lcd.setCursor(0, 0);
      lcd.print("DHT Read Error");
```

```
    delay(2000);
    return;
  }

  temperatures[i] = temp;
  humidities[i] = hum;
  delay(1000); // Small delay between readings
}

// Print readings to Serial Monitor
Serial.println("=== Temperature Readings ===");
for (int i = 0; i < NUM_READINGS; i++) {
  Serial.print("Temp: ");
  Serial.print(temperatures[i]);
  Serial.print("C, Hum: ");
  Serial.print(humidities[i]);
  Serial.println("%");
}

// Display the latest reading on LCD
lcd.clear();
lcd.setCursor(0, 0);
lcd.print("Temp: ");
lcd.print(temperatures[NUM_READINGS - 1]);
lcd.print("C");
```

```
  lcd.setCursor(0, 1);
  lcd.print("Hum: ");
  lcd.print(humidities[NUM_READINGS - 1]);
  lcd.print("%");

  delay(5000); // Wait before next batch of readings
}
```

For using C Static memory allocation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_READINGS 5
#define SENSOR_PIN 2 // Example pin number

// Function to simulate reading temperature from sensor
int readTemperature(int pin) {
    // In real embedded C, here you'd read digital/analog sensor values
    // For simulation, we return a random temperature between 20-30
    return 20 + rand() % 11;
}

int main() {
    // Initialize random number generator
    srand((unsigned int)time(NULL));
```

```
// Static memory allocation for temperature readings
int temperatures[NUM_READINGS];


printf("Sensor connected to pin: %d\n", SENSOR_PIN);


// Read temperatures and store in static array
for (int i = 0; i < NUM_READINGS; i++) {
    temperatures[i] = readTemperature(SENSOR_PIN);
}


// Print the temperature readings
printf("=== Temperature Readings (Static Memory) ===\n");
printf("Temperatures: ");
for (int i = 0; i < NUM_READINGS; i++) {
    printf("%d ", temperatures[i]);
}
printf("\n");


return 0;}
```

3.3.5.2 Dynamic memory allocation

Definition:
Dynamic memory allocation is the process of allocating memory **during program execution** rather than at compile time.

It allows the program to request and release memory as needed, making it suitable for data whose size is not known in advance.

3.3.5.3Characteristics:

-Memory is allocated and deallocated **at runtime**.

-Size of memory can **vary** and is determined during execution.

-Data structures or variables can **grow or shrink** as needed.

-Memory management is **manual** (requires explicit deallocation) or automatic (with garbage collection in some languages).

3.3.5.4 Advantages and disadvantages of Dynamic memory allocation

Advantages	Disadvantages
- Flexible memory usage (can allocate size at runtime) - Useful when the amount of data is unknown beforehand - Can grow or shrink data structures as needed	- Slower access compared to static memory - Programmer must manually manage memory (risk of memory leaks) - Can cause fragmentation if used heavily

3.4.5.5 Example for Arduino code for dynamic memory allocation

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>


// Function to simulate reading temperature from a sensor

int readTemperature() {

   return 20 + rand() % 11;  // Random temperature between 20 and
30

}


// Function to simulate reading humidity from a sensor

int readHumidity() {
```

```
    return 50 + rand() % 21;  // Random humidity between 50% and
70%

}


int main() {

    srand((unsigned int)time(NULL));


    int numReadings;

    printf("Enter number of temperature readings: ");

    scanf("%d", &numReadings);


    if (numReadings <= 0) {

        printf("Invalid input. Using default value 5.\n");

        numReadings = 5;

    }


    // Dynamic memory allocation

    int* temperatures = (int*)malloc(numReadings * sizeof(int));

    int* humidities = (int*)malloc(numReadings * sizeof(int));


    if (temperatures == NULL || humidities == NULL) {

        printf("Memory allocation failed!\n");

        return 1;

    }
```

```
    // Read temperatures and humidities
    for (int i = 0; i < numReadings; i++) {
        temperatures[i] = readTemperature();
        humidities[i] = readHumidity();
    }

    // Print readings
    printf("=== Dynamic Memory Temperature Readings ===\n");
    for (int i = 0; i < numReadings; i++) {
        printf("Reading %d: Temp = %d°C, Hum = %d%%\n", i + 1,
temperatures[i], humidities[i]);
    }

    // Free allocated memory
    free(temperatures);
    free(humidities);

    return 0;
}
```

For using C Dynamic memory allocation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h> // for sleep function
```

```
// Simulate reading temperature from a DHT sensor
float readTemperature() {
    return 20.0 + (rand() % 1000) / 100.0; // 20.0°C to 30.0°C
}

// Simulate reading humidity from a DHT sensor
float readHumidity() {
    return 50.0 + (rand() % 2000) / 100.0; // 50% to 70%
}

int main() {
    srand((unsigned int)time(NULL));

    int numReadings;
    printf("Enter number of readings: ");
    scanf("%d", &numReadings);

    if (numReadings <= 0) {
        printf("Invalid input! Exiting program.\n");
        return 1;
    }

    // Dynamic memory allocation
    float* temperatures = (float*)malloc(numReadings * sizeof(float));
    float* humidities = (float*)malloc(numReadings * sizeof(float));
```

```
if (temperatures == NULL || humidities == NULL) {
    printf("Memory allocation failed!\n");
    return 1;
}

printf("=== Reading Temperature and Humidity ===\n");

// Read temperatures and humidities dynamically
for (int i = 0; i < numReadings; i++) {
    temperatures[i] = readTemperature();
    humidities[i] = readHumidity();

    printf("Reading %d: Temp = %.2f°C, Hum = %.2f%%\n",
           i + 1, temperatures[i], humidities[i]);

    sleep(1); // simulate 1-second delay
}

// Display last reading (simulate LCD output)
printf("\n=== Latest Reading ===\n");
printf("Temp: %.2f°C\n", temperatures[numReadings - 1]);
printf("Hum: %.2f%%\n", humidities[numReadings - 1]);

// Free allocated memory
```

```
    free(temperatures);

    free(humidities);


    return 0;

}
```

3.4.6 Embedded system firmware nose

3.4.7 Definition: Embedded system noise **refers to** unwanted electrical or signal disturbances **that interfere with the** normal operation **of an embedded system or its sensors.**

3.4.8 Types of noise

Type of Noise	Description	Typical Cause
White Noise	Random fluctuations without a pattern	Thermal noise, electronic components
Spike Noise	Sudden, sharp changes in readings	Switching circuits, EMI, electrical interference
Drift	Slow, gradual change in signal over time	Temperature variation, sensor aging
Periodic Noise	Regular, repeating disturbance	External signals like motors, power lines (50/60 Hz)

3.4.9 Techniques used to resolve the noise

Filter Type	Purpose	Key Features	Best For
Moving Average Filter	Reduces random fluctuations (white noise)	Simple and smooths data by averaging several past readings	Removing random noise in stable signals
Median Filter	Removes sudden spikes or outliers	Uses the median of nearby values; preserves signal edges	Filtering spike noise (e.g., sensor glitches)
Kalman Filter	Handles complex, dynamic, or drifting noise	Combines sensor data with system model for optimal estimation	Advanced filtering in motion or tracking systems
Low-Pass Filter	Removes high-frequency noise	Allows low-frequency signals, blocks fast-changing disturbances	Filtering electrical or signal interference

3.4.10 :Steps for handling noise in firmware development

-Identify the Type of Noise: Identify which type of noise affecting your code

-Choose an Appropriate Filtering Technique: Select the best filtering techniques that is suitable for you firmware code

-Implementing the Filter in Firmware: Once the

appropriate filter is selected, it can be implemented directly

into the sensor's firmware. For instance

-Testing : After the filter is applied, it is important to test the

effectiveness of the filtering

-Final consideration: Ensure that the chosen filter is computationally feasible

for the hardware platform

Steps for Moving average filter:

1. Choose a window size (e.g., 3, 5, or 7 samples).
2. Take the first NNN values of the data.
3. Compute their average.
4. Slide the window by one sample and repeat.

Situation	Recommended Window Size	Reason
Fast-changing signals	3–5	Keeps response quick
Slow, stable sensors (like temperature)	5–10	Smoother output
Very noisy sensors	10–20	Reduces noise more but adds delay

Example :Given temperature reading of 12.5,25.0,40.0.22.5,13.5

Let size be 3

What will be the output reading after filtering

Answer: We can only start computing after the **first 3 readings**, because the first two don't have 3 samples yet.

Step	Readings in Window	Average (Output)
1	(12.5)	— (not enough data yet)
2	(12.5, 25.0)	— (need 3 readings)
3	(12.5, 25.0, 40.0)	(12.5 + 25.0 + 40.0)/3 = **25.83**
4	(25.0, 40.0, 22.5)	(25.0 + 40.0 + 22.5)/3 = **29.17**
5	(40.0, 22.5, 13.5)	(40.0 + 22.5 + 13.5)/3 = **25.33**

The output after filter is 25.83,29.17,25.33

Example 2:Given temperature reading 25°C, 27°C, 50°C, 26°C

Use different window size to complete the following table

Window Size	Filtered Output	Smoothness	Response Speed
			Fast
			Medium
			Slow

Note: a)complete the window sizes used

b)Complete High or Low on each window size

c)Complete with Fast, Medium ,and Slow on response time for each window size

d)Which of the following window size is better for removing the noise

e) What will be the final output

Remember that window size is the first readings .

3.4.10.1 Implementing noise filtering in the firmware code

```
#define WINDOW_SIZE 5
float sensor_data[WINDOW_SIZE];
int index = 0;
float sum = 0;
float apply_filter(float new_data) {
sum -= sensor_data[index];
sensor_data[index] = new_data;
```

```
sum += new_data;
index = (index + 1) % WINDOW_SIZE;
return sum / WINDOW_SIZE;
}
```

3.5 Description of Concepts for Developing Firmware

3.5.1 Embedded C Programming concepts

3.5.1.1 data types

In embedded C programming, data types play a crucial role in defining the type of data that a variable can hold.

Data Type	Size	Range / Power	Represents	Example	Notes
Int	2 or 4 bytes	-2^{16} (16-bit) - 2^{32} (32-bit)	Integer values	int x = 10;	Size depends on MCU (16-bit or 32-bit)
Char	1 byte	2^{8}	Character / small integers	char ch = 'A';	Signed: −128 to 127, Unsigned: 0–255
Float	4 bytes	2^{32}	Single-precision floating point	float f = 3.14;	Range is IEEE 754, not exact 2^{32}
double	8 bytes	2^{64}	Double-precision floating point	double d = 6.28;	Range is IEEE 754, not exact 2^{64}
short	2 bytes	2^{16}	Short integers	short s = 32767;	Typical range: −32,768 to 32,767
Long	4 or 8 bytes	2^{32} / 2^{64}	Long integers	long l = 1234567890;	Depends on system architecture
unsigned	same as base type	0 to max of base type	Non-negative values	unsigned int u = 42;	Modifier only, not separate type
_Bool	1 byte	0 or 1	Boolean values	_Bool flag = 1;	

3.5.1.2 Structures and Unions

A structure is a user-defined data type that allows you to group variables of different types under a single name. Here's an example of defining and using a structure:

Unions A union is similar to a structure but allows you to use the same memory location for different types of variables.

Different between the structure and union

embedded C++:

Feature	Structure	Union
Memory Usage	Allocates separate memory for each member.	Shares the same memory location for all members.
Size Calculation	Size is the sum of the sizes of all members.	Size is determined by the largest member.
Member Access	All members are accessible simultaneously.	Only one member is accessible at a time.
Usage Scenario	When you need to store multiple pieces of data and all are active at the same time.	When you need to save memory and only one member needs to be active at a time.
Initialization	Each member can be independently initialized.	Only one member can be initialized at a time.
Memory Overhead	Typically has more memory overhead.	Typically has less memory overhead.
Syntax for Declaration	**cpp struct MyStruct { int x; float y; };**	**cpp union MyUnion { int x; float y; };**
Example	**cpp struct Point { int x; int y; };**	**cpp union Data { int intValue; float floatValue; };**
Accessing Members	**cpp Point p; p.x = 1; p.y = 2;**	**cpp Data d; d.intValue = 42;**
Common Use Cases	When you need to represent a collection of related data.	When you need to conserve memory and only use one piece of data at a time.

3.5.1.3 Bit fields

In C++ firmware development , bit fields are a way to define and allocate

specific numbers of bits within a data structure. Bit fields are useful in embedded systems where memory conservation is crucial.

Ex: struct Variable name{

int temperature:8;
int pressure:12;
bool status:1;
};

3.5.1.4 Preprocessor directives

In C++, preprocessor directives are commands that are processed before the actual compilation of the code begins. They are prefixed with a # symbol.

List of Preprocessor directive

1. File Inclusion Directives

Directive	Purpose	Example
#include <file>	Includes standard library header	#include <iostream>
#include "file"	Includes user-defined header	#include "myheader.h"
#pragma once	Ensures header file is included only once	#pragma once at top of myheader.h

2. Macro and Constant Directives

Directive	Purpose	Example
#define	Defines a constant or macro	#define PI 3.14159
#undef	Undefines a macro	#undef PI
#define with arguments	Creates a macro function	#define SQUARE(x) ((x)*(x))

3. Conditional Compilation Directives

Directive	Purpose	Example
#ifdef	Compiles code **if macro is defined**	#ifdef DEBUG
#ifndef	Compiles code **if macro is not defined**	#ifndef RELEASE
#if	Conditional compilation based on expression	#if VERSION >= 2
#elif	Else-if in conditional compilation	#elif VERSION == 1
#else	Else part of conditional compilation	#else
#endif	Ends a conditional block	#endif

4.Other Useful Directives

Directive	Purpose	Example

#error	Generates a compilation error with message	#error "Unsupported platform"
#line	Changes the compiler's reported line number	#line 100 "myfile.cpp"
#pragma	Special compiler instructions	#pragma warning(disable:4996)

3.6 API and HAL

API (Application Programming Interface): A set of rules and protocols that allows software applications to communicate with each other by defining methods and data formats.

HAL (Hardware Abstraction Layer): A layer that translates software requests into hardware instructions, enabling software to control and interact with hardware components easily.

3.6.2 Differentiation between HAL and API

Aspect	Hardware Abstraction Layer (HAL)	Application Programming Interface (API)
Purpose	Abstracts hardware-specific details, providing a standardized interface to the software layer.	Defines a set of rules and protocols for different software components to interact with each other.
Scope	Focused on abstracting and managing hardware interactions.	Broad, covering communication between various software modules.
Abstraction Level	Low-level, dealing with hardware-specific instructions.	Higher-level, dealing with functionality and communication patterns.
Dependencies	Typically dependent on the specific hardware architecture.	Can use lower-level HAL functionality to interface with hardware.
Portability	Enhances hardware portability by allowing software to be independent of specific hardware details.	Enhances software portability by providing a consistent interface between different software components.

Aspect	Hardware Abstraction Layer (HAL)	Application Programming Interface (API)
Implementation	Contains functions that directly interact with hardware components.	Contains functions that facilitate communication between different software modules.
Example Function	**initializeHardware()**: Initializes hardware-specific settings.	**sendDataToPeripheral()**: Sends data to a communication peripheral.
Usage in Code	Invoked when the software needs to interact with the hardware.	Invoked when different software modules need to communicate.

3.6.3 Advantages vs disadvantages of HAL and API

Category	HAL (Hardware Abstraction Layer)	API (Application Programming Interface)
Advantages	**Abstraction**: Simplifies hardware interaction by hiding low-level details.	**Flexibility**: Offers access to a wider range of hardware functionalities compared to HALs.
	Portability: HAL functions are often generic, allowing code to be reused across different microcontrollers	**Performance**: Can be more efficient than HALs as developers have direct control over hardware interaction, potentially improving performance.
	Developer Efficiency: Reduces development time by providing pre-written functions for common hardware tasks.	**Reusability**: Well-designed APIs can be reused across different applications
Disadvantages	**Performance Overhead**: HAL functions might add an extra layer of code, potentially impacting performance-critical applications.	**Complexity**: Requires a deeper understanding of the underlying hardware compared to HALs.
	Limited Flexibility: HALs might not offer access to all hardware functionalities	**Portability Concerns**: APIs might need adjustments when porting to different hardware platforms

	Vendor Dependence: HAL implementations are often vendor-specific for microcontrollers.	**Error Prone**: Developers have more control but also more responsibility for code correctness. Incorrect API usage could lead to hardware malfunctions.

3.6.4 Characteristic for both API and HAL

Characteristic	HAL (Hardware Abstraction Layer)	API (Application Programming Interface)
Purpose	Abstracts hardware details	Exposes software functionality to higher layers
Abstraction Level	Low to medium (close to hardware)	Medium to high (software-oriented)
Portability	Makes firmware portable across hardware platforms	Promotes code reuse and modularity in software
Functionality	Provides functions to control hardware peripherals	Defines functions/services for application use
Dependency	Hardware-specific but hides hardware differences	Can be hardware-independent; may use HAL internally
Use Case	Firmware accessing hardware without dealing with registers	Application/software using firmware or library services
Security	Focus on safe hardware access (e.g., prevent misconfigurations)	Focus on authentication, access control, and input validation

3.6.5 Designing your own API

3.6.5.1 Steps to design API

 Define purpose & requirements – know what the API should do and who will use it.

 Identify resources & operations – list the data, peripherals, and functions needed.

 Design interface – decide function names, parameters, return types, and conventions.

 Define data formats & types – specify input/output types, ranges, and units.

 Plan error handling – set error codes and handle invalid inputs safely.

 Include security – add authentication, access control, and input validation if needed.

 Document API – provide usage instructions, examples, and versioning.

 Implement & test – code the API and perform unit/integration tests.

 Review & refine – optimize usability, performance, and clarity.

3.6.5.2 Designing my temperature and humidity firmware API

```
#include <DHT.h>


// -------------------- Configuration --------------------

#define DHTPIN 2

#define DHTTYPE DHT22


#define TEMP_THRESHOLD 30.0   // Temperature threshold in Celsius

#define HUMID_THRESHOLD 70.0  // Humidity threshold in %


// -------------------- API Layer --------------------

DHT dht(DHTPIN, DHTTYPE);

bool sensorReady = false;


// Initialize the sensor

bool TempHumid_Init() {

  dht.begin();

  sensorReady = true;

  return sensorReady;
```

```
}

// Read temperature in Celsius
float TempHumid_ReadTemperature() {
  if (!sensorReady) return NAN;
  return dht.readTemperature();
}

// Read humidity in %
float TempHumid_ReadHumidity() {
  if (!sensorReady) return NAN;
  return dht.readHumidity();
}

// Check thresholds
bool TempHumid_CheckThreshold(float temp, float humid) {
  return (temp > TEMP_THRESHOLD || humid > HUMID_THRESHOLD);
}

// Get sensor status
int TempHumid_GetStatus() {
  if (!sensorReady) return 1; // Sensor not initialized
  if (isnan(dht.readTemperature()) || isnan(dht.readHumidity())) return 2; // Read error
  return 0; // OK
```

```
}

// -------------------- Main Application --------------------
void setup() {
  Serial.begin(9600);
  if (TempHumid_Init()) {
    Serial.println("Temperature & Humidity Monitoring Started");
  } else {
    Serial.println("Sensor Initialization Failed");
  }
}

void loop() {
  float temp = TempHumid_ReadTemperature();
  float humid = TempHumid_ReadHumidity();

  // Check for read errors
  int status = TempHumid_GetStatus();
  if (status == 1) {
    Serial.println("Error: Sensor not initialized");
    delay(2000);
    return;
  } else if (status == 2) {
    Serial.println("Error: Failed to read sensor");
    delay(2000);
```

```
    return;
  }

  // Print readings
  Serial.print("Temperature: ");
  Serial.print(temp);
  Serial.print(" °C\tHumidity: ");
  Serial.print(humid);
  Serial.println(" %");

  // Threshold check
  if (TempHumid_CheckThreshold(temp, humid)) {
    Serial.println("ALERT: Temperature or Humidity exceeds threshold!");
  }

  delay(2000); // Wait 2 seconds before next reading
}
```

API Functions for Temperature & Humidity Monitoring

-TempHumid_Init() – Initialize the sensor.

-TempHumid_ReadTemperature() – Read current temperature.

-TempHumid_ReadHumidity() – Read current humidity.

-TempHumid_CheckThreshold() – Check if temperature or humidity exceeds thresholds.

-TempHumid_GetStatus() – Check sensor status (OK / Error)./Error handling API function

3.6.6 HAL for GPIO process

1. Identify Hardware Components: List all peripherals (GPIO, ADC, UART, I2C, SPI, Timers, Sensors, etc.) used in the system.

2. Define Hardware Interfaces: Specify how the firmware will interact with each hardware component (e.g., read, write, configure).

3. Group Hardware Functions: Group related hardware operations into logical modules (e.g., GPIO module, UART module).

4. Design HAL APIs (Function Prototypes): Define high-level functions (e.g., HAL_GPIO_WritePin(), HAL_ADC_ReadValue()).

5. Implement Low-Level Drivers: Write code that directly accesses registers or hardware. These drivers are wrapped by the HAL functions.

6. Test HAL Modules Independently: Test each HAL function with actual hardware to ensure correct operation.

7. Integrate and Optimize: Combine HAL modules into the main firmware and optimize for performance and memory.

8. Document and Maintain: Record HAL function usage, parameters, and hardware dependencies.

☑ **In Short**

HAL Design Process = Identify hardware → Define interfaces → Create modular drivers → Provide uniform APIs → Test & document.

HAL For GPIO Functions

HAL for GPIO" means the firmware has functions like:

- HAL_GPIO_Init() – set pin as input/output
- HAL_GPIO_Write() – set pin high or low
- HAL_GPIO_Read() – check pin state

3.6.7 HAL for SPI Process

Is a **software interface** that simplifies communication with SPI hardware.

Steps

Step	Description
1	Identify SPI-connected devices and pins
2	Define necessary SPI operations (init, transmit, receive)
3	Create function prototypes for abstraction

4	Implement register-level SPI control in HAL
5	Test with actual hardware
6	Integrate and use HAL functions in application code

3.6.7.1 HAL For SPI Functions

-SPI_Init(config)**:** Initializes the SPI peripheral with desired configurations (clock speed, data order, mode).

-SPI_Transmit(data)**:** Transmits a data buffer through the SPI bus.

-SPI_Receive(data_buffer)**:** Receives a data buffer from the SPI bus.

-SPI_DeInit()**:** De-initializes the SPI peripheral and releases resources.

3.7 Implementation of firmware modules

Module	Key Functions / Responsibilities	Examples / Notes
Initialization Module	Sets up the system and hardware before it starts running.	
Control Module	Manages hardware components; command reception and decoding; feedback; error handling	Motor control, LED control
Communication Module	Protocol management (Bluetooth, Wi-Fi, USB, Serial); data transmission & reception; addressing/routing; security	Smart thermostat (Wi-Fi), fitness tracker (Bluetooth)
Memory Management Module	Memory allocation/deallocation; fixed/dynamic strategies; protection; monitoring & optimization	Prevents memory leaks, optimizes memory usage
Timer Module	Configure timers; interrupt handling; read/write counter values	Generating delays, periodic tasks
Interrupt Handler Module	Detects and responds to hardware/software interrupts	External signals triggering actions
Power Management Module	Manage power modes; voltage regulation; battery monitoring	Active, idle, sleep modes; battery-powered devices
User Interface (UI) Module	Input handling (buttons, touchscreen); display management; menu navigation;	Touchscreen, LEDs, LCD

	status indication; configuration; feedback	displays; user notifications
Error Handling Module	Error detection; reporting; classification; fault diagnosis; recovery; user notification	Logs errors, resets components, alerts user
Security Module	Protect system from unauthorized access; ensure confidentiality, integrity, and availability	Encryption, authentication, cyberattack protection
Data Acquisition Module (DAQ Module):	Collects raw data from sensors and sends it to the program for use.	

Here are some key functionalities and components typically found in a Security Module

1. **Authentication:** Verify user or entity identity (passwords, biometrics, tokens, MFA).
2. **Authorization:** Enforce access control and permissions for authenticated users.
3. **Encryption:** Protect sensitive data in memory, storage, and communication using cryptographic algorithms (AES, RSA, ECC).
4. **Secure Communication:** Safeguard data transmitted between the device and external systems.
5. **Secure Boot:** Ensure integrity and authenticity of firmware/software during system startup.
6. **Secure Storage:** Protect sensitive data and cryptographic keys in non-volatile memory.
7. **Secure Firmware Updates:** Verify firmware authenticity and integrity before updates.
8. **Intrusion Detection & Prevention:** Monitor for anomalies, unauthorized access, and potential breaches.
9. **Secure Logging & Auditing:** Maintain logs of security events, user actions, and system operations.
10. **Security Policy Enforcement:** Apply security policies and ensure compliance with standards/regulations.

3.7 Writing drivers source code : Writing drivers source code is the process of developing software that allows firmware to communicate and control hardware devices**.**

3.7.1 Identification of driver interface steps

- **Specifying the functions and commands:** the driver must provide to the higher-level software.

Examples: init(), read(), write(), control().

- **Defining data formats and protocols** :used between the driver and the hardware.

Example: SPI, I2C, UART communication.

- **Determining event and error handling mechanisms**.

Example: interrupts, status flags, error codes.

- **Providing a standardized interface** so that the same driver can be used by applications without knowing hardware details.

3.7.2 Device driver models example

Driver Model	Description / Key Features
Layered Model	Drivers are organized in layers; each layer interacts only with adjacent layers; improves modularity and maintainability.
Bus-Specific Driver Model	Drivers are specific to a particular bus (e.g., PCI, USB, I2C, SPI) and handle communication with devices on that bus.
Network Driver Model	Designed for network devices; manages network protocol stacks, packet transmission/reception, and network interfaces.
Virtual Driver Model	Abstracts hardware devices using virtual devices; allows a single driver to manage multiple similar devices or emulate hardware.
Monolithic Driver Model	All drivers run in the same kernel space; offers high performance but lower isolation and stability.
Microkernel Driver Model	Drivers run in user space or separate modules outside the kernel; improves fault isolation but may reduce performance.
Message-Passing Driver Model	Drivers communicate with the kernel and other components via messages; enhances modularity and fault tolerance.
Object-Oriented Driver Model	Uses object-oriented concepts (classes, inheritance) to represent hardware devices; improves code reuse and maintainability.

3.7.2.1 Steps to Develop a UI for Embedded Firmware

1. Define User Requirements and Use Cases

Objective: Understand user needs and key functionalities.

Method: -Identify firmware settings (parameters, communication, thresholds) and monitoring data (sensor readings, logs, status).

- Create use cases like setting configurations, updating firmware, and viewing logs.

Tools: User stories, use case diagrams.

2. Choose UI Platform and Interface Type

Objective: Select appropriate interface based on hardware and user needs.

Method:

-Embedded display: GUI on LCD/OLED (e.g., LVGL, TouchGFX).

-External interface: Web, desktop, or mobile app using UART, SPI, Wi-Fi, or Bluetooth.

Tools: GUI libraries for embedded or web technologies (HTML/CSS/JS, REST/MQTT).

3. Design UI Layout and Interaction Flow

Objective: Make UI intuitive and easy to navigate.

Method: -Wireframes or mockups for configuration screens, monitoring dashboards, charts, and status indicators.

-Group related settings and ensure smooth navigation.

Tools: Figma, Adobe XD, Balsamiq, QT Designer.

4. Implement Configurable Parameters

Objective: Allow users to modify firmware settings easily.

Method: Input fields, sliders, dropdowns; validate inputs; save to non-volatile memory (EEPROM/Flash); provide feedback.

Tools: LVGL/TouchGFX, HTML forms, memory drivers.

5. Real-Time Monitoring and Data Display

Objective: Display live data and system status.

Method: Graphs, charts, gauges, color-coded indicators; refresh data at appropriate intervals.

Tools: LVGL/TouchGFX for embedded, Chart.js/D3.js for web interfaces.

6. Implement Communication Between UI and Embedded System

Objective: Ensure reliable data exchange.

Method: Use UART, SPI, I²C, Wi-Fi, Bluetooth; structured data (JSON/XML); REST/MQTT for web; handle errors with timeouts/retries.

Tools: ESP-IDF, Arduino Wi-Fi libraries, AJAX for asynchronous updates.

7. User Authentication and Security

Objective: Protect the firmware settings and data.

Method: Authentication (username/password, role-based), encryption (SSL/TLS), audit logs for tracking changes.

Tools: OAuth2, SSL/TLS, role-based access control.

8. Testing the UI

Objective: Ensure functionality, usability, and reliability.

Method:

-Unit tests for UI components

-Usability tests for user-friendliness

-Functional tests to verify interaction with firmware

-Edge case tests for input validation and communication errors

-Integrated testing

3.7.3 Example of driver source code to for firmware that control temperature and humidity

```
#include <DHT.h>

#include <LiquidCrystal_I2C.h>

#include <Wire.h>
```

```
// Define the DHT22 pin
#define DHTPIN 2
#define DHTTYPE DHT22  // DHT 22  (AM2302),
//#define DHTTYPE DHT11 // DHT 11

// Define the LCD I2C address.  Use 0x27, 0x3F, or run the I2C scanner if you're not sure.
#define LCD_ADDRESS 0x27 // Common addresses include 0x27 and 0x3F
#define LCD_COLUMNS 16
#define LCD_ROWS    2

// Create DHT and LCD objects
DHT dht(DHTPIN, DHTTYPE);
LiquidCrystal_I2C lcd(LCD_ADDRESS, LCD_COLUMNS, LCD_ROWS);

void setup() {
  // Initialize serial communication for debugging
  Serial.begin(9600);
  Serial.println("DHT22 and LCD Example");

  // Initialize DHT sensor
  dht.begin();

  // Initialize LCD
```

```
  lcd.init();
  lcd.backlight(); // Turn on the backlight
  lcd.clear();      // Clear the display

  // Display a welcome message
  lcd.setCursor(0, 0);
  lcd.print("Initializing...");
  delay(2000); // Wait 2 seconds
}

void loop() {
  // Delay for a few seconds between measurements
  delay(2000);

  // Read temperature and humidity from DHT22
  float humidity = dht.readHumidity();
  float temperatureC = dht.readTemperature(); // Read in Celsius
  float temperatureF = dht.readTemperature(true); // Read in Fahrenheit

  // Check if any reads failed and exit early.
  if (isnan(humidity) || isnan(temperatureC) || isnan(temperatureF)) {
    Serial.println("Failed to read from DHT sensor!");
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("DHT Read Error");
```

```
  delay(2000);
  return; // Exit the loop, try again
}

// Compute heat index in Fahrenheit (the simplified version)
float hif = computeHeatIndex(temperatureF, humidity);

// Print the values to Serial Monitor
Serial.print("Humidity: ");
Serial.print(humidity);
Serial.print(" %\t");
Serial.print("Temperature: ");
Serial.print(temperatureC);
Serial.print(" *C ");
Serial.print(temperatureF);
Serial.print(" *F\t");
Serial.print("Heat Index: ");
Serial.print(hif);
Serial.println(" *F");

// Display the values on the LCD
lcd.clear();
lcd.setCursor(0, 0);
lcd.print("Temp: ");
lcd.print(temperatureC);
```

```
lcd.print("C");


lcd.setCursor(0, 1);
lcd.print("Hum: ");
lcd.print(humidity);
lcd.print("%");
```

Driver / Module	Hardware Controlled	Driver Functions / Usage
DHT22 Driver	DHT22 Temperature & Humidity Sensor	dht.begin(), dht.readTemperature(), dht.readHumidity()
LCD I2C Driver	16x2 I2C LCD Display	lcd.init(), lcd.backlight(), lcd.clear(), lcd.setCursor(), lcd.print()
Serial Driver	USB/UART Serial Interface	Serial.begin(), Serial.print(), Serial.println()
Computation Module	None (Software only)	computeHeatIndex()

Given a specific sensor (eg, temperature sensor) connected to a micro controller via SPI .Describe the steps you would take to write a basic driver to read data from the sensor

Answer: □

1.Study sensor datasheet → know registers & SPI settings

2. Configure SPI peripheral & pins

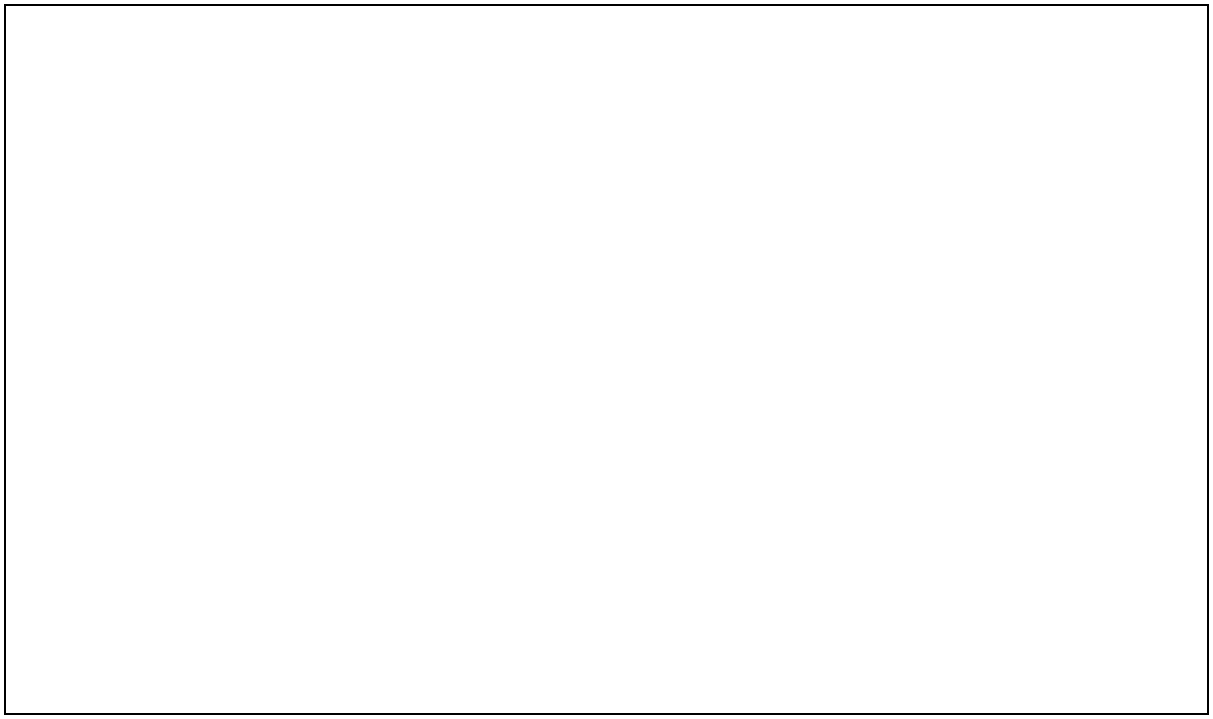
3. Implement low-level SPI read/write functions

4. Initialize sensor via SPI

5. Read raw data → process if needed

6. Provide high-level driver functions for firmware

7. Add error handling and test



3.7.4 Error Handling in driver source code

Error handling is the process of implementing mechanisms to detect, report, and recover from unexpected or erroneous conditions that may occur during the execution of embedded software.

3.7.4.1 Types of Errors in Firmware Development

-Hardware Errors: Malfunctions in physical components (chips, wiring, sensors).

-Software Errors: Bugs or incorrect logic in the firmware code or misuse of hardware abstractions.

-Data Errors: Incorrect, corrupted, or incomplete data processed by the system.

3.7.4.2 Key Elements of Error Handling

-Error Detection: Identify errors, anomalies, or exceptional conditions during operation.

-Error Reporting: Communicate error information to aid debugging and troubleshooting.

-Error Handling Strategies: Respond appropriately based on severity and impact.

3.7.4.3 **Common Error Handling Strategies**

-Detection: Use checksums, parity bits, or range checks.

-Reporting: Log errors with codes, timestamps, and relevant data.

-Correction: Attempt automatic fixes, e.g., retry communication or recalculate data.

-Graceful Recovery: Enter a safe state when errors are unrecoverable to prevent damage or malfunction.

3.8 Memory mapping : Memory mapping is the technique of **assigning specific memory addresses to hardware components or peripheral registers** in a microcontroller or embedded system.

3.8.1 Memory mapping methodology

Method	Description
Mapping Memory Directly	Access hardware registers using their fixed memory addresses directly.
Mapping Memory with Pointers	Use pointers to reference specific memory addresses for flexible access.
Mapping Memory with Structures	Define a struct that represents peripheral registers, allowing organized and readable access to hardware.

3.8.2 Advantages vs Disadvantages of memory mapping methodologies

Memory Mapping Method	Advantages	Disadvantages
Direct Memory Mapping	- Simple and straightforward to use- Fast access to hardware registers	- Hard to read/maintain in large systems- Less flexible for complex hardware
Memory Mapping with Pointers	- Flexible access to different addresses- Can handle dynamic hardware access	- Slightly slower than direct mapping- Pointer misuse can cause errors (crashes)
Memory Mapping with Structures	- Organized and readable code- Easier maintenance- Groups registers logically	- Slightly more overhead than direct mapping- Less efficient for extremely time-critical tasks

Learning outcome4: Deploy Firmware

4.1 Preparation of deployment environment

Firmware deployment refers to the process of transferring the firmware code onto the target device for execution.

4.1.1 Selection of deployment environment

Ways to deploy firmware into target device(embedded hardware)
-Virtual
-Physical

Deployment Method	Type	Description	Examples / Tools	Advantages	Limitations
Virtual Deployment	Software-based (no actual hardware used)	Firmware is tested or executed in a simulated or emulated environment instead of on the real device.	QEMU, Renode, Proteus, Simulink, virtual device labs, containerized CI	Fast, safe, no hardware required, good for early testing	Not accurate for timing, power, or hardware behavior
Physical Deployment	Hardware-based (on actual target)	Firmware is flashed and executed directly on the real embedded hardware.	JTAG/SWD, UART bootloader, USB DFU, OTA update, SD card boot, production programmer	Real-world accuracy, validates full system	Slower, risk of bricking, needs hardware setup

4.1..1.1 Virtual deployment tool

Method / Tool	Type	Description / Purpose	Examples / Tools	Advantages	Limitations	Use Cases / Target
Virtual Deployment (General)	Software-based	Firmware is tested or executed in a simulated or emulated	QEMU, Renode, Proteus, Simulink,	Fast testing, no hardware cost, no risk	Not accurate for timing, analog or real power	Early development, logic or algorithm testing

		environment (no hardware needed).	Docker, CI pipelines	of bricking	behavior	
Emulator / Simulator	Virtual Environment	Simulates MCU or full system behavior.	QEMU, Renode	Validates logic before hardware exists	Limited real-time accuracy	Development stage testing
Software-in-the-Loop (SIL) / Firmware-in-the-Loop (FIL)	Simulation Testing	Firmware runs inside simulation connected to model environment.	MATLAB Simulink, Vector tools	Detects logic and integration errors early	Complex setup	Control systems, automotive ECUs
Mocked HAL Testing	Host-based Simulation	Hardware drivers replaced with software mocks.	Custom mocks, unit test frameworks	Enables CI/unit testing	No real peripheral response	Continuous integration pipelines
Containerized / Cloud Simulation	Virtualized CI testing	Firmware logic runs automatically in container/ cloud test systems.	Docker, GitHub Actions	Fully automated test setup	Cannot emulate hardware timing	Regression & automation testing

4.1.1.2 Physical deployment tool

Method / Tool	Type	Description / Purpose	Examples / Tools	Advantages	Limitations	Use Cases / Target
Physical Deployment (General)	Hardware-based	Firmware is flashed or updated on the actual target hardware.	JTAG, SWD, UART bootloader, SPI/I²C programming, OTA	Real hardware testing, timing accuracy	Slower, hardware required, risk of bricking	Final testing, production, field update
JTAG / SWD	Hardware Interface	Debug and flash firmware	OpenOCD, ST-	Low-level	Needs programmer	Development,

		via debug port.	Link, J-Link	access, reliable	and physical access	debugging
UART / Serial Bootloader	Wired Programming	Updates firmware over serial port (RS232, USB-UART).	avrdude, esptool.py	Simple, low-cost	Slower transfer	Small MCUs, prototypes
USB DFU (Device Firmware Upgrade)	Update Protocol	USB standard to upload firmware without external programmer.	dfu-util, STM32 DFU, Atmel DFU	Easy for users, plug-and-play	Needs DFU bootloader	Embedded devices, IoT
SPI / I²C / ISP Programming	Wired Interface	Programs flash memory directly on board.	Vendor-specific ISP tools	Works for production programming	Requires fixture	Factory programming
SD Card / eMMC Boot	Removable Media	Firmware loaded from removable storage.	Bootloader setup	Simple manual updates	Manual insertion	IoT, development boards
OTA (Over-The-Air)	Wireless Update	Wireless update via WiFi, BT, LTE, etc.	ESP-IDF OTA, AWS IoT OTA	Remote updates, scalable	Needs network, secure design	IoT, remote devices
Ethernet / CAN Bootloader	Network-based Update	Updates sent via wired communication bus.	CANopen, TFTP bootloaders	Ideal for multi-device systems	Complex setup	Automotive, industrial systems
Production Gang Programmer	Manufacturing	Programs many devices simultaneously in production line.	Elnec, Dataman	Efficient mass flashing	High setup cost	Factory use
DFU (Device Firmwar	Protocol / Tool	USB-based method for updating	STM32 DFU, Atmel	Simple updates, no	Device must suppor	Embedded MCUs

e Upgrade)		firmware on actual device.	DFU, dfu-util	special programmer	t DFU mode	
Intel Firmware Update Tool (IFUT)	Vendor Tool	Updates Intel BIOS/ME firmware.	IFUT	Reliable and verified	Intel-only	Intel PCs, servers
AMI Aptio Setup & Configuration Tool	Vendor Tool	Updates/configures AMI BIOS/UEFI firmware.	AMI Aptio	OEM-level BIOS control	Proprietary	PCs, servers
Phoenix Technologies Armada	Vendor Tool	BIOS/UEFI firmware development and update platform.	Armada	BIOS customization	OEM use only	Embedded/PC systems
Insyde Software H2O (Insyde H2O)	Vendor Tool	BIOS/UEFI firmware management software.	InsydeH2O	Common in laptops	Proprietary	Laptops, industrial PCs
Dell OpenManage Essentials (OME)	Enterprise Tool	Central firmware/BIOS management for Dell hardware.	OME	Remote updates, automation	Dell-only	Enterprise servers, workstations
HP Image Assistant (HPIA)	Enterprise Tool	Analyzes and updates HP BIOS and firmware.	HPIA	Simplifies firmware maintenance	HP-only	HP laptops, PCs
Lenovo Update Xpress	Enterprise Tool	Automates firmware and driver deployment.	Update Xpress	Centralized control	Lenovo-only	Lenovo PCs, servers
IBM Client Management Solutio	Enterprise Tool	Manages firmware, BIOS, and driver updates for	CMS	Centralized, secure	IBM-only	IBM servers, enterprise clients

ns (CMS)		IBM systems.				

4.1.1.3 Hybrid deployment tool

Method / Tool	Type	Description / Purpose	Examples / Tools	Advantages	Limitations	Use Cases / Target
Hardware-in-the-Loop (HIL)	Virtual + Physical	Combines real hardware with simulation models for full system testing.	dSPACE, NI PXI, MATLAB/Simulink	Realistic testing with partial virtualization	Complex setup	Automotive, robotics, industrial control

4.1.1.4 Integration Testing on Temperature Monitoring System steps

1. **Identify Modules**:Determine all the modules/components to integrate:

-Sensor Module – reads temperature/humidity

-Data Processing Module – filters or calculates average

-Communication Module – sends data to display/server

-Display / User Interface – shows readings

-Alert / Actuator Module – triggers alarms or fans

2. **Define Test Objectives :**Specify what you want to verify:

-Correct data flow between modules

-Interaction between hardware and software

-System response to threshold events (e.g., high temperature alarm)

-Communication reliability (sending data to remote server/display)

3. **Develop Test Cases :**Create test cases covering:

-Normal temperature readings

-Boundary conditions (max/min thresholds)

-Faulty inputs (sensor disconnection, out-of-range readings)

- Communication failures

4. **Prepare Test Environment (2 Marks)**
   - Ensure necessary **hardware** (microcontroller, sensor, display, actuators)
   - Install required **software/drivers**
   - Configure **interfaces**, network, and thresholds
5. **Execute and Analyze (2 Marks)**
   - Run all test cases systematically
   - Record results, anomalies, and pass/fail outcomes
   - Analyze data for module interaction errors or incorrect system behavior
   - Document findings for corrective action

4.1.1.5 Firmware deployment environment comparisons

Criteria	Virtual Deployment Environment	Physical Deployment Environment
Execution Speed	Slower.	High speed of running.
Memory Usage	Flexible — can allocate more/less memory for testing. -Not always accurate to real constraints.	Exact memory usage as in the final device — allows detection of memory overflows or inefficiencies.
Power Consumption	Not measurable accurately — power use is simulated or ignored.	Real power consumption can be measured and optimized.
Maintainability	-High -easier to update, debug, and test without risking hardware.	-Lower -requires physical flashing, risk of wear or bricking hardware during updates.
Reliability Testing	Limited — can't test real sensors, timing, or analog behavior.	Full reliability validation with real-world conditions.
Cost	Low — only requires a PC and simulation software.	Higher — requires target hardware, tools, and programmers.

4.1.1.6 Role of firmware testing

- **Error Detection**: Firmware testing helps identify bugs, glitches, or logical errors that could cause the system to fail or behave unexpectedly.

- **Safety Assurance**: In critical systems (e.g., medical devices, automotive controls), firmware testing ensures that the device operates safely under all conditions, reducing the risk of harm or malfunctions.

- **Performance Validation**: Tests confirm that the firmware meets performance requirements, such as speed, response time, and power efficiency, ensuring that the system runs optimally.

-**Hardware Compatibility**: Testing ensures that the firmware interacts correctly with the hardware components, avoiding issues like memory corruption, communication failure, or hardware malfunctions.

-**Security**: Testing checks for vulnerabilities in the firmware that could be exploited by malicious attacks.

- **Compliance**: Many industries have regulations that embedded systems must meet. Testing ensures compliance with these standards, avoiding legal or safety issues.

4.1.1.7 STLC (Software testing life cycle) steps

The STLC for firmware testing is a structured process that ensures the firmware's quality, reliability, and robustness(**strength and stability** of a system) during all stages of its development lifecycle.

-**Requirement Analysis:** Identify and define testable firmware requirements.
-**Test Planning:** Plan objectives, scope, environment, and schedule.

-**Test Design:** Create test cases and scenarios for all conditions.

- **Test Environment Setup:** Prepare hardware, tools, and configurations.

- **Test Execution:** Run tests, record results, and perform regression testing.

- **Defect Reporting:** Log and track defects found during testing.

- **Test Closure:** Summarize results and lessons learned.

- **Release Readiness:** Confirm firmware meets release standards.

- **Post-Release Support:** Monitor, update, and fix issues after release.

4.1.1.7.1 Advantages vs Disadvantages of SDLC

Advantage	Description
Structured Approach	Provides a clear, step-by-step framework for software development.
Improved Project Management	Defines phases, timelines, and deliverables for better planning and control.

High Quality Output	Each phase includes review and testing to ensure software quality.
Cost and Time Control	Early planning and documentation help reduce rework and unexpected costs.
Better Communication	Encourages coordination between developers, testers, and clients.
Risk Management	Identifies and mitigates risks early in the development process.
Customer Satisfaction	Ensures requirements are understood and met systematically.

Disadvantage	Description
Time-Consuming	Detailed documentation and reviews can slow down development.
Less Flexibility	Hard to make changes once the process has started, especially in traditional models (like Waterfall).
High Initial Planning Effort	Requires a lot of effort and resources before coding begins.
Complex for Small Projects	May be too heavy or formal for small or simple software systems.
Late Testing Feedback	In some models, testing happens late, delaying defect discovery.
Dependence on Requirement Accuracy	If requirements are unclear, the whole process can fail.

4.2 Implementation of testing

4.2.1 Unit Testing in Embedded Systems Steps

1. **Choose a Unit Testing Framework:** Select a testing framework compatible with the embedded platform (e.g., Unity, CppUTest).
2. **Organize Code into Units:** Divide firmware into isolated modules or functions for testing.
3. **Write Unit Test Cases:** Create tests covering normal, edge, and error cases with assertions.
4. **Mock Dependencies:** Simulate hardware or external modules to test units independently.
5. **Compile and Link Tests:** Integrate and compile tests with firmware code.
6. **Execute Unit Tests:** Run tests on host or target hardware and collect results.
7. **Analyze Test Results:** Identify failures and debug issues in firmware.
8. **Iterate and Refactor:** Improve code based on test feedback for maintainability and correctness.
9. **Automate Test Execution:** Integrate unit tests into CI pipelines for automatic validation.

10. **Maintain Test Coverage:** Continuously update tests to cover new features or changes.

4.2.2 Integration Testing in Embedded Systems

Integration Approaches:

- **Top-Down:** Test high-level modules first, gradually integrating lower-level modules using stubs.
- **Bottom-Up:** Test low-level modules first, then integrate progressively into higher-level modules.

Integration Test Steps:

1. **Identify Integration Points:** Determine interfaces between software and hardware components.
2. **Develop Test Cases:** Create tests for data exchange, control signals, and error handling.
3. **Prepare Test Environment:** Set up hardware, peripherals, and tools required for testing.
4. **Instrumentation and Logging:** Collect data and logs during test execution for analysis.
5. **Execute Integration Tests:** Run tests simulating real-world scenarios and interactions.
6. **Data Analysis and Verification:** Compare observed behavior with expected results.
7. **Error Handling and Debugging:** Identify and resolve communication, synchronization, or functional issues.
8. **Iterative Testing and Refinement:** Refine tests and re-run after fixes or changes.
9. **Documentation and Reporting:** Record procedures, results, and observations for stakeholders.
10. **Regression Testing:** Re-run tests after changes to ensure no new integration issues arise.

Note: low level module read sensor data (Example :code read data from dht 11)

High level module :control the threshold level ,control data,update and send alert message

4.3 Export the firmware image

4.3.1 Definition

Exporting a firmware image is the process of **generating a complete, deployable binary file** from the compiled firmware code that can be flashed or uploaded to the target embedded hardware.

4.3.2 Description on .hex file

-A **.hex file** is a human-readable file format that stores the compiled firmware code for microcontrollers.
- It represents binary data in a standardized way, specifies memory addresses for storage, and ensures compatibility across different microcontroller architectures.

-It serves as the firmware image used for programming devices.

The role of hex file for firmware image

Role	Description
Representation	Converts binary firmware data into a human-readable format.
Standardization	Provides a standard format for programming microcontrollers.
Memory Addressing	Specifies exact memory locations to store the firmware.
Compatibility	Works across different microcontroller architectures.

4.3.3 List of files can be flushed int the micro controller

File Format	Description / Use	Advantages	Disadvantages	Flashing Tool / Method
.hex	Intel HEX format, human-readable, widely used for microcontrollers.	Human-readable, easy debugging, memory addressing supported, widely standardized.	Larger file size than raw binary, slower to flash.	AVRDUDE (AVR), ST-LINK Utility (STM32), MPLAB IPE (PIC), **Arduino IDE (Arduino boards)**
.bin	Raw binary format, contains pure compiled firmware data.	Small size, fast to flash, simple format.	No metadata or memory addresses, less human-readable, risk of flashing to wrong location.	STM32CubeProgrammer (STM32), Flash Magic (NXP), esptool.py (ESP), **Arduino IDE (Arduino boards)**
.elf	Executable and Linkable Format, used	Contains debug info, symbols, and	Not suitable for direct flashing; larger size.	Converted to .hex/.bin first, then flashed using platform tools

	for debugging and linking.	sections; useful for developme nt.		
.srec	Motorola S-record format, similar to HEX, for embedded programming.	Human-readable, memory address specified, supports checksum.	Larger than binary, less common than .hex.	Flash Magic (NXP), Segger J-Flash
.dfu	Device Firmware Update format, often for USB bootloaders.	Supports secure OTA updates, built-in verification.	Usually requires bootloader support, larger overhead.	STM32 DFU Tool, dfu-util (cross-platform)
.ihex	Variant of Intel HEX, commonly used for firmware programming.	Similar to .hex, standardize d, supports multiple platforms.	Slightly larger than binary, slower to flash.	MPLAB IPE (PIC), ST-LINK Utility (STM32), **Arduino IDE (Arduino boards)**
.axf	ARM Executable format, often used in ARM-based embedded systems.	Contains debug symbols, link info, supports developme nt and testing.	Cannot be directly flashed; must convert to .bin/.hex.	Converted via ARM tools (Keil, GCC) to .hex/.bin, then flashed

4.4.Identification of Micro controller

A microcontroller is a compact integrated circuit designed to govern a specific operation in an embedded system.

4.4.1 Different examples of microcontroller

Microcontroller / Platform	Definition	Architecture	Common Use / Notes	Advantages	Disadvantages

Atmel AVR	A family of microcontrollers developed by Atmel for embedded applications.	8-bit RISC	General-purpose microcontrollers; widely used in Arduino boards.	-Low power -Easy to program. -Large community support.	-Limited processing power, - 8-bit limitation for complex tasks.
Arduino	Open-source electronics platform combining microcontroller boards and IDE for easy programming.	Varies (AVR, SAMD, ESP32)	Prototyping, education, IoT, hobby projects.	Beginner-friendly, large library support, rapid prototyping.	Limited performance for high-speed or complex applications.
AVR	Low-power 8-bit microcontroller architecture by Atmel.	8-bit RISC	Embedded systems, sensor control, simple automation tasks.	Efficient, low-cost, reliable, simple instruction set.	Limited memory, low processing speed.
ATX Mega	High-pin-count AVR microcontrollers for larger projects.	8-bit AVR	Complex Arduino projects, robotics, large I/O applications.	Large I/O count, more memory, supports complex projects.	Still 8-bit, slower compared to 32-bit MCUs, higher cost than smaller AVRs.
ARM	32-bit RISC microcontroller architecture for high-performance embedded applications.	32-bit RISC (Cortex-M series common)	IoT devices, mobile electronics, industrial automation.	High performance, low power (Cortex-M), scalable, widely used.	More complex to program, higher cost, steeper learning curve.

8051	Classic microcontroller originally developed by Intel for embedded control applications.	8-bit CISC	Legacy embedded systems, education, simple control systems.	Simple, widely studied, low-cost, robust for basic tasks.	Obsolete architecture, limited performance, small memory.
ATmega328P	Popular microcontroller used in Arduino Uno, known for ease of use.	8-bit AVR	Hobbyist and educational projects, simple IoT and sensor interfacing.	Easy to program, low power, widely supported in Arduino ecosystem.	Limited I/O pins, low processing speed, small memory.

4.4 Creation of Hex file STEPS

1. **Write Firmware Code**

- Develop code in C, C++,Arduino IDE or assembly.
- Implement functionalities, device drivers, communication protocols, and application logic.

2. **Choose Development Environment**

- Set up IDEs, compilers, and debuggers.
- Select toolchain compatible with the target microcontroller.

3. **Compile Source Code**

- Translate high-level code into machine code (.bin).
- Apply proper compiler options, optimizations, and target architecture settings.

4. **Link Object Files**

- Combine multiple compiled files and libraries into a single executable.

5. **Generate .hex File**

- Convert the executable into a .hex file for microcontroller programming.

6. **Verify Firmware Image**

- Test the .hex file using simulators, emulators, or development boards.
- Optional: perform checksum, encryption, or compression checks.

7. **Program Microcontroller**

- Use hardware programmers, debugger probes, or bootloaders.
- Transfer the .hex file to the microcontroller's memory via interfaces like JTAG, SPI, or UART.

8. **Verify Programming**

- Confirm the firmware was successfully written and behaves correctly.

4.4.1 Selection of target device And exporting .hex file

Exporting a .hex file for a firmware image refers to the process of generating a .hex file from the compiled binary code of the firmware.

4.5 Flushing the .hex file to micro controller

1. **Prepare Hardware:** Connect the microcontroller to a programmer or debugger and power it on.

2. **Select Programming Mode:** Configure the appropriate interface or mode for flashing.

3. **Launch Programming Software:** Open the programmer's software or IDE.

4. **Load .hex File:** Select and verify the firmware .hex file to be programmed.

5. **Connect to Microcontroller:** Establish communication between software and microcontroller.

6. **Erase Memory (Optional):** Clear flash memory if required.

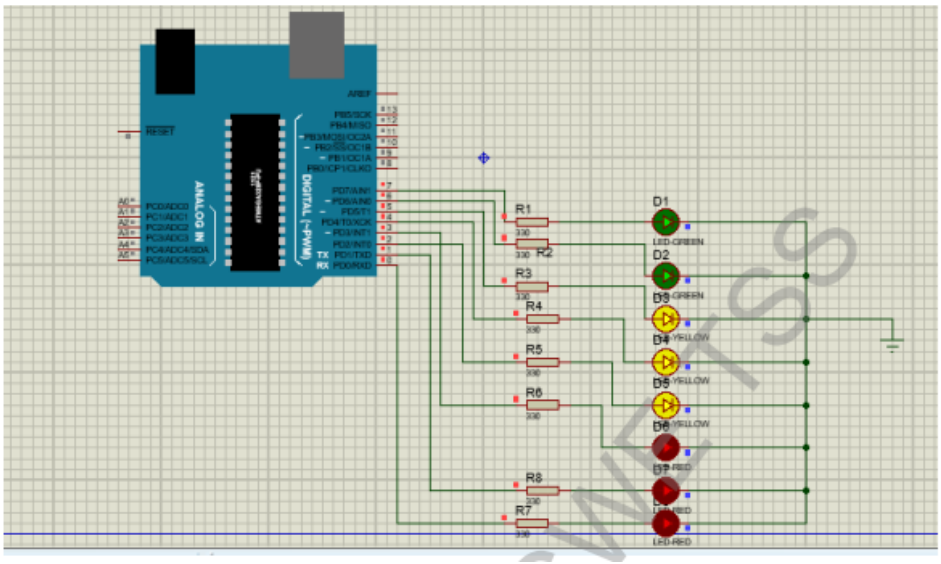
7. **Program Microcontroller:** Send the .hex file to the microcontroller to write it into flash memory.

8. **Verify Programming:** Ensure the firmware was correctly flashed.

9. **Reset/Power Cycle:** Restart the microcontroller to run the new firmware.

Example for simple project
The example for simple project

Look at the figure below and create the firmware to light the LEDs continuously after every second.



Answer

```
void setup() {
pinMode(0,OUTPUT);
pinMode(1,OUTPUT);
pinMode(2,OUTPUT);
pinMode(3,OUTPUT);
pinMode(4,OUTPUT);
pinMode(5,OUTPUT);
pinMode(6,OUTPUT);
pinMode(7,OUTPUT);
}

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

    digitalWrite(1,HIGH);
    digitalWrite(2,HIGH);
    digitalWrite(3,HIGH);
    digitalWrite(4,HIGH);
    digitalWrite(5,HIGH);
    digitalWrite(6,HIGH);
    digitalWrite(7,HIGH);
    digitalWrite(0,HIGH);
    delay(1000);
    digitalWrite(1,LOW);
    digitalWrite(2,LOW);
    digitalWrite(3,LOW);
    digitalWrite(4,LOW);
    digitalWrite(5,LOW);
    digitalWrite(6,LOW);
    digitalWrite(7,LOW);
    digitalWrite(0,LOW);
delay(1000);
  }
```

4.6 Documentation of the firmware

4.6.1 Definition

Firmware documentation is the process of creating clear and detailed records describing the firmware's design, functionality, implementation, usage, and maintenance.

4.6.2 Firmware technical documentation

- **Introduction:** Provide Overview of the firmware, its purpose, and its role in the embedded system.

- **Architecture Overview:** identify High-level system architecture, modules, layers, and communication protocols.

- **Design Considerations:** show Key design decisions including microcontroller choice, programming languages, frameworks, algorithms, and data structures.

- **Module Descriptions:** Detailed info on each module's purpose, functionality, interfaces, inputs/outputs, internal logic, and hardware interactions.

- **API Documentation:** Description of API functions/methods, parameters, return values, usage examples, and versioning.

- **Configuration & Customization:** Firmware settings, hardware configurations, feature toggles, and user-configurable options.

- **Usage Guide:** Instructions for installation, setup, initialization, and performing common operations.

- **Testing & Validation:** Test plans, test cases, tools, environments, results, and coverage analysis.

- **Troubleshooting Guide:** Error messages, debugging techniques, common issues, and workarounds.

- **Maintenance & Updates:** Version control, release management, applying patches, and firmware updates.

- **References & Resources:** Links to datasheets, specifications, standards, libraries, and community resources.

4.6.3 Firmware documentation tools

Tool	Description / Use
Doxygen	Generates documentation from source code comments for C, C++, Java, Python; outputs HTML, PDF, LaTeX.
Sphinx	Python-focused tool using reStructuredText; outputs HTML, PDF, ePub, plain text.

Javadoc	Java-specific tool; generates HTML documentation from formatted comments.
Markdown	Lightweight markup language; can create docs alongside code; tools like MkDocs or GitBook generate HTML.
Confluence	Wiki-style collaboration platform; supports rich text, multimedia, and integration with Jira/Bitbucket.
GitBook	Open-source platform for collaborative documentation websites; supports Markdown and version control.
Wiki (MediaWiki, DokuWiki)	Collaborative platform for creating, editing, and linking documentation pages.
Microsoft Word	General-purpose document authoring; widely used for formatted documentation.
Adobe FrameMaker	Professional authoring tool for structured technical manuals and complex documentation.
LaTeX	Typesetting system for precise control over technical/scientific document layout; generates high-quality PDFs.

4.6.4 Installation of documentation tool

Steps by step how to install documenting tool or application program
Select a Documentation Tool: Choose a documentation tool that best suits your needs.

Check System Requirements: Before installation, ensure that your computer system meets the requirements specified by the documentation tool.

Download or Install the Tool: Depending on the documentation tool chosen, follow the installation instructions provided by the tool's official website or documentation.

Configure the Tool (if necessary): Some documentation tools may require configuration settings to tailor them to your specific project or preferences.

Verify Installation: After installation, verify that the documentation tool is properly installed and functioning correctly.

Integrate with Development Workflow: Integrate the documentation tool into your firmware development workflow to ensure documentation is created and updated consistently.

Provide Training and Support: Offer training and support to team members on how to use the documentation tool effectively.

4.6.5 Documentation of project setup

4.6.5.1 Documenting enum and struct

- **A struct** is a composite data type that groups different types of data (variables) together under a single name, allowing for better data organization.

-**An enum** (short for enumeration) is a user-defined data type that consists of a fixed set of named constants.

Role of enum

- improve code readability,

-Improve code maintainability, and

-Enforce constraints on variable values.

Both enums (enumerations) and structs (structures) are essential tools for organizing and managing data and functionality within the codebase.

Documenting enums and structs in firmware development is crucial for ensuring code readability, maintainability, and ease of understanding for other developers who may work on the codebase in the future.

4.6.5.2 Documenting functions

Documenting functions in firmware development is crucial for ensuring that other developers (including your future self) can understand the purpose, usage, parameters, and return values of each function.

4.6.5.3 Documenting modules

Documenting modules in firmware development is essential for facilitating understanding, maintenance, and collaboration among developers.

4.6.5.4 Creation of reusable template

4.7 Firmware user Manual

4.7.1 Definition

A firmware user manual in embedded systems is a document designed to provide end-users with information on how to operate and interact with the embedded device or system that incorporates firmware

4.7.2: Elements of firmware user manual

-Introduction: Overview of the embedded system and the firmware's role.

-Getting Started: Device setup instructions, hardware connections, and component overview.

-Installation and Setup: Guide for installing drivers/software and configuring settings.

-Operation Guide: Instructions for using the device, interfaces, controls, and workflows.

-Feature Overview: Description of device functions, modes, and practical use cases.

-Troubleshooting: Tips for diagnosing issues, FAQs, and support resources.

-Safety and Regulatory Information: Precautions, warnings, and compliance with standards.

-Maintenance and Care: Guidelines for device upkeep, firmware updates, and storage.

-Appendices: Additional references, diagrams, technical specifications, and warranty info.

-Index and Table of Contents: Helps users quickly find topics within the manual.

Difference between firmware user manual and firmware documentation

here's a comparison table highlighting the differences between firmware user manual and firmware documentation:

Aspect	Firmware User Manual	Firmware Documentation
Purpose	Provides guidance for end-users on using the firmware and its features	Offers detailed technical information for developers, engineers, or system integrators
Target Audience	End-users, consumers, or non-technical individuals	Developers, engineers, system integrators, or technical personnel
Scope	Focuses on usage instructions, setup procedures, troubleshooting tips, and FAQs(**Frequently Asked Questions**)	Covers technical specifications, API references, architecture, implementation details, and release notes
Content	Step-by-step instructions, diagrams, illustrations, and usage scenarios	Code snippets, configuration settings, hardware requirements, software dependencies, and compatibility information

Aspect	Firmware User Manual	Firmware Documentation
Language	Typically written in non-technical, user-friendly language	Contains technical terminology, jargon, and industry-specific terms
Format	Often presented in a structured format with chapters, sections, and headings	Can be in the form of text documents, API documentation, wiki pages, or README files
Updates	Updated less frequently, mainly with major firmware releases	Updated regularly to reflect changes, bug fixes, and enhancements in the firmware
Accessibility	Emphasizes ease of understanding for non-technical users	Requires some level of technical proficiency to interpret and utilize effectively
Examples	Quick start guides, FAQs, and troubleshooting guides	API reference manuals, architecture diagrams, and technical specifications

Here is a table that outlines the tasks performed by both firmware developers and firmware users in the context of embedded system development:

Task	Firmware Developer	Firmware User
1. Requirement Analysis	Analyze system requirements and define firmware specifications.	Review firmware features and verify that they meet system needs.
2. Firmware Design	Design system architecture, flowcharts, and state machines.	Provide input on feature implementation based on usage needs.
3. Writing Firmware Code	Write code in low-level programming languages (C, C++, Assembly).	Test firmware against hardware, providing feedback.
4. Hardware Interface Implementation	Develop drivers and modules to interface with hardware peripherals.	Use hardware controlled by the firmware to perform tasks.
5. Memory Management	Manage RAM, ROM, and flash memory allocations in embedded systems.	Optimize usage of memory resources based on the firmware's behavior.

6. Testing and Debugging	Perform unit tests, integration tests, and hardware-in-loop testing.	Perform functional testing of firmware and report any issues.
7. Bootloader Development	Develop a bootloader for initializing the system and updating firmware.	Use bootloader functionality to update system firmware.
8. Power Management Implementation	Implement energy-saving algorithms, such as sleep modes or low-power states.	Observe system behavior under different power conditions.
9. Communication Protocol Development	Implement communication protocols (I2C, SPI, UART, etc.).	Use firmware to control communication between devices.
10. Interrupt Handling	Program interrupt service routines (ISR) to handle hardware interrupts.	Trigger interrupts and ensure proper system response.
11. Performance Optimization	Optimize firmware for low latency, power efficiency, and processing speed.	Provide feedback on performance and usability improvements.
12. Firmware Updates	Develop mechanisms for Over-the-Air (OTA) or manual firmware updates.	Receive and install firmware updates on the system.
13. Integration with RTOS	Integrate the firmware with a real-time operating system (RTOS).	Operate the system according to the scheduling and task prioritization set by the RTOS.
14. Security Implementation	Implement secure boot, encryption, and authentication mechanisms.	Ensure firmware security features are properly utilized.
15. Documentation	Create detailed documentation of firmware design, implementation, and usage.	Read documentation to understand how to use the firmware.
16. Deployment	Deploy the firmware to embedded devices.	Install the firmware and verify proper system operation.

Appendix

Integrated/Summative assessment

Integrated situation

ABC Electronics is a growing startup that specializes in creating home automation products. They are currently working on a new project to develop an environmental monitoring system that will be used to monitor indoor temperature and humidity. The company is facing a problem that they have the circuit design of this system but there is no embedded software to manage the functionality of this system hardware.

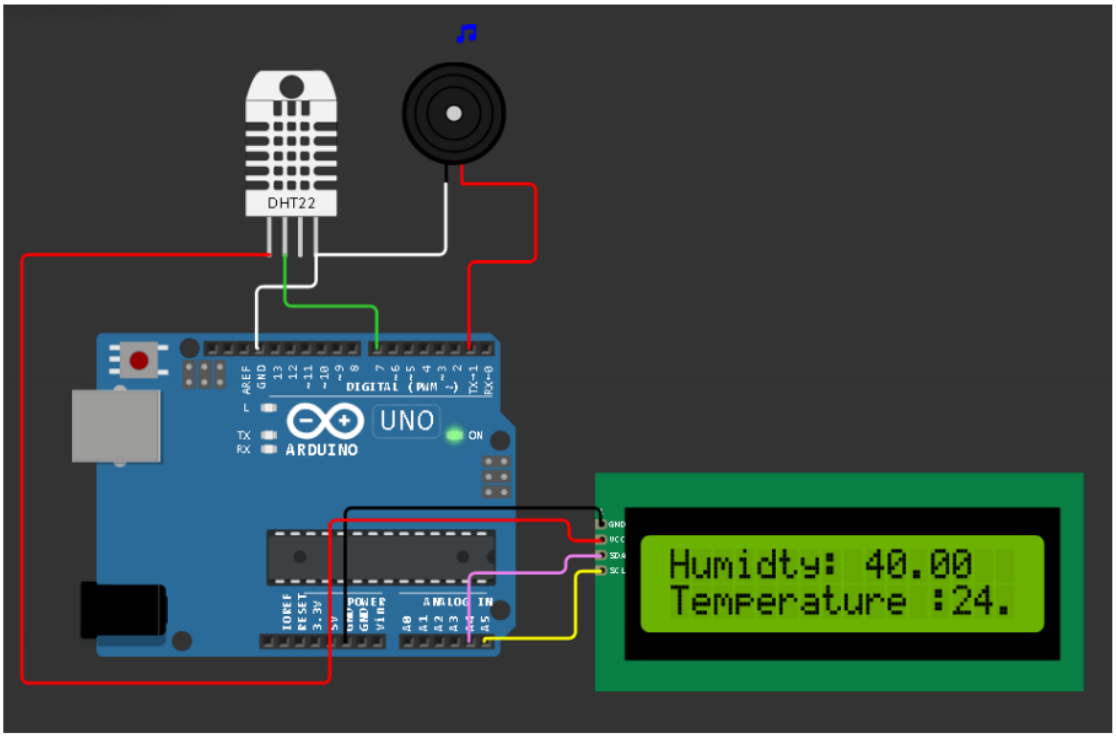
As a firmware developer, you are hired by ABC Electronics to develop a software that will solve this problem by being integrated into this system hardware to monitor the temperature and humidity levels in the clients' living spaces. The device will display real-time temperature and humidity data on the LCD screen and provide sound alerts when certain thresholds are crossed. The system will display temperature (°C) and humidity (%) data on the LCD screen in a user-friendly format. When the temperature and humidity rates out of normal conditions, the firmware should trigger audio alerts in the Buzzer and display warning messages.

Incorporate error handling routines to detect and manage sensor failures or communication issues effectively. Display appropriate error messages on the LCD to notify users of any problems.

After the work, prepare clear and concise documentation explaining how to use the device, interpret the data displayed, and generate the firmware image to be flashed in the system hardware.

Note: The normal or comfortable room temperature in Celsius degrees is generally considered to be between 20°C to 25°C (68°F to 77°F) and the recommended indoor relative humidity levels for comfort and health usually range between 40% to 60%.

The circuit diagram of the system is provided below:



The work must be performed within 6 hours (6).
All equipment, tools and materials are provided.Resources

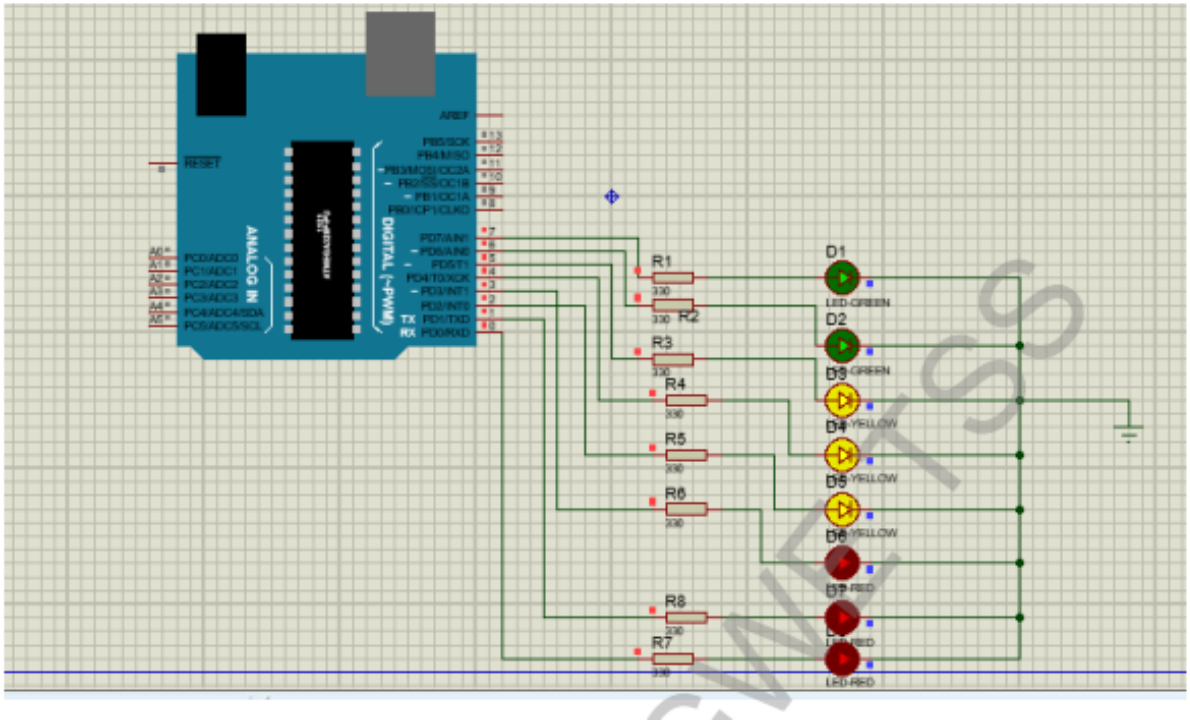
The questions of the module after integrated assessment

Question1)

a) Define “a microcontroller”.

b) List any four (4) examples of microcontroller.

c) Look at the figure below and create the firmware to light the LEDs continuously after every second.



Question2)

Firmware architecture is a blueprint that outlines the structure and organization of firmware within an embedded system.

a.)List the four (4) key diagrams used in firmware architecture.

b) Identify any four (4) hardware components needed in an automatic home temperature control system

c) Sketch the architecture design of automatic home temperature control system using Arduino.

Question3)

Design a system that measures the distance to objects in front of a sensor. Specify the components and the method you would use to measure and display the distance. write the necessary code to capture the sensor data, process it, and output the distance measurement, including the programming logic and interfaces used.

ANSWER:

Components:

1. **Arduino Uno (or any compatible board)** – The microcontroller that will control the sensor and process the data.
2. **Ultrasonic Sensor (HC-SR04)** – This sensor will measure the

distance to an object by emitting ultrasonic pulses and measuring the time it takes for the pulse to reflect back.
3. **Jumper Wires** – For connecting the sensor to the Arduino.
4. **Breadboard** (optional) – For making connections more organized.
5. **LED or Display (optional)** – For outputting the distance in a visual format (can be a 16x2 LCD or an LED for visual indication, but we will focus on the Serial Monitor for simplicity).

Code

```
// Define the pins for the HC-SR04 Ultrasonic Sensor
const int trig Pin = 9; // Trigger Pin connected to Arduino pin 9
const int echo Pin = 10; // Echo Pin connected to Arduino pin 10
// Variables to store the duration and calculated distance
long duration; // Duration for pulse
int distance; // Distance measured
void setup () {
// Start the Serial Monitor at a baud rate of 9600
Serial. Begin (9600); // Initialize serial communication
// Set the trigger pin as output and echo pin as input
pinMode (trig Pin, OUTPUT); // Set trig Pin as output
pinMode (echo Pin, INPUT); // Set echo Pin as input
}
void loop () {
// Send a 10-microsecond pulse to trigger the ultrasonic sensor
digital Write(trig Pin, LOW); // Ensure the trigger pin is low
delay Microseconds (2); // Wait for 2 microseconds
digital Write (trig Pin, HIGH); // Send the pulse to trigger
delay Microseconds (10); // Keep the trigger pin high for 10
microseconds
digital Write (trig Pin, LOW); // Stop sending pulse
// Read the duration of the pulse from the Echo pin
duration = pulse In(echo Pin, HIGH); // Measure pulse duration
// Calculate the distance (duration * speed of sound
// The speed of sound is approximately 343 meters per second or
0.0344 cm/µs
distance = duration * 0.0344 / 2 // Distance in centimeters
// Print the distance to the Serial Monitor
Serial. Print ("Distance: "); // Print the text (1 mark)
Serial. Print(distance); // Print the distance in centimeters
Serial.println(" cm"); // Print the unit (1 mark)
// Optional: Add a delay for stable readings (e.g., 1 second)
delay (500); // 500ms delay (0.5 seconds)
}
```

Question 4) Design a system that detects motion in front of a sensor. Specify the components and the method you would use to detect and display the

motion. Write the necessary code to capture the sensor data, process it, and output the motion detection status, including the programming logic and interfaces used.

Solution

1.Components for Motion Detection Project

-Arduino Uno – Microcontroller to process sensor data.

-PIR Motion Sensor (HC-SR501) – Detects motion by sensing infrared radiation changes.

-LED – Visual indicator for motion detection.

-220Ω Resistor – Current-limiting resistor for the LED.

-Breadboard – For easy circuit assembly.

-Jumper Wires – Connect components on the breadboard.

-USB Cable – To program the Arduino from a computer.

2. **Method / Approach**

-Sensor Placement: Place PIR sensor in front of the area to monitor for motion.

-Signal Reading: Read the digital output from the PIR sensor. HIGH = motion detected, LOW = no motion.

-Output / Display:

- Turn on LED if motion is detected.
- Print message to Serial Monitor.

-Programming Logic:

- Continuously monitor sensor input.
- Use conditional statements (if/else) to determine sensor state.
- Update outputs (LED & Serial) accordingly.

3.Code

```
// Motion Detection System using PIR Sensor


int pirPin = 2;      // PIR sensor output connected to digital pin 2

int ledPin = 13;     // LED connected to digital pin 13
```

```
int pirState = LOW;  // Variable to store PIR sensor status
int val = 0;         // Variable to read PIR sensor value

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(pirPin, INPUT);
  Serial.begin(9600);
  Serial.println("PIR Motion Sensor Initialized");
}

void loop() {
  val = digitalRead(pirPin); // Read PIR sensor

  if (val == HIGH) { // Motion detected
    digitalWrite(ledPin, HIGH);  // Turn LED ON
    if (pirState == LOW) {
      Serial.println("Motion Detected!");
      pirState = HIGH;
    }
  } else {
    digitalWrite(ledPin, LOW); // Turn LED OFF
    if (pirState == HIGH) {
      Serial.println("No Motion");
      pirState = LOW;
    }
  }
  delay(100); // Short delay to avoid spamming serial output
}
```

Question4) Design an embedded system that measures temperature and humidity using a sensor, displays the readings on an LCD, and triggers an LED alert if temperature or humidity exceeds predefined thresholds. Specify the components and write the Arduino code to capture sensor data, process it, and control the outputs.

Solution

Components:

-Arduino Uno – Microcontroller.

-DHT11 or DHT22 Sensor – Measures temperature and humidity.

-16x2 LCD Display (with I2C module optional) – Displays temperature and humidity readings.

-LED – Indicator for threshold alert.

-220Ω Resistor – Current limiting for LED.

-Breadboard – For assembling the circuit.

-Jumper Wires – For connections.

-USB Cable – Programming and power.

System Design / Logic

1. Read temperature and humidity from the DHT sensor.
2. Display the readings on the LCD.
3. Check if temperature > 30°C or humidity > 70% (thresholds).
    - If yes → turn on LED as an alert.
    - Else → keep LED off.
4. Repeat readings at 2-second intervals.

Arduino Code Example

```
#include <DHT.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

// Define DHT sensor
#define DHTPIN 2
#define DHTTYPE DHT11  // Use DHT22 if you have that

DHT dht(DHTPIN, DHTTYPE);
```

```
// Define LCD
LiquidCrystal_I2C lcd(0x27, 16, 2); // I2C address 0x27, 16x2 LCD

// LED pin
int ledPin = 13;

// Thresholds
float tempThreshold = 30.0; // Celsius
float humidityThreshold = 70.0; // Percent

void setup() {
  pinMode(ledPin, OUTPUT);
  dht.begin();
  lcd.init();
  lcd.backlight();
  Serial.begin(9600);
}

void loop() {
  // Read temperature and humidity
  float temp = dht.readTemperature();
  float humidity = dht.readHumidity();

  // Check if readings are valid
  if (isnan(temp) || isnan(humidity)) {
    Serial.println("Failed to read from DHT sensor!");
    lcd.setCursor(0,0);
    lcd.print("Sensor Error");
    delay(2000);
    return;
  }

  // Display on LCD
  lcd.setCursor(0,0);
  lcd.print("Temp: ");
  lcd.print(temp);
  lcd.print(" C");

  lcd.setCursor(0,1);
  lcd.print("Humidity: ");
  lcd.print(humidity);
  lcd.print(" %");

  // Serial output
  Serial.print("Temp: "); Serial.print(temp); Serial.print(" C, ");
  Serial.print("Humidity: "); Serial.print(humidity); Serial.println(" %");

  // Check thresholds
```

```
  if (temp > tempThreshold || humidity > humidityThreshold) {
    digitalWrite(ledPin, HIGH); // Turn LED on
  } else {
    digitalWrite(ledPin, LOW);  // Turn LED off
  }

  delay(2000); // Wait 2 seconds before next reading
}
```

END