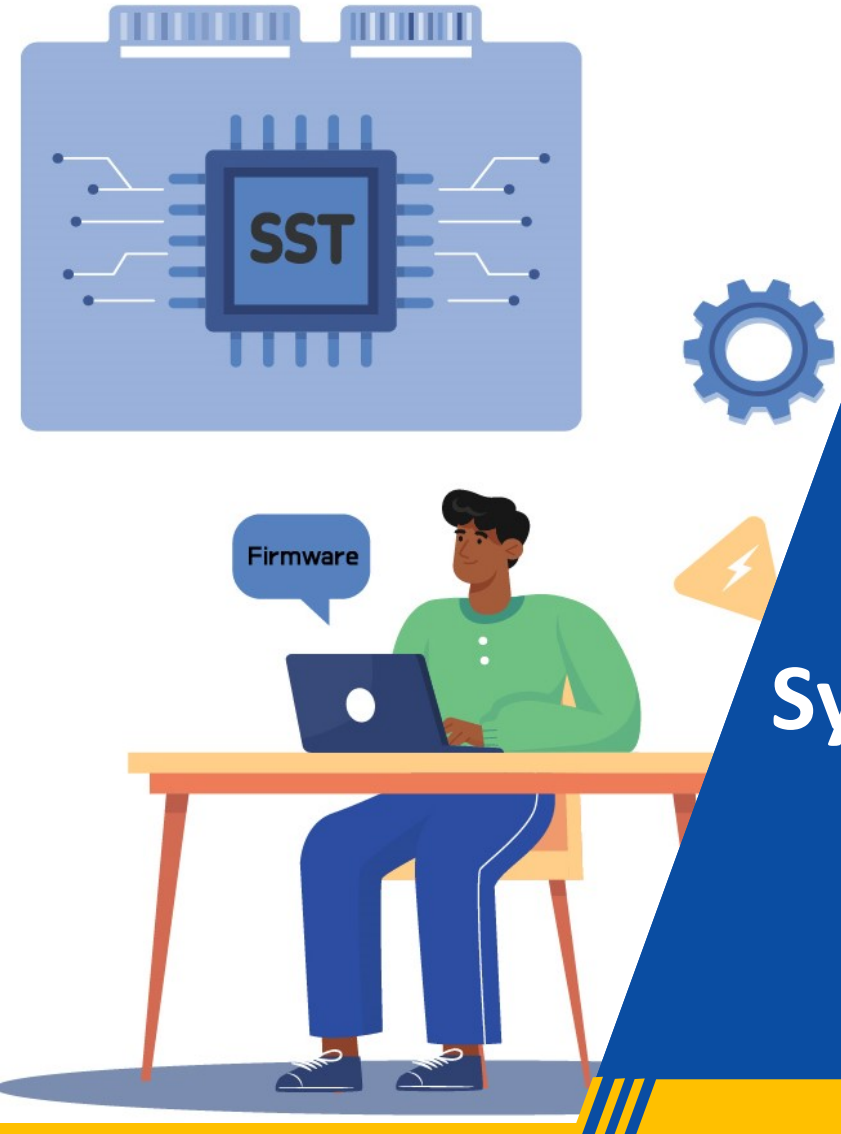
RQF LEVEL 4



CSAFD401

COMPUTER SYSTEM AND ARCHITECTURE

Embedded System Firmware Development

TRAINEE'S MANUAL

October 2024

EMBEDDED SYSTEM FIRMWARE DEVELOPMENT

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer's and trainee's manuals for the TVET Certificate IV in Computer system and architecture, specifically for the module **"CSAFD401: Embedded System Firmware Development."**

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) guiding policies and directives

Under Financial and Technical support

COORDINATION TEAM

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

Production Team

Authoring and Review

NISHIMIRWE Liliane

TUYISENGE Jean Claude

SENANI Alphonse

Validation

NIYONSHUTI Yves

HAKIZIMANA Evariste

MUGIRANEZA Jean Bosco

Conception, Adaptation and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe
SUA Lim
SAEM Lee
SOYEON Kim
WONYEONG Jeong
MANIRAKORA Alexis

Financial and Technical support

TQ KOICA through TQUM Project

TABLE OF CONTENT

ACRONYMS

AI: Artificial Intelligence

ARM: Advanced RISC Machine

BIOS: Basic Input/output System

Bluetooth: Bluetooth technology

Bootloader: Software that loads the main firmware.

CAN: Controller Area Network

CISC: Complex Instruction Set Computer

CMOS: Complementary Metal-Oxide-Semiconductor

CPLD: Complex Programmable Logic Device

DMA: Direct Memory Access

DSP: Digital Signal Processor

EEPROM: Electrically Erasable Programmable Read-Only Memory

Ethernet: Ethernet network

FLASH: Flash Memory

FPGA: Field-Programmable Gate Array

GCC: GNU Compiler Collection

GDB: GNU Debugger

GPIO: General-Purpose Input/Output

I2C: Inter-Integrated Circuit

ICE: In-Circuit Emulator

IDE: Integrated Development Environment

IoT: Internet of Things

ISR: Interrupt Service Routine

JTAG: Joint Test Action Group

KOICA: Korean International Cooperation Agency

MCU: Microcontroller Unit

ML: Machine Learning

PCB: Printed Circuit Board

RAM: Random Access Memory

RISC: Reduced Instruction Set Computer

ROM: Read-Only Memory

RTB: Rwanda TVET Board

RTL: Register Transfer Level

RTOS: Real-Time Operating System

SDK: Software Development Kit

SoC: System on a Chip

SPI: Serial Peripheral Interface

TQUM: TVET Quality Management Project

UART: Universal Asynchronous Receiver/Transmitter

USB: Universal Serial Bus

WiFi: Wireless Fidelity

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in computer system and architecture specifically for the module of "**Embedded System Firmware Development**". Trainees enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labour market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: CSAFD401: 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


FIRMWARE
LOADING (75%)

Indicative Contents

1.1. Collection of Data for Firmware Development

1.2. Analysing Collected Data

1.3. Extraction Of Firmware Requirements

Key Competencies for Learning Outcome 1: Identify Firmware Requirement.

Knowledge	Skills	Attitudes
• Description of firmware. • Description of Data Collection methods necessary for firmware development. • Identification of techniques for analysing data collected based on hardware and embedded system requirements. • Description of firmware requirements.	• Collecting data of firmware requirements. • Analysing collected data based on hardware requirements. • Analysing collected data based on embedded system requirements. • Extracting specific firmware requirements.	• Having Curiosity • Being Patient and Persistent. • Being Attentive to Detail • Having Adaptability • Having Collaboration • Having Critical Thinking • Being Responsible • Having Ethical Considerations • Being Self-Reliance

Duration: 20 hrs

Learning outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe clearly Firmware based on embedded system.
2. Describe clearly the methods used for data collection based on firmware development.
3. Collect clearly data of firmware based on embedded hardware device.
4. Identify effectively techniques of analysing data corrected based on embedded hardware requirements.
5. Identify effectively techniques of analysing data corrected based on embedded system requirements.
6. Analyse correctly data collected based on hardware requirements.
7. Analyse correctly data collected based on embedded system requirements.
8. Extract correctly specific firmware requirements based on firmware.

Resources

Equipment	Tools	Materials
• Computer • Projector	• Web browsers • Text editor	• Internet • Electricity

Indicative content 1.1: Collection of Data for firmware development.

Duration: 5 hrs

Theoretical Activity 1.1.1: Description of Firmware

Tasks:

1: You are requested to answer the following questions:

i What is embedded system firmware?
ii What is the difference between firmware and software?
iii Differentiate types of firmware.
iv What are the key components of firmware architecture?
v Explain applications of firmware.
vi Describe firmware development stage/ process.

2: Provide the answers for the asked questions and write them on flipchart/papers.

3: Present your findings to the trainer or your colleagues.

4: Ask for clarification if necessary.

5: Read the key readings 1.1.1 in trainee's manual.

Key readings 1.1.1.: Description of Firmware

1.1. Definition firmware

Firmware is software that provides the basic machine instructions that allow hardware to function and communicate with other software running on the device.

Or

Firmware is a specific type of computer software programmed on a hardware device that provides low-level control for a device's specific hardware.

1.2. Difference between firmware and software.

FIRMWARE	SOFTTWARE
Firmware is a specific type of software that is permanently stored in hardware devices.	Software refers to a general term for programs and applications that run on a computer or electronic device. Software can be broadly categorized into system software (like operating systems) and application software (like

	word processors, web browsers, and games).
It is a set of instructions or programs that are embedded into the hardware of a device during manufacturing.	It includes all the instructions that tell the hardware what to do and how to perform tasks
Firmware is permanently stored in the hardware especially in a chip of hardware.	Software is not permanently stored in the hardware.
Firmware is responsible for controlling various hardware components of a device, such as the BIOS (Basic Input/output System) in a computer, the software in embedded systems (like washing machines or microwave ovens), or the operating system in some mobile devices.	It is stored on various types of media, such as hard drives, solid-state drives, or optical disks, and loaded into the computer's memory (RAM) when the system starts. Software provides specific functionalities designed for.
Firmware is designed to be more stable and specific to the hardware it runs on. It is not meant to be modified or updated frequently.	Software can be easily updated and modified by users or developers.

1.3. Types of Firmware Data

- **BIOS (Basic Input/output System):**

BIOS firmware is essential for the booting process of a computer. It initializes hardware components during the boot process and provides the interface between the operating system and the computer's hardware.

- **UEFI (Unified Extensible Firmware Interface):**

UEFI is a modern replacement for BIOS. It provides more advanced features, such as support for larger hard drives, faster boot times, and a graphical user interface.

- **Bootloader:**

A bootloader is a small program that loads the operating system into a computer's memory during the boot process. It is stored in the firmware and helps initiate the operating system.

- **Device Firmware:**

Various hardware components in a computer system, such as hard drives, network adapters, graphics cards, and printers, have their own firmware. This firmware provides low-level control and ensures proper communication between the hardware and the operating system.

- **Embedded Firmware:**

Many embedded systems, such as microcontrollers in household appliances, cars, and industrial machines, have their own firmware. This firmware is tailored specifically for the embedded device's functionality.

- **Peripheral Firmware:**

Peripherals like keyboards, mice, and monitors may have their own firmware to support advanced features or customization options. For example, gaming mice often have firmware that allows users to customize button functions and sensitivity settings.

- **Network Device Firmware:**

Routers, modems, and other network devices have firmware that controls their networking capabilities. This firmware ensures proper data transmission, security protocols, and network configurations.

- **System Management Firmware:**

Servers and enterprise-level systems often have firmware that allows remote management and monitoring. This firmware provides features like out-of-band management, system diagnostics, and error logging.

- **Security Firmware:**

Some devices, especially in the realm of cybersecurity, have security-focused firmware. This firmware helps in tasks such as encryption, secure boot processes, and protecting against unauthorized access.

- **Software-defined Firmware:**

With advancements in technology, some firmware elements have become software-defined, meaning they can be updated or modified through software updates. This allows manufacturers to fix bugs, improve performance, or add new features without requiring hardware replacements.

1.4. Types of firmware

- **Three types of firmware**

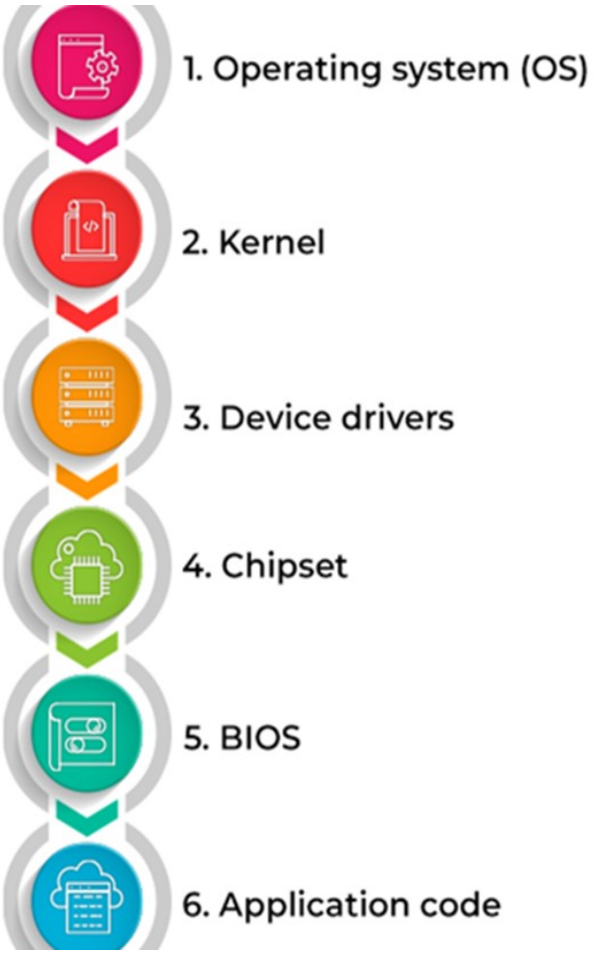
✓ **Low-level firmware**: Low-level firmware is considered an intrinsic part of a device's hardware. It is often stored on non-volatile, read-only chips like ROM and therefore cannot be rewritten or updated.

✓ **High-level firmware:** High level firmware allow updates and is generally more complex than low-level firmware. In a computer, high-level firmware resides on flash memory chips.

- ✓ **Subsystem firmware:** Subsystem firmware often comes as part of an embedded system. It is comparable to high-level firmware as it can be updated and is more complex than low-level firmware.

They are parts of a more extensive system that can work independently. It often looks like its device because the microcode for this firmware level is built into the central processing unit (CPU), the liquid crystal display units (LCD), and the flash chips. Also, it is like high-level firmware in terms of operation.

1.5. The key components of firmware architecture



1.5.1. Operating system (OS)

An operating system is a program that provides standard services for computer programs and manages its hardware and software resources. It allows resource sharing to allow multiple processes to run simultaneously without knowledge of each other's existence. A boot program helps load it onto a computer, then performs its managing functions. Examples of OS include Windows and Linux, which both include firmware.

1.5.2. Kernel

A kernel is a part of an OS software with complete control over the system. It facilitates communications between hardware and software components. It manages hardware resources such as memory, CPU, and input/output devices. It also handles conflicts regarding resource allocation and optimizes the resources.

1.5.3. Device drivers

Device drivers are programs that enable interaction with hardware devices. Without them, the hardware device cannot work. Device drivers are hardware-dependent and operating-system-specific. OS and other programs can interact with hardware through these drivers and act as translators.

1.5.4. Chipset

ROM and flash memory chips hold the firmware; since they are non-volatile, manufacturers may use the chips to store the firmware's permanent instructions. These chips can be rewritten and upgraded. Flash memory chips are reprogrammed during updates, while ROM integrated circuits need to be manually replaced.

1.5.5. BIOS

Basic input/output system (BIOS) firmware is installed during production, providing the computer with instructions on performing basic tasks such as keyboard control and booting. One can also use it to identify and configure hardware such as computer hard drives.

1.5.6. Application code

An application code refers to a set of programs designed to carry out a specific function and run on top of a system code. The application code in firmware enables it to send instructions to devices to function or perform basic tasks. It allows for low-level control.

1.6. Applications of firmware

- **Embedded Systems:**
  Firmware is used in embedded systems such as microcontrollers, sensors, and Internet of Things devices to control their behaviour. Examples include home appliances (washing machines, microwaves), medical devices, and automotive control systems.
- **Device Drivers:**
  Firmware often includes device drivers that allow the operating system to communicate with hardware components. This is crucial for peripherals like printers, graphics cards, and network adapters.
- **Gaming Consoles:**

Gaming consoles use firmware to control hardware components and enable various features, including graphics rendering, audio processing, and network connectivity.

- **Networking Devices:**

Routers, modems, and switches use firmware to manage network traffic, security protocols, and wireless communication standards. Firmware updates can enhance security and improve network performance.

- **Storage Devices:**

Hard drives, solid-state drives (SSDs), and flash drives have firmware that manages data storage, error correction, and wear levelling in the case of SSDs and flash memory.

- **Consumer Electronics:**

Smart TVs, digital cameras, and audio devices utilize firmware to provide user interfaces, process multimedia content, and connect to external devices through interfaces like HDMI and USB.

- **Automotive Systems:**

Cars use firmware to control various systems, including engine management, anti-lock braking systems (ABS), airbags, and entertainment systems. Firmware updates can improve fuel efficiency and overall performance.

- **Printers and Imaging Devices:**

Printers, scanners, and imaging devices have firmware to control the printing process, manage print queues, and ensure accurate scanning and image processing.

- **Industrial Automation:**

Firmware is used in programmable logic controllers (PLCs) and industrial robots to control manufacturing processes, monitor sensors, and ensure the safety and efficiency of industrial automation systems.

- **Wearable Devices:**

Smartwatches, fitness trackers, and health monitoring devices use firmware to collect and process data from sensors, display information to users, and communicate with smartphones or other devices.

- **Security Systems:**

Security cameras, access control systems, and alarm systems use firmware to capture, process, and transmit data, ensuring the security of homes and businesses.

- **Medical Devices:**

Various medical devices, such as heart rate monitors, insulin pumps, and pacemakers, rely on firmware to collect and process patient data and provide appropriate responses or alerts.

1.7. Firmware development stage/ process.

- **Choose the hardware platform:**

The hardware platform is an important decision that affects the overall performance and capabilities of the product. The choice of hardware platform is typically based on the requirements and the budget.

- **Design the firmware architecture:**

Once the requirements have been gathered and analysed, the next step is to design the firmware. This may involve creating a high-level design for the firmware, as well as defining the architecture and components that will be used.

- **Implementation:**

After the design has been completed, the next step is to implement the firmware. This typically involves writing code in a low-level programming language, such as C or assembly, as well as testing and debugging the firmware to ensure that it is functioning correctly.

- **Testing:**

Once the firmware has been implemented, it is important to thoroughly test it to ensure that it is functioning correctly and meets the requirements of the project. This may involve performing a variety of different types of testing, such as unit testing, integration testing, and acceptance testing.

- **System test:**

It ensures that the entire product fits the specifications. Generally, this testing does not fall under the purview of engineers, although it does fit into the test harness they developed.

- **Integration test:**

It ensures that the architecture's connected subsystems interact as they should. It verifies that the output is correct. This testing is often best developed by a testing group or software engineer.

- **Unit test:**

It involves individual software components to determine if they are performing correctly at the intermediate design level. The ideal people to develop this testing are those who write code under the test.

- **Firmware deployment:**

After the firmware has been tested and verified, it is ready to be deployed on the target device. This may involve flashing the firmware onto the device, as well as performing any necessary configuration tasks.

- **Maintenance and updates:**

Even after the firmware has been deployed, it is important to maintain and update it over time as needed. This may involve fixing bugs, adding new features, or making other updates to the firmware.

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. Primary data collection methods

- **Surveys and Questionnaires:** These involve asking individuals or groups a series of questions to gather information. Surveys can be conducted in person, via phone, through mail, or online.
- **Interviews:** Conducting structured, semi-structured, or unstructured interviews with individuals or groups to gather qualitative data through conversations.
- **Observation:** Directly observing and recording behaviours, events, or phenomena as they occur in a natural setting without interfering.
- **Experiments:** Controlled situations used to test hypotheses and gather data under specific conditions.

2.2. Secondary data collection methods

Secondary Data Collection: Gathering data from existing sources such as books, journals, databases, or records collected by others.

Theoretical Activity 1.1.2: Description of data collection methods

Tasks:

1: You are requested to answer the following questions:

i Description of data collection methods.

ii What are the tools used in data collection.

2: Provide the answers for the asked questions and write them on flipchart/papers.

3: Present your findings to the trainer or your colleagues.

4: Ask for clarification if necessary.

5: Read the key readings 1.1.2 in trainee's manual.

Key readings 1.1.2: Description of data collection methods

Data collection is the process of collecting and evaluating information or data from multiple sources to find answers to research problems, answer questions, evaluate outcomes, and forecast trends and probabilities.

It is an essential phase in all types of research, analysis, and decision-making, including that done in the social sciences, business, and healthcare.

During data collection, researchers must identify the data types, the sources of data, and the methods being used. We will soon see that there are many different data

collection_methods. Data collection is heavily reliance on in research, commercial, and government fields.

- **Before an analyst begins collecting data, they must answer three questions first:**

- What is the goal or purpose of this research?
- What kinds of data are they planning on gathering?
- What methods and procedures will be used to collect, store, and process the information?

Additionally, we can divide data into qualitative and quantitative types. Qualitative data covers descriptions such as color, size, quality, and appearance. Unsurprisingly, quantitative data deals with numbers, such as statistics, poll numbers, percentages, etc.

- **Why Do We Need Data Collection?**

Before a judge makes a ruling in a court case or a general creates a plan of attack, they must have as many relevant facts as possible. The best courses of action come from informed decisions, and information and data are synonymous.

The concept of data collection is not new, as we will see later, but the world has changed. There is far more data available today, and it exists in forms that were unheard of a century ago. The data collection process has had to change and grow, keeping pace with technology.

Whether you are in academia, trying to conduct research, or part of the commercial sector, thinking of how to promote a new product, you need data collection to help you make better choices.

Now that you know what data collection is and why we need it, let's look at the different methods of data collection. Data collection could mean a telephone survey, a mail-in comment card, or even some guy with a clipboard asking passers-by some questions.

1. What Are the Difference between Primary and secondary Data Collection Methods?

Primary and secondary methods of data collection are two approaches used to gather information for research or analysis purposes.

Let's explore each data collection method in detail

1.1. Primary Data Collection

The first techniques of data collection is Primary data collection which involves the collection of original data directly from the source or through direct interaction with the respondents. This method allows researchers to obtain firsthand information tailored to their research objectives.

There are various techniques for primary data collection, including:

a. **Surveys and Questionnaires:** Researchers design structured questionnaires or surveys to collect data from individuals or groups. These can be conducted through face-to-face interviews, telephone calls, mail, or online platforms.
b. **Interviews:** Interviews involve direct interaction between the researcher and the respondent. They can be conducted in person, over the phone, or through video conferencing. Interviews can be structured (with predefined questions), semi-structured (allowing flexibility), or unstructured (more conversational).
c. **Observations:** Researchers observe and record behaviors, actions, or events in their natural setting. This method is useful for gathering data on human behavior, interactions, or phenomena without direct intervention.
d. **Experiments:** Experimental studies involve manipulating variables to observe their impact on the outcome. Researchers control the conditions and collect data to conclude cause-and-effect relationships.
e. **Focus Groups**: Focus groups bring together a small group of individuals who discuss specific topics in a moderated setting. This method helps in understanding the opinions, perceptions, and experiences shared by the participants.

1.2. Secondary Data Collection

The next techniques of data collection is Secondary data collection which involves using existing data collected by someone else for a purpose different from the original intent. Researchers analyse and interpret this data to extract relevant information.

Secondary data can be obtained from various sources, including:

a. **Published Sources:** Researchers refer to books, academic journals, magazines, newspapers, government reports, and other published materials that contain relevant data.
b. **Online Databases:** Numerous online databases provide access to a wide range of secondary data, such as research articles, statistical information, economic data, and social surveys.

c. **Government and Institutional Records:** Government agencies, research institutions, and organizations often maintain databases or records that can be used for research purposes.
d. **Publicly Available Data:** Data shared by individuals, organizations, or communities on public platforms, websites, or social media can be accessed and utilized for research.
e. **Past Research Studies:** Previous research studies and their findings can serve as valuable secondary data sources. Researchers can review and analyse the data to gain insights or build upon existing knowledge.

2.Data Collection Tools

2.1.Surveys and Questionnaires

- **Google Forms**: A free tool for creating surveys and collecting responses online.
- **SurveyMonkey**: An online survey platform that allows for easy creation, distribution, and analysis of surveys.
- **Typeform**: A tool for designing interactive surveys and forms that enhance user engagement.

2.2. Interviews

- **Zoom**: A video conferencing tool suitable for conducting remote interviews with participants.
- **Microsoft Teams**: Another platform for video conferencing that allows for recording and transcription of interviews.
- **Otter.ai**: A transcription service that can transcribe spoken interviews in real-time, making it easier to analyze responses.

2.3. Observations

- **Evernote**: A note-taking app for documenting observations during user interactions with the embedded hardware.
- **Microsoft OneNote**: A digital notebook that can be used to capture notes, images, and observations.
- **Video Recording Tools**: Cameras or smartphones to record user interactions for later analysis (ensure user consent).

2.4. Document Review

- **Mendeley**: A reference manager that helps organize and annotate documents and research papers.
- **Zotero**: Another reference management tool for collecting, organizing, and reviewing research documents.
- **Google Drive**: For storing and sharing documents related to design specifications, user manuals, and feedback.

2.5. Focus Groups

- **FocusGroupIt**: An online platform for organizing and conducting focus groups with participants from different locations.
- **Miro**: A collaborative online whiteboard tool that can facilitate brainstorming and discussion in focus groups.
- **Doodle**: A scheduling tool that helps organize focus group sessions by finding convenient times for participants.

2.6. Experiments

- **MATLAB**: A programming environment for numerical computing that can be used to analyze experimental data.
- **R**: A programming language and software environment for statistical computing and graphics, useful for analyzing experimental results.
- **SPSS**: A software package used for statistical analysis that can help interpret data from experiments.

2.7. Case Studies

- **Case Study Template Tools**: Tools like Canva can be used to design visually appealing case study reports.
- **Google Docs**: A collaborative word processing tool to write and share case studies with team members.
- **Notion**: A versatile workspace for creating, organizing, and documenting case studies collaboratively.

Practical Activity 1.1.3: Collecting data for firmware development.

Task:

1: Referring to the key readings 1.1.3 you are requested to perform the following task.
As firmware developer, you are requested to collect data.
2: present your final works to your trainer/classmates
3: Ask for clarification where it is necessary.

Key readings 1.1.3: Steps to collect data when developing firmware

Example:

- **By using the developed embedded hardware called "smart thermostat"**

1. Set up the embedded hardware

Install the hardware**:** Mount the thermostat in the desired location (e.g., wall or HVAC system).

Power it on: Ensure proper electrical connections to provide power.

Connect to network**:** Link the thermostat to Wi-Fi or any communication protocol (e.g., Zigbee, Bluetooth).

2. Identify Sensors and Inputs

Temperature sensors**:** Verify functionality of the sensors that measure ambient temperature.

Humidity and motion sensors**:** If present, ensure these sensors are active and collecting environmental data.

User interface inputs**:** Capture any manual user input, like temperature adjustments via the thermostat screen or mobile app.

3.Define Data Points of Interest

- **Identify key metrics:** What data will help you achieve your goals? This could include:
  - **Performance metrics:** CPU usage, memory usage, execution time.
  - **Hardware sensor data:** Temperature, pressure, acceleration.
  - **User input:** Button presses, touch events, sensor readings.
  - **Error logs:** System crashes, unexpected events.
- **Choose appropriate data formats:** Consider the size and frequency of data, and the ease of processing and storage. Common formats include:
  - **Text files:** Simple and widely supported, but can be large.
  - **Binary files:** Efficient for large amounts of data, but requires specific parsing.
  - **Specialized data formats:** Like JSON or XML, for structured data.

4.Select Data Collection Methods

- **Hardware debugging tools:**
  - **Logic analyzers:** Capture digital signals on the microcontroller's pins.
  - **Oscilloscope:** Measure analog signals.
  - **JTAG debugger:** Provides access to internal registers and memory.
- **Software-based logging:**
  - **Print statements:** Simple but can be inefficient.
  - **Logging libraries:** Offer more advanced features like timestamps, levels, and filtering.
  - **Telemetry frameworks:** Designed for sending data over networks.
- **Third-party tools:**
  - **Profiling tools:** Analyze code execution and identify performance bottlenecks.
  - **Memory debuggers:** Detect memory leaks and other memory-related issues.
  - **Network analyzers:** Monitor network traffic.

5. Implement Data Collection in Firmware

- **Choose appropriate locations:** Place data collection points strategically to capture the most relevant information.
- **Consider timing and frequency:** Collect data at regular intervals or on specific events.
- **Minimize performance impact:** Avoid excessive data collection that slows down the system.

6.Store and Process Data

- **Choose a suitable storage medium:** This could be on-device memory, an SD card, or a remote server.
- **Consider data privacy and security:** If you're collecting user data, ensure it's handled responsibly.

I. **Collect data for firmware development by using bellow method:**

- **Observation method**

When gathering data through **observation** to develop firmware, the following aspects would be closely monitored:

1. **User Interactions**
   - Observe how users physically interact with the thermostat (e.g., adjusting temperature settings manually, using the touchscreen or buttons).
   - Monitor how often users manually override automated settings to understand where automation may need adjustments.

2. **Environmental Responses:**
    - Track how the thermostat responds to changes in room temperature over time and whether these changes align with user expectations.
    - Note how efficiently the thermostat adjusts heating or cooling in response to environmental factors (e.g., open windows, sunlight).
3. **User Behavior Patterns:**
    - Observe patterns in user behavior, such as what time of day users are most likely to adjust the thermostat and under what conditions (e.g., temperature drops in the evening).
    - Watch for potential frustrations or points of confusion when users interact with the device.
4. **System Response Time:**
    - Evaluate how quickly the thermostat reacts to user inputs or environmental changes, which is crucial for determining response-time requirements in the firmware.
    - Check for delays or misalignments between user expectations and the system's actual behavior.
5. **Error Handling and Feedback:**
    - Observe how the system handles errors (e.g., loss of connection to Wi-Fi, power fluctuations) and whether users receive adequate feedback or notifications.
    - Determine if users can easily troubleshoot or need support, which can inform the design of the firmware's error management functions.

➢ **Documentation method**

Documentation is a data collection method that involves analysing existing written, physical, or electronic materials. This can include documents, reports, records, photographs, videos, or any other form of recorded information.
When gathering information through **documentation** to develop firmware of the Smart Thermostat, the following key factor of documentation to be corrected:

1. **Hardware Specifications**
    - Review the technical specifications of the thermostat hardware, such as the microcontroller's processing power, memory capacity, and sensor capabilities (temperature, humidity, etc.).
    - Examine the connectivity options (Wi-Fi, Bluetooth) and power management details to ensure the firmware efficiently communicates and operates within the hardware's constraints.
2. **Existing Firmware Documentation**

- Study any documentation of previous or existing firmware used with the thermostat, including its architecture, features, known limitations, and bug reports.
- Understand how the current firmware interacts with hardware components, what processes work well, and where improvements are needed.

3. **Communication Protocols and APIs**
   - Review the communication protocols (e.g., MQTT, Zigbee) that the thermostat uses to connect to other devices or cloud services.
   - Examine any available APIs that allow the thermostat to communicate with smartphones, home automation systems, or cloud-based platforms. This will inform how to integrate the new firmware with external systems.
4. **User Manuals and Setup Guides**
   - Analyze user manuals and setup guides to understand the intended user workflow and common troubleshooting steps. This helps identify features that can be automated or simplified in the firmware to enhance usability.
   - Identify areas where users frequently encounter issues during setup or operation, which can provide insight into firmware features that need enhancement or simplification.
5. **Compliance and Regulatory Documentation**
   - Ensure that the thermostat hardware complies with industry standards for safety, energy efficiency, and wireless communication. Review compliance requirements (e.g., UL, CE, or Energy Star certifications) to ensure the firmware adheres to these standards.
   - Consider security and privacy regulations, particularly if the thermostat collects user data, to ensure the firmware incorporates robust data protection measures.
6. **Competitor Analysis**
   - Examine documentation from similar smart thermostats in the market, particularly around features, firmware functionality, and user reviews.
   - Identify industry best practices and potential gaps in competitors' products that could be addressed by your firmware.

Example Documentation of Smart Thermostat

1. Hardware Specifications

- **Microcontroller:** Model ABC123, 32-bit ARM Cortex-M4, 80 MHz clock speed, 512 KB Flash memory, 64 KB RAM.
- **Sensors:**
  - Temperature sensor: Range -10°C to 50°C, accuracy ±0.5°C.
  - Humidity sensor: Range 0% to 100% RH, accuracy ±3%.

- **Connectivity:**
  - Wi-Fi 802.11 b/g/n, 2.4 GHz frequency.
  - Bluetooth Low Energy (BLE) 4.2.
- **Display:** 2.4-inch capacitive touchscreen, 320x240 resolution.
- **Power Management:** 3.7V lithium-ion battery, 2500 mAh, with low-power sleep modes.
- **I/O Ports:**
  - GPIO pins for additional sensor/relay connections.
  - Micro-USB for power and debugging.

2. Existing Firmware Documentation

- **Current Firmware Version:** v1.4.5 (released March 2023).
- **Key Features:**
  - Automatic temperature control based on sensor data.
  - User control via touchscreen and mobile app.
  - Wi-Fi connectivity for remote access.
  - Basic scheduling features for heating/cooling cycles.
- **Known Issues:**
  - Occasional delay in response when adjusting settings manually.
  - Inconsistent Wi-Fi reconnection after power outages.
  - Limited customization for energy-saving modes.
- **Bug Reports:**
  - Some users report a lag in the thermostat's adjustment to rapid environmental changes (e.g., sudden drop in temperature).
  - Connectivity drops when switching between 2.4 GHz and 5 GHz networks.

3. Communication Protocols and APIs

- **Supported Protocols:**
  - **MQTT**: For sending/receiving data between the thermostat and cloud service.
  - **Zigbee**: For smart home integration (optional module).
- **API Documentation:**
  - RESTful API for integrating with mobile apps and home automation systems.
  - Endpoints for reading sensor data, adjusting temperature settings, and controlling modes (e.g., Away, Home, Sleep).
  - API supports GET/POST methods for user commands and data synchronization.

4. User Manuals and Setup Guides

- **User Workflow:**

    - Initial setup involves connecting to Wi-Fi via a mobile app and setting user preferences (e.g., temperature range, energy-saving mode).
    - Basic manual controls available via touchscreen, including temperature adjustment, mode switching, and scheduling.
    - Mobile app provides remote control and advanced scheduling.
- **Common Issues from User Feedback:**
    - Difficulty in setting up Wi-Fi in areas with weak signals.
    - Confusion over how to manually override the automatic mode.
    - Need for more advanced customization options, particularly for energy-saving.

5. Compliance and Regulatory Documentation

- **Safety Standards:** Compliant with UL and CE safety standards for electrical devices.
- **Energy Efficiency Certifications:** ENERGY STAR certified for energy-saving operation.
- **Wireless Communication Compliance:** Meets FCC regulations for Wi-Fi and Bluetooth transmission in residential settings.
- **Data Security and Privacy Compliance:**
    - Data collected from the thermostat (temperature, humidity, usage patterns) complies with GDPR and CCPA standards for user data protection.
    - Data encryption protocols (AES-256) used for transmitting information over Wi-Fi.

6. Competitor Analysis

- **Nest Learning Thermostat:**
    - Key Feature: Learning algorithm that adapts to user behavior over time.
    - **Weakness:** Some users find the learning algorithm difficult to reset when behavior patterns change (e.g., after vacations).
- **Ecobee Smart Thermostat:**
    - Key Feature: Built-in Alexa for voice control.
    - **Weakness:** Users report occasional lag in voice commands, especially in remote locations with weak Wi-Fi.
- **Comparison Insights:**
    - Your firmware could offer more precise temperature adjustments based on real-time data rather than predictive algorithms.
    - Focus on improving manual controls and customization options to give users more direct control over the system.

➢ **Interview data collection method**

Interview Information Gathering of Smart Thermostat Firmware Development

1. What features do you find most useful in the Smart Thermostat, and which ones do you rarely use or find frustrating?

Answer:

- I really like the automatic temperature adjustment, especially when it changes based on the time of day. However, I rarely use the scheduling feature because it is difficult to set up, and sometimes the thermostat does not respond to the changes I program. The manual override could be easier to access, too.

2. How do you typically control your thermostat?

Do you rely more on the touchscreen, mobile app, or voice commands?

Answer:

- I mostly use the mobile app because it is convenient, especially when I am not at home. The touchscreen is nice for quick adjustments, but it can be slow sometimes. I have not tried voice commands much because the response time is a bit laggy.

3. Have you encountered any issues with the thermostat's connectivity (e.g., Wi-Fi, mobile app)? How did you resolve them?

Answer:

- Yes, I have had issues where the thermostat loses Wi-Fi connection, especially after power outages. It usually takes a manual reset to reconnect, which can be annoying. Sometimes it struggles to reconnect on its own.

4. Do you think the thermostat responds quickly enough when you adjust the temperature? If not, where do you notice delays?

Answer:

- No, I have noticed that when I manually adjust the temperature on the touchscreen or the app, there is a slight delay before the change actually takes effect. It is not a big issue, but it would be nice if it were more responsive.

5. How important is energy-saving functionality to you, and what could be improved in this area?

Answer:

- Energy saving is very important to me, especially in terms of reducing my bills. I think the current energy-saving mode works okay, but I would like more customization options to fine-tune it to my needs. It could be smarter about adjusting when I am home versus when I am away.

7. **Have you ever experienced any issues with the thermostat's sensors (e.g., temperature accuracy)?**
   How did it affect your experience?

Answer:

- Yes, sometimes the temperature reading feels off, especially when it changes rapidly. It might say it is a comfortable temperature, but the room still feels too hot or cold. I had to manually adjust it until the room felt right.

7. What would make the Smart Thermostat easier to use on a day-to-day basis?

Answer:

- I would love it if the manual controls were more intuitive. A quicker way to override the automatic settings would be helpful. In addition, a clearer way to see what mode it is in without needing to dive into the settings would make it easier to use.

8. How do you feel about the thermostat's security features, especially when using remote access through the app?

Answer:

- I have not had any major concerns, but I would feel more secure if there were more visible security features, like notifications when the thermostat is accessed remotely or regular prompts to update passwords.

9. If you could add one new feature to the Smart Thermostat, what would it be?

Answer:

- I would add better integration with other smart home devices. For example, I'd like it to automatically lower the temperature when I lock the doors and leave, or turn on when I open the garage door after coming home.

10. What challenges do you face when setting up or maintaining the thermostat?

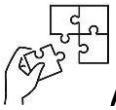
Answer:

The initial setup was tricky, especially getting the Wi-Fi to connect. Maintaining it isn't difficult, but I wish it would notify me when it's due for a software update or when it needs maintenance.

Points to Remember

- **Firmware** is specialized software permanently embedded in hardware devices, controlling their functions.
- There are three types of firmware: **low-level firmware**; **high-level firmware**, **subsystem firmware**.
- **Firmware architecture components includes:** the bootloader, kernel, device drivers, application code, memory management, and communication protocols.

- There are different applications of firmware like: electronics, embedded systems, networking gear, IoT devices, and peripherals.
- Data collection methods include surveys, interviews, observations, document review s, focus groups, experiments, and case studies, used individually or in combination t o gather information for analysis.
- While collecting data for firmware development pass through the following steps:
  1. Set up the Smart Thermostat
  2. Identify Sensors and Inputs
  3. Define Data Points to Collect
  4. Program the Firmware for Data Logging
  5. Test Data Collection Process



Application of learning 1.1.

The ABCD Company wants to collect data on embedded hardware of a thermostat. you are requested to collect data to ensure the firmware development.

Indicative content 1.2: Analysing collected data

Duration: 8 hrs

Theoretical Activity 1.2.1: Identification of hardware requirement and embedded system requirement

Tasks:

1: Answer the following questions:

a) What are the hardware requirements needed before developing a firmware?

b) What are the embedded system requirements?

2: Provide the answers for the asked questions and write them on flipchart/papers.

3: Present your findings to the trainer or your colleagues.

4: ask for clarification if necessary.

5: read the Key readings 1.2.1 in their manuals.

Key readings 1.2.1:Hardware requirements

- **Processing Power**
  - **CPU (Central Processing Unit):** Choose a CPU with the appropriate processing speed and cores based on the complexity of data analysis algorithms. Consider multi-core processors for parallel processing tasks.
  - **GPU (Graphics Processing Unit):** If your data analysis involves complex calculations (such as machine learning models), a GPU can significantly speed up the process.
  - **RAM (Random Access Memory**): Sufficient RAM is crucial for handling large datasets. The amount of RAM required depends on the size of the dataset and the complexity of the analysis. Analyse your data size to determine the RAM needs accurately.
- **Power Design**
  - **Power Efficiency:** Design the system for optimal power efficiency, especially if it operates on battery power. Low-power components and sleep modes are essential to conserve energy.
  - **Power Management ICs:** Utilize power management ICs to regulate and optimize power supply to different components based on their usage patterns. This ensures that power is allocated where and when it is needed.
- **Communication and Interfaces**
  - **Networking:** Consider the type of network interfaces needed (Ethernet, Wi-Fi, cellular) for data transfer. Ensure compatibility with the data analysis tools and systems.

- **Protocols:** Implement appropriate communication protocols (e.g., MQTT, HTTP, CoAP) depending on the network architecture and security requirements.
- Interfacing with Sensors: If your firmware collects data from sensors, ensure the firmware can interface with various sensor types (analog, digital, I2C, SPI) to gather necessary data.

- **Storage Capacity**
- **Primary Storage (Flash Memory, SSD, HDD):** Choose storage based on the volume of data to be collected and processed. Flash memory is common in embedded systems due to its speed and durability. SSDs and HDDs are suitable for larger storage requirements but consume more power.
- **Secondary Storage:** For data redundancy and backup, consider secondary storage solutions. Cloud storage or network-attached storage (NAS) devices can be integrated into the firmware architecture.

- **Additional Considerations**
- **Security:** Implement encryption algorithms and secure boot mechanisms to protect both data at rest and in transit. Security modules and hardware accelerators can enhance the overall security of the system.
- **Temperature and Environmental Considerations:** Ensure that the hardware components can operate within the specified temperature range and environmental conditions where the firmware will be deployed.

2.Embedded system requirements

✓ **Functional Requirements**

- **Task Execution:** Define the core functions the firmware needs to perform.
- **Input Processing:** Specify how inputs from sensors or user interfaces are processed.
- **Output Handling:** Describe how the firmware will generate outputs for displays or actuators.
- **Error Handling:** Detail how errors or exceptional situations will be detected and managed.

✓ **Performance Requirements**

- **Response Time:** Define the maximum acceptable delay for processing inputs and generating outputs.
- **Throughput:** Specify the number of tasks or operations the system should handle per unit of time.
- **Concurrency:** Describe how the firmware handles multiple tasks simultaneously.

✓ **Power Requirements**

- **Power Consumption:** Define the acceptable power usage in various operating modes.
- **Sleep Modes:** Specify how and when the system will enter low-power states.
- **Battery Life:** If applicable, define the expected duration of operation on a single battery charge.

✓ **Environmental Requirements**

- **Operating Temperature:** Define the acceptable temperature range for the system to operate.
- **Humidity:** Specify the acceptable humidity levels.
- **Vibration/Shock Resistance:** Specify the tolerance to vibration and mechanical shock. **EMI/EMC Compliance:** Ensure electromagnetic interference and compatibility compliance.

✓ **Form Factor and Size Analysis**

- **Dimensions:** Specify the physical dimensions and weight constraints of the embedded system.
- **Mounting Requirements**: Describe how and where the system will be mounted or installed.

✓ **Communication and Interface Analysis**

- **Data Protocols:** Specify the communication protocols (such as UART, SPI, I2C) used for internal and external communication.
- **Network Connectivity:** Specify if the system requires networking capabilities (Ethernet, Wi-Fi, Bluetooth).
- **User Interface:** Describe the interfaces the firmware needs to interact with, such as buttons, touchscreens, or other sensors.

✓ **Analog and Digital Input/Output Analysis**

- **Analog Inputs:** Specify the range and precision of analog sensors the system will interface with.
- **Digital Inputs/Outputs:** Define the types and number of digital inputs and outputs the system supports.
- **Signal Processing:** Describe any necessary signal conditioning or processing for input signals.

Practical Activity 1.2.2: Analysing collected data based on hardware requirements.

Task:

1: Referring to the key readings 1.2.2 you are requested to perform the following task.

i. Given a dataset of collected data on an embedded thermostat's hardware, you are requested to analyse this data of embedded thermostat's hardwar e by identifying potential performance, compatibility issues, or areas for op timization based on the hardware requirements?

2: present your final works to your trainer/classmates

3: Ask for clarification where it is necessary.

Key readings 1.2.2: Analysing collected data based on hardware requirements.

1. Steps of analysing data collected on firmware:

Analysing data is a systematic process that involves several key steps. Here's a general outline of the steps you might follow:

- **Define Objectives**

Clearly outline what you want to achieve with the analysis. What questions are you trying to answer?

- **Data Collection**

Gather the data you need. This could be from surveys, experiments, databases, or other sources.

- **Data Cleaning**

Prepare your data for analysis by removing any inaccuracies, duplicates, or irrelevant information. This may also involve handling missing values.

- **Data Exploration**

Conduct an exploratory data analysis (EDA) to understand the data's structure and characteristics. Use visualizations and summary statistics to identify patterns or anomalies.

- **Data Transformation**

Depending on your analysis needs, you may need to transform the data. This could include normalizing values, creating new variables, or aggregating data.

- **Perform Analysis**

Apply the selected methods to your data. This might involve running statistical tests, building models, or creating visual representations.

- **Interpret Results**

Analyse the output of your analysis in the context of your original objectives. What

do the results mean? Do they answer your questions?

- **Communicate Findings**

Present your findings in a clear and concise manner. Use visualizations, reports, or presentations to convey your insights effectively.

- **Make Decisions**

Based on your analysis, make informed decisions or recommendations. Consider the implications of your findings.

- **Review and Iterate**

Reflect on the analysis process. What worked well? What could be improved? If necessary, revisit earlier steps to refine your analysis.

2. Analyse collected data based on hardware requirements.

When analysing collected data in the context of embedded hardware requirements, it is essential to follow a systematic approach that ensures all aspects of the hardware specifications are met.

2.1 This analysis can be broken down into several key steps:

1. Define Hardware Requirements: Before any data analysis can take place, it is crucial to clearly define the hardware requirements for the embedded system. These requirements typically include specifications such as processing power, memory capacity, energy consumption, input/output interfaces, and environmental constraints (e.g., temperature ranges). Understanding these parameters helps in setting benchmarks for performance evaluation.

2. Data Collection: Once the hardware requirements are established, data must be collected from various sources related to the embedded system's operation. This may include performance metrics (CPU usage, memory utilization), operational logs (error rates, response times), and environmental data (temperature readings). Effective data collection mechanisms should be implemented to ensure that relevant information is captured accurately.

3. Data Preprocessing: The raw data collected often contains noise or irrelevant information that can skew analysis results. Therefore, pre-processing steps such as filtering out outliers, normalizing values, and transforming data formats are necessary. This step ensures that only clean and relevant data is used for further analysis.

4. Performance Analysis: With preprocessed data in hand, engineers can conduct performance analyses to evaluate how well the embedded hardware meets its defined requirements. Key performance indicators (KPIs) should be established based on the initial requirements. For example:

- **Processing Power:** Analyze CPU load during peak operations to determine if it stays within acceptable limits.
- **Memory Usage:** Monitor memory allocation patterns to identify potential leaks or inefficiencies.

- **Energy Consumption:** Evaluate power usage under different operational scenarios to ensure compliance with energy efficiency standards.

5. Fault Detection and Diagnosis: Data analytics techniques can also be employed to detect faults or anomalies in the system's operation. By analyzing historical performance data against expected behavior defined by hardware requirements, engineers can identify patterns indicative of failures or inefficiencies. Techniques such as statistical analysis and machine learning algorithms can aid in predicting potential issues before they lead to system failures.

6. Reporting and Visualization: After completing the analysis, results should be compiled into comprehensive reports that highlight findings related to hardware performance against specified requirements. Visualization tools can help present this data effectively through graphs and charts that illustrate trends over time or comparisons against benchmarks.

8. **Iterative Improvement:** Finally, based on insights gained from the analysis, recommendations for improvements should be made regarding both hardware design and software optimization strategies. This iterative process allows for continuous enhancement of embedded systems by refining both their hardware components and their operational algorithms.

Steps of Analyzing Collected Data Based on Embedded Hardware Requirements

1. Introduction to Embedded Hardware Requirements

Embedded systems are specialized computing systems that perform dedicated functions within larger mechanical or electrical systems. The analysis of collected data in the context of embedded hardware requirements involves understanding how the hardware specifications influence data collection, processing, and storage capabilities. This analysis is crucial for ensuring that the embedded system meets its performance, power consumption, and reliability targets.

2. Defining the Data Collection Objectives

Before analyzing collected data, it is essential to define what data needs to be collected based on the application of the embedded system. For example, if an embedded system is designed for environmental monitoring, it may need to collect temperature, humidity, and air quality data. The objectives will guide the selection of sensors and dictate the necessary hardware specifications.

3. Identifying Hardware Specifications

The next step is to identify the hardware specifications required for effective data collection:

- **Microcontroller/Processor:** The choice of microcontroller affects processing speed and capability. For instance, a low-power microcontroller may be sufficient for simple tasks but may struggle with complex computations.
- **Memory Requirements:** Depending on the volume of data being collected

(e.g., high-resolution sensor readings), adequate RAM and flash memory must be allocated to store temporary and permanent data.

- **Sensor Compatibility:** The selected sensors must be compatible with the microcontroller in terms of communication protocols (e.g., I2C, SPI) and voltage levels.
- **Power Consumption:** For battery-operated devices, power consumption becomes critical. Analyzing how much power each component consumes during operation can help optimize battery life.

4. Data Collection Process

Once hardware specifications are established, data can be collected through various methods:

- **Real-time Data Acquisition:** Sensors continuously send data to the microcontroller for immediate processing.
- **Batch Processing:** Data can be collected over a period and processed in batches to reduce power consumption during idle times.

5. Analyzing Collected Data

After collecting data, analysis involves several steps:

- **Data Validation:** Ensure that the collected data is accurate by checking against expected ranges or using checksums.
- **Statistical Analysis:** Use statistical methods to analyze trends or anomalies in the data set. For example, calculating mean values or standard deviations can provide insights into sensor performance over time.
- **Performance Metrics Evaluation:** Assess whether the embedded system meets its design requirements based on metrics such as response time (how quickly it processes incoming sensor data) and throughput (the amount of data processed in a given time).

6. Optimization Based on Analysis

Based on the analysis results:

- If certain sensors are found to consume excessive power without providing significant benefits in accuracy or resolution, they might be replaced with more efficient alternatives.
- If processing delays are observed due to insufficient memory or CPU speed, upgrading these components could enhance overall system performance.

Example of analyzing Collected Data on Thermostats Based on Embedded Hardware Requirements

1. Introduction to Thermostat Functionality and Data Collection

Thermostats are devices that regulate temperature by controlling heating and cooling systems. They can be either mechanical or digital, with modern thermostats often incorporating embedded hardware for enhanced functionality.

The data collected from these devices can include temperature readings, humidity levels, user settings, and system performance metrics. Analyzing this data is crucial for understanding the hardware requirements necessary for efficient operation.

2. Embedded Hardware Components in Thermostats

To analyze the collected data effectively, it is essential to understand the embedded hardware components typically found in thermostats:

- **Microcontroller (MCU)**: The core component that processes inputs from sensors and executes control algorithms. It must have sufficient processing power and memory to handle real-time data.
- **Temperature Sensors**: These sensors (e.g., thermistors or RTDs) measure ambient temperature. Their accuracy and response time directly affect the thermostat's performance.
- **Humidity Sensors**: Some thermostats also monitor humidity levels to optimize comfort and energy efficiency.
- **User Interface (UI)**: This includes displays (LCD or LED) and input methods (buttons or touchscreens) that allow users to set preferences.
- **Communication Modules**: Many modern thermostats feature Wi-Fi or Bluetooth connectivity for remote control via smartphones or integration into smart home systems.

3. Data Types Collected from Thermostats

The types of data collected from thermostats can be categorized as follows:

- **Environmental Data**: This includes real-time temperature and humidity readings, which are critical for maintaining comfort levels.
- **Operational Data**: Information about system performance, such as run times of heating/cooling cycles, energy consumption statistics, and fault detection alerts.
- **User Interaction Data**: Logs of user settings changes, schedules programmed by users, and usage patterns over time.

4. Analysis of Hardware Requirements Based on Collected Data

When analyzing the collected data concerning embedded hardware requirements, several factors must be considered:

- **Processing Power**: The MCU must be capable of handling multiple sensor inputs simultaneously while executing control algorithms efficiently. For example, a thermostat that integrates machine learning capabilities may require a more powerful processor than a basic model.
- **Memory Capacity**: Sufficient RAM is necessary to store temporary data during processing, while flash memory is needed for firmware updates and long-term storage of operational logs.
- **Sensor Accuracy & Type**: The choice of sensors impacts both the quality of data collected and the overall design complexity. High-quality sensors may

require additional circuitry for signal conditioning.

- **Power Consumption**: Energy-efficient designs are crucial for battery-operated models or those aiming to minimize energy use in wired installations.
- **Connectivity Options**: Depending on whether the thermostat will connect to a home network or smart home ecosystem, appropriate communication modules must be integrated without compromising power efficiency.

1.2 Data Analysis Tools

- **Disassemblers and Decompilers:** Tools like IDA Pro, Ghidra, and Binary Ninja can disassemble or decompile firmware to analyze its code structure and identify potential vulnerabilities.
- **Hex Editors:** Hex editors like HxD or 010 Editor can be used to examine the raw bytes of the firmware and search for specific patterns or strings.
- **String Search Tools:** Tools like grep can be used to search for sensitive data, such as passwords, API keys, or personal information.
- **Data Flow Analysis:** Tools can analyze the flow of data within the firmware to identify potential vulnerabilities, such as buffer overflows or memory leaks.
- **Static Analysis Tools:** Static analysis tools can automatically scan firmware for potential vulnerabilities without executing the code.

1.3 Statistical Analysis

- **SPSS (Statistical Package for the Social Sciences):** A comprehensive statistical analysis software widely used in social sciences, market research, and healthcare.
- **R:** A free and open-source programming language and environment for statistical computing and graphics.
- **SAS (Statistical Analysis System):** A powerful statistical software suite used in academia, industry, and government.
- **Python with libraries like NumPy, pandas, and SciPy:** Python is a versatile language with extensive libraries for data analysis and scientific computing.

1.4 Data Visualization

- **Tableau:** A powerful data visualization tool that allows you to create interactive dashboards and reports.
- **Power BI:** Microsoft's business intelligence tool for analyzing and visualizing data.
- **QlikView:** A self-service business intelligence platform that enables users to explore data and create visualizations.

- **Matplotlib and Seaborn:** Python libraries for creating static and interactive visualizations.

1.5 Machine Learning and Artificial Intelligence

- **TensorFlow:** An open-source platform for machine learning, developed by Google.
- **PyTorch:** A Python-based machine learning library developed by Facebook.
- **Scikit-learn:** A Python library for machine learning algorithms, including classification, regression, and clustering.
- **KNIME:** An open-source platform for data analysis, machine learning, and data mining.

1.6 Big Data Analysis

- **Hadoop:** A framework for processing and storing large datasets across clusters of computers.
- **Spark:** A fast and general-purpose cluster computing system that can be used for data analysis and machine learning.
- **Apache Flink:** A distributed streaming data processing framework for processing real-time data streams.

1.7. Most known used Data analysis Tools

- **Excel:** While primarily a spreadsheet software, Excel can also be used for basic data analysis and visualization.
- **Google Sheets:** A cloud-based spreadsheet application with similar capabilities to Excel.
- **OpenOffice Calc:** A free and open-source alternative to Excel.

1.8.Techniques for Identifying Sensitive Data

- **String Search:** Search for common patterns of sensitive data, such as credit card numbers, email addresses, or passwords.
- **Regular Expressions:** Use regular expressions to define patterns for sensitive data and search for matches.
- **Data Classification:** Classify data based on its sensitivity level and prioritize analysis of critical data.

2.Techniques for Identifying Vulnerabilities

- **Code Analysis**: Analyse the code for common vulnerabilities, such as buffer overflows, integer overflows, and memory leaks.
- **Static Analysis:** Use static analysis tools to automatically scan the firmware for potential vulnerabilities.
- **Dynamic Analysis:** Execute the firmware in a controlled environment and monitor its behavior for signs of vulnerabilities.

- Fuzzing: Use fuzzing techniques to test the firmware's resilience to unexpected inputs and identify vulnerabilities.

✓ **Data Analysis Techniques:**

- **Descriptive Statistics**: Calculate mean, median, mode, standard deviation, and other statistical measures to summarize the data.
- Visualization: Create graphs, charts, or other visual representations to identify trends and patterns.
- **Correlation Analysis:** Determine the relationships between different variables in the data.
- **Anomaly Detection**: Identify unusual or unexpected data points that may indicate problems.
- **Benchmarking**: Compare the performance of the embedded thermostat against benchmarks or industry standards.
- Potential Performance Issues:
- **Insufficient Processing Power:** The microcontroller may not be able to handle the workload efficiently.
- **Memory Constraints**: Limited memory may restrict the system's capabilities.
- **Sensor Limitations:** Inaccurate or unreliable sensor data can affect the system's performance.
- **Network Connectivity Issues:** Poor network connectivity can impact the system's responsiveness.

✓ **Compatibility Issues:**

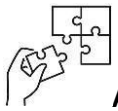
- **Driver Conflicts**: Incompatible or outdated drivers can cause problems.
- **Hardware Incompatibility**: Components may not be fully compatible with each other.
- **Software Conflicts:** The firmware may conflict with other software running on the system.

✓ **Optimization Opportunities:**

- **Algorithm Optimization**: Improve the efficiency of algorithms used by the firmware.
- **Resource Allocation**: Optimize the allocation of resources (e.g., CPU, memory) to critical tasks.
- **Power Management**: Implement power-saving techniques to reduce energy consumption.
- **Firmware Updates:** Update the firmware to address known issues and improve performance.

Points to Remember

- Hardware requirements needed before developing a firmware include a suitable microcontroller or microprocessor, a compatible development board, and reliable power supply, debugging tools, programming interfaces, and test equipment.
- The software requirements for embedded systems include a real-time operating system (RTOS) or bare-metal programming environment, development tools such as compilers and debuggers, and libraries for hardware abstraction.
- **Steps of analysing data collected**
    1. Define Objectives
    2. Data Collection
    3. Data Cleaning
    4. Data Exploration
    5. Data Transformation
    6. Perform Analysis
    7. Interpret Results
    8. Communicate Findings
    9. Make Decisions
    10. Review and Iterate



Application of learning 1.2.

The ABCD Company makes smart home devices, including a thermostat. They have collected data from their thermostat hardware to check how it's performing. You are requested to analyse the data.

Indicative content 1.3: Extraction of firmware requirements

Duration: 7 hrs

Theoretical Activity 1.3.1: Description of firmware requirements extraction

Tasks:

1: You are requested to answer the following questions.

i. Describe extraction of firmware requirements?
ii. What are the two key points to consider while identifying system usage?
iii. What is functional requirement of a firmware?
iv. What is non-functional requirement of a firmware?
v. What do you mean by system feasibility?
vi. List key aspects of system feasibility.

2: Provide the answers for the asked questions and write them on flipchart/paper.

3: Present your findings to the trainer or your colleagues.

4: Ask for clarification if any.

5: Read the key readings 1.3.1 in trainee's manual.

6: Ask to trainer for clarification if any.

Key readings 1.3.1.: Description of firmware requirements extraction

1. **Extraction of firmware requirements**

Extraction of firmware requirements is the process of gathering, analysing, and docum enting specific firmware requirements.

This process is based on identification of system usage and system feasibility.

2. **Identification of system usage**

a) **Functional requirements**: functional requirements are the capabilities that the firmware must possess to meet its intended purpose. They describe what the firmware should do and how it should respond to various inputs or conditions.

Functional requirements of firmware specifications:

- The core functions and features the firmware must perform
- How the firmware should interact with hardware components
- Specific operations or algorithms it needs to execute
- Input/output behaviors and data processing requirements
- System states and transitions between them

b) Non-functional requirements: These are the requirements that focus on how well the system performs

- **Non-functional requirements of firmware specifications:**
  - Performance: Speed, response time, throughput
  - Reliability: Error handling, fault tolerance, recovery mechanisms
  - Security: Data protection, access control, encryption
  - Memory usage: ROM/RAM utilization
  - Power consumption: Energy efficiency, battery life
  - Maintainability: Code structure, modularity, documentation
  - Portability: Ability to run on different hardware platforms
  - Scalability: Ability to handle increased load or complexity
  - Compliance: Adherence to standards and regulations
  - Usability: User interface responsiveness (if applicable)

3. System feasibility

System feasibility is an assessment of whether a proposed system is viable, practical, and worth implementing. It's typically conducted early in the project lifecycle to determine if the project should proceed.

4. Key aspects of system feasibility:

The followings are key aspects of system feasibility and corresponding questions to ask yourself when evaluating them.

a. Technical Feasibility

This evaluates the technical capabilities required to extract and define the firmware requirements.

Key considerations include:

- **Hardware Compatibility**: Can the system's hardware support the firmware? Is the firmware compatible with the target architecture, such as microcontrollers, processors, or embedded systems?
- **Tools and Technology**: What tools are needed to extract the firmware? This may include hardware debuggers, software analysis tools, and reverse-engineering methods. Is the necessary technology available (e.g., JTAG, SWD, or debugging interfaces)?
- **Firmware Extraction Techniques**: What techniques will be used to extract the firmware (e.g., memory dumps, direct access to firmware storage, or using software interfaces)? Are the existing extraction methods robust enough to handle the firmware in question?
- **System Requirements**: What specific system resources (e.g., memory, processor capabilities) are required to extract, analyze, and store firmware? Can these be met by the available hardware and software infrastructure?

- **Security and Encryption**: If the firmware is encrypted or has security measures (e.g., secure boot, DRM), can they be bypassed or decrypted? Is the firmware protected in such a way that extraction is technically feasible?

b. Economic Feasibility

Economic feasibility assesses the financial aspects of extracting firmware requirements, including whether the benefits of extracting the firmware and creating requirements justify the costs.

Key considerations include:

- **Cost of Tools**: What is the cost of the tools needed for extraction (e.g., hardware probes, software for disassembling firmware)? Is the investment justified by the value of the extracted firmware requirements?
- **Labor Costs**: How many hours of work are needed to extract, analyze, and document the firmware requirements? What is the cost of skilled personnel (e.g., embedded system engineers, reverse engineers)?
- **Return on Investment (ROI)**: What is the potential business value of extracting and using the firmware? Will it enable the company to build better products, troubleshoot issues, or enable third-party development or modification?
- **Alternative Solutions**: Are there cheaper or more efficient ways to gather firmware requirements (e.g., using open-source firmware, vendor documentation)? Can existing firmware be repurposed, or is the extraction and analysis of new firmware essential?

c. Legal Feasibility

Legal feasibility ensures that extracting firmware requirements complies with all applicable laws and regulations. This is especially important in cases where firmware extraction may involve proprietary or protected software.

Key considerations include:

- **Intellectual Property (IP)**: Does the firmware have proprietary code, patents, or copyright protections? Will extracting the firmware violate intellectual property laws (e.g., reverse engineering restrictions)?
- **Licensing**: Is the firmware covered under any licenses (e.g., open-source licenses, proprietary licenses) that restrict reverse engineering or require specific conditions to be met for extraction?
- **Export Control Laws**: Are there any restrictions related to the distribution or modification of firmware, especially in certain countries (e.g., ITAR, EAR in the United States)?

- **Confidentiality Agreements**: Does the firmware belong to a third party with whom there are non-disclosure or confidentiality agreements that prohibit extraction or analysis?

d. Operational Feasibility

Operational feasibility examines whether extracting firmware requirements can be done effectively in the context of the organization's operations and processes.

Key considerations include:

- **Skill Set of the Team**: Do the engineers or technical staff have the necessary expertise in firmware extraction, reverse engineering, or embedded systems analysis? Will additional training be required?
- **Infrastructure and Resources**: Do the current operational systems support the tools and processes required for firmware extraction? Is there enough computational power or lab equipment (e.g., oscilloscopes, debuggers) available?
- **Impact on Existing Workflows**: Will extracting firmware interfere with ongoing development or production workflows? Will it require downtime for hardware or systems in use?
- **End-User Requirements**: How will the extracted firmware requirements be used in the operational context? Will they lead to system enhancements, debugging, or the creation of new firmware versions? Are operational needs aligned with the firmware capabilities?

e. Scheduling Feasibility

Scheduling feasibility evaluates whether the extraction of firmware requirements can be completed within the desired time frame, including any dependencies and timelines for related activities.

Key considerations include:

Timeline for Extraction: How long will it take to complete the extraction and documentation of firmware requirements? Does the process involve reverse engineering, debugging, or testing that will take significant time?

Project Milestones: Are there key milestones for this project (e.g., initial extraction, analysis, testing, final documentation)? Are these achievable within the expected timeline?

Dependency on Other Tasks: Is firmware extraction dependent on hardware availability, availability of documentation, or testing phases that could cause delays?

External Factors: Are there any external deadlines or time constraints (e.g., product launches, compliance timelines) that the extraction process needs to align with?

Practical Activity 1.3.2: Extracting Firmware requirements

Task:

1: Referring to the key readings 1.3.2, you are requested to perform the following task.
As firmware developer, you are requested to collect data.
2: present your final works to your trainer/classmates
3: Ask for clarification where it is necessary.
4: Ask to trainer for clarification if any.

Key readings 1.3.2.: Steps for extracting firmware requirements:

1.Define System Objectives:

Understand the purpose of the system and the desired functionalities.

Clarify how the firmware should interact with hardware components (e.g., sensors, LEDs, buttons).

2.Gather Stakeholder Input:

Engage stakeholders (e.g., hardware engineers, product managers, users) to gather expectations and requirements.

Identify use cases and edge cases that the firmware needs to handle.

3.Analyze Hardware Specifications:

Understand the hardware platform, including microcontroller, sensors, peripherals, power constraints, and communication interfaces.

Ensure the firmware requirements align with hardware capabilities.

4. Understand Operational Scenarios:

Determine how the system operates in different scenarios (e.g., normal operation, error states, low-power modes).

Define interactions between the firmware and external components (e.g., user actions, system inputs).

5.Identify Key Functionalities:

List the core functionalities the firmware must support (e.g., input detection, communication, signal processing).

Specify timing constraints, response times, and precision for critical operations.

6.Define Input and Output Specifications:

Determine how inputs (e.g., button presses) are detected and processed.

Specify how the firmware will control outputs (e.g., LED blinks, motor control) in response to inputs.

7.Set Power Management Requirements:

Define power-saving modes and conditions for entering/exiting low-power states.

Specify how to balance performance with energy efficiency.

8.Consider Error Handling and Fault Tolerance:

Define mechanisms for detecting, reporting, and handling errors (e.g., sensor failures, communication issues).

Ensure the system can recover from faults or unexpected conditions.

9.Ensure Scalability and Modularity:

Ensure the firmware can be easily adapted for future expansions (e.g., adding features, supporting new hardware).

Plan for a modular architecture to simplify updates and maintenance.

10.Plan for Testing and Debugging:

Identify how the firmware will be tested (e.g., unit tests, integration tests, debugging tools).

Define logging and monitoring mechanisms to track system performance and diagnose issues.

11.Consider Firmware Update Capabilities:

Define how firmware updates will be handled (e.g., over-the-air updates, bootloader support).

Ensure the update process is secure and reliable.

12.Review and Validate Requirements:

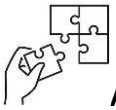
Review the collected requirements with stakeholders to ensure they meet the system's needs.

Validate that the requirements are feasible given the hardware and development constraints.

Points to Remember

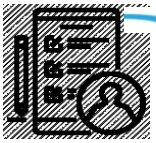
- **Extraction of firmware requirements** is the process of gathering, analysing, and documenting specific firmware requirements.
- **Functional requirements** are the capabilities that the firmware must possess to meet its intended purpose.
- **Non-functional requirements** are those requirements that focus on how well the system performs.
- Key aspects of System feasibility are: **technical, economic, legal, operational and scheduling feasibility.**
- **Steps for extracting firmware requirements:**
    1. Define System Objectives
    2. Gather Stakeholder Input

3. Analyse Hardware Specifications
4. Understand Operational Scenarios
5. Identify Key Functionalities
6. Define Input and Output Specifications
7. Set Power Management Requirements
8. Consider Error Handling and Fault Tolerance
9. Ensure Scalability and Modularity
10. Plan for Testing and Debugging
11. Consider Firmware Update Capabilities
12. Review and Validate Requirements



Application of learning 1.3.

ABCD company needs to develop an LED blink system that provides visual feedback in response to user actions, specifically button presses. You are requested to extract comprehensive firmware requirements to ensure the LED blink system operates correctly and efficiently with a capability to detect various types of buttons presses and hence providing immediate and appropriate visual feedback through LED blinks.



Learning outcome 1 end assessment

Theoretical assessment

1. What is the definition of firmware?
2. What are the different types of firmware?
3. What types of requirements should be considered when you are analysing data for firmware development?
4. Which of the following is a key aspect to consider when extracting firmware requirements for a device?
    a) User interface design
    b) Power consumption
    c) Hardware compatibility
    d) All the above
5. True or False Questions on Firmware Requirements
    a) Processing power is an essential factor in determining the performance of firmware in a device.
    b) Power design considerations are irrelevant when developing firmware for low-power devices.
    c) Communication and interfaces play a critical role in how firmware interacts with other hardware components.
6. Match the embedded system requirements with their descriptions:

Embedded System Requirements	**Descriptions**	Answers
1. Functional	A) Relates to how well the system operates under specific conditions.	1…….
2. Performance	B) Refers to the system's ability to function correctly in different environments.	2……..
3. Power	C) Involves the amount of power consumed during operation.	3……..
4. Environmental	D) Specifies the tasks and features the system must perform.	4…….

7. **Fill in the gaps with the appropriate terms related to firmware applications and development processes referred to these: Automotive, settings, Features, Planning, Communication and interaction.**

a) The main purpose of firmware is to provide _____ between the hardware and software in a device.

b) Common applications of firmware include _____, consumer electronics, and industrial automation.

c) The first stage in the firmware development process is _____, where requirements are gathered and analyzed.

d) Firmware can be updated for improvements, bug fixes, or to add new _____ to a device.

Practical assessment

In the ABCD Company, a technician is assigned to develop firmware for available a new smart home thermostat designed and printed to automatically adjust heating and cooling based on user habits and surrounding environmental conditions. To ensure the firmware functions seamlessly with the hardware and adheres to embedded system specifications. You are requested to collecting, analyse and extract key data related to that thermostat.

Reference

Doe, J. (2020). *The Firmware Handbook.* (J. Smith, Ed.) Tech Press.

Douglass, B. P. (2016). *Embedded Systems: Design and Applications.* Amsterdam: Elsevier.

Himpe, V. (2015). *Embedded Firmware Solutions: Development Best Practices for the Internet of Things.* New York: Apress.

Ibrahim, D. (2014). *Designing Embedded Systems with 32-Bit PIC Microcontrollers and MikroC.* Oxford: Newnes.

Meer, J. A. (2018). *Designing Embedded Systems with Arduino.* Berlin: Springer.

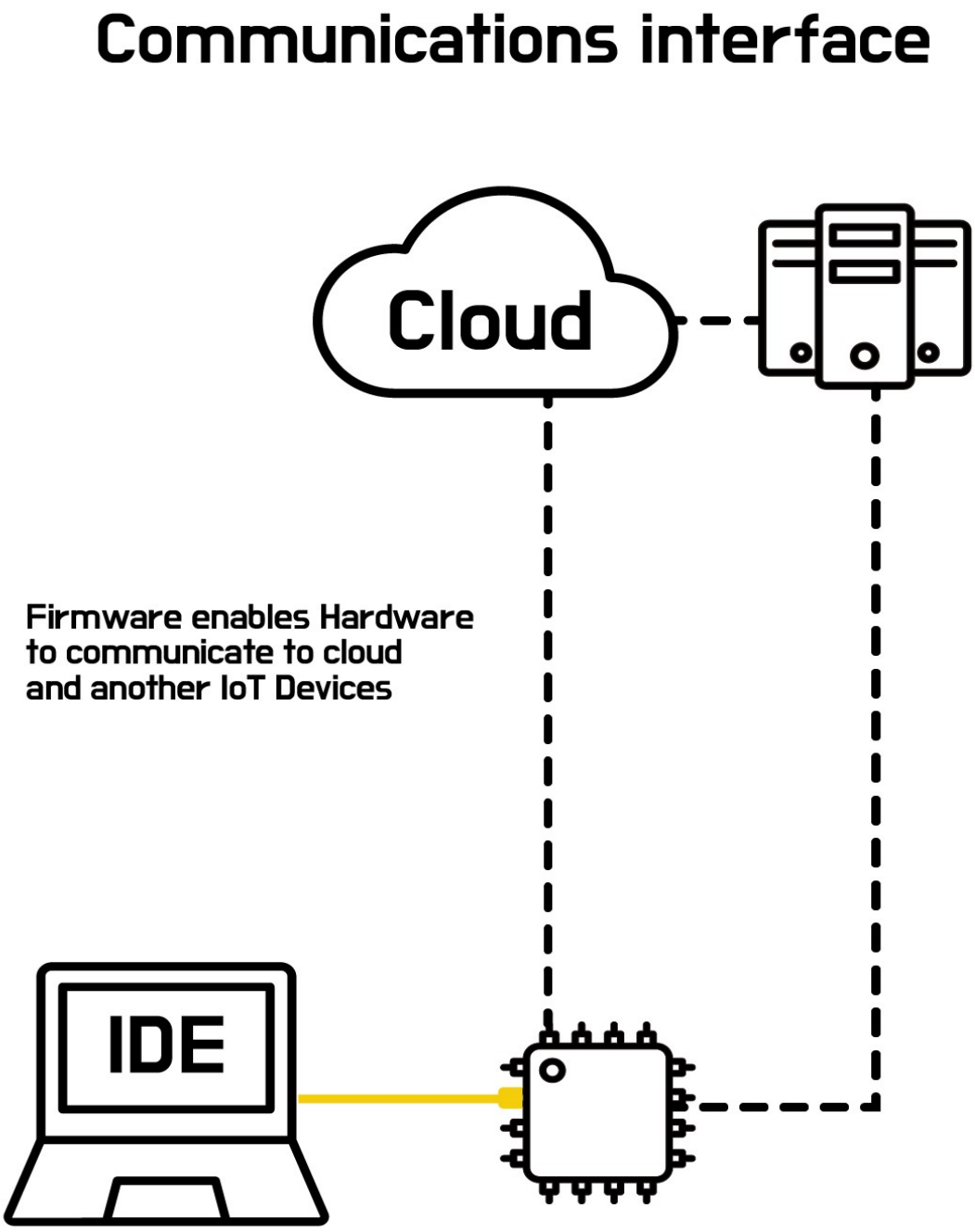
Oshana, R. (2019). *Software Engineering for Embedded Systems.* CRC Press.

Peter Barry, P. C. (2012). *Real-Time Embedded Systems: Design Principles.* Hoboken: Wiley.

Simon, D. (2021). *Firmware Development for Embedded Systems.* Springer.

Valvano, J. (2018). *Embedded Systems: Real-Time Operating Systems.* Wiley.

Wilmshurst, T. (2010). *Designing Embedded Systems with PIC Microcontrollers.* Oxford: Newnes.


Communications interface
Cloud
Firmware enables Hardware
to communicate to cloud
and another IoT Devices
IDE

Indicative Contents

2.1. Selection of Tools, Materials, and Equipment

2.2. Preparation of Drawing Environment

2.3. Drawing Firmware Architecture Diagrams

2.4. Documentation of Firmware Architecture

Key Competencies for Learning Outcome 2: Design Firmware Architecture

Knowledge	Skills	Attitudes
• Description of Tools, materials, and equipment • Description of Diagramming. • Identification of Tools, Materials, and Equipment.	• Selecting Tools, materials, and equipment. • Preparing drawing environment for Firmware Design. • Drawing Firmware architecture diagrams. • Documenting firmware architecture.	• Having Curiosity • Being Patient and Persistent. • Being Attentive to Detail • Having Adaptability • Having Collaboration • Having Critical Thinking • Being Responsible • Having Ethical Considerations • Being Self-Reliance

Duration: 10 hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Define correctly Diagramming Tool as used in firmware design.
2. Describe correctly Diagramming Tool as used in firmware design.
3. Identify effectively Tools, Materials, and Equipment as used in Firmware design.
4. Prepare appropriately drawing environment for Firmware Design as used in firmware design.

5. Draw properly Firmware architecture diagrams as used in hardware architecture.

Resources

Equipment	Tools	Materials
• Computer • Projector • White board or blackboard	• Web browsers • Text editor • Microsoft Visio • Lucidchart • Draw.io • Edraw Max • Smart Draw • Adobe Illustrator • Text Editor	• Internet • Electricity • Notebooks

Indicative content 2.1: Selection of Tools, materials, and equipment

Duration: 1 hrs

Theoretical Activity 2.1.1: Description of tools, materials and equipment used in designing firmware architecture

Tasks:

1: You are requested to answer the following questions.

i. Describe Tools used for designing firmware architecture?
ii. What do you mean by materials used in firmware architecture?
iii. Give any two examples of equipment most used for designing firmware architecture.
iv. What are key factors to consider while selecting tools, materials and equipment for designing firmware architecture?

2: Provide the answers for the asked questions.

3: Present your findings.

4: Ask for clarification if any.

5: Read the key readings 2.1.1 in trainee's manual.

Key readings 2.1.1.: Selection of tools, materials and equipment for firmware architecture designing

1. Tools

These are software applications and programs that firmware developers use to create, analyse, and manage code and designs for firmware architecture.

Examples:

- Integrated Development Environments (IDEs) like Arduino Uno, Eclipse CDT, IAR Embedded Workbench, and Keil MDK
- Version Control Systems such as Git, Subversion (SVN), and Mercurial
- UML Modelling Software including Enterprise Architect, Visual Paradigm, and StarUML
- Static Code Analysis Tools like Coverity, SonarQube, and PC-lint
- Documentation Generators such as Doxygen, Javadoc, and Sphinx

2. Materials

Physical or digital resources that are consumed, referenced, or directly used in the process of designing a firmware architecture. These items provide essential information, serve as components in the development process, or act as inputs for the tools and equipment used.

Examples

- Printed Circuit Boards (PCBs
- Microcontrollers (e.g., ARM Cortex-M series chips)
- Sensors (e.g., temperature sensors, accelerometers)
- Actuators (e.g., motors, relays)
- Passive components (resistors, capacitors, inductors)
- Connectivity Cables and Adapters
- Storage Devices (e.g., SD cards, USB flash drives, EEPROM chips, Flash memory modules)

3. Equipment

Physical hardware devices used for developing, testing, and debugging firmware and the systems it runs on.

Examples:

- Power Supplies
- Multimeters
- Oscilloscopes
- Logic Analyzers

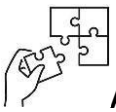
4. Factors to consider while selecting tools, materials and equipment

When selecting these tools, materials, and equipment, consider factors such as:

- Project requirements: Ensure the chosen items meet the specific needs of your firmware project.
- Budget constraints: Balance cost with features and capabilities.
- Team expertise: Choose tools that align with your team's skills and experience.
- Scalability: Select items that can accommodate future project growth and complexity.
- Industry standards: Ensure compliance with relevant standards and best practices.
- Support and documentation: Choose well-supported tools with comprehensive documentation.

Points to Remember

- Types of tools used for firmware architecture designing are: Integrated Development Environments (IDEs), Version Control Systems, UML Modelling Software, Static Code Analysis Tools.
- **Examples of materials used when designing a firmware:** Printed Circuit Boards (PCBs), Microcontrollers, Sensors, Actuators, Connectivity Cables
- **Example of Equipment used when designing firmware:** Power Supplies, Multimeters, Oscilloscopes, Logic Analysers.



Application of learning 2.1.

The ABCD Company is willing to Design a Smart Plant Monitor that measures soil moisture, light levels, and temperature, then sends alerts to a user's smartphone when the plant needs attention. As a firmware developer, you are hired to carry out this project. Select tools, materials and equipment required to accomplish the project.

Indicative content 2.2: Preparation of drawing environment.

Duration: 3 hrs

Theoretical Activity 2.2.1: Description of diagramming tool.

Tasks:

1: Introduce the activity and answer the following questions

i. What is a Diagramming Tool?
ii. List example of Diagramming Tool
iii. What are the different ways to gather complete hardware information about a microcontroller or processor?
iv. Identify the hardware components commonly involved in firmware design.

2: Provide the answers and present their findings.

3: Agreement of their findings together with trainees and trainer.

4: Follow the expert view.

5: read the Key readings 2.2.1

Key readings 2.1.1.: Description of diagramming tool

- **Definition**

  Diagramming tools are Tool and Software that allows the user to create flow charts, organization charts and similar diagrams. It is similar to a drawing program, but specialized for creating interconnected diagrams. It comes with a palette of predefined shapes and symbols, and usually keeps the lines connected between them if they are edited and rearranged.

- **Example of diagramming software.**

✓ **Edraw Max:** are software *business technical diagramming software* which helps create flowcharts, organizational charts, mind map, network diagrams, floor plans.

✓ **Lucidchart** is a web-based diagramming application that allows users to visually collaborate on drawing, revising and sharing charts and diagrams, and improve processes, systems, and organizational structures.

✓ **SmartDraw** is a web-based diagramming tool used by teams to collaborate on and make flowcharts, organization charts, mind maps, project charts, and other business visuals.

✓ **Draw.io is proprietary software** for making diagrams and charts.

- ✓ **Microsoft Visio** is software for drawing a variety of diagrams. These include flowcharts, org charts, building plans, floor plans, data flow diagrams, process flow diagrams, business process modeling, swimlane diagrams, 3D maps, and many more.

- **Identification of Symbols and Notation**

In the context of firmware, symbols represent components such as microcontrollers, memory units, I/O devices, or communication interfaces, while notations provide specific details like address ranges, data types, or port numbers. You identify these symbols and notations based on the hardware architecture and firmware design.

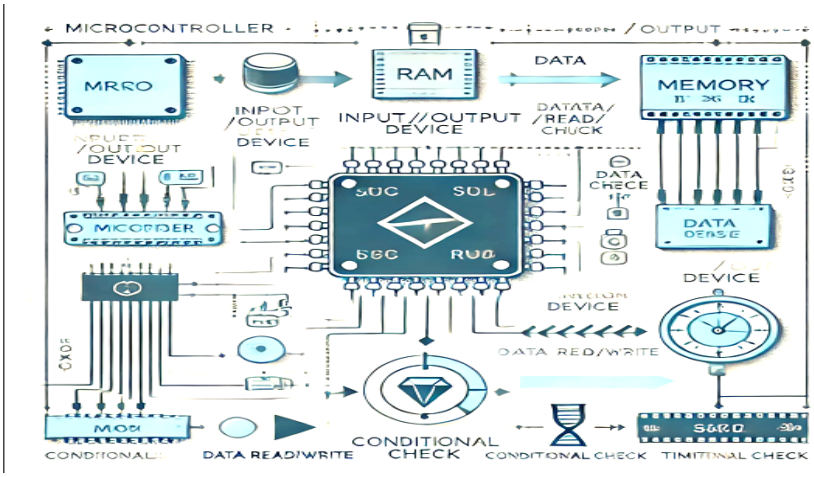
Symbols are applied to represent hardware components in the firmware diagram, such as using a rectangle for a processor or an arrow for data flow.

Here are some additional symbols that are commonly used in firmware diagrams:

- ✓ **Cylinder(())**: Represents memory storage units such as ROM, RAM, or external storage.
- ✓ **Oval()**: Used for start or stop points in flowcharts, which can indicate the beginning or end of a firmware process.
- ✓ **Diamond()**: Represents decision points, like conditional branching in firmware logic (e.g., checking if a flag is set).
- ✓ **Trapezoid ()**: Used for indicating input/output operations, such as when data is being read from or written to sensors or actuators.
- ✓ **Hexagon ()**: Sometimes used for representing communication protocols or specialized control sequences.
- ✓ **Parallel Line ()**: Can represent a bus for communication, such as SPI or I²C communication lines.
- ✓ **Clock Symbol** () : For timing or synchronization signals, especially important in real-time firmware.
- ✓ **Double Arrow (Bidirectional)**  : Represents two-way communication between components like in full-duplex serial communication.

Including these symbols along with rectangles, arrows, and parallelograms can help you create a more comprehensive and accurate representation of the firmware's architecture and interactions.

Here is the diagram illustrating the interconnection of various firmware-related components using the symbols we discussed, such as a microcontroller, memory, I/O device, decision point, and more. It visually shows how these components interact in a firmware system.



- Microcontroller: Represented by a rectangle at the center.
- Input/output Device: A parallelogram to the left of the microcontroller.
- Memory (RAM/ROM): A cylinder above the microcontroller.
- Conditional Check: A diamond symbol below the microcontroller.
- Bidirectional Communication: A double arrow to the right connecting to a sensor.
- Start of Firmware Process: An oval symbol.
- Data Read/Write: A trapezoid connected to the I/O device.
- Timing Signals: A clock symbol next to the microcontroller.

Notations are used to clarify the configuration, timing, or interactions between the firmware and hardware components, such as specifying memory addresses or bus protocols.

✓ **Gathering Complete Hardware Information**

Collecting complete hardware information is important for designing effective firmware. This helps ensure that the firmware works well with the hardware, leading to better performance.

Commonly methods used and recommended for gathering comprehensive hardware information about microcontrollers and processors:

- **Datasheets**: Review the datasheets of the microcontroller and other hardware components for detailed specifications.
- **Manufacturer Documentation**: Use application notes and technical manuals from manufacturers to understand integration and configuration.
- **Community Resources**: Leverage forums, user guides, and online communities for additional insights and real-world applications.

- **Hardware Inspection**: Physically inspect the hardware for component identification and connection layout.

For gathering Complete Hardware Information focus on Hardware Components and Microcontroller/Processor:

➢ **Microcontroller/Processor Information**

- **Type and Model**: Know which microcontroller or processor you are using (e.g., ARM Cortex-M, AVR).
- **Architecture**: Understand whether it is 8-bit, 16-bit, or 32-bit, as this affects how it processes information.
- **Clock Speed**: Check the speed at which the processor operates; this affects performance.
- **Memory Specifications**:
  - **Flash Memory**: How much space is available for storing the program?
  - **RAM**: How much memory is available for running tasks?
- **I/O Ports**: Know how many input and output ports are available and their types (digital or analog).
- **Power Requirements**: Understand the voltage and current needed to power the microcontroller, including options for low power use.
- **Peripheral Support**: Check for built-in features like timers, analog-to-digital converters (ADCs), and communication interfaces (I2C, SPI, UART).

Aspect	Microcontroller Unit(MCU)	Microprocessor Unit (MPU)
Integrated Components	Typically contains CPU, memory, I/O	CPU only, requires external components
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	Compact, smaller footprint	Larger footprint due to external components

- **Identify Hardware Components**

Designing firmware architecture involves understanding the hardware components of the system.

Steps to gather hardware component information:

- **Processor:** Determine the CPU architecture, type, speed, and capabilities.
- **Memory:** Identify RAM, ROM, Flash memory, and their sizes.
- **Peripherals:** List all components like sensors, actuators, communication modules (Wi-Fi, Bluetooth, etc.), input/output ports, etc.
- **Power Management:** Understand power sources, batteries, voltage regulators, etc.

Practical Activity 2.2.2: Installing Diagramming tool

Task:

1: Read key reading 2.2.2 and ask clarification where necessary

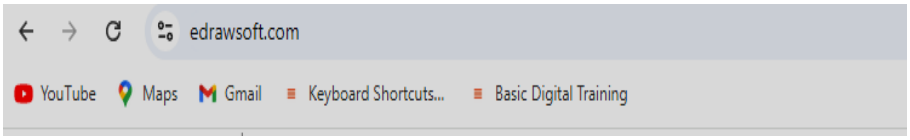
2: Referring to the steps provided in key reading 2.2.2, you are requested to go to the computer lab and install diagramming tools.

3: Present your work.

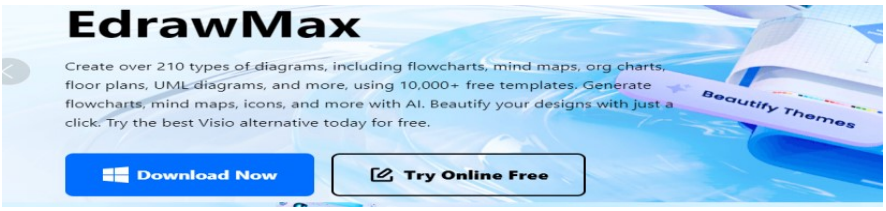
4: Perform the task provided in application of learning 2.2

Key readings 2.2.2: Installing Diagramming tool

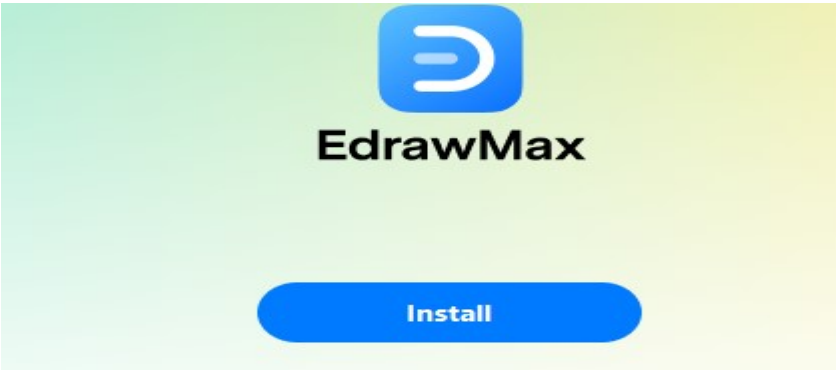
- **Diagramming installation steps :**
  1. Visit specified Diagramming tool websites (website of Edraw Max)



  2. Download the Diagramming tool from websites(Edraw Max)



3. Install Diagramming tool(Edraw Max)



4. Launch specified Diagramming tool**(**Edraw Max)

Points to Remember

- There is an example **of diagramming software** such as: Edraw Max, Lucid chart, Smart Draw, Draw IO and Microsoft Visio.
- **Step for installing diagramming software**
  1. Visit specified Diagramming tool websites (website of Edraw Max)
  2. Download the Diagramming tool (Edraw Max)
  3. Install Diagramming tool (Edraw Max)
  4. Launch specified Diagramming tool (Edraw Max).

Application of learning 2.2.

The **ABCD Company** is a computer science student who needs to install a diagramming tool on her laptop to complete her assignment on software design. You are requested to install Edraw Max so that she can perform her assignment.

Indicative content 2.3: Drawing Firmware architecture diagrams.

Duration: 3 hrs

Theoretical Activity 2.3.1: Description of Firmware architecture diagrams.

Tasks:

1: Introduce the activity and answer the following questions

Introduce the activity and answer the following questions

i. What is the definition of firmware modules?
ii. What are types of diagrams used in firmware design?
iii. What is a hardware block diagram?
iv. What are the basic components typically shown in a hardware block diagram?

2: Provide the answers and present their findings.

3: Agreement of their findings together with trainees and trainer.

4: follow the explanations of trainer.

5: Read the Key readings 2.3.1

Key readings 2.3.1.: Description of Firmware architecture diagrams

Firmware architecture diagrams

i. Firmware Modules.

Firmware modules are specific software components or programs embedded within hardware devices, serving as a bridge between the hardware and higher-level software. They are essentially a set of instructions or code that enable the hardware to function and interact with other software systems**.**

Common methods to identify firmware modules

- **Manufacturer Documentation:** Check the manufacturer's documentation or official website. Often, they provide detailed information about firmware modules, their functionalities, and sometimes even their names.
- **Device Interfaces:** Some devices provide access to firmware modules through interfaces like a command-line interface (CLI) or a web interface. Logging into these interfaces might allow you to explore and identify different modules.
- **File Analysis:** If you have access to firmware files, you can analyze them using tools like binwalk, firmware-mod-kit, or IDA Pro. These tools can help identify different modules within the firmware by analyzing file structures, headers, and code.

- **Debugging Tools:** Utilize debugging tools or firmware analysis tools like JTAG, Serial Debuggers, or specialized hardware debuggers that can provide access to the firmware running on a device. These tools might offer insights into the modules present.
- **Reverse Engineering:** This method involves examining the firmware's code or binaries to identify various modules, their functions, and interactions. It's a more advanced and complex approach that requires knowledge of reverse engineering techniques

ii. Types of Diagram

In firmware development, various types of diagrams can be used to visually represent different aspects of the firmware architecture, design, and functionality.

Types of diagram	Purpose	Usage
Block Diagrams	Block diagrams provide a high-level overview of the firmware architecture by representing major functional blocks and their interconnections.	Useful for understanding the overall structure and flow of data between different components in the firmware.
Flowcharts	Flowcharts depict the flow of control within the firmware, illustrating the sequence of operations and decision points.	Effective for representing the logic and control flow of algorithms, processes, or state machines within the firmware.
State Diagrams	State diagrams model the different states that a system or component can be in and the transitions between these states.	Useful for representing the behaviour of firmware components that exhibit state dependent behaviour, such as finite state machines.
Sequence Diagrams	Sequence diagrams illustrate the interactions and order of execution between different components or objects over time.	Helpful for visualizing the dynamic behaviour of firmware components, especially during communication or interaction scenarios.

Class Diagrams	Class diagrams depict the structure of the firmware by illustrating classes, their attributes, and relationships between them.	Class diagrams depict the structure of the firmware by illustrating classes, their attributes, and relationships between them.
Component Diagrams	Component diagrams illustrate the physical organization of software components and their dependencies.	Useful for representing the modular structure of the firmware, showing how components are connected and dependent on each other.
Deployment Diagrams	Deployment diagrams depict the physical deployment of software components on hardware devices.	used in software engineering to visualize the deployment of software components within a system and their interactions with hardware components
Communication Diagrams	Communication diagrams show how objects or components collaborate to achieve a specific functionality	Useful for visualizing the flow of messages and interactions between different components during runtime.
Timing Diagrams	Timing diagrams illustrate the timing and duration of signals, events, or actions within the firmware	Helpful for representing time sensitive aspects of firmware behaviour, especially in real-time systems.
Data Flow Diagrams (DFD):	DFDs represent the flow of data within the firmware, showing how data is processed and transformed	Useful for understanding the data-centric aspects of the firmware and how information is manipulated.

iii. Create Hardware Block Diagram

Creating a hardware block diagram within the context of firmware involves visually representing the key hardware components and their interconnections.

Here's a simplified textual representation of how you might structure a hardware block diagram, the basic components typically shown in a hardware block diagram are:

Hardware Platform:

• Represents the overall hardware environment where the firmware runs. This could be a specific embedded system or device.

Microcontroller or Processor:

The core processing unit responsible for executing the firmware code. It includes the CPU, memory, and possibly other essential components.

Peripherals & I/O:

This section includes various peripherals and input/output interfaces connected to the microcontroller. Examples include sensors (e.g., temperature sensors, accelerometers), actuators (e.g., motors), and communication modules (e.g., UART, SPI, I2C).

Power Supply Circuit:

The power supply circuit that provides the necessary power to the hardware components. This may include voltage regulators, power management components, and power sources.

This is a basic representation, and the actual hardware block diagram may vary based on the complexity and specific requirements of your firmware and hardware design.

iv. Create The Functional Flow Diagram(flowchart)

Creating a functional flow diagram (flowchart) in firmware involves representing the logical flow of operations and decision points within the firmware.

Here's a simplified textual representation of how you might structure a functional flow diagram:

- **Initialize System and Components:**

Represents the initialization phase where the firmware sets up the system and initializes various components.

- **Read Sensor Data:**

Involves reading data from sensors, such as temperature sensors. The decision-making process may depend on these sensor readings.

- **Process Sensor Data and Make Decisions:**

Describes the logic for processing the sensor data and making decisions based on the data received. This could involve control flow based on conditions.

- **Control Actuators:** Represents the stage where the firmware sends commands to actuators (e.g., motors) based on the processed data and decisions.
- **Update User Interface and Display Data:**

Involves updating the user interface and displaying relevant data to the user. This step is often important for providing feedback and information.

- **Check System State and Handle Errors:**

Checks the overall system state and includes error handling mechanisms. This step ensures that the firmware responds appropriately to unexpected conditions.

- **Marks the end of the functional flowchart.**

This is a simplified example, and the actual flowchart would depend on the specific functionalities and requirements of your firmware. Use a diagramming tool to create a visual representation of the flowchart, including appropriate symbols for processes, decisions, input/output, and connectors.

v. Represent Firmware Modules

Representing firmware modules can be done using various diagramming techniques. Below is an outline illustrating the organization of a modular firmware architecture"

- **Firmware Application:** Represents the main application module responsible for orchestrating the overall firmware functionality.
- **Communication Module:** A separate module responsible for handling communication tasks, such as sending and receiving data over communication protocols (e.g., UART, SPI, I2C).
- **Sensor Module:** Represents a module dedicated to interacting with sensors, collecting data, and passing it to other modules for processing.
- **Data Processing Module:** Focuses on processing data received from sensors or other sources. This could involve algorithms, calculations, or transformations.
- **Control Module:** Manages the control logic, making decisions based on processed data, and sending commands to actuators or other control components.
- **User Interface Module:** Handles interactions with users, including input processing and providing feedback. This module may update the user interface based on data and events from other modules. Each module has a specific responsibility, and interactions between modules are well-defined. This modular approach enhances the maintainability, scalability, and reusability of the firmware. In a graphical representation, you would use shapes and symbols to depict each module and lines/arrows to show their connections and interactions. Consider using a diagramming tool for creating a visual representation tailored to your firmware architecture.

vi. Create an architecture Diagram

Creating a firmware architecture diagram involves visually representing the key components, layers, and interactions within the firmware.

Here's a simplified textual representation of how you might structure a firmware architecture:

- **Firmware Application:**

Represents the main application layer responsible for coordinating the overall functionality of the firmware.

- **Communication Layer:**

Handles communication protocols, such as UART, SPI, or I2C. This layer facilitates the exchange of data between the firmware and external devices.

- **Data Processing Layer:**

Focuses on processing data received from various sources, which may include sensors, communication modules, or external inputs.

- **Control Logic Layer:**

Manages the control flow and decision-making processes. This layer interprets processed data and determines the appropriate actions to be taken by the firmware.

- **Hardware Abstraction Layer (HAL):**

Provides an abstraction between the firmware and the underlying hardware. It includes functions and interfaces that allow the firmware to interact with the hardware components without directly dealing with low-level details.

- **Device Drivers:**

Interface with specific hardware components, translating generic commands from the firmware into operations that the hardware understands. Each device driver corresponds to a specific peripheral or component.

In a graphical representation, you would use shapes and symbols to depict each layer and lines/arrows to show their connections and interactions

vii. Hardware Interaction

In firmware development, hardware interaction refers to the communication and coordination between the firmware (software) and the underlying hardware components. This interaction is crucial for the firmware to control and utilize the hardware resources effectively.

Here are key aspects of hardware interaction in firmware:

- **Device Initialization:**

Firmware is responsible for initializing and configuring hardware components during the system startup. This includes setting up registers, configuring communication protocols, and ensuring that the hardware is in the desired state.

- **Read and Write Operations**:

Firmware interacts with hardware by reading data from and writing data to various registers or memory locations. These operations are essential for controlling the behavior of peripherals, sensors, and other hardware components.

- **Interrupt Handling:**

Many hardware devices generate interrupts to signal specific events or conditions. Firmware must implement interrupt service routines (ISRs) to respond to these interrupts promptly. This includes handling tasks like updating variables, acknowledging interrupts, and taking appropriate actions.

- **Sensor Data Acquisition:**

Firmware often interfaces with sensors to collect data. This may involve configuring the sensors, initiating data acquisition, and processing the received data for further use.

- **Actuator Control:**

Firmware sends commands to actuators, such as motors or relays, to control physical processes. This interaction requires precise timing and coordination to achieve the desired output.

- **Communication Protocols:**

Firmware communicates with external devices or systems using various communication protocols such as UART, SPI, I2C, or others. Implementing these protocols correctly is essential for reliable data exchange.

- **Power Management:**

Firmware may be involved in managing power states and modes of the hardware. This includes putting components into low-power states when not in use and waking them up as needed.

- **Error Handling:**

Firmware needs to handle errors and exceptions that may occur during hardware interaction. This includes addressing issues like communication errors, sensor malfunctions, or unexpected hardware behaviour.

- **Hardware Abstraction:**

To enhance portability and maintainability, firmware often includes a hardware abstraction layer (HAL) that provides a standardized interface to interact with hardware. This allows the same firmware code to work across different hardware platforms.

- **Real-time Constraints**:

Some firmware applications, especially in embedded systems, have real-time constraints. Firmware must adhere to specific timing requirements to ensure timely and predictable responses to hardware events.

- **Testing and Debugging:**

Firmware developers use various tools and techniques for testing and debugging hardware interactions. This may involve simulators, emulators, or debugging hardware-specific issues using breakpoints and logging.

Successful hardware interaction in firmware requires a deep understanding of the hardware architecture, precise control over timing, and adherence to the specifications of the hardware components. Additionally, documentation plays a crucial role in ensuring that developers understand how to interact with the hardware effectively.

viii. Data Flow and Communication (Sequence Diagram)

A Sequence Diagram is a powerful tool to visually represent the interactions and flow of data between different components or objects in a system.

In the context of firmware development, a Sequence Diagram can illustrate the communication between firmware modules, hardware components, or external systems.

Here's a simplified textual representation

1. **Application**:

The main application layer initiates the communication and data flow.

2. **Communication Layer:**

This layer handles low-level communication protocols (e.g., UART, SPI) and manages the exchange of data between the firmware and external devices.

3. **Data Processing Module:**

The data processing module receives raw data, processes it, and generates processed

Practical Activity 2.3.2: Drawing Firmware architecture diagrams.

Task:

1: Read key reading 2.3.2 and ask clarification where necessary

2: Referring to the steps provided in key reading 2.3.2, you are requested to go to the computer lab and design the firmware Diagram architecture.

3: Present your work to the trainer and whole class.

4: Perform the task provided in application of learning 2.3

Key readings 2.3.2 Drawing firmware architecture diagrams

Step of Drawing Firmware architecture diagrams (by using Edraw Max):

Step 1: Start Your Computer

1. **Turn on your computer** and log in.
2. Ensure **EdrawMax Desktop** is installed. If not, download and install it from the official website.

Step 2: Open EdrawMax

1. **Open EdrawMax** from your desktop or start menu.
2. **Work offline/online** by opening the application without/with an internet connection.

Step 3: Create a New Diagram

1. In the **New** tab on the left panel, **choose a diagram type**.
2. Since firmware diagrams often include flowcharts, processes, or data flows, select **"Software & Database"** or **"Flowchart"** from the diagram categories.

3. Choose a **blank drawing** or use a template that best fits the firmware diagram you want to create.

Step 4: Add Shapes and Elements for Hardware and Software Components

1. **Drag and drop shapes** from the left panel (e.g., rectangles, ellipses, or specific shapes related to firmware like microchips, controllers, etc.).
2. **Label each shape** for clarity. For example:
    - Use rectangles to represent **hardware components** like processors, memory units, or sensors.
    - Use shapes like ovals or cylinders for **software components** such as drivers, bootloaders, or real-time operating systems (RTOS).

Step 5: Create Layers to Separate Hardware and Software

1. **Use layers** to differentiate between hardware and software. This makes the diagram easier to understand.
    - Go to the **View** tab and click **Layers** to add a new layer for hardware and another for software.
    - Assign hardware components to one layer and software components to another.

Step 6: Connect the Components

1. **Select a shape**, and you will see small blue handles (connection points) around it.
2. **Drag and connect these points** to another shape to indicate the flow of data, control signals, or communication between hardware and software components.
    - For example, connect a **microcontroller** to a **sensor** and then to a **firmware module** that controls the sensor's data.

Step 7: Customize Connectors

1. **Click on the connectors** to customize their appearance.
    - Change the connector style (straight, curved, or angled) using the top toolbar.
    - Add **arrows** to indicate the direction of data flow.
    - Use different colours to differentiate between hardware and software communication paths.

Step 8: Add Text and Annotations

1. **Double-click any shape** to add text. For example:
    - Label each component clearly (e.g., "Microcontroller", "UART Driver", "Sensor", etc.).
    - Use **annotations** or **callouts** (found under the “Insert” tab) to describe specific processes, like firmware updates, data transmission, or power management.

Step 9: Organize the Diagram

1. **Align the shapes** neatly using the alignment tools in the **Home** tab (Align Left, Align Right, Center, etc.).
2. **Group related elements**: For example, group hardware components separately from software components by selecting the shapes and using the **Group** option.

Step 10: Add Additional Symbols

1. **Access the symbol library** for specialized symbols. Go to the left panel, select **Symbols**, and search for terms like "chip", "microcontroller", "firmware" to find relevant icons.
2. **Drag and drop these symbols** into your diagram for more accurate representation of specific components.

Step 11: Save and Export the Diagram

1. Once you're satisfied with the diagram, **save your work** by clicking **File > Save** or using the shortcut Ctrl+S (Windows) / Cmd+S (macOS).
2. **Export your firmware diagram** in various formats such as **PNG, JPEG, PDF**, or **SVG** by going to **File > Export**.

Step 12: Review and Share

I. **Review your diagram** to ensure it clearly represents the interaction between hardware and software components in the firmware.

II. **Share the diagram** as needed by saving it in a sharable format like PDF or image and distributing it via email or cloud storage.

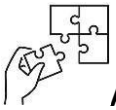
Key Elements for Firmware Diagrams:

- **Hardware Components**: Microcontroller, processors, sensors, actuators, memory, communication interfaces (UART, I2C, etc.).
- **Software Components**: Firmware, drivers, bootloaders, real-time operating system (RTOS), middleware, applications.
- **Data Flow and Control Signals**: Arrows to indicate the flow of data, interrupts, control signals, or power management interactions.

Points to Remember

- **Firmware Modules** are specific software components or programs embedded within hardware devices, serving as a bridge between the hardware and higher-level software
- **Common methods** to identify firmware modules are Manufacturer Documentation, Device Interfaces, File Analysis, Debugging Tools and Reverse Engineering.

- **Types of Diagrams** are Block Diagrams, Flowcharts, State Diagrams, Sequence Diagrams, Class Diagrams, Component Diagrams, Deployment Diagrams and Data Flow Diagrams (DFD).
- **Step of Drawing Firmware architecture diagrams (by using Edraw Max):**
    1. Start Your Computer
    2. Open EdrawMax
    3. Create a New Diagram
    4. Add Shapes and Elements for Hardware and Software Components
    5. Create Layers to Separate Hardware and Software
    6. Connect the Components
    7. Customize Connectors
    8. Add Text and Annotations
    9. Organize the Diagram
    10. Add Additional Symbols
    11. Save and Export the Diagram
    12. Review and Share



Application of learning 2.3.

ABC company is designing the firmware architecture for their environmental monitori ng system. The architecture should include modules for sensor data acquisition, data p rocessing and analysis**,** user interface management**,** and alert handling**.** The firmware s hould be modular, scalable**,** and efficient to ensure reliable and responsive operation. Additionally, the architecture should consider future enhancements such as connectivi ty options and integration with other home automation systems**.**

Note: You are required to use Diagramming tools to Design the firmware architecture Diagram for their environmental monitoring system**.**

Indicative content 2.4: Documentation of firmware architecture

Duration: 3 hrs

Theoretical Activity 2.4.1: Description of firmware architecture documentation

Tasks:

1: You are requested to answer the following questions.

i. What do you understand by documentation of firmware architecture?
ii. What is the purpose of firmware documentation?
iii. List key components of firmware documentation.
iv. Give documentation formats
v. Which tools used for documentation
vi. What are best practices for documentation
vii. Give any two challenges associated with making a documentation
viii. What to do for addressing security concern while documenting firmware architecture
ix. What to do in regard to compliance and regulations while preparing documentation.

2: Provide the answers for the asked questions and write them on flipchart/paper.
3: Present your findings to the trainer or your colleagues.
4: Ask for clarification if any.
5: Read the key readings 2.4.1 in trainee's manual.
6: Ask to trainer for clarification if any.

Key readings 2.4.1.: Documentation of firmware architecture

a. Definition

Documenting firmware architecture is the process of creating a comprehensive and structured record of a firmware system's design, components, and interactions.

b. Purpose of documentation

Documentation:

- Serves as a reference for developers and stakeholders
- Facilitates maintenance and future updates
- Aids in onboarding new team members
- Supports troubleshooting and debugging efforts

c. Key components to document

i. **System overview**: System overview provide a High-level description of the firmware's purpose and functionality as well as major subsystems and their interactions
ii. **Hardware abstraction layer**: Documentation on Interface between firmware and hardware as well as details on hardware-specific implementations
iii. **Software modules:** Description of each module's function and responsibilities along with interfaces and dependencies between modules
iv. **Data flow**: Diagrams showing how data moves through the system and key data structures and their purposes
v. **State machines:** Description of system states, transitions and condition triggering state changes
vi. **Memory map**: Layout of program memory, data memory, and peripherals together with memory allocation for different components
vii. **Bootloader**: Boot sequence, initialization process and any security measures during boot
viii. **Interrupt handlers**: List of interrupts and their handlers Priorities as well as potential conflicts
ix. **Communication protocols**: Details of internal and external communication methods, protocol specifications and implementations

d. Documentation formats

i. Textual descriptions
ii. Flowcharts and block diagrams
iii. UML diagrams (e.g., class diagrams, sequence diagrams)
iv. Comments within the source code

5. Tools used for documentation

i. Specialized software architecture tools (e.g., Enterprise Architect)
ii. Diagramming tools (e.g., Draw.io, Visio)
iii. Documentation generators (e.g., Doxygen for code documentation)

6. Best practices

i. Keep documentation up-to-date with code changes
ii. Use clear, concise language
iii. Include version history and changelog
iv. Provide examples and use cases where appropriate
v. Ensure accessibility to all team members

7. Challenges

i. Balancing detail with readability

ii. Maintaining documentation alongside rapid development
iii. Ensuring consistency across different documents

8. **Security considerations**
   i. Protect sensitive information in documentation
   ii. Document security features and potential vulnerabilities
9. **Compliance and standards**
   i. Adhere to industry-specific documentation standards
   ii. Include information relevant for certification processes

Practical Activity 2.4.2: Documenting firmware architecture.

Task:

1: Read key reading 2.4.2 and ask clarification where necessary
2: Referring to the steps provided in key reading 2.4.2, you are requested to go to the computer lab and design the firmware Diagram architecture.
3: Present your work to the trainer and whole class.
4: Perform the task provided in application of learning 2.4

Key readings 2.4.2: Documenting firmware architecture

Steps to document firmware architecture

1.Define the Purpose and Scope:

Clearly outline the objectives of the firmware and its intended functionality. Specify the scope of the document, including which components and systems will be covered.

2.Identify System Requirements:

Gather both functional and non-functional requirements for the firmware. Document hardware specifications, including microcontrollers, sensors, and peripherals that will be used.

3.Create a High-Level Overview:

Develop a block diagram or flowchart that provides a visual representation of the system architecture, highlighting major components and their interactions. Include key modules such as input handling, processing logic, and output control.

4.Detail Component Descriptions:

For each component/module, provide detailed descriptions, including:

Functionality: What each module does.

Interfaces: Communication methods with other modules (e.g., GPIO, I2C, UART).

Data Flow: How data is processed and transferred between components.

Document Control Flow:

Outline the main control flow of the firmware, describing:

Initialization procedures.

Main execution loop.

Event handling (e.g., interrupts or polling mechanisms).

5.Define State Management:

Identify and describe the different states of the system (e.g., idle, active, error).

Use state diagrams or flowcharts to illustrate transitions between states based on inputs or events.

6.Outline Timing Requirements:

Document timing constraints for the firmware, including:

Response times for inputs and outputs.

Timing for state transitions or periodic tasks.

Debounce times for inputs, if applicable.

7.Error Handling Mechanisms:

Describe how the firmware will handle errors or unexpected conditions, including:

Fault detection and reporting.

Recovery mechanisms (e.g., retries, fallbacks).

8.Testing and Validation Strategies:

Outline the testing strategies that will be used to validate the firmware, such as:

Unit tests for individual components.

Integration tests to ensure components work together.

Performance testing under various conditions.

9.Version Control and Change Management:

Establish a version control system for the firmware documentation.

Document how changes will be tracked and communicated throughout the development process.

10.Review and Validation:

Collaborate with stakeholders to review the architecture document.

Validate that the documented architecture meets all specified requirements and is feasible for implementation.

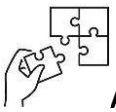
11.Finalize Documentation:

Compile all information into a clear, organized format.

Ensure accessibility for all team members and stakeholders involved in the firmware development.

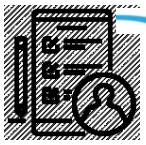
Points to Remember

- **Documenting firmware architecture** is the process of creating a comprehensive and structured record of a firmware system's design, components, and interactions.
- **Purpose of documentation include, serving** as a reference for developers and stakeholders and facilitating maintenance and future update.
- **Key components to document incorporate** System overview, Hardware abstraction layer, Software modules, Data flow, State machines, Interrupt handlers and Communication protocols.
- **Tools for documentation are the following:** Specialized software architecture tools, Diagramming tools and Documentation generators
- **Best practices include** Keep documentation up-to-date with code change, Use clear, concise language and including version history
- **Security considerations incorporates:** Protecting sensitive information in documentation as well as Documenting security features and potential vulnerabilities
- **Steps to document firmware architecture**
    1. Define the Purpose and Scope
    2. Identify System Requirements
    3. Create a High-Level Overview
    4. Detail Component Descriptions
    5. Define State Management
    6. Outline Timing Requirements
    7. Error Handling Mechanisms
    8. Testing and Validation Strategies
    9. Version Control and Change Management
    10. Review and Validation
    11. Finalize Documentation



Application of learning 2.4.

The ABCD Company is a start-up Enterprise that is working on a project of an LED blink system that provides visual feedback in response to user actions, specifically button presses with capability to detect various types of buttons presses and hence providing immediate and appropriate visual feedback through LED blinks. Suppose you are the one who has implemented the firmware for this embedded system. Make documentation for LED blink system firmware architecture.



Learning outcome 2 end assessment

Written assessment

I. Answer **TRUE** for a correct statement and **FALSE** for wrong statement.

a. The selection of tools for firmware development includes both software and hardware tools.

b. The installation of a drawing tool is necessary only for creating hardware block diagrams, not for other firmware architecture diagrams.

c. In a hardware block diagram, it's not necessary to include external components that interact with the microcontroller**.**

d. Hardware interaction in firmware architecture refers only to how the firmware controls the hardware, not how it responds to hardware events.

e. Drawing templates in firmware architecture design can help maintain consistency across different diagrams and projects.

f. Applying a colour scheme in firmware architecture diagrams is purely for aesthetic purposes and doesn't affect the diagram's readability**.**

g. The functional flow diagram and the flowchart are two distinct types of diagrams used in firmware architecture**.**

h. It's unnecessary to document the memory map in firmware architecture documentation.

i. Interrupt handlers and their priorities should be clearly documented in firmware architecture.

j. Documentation of firmware architecture should exclude details about hardware interfaces.

k. Version control information is an essential part of firmware architecture documentation.

l. Firmware architecture documentation should describe the boot sequence and initialization process.

m. Power management strategies are irrelevant in firmware architecture documentation.

n. Firmware architecture documentation should include information about real-time constraints and timing requirements.

II. Choose the letter corresponding to the correct answer

1. Which of the following is NOT typically considered a material in firmware development?

a) Development boards
b) Sensors
c) Integrated Development Environment (IDE)
d) Cables and connectors

2. Which symbol is commonly used to represent a decision point in a flowchart?

a) Rectangle
b) Diamond
c) Circle
d) Arrow

3. Which of the following is NOT typically a type of diagram used in firmware architecture?

a) Hardware Block Diagram
b) Functional Flow Diagram
c) Sequence Diagram
d) Gantt Chart

4. Which of the following best describes a firmware module?

a) A physical component on the PCB
b) A self-contained unit of code that performs a specific function
c) A type of microcontroller
d) A hardware debugging tool

5. Which of the following is NOT typically included in the documentation of firmware architecture?

a) Description of firmware modules
b) Explanation of design decisions
c) Detailed code implementations
d) Overview of communication protocols

6. Which diagram type is best suited for representing the overall structure and organization of the firmware?

a) Flowchart
b) Sequence Diagram
c) Architecture Diagram
d) Circuit Diagram

7. Which of the following is NOT a common consideration when selecting tools for firmware development?

a) Compatibility with the target hardware
b) Debugging capabilities
c) Graphic design features
d) Programming language support

III. Complete the sentence by a convenient word or expression

a) When gathering hardware information, the _____ specifications are crucial as they determine the available memory, processing power, and peripherals for the firmware.

b) A _____ diagram shows the sequence of interactions between different components or modules in the firmware.

c) In firmware development, _____ are used to simulate and test the behaviour of the firmware before deploying it to actual hardware.

d) In firmware architecture, _____ diagrams are used to illustrate how data is passed between different modules or components of the system.

e) The _____ diagram provides a high-level view of the system's software components and their relationships.

f) _____ is essential equipment for debugging and testing firmware, allowing developers to observe signals and voltages in the hardware.

Practical assessment

Smart Light Inc. is developing a basic smart light bulb that can be controlled via a smartphone app. The bulb will offer remote control capabilities, adjustable brightness and colour, and basic scheduling functionality, enhancing home lighting experiences while promoting energy efficiency. The light bulb needs to have the following features:

- Turn on/off
- Adjust brightness
- Change colour (RGB)
- Connect to Wi-Fi for remote control
- Implement a simple scheduling feature (turn on/off at specific times)

As a firmware technician, you are tasked to Design Firmware Architecture based on above requirements.

Reference

Doe, J. (2020). *The Firmware Handbook.* (J. Smith, Ed.) Tech Press.

Douglass, B. P. (2016). *Embedded Systems: Design and Applications.* Amsterdam: Elsevier.

Himpe, V. (2015). *Embedded Firmware Solutions: Development Best Practices for the Internet of Things.* New York: Apress.

Ibrahim, D. (2014). *Designing Embedded Systems with 32-Bit PIC Microcontrollers and MikroC.* Oxford: Newnes.

Meer, J. A. (2018). *Designing Embedded Systems with Arduino.* Berlin: Springer.

Oshana, R. (2019). *Software Engineering for Embedded Systems.* CRC Press.

Peter Barry, P. C. (2012). *Real-Time Embedded Systems: Design Principles.* Hoboken: Wiley.

Simon, D. (2021). *Firmware Development for Embedded Systems.* Springer.

Valvano, J. (2018). *Embedded Systems: Real-Time Operating Systems.* Wiley.

Wilmshurst, T. (2010). *Designing Embedded Systems with PIC Microcontrollers.* Oxford: Newnes.

Indicative Contents

3.1 Identification of Programming Languages Used for Firmware Development

3.2 Preparation of Development Environment

3.3 Selection of Communication Protocols

3.4 Identification of Default Segments of Data Memory

3.5 Performance of Memory Management in Embedded C

3.6 Description of Concepts for Developing Firmware

3.7 Implementation Firmware Modules

3.8 Writing Drivers Source Code

Key Competencies for Learning Outcome 3: Implement Firmware System Design

Knowledge	Skills	Attitudes
• Identification of programming Languages • Description of firmware development Environment • Identification of default segments of data memory. • Description of embedded C Programming concepts. • Description of Application of Application Programming Interfaces (APIs. • Description of Hardware Abstraction Layers (HALs) fundamentals. • Identification of HAL design process.	• Selecting communication protocols • Performing memory management using Embedded C programming • Implementing firmware modules. • Implementing drivers' source code.	• Having Curiosity • Being Patient and Persistent. • Being Attentive to Detail • Having Adaptability • Having Collaboration • Having Critical Thinking • Being Responsible • Having Ethical Considerations • Being Self-Reliance

Duration: 40 hrs

Learning outcome 3 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Identify clearly programming Languages based on firmware development.
2. Prepare effectively firmware development environment based on firmware development.
3. Select appropriately communication protocols based on firmware development.
4. Identify correctly default segments of data memory based on firmware development.
5. Perform effectively memory management based on Embedded C programming.
6. Describe clearly embedded C Programming concepts based on firmware development
7. Describe clearly Application of Application Programming Interfaces (APIs) based on firmware development.
8. Describe clearly Hardware Abstraction Layers (HALs) fundamentals based on firmware development.
9. Identify effectively HAL design process based on firmware development.
10. Implement effectively firmware modules based on firmware development.
11. Implement effectively writing drivers' source code based on firmware development.

Resources

Equipment	Tools	Materials
• Computer • White Board	• Microsoft Visio • Text Editors • Arduino IDE	• Internet • Notebooks

Indicative content 3.1: Identification of programming languages used for firmware development

Duration: 5hrs

Theoretical Activity 3.1.1: Description of programming languages used in firmware development

Tasks:

1: You are requested to answer the following questions:

i. List the types of programming languages
ii. Give most used programming language used for firmware development
iii. Enumerate advantages and disadvantages for each programming language.
iv. Compare the most used programming languages for firmware development.

2: Provide the answers for the asked questions.

3: Present your findings to the trainer or your colleagues.

4: Ask for clarification if any.

5: Read the key readings 3.1.1 in trainee's manual.

Key readings 3.1.1.: Description of programming languages used for firmware development

1. **Types of Programming Languages for Firmware Development**
   I. **Low-level languages:** These languages provide direct hardware control and are often used for firmware development due to their efficiency and close-to-hardware nature.

Examples

a) Assembly
b) C
c) Forth
d) B
e) PL/M (Programming Language for Microcomputers)

Advantages

- Direct hardware control
- Efficient resource usage
- Small memory footprint

Disadvantages

- Steep learning curve
- Time-consuming development

- Platform-dependent code

II. **Mid-level languages:** These languages offer a balance between low-level control and higher-level abstractions, making them popular for firmware development.

Examples

a) C++

b) Rust

Advantages:

- Balance between control and abstraction
- Object-oriented programming support
- Better code organization

Disadvantages:

- Potentially larger memory footprint
- May introduce overhead in resource-constrained systems

III. **High-level languages:** While less common for firmware, these languages are sometimes used for rapid prototyping or in specific scenarios where their features outweigh performance concerns.

Examples

a) Python

b) JavaScript

c) Lua

d) Java

e) C#

Advantages:

- Rapid prototyping
- Easier to read and maintain
- Rich ecosystem of libraries

Disadvantages:

- Less efficient resource usage
- Limited direct hardware control
- May require more powerful hardware

IV. **Domain-specific languages (DSLs):** These languages are designed for specific types of firmware or hardware configurations.

Examples

a) VHDL (for FPGAs and ASICs)

b) Verilog (for FPGAs and ASICs)

c) System Verilog (extended version of Verilog)

d) P4 (for programmable network devices)

e) LabVIEW (for test and measurement systems)

Advantages:

- Optimized for specific hardware or tasks
- Can improve productivity in their domain

Disadvantages:

- Limited applicability outside their domain
- Smaller developer community

2. Most used programming Languages

i. Python is an understood high-level programming language for general-purpos programming. is a computer programming language often used to build website and software, automate tasks, and conduct data analysis.

Advantages	**Disadvantages**
Extensive libraries	Speed limitations
Improved productivity	Undeveloped database access layer
Free and open source	Design restrictions

ii. C

C is a general-purpose programming language created by Dennis Ritchie at the Be Laboratories in 1972. It is a very popular language, despite being old. The main reaso for its popularity is because it is a fundamental language in the field of computer science

Advantages	**Disadvantages**
C programming language is a build 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

iii. C++

is an object-oriented programming (OOP) language that is viewed by many as the bes language for creating large-scale applications? C++ is a superset of the C language. related programming language, Java, is based on C++ but optimized for the distributio of program objects in a network **such as the internet.**

Advantages	**Disadvantages**
Portability allows developing programs irrespective of hardware	When C++ used for web applications it is 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	It has no security

iv. 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 prototyping development, it will do a ton of things for you	C# is less flexible than C++.C# depends greatly on .NET framework.
Automatic garbage collection	NET application needs a window platform to execute
Strong memory backup	C# is slower to run

v. **Rust**

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

3. 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.

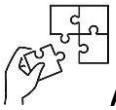
The choice of language depends on various factors such as hardware constraints, performance requirements, development team expertise, and the specific application needs.

- **Language variety:** Several programming languages are used for firmware development, including Python, C, C++, C#, and Rust. Each has its own set of advantages and disadvantages.

- **Low-level vs. high-level trade-offs:** Low-level languages like C and Assembly offer better performance and closer hardware control, while high-level languages like Python provide easier development but may sacrifice speed.
- **C/C++ prominence**: C and C++ are widely used in firmware development due to their efficiency, portability, and extensive community support. However, they lack some modern features like built-in memory safety.
- **Emerging alternatives**: Newer languages like Rust are gaining attention in firmware development, offering strong memory safety guarantees while maintaining performance close to C/C++.

Points to Remember

- Firmware development languages range from low-level to high-level with trade-offs between hardware control and ease of use.
- C and C++ dominate firmware development due to their efficiency and extensive support.
- Language choice depends on hardware constraints
- Each language has unique advantages and disadvantages



Application of learning 3.1.

Nowadays buildings use advanced technologies including smart door lock. As a firmware developer what would be your recommendation in regard to the programming language to be used to develop firmware of smart door lock and why?

Indicative content 3.2: Preparation of Development Environment

Duration: 5 hrs

Theoretical Activity 3.2.1: Description of Integrated Development Environment

Tasks:

1: You are requested to answer the following questions.

i. What do you mean by Integrated Development Environment in context of firmware development?
ii. Explain the types of IDEs
iii. What are features commonly found in IDEs?

2: Provide the answers for the asked questions.

3: Present your findings to the trainer or your colleagues.

4: Ask for clarification if any.

5: Read the key readings 3.1.1 in trainee's manual.

6: Ask to trainer for clarification if any.

Key readings 3.2.1: Identification of Integrated Development Environment IDE

1.Definition

An Integrated Development Environment (IDE) for firmware development is a software application that provides comprehensive facilities to programmers for embedded software development.

2.Types of IDEs

General-Purpose IDEs: These support multiple programming languages and offer a wide range of tools for various types of software development. Examples include Visual Studio, IntelliJ IDEA, Eclipse and Arduino.

Language-Specific IDEs: These are tailored to support specific programming languages or technologies. For instance, PyCharm is dedicated to Python development, Android Studio for Android app development, and Xcode for iOS/macOS development.

Web Development IDEs: Focused on web technologies like HTML, CSS, JavaScript, and related frameworks, these IDEs include tools for web development, debugging, and deployment. Examples are Visual Studio Code, Atom, and Sublime Text.

3.Features commonly found in IDEs:

Code Editor: Syntax highlighting, auto-indentation, code completion, and

refactoring tools.

Debugger: Allows developers to debug their code by setting breakpoints, examining variables, and stepping through code execution.

Compiler/Interpreter: Translates source code into machine code (for compiled languages) or interprets code (for scripting languages) to execute programs.

Version Control Integration: Support for version control systems like Git, allowing collaboration and tracking changes.

Build Automation: Tools for compiling, building, and packaging applications.

Project Management: Facilities to manage project files, dependencies, and libraries.

Plugins and Extensions: Extend functionality by adding plugins for additional languages, tools, or features.

Points to Remember

Practical Activity 3.2.2: Installing Arduino IDE

Task:

1: You are requested to perform the following activity:

Activity**:** Install Arduino IDE on your computer.

2: Read key reading 3.2.2

3: Present your work to the trainer and the whole class

4: Ask question if any.

Key reading: 3.2.2.: Installing IDE

Steps for installing Arduino IDE

1. Go to the official IDE website.
2. Download IDE software.
3. Run the installer.
4. Follow the installation wizard, accepting the license agreement and default options.
5. Allow the installer to install device driver software.
6. Launch the IDE.

Practical Activity 3.2.3: Configuring IDE

Task:

1: You are requested to perform the following activity:

Activity: Configure Arduino IDE on your computer.

2: Read key reading 3.2.3

3: Present your work to the trainer and the whole class

4: Ask question if any.

Key reading: 3.2.2.: Configuring IDE

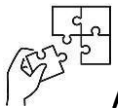
Steps for configuring IDE

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

Points to Remember

- **An IDE** is a software application for comprehensive firmware development.
- **Types of IDEs** are General-Purpose, Language-Specific and Web Development IDEs.
- **Common IDE Features include** Code Editor**,** Debugger**,** Compiler/Interpreter, Version Control Integration, Build Automation, Project Management, Plugins and Extensions.
- **Steps for IDE installation**
  1. Go to the official IDE website.
  2. Download IDE software.
  3. Run the installer.
  4. Follow the installation wizard, accepting the license agreement and default options.
  5. Allow the installer to install device driver software.
  6. Launch the IDE.
- **Steps for IDE Configuration**

1. Setting Preferences
2. Adding Plugins
3. Setting up Tools
4. Project Configuration



Application of learning 3.2.

You're a hobbyist starting your first Arduino project. Your task is to set up the Arduino Integrated Development Environment (IDE) on your Windows 10 computer, configure it for use with an Arduino Uno board, and prepare it for your first project. You need to install the Arduino IDE, properly configure it and verify the setup by uploading a simple program that blinks an LED.

Indicative content 3.3: Selection of communication protocols

Duration: 5hrs

Theoretical Activity 3.3.1: Description of communication protocols

Tasks:

1: You are requested to answer the following questions:

i. What do you mean by protocol?
ii. What is communication protocol?
iii. Give two types of Communication Protocols in Embedded Systems
iv. Discuss advantages and disadvantages of each type of communication protocols.
v. What is the importance of communication protocol?
vi. Explain: USB, SPI, I2C, UART

2: Provide the answers for the asked questions.
3: Present your findings to the trainer or your colleagues.
4: Ask for clarification if any.
5: Read the key readings 3.3.1 in trainee's manual.
6: Ask to trainer for clarification if any.

Key readings 3.3.1.: Description of communication protocols

a. Definitions

- **Protocol**: A set of rules and regulations is called a protocol.
- **Communication Protocol**: A set of rules and regulations that allow two electronic devices to connect to exchange the data with one and another.
- **Communication protocols in firmware** refer to a set of rules and conventions that define the format and sequence of messages exchanged between devices or systems. These protocols establish guidelines for data transmission, error detection, correction, and overall system communication**.**

Fig communication protocols

b. **Importance of Communication Protocols**

- ✓ **Interoperability:** Different systems can communicate effectively if they adhere to the same protocols.
- ✓ **Reliability:** Well-defined protocols ensure data integrity and minimize errors during transmission.
- ✓ **Efficiency:** Protocols help optimize data exchange, reducing latency and improving overall system performance.
- ✓ **Standardization:** Protocols provide a standard framework for communication, facilitating compatibility among different devices and manufacturers.

c. **Types of communication protocols in embedded systems**

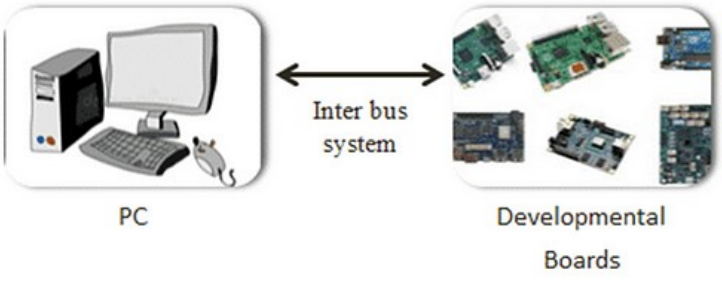
Communication protocols are broadly classified into two types:

Inter System Protocol

Intra System Protocol

- ✓ **Inter System Communication Protocols**

Inter system protocols establish communication between two communicating devices i.e. between PC and microprocessor kit, developmental boards, etc. In this case, the communication is achieved through inter bus system.

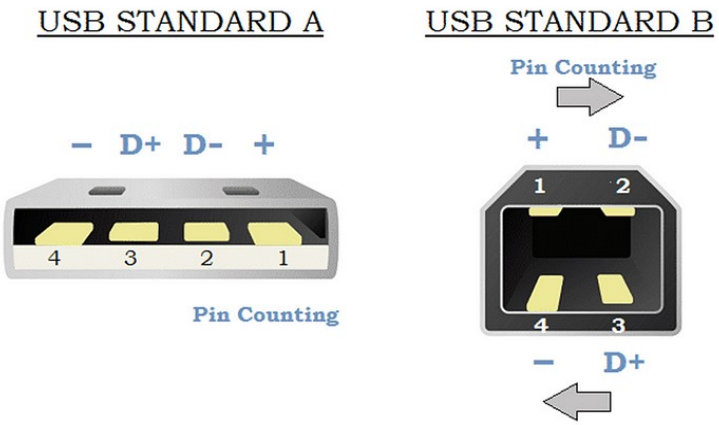


- ✓ **Types of Inter System Communication Protocols**
  - USB Communication protocols
  - UART Communication protocols
  - USART Communication protocols

USB Communication Protocols

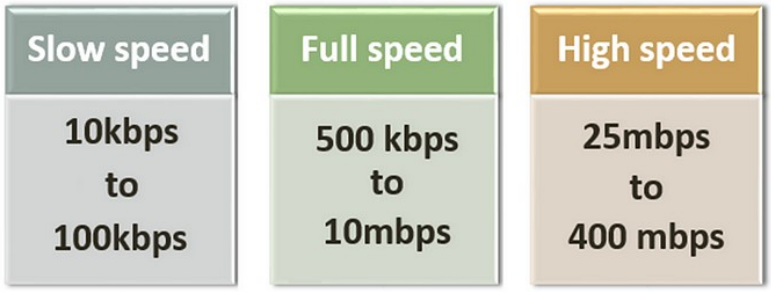
Universal **S**erial **B**us (USB) is a two-wired serial communication protocol. It allows 127 devices to be connected at any given time. USB supports plug & play functionality. USB protocol sends and receives the data serially between host and external peripheral devices through data signal lines D+ and D-. Apart from two data lines, USB has VCC and Ground signals to power up the device.

The USB pin out is shown in Figure



Data is transmitted in the form of packets where two devices communicate each other. Data packets compose of 8 bits (byte) with LSB (Least Significant Bit) transmitted first. USB associates NRZI (Non-Return to Zero Invert) encoding scheme to transmit data with sync field to synchronize the host system and receiver clock signals.

In USB, data is transferred in three different speeds such as:



Advantages of USB Communication Protocol

Fast and simple.

It is of low cost.

Plug and Play hardware.

Disadvantages of USB Communication Protocol

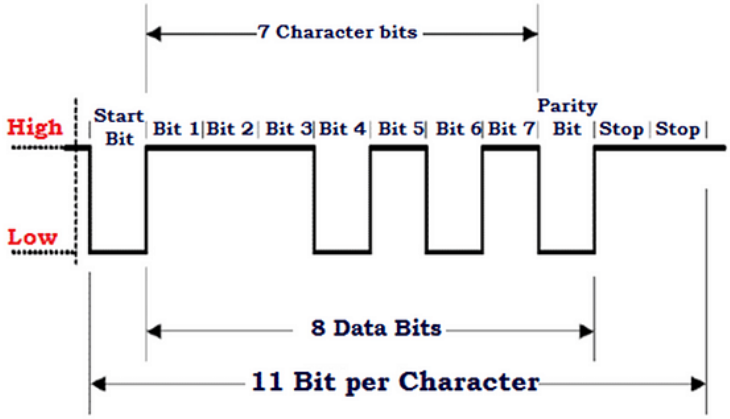
Needs powerful master device.

Specific drivers are required.

Universal Asynchronous Receiver/Transmitter UART Communication Protocols

The **Universal Asynchronous Receiver/Transmitter (UART)** is a hardware communication protocol used for asynchronous serial communication between devices. It facilitates the transfer of data between two devices (typically a microcontroller, computer, or peripheral) over a serial interface, where the timing of the data bits is managed by the transmitter and receiver independently.

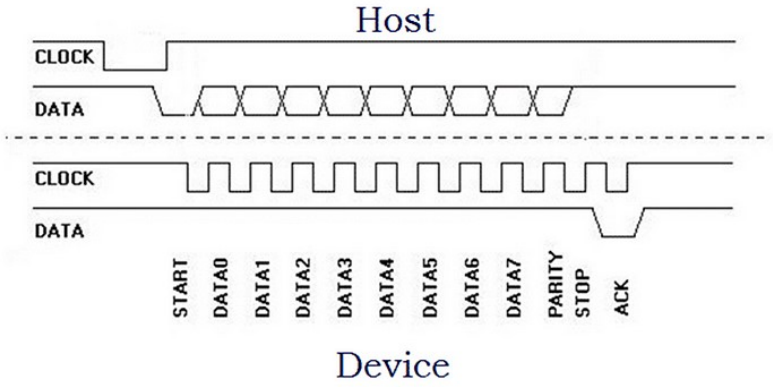
Its main purpose is to transmit and receive data serially. UART transmits data asynchronously. When receiver end detects the start bit, it starts to read the data bits at specific baud rate. UART works under half duplex communication mode meaning it either transmits or receives at a time.



Example: Emails, SMS.

USART Communication Protocol

Universal **S**ynchronous **A**synchronous **R**eceiver/**T**ransmitter (USART) is identical to that of UART with only added functionality synchronous. USART encompass the abilities of UART, which enables application of both depending on the applications area.



Advantages of UART/ USART Communication Protocol

Clock signal is not required

Cost effective

Uses parity bit for error detection

Requires only 2 wires for data communication

Disadvantages of UART/ USART Communication Protocol

Doesn't support multiple master slave functionality

Baud rate of communicating UART should be within 10 percent of each other

Intra System Communication Protocols

The Intra system protocol establishes communication between components within the circuit board. In embedded systems, intra system protocol increases the number of components connected to the controller. Increase in components lead to circuit complexity and increase in power consumption. Intra system protocol promises secure access of data from the peripherals.

Types of Intra System Communication Protocols

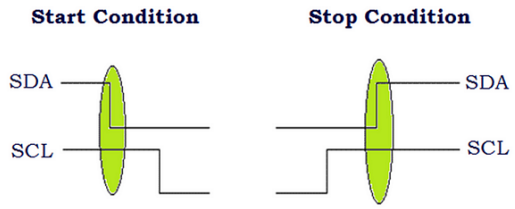
I2C Protocol

SPI Protocol

CAN Protocol

I2C Communication Protocols

Inter Integrated Circuit (I2C) is a serial communication protocol developed by Philips Semiconductors. The main purpose of this protocol is to provide easiness to connect peripheral chips with microcontroller. I2C necessitates two wires SDA (Serial Data Line) and SCL (Serial Clock Line) to carry information between devices. It is a master to slave communication protocol. In order to establish communication, master device initially sends the target slave address along with R/W (Read/Write) flag. The corresponding slave device will move into active mode leaving other devices in off state. Once the slave device is ready, communication starts between master and slave devices. One bit acknowledgment is replied by the receiver if transmitter transmits 1 byte (8 bits) of data. A stop condition is issued at the end of communication between devices.



Advantages of I2C Communication Protocols

Provides good communication between onboard devices which are accessed infrequently

Addressing mechanism eases master slave communication

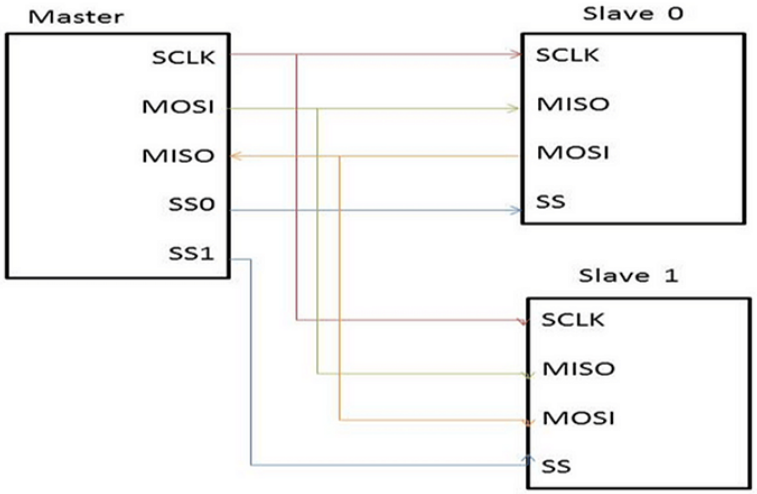
Cost and circuit complexity does not end up on number of devices

Disadvantages of I2C Communication Protocols

The biggest disadvantage of I2C Communication Protocols is its limited speed.

Serial Peripheral Interface (SPI) Communication Protocols

SPI (**S**erial **P**eripheral **I**nterface) is one of the serial communication protocols developed by Motorola. It is a 4-wire protocol namely MOSI (**M**aster **O**ut **S**lave **I**n), MISO (**M**aster **I**n **S**lave **O**ut, SS (**S**lave **S**elect), and SCLK (Serial Clock). As I2C protocol, SPI is also a master to slave communication protocol. In SPI, the master device first configures the clock at a particular frequency. Furthermore, the SS line is used to select the appropriate slave by pulling the SS line low where it is normally held high. The communication is established between the selected slave and the master device as soon as appropriate slave device is selected. SPI is a full duplex communication protocol. SPI doesn't limit data transfer to 8-bit words.



Advantages of SPI Communication Protocols

Faster than asynchronous serial communication protocol.

Support multiple slaves' connectivity.

Universally accepted protocol and low cost.

Disadvantages of SPI Communication Protocol

Requires more wires than other communication protocols.

Master device should control all slave communications (slave-slave communication is impossible).

· Numerous slave devices lead to circuit complexity.

The choice of protocol depends on factors such as data transfer speed, distance between devices, power consumption, and the specific requirements of the application or system in which the firmware operates.

Example problem on protocols selection

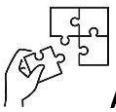
Wearable health monitor that tracks various vital signs and communicates with multiple peripherals may use communication protocols for each component as follow:

Component	**Protocol**

Heart Rate Sensor	**I2C:** IC2 is suitable for short-distance communication and requires only two wires. It's ideal for simple sensors that don't need high-speed data transfer.
Blood Oxygen Sensor	**I2C:** Similar to the heart rate sensor, I2C is sufficient for the data rate required and keeps pin usage minimal.
Motion Sensor (Accelerometer/Gyroscope):	**SPI**: SPI offers higher speed than I2C, which is beneficial for the potentially frequent updates from a motion sensor. It's also full-duplex, allowing simultaneous data transfer in both directions.
Small OLED Display	**SPI**: OLED displays often require faster data transfer rates to update the screen smoothly. SPI provides the necessary speed and is commonly supported by OLED display modules.
Bluetooth Low Energy (BLE) Module	**UART**: UART is a common interface for BLE modules. It's simple to implement and provides sufficient speed for BLE communication.
External Flash Memory	**SPI**: SPI is ideal for external flash memory due to its high speed and support for burst read/write operations, which are common in flash memory access.
USB Port	**USB**: The USB protocol is necessary for USB functionality. It will be used for both charging and data transfer when connected to a computer.

Points to Remember

- **Communication protocols in firmware** refer to a set of rules and conventions that define the format and sequence of messages exchanged between devices or systems.
- **Types of Communication Protocols in Embedded Systems are** Inter System Protocol and Intra System Protocol.



Application of learning 3.3

You are a firmware developer at a medical technology startup. Your team is developing a new wearable health monitor that tracks various vital signs and communicates with multiple peripherals. Your task is to select appropriate communication protocols for different components of the device.

Indicative content 3.4: Identification of default segments of data memory

Duration: 5hrs

Theoretical Activity 3.4.1: Description of default segments of data memory

Tasks:

1: You are requested to answer the following questions:

i. What do you mean by default segment of data memory?
ii. What are the types of default segments?
iii. Explain the use of each type of default data segment.

2: Provide the answers for the asked questions.

3: Present your findings to the trainer or your colleagues.

4: Ask for clarification if any.

5: Read the key readings 3.4.1 in trainee's manual.

6: Ask to trainer for clarification if any.

Key readings 3.4.1: Description of default segments of data memory

a. Definition of default segmant

Default segments in firmware typically refer to predefined sections or areas within the firmware code where specific types of data or instructions are stored.

b. Types of default segments

- ✓ ***Data segment***: This segment typically stores initialized global and static variables. When your program starts, the data in this segment is set to predefined values.
- ✓ ***bss*** (Block Started by Symbol) **segment**: This segment holds uninitialized or zero-initialized global and static variables. It doesn't store the actual data but allocates memory for variables before runtime.
- ✓ ***Heap*** **segment:** The heap is dynamically allocated memory during runtime. It's commonly used for dynamic memory allocation, such as when using functions like **malloc()** or **new** in languages like C and C++.
- ✓ ***Stack*** **segment**: The stack stores local variables, function parameters, return addresses, and other function call-related information. It operates in a Last-In-First-Out (LIFO) manner and manages function calls and returns.

These segments help organize and manage different types of data within a program's memory space, each with its own purpose and characteristics.

c. Detailed description of segments

i. Data segment

- Purpose: Stores initialized global and static variables
- Content: Variables with predefined values at compile time
- Access: Read-write
- Lifetime: Entire program execution

ii. bss segment

- Name stands for "Block Started by Symbol"
- Purpose: Stores uninitialized global and static variables
- Content: Variables initialized to zero or null pointers
- Access: Read-write
- Lifetime: Entire program execution
- Takes up less space in the executable file than .data

iii. Heap segment

- Purpose: Dynamic memory allocation
- Content: Objects and data structures allocated at runtime
- Access: Read-write
- Lifetime: Managed by the programmer (allocation/deallocation)
- Grows upward in memory (usually)
- Prone to fragmentation over time

iv. Stack segment

- Purpose: Manages function calls and local variables
- Content: Function parameters, return addresses, local variables
- Access: Read-write
- Lifetime: Automatic (variables are created/destroyed as functions are called/returned)
- Grows downward in memory (usually)
- Uses Last-In-First-Out (LIFO) structure

Points to Remember

- **Data segment**: This segment typically stores initialized global and static variables. When your program starts, the data in this segment is set to predefined values.
- **bss** (Block Started by Symbol) **segment**: This segment holds uninitialized or zero-initialized global and static variables. It doesn't store the actual data but allocates memory for variables before runtime.
- **Heap segment**: The heap is dynamically allocated memory during runtime. It's commonly used for dynamic memory allocation, such as when using functions like **malloc()** or **new** in languages like C and C++.
- **Stack segment**: The stack stores local variables, function parameters, return addresses, and other function call-related information.

Application of learning 3.4.

You are developing a simple temperature monitoring system for a greenhouse using an embedded system. The system needs to store the following data:

1. The current temperature (which changes frequently)
2. The maximum allowed temperature (a constant value)
3. The minimum allowed temperature (a constant value)
4. An array to store the last 24 hourly temperature readings.

Your task is to allocate memory effectively using the appropriate segments in firmware.

Indicative content 3.5: Performance of memory allocation in embedded

Duration: 5hrs

Theoretical Activity 3.5.1: Description of memory allocation in embedded C

Tasks:

1: You are requested to answer the following questions.

i. Describe memory allocation.
ii. Explain the two types of memory allocation
iii. List the characteristics of static memory allocation
iv. List the characteristics of dynamic memory allocation
v. Enumerate types of statically allocated variables
vi. What are advantages of static memory allocation?
vii. What are disadvantages of static memory allocation?
viii. What are best practices for static memory allocation?
ix. Give the characteristics of dynamic memory allocation
x. What are the advantages of dynamic memory allocation?
xi. What are the disadvantages of dynamic memory allocation?
xii. Explain best practice for dynamic memory allocation
xiii. Compare static and dynamic memory allocation
xiv. Give the key functions used in dynamic memory allocation.

2: Provide the answers for the asked questions.

3: Present your findings to the trainer or your colleagues.

4: Ask for clarification if any.

5: Read the key readings 3.5.1 in trainee's manual.

Key readings 3.5.1.: Description of memory allocation

1. Definition

Memory allocation refers to the process of reserving a portion of a computer's available memory for use by a program or process. It involves assigning specific memory addresses to store data, instructions, or other information needed for the program to run efficiently.

2. Types of memory allocation

Static memory allocation

Static memory allocation refers to the allocation of memory at compile time.

The size and lifetime of statically allocated variables are determined before the program runs.

Characteristics of static memory allocation

- Allocated on the stack or in the global memory area
- Size is fixed and cannot be changed during runtime
- Allocation and deallocation are automatically handled by the compiler

Types of Statically Allocated Variables

1. **Global Variables**: Declared outside of any function, accessible throughout the program.
2. **Static Variables**: Retain their value between function calls, can be local or global.
3. **Local Variables**: Declared inside functions, automatically allocated and deallocated.

Example of static memory allocation

```
int globalVar = 10;  // Global variable

static int staticVar = 5;  // Static global variable

void function() {

   static int staticLocalVar = 0;  // Static local variable

   int localVar = 20;  // Local variable

   // ...

}
```

Advantages of static memory allocation

1. **Simplicity**: Easy to implement and understand.
2. **Speed**: No runtime overhead for allocation/deallocation.
3. **Predictability**: Memory usage is known at compile time.
4. **No Fragmentation**: Memory layout is contiguous and fixed.

Disadvantages of memory allocation

1. **Inflexibility**: Cannot adjust memory usage based on runtime needs.
2. **Potential Waste**: Must allocate for worst-case scenarios, which may waste memory.
3. **Limited Scope**: Not suitable for data structures that need to grow or shrink.

Best Practices for Static Allocation

1. Use for fixed-size data structures and variables.
2. Prefer local variables over global when possible to minimize scope.
3. Use const qualifier for read-only data to allow placement in ROM.

4. Be mindful of stack usage, especially with large local arrays.

2. Dynamic Memory Allocation

Definition

Dynamic memory allocation refers to the allocation of memory at runtime. It allows for flexible memory usage based on program needs during execution.

Characteristics of dynamic memory allocation

- Allocated on the heap
- Size can be determined at runtime
- Requires explicit allocation and deallocation by the programmer

Key Functions used in dynamic memory allocation

1. malloc(): Allocates a block of uninitialized memory
2. calloc(): Allocates a block of zero-initialized memory
3. realloc(): Resizes a previously allocated memory block
4. free(): Deallocates a previously allocated memory block

Example

```
#include <stdlib.h>

void function() {
   int *dynamicArray;
   int size = 10;
 dynamicArray = (int *)malloc(size * sizeof(int));
   if (dynamicArray == NULL) {
      // Handle allocation failure
      return;
   }
   // Use the dynamically allocated memory
   for (int i = 0; i < size; i++) {
      dynamicArray[i] = i;
   }
// Don't forget to free the allocated memory
   free(dynamicArray);
}
```

Advantages of dynamic memory allocation

1. **Flexibility**: Can allocate memory as needed during runtime.
2. **Efficiency**: Can potentially use memory more efficiently than static allocation.
3. **Scalability**: Suitable for data structures that need to grow or shrink.

Disadvantages of dynamic memory allocation

1. **Complexity**: Requires careful management to avoid leaks and errors.
2. **Overhead**: Allocation and deallocation have runtime costs.
3. **Fragmentation**: Can lead to memory fragmentation over time.
4. **Unpredictability**: Allocation can fail if memory is exhausted.
1. **Limited Resources**: Many embedded systems have very limited heap space.
2. **Real-time Constraints**: Dynamic allocation can introduce unpredictable delays.
3. **Long-running Systems**: Memory leaks are particularly problematic.
4. **Fragmentation**: Can be a significant issue in long-running systems.

Best Practices for Dynamic Allocation in Embedded C

1. **Avoid if Possible**: Use static allocation when the memory needs are known and fixed.
2. **Allocate Early**: If dynamic allocation is necessary, try to do it during initialization.
3. **Fixed-size Allocations**: Use fixed-size allocations to reduce fragmentation.
4. **Memory Pools**: Implement custom memory pools for frequently allocated objects.
5. **Error Checking**: Always check for allocation failures and handle them gracefully.
6. **Consistent Pairing**: Ensure every malloc () has a corresponding free().
7. **Avoid Frequent Allocation/Deallocation**: Reuse allocated memory when possible.
8. **Use Static Analysis Tools**: To detect potential memory leaks and other issues.

Comparison of Static and Dynamic Allocation

Aspect	**Static Allocation**	**Dynamic Allocation**
When is memory allocated?	Compile time	Runtime
Flexibility	Fixed size	Flexible size
Speed of allocation	Fast (done at compile time)	Slower (runtime overhead)
Memory efficiency	Potentially wasteful	Can be more efficient
Fragmentation	No fragmentation	Can lead to fragmentation
Complexity	Simple to use	More complex, requires careful management

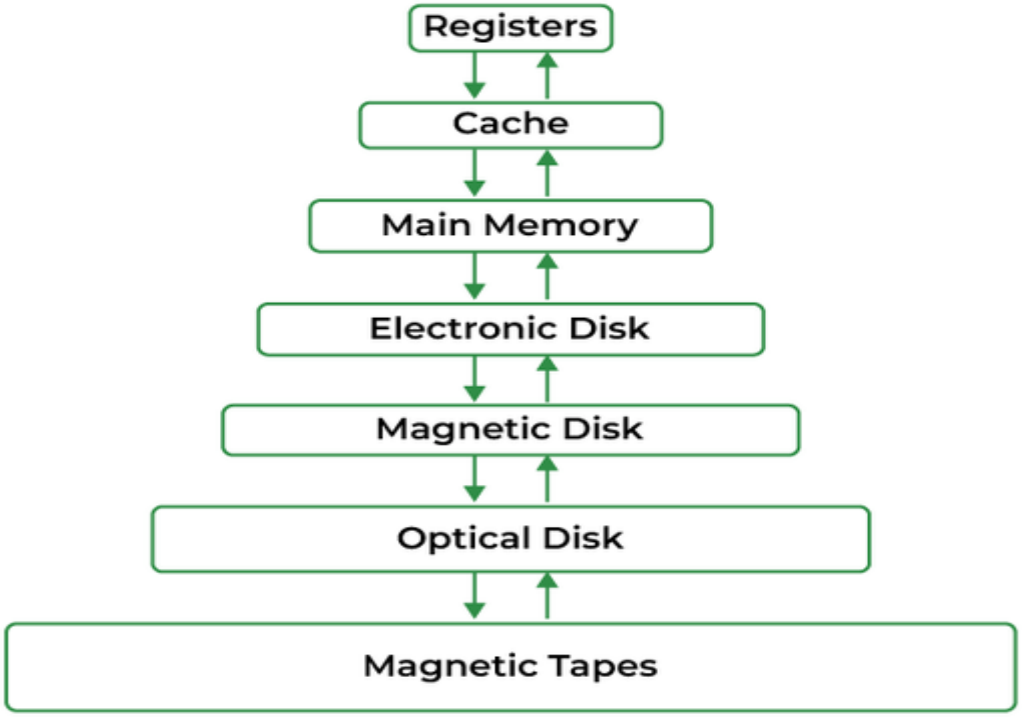
Suitable for	Fixed-size data, predictable memory needs	Variable-size data, unpredictable memory needs
Stack vs Heap	Usually stack (except globals)	Always heap
Lifetime management	Automatic	Manual (requires explicit freeing)

Conclusion

In embedded C programming, the choice between static and dynamic memory allocation is crucial and depends on various factors including system constraints, performance requirements, and the nature of the data being managed. Static allocation offers simplicity and predictability, making it preferable in many embedded scenarios, especially those with hard real-time constraints. Dynamic allocation provides flexibility but requires careful management and consideration of its implications on system behavior.

In practice, many embedded systems use a combination of both methods, leveraging the strengths of each where appropriate. Regardless of the chosen method, thorough understanding, careful implementation, and rigorous testing are essential for creating robust and efficient embedded systems.

Performing static and dynamic allocation management system



To effectively manage memory allocation in programming, it is essential to understand the two primary types of memory allocation: static and dynamic. Each type has its own characteristics, advantages, and disadvantages. Below is a detailed explanation of how to perform static and dynamic allocation management.

1. Understanding Static Memory Allocation

Static memory allocation occurs at compile time. The size of the memory required for variables or data structures is determined before the program runs. This means that once allocated, the size cannot change during runtime. Here are some key points regarding static memory allocation:

- **Fixed Size:** The amount of memory allocated is fixed and known at compile time.
- **Lifetime:** The lifetime of statically allocated variables lasts for the duration of the program.
- **Memory Management:** The compiler manages this memory, typically storing it in a program's data segment.
- **Speed:** Accessing statically allocated memory is generally faster since it does not involve any overhead associated with dynamic allocation.

Example:

```
int array[100]; // Static allocation of an array with 100 integers
```

2. Understanding Dynamic Memory Allocation

Dynamic memory allocation occurs at runtime, allowing programs to request or release memory as needed. This flexibility is crucial for applications where the amount of required memory cannot be predetermined. Key aspects include:

- **Variable Size:** Memory can be allocated in varying sizes based on runtime requirements.
- **Lifetime:** Dynamically allocated memory remains available until explicitly freed by the programmer.
- **Functions Used:** Common functions for dynamic allocation in C include **malloc()**, **calloc()**, **realloc()**, and **free()**.
- **Overhead:** Dynamic allocation involves some overhead due to bookkeeping by the allocator.

Example:

```
int *array = (int *)malloc(100 * sizeof(int)); // Dynamic allocation for an array of 100 integers

if (array == NULL) {

   // Handle malloc failure

}
```

3. Managing Static Allocation

When managing static allocations, consider the following strategies:

- **Predefine Sizes:** Always define sizes based on expected maximum usage to avoid overflow.
- **Use Constants:** Utilize constants or macros for defining sizes to improve code readability and maintainability.

4. Managing Dynamic Allocation

For effective management of dynamic allocations, follow these steps:

- **Allocate Memory Wisely:** Use appropriate functions (**malloc**, **calloc**) based on whether you need uninitialized or zero-initialized memory.
- **Check for NULL:** Always check if your pointer returned from a dynamic allocation function is NULL to handle potential out-of-memory situations gracefully.
- **Free Allocated Memory:** Ensure that every dynamically allocated block of memory is freed using **free()** when it is no longer needed to prevent memory leaks.

Example:

```
free(array); // Freeing dynamically allocated array
```

5. Combining Both Approaches

In many applications, a combination of both static and dynamic allocations may be necessary:

- Use static arrays for fixed-size data that does not change during execution.
- Use dynamic allocations when dealing with variable-sized data structures such as linked lists or trees where size cannot be predetermined.

By understanding both static and dynamic allocations and their respective management techniques, developers can optimize resource usage while preventing common issues such as memory leaks or buffer overflows.

Practical Activity 3.5.2: Performing static and dynamic allocation management system.

Task:

1: You are requested to perform the following activity:

You are requested to design a memory management system.

2: Read key reading 3.5.2

3: Present your work to the trainer and the whole class

4: Ask question if any.

Key reading: 3.5.2: Developing a memory management allocation system in C

To develop a memory management allocation system in C, we will create a simple program that demonstrates both static and dynamic memory allocation. The program will include functions to allocate, deallocate, and manage memory effectively. Below are the steps along with the corresponding C code.

Step 1: Include Necessary Headers

We need to include standard libraries for input/output operations and memory management.

```
#include <stdlib.h> // For malloc(), free()
#include <string.h> // For memset(), memcpy()
#include <stdio.h> // For debugging (printf())
#include <stdint.h> // For fixed-width types (uint8_t, size_t)
```

Step 2: Define Constants

Define constants for the maximum size of static arrays.

```
#define MAX_SIZE 100
```

Step 3: Function Prototypes

Declare function prototypes for allocating and freeing memory.

```
void allocateStaticArray();
void allocateDynamicArray(int size);
```

Step 4: Implement Static Memory Allocation

Create a function that allocates a static array and initializes it.

```
void allocateStaticArray() {
   int staticArray[MAX_SIZE];

   // Initialize the static array
   for (int i = 0; i < MAX_SIZE; i++) {
      staticArray[i] = i + 1; // Fill with values 1 to MAX_SIZE
   }

   // Print the static array values
   printf("Static Array Values:\n");
   for (int i = 0; i < MAX_SIZE; i++) {
      printf("%d ", staticArray[i]);
   }
   printf("\n");
}
```

Step 5: Implement Dynamic Memory Allocation

Create a function that allocates a dynamic array based on user input.

```
void allocateDynamicArray(int size) {
   // Allocate memory dynamically using malloc
   int *dynamicArray = (int *)malloc(size * sizeof(int));

   // Check if malloc succeeded
   if (dynamicArray == NULL) {
      printf("Memory allocation failed!\n");
      return;
   }

   // Initialize the dynamic array
   for (int i = 0; i < size; i++) {
      dynamicArray[i] = (i + 1) * 2; // Fill with values (2, 4, ..., size*2)
   }

   // Print the dynamic array values
   printf("Dynamic Array Values:\n");
```

```
    for (int i = 0; i < size; i++) {
        printf("%d ", dynamicArray[i]);
    }
    printf("\n");

    // Free allocated memory to prevent memory leaks
    free(dynamicArray);
}
```

Step 6: Main Function Implementation

In the **main** function, call both allocation functions and handle user input for dynamic allocation.

```
int main() {
    int dynamicSize;

    // Perform static allocation demonstration
    allocateStaticArray();

    // Ask user for size of dynamic array to allocate
    printf("Enter the size of the dynamic array: ");
    scanf("%d", &dynamicSize);

    if (dynamicSize > 0 && dynamicSize <= MAX_SIZE) {
        allocateDynamicArray(dynamicSize);
    } else {
        printf("Invalid size! Please enter a value between 1 and %d.\n", MAX_SIZE);
    }

    return 0;
}
```

Complete Code

Here is the complete code combining all steps:

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

#define MAX_SIZE 100

void allocateStaticArray();
void allocateDynamicArray(int size);
```

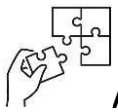
```
void allocateStaticArray() {
  int staticArray[MAX_SIZE];

  for (int i = 0; i < MAX_SIZE; i++) {
    staticArray[i] = i + 1;
  }

  printf("Static Array Values:\n");
  for (int i = 0; i < MAX_SIZE; i++) {
    printf("%d ", staticArray[i]);
  }
  printf("\n");
}

void allocateDynamicArray(int size) {
  int *dynamicArray = (int *)malloc(size * sizeof(int));

  if (dynamicArray == NULL) {
    printf("Memory allocation failed!\n");
    return;
  }

  for (int i = 0; i < size; i++) {
    dynamicArray[i] = (i + 1) * 2;
  }

  printf("Dynamic Array Values:\n");
  for (int i = 0; i < size; i++) {
    printf("%d ", dynamicArray[i]);
  }
  printf("\n");

  free(dynamicArray);
}

int main() {
  int dynamicSize;

  allocateStaticArray();
```

```
    printf("Enter the size of the dynamic array: ");
    scanf("%d", &dynamicSize);

    if (dynamicSize > 0 && dynamicSize <= MAX_SIZE) {
        allocateDynamicArray(dynamicSize);
    } else {
        printf("Invalid size! Please enter a value between 1 and %d.\n", MAX_SIZE);
    }

    return 0;
}
```

This program demonstrates both static and dynamic memory allocations in C. It initializes an array statically with fixed values and allows users to specify the size of a dynamically allocated array at runtime. It also includes error handling to ensure proper memory management practices are followed.

Points to Remember

- Memory allocation means reserving computer memory for programs
- Types of **memory allocation** are Static memory allocation and Dynamic memory allocation.
- **Static memory allocation** in embedded system involves reserving memory for variables during compile-time.
- **Dynamic memory allocation** in embedded system refers to the process of allocating memory for variables or data structures during runtime, as opposed to static allocation done at compile time
- In dynamic memory allocation, memory is allocated as needed, reducing wastage and potentially optimizing memory usage.
- In static memory a allocation, memory is allocated at compile time, making it predictable and deterministic.
- **Steps for performing memory management**
    1. Include Necessary Headers
    2. Define Constants
    3. Function Prototypes
    4. Implement Static Memory Allocation
    5. Implement Dynamic Memory Allocation
    6. Main Function Implementation



Application of learning 3.5

You are developing firmware for an advanced environmental monitoring system. The system collects data from various sensors (temperature, humidity, air quality, etc.) and stores it for later transmission. The number of active sensors can vary based on the deployment configuration, and the data collection frequency can be adjusted remotely.

Requirements:

1. Support up to 10 different types of sensors
2. Each sensor can produce variable-length data (2 to 20 bytes)
3. The system should store up to 24 hours of data before transmission
4. The data collection frequency can range from once per minute to once per hour
5. The system runs on a microcontroller with 64KB of RAM and 256KB of flash memory
6. The system should be able to handle sensor addition/removal without recompilation

Task: Design a memory management system that efficiently handles the storage of sensor data, balancing between static and dynamic allocation to optimize performance and flexibility.

Indicative content 3.6: Description of Concepts for Developing Firmware

Duration: 5 hrs

Theoretical Activity 3.6.1: Description of Embedded C programming concept

Tasks:

1: Answer the following questions:

i. What are data types?
ii. What is the difference between structures and Unions?
iii. What is Bit fields?
iv. Describe Pre-processor directives.
v. Differentiate application programming interface and hardware abstraction layers.
vi. Identify Hardware design interface.

2: Provide the answers for the asked questions and write them on flipchart/papers.

3: Present your findings to the trainer or your colleagues.

4: ask for clarification if necessary.

5: read the Key readings 3.6.1 in their manuals.

Key readings 3.6.1.: Description of Embedded C programming concept

✓ ***Embedded C Programming concepts***

a. Definition

Embedded C programming is a specialized subset of the C programming language designed for developing software for embedded systems. Embedded systems are computer systems that are embedded in other devices, such as microcontrollers, smart appliances, and medical equipment.

b. **Key Characteristics of Embedded C**

- ✓ **Efficiency:** Embedded C is designed to be efficient in terms of memory usage and execution speed, making it suitable for resource-constrained devices.
- ✓ **Real-Time Capabilities:** It is often used in applications that require real-time responses, such as control systems and robotics.
- ✓ **Hardware Interaction:** Embedded C provides direct access to hardware peripherals, allowing programmers to interact with sensors, actuators, and other devices.
- ✓ **Portability:** Embedded C code can be easily ported to different hardware platforms, making it versatile for various embedded systems.

c. **Embedded C Applications**

- ✓ **Microcontroller Programming:** Controlling microcontrollers for tasks like motor control, data acquisition, and communication.
- ✓ **Real-Time Systems:** Developing control systems for applications such as robotics, industrial automation, and medical devices.
- ✓ **Consumer Electronics:** Programming embedded systems in devices like smartphones, smart appliances, and gaming consoles.
- ✓ **Automotive Electronics:** Developing firmware for car systems, including engine control units, infotainment systems, and safety features.

d. **Key Concepts in Embedded C**

- ✓ **Pointers:** Understanding pointers is essential for working with memory addresses and accessing hardware registers.
- ✓ **Memory Management:** Efficiently managing memory usage is crucial due to the limited resources available in embedded systems.
- ✓ **Interrupts:** Handling interrupts from hardware devices for timely responses and event-driven programming.
- ✓ **Peripheral Control:** Interacting with hardware peripherals (e.g., timers, ADC, UART) using specific registers and control mechanisms.
- ✓ **Real-Time Operating Systems (RTOS):** Understanding how to use RTOSes to manage tasks, resources, and scheduling in real-time applications.

e. Data types

Data type in C define the kind of values a variable can hold and the operations that can be performed on them.

- ✓ **Fundamental Data Types or primitive data types in C**
- **int:** Integer values (e.g., -10, 0, 42)
- **float:** Single-precision floating-point numbers (e.g., 3.14, -2.5)
- **double:** Double-precision floating-point numbers (e.g., 123.456, -0.001)
- **char:** Single characters (e.g., 'a', 'b', '1', '$')
- **void:** Used to represent the absence of a value, often used for function return types or pointers.
- ✓ **Derived Data Types or Non-Primitive Data Types**
- **Arrays:** Ordered collections of elements of the same data type (e.g., int numbers[5]).
- **Pointers:** Variables that store memory addresses of other variables (e.g., int *ptr).
- **Structures:** User-defined data types that group related variables of different data types (e.g., struct student { int id; char name[20]; float gpa; }).
- **Unions:** Similar to structures, but members share the same memory location, allowing for different interpretations of the same data.

Example:

```
int age = 25; // Integer variable
float pi = 3.14159; // Floating-point variable
char initial = 'A'; // Character variable

int numbers[5] = {1, 2, 3, 4, 5}; // Array of integers
int *ptr = &age; // Pointer to the age variable
```

✓ **Criteria of choosing the right Data Type:**

- **Memory Usage:** Consider the memory requirements of different data types. For example, int uses less memory than double.
- **Range of Values:** Ensure the data type can accommodate the expected range of values.
- **Precision:** Choose appropriate data types for numerical calculations based on the required precision.
- **Readability:** Use meaningful variable names and comments to improve code readability.

✓ **Structures and Unions**

Structures and **unions** are two user-defined data types in C that allow you to group related variables of different data types into a single entity.

Structures

- **Purpose:** Structures are used to define a collection of variables, each with a specific name and data type. This helps organize and manage related data.
- **Syntax:**

```
struct struct_name {
  data_type member1;
  data_type member2;
  // ...
};
```

Description of code keywords

- struct_name: The name of the structure.
- data_type: The data type of each member.
- member: The name of each member.

- **Example:**

```
struct student {
  int id;
  char name[20];
  float gpa;
};
```

➢ **Unions**

- **Purpose:** Unions are similar to structures, but all members share the same memory location. This allows you to interpret the same data in different ways.
- **Syntax:**

```
union union_name {
  data_type member1;
  data_type member2;
  // ...
};
```

Desription of code keywords

- union_name: The name of the union.
- data_type: The data type of each member.
- member: The name of each member.

- **Example:**

```
union data {
  int integer;
  float floating;
  char character;
   };
```

Key Differences

Feature	**Structures**	**Unions**
Memory Usage	Each member has its own memory location.	All members share the same memory location.
Access	Members can be accessed independently.	Only one member can be accessed at a time.
Data Interpretation	Members can be interpreted as different data types.	The same memory location can be interpreted as different data types.

When to Use Structures and Unions:

- **Structures:** Use structures to group related data that needs to be treated as a single entity, such as a student record or a product information.
- **Unions:** Use unions when you need to interpret the same data in different ways, such as representing a value as either an integer or a floating-point number.

Example Usage:

```
struct student student1;
student1.id = 101;
strcpy(student1.name, "Alice");
student1.gpa = 3.5;
```

```
union data value;
value.integer = 42;
printf("Integer value: %d\n", value.integer);
value.floating = 3.14;
printf("Floating-point value: %f\n", value.floating);
```

Bit fields

Bit fields are a C language feature that allows you to pack multiple variables of different data types into a single memory location.

This is useful for saving memory space in embedded systems or when you need to manipulate individual bits of a data structure.

Syntax:

```
struct struct_name {
   data_type member_name : width;
};
```

Desription of code keywords

- struct_name: The name of the structure.
- data_type: The base data type of the member.
- member_name: The name of the member.
- width: The number of bits allocated for the member.

Example:

```
struct flags {
   unsigned int bit1 : 1;
   unsigned int bit2 : 1;
   unsigned int bit3 : 1;
};
```

In this example, the flags structure defines three bits. Each bit can be either 0 or 1.

Key Points:

- **Memory Efficiency:** Bit fields can save memory by packing multiple variables into a single memory location.
- **Portability:** The exact size of bit fields may vary depending on the compiler and target architecture, so use them with caution in portable code.
- **Alignment:** Bit fields may not be aligned on byte boundaries, which can affect performance in some cases.
- **Readability:** While bit fields can be efficient, they can also make code less readable if used excessively.

When to Use Bit Fields:

- **Memory-constrained systems:** When memory is limited, bit fields can help conserve space.
- **Hardware-specific data structures:** If you need to interact with hardware that uses specific bit patterns.
- **Flags or status indicators:** Bit fields can be used to represent a set of flags or status indicators.

Example Usage:

```
struct flags flags;
flags.bit1 = 1;
flags.bit2 = 0;
flags.bit3 = 1;

// Accessing individual bits
if (flags.bit1 == 1) {
   // Bit 1 is set
}
```

Caution:

- **Compiler-dependent behavior:** The exact behavior of bit fields may vary between compilers.
- **Portability:** Avoid using bit fields for critical or platform-dependent code.

Preprocessor directives

Preprocessor directives are instructions to the C preprocessor, which is a program that processes your source code before its compiled.

They are used to perform tasks like including header files, defining macros, and conditionally compiling code.

Common Preprocessor Directives:

- **#include:** Includes the contents of another file.

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

#define: Defines a macro, which is a textual substitution.

```
#define PI 3.14159
```

#ifdef, #ifndef, #else, #endif: Conditional compilation directives.

```
#ifdef DEBUG
   printf("Debug mode is enabled\n");
#else
   printf("Debug mode is disabled\n");
#endif
```

#if, #elif, #else, #endif: Conditional compilation based on expressions.

```
#if VERSION == 1
   // Code for version 1
```

```
#elif VERSION == 2
  // Code for version 2
#else
  // Default code
#endif
```

- **#pragma:** Compiler-specific directives for various purposes.

Example:

```
#include <stdio.h>

#define MAX_LENGTH 100

int main() {
  char name[MAX_LENGTH];

  printf("Enter your name: ");
  scanf("%s", name);

  printf("Hello, %s!\n", name);

  return 0;
}
```

In this example:

- #include <stdio.h> includes the standard input/output header file.
- #define MAX_LENGTH 100 defines a constant for the maximum length of the name.

Preprocessor directives are a powerful tool for organizing and managing C code. They can be used to make code more modular, readable, and maintainable.

✓ ***Application of Application Programming Interfaces (APIs) and Hardware Abstraction Layers (HALs) fundamentals***

APIs (Application Programming Interfaces) and **HALs (Hardware Abstraction Layers)** are both essential components of software development, but they serve different purposes and have distinct characteristics.

- APIs versus HALs

Feature	API	HAL
Purpose	Provides a standardized interface for software components	Hides hardware-specific details from the application

Scope	Can be used for various types of interactions (e.g., network, file system, graphics)	Typically focused on hardware-related interactions
Implementation	Can be implemented in software or hardware	Often implemented in software
Abstraction Level	Can provide varying levels of abstraction	Typically provides a low-level abstraction of hardware
Usage	Used by application developers to interact with the system	Used by device drivers and other system components

- **Operating System APIs:** Windows API, macOS API, Linux API provide a standardized interface for applications to interact with the operating system.
- **Hardware-Specific APIs:** CUDA (NVIDIA GPUs), OpenCL (AMD GPUs, CPUs), and Vulkan (cross-platform graphics API) are examples of hardware-specific APIs.
- **Custom-Built APIs:** Many applications and devices have their own custom-built APIs to provide specific functionalities.

The API and HAL Landscape

The API and HAL landscape is essential for software developers who want to build applications that interact with other systems or devices. By staying informed about the latest trends and technologies, developers can leverage APIs and HALs to create innovative and powerful software. The API and HAL landscape is vast and constantly evolving.

Operating System APIs

- **Windows API (Win32 API):** The primary API for Windows applications.
- **macOS API (Cocoa API):** The API for macOS applications.
- **Linux API:** A collection of APIs for Linux systems, including POSIX, GTK+, and Qt.

Hardware-Specific APIs

- **CUDA (Compute Unified Device Architecture):** NVIDIA's API for GPU computing.
- **OpenCL (Open Computing Language):** A cross-platform API for general-purpose computing on GPUs and CPUs.
- **Vulkan:** A new, low-level graphics API designed for high-performance applications.

Third-Party APIs

- **Google Maps API:** Provides access to Google Maps data and services.
- **Twitter API:** Allows developers to interact with the Twitter platform.

- **Facebook API:** Provides access to Facebook's social graph and data.

➢ **Custom-Built APIs**

- Many companies and organizations develop their own custom APIs to expose specific functionalities or services.

➢ **Trends and Future Directions**

- **Cloud-based APIs:** The increasing popularity of cloud computing has led to the development of many cloud-based APIs, such as Amazon Web Services (AWS) and Microsoft Azure.
- **Microservices Architecture:** The adoption of microservices architecture has led to the creation of smaller, more focused APIs.
- **API Economy:** The growing importance of APIs has created a new economic model, where companies can monetize their data and services through APIs.

The API Scope

APIs, or Application Programming Interfaces, are like gateways that allow different software applications to communicate and interact with each other. They define the methods and data formats that applications can use to request and exchange information.

The scope of an API refers to what functionalities, data, or operations it provides access to. API scopes are essentially permissions or levels of access granted to developers or applications when they use an API.

For instance, an API for a social media platform might have different scopes such as read-only access to a user's profile, permission to post on behalf of a user, or access to a user's friend list. These scopes determine what actions or data the developer can utilize within their application.

Controlling API scopes is crucial for security and privacy reasons. By defining specific scopes, API providers can manage and limit access to sensitive data, ensuring that only authorized actions are permitted.

API Characteristics

APIs, or Application Programming Interfaces, serve as intermediaries that enable different software systems to communicate and interact with each other.

1. **Interoperability:** APIs facilitate the interaction between different software systems, allowing them to work together regardless of the technology stack or platform they're built upon.

2. **Abstraction:** APIs abstract underlying complexities, presenting a simplified interface that developers can use without needing to understand the internal workings of the system.

3. **Reusability:** APIs promote code reuse by providing a set of functions or procedures that can be utilized across various applications or services.

4. **Modularity:** APIs are often designed in a modular fashion, allowing developers to access specific functionalities or services without impacting the entire system.

5. **Consistency:** APIs maintain consistency in their interface and functionality to ensure predictability and ease of use for developers integrating them into their applications.

6. **Security:** APIs need robust security measures to protect sensitive data and prevent unauthorized access or misuse.

7. **Documentation**: Clear and comprehensive documentation is crucial for APIs, providing developers with information on how to use the API, its endpoints, request/response formats, and any limitations or best practices.

8. **Versioning and backward compatibility: APIs** often evolve, and maintaining backward compatibility or providing clear versioning strategies is vital to ensure existing applications continue to function while accommodating updates.

9. **Performance:** Efficient APIs are essential for optimal performance. They should handle requests swiftly, with minimal latency and downtime.

10. **Scalability:** APIs should be designed to scale with increasing demand, ensuring they can handle a growing number of users and transactions without compromising performance.

- **Designing your own API**

1. **Identify Needs:** Understand the purpose, audience, and requirements of the API.

2. **Define Endpoints**: Determine what functionalities the API will provide and create clear endpoints.

3. **Design Protocols:** Choose appropriate communication protocols (REST, GraphQL, etc.) and data formats (JSON, XML, etc.).

4. **Prioritize Usability**: Ensure simplicity, consistency, and intuitive design for ease of use.

5. **Document Thoroughly**: Create comprehensive documentation with examples and explanations to guide users.

6. **Ensure Security:** Implement authentication, authorization, and data encryption to protect the API.

7. **Plan for Future Expansion:** Allow for versioning and scalability as needs evolve. Creating APIs and HALs involves understanding the specific requirements of the software or hardware system, ensuring interoperability, abstraction, and ease of use for developers interacting with these interfaces

- **API design in C**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curl/curl.h>
#include <json-c/json.h>
```

```
// Structure to represent weather data
typedef struct {
   char *city;
   char *state;
   double temperature;
   char *description;
} WeatherData;

// Function to fetch weather data from a REST API
WeatherData* getWeatherData(const char *city, const char *state) {
   CURL *curl;
   CURLcode res;
   char *url = NULL;
   char *response = NULL;
   long response_code = 0;

    // Construct the API URL
   url = malloc(strlen(city) + strlen(state) + 32);
   sprintf(url, "https://api.example.com/weather?city=%s&state=%s", city, state);

   // Initialize curl
   curl = curl_easy_init();
   if (curl) {
     curl_easy_setopt(curl, CURLOPT_URL, url);
     curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
      curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_callback);
     curl_easy_setopt(curl, CURLOPT_WRITEDATA, &response);

      // Perform the request
      res = curl_easy_perform(curl);
      if (res != CURLE_OK) {
      fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
        return NULL;
      }

      // Get the response code
      curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response_code);

      // Parse the JSON response
```

```
    json_object *json = json_tokener_parse(response);
    if (!json) {
    fprintf(stderr, "Error parsing JSON response\n");
    return NULL;
    }

    // Extract weather data from the JSON response
    WeatherData *weather = malloc(sizeof(WeatherData));
    weather->city = strdup(json_object_get_string(json_object_get(json, "city")));
    weather->state = strdup(json_object_get_string(json_object_get(json, "state")));
    weather->temperature = json_object_get_double(json_object_get(json,
    "temperature"));
     weather->description = strdup(json_object_get_string(json_object_get(json,
    "description")));

    // Free memory
    json_object_put(json);
    free(response);
    return weather;
  }

  return NULL;
}

// Callback function to write the API response to a string
static size_t write_callback(char *ptr, size_t size, size_t nmemb, void *userdata) {
  char **response = (char **)userdata;
  size_t realsize = size * nmemb;
  *response = realloc(*response, strlen(*response) + realsize + 1);
  if (*response == NULL) {
  return 0;
  }
  strcpy(*response + strlen(*response), ptr);
  *response[strlen(*response)] = 0;
  return realsize;
}
int main() {
  WeatherData *weather = getWeatherData("San Francisco", "CA");
  if (weather) {
    printf("City: %s, State: %s\n", weather->city, weather->state);
```

```
    printf("Temperature: %.2f degrees Celsius\n", weather->temperature);
    printf("Description: %s\n", weather->description);
    free(weather->city);
    free(weather->state);
    free(weather->description);
    free(weather);
  } else {
  fprintf(stderr, "Failed to fetch weather data\n");
    }

  return 0;
}
```

✓ ***Identification of HAL design process***

The Hardware Abstraction Layer (HAL) is a low-level interface provided by many microcontroller manufacturers to abstract the hardware details and simplify the process of writing code that interfaces with hardware peripherals.

Here's a general overview of the HAL design process for GPIO, SPI, EEPROM, and Memory devices:

HAL Design for GPIO (General-Purpose Input/Output)

2. **Initialization**
   • The HAL typically includes functions to initialize GPIO pins, specifying their mode (input/output), speed, pull-up/down resistors, and other parameters.
3. **Configuration and Control**
   • Functions are provided to configure GPIO pins individually or in groups, allowing for setting or clearing specific pins, reading pin values, or altering pin configurations (e.g., switching between input/output modes).
4. **Interrupt Handling**
   HAL might support interrupt configuration for GPIO pins, enabling interrupt triggers on specific events like rising/falling edges, level changes, etc.

HAL Design for GPIO (General-Purpose Input/Output)

Purpose: A GPIO HAL provides a standardized interface for interacting with GPIO pins on a microcontroller or other embedded system. It abstracts away the underlying hardware details, making it easier for application developers to use GPIO pins without needing to delve into the specifics of the hardware.

Key Components:

- **Pin Configuration:** Functions to configure the mode of a GPIO pin (input, output, etc.) and its internal pull-up or pull-down resistors.

- **Read/Write Operations:** Functions to read the value of an input pin or write a value to an output pin.
- **Interrupt Handling:** Functions to enable or disable interrupts on GPIO pins and handle interrupt events.

Example C Implementation:

```
#include <stdio.h>
#include "gpio.h"

int main() {
    // Initialize GPIO
    gpio_init();

    // Configure pin 1 as an output
    gpio_set_mode(1, GPIO_MODE_OUTPUT);

    // Write a value to pin 1
    gpio_write(1, 1); // Set pin 1 high

    // Read the value of pin 2
    int value = gpio_read(2);
    printf("Value of pin 2: %d\n", value);

    return 0;
}
```

Explanation

1. The gpio_init() function initializes the GPIO subsystem.
2. The gpio_set_mode() function configures pin 1 as an output.
3. The gpio_write() function writes a value of 1 to pin 1.
4. The gpio_read() function reads the value of pin 2 and prints it to the console.

- **Error Handling:** Implement error handling mechanisms to detect and handle errors that may occur during GPIO operations.
- **Multiple GPIO Pins:** Design the HAL to support multiple GPIO pins and allow the application to control them independently.
- **Interrupt Handling:** If your microcontroller supports interrupts, implement a mechanism for handling GPIO interrupts to trigger events when the pin state changes.
- **Platform-Specific Considerations:** The specific implementation of the GPIO HAL will depend on the target hardware platform and its capabilities.

HAL Design for SPI

HAL Design for SPI (Serial Peripheral Interface)

Purpose: An SPI HAL provides a standardized interface for communicating with devices over the SPI bus. It abstracts away the underlying hardware details, making it easier for application developers to use SPI.

Key Components:

- **Device Selection:** Functions to select the specific SPI device to communicate with.
- **Data Transfer:** Functions for sending and receiving data over the SPI bus.
- **Clock and Mode Configuration:** Functions to configure the SPI clock speed and mode (e.g., mode 0, mode 1, mode 2, mode 3).
- **Interrupt Handling:** Functions for handling SPI interrupts, if supported.

Example C Implementation:

```
#include <stdio.h>
#include "spi.h"

int main() {
  // Initialize SPI
  spi_init();

  // Select device 0
  spi_select_device(0);

  // Configure SPI mode and clock speed
  spi_set_mode(SPI_MODE_0);
  spi_set_clock_speed(1000000); // 1 MHz

  // Send data
  uint8_t data_out = 0x55;
  uint8_t data_in;
  spi_write_read(data_out, &data_in);

  printf("Sent: 0x%02X, Received: 0x%02X\n", data_out, data_in);

  return 0;
}
```

Explanation

1. The spi_init() function initializes the SPI subsystem.
2. The spi_select_device() function selects device 0 for communication.
3. The spi_set_mode() and spi_set_clock_speed() functions configure the SPI mode and clock speed.

4. The spi_write_read() function sends the data data_out to the selected device and receives the response in data_in.
    - **Chip Select:** Some SPI devices require a chip select signal to be asserted before communication can begin. The HAL should provide a function to control the chip select pin.
    - **Data Format:** The HAL should allow for different data formats, such as 8-bit, 16-bit, or 32-bit.
- **Error Handling:** Implement error handling mechanisms to detect and recover from errors during SPI communication.
- **Interrupt Handling:** If your microcontroller supports SPI interrupts, the HAL can provide functions for enabling and handling interrupts.

HAL design for EEPROM and Memory devices

Purpose: A HAL for EEPROM and memory devices provides a standardized interface for accessing and manipulating data stored in these devices. It abstracts away the underlying hardware details, making it easier for application developers to work with memory.

Key Components:

- **Address Space:** Define the address space of the memory device, specifying the range of valid addresses.
- **Read/Write Operations:** Provide functions for reading and writing data to specific memory addresses.
- **Error Handling:** Implement mechanisms to detect and handle errors that may occur during memory access (e.g., read/write failures, address out of bounds).
- **Initialization:** Provide a function to initialize the memory device and ensure it's ready for use.

Example C Implementation:

```
#include "eeprom.h"

int main() {
  // Initialize EEPROM
  eeprom_init();

  // Write data to a specific address
  uint8_t data = 0x55;
  eeprom_write(0x100, &data, 1);

  // Read data from the same address
  uint8_t read_data;
```

```
    eeprom_read(0x100, &read_data, 1);
    printf("Read data: 0x%02X\n", read_data);
    return 0;
}
```

Explanation:

1. The eeprom_init() function initializes the EEPROM device.
2. The eeprom_write() function writes the value data to the memory address 0x100.
3. The eeprom_read() function reads the value at address 0x100 and stores it in read_data.

- **Memory Size:** Determine the total size of the memory device and ensure that your application doesn't exceed its capacity.
- **Write Cycles:** Some memory devices have limited write cycles. Consider implementing mechanisms to minimize write operations and distribute writes evenly across the memory space.
- **Error Correction:** If the memory device supports error correction, implement mechanisms to detect and correct errors.
- **Performance Optimization:** Optimize memory access patterns to improve performance, especially for frequently accessed data.

Practical Activity 3.6.2: Application of embedded c programming in firmware development

Task:

1: Referring to the previous theoretical activities (3.6.1) you are requested to perform the following task.

You have task to develop firmware for a smart home thermostat. Given the hardware specifications (e.g., ARM Cortex-M4 processor, temperature and humidity sensors, Wi-Fi connectivity) and the embedded system requirements (e.g., RTOS, C programming language), how would you design the firmware to ensure it effectively controls the temperature, integrates with other smart home devices, and provides a seamless user experience?

2: List out procedures/steps to be used to perform the given task.

3: Referring to procedures provided on task 2, perform the given task.

4: Present your work to the trainer and whole class

5: Read key reading 3.6.2 and ask clarification where necessary

6: Perform the task provided in application of learning 3.6

Key readings 3.6.2: Application of embedded c programming in firmware development

Steps to follow in development of a smart home thermostat firmware

1. **Hardware Initialization and Configuration**
   - **Include necessary header files:**

     ```
     #include <stdio.h>
     #include "stm32f10x.h" // Assuming STM32F10x microcontroller
     ```
   - **Initialize clock and peripherals:**

     ```
     void init_clock() {
       // Configure clock settings
     }

     void init_gpio() {
       // Configure GPIO pins for sensors and actuators
     }

     void init_adc() {
       // Initialize ADC for temperature and humidity measurements
     }

     // ... other initialization functions
     ```
2. **Sensor Data Acquisition**
   - **Read sensor data:**

     ```
     float read_temperature() {
       // Read temperature from sensor
       return temperature_value;
     }

     float read_humidity() {
       // Read humidity from sensor
       return humidity_value;
     }
     ```
3. **Temperature Control Algorithm**
   - **Implement control logic:**

     ```
     void control_temperature(float target_temperature) {
       // Calculate heating/cooling output based on target and current temperature
       // Adjust heating/cooling elements accordingly
     ```

```
}
```

4. **User Interface**
   - **Implement user input and output:**

```
void user_interface() {
    // Handle user input (e.g., temperature adjustments)
    // Display temperature and other relevant information
}
```

5. **Network Communication**
   - **Establish Wi-Fi connection:**

```
void init_wifi() {
    // Connect to Wi-Fi network
}
```

   - **Send and receive data:**

```
void send_data(float temperature, float humidity) {
    // Send data to a server or other devices
}

void receive_commands() {
    // Receive commands from a server or other devices
}
```

6. **Integration with Other Devices**
   - **Implement protocols and APIs:**

```
void integrate_with_smart_home_hub() {
    // Use appropriate protocols (e.g., MQTT, Zigbee) to communicate with
the hub
}
```

7. **Power Management**
   - **Optimize power consumption:**

```
void enter_low_power_mode() {
    // Put the device into a low-power state
}

void wake_up_from_low_power() {
    // Wake up the device from low-power mode
}
```

8. **Testing and Debugging**
   - **Write test cases:**

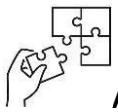
```
void test_temperature_control() {
    // Test temperature control algorithm under various conditions
}
```

// ... other test cases

- **Use debugging tools:**
  // Use a debugger to step through the code, set breakpoints, and inspect variables

Points to Remember

- **Data type** in C define the kind of values a variable can hold and the operations that can be performed on them.
- **Embedded C Programming:** Embedded C programming is a specialized subset of the C programming language that is tailored specifically for developing software intended for embedded systems.
- **Structures and Unions:** Structures and unions are two user-defined data types in C that enable programmers to group related variables of different data types into a single entity. A structure allows you to create a complex data type that can contain multiple members, each potentially of different types.
- **Bit Fields:** Bit fields are a feature in the C language that enables packing multiple variables of different data types into a single memory location by specifying the exact number of bits allocated for each variable within a structure.
- **Preprocessor Directives**: Preprocessor directives are instructions given to the C preprocessor—a program that processes your source code before it gets compiled into machine code. These directives begin with the '#' symbol and include commands such as #define for defining macros or constants, #include for including header files, and #ifdef for conditional compilation based on defined macros.
- **Steps to follow in development thermostat firmware.**
  1. Hardware Initialization and Configuration
  2. Sensor Data Acquisition
  3. Temperature Control Algorithm
  4. User Interface
  5. Network Communication
  6. Integration with Other Devices
  7. Power Management
  8. Testing and Debugging



Application of learning 3.6.

ABCD Company, Given the task of developing firmware for a new smart home security system with features like motion detection, door/window sensors, and remote monitoring, what essential components must be included, and what challenges and considerations should be addressed during the development process?

Indicative content 3.7: Implementation firmware modules

Duration: 5 hrs

Theoretical Activity 3.7.1: Description of firmware modules

Tasks:

1: Answer the following questions:

i. Describe firmware modules

2: Provide the answers for the asked questions and write them on flipchart/papers.

3: Present your findings to the trainer or your colleagues.

4: trainees follow clarification if any.

5: read the Key readings 3.7.1 in their manuals.

Key readings 3.7.1.: Description of firmware modules

Firmware modules are software components embedded within electronic devices that control specific functionalities. They reside on hardware devices like microcontrollers, allowing them to perform various tasks, such as managing input/output operations, handling communication protocols, controlling hardware components, and executing specific functions.

These modules form the core components of firmware in many embedded systems. Each serves a specific purpose in ensuring the proper functioning of the device.

Initialization Module:

Initializes hardware and software components during system startup. In firmware, an initialization module refers to a section of code responsible for setting up and preparing various hardware components, peripherals, and system parameters when a device or system is powered on or reset. This module typically executes during the boot process to ensure that all essential elements of the hardware are correctly configured and ready for the system to operate.

- **Initialization Module in C**

```
void initialize_system() {
    // Initialize hardware components (e.g., clock, GPIO, peripherals)
    // Configure system settings (e.g., interrupts, timers)
    // ...
```

}

e) **Data Acquisition Module:**

Collects raw data from sensors or external sources. A Data Acquisition Module (DAQ) in firmware refers to a software component responsible for gathering, processing, and potentially storing or transmitting data from various sensors, devices, or sources. It typically involves interfacing with hardware components to capture analog or digital signals, converting them into a digital format, and managing the flow of data within a system.

Data Acquisition Module in C

```
void acquire_data(void) {
   // Read sensor data (e.g., temperature, humidity, pressure)
   // Store data in a buffer
   // ...
}
```

f) **Data Processing Module:** Processes the acquired data, performing calculations or transformations as required. This module within firmware could be tailored to the specific needs of the device or system it operates in. For instance, in a sensor device, the data processing module might focus on interpreting sensor readings, while in a communication device, it could manage the incoming and outgoing data packets.

```
void process_data(void) {
   // Perform calculations or analysis on acquired data
   // Update internal state or output results
   // ...
}
```

g) **Control Module:**

Manages the overall behavior and control of the system based on input, sensors, and processed data. A control module in firmware typically refers to a specific section or component of software code embedded in a device's firmware that manages or controls certain functions or features.

```
void control_system(void) {
   // Make decisions based on data and system state
   // Send control signals to actuators or other devices
```

```
    // ...
}
```

h) **Communication Module:**

Facilitates communication between the embedded system and external devices or networks. This module typically includes protocols, drivers, and algorithms necessary for transmitting and receiving data between devices

```
void send_data(void) {
    // Transmit data over a communication channel (e.g., UART, I2C, SPI)
    // ...
}

void receive_data(void) {
    // Receive data from a communication channel
    // Process and store the received data
    // ...
}
```

i) **Memory Management Module:**

Manages memory allocation and deallocation for efficient use of available memory resources. The Memory Management Module in firmware plays a crucial role in ensuring that the system's memory is efficiently utilized, protected, and managed for the smooth operation of the overall computing system.

```
void allocate_memory(size_t size) {
    // Allocate memory for data storage
    // ...
}

void free_memory(void *ptr) {
    // Free allocated memory
    // ...
}
```

j) **Timer Module:**

Provides timing-related functionalities, often used for scheduling tasks or generating time-related events. It is a critical part of embedded systems and software applications that require time-sensitive operations or periodic tasks. In firmware development, timer modules often rely on hardware timers

provided by microcontrollers or processors. Firmware configures these timers by setting their parameters like frequency, mode of operation (e.g., one-shot or continuous), and the desired duration.

```
void start_timer(uint32_t delay) {
   // Start a timer with the specified delay
   // ...
}

void stop_timer(void) {
   // Stop the timer
   // ...
}
```

k) **Interrupt Handler Module:**

Handles interrupts, allowing the system to respond to external events or asynchronous signals. In firmware, interrupt handlers need to be fast and efficient, as they deal with real-time events and must respond quickly to prevent data loss, system instability, or other issues. Writing robust interrupt handlers often involves low-level programming and a deep understanding of both the hardware and software interactions within the system.

```
void interrupt_handler(void) {
   // Handle the interrupt (e.g., read sensor data, process data)
   // ...
}
```

✓ **Power Management Module:** Controls power-related functionalities to optimize energy consumption and extend battery life in portable devices The Power Management Module in firmware refers to a component or module within a device's firmware that controls and manages power-related functionalities. It's responsible for overseeing power-related tasks.

```
void enter_low_power_mode(void) {
   // Put the system into a low-power state
   // ...
}

void wake_up_from_low_power_mode(void) {
   // Wake up the system from low-power mode
```

```
    // ...
}
```

✓ **User Interface Module:** Manages interactions with users, providing input/output functionalities such as displays, buttons, or touch interfaces. In firmware, the UI Module plays a crucial role in facilitating communication between the user and the underlying functionality of the device. It often involves displaying menus, status indicators, prompts, and other visual elements that allow users to navigate, configure settings, initiate actions, and receive feedback.

```
void update_display(void) {
    // Update the display with relevant information
    // ...
}

void handle_user_input(void) {
    // Process user input (e.g., button presses, touch events)
    // ...
}
```

✓ **Error Handling Module:** Deals with system errors or exceptional conditions, ensuring graceful degradation or recovery from faults. Error handling modules in firmware play a crucial role in ensuring the stability, reliability, and security of devices. They are responsible for managing and responding to errors that occur during the operation of the device or system.

```
void handle_error(int error_code) {
    // Log the error and take appropriate action
    // ...
}
```

✓ **Security Module:** Implements security measures to protect the system from unauthorized access, data breaches, or tampering. Security modules in firmware are crucial, especially in devices connected to networks or handling sensitive information. They help mitigate various security risks and are constantly evolving to counter emerging threats in the cybersecurity landscape.

```
void initialize_security(void) {
    // Configure security features (e.g., encryption, authentication)
```

```
    // ...
}

void check_security(void) {
    // Verify security measures and detect unauthorized access
    // ...
}
```

✓ **Bootloader Module:** Responsible for the initial booting of the system, often loading the operating system or firmware. The main function of the bootloader is to perform the initial system checks, load the operating system kernel or application into the memory, and then transfer control to that code.

In embedded systems and devices, the bootloader is particularly essential because it is the first software that runs when the device is powered on. It ensures that the hardware components are functional and loads the necessary software components for the device to start operating

```
void bootloader_init(void) {
    // Initialize the bootloader
    // ...
}

void load_application(void) {
    // Load the main application firmware
    // ...
}
```

✓ **Diagnostic and Debugging Module:** Aids in diagnosing issues, providing tools and functionalities to debug and troubleshoot the system during development or operation. Its primary purpose is to assist in identifying, analyzing, and resolving issues or errors that may occur during the operation of the device.

```
void log_debug_message(const char *message) {
    // Log a debug message for troubleshooting
    // ...
}

void enable_debug_mode(void) {
    // Enable debug features (e.g., logging, breakpoints)
    // ...
```

}

In firmware development, these modules are designed and integrated to work cohesively, ensuring the embedded system's proper operation, reliability, and security. Each module may have specific functions and interfaces tailored to the requirements of the device or system it supports.

Practical Activity 3.7.2: Implementing firmware modules

Notes to the trainer

1: Referring to the previous theoretical activities (3.7.2) you are requested to perform the following task.

How would you implement a firmware module to control an LED on a microcontroller? Describe the steps for initializing the microcontroller pin, turning the LED on, and turning it off.

2: List out procedures/steps to be used to perform the given task.

3: Referring to procedures provided on task 2, perform the given task.

4: Present your work to the trainer and whole class

5: Read key reading 3.7.2 and ask clarification where necessary

6: Perform the task provided in application of learning 3.7

Key readings 3.7.2. Implementing firmware modules

The steps of developing firmware modules and c programming codes

1. Requirement Gathering

- Define the functionality and requirements of the firmware module (e.g., controlling an LED, reading a sensor).
- **No code at this stage.**

2. Design the Module Architecture

- Outline the structure of the module, including the main functions and interfaces.
- **No code at this stage.**

3. Choose the Appropriate Tools and Frameworks

- Set up your development environment, such as IDEs like Arduino IDE or STM32CubeIDE.
- **No code at this stage.**

4. Write the Code

- Initialize hardware components and implement core functionality. Here's an example for an LED control module:

```
#include <avr/io.h>     // For AVR microcontroller

#define LED_PIN PB0     // Define pin for LED

void initLED() {
  // Set LED_PIN as an output
  DDRB |= (1 << LED_PIN);
}

void turnOnLED() {
  PORTB |= (1 << LED_PIN);   // Set LED_PIN high to turn on LED
}

void turnOffLED() {
  PORTB &= ~(1 << LED_PIN);  // Set LED_PIN low to turn off LED
}
```

5. Test and Debug the Code

- Use simple test routines to verify the functionality of the firmware module. Example test code for the LED:

```
int main() {
  initLED();  // Initialize the LED

  while (1) {
    turnOnLED();   // Turn on the LED
    _delay_ms(1000); // Wait for 1 second
    turnOffLED();  // Turn off the LED
    _delay_ms(1000); // Wait for 1 second
  }

  return 0;
}
```

6. Optimize the Module

- Review the code for efficiency, such as reducing power consumption in low-power applications. Use sleep modes if supported:

```
void goToSleep() {
  // Set MCU to sleep mode (for example, using AVR)
  sleep_mode();  // Requires setting sleep mode configuration elsewhere
}
```

7. Integration with the Full System

- Integrate the module with other parts of the firmware, ensuring proper communication and functionality.

```
// Example integration with a sensor
void readSensorData() {
   // Code to read data from a sensor (e.g., analog input)
   uint16_t sensorValue = ADC_Read();  // Hypothetical function to read ADC
   // Process the sensor value as needed
}
```

8. Documentation

- Document the code and module functionality using comments and tools like Doxygen.

```
/**
 * @brief Initializes the LED pin as an output.
 */
void initLED();
```

9. Version Control and Maintenance

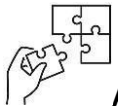
- Use Git or another version control system to manage changes. Example Git commands:

```
git init                # Initialize a new Git repository
git add .               # Add all files to staging
git commit -m "Initial commit of LED control module"  # Commit changes
```

Points to Remember

- Firmware modules are software components embedded within electronic devices that control specific functionalities.
- Firmware development, these modules are designed and integrated to work cohesively, ensuring the embedded system's proper operation, reliability, and security**.**
- Each module may have specific functions and interfaces tailored to the requirements of the device or system it supports.
- **Key steps for developing firmware modules:**

1. Requirement Gathering
2. Design the Module Architecture
3. Choose the Appropriate Tools and Frameworks
4. Write the Code
5. Test and Debug the Code
6. Optimize the Module
7. Integration with the Full System
8. Documentation
9. Version Control and Maintenance



Application of learning 3.7.

The ABCD needs a technician to develop firmware module of their thermostat, how would you apply the implementation of firmware modules for a new smart thermostat that is capable of controlling temperature, humidity, and fan speed? How would you ensure integration with a smart home hub and create a user-friendly interface while focusing on functionality, efficiency, and security?

Indicative content 3.8: Writing drivers source code

Duration: 5 hrs

Theoretical Activity 3.8.1: Description of writing drivers source code

Tasks:

1: Answer the following questions:

i. Identify drive interface

ii. Describe memory-mapping methodologies.

2: Provide the answers for the asked questions and write them on flipchart/papers.

3: Present your findings to the trainer or your colleagues.

4: trainees follow clarification if any.

5: read the Key readings 3.8.1 in their manuals.

Key readings 3.8.1.: Description of Writing drivers source code

Writing drivers involves a detailed understanding of hardware and specific operating system requirements. Depending on the device and the platform, drivers can vary significantly.

- **Identification of Driver Interfaces**

  Driver interfaces refer to the methods and protocols through which the operating system interacts with and manages hardware devices. Identifying driver interfaces involves understanding the various ways in which drivers communicate with the operating system and hardware.

a. Driver interfaces

1. **Device File Interface:** In Unix-like systems, devices are represented as files in the file system. Drivers often interact with devices through device files (e.g., /dev/sda for a hard disk). Read, write, open, close, and other file operations are used to communicate with the device.
2. **Kernel-to-User Interface:** Drivers provide interfaces for user-level applications to interact with the hardware. These interfaces might involve system calls, IOCTL (input/output control) commands, or shared memory access.
3. **Hardware Abstraction Layer (HAL) Interfaces:** Some operating systems use a hardware abstraction layer to provide a uniform interface for drivers. This abstraction shields drivers from hardware specifics, allowing easier development of drivers for different hardware devices.

4. **Bus Interfaces:** Drivers communicate with devices connected to various buses (e.g., PCI, USB, I2C). The driver needs to understand the protocols and commands specific to these buses to manage communication with devices connected to them.
5. **Interrupt Handling:** Hardware devices often use interrupts to signal the CPU that they need attention. Drivers must handle interrupts efficiently, acknowledging them and responding appropriately.
6. **Memory-Mapped I/O:** Some drivers use memory-mapped I/O, where the hardware device's registers are mapped to memory addresses. Drivers read from and write to these memory locations to control the hardware.
7. **Driver APIs and Libraries:** Operating systems provide APIs and libraries that drivers can use to access certain functionalities. These might include libraries for graphics, networking, sound, etc., which provide a higher-level interface for drivers to interact with.
8. **Plug and Play (PnP) Interfaces:** Modern systems support plug-and-play functionality, where devices can be added or removed dynamically. Drivers need to support PnP interfaces to handle device discovery, configuration, and initialization.

Understanding these interfaces is crucial when developing drivers. Different hardware may require different interfaces, and proper interaction with the OS is essential for the driver to function correctly.

b. Device Driver Models

Device driver models are software frameworks or architectures that enable communication between the operating system (OS) and hardware devices. They define how the OS interacts with various hardware components, allowing for standardization and ease of development.

Device driver models

1. **Monolithic Model:** This traditional model involves drivers tightly integrated into the operating system kernel. They have direct access to the hardware and can execute privileged instructions. While efficient, any driver issue can potentially crash the entire system.

2. Layered Model: In this model, drivers are organized into layers based on their functionality. Each layer abstracts hardware interfaces, allowing for easier debugging and maintenance. It enhances system stability by isolating driver failures to specific layers.

3**. Microkernel Model:** Here, device drivers run as user-space processes rather than in the kernel. This design improves system stability and security since driver errors

are less likely to crash the entire system. Communication with hardware occurs through message passing between the microkernel and drivers.

4. **Unified Driver Model (UDM):** UDM aims to create a single driver model that works across different hardware types and platforms. This model simplifies driver development by providing a uniform interface for various devices.

5. **Framework-based Model:** Operating systems like Windows use frameworks such as the Windows Driver Model (WDM) or Windows Driver Framework (WDF) to standardize driver development. These frameworks provide a set of APIs, reducing the complexity of driver coding and ensuring compatibility.

Each model has its own advantages and trade-offs, balancing factors like performance, security, and ease of development. The choice of a particular model often depends on the specific requirements of the operating system and the hardware it supports.

c. **Driver module definition**

A driver module, in the context of computer systems and software, typically refers to a component that enables communication and interaction between hardware devices or peripherals and the operating system. It is a piece of code that allows the operating system to control or communicate with a specific hardware device.

These driver modules act as intermediaries, providing an interface for the operating system to send commands, receive data, or manage the functions of hardware components such as printers, graphic cards, network adapters, and more. Drivers can be either built into the operating system or installed separately. They're crucial for ensuring that hardware devices function correctly and efficiently within a given computing environment.

d. **Encapsulation and data hiding**

- **Encapsulation** involves bundling the data (attributes) and methods (functions or procedures) that operate on the data within a single unit or class. This mechanism restricts direct access to some of the object's components, usually providing public methods or interfaces to interact with the data. The idea is to hide the internal workings and state of an object, allowing controlled access and modification. This helps in ensuring the integrity of the data and prevents unintended interference from external code
- **Data hiding** is a principle related to encapsulation that emphasizes restricting the visibility of certain aspects of an object. It involves making the internal state of an object inaccessible to the outside world, except through designated methods or interfaces. By hiding the implementation details and exposing only

necessary functionalities, data hiding ensures that the object's integrity and consistency are maintained.

Encapsulation and data hiding contribute to building robust, modular, and secure code. They promote better code maintenance, reduce dependencies, and enable easier modifications or enhancements without affecting the rest of the codebase.

e. **Abstract data types**

Abstract Data Types (ADTs) are a key concept in computer science that represent a logical mathematical model for data types. They define a set of operations and behaviors without specifying their implementation details.

Think of ADTs as a blueprint or a set of rules that describe how a data type should behave, without specifying how these behaviors are actually implemented.

- **Abstract data types (ADTs)**

1. **Stack:** An abstract data type that follows the Last-In-First-Out (LIFO) principle, allowing operations like push (adding an element to the top) and pop (removing the top element).

2. **Queue:** An abstract data type that follows the First-In-First-Out (FIFO) principle, enabling operations like enqueue (adding an element to the rear) and dequeue (removing an element from the front).

3**. List:** An abstract data type representing an ordered collection of elements that may allow various operations such as insertion, deletion, traversal, and retrieval.

4**. Map (Dictionary**): An abstract data type that stores a collection of key-value pairs and allows operations like insertion, deletion, and lookup based on keys.

f. **Callback functions**

Refer to a mechanism where a function pointer is passed as an argument to another function, allowing the receiving function to execute the specified callback function at a later time or in response to a particular event.

In embedded systems or firmware development, callback functions are commonly used in scenarios where asynchronous or event-driven operations occur. For instance, in handling interrupts, managing asynchronous I/O operations, implementing event-driven frameworks, or interacting with hardware peripherals, callback functions are quite useful

#include <stdio.h>

```
// Define a callback function type
typedef void (*InterruptHandler)(void);

// Simulate hardware interrupt
void simulateInterrupt(InterruptHandler callback) {
   printf("Interrupt occurred...\n");
   callback();
}

// Callback function 1
void callbackFunction1() {
   printf("Callback function 1 called!\n");
}

// Callback function 2
void callbackFunction2() {
   printf("Callback function 2 called!\n");
}

int main() {
   // Simulate an interrupt and pass callbackFunction1
   simulateInterrupt(callbackFunction1);

   // Simulate another interrupt and pass callbackFunction2
   simulateInterrupt(callbackFunction2);

   return 0;
}
```

- **InterruptHandler** is a function pointer type that represents the signature of the callback function.

- **SimulateInterrupt** is a function that simulates a hardware interrupt. It takes a callback function as an argument and calls it when an interrupt occurs.
- **CallbackFunction1 and callbackFunction2** are two different callback functions that handle the interrupt in their respective ways. When simulateInterrupt is called with a specific callback function, it triggers an interrupt and calls the provided callback function.

g. **Error Handling**

Error handling in drivers' source code is crucial for robustness and stability in software. Drivers are software components that enable communication between the operating system and hardware **devices.**

Here are some common practices for error handling in driver source code:

1. Return Values: Functions in drivers often return error codes or status values to indicate success or failure. These return values should be checked by the calling functions to handle errors appropriately.

2. Error Codes: Define clear error codes or enumerations to represent different types of errors that might occur. These codes help identify the nature of the error and aid in debugging.

3. **Error Propagation**: Propagate errors up the call stack to higher-level functions or the operating system if they cannot be handled within the driver itself. This ensures that errors are appropriately dealt with at higher levels of the software stack.

4. **Logging and Debugging:** Use logging mechanisms to record errors and relevant information like timestamps, error codes, and context. This helps in diagnosing issues during development and debugging.

5. Graceful Recovery: Attempt to recover from errors when possible. For instance, resetting a device or retrying a failed operation might resolve certain errors without causing the entire system to fail.

6. **Resource Cleanup:** Ensure proper cleanup of resources (memory, handles, file descriptors, etc.) in case of errors to prevent memory leaks or resource exhaustion.

7. **Robustness Testing:** Test the driver code thoroughly by simulating different error scenarios. This includes boundary cases, unexpected inputs, and hardware failures to validate the driver's behavior in adverse conditions.

8. **Documentation:** Document error handling strategies, expected error cases, and how errors are propagated or handled within the driver code. This

documentation aids other developers in understanding and maintaining the code.

9. **Exception Handling (if applicable):** In languages or environments that support exceptions, use exception handling mechanisms to catch and handle exceptional conditions that may arise during driver execution.

10. **Alerts and Notifications:** In some cases, especially for critical errors, it might be necessary to generate alerts or notifications to inform system administrators or users about the error state. Remember, error handling in driver code should be thorough, well-documented, and designed to ensure that the driver behaves predictably even in the presence of unexpected or erroneous conditions.

h. Memory Mapping Methodologies

Memory mapping refers to the technique of associating files on disk with a portion of memory. It allows direct, efficient access to files by mapping their contents into a virtual address space.

- **Mapping Memory Directly:** In this methodology, memory is accessed directly using addresses. Programs can interact with specific memory locations by directly referencing their addresses. This approach requires a good understanding of memory addresses and can be error-prone if not handled carefully, leading to issues like segmentation faults or memory corruption if used improperly.
- **Mapping Memory with Pointers:** Pointers are variables that store memory addresses. They are used to indirectly access memory locations, providing flexibility and dynamic memory allocation. Pointers can dynamically allocate memory during runtime, making it easier to manage memory and data structures like arrays, linked lists, and trees. They allow for more versatile memory access and manipulation compared to direct memory mapping.
- **Mapping Memory with Structures:** Structures (or structs) in programming languages allow developers to define a composite data type that can hold multiple variables of different types. When dealing with memory mapping, structures can be used to organize and map memory by grouping related variables together.

  This approach aids in creating complex data structures that reflect the relationships between different pieces of data, enhancing code readability and maintainability.

i. Memory mapping provides several advantages:

- **Efficiency:** Accessing memory-mapped files can be faster than traditional file I/O operations because it reduces the need for explicit read/write operations.

- **Simplified Access:** It simplifies file handling by allowing programmers to treat files as if they were large arrays in memory, enabling direct manipulation.
- **Sharing Data:** Memory mapping enables multiple processes to access the same data concurrently by mapping the same file into their memory space.

j. Using Pointer arrays in Driver design

Pointer arrays are arrays whose elements are pointers to other variables rather than actual data. using pointer arrays can be quite handy, especially in driver design where you might need to manage various resources or data structures dynamically. Pointer arrays are arrays that hold memory addresses pointing to other variables or data structures rather than holding the actual data directly. In driver design, you might encounter scenarios where you need to manage multiple resources or instances of a particular data structure. Using pointer arrays allows you to efficiently handle these situations. Let us consider an example where you're designing a driver for a simple device that manages multiple sensors. You might use a pointer array to keep track of these sensor instances.

```
#include <stdio.h>

#include <stdlib.h>


// Define a structure for the sensor

typedef struct {

   int sensor_id;

   float value;

   // Other sensor properties

} Sensor;


// Maximum number of sensors the driver can handle

#define MAX_SENSORS 10


// Array to hold pointers to Sensor instances

Sensor* sensor_array[MAX_SENSORS];


// Function to initialize a sensor
```

```
Sensor* initialize_sensor(int id) {
    Sensor* new_sensor = (Sensor*)malloc(sizeof(Sensor));
    if (new_sensor != NULL) {
        new_sensor->sensor_id = id;
        new_sensor->value = 0.0;
        // Initialize other properties
    }
    return new_sensor;
}
// Function to add a sensor to the driver
int add_sensor_to_driver(int id) {
    for (int i = 0; i < MAX_SENSORS; ++i) {
        if (sensor_array[i] == NULL) {
            sensor_array[i] = initialize_sensor(id);
            if (sensor_array[i] == NULL) {
                return -1; // Failed to initialize sensor
            }
            return i; // Return index of added sensor
        }
    }
    return -1; // No space for new sensor
}
// Function to read sensor value by ID
float read_sensor_value(int id) {
    for (int i = 0; i < MAX_SENSORS; ++i) {
        if (sensor_array[i] != NULL && sensor_array[i]->sensor_id == id) {
            return sensor_array[i]->value;
        }
    }
```

```
    return -1; // Sensor not found or uninitialized
}
// Function to release memory allocated to sensors
void release_sensors() {
    for (int i = 0; i < MAX_SENSORS; ++i) {
        if (sensor_array[i] != NULL) {
            free(sensor_array[i]);
            sensor_array[i] = NULL;
        }
    }
}
```

This example demonstrates how you might use a pointer array to manage sensor instances dynamically. The **add_sensor_to_driver** function adds a new sensor to the driver, **read_sensor_value** reads the value of a specific sensor, and **release_sensors** frees the allocated memory when the driver is no longer needed. Pointer arrays provide flexibility in managing multiple instances of resources or data structures in a driver design. However, it is crucial to manage memory properly to avoid memory leaks and undefined behavior.

Practical Activity 3.8.2: Application of device driver development

Task:

1: Referring to the previous theoretical activities (3.8.2) you are requested to perform the following task.

You have task of developing a simple device driver in c programing language.

2: List out procedures/steps to be used to perform the given task.

3: Referring to procedures provided on task 2, perform the given task.

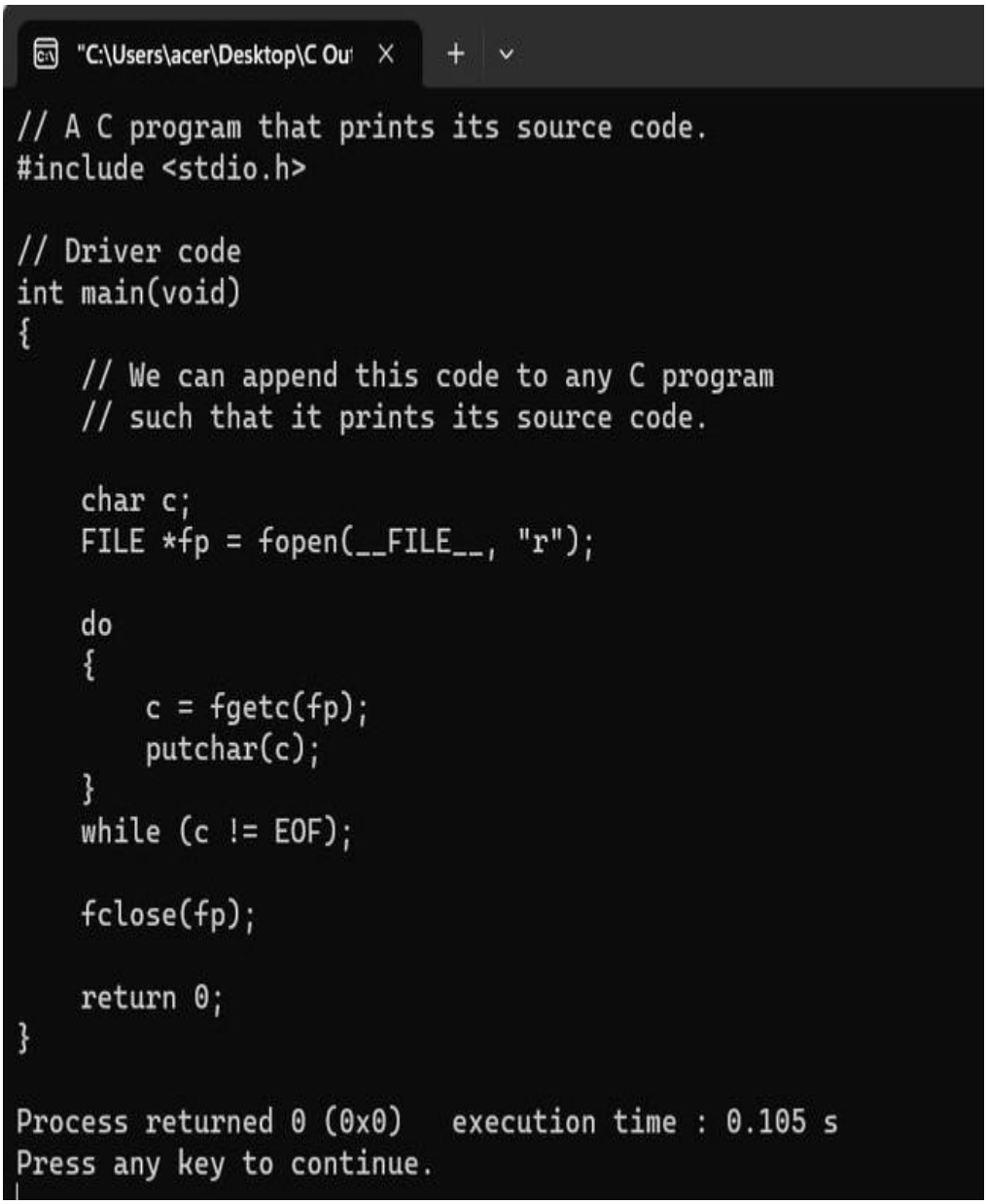
4: Present your work to the trainer and whole class

5: Read key reading 3.8.2 and ask clarification where necessary

6: Perform the task provided in application of learning 3.8

Key readings 3.8.2: Application of device driver development

Steps of developing simple device driver in c programing language



Simple Printer Driver C Programming Code

Creating a simple printer driver in C involves several steps, including defining the necessary structures, implementing the required functions, and handling communication with the printer. Below is a basic example of a unidirectional printer driver that demonstrates these concepts.

1. Include Necessary Headers

```
#include <windows.h>
```

```
#include <winspool.h>
#include <commdlg.h>
#include <gdi.h>
#include <winddi.h>
#include <tchar.h>
#include <strsafe.h>
```

2. Define Printer Structure

This structure will hold information about the printer's state and configuration.

```
struct printer {
   int busy; // Indicates if the printer is busy
   pthread_mutex_t lock; // Mutex for thread safety
};
```

3. Initialize Printer

This function initializes the printer structure.

```
void init_printer(struct printer *p) {
   p->busy = 0;
   pthread_mutex_init(&p->lock, NULL);
}
```

4. Print Function

This function simulates sending data to the printer.

```
void print(struct printer *p, const char *data) {
   pthread_mutex_lock(&p->lock);

   if (p->busy) {
      printf("Printer is busy.\n");
      pthread_mutex_unlock(&p->lock);
      return;
   }

   p->busy = 1; // Set busy flag
   printf("Printing: %s\n", data);

   // Simulate printing time
   sleep(2);

   p->busy = 0; // Reset busy flag
   printf("Print complete.\n");

   pthread_mutex_unlock(&p->lock);
```

```
}
```

5. Main Function

The main function initializes the printer and sends a print command.

```
int main() {
    struct printer my_printer;

    init_printer(&my_printer);

    print(&my_printer, "Hello, World!");

    return 0;
}
```

6. Compile and Run

To compile this code, save it as printer_driver.c and use the following command:

```
gcc -o printer_driver printer_driver.c -lpthread
```

Then run it using:

```
./printer_driver
```

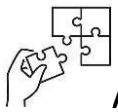
This simple example demonstrates how to create a basic structure for a printer driver in C. It includes initialization, locking mechanisms for thread safety, and a simulated print function.

Points to Remember

- **Driver interfaces** are Device File Interface, Kernel-to-User Interface, Hardware Abstraction Layer (HAL) Interfaces, Bus Interfaces, Interrupt Handling, Memory-Mapped I/O, Driver APIs and Libraries and Plug and Play (PnP) Interfaces.
- **Memory Mapping Methodologies** refers to the technique of associating files on disk with a portion of memory. They include the following technologies: Mapping Memory Directly, Mapping Memory with Pointers, and Mapping Memory with Structures.
- **Steps to follow in development device driver using c programming language.**
    - **1.** Include Necessary Headers
    - **2** Define Printer Structure
    - **3.** Initialize Printer
    - **4.** Print Function

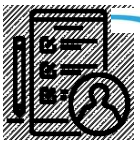
5. Main Function

6. Compile and Run



Application of learning 3.8

ABCD Company need a firmware developer for driver development. You are requested to write a printer driver to support both local and network-connected printers, while ensuring seamless integration with various operating systems and applications.



Learning outcome 3 end assessment

Theoretical assessment

I.True/False Questions

a) UART, SPI, and I2C are all types of communication protocols.

b) Dynamic memory allocation is always preferred over static memory allocation in embedded systems.

c) The .data segment in memory contains uninitialized global and static variables.

d) APIs and HALs serve the same purpose in firmware development.

e) The bootloader module is responsible for updating the firmware and managing the boot process.

II.Multiple Choice Questions

1.Which of the following is NOT a common type of memory segment in embedded systems?

a) .data

b). bss

c) Heap

d) .exe

2.What does IDE stand for in the context of software development?

a) Integrated Development Environment

b) Interface Design Engine

c) Internal Debug Emulator

d) Intelligent Device Executor

3. Which communication protocol is best suited for short-distance, high-speed communication between integrated circuits?

a) UART

b) USB

c) I2C

d) Ethernet

4. What is the primary purpose of a Hardware Abstraction Layer (HAL)?

a) To provide a direct interface to hardware

b) To abstract hardware-specific details from the application code

c) To replace device drivers

d) To optimize memory usage

5.Which of the following is NOT a typical module in firmware development?

a) Data Acquisition Module

b) Control Module

c) Social Media Module

d) Power Management Module

f) What is the main advantage of using bit fields in embedded C programming?
   a) Improved code readability
   b) Increased execution speed
   c) Memory conservation
   d) Enhanced security

g) Which memory allocation method is typically faster but less flexible?
   a) Static Memory Allocation
   b) Dynamic Memory Allocation
   c) Virtual Memory Allocation
   d) Segmented Memory Allocation

h) What is the primary purpose of a callback function in driver design?
   a) To initialize hardware
   b) To handle asynchronous events
   c) To allocate memory
   d) To encrypt data

i) Which of the following is NOT a characteristic of a well-designed API?
   a) Consistency
   b) Simplicity
   c) Hardware dependency
   d) Extensibility

j) What is the main purpose of the Error Handling Module in firmware?
   a) To prevent all errors from occurring
   b) To log errors for later analysis
   c) To detect, report, and manage error conditions
   d) To crash the system when an error occurs

III. **Fill-in-the-blank Questions**

1. The _____ protocol is commonly used for serial communication between microcontrollers and peripherals.
2. In embedded systems, the _____ is responsible for managing the last-in, first-out (LIFO) data structure used for function calls and local variables.
3. _____ memory allocation allows for more efficient use of memory but can lead to fragmentation.
4. The process of _____ an IDE involves setting up compilers, debuggers, and other tools specific to the target hardware.
5. _____ directives are commands for the preprocessor in C programming that are processed before the actual compilation begins.
6. A _____ is a software component that allows the operating system to interact with a hardware device.

7. The _____ module in firmware is responsible for efficiently managing power consumption and implementing power-saving strategies.
8. _____ functions are used in driver design to handle hardware-specific operations without exposing the implementation details.
9. The _____ segment of memory contains initialized global and static variables.
10. A _____ is a programming interface that defines how software components should interact with each other.

Practical assessment

You are tasked with developing firmware for a smart home thermostat system. The thermostat needs to read temperature data from a sensor, control an HVAC system, and communicate with provided interface. Your task is to write the required drivers.

Requirements:

1. Use an STM32F4 microcontroller.
2. Read temperature data from a digital temperature sensor using I2C.
3. Control the HVAC system using GPIO pins.
4. Communicate with a mobile app using UART for debugging and USB for data transfer.
5. Implement efficient memory management.
6. Create a modular firmware architecture.
7. Develop necessary device drivers.

Reference

Doe, J. (2020). *The Firmware Handbook.* (J. Smith, Ed.) Tech Press.

Douglass, B. P. (2016). *Embedded Systems: Design and Applications.* Amsterdam: Elsevier.

Himpe, V. (2015). *Embedded Firmware Solutions: Development Best Practices for the Internet of Things.* New York: Apress.

Ibrahim, D. (2014). *Designing Embedded Systems with 32-Bit PIC Microcontrollers and MikroC.* Oxford: Newnes.

Meer, J. A. (2018). *Designing Embedded Systems with Arduino.* Berlin: Springer.

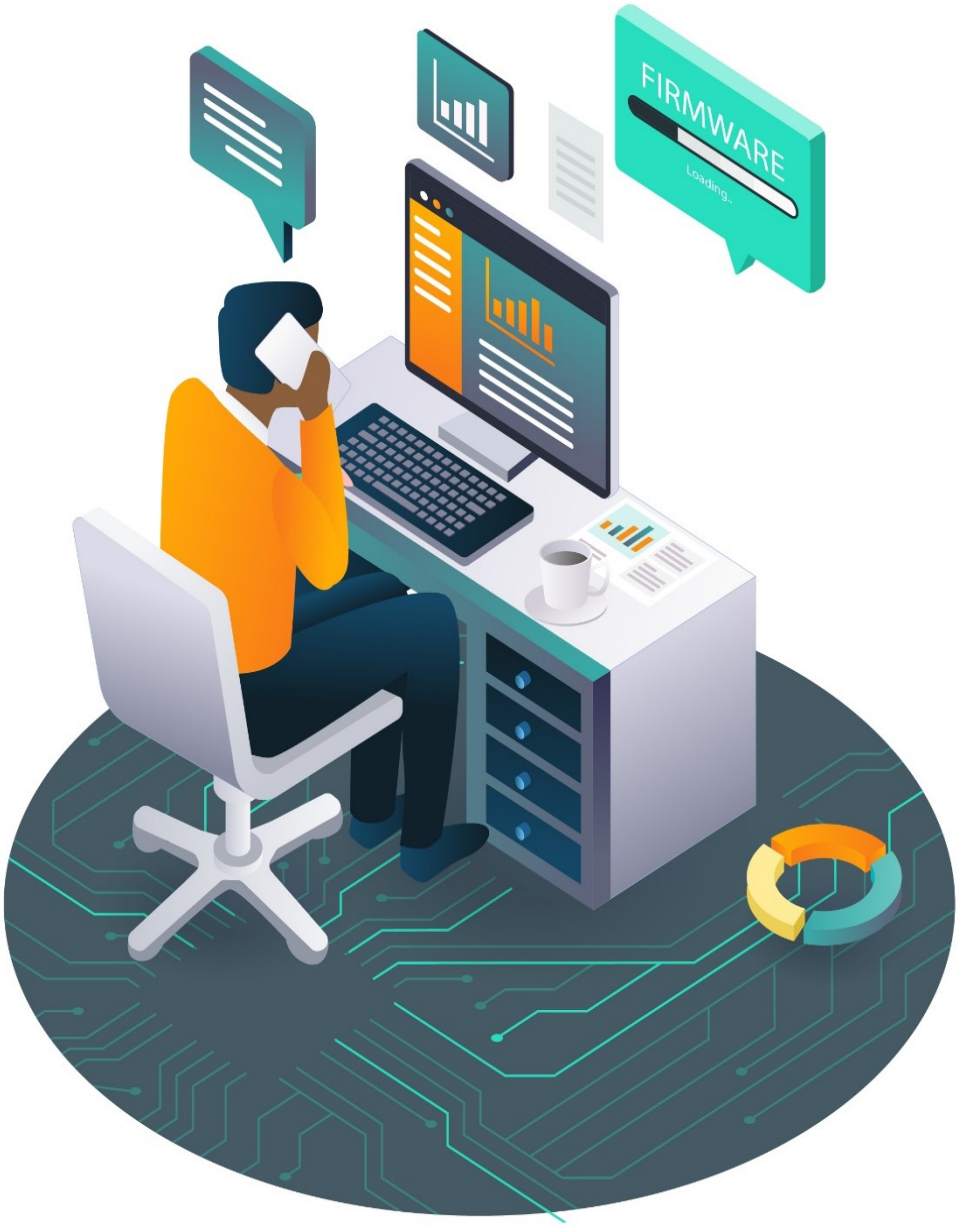
Oshana, R. (2019). *Software Engineering for Embedded Systems.* CRC Press.

Peter Barry, P. C. (2012). *Real-Time Embedded Systems: Design Principles.* Hoboken: Wiley.

Simon, D. (2021). *Firmware Development for Embedded Systems.* Springer.

Valvano, J. (2018). *Embedded Systems: Real-Time Operating Systems.* Wiley.

Wilmshurst, T. (2010). *Designing Embedded Systems with PIC Microcontrollers.* Oxford: Newnes.


FIRMWARE
Loading..

Indicative Contents

4.1 Preparation of Deployment Environment

4.2 Testing the Firmware

4.3 Exporting the Firmware Image

4.4 Documentation of The Firmware

Key Competencies for Learning Outcome 4: Deploy Firmware

Knowledge	Skills	Attitudes
• Identification firmware deployment tools. • Description of testing technics. • Description of .hex file	• Selecting deployment environment for firmware deployment. • Configuring both virtual and physical environments • Implementing testing technics. • Creating of hex file • Selecting target device • Exporting firmware image • Flashing the .hex file to microcontroller • Implementing Firmware technical documentation. • Documenting firmware user manual.	• Having Curiosity • Being Patient and Persistent. • Being Attentive to Detail • Having Adaptability • Having Collaboration • Having Critical Thinking • Being Responsible • Having Ethical Considerations • Being Self-Reliance

Duration: 10 hrs

Learning outcome 4 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Select effectively deployment environment as used in firmware deployment.
2. Identify clearly firmware deployment tools as used in firmware deployment.
3. Describe clearly testing techniques as used in firmware deployment.
4. Document properly firmware user manual as used in firmware documentation.

Resources

Equipment	Tools	Materials
• Computer	• Microsoft Visio • Text Editors • Doxygen • XLoader • Natural Docs • Sphinx • Doxypress	• Internet • Notebooks

Indicative content 4.1: Preparation of deployment environment.

Duration: 1 hrs

Theoretical Activity 4.1.1: Introduction to Deployment Environment.

Tasks:

1: Introduce the activity and answer the following questions

i. Define the term 'environment' in the context of firmware deployment?
ii. Give two examples use to select firmware deployment environment.
iii. Identify firmware deployment tool used in deployment.

2: Provide the answers and present their findings.

3: Agreement of their findings together with trainees and trainer.

4: follow the explanations of trainer.

5: read the Key readings 4.1.1

Key readings 4.1.1.: Introduction to Deployment Environment.

i. **Firmware Deployment Environment**

Refers to the set of tools, processes, and infrastructure required to distribute and install firmware updates onto embedded devices.

ii. **Selection of deployment environment**

- ✓ **Virtual Environment**: use Software-based simulation of hardware; includes cloud or local virtual machines. (Uses software to simulate hardware).
    - Advantages**:** Cheaper, flexible, and easy to set up and change.
    - Disadvantages: May not show real hardware performance; less accurate.
- ✓ **Physical Environment:** Uses actual hardware for testing and deployment.
    - Advantages**:** Provides accurate performance data, necessary for final checks.
    - Disadvantages**:** More expensive, less flexible, takes more time to set up.

iii. Identification of Firmware Deployment Tools

- **IDEs (Integrated Development Environments)**: These are user-friendly programs that allow you to write and upload firmware. Examples include **Keil** and **IAR**.
- Tools that combine coding, debugging, and compiling in one place.
- Examples: **Keil µVision**, **IAR Embedded Workbench**, **Atmel Studio**.

- ➢ **Command-Line Tools**:
    - Used for deploying firmware using terminal commands.
    - Examples: **dfu-util** (for USB devices), **fwupd** (for Linux).
- ➢ **In-System Programmers (ISP)**:
    - Directly flash firmware onto microcontrollers while connected.
    - Examples: **JTAG**, **ST-Link**.
- ➢ **Over-the-Air (OTA) Tools**:
    - Used for remote firmware updates without needing to connect to the device.
    - Examples: **ESP32 OTA**, **NXP's MCU OTA**.
- ➢ **Debugging and Flashing Tools**:
    - Tools for testing and deploying firmware on actual hardware.
    - Examples: **OpenOCD**, **SEGGER J-Link**(EurthTech)(Embedded Wala)(Mikroe Help).

These tools help developers deploy and manage firmware on devices efficiently, whether it's remote or through direct connection.

Practical Activity 4.1.2: Preparing deployment environment.

Task:

1: Read key reading 4.1.2 and ask clarification where necessary

2: Referring to the previous theoretical activities (4.1.1) and step provided in key reading 4.1.2 you are requested to go to the computer lab to prepare deployment environment (virtual or physical) and set it up for testing.

3: Present your work to the trainer and whole class.

4: Ask question where necessary.

Key readings 4.1.2: Preparing deployment environment

Here are the steps for preparing a firmware Virtual Deployment Environment:

1. **Select Virtualization Software** – Choose a platform like XLoader, VMware, or QEMU for virtual machine (VM) management.
2. **Install Virtual Machine** – Set up the virtual machine with the required OS (e.g., Linux, Windows) for the target firmware.
3. **Configure Virtual Hardware** – Allocate resources like CPU, memory, storage, and peripherals to match the target system.
4. **Install Cross-Compilation Tools** – Set up compilers, debuggers, and SDKs (e.g., GCC, OpenOCD) for developing and testing firmware.
5. **Prepare Network and Communication** – Configure network settings and communication interfaces (e.g., serial, USB) for device emulation.
6. **Install Target Firmware** – Load the firmware binary onto the virtual environment for deployment testing.
7. **Test and Debug** – Use debugging tools to test the firmware's functionality and troubleshoot issues in the virtual environment.
8. **Monitor Performance** – Track system resource usage and performance for optimization and validation.

NOTE: These steps create a virtual space to safely deploy and test firmware before live deployment on actual hardware.

Here are the steps to prepare for **physical firmware deployment**:

1. **Compile or Obtain Firmware**

Develop the firmware using an IDE like **Arduino IDE** or **PlatformIO**, or obtain the precompiled firmware in the form of a **.hex** or **.bin** file.

Ensure the firmware is compatible with the target hardware (e.g., microcontroller or development board).

2. **Set Up the Target Hardware**

Connect the Device**:** Use a USB cable, serial interface, or programmer (e.g., AVR ISP or JTAG) to physically connect the microcontroller or device to your computer.

Ensure the device has power, and all necessary peripherals (sensors, displays, etc.) are connected, if needed.

3. **Select the Deployment Tool**

- Choose a tool based on your hardware platform:
  - Arduino IDE **or** PlatformIO for Arduino-compatible boards.
  - XLoader for simple firmware uploading to AVR-based boards.
  - avrdude for more advanced firmware flashing to AVR microcontrollers.
  - Flip for Atmel microcontrollers.
  - Teensy Loader for Teensy boards.

4. **Configure Tool Settings**

- Set the correct **board type**, **port**, and **baud rate** (if needed).
- Ensure the tool recognizes the connected device (e.g., select the correct COM port in Arduino IDE).

5. **Load and Flash the Firmware**

- Upload the compiled firmware (.hex or .bin) to the target device using the selected tool. For example:
  - In Arduino IDE, click Upload after compiling the code.
  - In XLoader**,** select the firmware file, set the correct COM port, and click Upload**.**
  - Fo**r** avrdude, run a command to upload the firmware file to the microcontroller.

6. **Verify the Deployment**

- Confirm that the upload was successful by checking the tool's status or output.
- For Arduino IDE **or** avrdude**,** you'll typically see a success message after flashing.

7. **Test the Firmware**

- Test the physical hardware to ensure the firmware is functioning correctly.
- Use serial monitors (e.g., in Arduino IDE) or other debugging methods to check outputs or interactions with peripherals.

8. **Troubleshooting (if needed)**

- If the firmware doesn’t work as expected, review the connections, firmware compatibility, and settings.
- Use debugging tools to pinpoint any issues in the deployment or execution.

Summary

Compile or obtain firmware.

Set up and connect the target hardware.

Select the appropriate deployment tool.

Configure tool settings for the hardware.

Flash the firmware to the device.

Verify the upload.

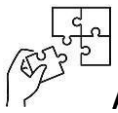
Test the device to ensure functionality.

Troubleshoot if necessary.

These steps help you prepare and execute the physical deployment of firmware onto microcontrollers or embedded devices.

Points to Remember

- **There are two examples used to select firmware deployment which are:** Virtual Environment, Physical Environment
- **Tools used in firmware deployment are:** IDEs (Integrated Development Environments)**,** Command-Line Tools**,** In-System Programmers (ISP)**,** Over-the-Air (OTA) Tools**,** Debugging and Flashing Tools
- Here are the **steps for preparing a firmware Virtual Deployment Environment:**
    1. Select Virtualization Software
    2. Install Virtual Machine
    3. Configure Virtual Hardware
    4. Install Cross-Compilation Tools
    5. Prepare Network and Communication
    6. Install Target Firmware
    7. Test and Debug
    8. Monitor Performance
    - **Here are the steps to prepare for physical firmware deployment**:
        1. Compile or obtain firmware.
        2. Set up and connect the target hardware.
        3. Select the appropriate deployment tool.
        4. Configure tool settings for the hardware.
        5. Flash the firmware to the device.
        6. Verify the upload.
        7. Test the device to ensure functionality.
        8. Troubleshoot if necessary.



Application of learning 4.1.

ABCD Company is company that deploy a firmware for a set of IoT devices. Before proceeding with the deployment, you are requested to prepare the deployment environment and set it up for testing.

Indicative content 4.2: Testing the firmware.

Duration: 3 hrs

Theoretical Activity 4.2.1: Description of testing techniques.

Tasks:

1: Read and answer questions within task described below:

i. What is the definition of testing techniques in firmware testing?
ii. List the different techniques used in firmware testing?
iii. What are the advantages and disadvantages of firmware testing techniques?
iv. What does STLC stand for in firmware testing?

2: Provide the answer for the asked questions and write them on papers or flip chat.

3: Present the findings/answers to the whole class.

4: Ask questions where necessary.

5: For more clarification, read the key readings 4.2.1.

Key readings 4.2.1.: Description of firmware testing techniques

- **Testing techniques in firmware testing**

Firmware testing is crucial to ensure that firmware functions correctly and meets the intended specifications. Various techniques are used to test firmware, each with its advantages and disadvantages.

Firmware testing is essential to ensure that firmware operates as expected. This handout focuses on the practical implementation of two key testing techniques: Unit Testing and Integration Testing.

- **Techniques used in firmware testing:**

Some techniques include:

- ✓ **Unit Testing:** Testing individual components or functions of the firmware in isolation. Purpose of this Verify that each part works correctly on its own.
- ✓ **Integration Testing:** Testing the interactions between different components or modules of the firmware. It Purpose is ensure that combined components work together as expected.

- **Advantages and Disadvantages of firmware testing techniques**
  - **Unit Testing:**
    - Advantages: Early detection of bugs, easy to isolate problems.

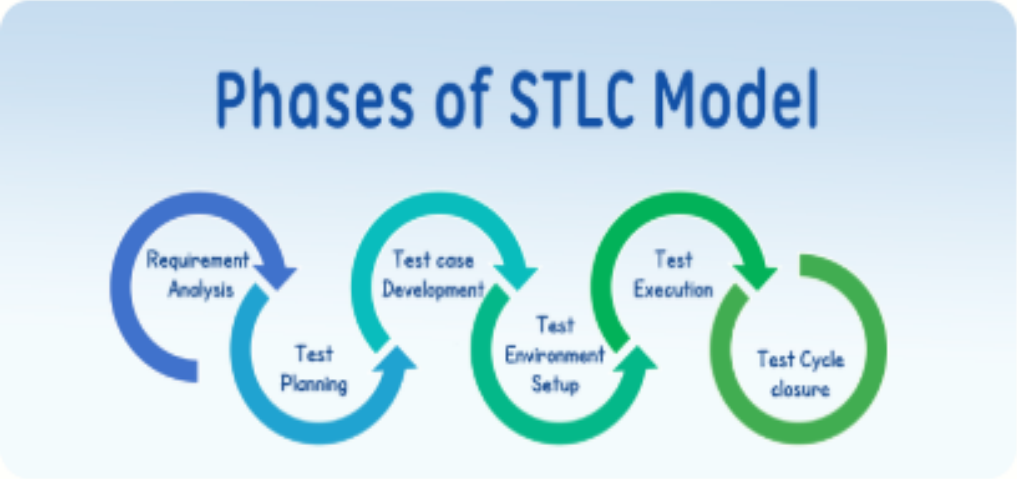
- Disadvantages: Doesn't test interactions between components, may not reflect real-world usage.

- **Integration Testing**
  - Advantages: Identifies issues in interactions, ensures components work together.
  - Disadvantages: Can be complex to set up, may not cover all scenarios.

- **The Software Testing Life Cycle (STLC)**

The Software Testing Life Cycle (STLC) is a systematic process used to ensure the quality and functionality of firmware. It involves a series of phases that guide the testing process from start to finish, ensuring that all aspects of the firmware are thoroughly tested.

The **Software Testing Life Cycle (STLC)** is a series of well-defined phases or steps that guide the process of software testing. It ensures that the software is thoroughly tested before it is released. STLC defines the entire process of testing from the beginning to the end, covering all aspects of test planning, design, execution, and closure.



Here are the key phases of STLC:

1. Requirement Analysis

- **Objective**: Understand the testing requirements based on the project's specifications.
- **Activities**:
  - Review project documentation (e.g., requirement documents, user stories, etc.).
  - Identify testable requirements.
  - Clarify ambiguities in requirements with stakeholders.
  - Determine testing types (functional, non-functional, regression, etc.).

- **Outcome**: Test plan preparation, requirement traceability matrix (RTM).

2. Test Planning

- **Objective**: Define the overall strategy and resources for testing.
- **Activities**:
    - Define the scope, objectives, and test deliverables.
    - Select test strategy (manual vs. automated testing).
    - Assign roles and responsibilities for the testing team.
    - Create a detailed test plan document.
    - Estimate the resources (tools, team size, infrastructure).
- **Outcome**: A test plan document that outlines the testing approach, schedule, and resource requirements.

3. Test Design

- **Objective**: Design the actual test cases and prepare testing resources.
- **Activities**:
    - Develop detailed test cases and test scripts based on the requirements and design documents.
    - Create test data for executing the test cases.
    - Identify test environments (hardware, software, network configurations, etc.).
    - Prepare traceability matrix to map test cases to requirements.
- **Outcome**: Test cases, test scripts, test data, and test environment setup.

4. Test Environment Setup

- **Objective**: Set up the environment in which tests will be executed.
- **Activities**:
    - Set up hardware, software, network, and databases to replicate the production environment.
    - Install required tools and applications for testing.
    - Ensure access to the required resources (e.g., test data, credentials).
- **Outcome**: A stable and ready-to-use test environment.

5. Test Execution

- **Objective**: Execute the designed test cases and report defects.
- **Activities**:
    - Execute the test cases (manual or automated).
    - Record the results (pass/fail).
    - Log defects for failed test cases.
    - Communicate with developers for defect resolution.
- **Outcome**: Test execution reports, defect reports.

6. Defect Reporting and Tracking

- **Objective**: Track and manage defects identified during testing.
- **Activities**:

- Log defects in a defect tracking tool (e.g., Jira, Bugzilla).
- Prioritize and assign severity to defects.
- Work with the development team for defect resolution.
- Retest fixed defects.

- **Outcome**: A defect report and status updates for each defect.

7. Test Closure

- **Objective**: Conclude the testing process and prepare closure reports.
- **Activities**:
  - Prepare test summary reports detailing the test results, defect status, and coverage.
  - Document lessons learned and process improvements.
  - Review the testing process and determine whether all testing goals have been achieved.
  - Release test deliverables and test environment.
  - Archive test cases, test scripts, and other relevant artifacts.
- **Outcome**: Test closure report, archived test artifacts.

Benefits of STLC:

- **Structured Process**: STLC provides a clear and organized approach to software testing, ensuring thorough testing and reducing the risk of defects.
- **Defect Prevention**: By performing detailed analysis and planning upfront, potential issues can be identified early.
- **Traceability**: The STLC maintains a traceability matrix, which helps ensure all requirements are tested and met.
- **Efficiency**: Each phase builds on the previous one, improving overall testing efficiency and reducing the time to market.

Practical Activity 4.2.2: Implementing the Testing Techniques.

Task:

1: Read key reading 4.2.2 and ask clarification where necessary

2: Referring to the previous theoretical activities (4.2.1) and step provided in key reading 4.3.2 you are requested to go to the computer lab and perform the given task bellow:

i. Your task is to implement a variety of firmware testing techniques

3: Present your work to the trainer and whole class.

4: Perform the task provided in application of learning 4.2

Key readings 4.2.2: Implementing testing techniques steps

- **Steps of testing using Unit test**

1. **Identify Testable Units**:
   - Break down the firmware into smaller, testable units (e.g., functions, modules).
   - Determine the key functionality of each unit.
2. **Create Test Cases**:
   - Write test cases that cover different scenarios and inputs for each unit.
   - Include positive (expected behavior) and negative (error handling) tests.
3. **Prepare Test Environment**:
   - Set up a controlled environment where the unit can be tested in isolation.
   - Use mocking or stubbing to simulate interactions with other components.
4. **Execute Tests**:
   - Run the test cases on the individual unit.
   - Use unit testing frameworks (e.g., Google Test, Unity) for automation.
5. **Record and Analyze Results**:
   - Document the test results and check if the unit passed or failed.
   - Review failed tests to identify and fix defects.
6. **Refactor and Retest**:
   - Make necessary code changes to fix any issues found.
   - Retest the unit to ensure that defects are resolved.

Best Practices:

- Ensure test cases are comprehensive and cover edge cases.
- Automate tests to improve efficiency and repeatability.
- Regularly update test cases as the firmware evolves.

Objective: Verify that different components or modules of the firmware work together as expected.

- **Steps of testing using Integration Test**

1. **Define Integration Points**:
   - Identify the interactions and interfaces between different components or modules.
   - Determine the integration scenarios that need testing.
2. **Create Integration Test Cases**:
   - Develop test cases that focus on the interactions between integrated units.

- Include tests for data flow, control flow, and error handling between components.

3. **Prepare Integration Environment**:
   - Set up an environment that includes all the components being integrated.
   - Ensure that dependencies are correctly configured and accessible.
4. **Execute Tests**:
   - Run the integration test cases in the prepared environment.
   - Use integration testing tools and frameworks as needed.
5. **Record and Analyze Results**:
   - Document the results and check if the integrated components function correctly together.
   - Investigate and resolve any issues that arise during integration.
6. **Validate and Retest**:
   - Validate that integration issues are fixed.
   - Re-run tests to ensure that changes do not affect the overall integration.

Best Practices:

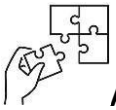
- Test with real or simulated data to mimic actual usage scenarios.
- Focus on critical integration points and scenarios that may cause issues.
- Regularly perform integration tests as new components are added or changed.

Points to Remember

- **Firmware testing** is crucial to ensure that firmware functions correctly and meets the intended specifications. Various techniques are used to test firmware
- Firmware testing is essential to ensure that firmware operates as expected. To implement it require some testing techniques that include for **Unit Testing** and **Integration Testing.**
- **Advantage** of testing Unit is ensuring components work together while **Disadvantage** Can be caused of complex to set up.
- **The Software Testing Life Cycle (STLC)** is a systematic process used to ensure the quality and functionality of firmware.
- **Steps of testing using Unit test**:
  1. Identify Testable Units
  2. Create Test Cases
  3. Prepare Test Environment
  4. Execute Tests
  5. Record and Analyse Results
  6. Refactor and Retest

- **Steps of testing using Integration Test**:

1. Define Integration Points
2. Create Integration Test Cases
3. Prepare Integration Environment
4. Execute Tests
5. Record and Analyse Results
6. Validate and Retest



Application of learning 4.2.

ABCD Company developing a smart home thermostat. The thermostat controls home heating and cooling, integrates with the provided interface, and connects to the internet via Wi-Fi for remote management. Your task is to implement a variety of firmware testing techniques to ensure the thermostat's firmware operates correctly, safely, and efficiently before it is released to customers.

Indicative content 4.3: Exporting the firmware image.

Duration: 3hrs

Theoretical Activity 4.3.1: Description on .hex file.

Tasks:

1: Answer the following questions:

I. Describe .hex file used for when exporting firmware.

2: Provide the answer for the asked questions and write them on papers or flip chat.

3: Present the findings/answers to the whole class.

4: Ask questions where necessary.

5: For more clarification, read the key readings 4.3.1.

Key readings 1.1.1.: Description on. hex file

A. hex file

Hex file, also known as **Intel HEX format**, is a text-based file format used to represent binary data in a human-readable form. This format is commonly used to store and transfer firmware, microcontroller code, or other binary data to be programmed into electronic devices.

Characteristics of an hex File

Format: The file contains ASCII-encoded hexadecimal data representing machine instructions, memory addresses, and other configuration data.

Intel HEX Format: The data is organized in lines (records) of hexadecimal numbers, each line typically following this structure:

- ✓ BBAAAATTDD...DDCC
  - **BB**: Byte count (number of data bytes in the record).
  - **AAAA**: Address (memory location where the data should be loaded).
  - **TT**: Record type (defines the type of the record, like data record, end of file, etc.).
  - **DD**: Data (the actual machine code or data to be loaded into memory).
  - **CC**: Checksum (used to verify the integrity of the data).

Usage:

- ✓ **Flashing Firmware:** This file is used by programmers or bootloaders (such as **AVRDUDE**, **Teensy Loader**, **XLoader**) to upload firmware to microcontrollers (e.g., Arduino, STM32, PIC).

- ✓ **Memory Layout**: The file describes not only the data but also where in the device's memory the data should be placed, which is crucial for embedded systems.
- ✓ **Readability**: Though the data inside the file is human-readable as hexadecimal text, it is not directly readable as source code.

Practical Activity 4.3.2: Creating, exporting, and flashing hex file.

Task:

1: Read key reading 4.3.2 and ask clarification where necessary

2: Referring to the previous theoretical activities (4.3.1) and step provided in key reading 4.3.2 you are requested to go to the computer lab
to create firmware Hex File, export it as extension format and then flash it to the STM32 microcontroller.

3: Present your work to the trainer and whole class.

4: Perform the task provided in application of learning 4.3

Key readings 4.3.2: Creating, Exporting and Flashing Hex File.

- **Creating a .hex File**

Creating a **.hex file** involves several steps that depend on the platform and tool you're using to develop firmware for embedded systems. Below is a general guide on how to create a **.hex file**, using common IDEs like **Arduino IDE** and **PlatformIO**.
Objective: Convert your firmware code into a .hex file format for programming microcontrollers.
Using Arduino IDE (for Arduino-based microcontrollers)
Step-by-step process:

1. **Write or Open the Sketch**:
   - Open the Arduino IDE and write your program in the sketch (e.g., .ino file), or load an existing sketch.
2. **Select the Board**:
   - Go to **Tools > Board** and select the correct board (e.g., Arduino Uno, Mega, etc.).
3. **Select the Port**:
   - Go to **Tools > Port** and select the COM port where your Arduino is connected.
4. **Compile the Sketch**:

- Click the **Verify** button (checkmark icon) to compile the code.

5. **Locate the HEX File**:
   - After compilation, the Arduino IDE generates a .hex file and saves it in a temporary directory.
   - To find the .hex file:
     - Enable verbose output: Go to **File > Preferences** and check the box for **Show verbose output during: compilation**.
     - Recompile the sketch. During the output, the temporary directory path containing the .hex file will be shown in the IDE console. Navigate to this folder to find the .hex file.
   - Ensure the project builds successfully, generating the .hex file along with other output files.
   - The .hex file is usually located in the Debug or Release folder of your project directory.

➢ **Exporting the .hex File**

Objective: Prepare the .hex file for deployment or flashing onto the microcontroller.

1. **Verify Compilation**:
   - Ensure the firmware has been successfully compiled and the .hex file is present in the output directory.
2. **Export the .hex File**:
   - STM32CubeIDE**:** The .hex file is generated in the project's output folder by default.
   - Keil uVision**:** Enable **Create HEX** File in the **Options for Target** settings.
   - MPLAB X IDE**:** The .hex file is automatically generated in the dist directory.

- **Flash the .hex File**:
  - Use appropriate programming tools or bootloaders to upload the .hex file to your microcontroller.

Key Points:

- Always ensure you select the correct target device and configure settings accurately.
- Verify the .hex file generation and location before flashing.

By following these steps, you can effectively create, configure, and export a .hex file for successful firmware deployment.

Exporting HEX Files as Firmware Images

The general process of exporting HEX Files as Firmware Images are:

1. **Identify the target device's firmware format:**
    - Different devices may have specific firmware image formats (e.g., .bin, .img, .fw).
    - Consult the device's documentation or manufacturer's website to determine the required format.
2. **Use a conversion tool:**
    - Many tools and programming environments can convert HEX files to other formats.
    - Some common options include:
        - **Custom scripts:** If you have programming experience, you can create scripts to convert HEX files to your desired format.
        - **Integrated development environments (IDEs):** Many IDEs, like Keil uVision or Arduino IDE, have built-in features for converting HEX files.
        - **Dedicated conversion tools:** There are specialized tools available for converting HEX files to various firmware image formats.
3. **Specify the target format:**
    - Configure the conversion tool to output the firmware image in the correct format.
    - This often involves specifying the target device or microcontroller model.
4. **Execute the conversion:**
    - Run the conversion tool on your HEX file.
    - The tool will generate the firmware image in the specified format.

➢ **Steps of flashing HEX file to microcontroller**

1. **Prepare the Required Tools:**

Microcontroller Board: Ensure the target microcontroller is available (e.g., Arduino, STM32, PIC).

HEX File: Ensure the firmware has been compiled into a HEX file.

USB Programmer or Debugger: A device like an In-System Programmer (ISP), JTAG, or USB-to-serial adapter (e.g., ST-Link, USBasp) for flashing.

Flashing Software: Install a tool to upload the HEX file. Common tools include:

avrdude (for AVR microcontrollers)

STM32CubeProgrammer (for STM32 microcontrollers)

MPLAB IPE (for PIC microcontrollers)

Driver Software: Ensure any necessary drivers for the programmer or the microcontroller are installed on your computer.

2. **Connect the Microcontroller:**

USB or ISP Connection: Connect the microcontroller to your computer using a USB cable or an ISP programmer.

Power Supply: Make sure the microcontroller is powered properly (either through USB or an external power source).

3. **Open Flashing Software:**

Launch the flashing tool (e.g., avrdude, STM32CubeProgrammer, MPLAB IPE).

Choose the correct **microcontroller model** and **port** for the flashing tool.

4. **Select the HEX File:**

In the flashing software, browse to and select the HEX file you want to flash to the microcontroller.

Ensure that the HEX file is the correct version and compatible with the microcontroller.

5. **Configure the Flashing Tool (if needed):**

Choose the correct **programming interface** (USB, serial, ISP, JTAG).

Set additional options like **fuses** or **memory settings** if your microcontroller requires these (applicable for some MCUs like AVR).

6. **Start the Flashing Process:**

Initiate the flashing process by pressing the **"Upload"**, **"Flash"**, or **"Program"** button in the software.

The tool will transfer the HEX file to the microcontroller's memory.

7. **Verify the Flashing Process:**

Most tools will display a progress bar during flashing and show a success message when the process is complete.

Some tools may also verify the flashed firmware by reading back the data from the microcontroller and comparing it with the HEX file.

Points to Remember

- **A.hex file,** also known as **Intel HEX format**, is a text-based file format used to represent binary data in a human-readable form. This format is commonly used to store and transfer firmware, microcontroller code, or other binary data to be programmed into electronic devices.
- **Steps of creating hex file:**
  1. Write or Open the Sketch
  2. Select the Board
  3. Select the Port
  4. Compile the Sketch
  5. Locate the HEX File
- **Steps of exporting using hex file:**
  1. Identify the target device's firmware format
  2. Use a conversion tool
  3. Specify the target format
  4. Execute the conversion
- **Steps of flashing HEX file to microcontroller**
  1. Prepare the Required Tools:
  2. Connect the Microcontroller:
  3. Open Flashing Software:
  4. Select the HEX File
  5. Configure the Flashing Tool (if needed)
  6. Start the Flashing Process
  7. Verify the Flashing Process

Application of learning 4.3.

You are working on an IoT project that uses a **temperature and humidity sensor** (e.g., DHT11) connected to an **STM32 microcontroller**. The microcontroller reads the sensor data and sends it to a cloud server over Wi-Fi.

Task: you are requested to create firmware Hex File, export it as extension format and then flash it to the STM32 microcontroller.

Indicative content 4.4: Documentation of the firmware

Duration: 3 hrs

Theoretical Activity 4.4.1: Firmware technical documentation and user manual.

Tasks:

1: Answer the following questions:

I. Describe technical documentation of firmware.

2: Provide the answer for the asked questions and write them on papers or flip chat.

3: Present the findings/answers to the whole class.

4: Ask questions where necessary.

5: For more clarification, read the key readings 4.4.1**.**

Key readings 4.4.1.: Firmware technical documentation and user manual

✓ **Technical Documentation of the Firmware:**

Firmware documentation refers to the creation of detailed, clear, and structured documents that describe how the firmware operates, including the design, structure, and functionality of the code. It helps developers and engineers understand, maintain, and improve the system.

Selection of Documentation Tool

Choosing the right tool for documenting firmware is crucial for maintaining clear and organized documentation. Consider:

- **Ease of Use:** Simple tools like Markdown editors (Typora, StackEdit) or advanced tools like Doxygen, Sphinx.
- **Integration:** The tool should integrate with code repositories like GitHub or GitLab.
- **Collaboration:** Support for team-based documentation.
- **Output Formats:** Ability to export to multiple formats (HTML, PDF, etc.)

Installation of Documentation Tool

- Dioxygen (popular for documenting firmware): Install using package managers like apt for Linux or brew for macOS.
- Linux: sudo apt install doxygen
- macOS: brew install doxygen
- **Sphinx**: Python-based, install via pip: pip install sphinx

- After installation, verify by running doxygen -v or sphinx-build --version.

Documentation Project Setup

- **Initial Configuration:** Run configuration commands like doxygen -g to generate a default configuration file.
- **Directory Structure:** Organize folders for documentation (e.g., docs/, src/ for code).
- **Style Setup:** Configure output style, file names, and directories in the configuration file.

Comments Fundamentals

- **Single-line Comments:** Use // or # for short, inline explanations.
- **Multi-line Comments:** For more detailed explanations, use block comments /* ... */.
- **Documentation Comments:** Tools like Doxygen or Sphinx parse special comments:
- Doxygen: /// or /** */ for generating detailed documentation.
- Sphinx: ReST or Markdown comments for documentation.

Documenting Enums and Structs

Definition:

- **Enum (Enumeration):** A data structure in programming that defines a set of named constants, improving code readability and reducing errors.
- **Struct (Structure):** A composite data type used to group related variables under a single name, often representing objects or complex data.

➢ **Best Practices for Documenting Enums:**

- **Define Purpose:** Clearly explain the role of the enum and why each constant is needed.
- **Provide Context:** Mention where the enum is used in the firmware, including specific modules or functions.

Example:

```c
/**
 * @enum LightMode
 * Enum representing the different modes of an LED light.
 */
typedef enum {
    OFF,        /**< LED is off */
    ON,         /**< LED is on */
    BLINKING    /**< LED is blinking */
} LightMode;
```

➢ **Best Practices for Documenting Structs:**

- **Explain Structure:** Define each member of the struct and its significance in the code.
- **Provide Usage Example:** Show how the struct is initialized or manipulated within the code.

  Example:

```c
/**
 * @struct DeviceConfig
 * Structure representing the configuration of a device.
 */
typedef struct {
    int device_id;      /**< Unique identifier for the device */
    char* name;         /**< Name of the device */
    bool is_active;     /**< Device status: active/inactive */
} DeviceConfig;
```

Documenting Functions

- **Purpose:** Functions are the building blocks of firmware, and documenting them is essential for code maintainability, debugging, and extending functionality.
- **Key Elements to Document:**

- **Function Purpose:** A brief description of what the function does.
- **Parameters:** A list of input parameters, their types, and a description of what each represents.
- **Return Value:** The type and meaning of the value the function returns, if any.

- **Side Effects:** Mention any changes the function makes to global variables or system state.

Example:

```
/**
 * @brief Initializes the device.
 *
 * This function sets up the device with the given configuration parameters.
 *
 * @param config A pointer to the DeviceConfig structure containing device settings.
 * @return int Returns 0 on success, -1 on failure.
 */
int device_init(DeviceConfig* config);
```

Keep function descriptions concise and focused on the primary action.
Clearly describe the expected range and behavior of input parameters.

Documenting Modules

A **module** in firmware refers to a self-contained unit of code that implements a specific functionality. For example, in a hardware system, you might have a **UART (Universal Asynchronous Receiver-Transmitter) module** responsible for serial communication, or a **Timer module** that manages system timers.

Documenting a module is critical because it helps developers understand what the module does, how to use it, and how it interacts with other parts of the firmware. This is especially important in complex systems where many modules work together.

➢ **Key Elements in Module Documentation:**

Module Overview:

- **Purpose:** This is a high-level description of what the module does and why it exists in the firmware.
- **Example:** For a **UART Module**, the overview might explain that it handles communication between the microcontroller and other devices using a serial protocol.

```
### UART Communication Module
This module handles communication over UART (Universal Asynchronous Receiver-Trans

**Key Features:**
- Initialization and configuration of UART.
- Sending and receiving data.
- Error detection and handling.

**Dependencies:**
- Hardware UART interface.
- Standard I/O libraries.
```

➢ **Key Features and Functions:**

- **List of Features:** Summarize the core functionalities provided by the module.
- **Functions Overview:** Identify the key functions that the module provides for other parts of the firmware.
- For instance, a **UART Module** might include functions like UART_Init(), UART_Send(), and UART_Receive().

Creation of a Reusable Template

A **reusable template** in firmware documentation is a predefined structure or format that you use repeatedly when documenting different parts of your firmware, such as modules, functions, data structures (enums and structs), etc. This ensures **consistency**, **completeness**, and **ease of use** across your documentation.

Why Use a Template?

- **Consistency:** All parts of the firmware will be documented in the same format, making it easier for anyone (developers, testers, maintainers) to read and understand the documentation.
- **Efficiency:** Instead of starting from scratch every time you need to document something, you can follow the template. This saves time and ensures you don't forget important details.
- **Comprehensiveness:** The template acts as a checklist, ensuring that all necessary information is included in the documentation.

What Goes into a Documentation Template?

➢ **Module/Function Overview:**

- **Purpose:** Give a brief description of what the module or function does.

- **Version:** Specify the current version of the firmware/module.
- **Author:** Indicate who wrote or last updated the code.
- **Date:** Record when the documentation was last updated or created.

```
## UART Module
- **Description:** This module handles UART communication between the firmware and ext
- **Version:** 1.0.2
- **Author:** John Doe
- **Date:** 18th September 2024
```

✓ **Firmware User Manual in Documentation**

A **Firmware User Manual** is a comprehensive guide designed for users who interact with your firmware. It provides detailed instructions on how to install, configure, and use the firmware in real-world scenarios.

1. Title Page

- **Project Name, Version, Date, Author/Organization.**
- **Example:** *"Firmware User Manual for XYZ Device(smart device firmware), Version 1.0.0, September 2024."*

```
Firmware User Manual
Project: Smart Device Firmware
Version: 1.0.0
Date: September 18, 2024
Author: John Doe
Organization: XYZ Technologies
```

2. Table of Contents

- Clear navigation guide to different sections of the manual.

3. Introduction

- **Overview:** Describe the firmware's purpose and functionalities.
- **Target Audience:** Define who the manual is for (e.g., technicians, users).
- **System Requirements:** Hardware/software requirements needed to run the firmware.

```
The Smart Device Firmware is designed to control home automation devices, such as ligh

**System Requirements:**
- Device: XYZ Smart Hub
- Minimum RAM: 512 MB
- Power Source: 5V DC
```

➢ **Installation Instructions**

This section provides step-by-step guidance on how to install the firmware onto the device.

- **Prerequisites:** List any tools or files needed before installation (e.g., drivers, cables, software).
- **Firmware Update Process**
  - How to connect the device.
  - Steps to upload or flash the firmware onto the hardware.
- **Post-Installation Steps:** Verifying that the installation was successful and that the device is operational.

Example:

```
**Prerequisites:**
- USB to Serial Adapter
- XYZ Smart Hub Device
- Firmware File: SmartHubFirmware_v1.0.0.hex
- XYZ Firmware Updater Software

**Installation Steps:**
1. Connect the USB to Serial Adapter to your computer.
2. Open the XYZ Firmware Updater software.
3. Select the firmware file (SmartHubFirmware_v1.0.0.hex).
4. Click "Upload" to flash the firmware to the Smart Hub.
5. Once the upload is complete, the device will reboot automatically.
```

- **Configuration Guide**

After installation, the firmware might need to be configured to fit the specific use case or hardware. This section guides the user through setting up the firmware.

- **Initial Setup:** Describe how to perform the initial configuration.
- **Configurable Parameters:** Provide details on key configuration settings such as communication protocols, device IDs, network settings, etc.

- **Using Configuration Tools:** If your firmware comes with software tools, describe how to use them for configuration.

```
**Initial Setup:**
After the firmware has been installed, connect to the device using a USB connection. U

**Configurable Parameters:**
- **Device ID:** Set a unique ID for each device on the network.
- **Baud Rate:** Configure the UART baud rate for communication (default: 9600).
- **Wi-Fi Settings:** Connect the device to your local Wi-Fi network by entering the S

**Using the Configuration Tool:**
1. Open the XYZ Configuration Tool.
2. Navigate to the "Network" tab to set up Wi-Fi credentials.
3. Apply the changes and restart the device.
```

Usage Instructions:

- Basic Operations**:** Guide on how to start/stop/reset the firmware.
- Advanced Operations: Explain any advanced features or custom commands.
- **Command List (if applicable):** Include key commands for interacting with the firmware.

```
**Basic Operations:**
- To start the Smart Device, press the "Power" button. The LED will blink, indicating
- To reset the device, hold the "Reset" button for 5 seconds until the LED flashes rap

**Advanced Operations:**
- Use the command `device_status` to check the current status of the device through a
- Use `set_mode <mode>` to switch between different operating modes (e.g., `set_mode A

**Command List:**
- `device_status`: Displays current status.
- `set_mode <mode>`: Changes the device's mode.
- `get_config`: Retrieves the current configuration settings.
```

Troubleshooting

- Common Issues and Fixes: List common problems and provide solutions.
- Error Codes: Document any error codes the firmware might generate and how to resolve them.
- Reset Procedures: How to perform a factory reset or restore the device.

Firmware Updates:

- Checking for Updates: How users can check for and download firmware updates.
- Updating Process: Steps to update the firmware.
- Backup & Restore: Instructions on how to back up configurations before updating.

Safety Precautions

- Guidelines for safely handling and operating the firmware and hardware, if necessary.

Glossary (Optional)

- Definitions for technical terms and acronyms used in the manual.

Appendix (Optional)

- Additional information like schematics, detailed logs, or technical diagrams.

This handout provides a streamlined guide for creating and using a **Firmware User Manual**, focusing on practical use, configuration, troubleshooting, and maintenance of firmware systems.

Practical Activity 4.4.2: Installing Documentation Tool.

Task:

1: Read key reading 4.4.2 and ask clarification where necessary

2: Referring to the previous theoretical activities (4.4.1) and step provided in key reading 4.4.2 you are requested to go to the computer lab and perform the given task bellow:

Your task is to install **Doxygen** on your Windows machine, **verify** its installation and configure it to generate documentation.

3: Present your work to the trainer and whole class.

4: Perform the task provided in application of learning 4.4

Key readings 4.4.2: Installing documentation tool on windows "doxygen"steps

Step 1: Download Doxygen

1. **Go to the Doxygen website**: http://www.doxygen.nl/download.html.
2. **Download the Windows Installer**: Look for the .exe installer under the **Windows binaries** section.

Step 2: Install Doxygen

1. **Run the Installer**:
   - Double-click the downloaded .exe file to launch the installation wizard.
2. **Follow the installation steps**:
   - Click **Next**.
   - Accept the **license agreement**.
   - Choose the installation folder (default location is recommended).
   - Click **Install** to complete the installation.

Step 3: Verify Doxygen Installation

1. **Open Command Prompt**:
   - Press Win + R, type cmd, and press **Enter** to open the command prompt.
2. **Check if Doxygen is installed**:
   - Type doxygen -v in the command prompt and press **Enter**.
   - You should see the Doxygen version number, which confirms successful installation.

Step 4: Install Graphviz (for generating diagrams)

1. **Go to the Graphviz download page**: https://graphviz.gitlab.io/download/.
2. **Download the Windows Installer**:
   - Select the appropriate installer based on your system (e.g., graphviz-setup.exe).
3. **Install Graphviz**:
   - Run the .exe file, and follow the installation wizard.

Step 5: Verify Graphviz Installation

1. **Open Command Prompt** again.
2. **Type**:

```
dot -version
```

You should see the installed Graphviz version number, confirming it's installed

correctly.

Step 6: Setting Up Doxygen with Graphviz

1. **Open Doxygen** (can be done by searching it in the start menu).
2. **Configure Project**:
    - Open the Doxygen GUI, and create a new configuration file by clicking **File > New**.
    - In the configuration window, enable **EXTRACT_ALL** to document all code elements.
3. **Enable Diagrams**:
    - Scroll down to the **Dot** section (used for diagrams).
    - Set **HAVE_DOT** to YES.
    - Set the path to the Graphviz dot.exe file by clicking **Browse** next to the Dot Path field (e.g., C:\Program Files\Graphviz\bin\dot.exe).

Step 7: Generate Documentation

1. **Select Your Firmware Codebase**:
    - Point Doxygen to the folder where your firmware code resides.
    - Add comments in your code using /// or /** */ syntax.
2. **Run Doxygen**:
    - Click **Run > Run doxygen** to generate HTML and/or PDF documentation from your code.
3. **Open the Documentation**:
    - After running, open the index.html file in the generated html folder to view your documentation in a browser.

Next Steps

You now have Doxygen and Graphviz installed on Windows and configured to generate technical documentation for your firmware. You can now:

- Add comments to your firmware code to document functions, modules, structs, etc.
- Use Doxygen to regenerate the documentation whenever your firmware is updated.

Practical Activity 4.4.3: Creating reusable template

Task:

1: Read key reading 4.4.2 and ask clarification where necessary

2: Referring to the previous theoretical activities (4.4.1) and step provided in key reading 4.3.2 you are requested to go to the computer lab and perform the given task bellow:

You are requested to create a reusable firmware documentation template.

3: Present your work to the trainer and whole class.

4: Perform the task provided in application of learning 4.4

Key readings 4.4.3: Creating reusable template

Creating a **reusable template** for firmware documentation ensures consistency and efficiency when documenting multiple projects or future updates. Here's a practical guide on how to set up a **reusable Doxygen template** for firmware documentation in Windows.

Step-by-Step Guide: Creating a Reusable Template in Firmware Documentation (Windows)

Step 1: Configure a Standard Doxygen Project

1. **Open Doxygen GUI** on your Windows machine.
2. **Create a New Doxygen Configuration**:
    - Click on **File > New** to create a new configuration file.
    - Choose a folder where you want to save the configuration (you can name this as "Firmware_Template").
    - This will generate a default configuration file, Doxyfile, that you can customize and reuse.

Step 2: Set Up Core Documentation Sections

In the Doxygen GUI, configure the following **core sections** for your firmware project, which you can reuse:

1. **Project Name and Version**:
    - In the **Project** tab, set a generic project name, version, and brief description.

Example:

- **PROJECT_NAME** = Firmware Documentation Template
- **PROJECT_VERSION** = 1.0
- **PROJECT_BRIEF** = Template for documenting embedded firmware projects

2. **Input Files**:

- Set **INPUT** paths where your firmware source code resides (e.g., .c, .h files). This will pull in files for future projects.
  - Example

```
plaintext
INPUT = src/ include/
```

  - This can be edited for future projects depending on the file structure.

3. **Output Directory**:
   - Specify where the generated documentation (HTML, PDF, etc.) should be stored.
     - Example:
       - **OUTPUT_DIRECTORY** = docs/
4. **Extract All Code Comments**:
   - Enable the extraction of all elements from your source code by setting

```
EXTRACT_ALL = YES
```

5. **Enable Diagram Generation (Optional)**:
   - To generate call graphs and class diagrams, ensure **Graphviz** is enabled:

```
plaintext
HAVE_DOT = YES
```

Set the **DOT_PATH** to the location of dot.exe (e.g., C:\Program Files\Graphviz\bin).

Step 3: Customize Documentation for Firmware-Specific Sections

To create a **template** that's ready to reuse across various firmware projects, set up sections that will remain consistent:

1. **Modules and Files Documentation**:
   - Organize your firmware into modules and document each module separately.
     - Example:

```
@defgroup Sensor_Module Sensor Module
@brief Module to control temperature sensors
```

2. **Documenting Functions**:
   - Create a standard format for function documentation. Use this template:

```c
/**
 * @brief   Reads temperature from sensor
 * @param   sensor_id: ID of the temperature sensor
 * @retval  Temperature value in Celsius
 */
int Read_Temperature(int sensor_id);
```

3. **Documenting Data Structures (Enums and Structs)**:
   - Standardize comments for enums and structs:

```c
/**
 * @enum Sensor_Status
 * @brief Enumeration of sensor status codes
 */
typedef enum {
    SENSOR_OK,
    SENSOR_ERROR,
    SENSOR_DISCONNECTED
} Sensor_Status;
```

```c
/**
 * @struct SensorData
 * @brief Structure to store sensor readings
 */
typedef struct {
    int sensor_id;
    float temperature;
} SensorData;
```

General Comment Style:

- Document inline comments for constants, variables, and functions with standard tags like @brief, @param, @retval, @note, etc.

Step 4: Save and Reuse the Template

Once you've set up the initial configuration, you can **save this Doxygen configuration file (Doxyfile) as a reusable template**:

1. **Save the Configuration**:

- Click on **File > Save** and store the configuration file (Doxyfile) in a dedicated folder like C:\DoxygenTemplates\Firmware_Template\.

2. **Create a Batch Script for Reusability**:
   - To make reuse even easier, you can create a batch script (generate_docs.bat) that runs Doxygen with this template on any firmware project. Create a .bat file with the following content:

```
@echo off
doxygen C:\DoxygenTemplates\Firmware_Template\Doxyfile
pause
```

Place this script in any new project folder to generate documentation based on the saved template.

Step 5: Modify and Apply the Template for New Firmware Projects

For each new firmware project:

1. **Copy the Template Folder**:
   - Copy the entire Firmware_Template folder into your new project directory.
2. **Edit the Doxyfile**:
   - Open the Doxyfile in a text editor or in the Doxygen GUI, and update specific details like:
     - **PROJECT_NAME**
     - **PROJECT_VERSION**
     - **INPUT** paths (point to your new project's source files).
3. **Generate Documentation**:
   - Run Doxygen by double-clicking your batch script (generate_docs.bat), or open Doxygen GUI and click **Run**.

Step 6: Enhancing the Template for Multiple Output Formats

To enhance your template and make it more flexible:

1. **HTML and PDF Output**:
   - Enable both HTML and LaTeX (for PDF) output in the configuration file by setting

```
GENERATE_HTML = YES
GENERATE_LATEX = YES
```

Create a Standard README File:

- Add a README.md file that provides basic instructions on how to use the template, edit the Doxyfile, and generate documentation for the project.

Example Reusable Template Structure

```makefile
/DoxygenTemplates/Firmware_Template/
    |-- Doxyfile           # Doxygen configuration template
    |-- generate_docs.bat  # Batch file to automate documentation generation
    |-- README.md          # Instructions for using the template
    |-- src/               # Placeholder for source code (empty)
    |-- include/           # Placeholder for header files (empty)
```

N.B This setup gives you a **reusable template** for firmware documentation using Doxygen on Windows.

Points to Remember

Firmware technical documentation include for Selection of Documentation Tool, Installation of Documentation Tool, Documentation Project Setup, Documentation Project Setup, Comments Fundamentals**,** Documenting Enums and Structs, Documenting Modules and Creation of a Reusable Template and Documenting Functions**.**

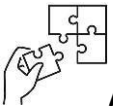
Step-by-Step Installation of Documentation Tool on Windows: Doxygen

Step 1. Download Doxygen
Step 2: Install Doxygen
Step 3: Verify Doxygen Installation
Step 4: Install Graphviz (for generating diagrams)
Step 5: Verify Graphviz Installation
Step 6: Setting Up Doxygen with Graphviz

Step 7: Generate Documentation

Step of creating reusable template:

1. Configure a Standard Doxygen Project
2. Set Up Core Documentation Sections
3. Customize Documentation for Firmware-Specific Sections
4. Save and Reuse the Template
5. Modify and Apply the Template for New Firmware Projects
6. Enhancing the Template for Multiple Output Formats



Application of learning 4.4.

You are working on a project for ABCD Company to develop embedded software for a robotic system that integrates a temperature and humidity sensor with an STM32 microcontroller. To handle the increasing complexity of codebase, you will implement automatic documentation using Doxygen and Graphviz. You are requested to create a reusable firmware documentation template to streamline future projects, ensuring that the code is easy to understand, maintain, and well-documented for other developers.

Learning outcome 4 end assessment

Theoretical assessment

1. **What does it mean to "flash" a .hex file to a microcontroller?**
    a. To delete the existing firmware
    b. To write new firmware to the microcontroller's memory
    c. To power cycle the microcontroller
    d. To reset the microcontroller
2. **What type of file is a .hex file?**
    a. Text file
    b. Binary file
    c. Executable file
    d. Image file
3. **What is the primary purpose of flashing a .hex file?**
    A) To update the firmware on a microcontroller
    B) To format the microcontroller's memory
    C) To compile source code
    D) To create a backup of the firmware
4. **What hardware is typically needed to flash a .hex file to a microcontroller?**
    A) A printer
    B) A programmer or debugger
    C) A display screen
    D) A keyboard
5. **What software tools can be used for flashing a .hex file?**
    A) Text editor
    B) Flashing software or IDE
    C) Graphic design software
    D) Web browser
6. What is a .hex file?
7. What are the two main types of deployment environments for firmware?
8. What are some common techniques used for testing firmware?
9. Describe the tools or software typically used for flashing a .hex file to a microcontroller?
10. What is the purpose of a .hex file in firmware export?

Practical assessment

ABCD Company has just released new **firmware** for a series of **smart home devices**. The engineering team has provided you with the technical specifications, firmware change logs, and testing results. You are requested deploy designed firmware and to create user-friendly **firmware documentation manual**. You will be using a documentation tool like **Doxygen** to ensure that the information is well-organized, consistent, and easy to update, helping users understand the new features, installation procedures, and troubleshooting tips for the firmware. Your goal is to empower users to effectively utilize the firmware in their smart home systems.

Reference

Doe, J. (2020). *The Firmware Handbook.* (J. Smith, Ed.) Tech Press.

Douglass, B. P. (2016). *Embedded Systems: Design and Applications.* Amsterdam: Elsevier.

Himpe, V. (2015). *Embedded Firmware Solutions: Development Best Practices for the Internet of Things.* New York: Apress.

Ibrahim, D. (2014). *Designing Embedded Systems with 32-Bit PIC Microcontrollers and MikroC.* Oxford: Newnes.

Meer, J. A. (2018). *Designing Embedded Systems with Arduino.* Berlin: Springer.

Oshana, R. (2019). *Software Engineering for Embedded Systems.* CRC Press.

Peter Barry, P. C. (2012). *Real-Time Embedded Systems: Design Principles.* Hoboken: Wiley.

Simon, D. (2021). *Firmware Development for Embedded Systems.* Springer.

Valvano, J. (2018). *Embedded Systems: Real-Time Operating Systems.* Wiley.

Wilmshurst, T. (2010). *Designing Embedded Systems with PIC Microcontrollers.* Oxford: Newnes.

October, 2024