Processor Cores: What Are They?

A **processor core** is the **brain** of a CPU (Central Processing Unit) or **microcontroller**. It is responsible for **executing instructions** and performing calculations

ARM Cortex cores are a family of processor cores developed by **ARM Holdings**. They are widely used in microcontrollers (MCUs) and processors due to their **efficiency, scalability, and performance**.

Types of ARM Cortex Cores

ARM Cortex cores are divided into three main categories based on performance and application needs:

1. **Cortex-A Series** (Application Processors)
    - High-performance cores used in smartphones, tablets, and embedded Linux systems.
    - Examples: Cortex-A53, Cortex-A72, Cortex-A78
2. **Cortex-R Series** (Real-time Processors)
    - Designed for real-time, safety-critical applications like automotive and industrial control.
    - Examples: Cortex-R4, Cortex-R5
3. **Cortex-M Series** (Microcontroller Processors)
    - Optimized for **low power** and **embedded applications** like IoT, robotics, and industrial automation.
    - This is the most relevant category for **STM32 microcontrollers**.

Comprehensive Guide to STM Microcontrollers (STM32)

STM microcontrollers, developed by STMicroelectronics, are widely used in embedded systems, robotics, industrial automation, IoT, and real-time computing applications. They are based on ARM Cortex-M architectures and provide a balance of performance, power efficiency, and peripheral integration.

This guide will cover STM32 architecture, core series, peripherals, memory organization, real-time processing capabilities, development tools, and integration with external components such as sensors and motors.

Block Diagram (STM32F411CE)

The microcontroller consists of several **hardware blocks**:

1. **ARM Cortex-M4 CPU** – Executes instructions and processes data.
2. **Flash Memory** – Stores program code.

3. **SRAM** – Stores runtime data.
4. **Timers** – Generates PWM signals and manages timing events.
5. **Communication Peripherals** – Handles **UART, SPI, I2C, CAN, USB, I2S**.
6. **ADC/DAC** – Converts **analog signals to digital** and vice versa.
7. **Clock System** – Provides different clock sources (internal/external).

1. STM32 Microcontroller Architecture

1.1 Overview of Microcontroller Architecture

A microcontroller is a compact integrated circuit designed to execute specific embedded applications. Unlike general-purpose processors, microcontrollers include built-in memory, processing units, and peripherals within a single chip, enabling them to operate efficiently in real-time control systems.

STM32 microcontrollers follow the Harvard architecture, where instruction and data buses are separate, allowing simultaneous fetch and execution of instructions. This improves execution speed and power efficiency, making them suitable for embedded applications.

1.2 Core Components of STM32 Microcontrollers

Each STM32 microcontroller consists of several fundamental components:

Component	Function
ARM Cortex-M Core	Executes instructions, handles task scheduling, and processes
Flash Memory	Non-volatile storage for firmware and application code.
SRAM (Static RAM)	Temporary storage for variables and real-time computations.
GPIO (General-Purpose I/O)	Interfaces with external components such as sensors, actuators,
Clock System (RCC)	Manages internal and external clock sources to optimize power consumption and processing speed.
Timers & PWM Controllers	Control time-based functions, motor speed regulation, and signal
ADC (Analog-to-Digital	Converts analog sensor signals into digital data for processing.
DAC (Digital-to-Analog	Converts digital signals into analog output, useful for audio
Communication Interfaces	Enable data exchange via UART, SPI, I2C, CAN, USB, and
NVIC (Nested Vectored Interrupt Controller)	Handles interrupts and ensures real-time response.
Power Management System	Provides low-power modes for energy-efficient operation in

Each of these components plays a crucial role in embedded system design, enabling STM32 microcontrollers to handle a wide range of real-time applications.

2. STM32 Series and Core Variants

STM32 microcontrollers are divided into multiple families, each optimized for different applications. The main distinguishing factor is the ARM Cortex-M core used in each series.

2.1 ARM Cortex-M Core Variants

Core	Max Clock Speed	Key Features	Example STM32 Series
Cortex-M0/ M0+	~48 MHz	Energy-efficient, basic computing	STM32F0, STM32L0
Cortex-M3	~72 MHz	Balanced performance and power	STM32F1,
Cortex-M4	~180 MHz	Digital Signal Processing (DSP), Floating Point	STM32F3,
Cortex-M7	~480 MHz	Advanced DSP, dual-issue instruction pipeline, high performance	STM32F7, STM32H7
Cortex-M33	~160 MHz	Enhanced security, TrustZone, low-power AI	STM32U5,

Each of these cores provides different levels of computing capability, allowing developers to select the appropriate microcontroller based on application requirements.

2.2 STM32 Series and Applications

STM32 microcontrollers are categorized into different series based on their performance and target applications:

- **STM32F0 Series** (Cortex-M0): Entry-level microcontrollers designed for cost-sensitive applications such as home automation and simple IoT devices.
- **STM32F1 Series** (Cortex-M3): General-purpose microcontrollers widely used in industrial control and embedded applications.
- **STM32F3 Series** (Cortex-M4): Features additional DSP capabilities, making them suitable for motor control and digital signal processing.
- **STM32F4 Series** (Cortex-M4): High-performance microcontrollers with enhanced floating-point capabilities, used in robotics, medical devices, and audio processing.
- **STM32F7 Series** (Cortex-M7): Advanced microcontrollers designed for AI-driven applications, real-time image processing, and edge computing.
- **STM32H7 Series** (Cortex-M7): High-end microcontrollers with a dual-core architecture, optimized for industrial automation and automotive applications.

Each series provides a range of features and configurations, allowing developers to select the appropriate model based on power consumption, performance, and real-time constraints.

3. Memory Architecture and Bus System

STM32 microcontrollers incorporate multiple memory regions, including:

- **Flash Memory** (for firmware storage)
- **SRAM** (for runtime data storage)

- **EEPROM** (for non-volatile data storage)

The internal memory system is managed through the **AHB (Advanced High-Performance Bus)** and **APB (Advanced Peripheral Bus)**, ensuring efficient data transfer between the processor core, memory, and peripherals.

3.1 Flash Memory Organization

STM32 microcontrollers use NOR Flash memory, which allows in-system programming. The flash memory is divided into sectors, enabling partial erasure and updates without affecting the entire program.

3.2 SRAM Organization

STM32 devices feature multiple SRAM regions, including main system RAM and peripheral-specific RAM blocks. High-performance series, such as STM32H7, utilize **Tightly Coupled Memory (TCM)** to minimize latency in critical operations.

4. Peripheral Integration and Communication Interfaces

STM32 microcontrollers include a wide range of peripheral interfaces for hardware integration:

- **UART (Universal Asynchronous Receiver-Transmitter):** Used for serial communication with sensors, GPS modules, and Bluetooth modules.
- **SPI (Serial Peripheral Interface):** Provides high-speed data transfer for displays, memory chips, and sensors.
- **I2C (Inter-Integrated Circuit):** Enables communication with multiple low-speed devices such as EEPROMs and environmental sensors.
- **CAN (Controller Area Network):** Commonly used in automotive and industrial control systems for robust data exchange.
- **USB (Universal Serial Bus):** Supports USB host and device functionalities, useful for interfacing with external storage and peripherals.

The availability of these communication protocols allows STM32 microcontrollers to act as central processing units in complex embedded systems.

5. Real-Time Processing and RTOS Integration

. What is Task Scheduling?

Task scheduling is the process of managing multiple tasks in an STM32 microcontroller to ensure efficient execution and real-time performance. It ensures critical tasks run on time while optimizing CPU usage.

2. Types of Task Scheduling

A. Bare-Metal Scheduling (Without RTOS)

1. **Superloop (Polling-Based)** – Tasks run sequentially inside an infinite loop. Simple but inefficient.
2. **Interrupt-Driven Scheduling** – High-priority tasks execute when an event occurs, improving efficiency.
3. **Timer-Based Scheduling** – Uses hardware timers to trigger tasks at set intervals.

Pros: Simple, no extra software required.
Cons: Limited multitasking, inefficient CPU usage.

B. RTOS-Based Scheduling (With FreeRTOS, ChibiOS, etc.)

1. **Preemptive Scheduling** – Higher-priority tasks interrupt lower-priority ones, ensuring real-time execution.
2. **Round Robin Scheduling** – Each task gets equal CPU time, useful when all tasks are equally important.

Pros: Efficient multitasking, priority-based execution, better power management.
Cons: Requires an RTOS, adds complexity.

3. Applications in STM32-Based Systems

- **Autonomous Robots:** Sensor reading (high priority), motor control (medium), telemetry (low).
- **Industrial Automation:** Machine monitoring (high), actuator control (medium), data logging (low).
- **IoT Systems:** Network communication, power management.

4. Conclusion

- **Bare-metal scheduling** is best for simple tasks but lacks real-time guarantees.
- **RTOS scheduling** is essential for real-time applications, providing better efficiency and multitasking.
- STM32 microcontrollers support **FreeRTOS** and **timer-based scheduling** for optimized performance.

STM32 microcontrollers support real-time operating systems (RTOS) for efficient task scheduling.

- **FreeRTOS**: A widely used open-source RTOS that allows multi-threaded execution, event handling, and real-time constraints management.
- **Zephyr OS**: An open-source RTOS optimized for IoT applications, supporting secure networking and device management.

The **NVIC (Nested Vectored Interrupt Controller)** is responsible for handling real-time task execution by prioritizing interrupts and ensuring minimal latency.

6. Low-Power Modes and Energy Efficiency

STM32 microcontrollers offer multiple power-saving modes:

- **Run Mode**: Full-speed operation with all peripherals enabled.
- **Sleep Mode**: CPU is stopped, but peripherals remain active.

- **Stop Mode**: Low-power mode with RAM retention.
- **Standby Mode**: Ultra-low-power mode with wake-up functionality.

A. Run Mode (Full-Speed Operation)

- The microcontroller is fully active, with the **CPU and all peripherals running at full clock speed**.
- This mode consumes the most power but provides maximum performance.
- Used in applications requiring continuous processing, such as **real-time control systems** and **high-speed data acquisition**.

B. Sleep Mode (CPU Stopped, Peripherals Active)

- The **CPU is halted**, but peripherals (such as UART, SPI, and I2C) remain active.
- Power consumption is lower than in Run Mode since the processor is idle.
- **Wake-up sources**: Any peripheral interrupt (e.g., UART receiving data, a timer event, or an external interrupt).

Use Cases:

- Keeping communication interfaces (e.g., Bluetooth, Wi-Fi, CAN bus) active while the CPU sleeps.
- Reducing power usage in embedded systems while waiting for external triggers.

C. Stop Mode (Low Power with RAM Retention)

- **The CPU and most peripherals are turned off, but RAM contents remain intact**.
- The system can **quickly resume execution** from where it left off.
- **Clocks are disabled**, except for low-power sources like the **Low-Speed Internal (LSI) clock** or **Real-Time Clock (RTC)**.
- **Wake-up sources**:
    - External interrupts (e.g., GPIO pin change).
    - RTC alarm or timer.
    - Low-power timers (LPTIM).

Power Consumption:

- Much lower than Sleep Mode but slightly higher than Standby Mode.

Use Cases:

- **IoT devices**: The system can sleep while periodically waking to send sensor data.
- **Robotic control systems**: The controller can power down while waiting for an external trigger.
- **Energy-efficient embedded applications** where rapid wake-up is needed.

D. Standby Mode (Ultra-Low-Power Mode with Wake-Up Functionality)

- The **CPU, clocks, and peripherals are completely powered down**.
- Only a **tiny portion of backup RAM remains active** (if enabled).
- Wake-up resets the CPU, meaning execution starts from the **bootloader** unless backup registers store necessary information.
- **Wake-up sources**:
    - External events (button press, motion sensor).

    - RTC alarm (scheduled wake-up).
    - Wake-up pins (special GPIOs designed for ultra-low-power wake-up).

Power Consumption:

- **Extremely low**, making this mode ideal for long-term power conservation.

Use Cases:

- **Battery-powered sensors** that only wake up at scheduled intervals.
- **Smart meters** that wake up once per day for data transmission.
- **Sleep-mode operation in portable electronics** to preserve power.

TM32 microcontrollers **do not remain in low-power mode permanently**—they wake up in response to specific events.

Common Wake-Up Sources

1. **External GPIO Interrupts (EXTI)**
    - A signal from an external source (e.g., button press, sensor activation) triggers an interrupt that wakes up the microcontroller.
    - Used in event-driven applications, such as **motion-triggered cameras**.
2. **Real-Time Clock (RTC) Alarm**
    - The internal RTC triggers an alarm at a scheduled time, waking up the system.
    - Used for **periodic data logging in IoT applications**.
3. **Low-Power Timer (LPTIM)**
    - A specialized timer that runs even in low-power modes, allowing wake-up at precise intervals.
    - Useful for **scheduled wake-ups without an external clock source**.
4. **Watchdog Timer (IWDG/WWDG)**
    - If the system becomes unresponsive, the watchdog resets the MCU, effectively "waking it up" into a fresh state.
    - Used in safety-critical applications (e.g., **automotive systems**).

Wake-Up Sequence in STM32

1. The microcontroller enters **Stop Mode** or **Standby Mode**.
2. A **wake-up source (GPIO, RTC, EXTI, etc.)** triggers an interrupt.
3. The system resumes execution:
    - In **Stop Mode**, execution continues where it left off (RAM retained).
    - In **Standby Mode**, execution starts from the beginning (RAM is lost, except for backup registers).

These power management features make STM32 microcontrollers ideal for battery-powered applications, such as wearable devices and remote IoT sensors.

7. Development Tools and Programming Frameworks

Developing applications for STM32 microcontrollers requires a combination of hardware and software tools:

- **STM32CubeIDE**: The official integrated development environment for STM32.
- **PlatformIO**: A flexible development framework for embedded systems.
- **Keil uVision**: A professional-grade IDE with advanced debugging capabilities.
- **STM32CubeMX**: A graphical tool for peripheral configuration and code generation.

Using these tools, developers can implement real-time applications, optimize memory usage, and interface STM32 microcontrollers with external components.

Functional Overview of STM32F411CE Microcontroller

The **STM32F411CE** is a **high-performance** microcontroller from STMicroelectronics, built around the **ARM Cortex-M4** processor with an **FPU (Floating Point Unit)**. It is optimized for **low power consumption, real-time performance, and efficient memory management**. Below is a detailed breakdown of its key functional components.

2. Adaptive Real-Time Memory Accelerator (ART Accelerator™)

The **ART Accelerator** is a hardware feature designed to speed up the execution of programs stored in **Flash memory**.

How it Works:

- The **Flash memory** is usually **slower than the CPU**, which can create bottlenecks.
- The **ART Accelerator prefetches** instructions and stores frequently used **branches** (instructions) in a cache.
- This allows **zero wait-state execution**, meaning the **CPU runs at full speed** without waiting for data from Flash.

Performance Boost: STM32F411CE achieves up to **105 DMIPS (Dhrystone MIPS)**, making it much faster than traditional Flash memory-based microcontrollers.

3. Batch Acquisition Mode (BAM)

Batch Acquisition Mode (BAM) is a power-saving feature that allows the microcontroller to **process sensor data with minimal power consumption**.

How BAM Works:

- When the system is in **low-power mode**, **data acquisition** continues using **DMA (Direct Memory Access)**.
- This **bypasses the CPU** and allows **data from sensors (I2C, SPI, UART, ADC, etc.) to be stored in SRAM**.
- The CPU wakes up only when **data processing is required**, reducing **overall power usage**.

Example: In a **wearable health monitor**, BAM allows continuous **heart rate data collection** while keeping power consumption low.

5. Embedded Flash Memory (512 KB)

The **STM32F411CE** has **512 KB of internal Flash memory**, which is used for **storing programs and data**.

Key Features:

✔ **Supports Power Saving:** Flash can be turned off in **low-power modes** (e.g., DeepSleep) to reduce power consumption.
✔ **One-Time Programmable (OTP) Memory:** Includes **16 blocks of 32 bytes** for storing **permanent data** (e.g., device ID, security keys).

Example: Used in **IoT devices** to store **firmware updates and encryption keys** securely.

6. Cyclic Redundancy Check (CRC) Unit

The **CRC unit** is used to **detect errors in data storage and transmission**.

How It Works:

- The system generates a **CRC code (checksum)** for **stored or transmitted data**.
- When the data is read or received, the **checksum is recalculated** and compared with the original CRC code.
- If the values **do not match**, it means **data corruption has occurred**.

Common Use: Verifying **firmware integrity** in **critical systems (medical devices, automotive applications)**.

7. Embedded SRAM (128 KB)

The **SRAM (Static RAM)** is used for **storing runtime variables and fast-access data**.

Key Features:

✔ **128 KB of SRAM** for storing real-time sensor data and program variables.
✔ **Zero-Wait-State Access:** Can be accessed **at full CPU speed** with **no delay**.

Example: Stores **real-time sensor data** in **autonomous drones**, allowing fast decision-making.

8. Multi-AHB Bus Matrix

The **Advanced High-performance Bus (AHB) matrix** allows **multiple components** to access memory and peripherals **simultaneously without slowdowns**.

Key Features:

✔ **Dedicated Paths:** The CPU, DMA, and peripherals can access memory at the same time.
✔ **Efficient Communication:** Ensures **fast data transfer** between **Flash, SRAM, and peripherals**.

Example: Allows **camera image processing and motor control** to run simultaneously in a **robot** without delays.

9. Direct Memory Access (DMA) Controller

The **DMA Controller** allows **data transfer between memory and peripherals without using the CPU**.

Benefits:

✔ **Reduces CPU Load:** The CPU can perform other tasks while DMA handles data transfer.
✔ **Faster Data Processing:** Supports **memory-to-memory, peripheral-to-memory, and memory-to-peripheral transfers**.
✔ **Circular Buffer Mode:** Allows **continuous data collection** from **sensors** without software intervention.

Example: Transfers live sensor data (gyroscopes, accelerometers) to memory in a self-balancing robot without CPU involvement.

11. External Interrupt/Event Controller (EXTI)

The **EXTI** allows external signals (from buttons, sensors) to trigger **interrupts**.

✔ **Supports Up to 81 GPIOs** for interrupts
✔ **Can Detect Rising/Falling Edge Pulses** (e.g., button presses, motion sensors)

Example: Used in **touch-based interfaces** or **proximity sensors** in robots.

12. Clock System and Power Management

✔ **16 MHz Internal RC Oscillator** (Default clock on reset)
✔ **External Clock (4 - 26 MHz) Support**
✔ **PLL for High-Speed Operations (Up to 100 MHz)**

Example: In a **real-time control system**, high-speed clocking ensures **precise motor control**.

13. Boot Modes

✔ **Boot from Flash (Default Mode)** → Normal operation
✔ **Boot from System Memory** → Factory Bootloader (for firmware updates via UART, SPI, I2C, USB)
✔ **Boot from SRAM** → Used in debugging and RAM-based execution

14. Power Supply and Low-Power Modes

✔ **Operates from 1.7V to 3.6V**
✔ **Multiple Low-Power Modes (Sleep, Stop, Standby, Deep Sleep)**
✔ **Supports Battery Backup Mode (VBAT) for RTC and Low-Power Sensors**

Example: Used in **smartwatches**, where low-power mode extends battery life while keeping **real-time clock (RTC) active**.

IMPORTANT PARTS

1. Harvard Architecture

Definition

The **Harvard architecture** is a computer architecture where the **instruction memory** and **data memory** are **physically separate**, each with its own dedicated bus.

How It Works

- The **CPU has two independent memory buses**:
  - One for fetching **instructions** (code).
  - One for reading/writing **data**.
- This allows the **CPU to fetch instructions and access data simultaneously**, leading to **faster execution**.

Comparison with Von Neumann Architecture

Feature	Harvard Architecture	Von Neumann Architecture
Memory Buses	Separate instruction and data buses	Single shared bus for both
Speed	Faster execution	Slower due to memory bottleneck
Parallel Processing	Can fetch instructions while accessing data	Fetching instructions and data happens sequentially
Example	STM32F411CE (Cortex-M4)	Many general-purpose CPUs

Example in Embedded Systems

- **Robotics**: In an STM32 microcontroller, real-time control applications benefit from fast data processing.
- **Digital Signal Processing (DSP)**: Processing audio or video data requires rapid access to both **instructions and data**, making Harvard architecture ideal.

2. 32-bit RISC Processor

Definition

A **32-bit RISC processor** is a CPU that:

1. Uses **32-bit registers and data paths**, meaning it processes **32-bit data at a time**.
2. Follows the **RISC (Reduced Instruction Set Computing)** design philosophy.

Key Features of RISC (Reduced Instruction Set Computing)

- **Simple Instructions** → Executes most instructions in **one clock cycle**.
- **Load/Store Architecture** → Only a few instructions directly access memory (reduces complexity).
- **Fixed-Length Instructions** → Helps in efficient instruction decoding.
- **Pipeline-Friendly** → Instructions are executed in stages, allowing multiple instructions to be in progress simultaneously.

Comparison: RISC vs. CISC (Complex Instruction Set Computing)

Feature	RISC (Used in STM32F411CE)	CISC (Used in x86 CPUs)
Instruction Complexity	Simple, fixed-length	Complex, variable-length
Execution Speed	Faster, often **one instruction per cycle**	Slower due to microcode execution
Power Consumption	Lower	Higher
Example	ARM Cortex-M4	Intel x86

Example in Embedded Systems

- **Motor Control**: The RISC processor in STM32 allows **real-time motor speed adjustments** with minimal delay.
- **Battery-Powered Devices**: Since RISC consumes less power, it is widely used in **IoT devices and wearables**.

3. DSP (Digital Signal Processing) Support

Definition

Digital Signal Processing (DSP) refers to mathematical operations on **real-world signals** (audio, image, motion, etc.) in digital form.

Why is DSP Needed?

Many sensors output **analog signals** (sound waves, light intensity, acceleration). DSP helps:

1. Convert analog data into a digital form.
2. Perform filtering, compression, and analysis.
3. Improve the accuracy and efficiency of real-time systems.

Key DSP Features in STM32F411CE

- **Multiply-Accumulate (MAC) Unit** → Performs **fast mathematical computations**.
- **Saturation Arithmetic** → Prevents overflow errors in calculations.
- **Single-Cycle Multiply** → Performs multiplication in one clock cycle.

Example in Embedded Systems

- **Audio Processing** → Noise cancellation in microphones.
- **Image Processing** → Edge detection in a **robot's camera** for object recognition.
- **Motion Control** → Processing **IMU (Inertial Measurement Unit) data** to stabilize a drone.

4. Floating Point Unit (FPU)

Definition

A **Floating Point Unit (FPU)** is a specialized hardware unit for handling **floating-point (decimal) calculations**efficiently.

Why is FPU Important?

- Many embedded applications require **precise mathematical calculations**.
- Without an FPU, floating-point calculations would be **slow** since they must be handled by **software emulation**.
- With an FPU, these calculations happen in **hardware**, making them much **faster**.

Types of Floating Point Representation

1. **Single-Precision (32-bit float)** → Used in STM32F411CE.
2. **Double-Precision (64-bit float)** → Used in more advanced CPUs.

Performance Benefits of FPU

Without FPU	With FPU
Floating-point operations are slow (emulated in software)	Hardware acceleration for floating-point math
Takes multiple clock cycles	Takes just **one or a few** clock cycles
Not suitable for real-time calculations	Optimized for **real-time processing**

Example in Embedded Systems

- **Robotics** → FPU speeds up inverse kinematics calculations for **robot arm movement**.
- **Drones** → FPU accelerates **quaternion calculations** for precise **flight control**.
- **Machine Learning** → AI models on microcontrollers require fast **matrix operations** (e.g., neural networks).

5. Nested Vectored Interrupt Controller (NVIC)

Definition

The **NVIC (Nested Vectored Interrupt Controller)** is a hardware module that manages **interrupts** in real-time systems.

What is an Interrupt?

An **interrupt** is a signal that pauses the normal execution of a program to handle an **important event**.

How NVIC Works

1. **Prioritizes Interrupts** → High-priority events (e.g., emergency stop) are handled first.
2. **Nested Interrupts** → A higher-priority interrupt **can preempt** (interrupt) a lower-priority one.
3. **Vector Table** → Stores the addresses of **interrupt service routines (ISR)**.

Example in Embedded Systems

- **Self-driving Car**: NVIC ensures **emergency braking (high-priority)** is executed **before less critical tasks (like speed adjustment)**.
- **Industrial Automation**: If a temperature sensor exceeds a threshold, the NVIC triggers an emergency shutdown.
- **Robotics**: When an obstacle is detected, the robot's **motor stop command executes immediately**.

6. Memory Protection Unit (MPU)

Definition

The **Memory Protection Unit (MPU)** is a security feature that **restricts access to certain memory areas**.

Why is MPU Important?

- Prevents **malicious code** or software bugs from corrupting critical memory.
- **Enhances system reliability** in real-time applications.
- Supports **multi-tasking** by isolating processes.

How MPU Works

- The memory is divided into **regions**.
- Each region has **access control rules** (read, write, execute permissions).
- If a process **violates the rules**, an **exception occurs**, preventing memory corruption.

Example in Embedded Systems

- **Secure Bootloader** → Protects firmware from unauthorized modifications.
- **Medical Devices** → Ensures safe execution of software to **prevent life-threatening failures**.
- **RTOS (Real-Time Operating Systems)** → Keeps tasks **isolated** to prevent memory corruption.

Harvard Architecture	Faster execution due to separate instruction/data buses
32-bit RISC Processor	Efficient and power-saving execution of instructions
DSP Support	Accelerates audio, image, and motion processing
FPU	Enhances floating-point mathematical operations
NVIC	Manages real-time interrupts efficiently
MPU	Provides security by preventing unauthorized memory access