**Key steps in the boot process**

**1. Reset Mechanism**

When the system undergoes a reset, it starts the initialization process to ensure that all components are in a known state. There are two primary types of resets:

**Cold Reset**

* A complete reset of the system, clearing all registers and memory states.
* Typically triggered by power-on, a reset pin, or a software command.
* Reinitializes all hardware components, starts the entire boot sequence from scratch.
* **Use case :**

1. **Power-on-initialization :** Ensures the system starts from a completely known state, initializing all hardware components and clearing all memory.

**Warm Reset**

* A partial reset that preserves some of the system states.
* Triggered by software commands, watchdog timers, or specific hardware conditions.
* **Use case :**

1. **Software updates :** Allows updates to be applied without fully reinitializing the hardware, saving time and preserving system states.
2. **System parameters reconfiguration :** Applies new settings without losing the current state of the system.
3. **Fast booting :** Skips some initialization steps by preserving certain states, allowing the system to become operational more quickly.

**2. Branching Code**

* The reset handler code includes branching code that determines the next steps in the boot process.
* This branching code is responsible for verifying the integrity of the system (e.g., checking non-volatile memory) and deciding whether to jump to the bootloader or handle errors.

Upon exiting the reset state, the processor starts executing the branching code located at a predefined reset vector address in the boot ROM.

1. **Execution Start:** The processor begins execution at the reset vector address.
2. **Reset Handler:** The reset vector points to the reset handler, which is the entry point for the initial code execution.

**Reset Vector:**

* The reset vector is the first entry in the vector table and points to the reset handler.
* When the microcontroller exits a reset state, it uses the reset vector to determine where to start executing code.
* The reset vector in the vector table typically points to an address within the boot ROM.

1. **System Initialization:** Basic hardware components are initialized, such as the clock system, memory interface, and essential peripherals.
2. **Branch to Bootloader:** Once basic initialization is complete, the code branches to the bootloader.

**3. Bootloader Code**

The bootloader is responsible for more complex initialization and loading the main application code.

**Functions:**

1. **Hardware Initialization:**

During the boot process, the bootloader performs additional hardware initialization tasks that go beyond the basic setup handled by the reset handler. This includes configuring peripherals and preparing the system for firmware loading and execution.

1. **Peripheral Initialization:**

* Set up UART for serial communication.
* Initialize I2C, SPI, or other communication interfaces needed for firmware loading.
* Configure GPIOs for input/output operations.

2. **Memory Setup:**

* Configure and initialize external memory interfaces (e.g., SDRAM, NAND flash).
* Set up internal memory regions and ensure proper memory mapping.
  1. **Clock Configuration:**
* Configure system clocks and ensure all necessary peripherals are clocked correctly.

**4. Watchdog Timer:**

* Initialize and enable the watchdog timer to ensure system recovery in case of failures during boot.

**Example code :**

**void initialize\_hardware(void) {**

**configure\_uart();**

**init\_i2c();**

**setup\_spi();**

**configure\_gpios();**

**init\_external\_memory();**

**configure\_clocks();**

**enable\_watchdog\_timer();**

**}**

1. **Firmware Loading:**

The bootloader loads the main application firmware from non-volatile memory (e.g., flash memory) into RAM, ensuring that the code is ready for execution

* **Determine Firmware Location:** Identify the location of the firmware image in non-volatile memory.
* **Firmware image :** A **firmware image** is a compiled and packaged binary code that includes all the necessary software components for an embedded device to operate. It is stored in non-volatile memory and loaded into RAM during the boot process, where it is executed by the CPU to perform the device's intended functions. This image is crucial for ensuring that the device boots up correctly, initializes its hardware, and runs its application code reliably.
* **Copy Firmware to RAM:** Transfer the firmware image from non-volatile memory to RAM for faster execution.
* **Handle Errors:** Implement error handling to manage issues like failed reads or corrupted data.

**Example code :**

**void load\_firmware(void) {**

**uint32\_t firmware\_size = get\_firmware\_size();**

**uint32\_t\* source\_addr = (uint32\_t\*)FLASH\_FIRMWARE\_START;**

**uint32\_t\* dest\_addr = (uint32\_t\*)RAM\_FIRMWARE\_START;**

**for (uint32\_t i = 0; i < firmware\_size; i++) {**

**\*dest\_addr++ = \*source\_addr++;**

**}**

**}**

1. **Integrity Checks:**

* To ensure the firmware loaded into RAM is valid and not tampered with, the bootloader performs integrity checks using **checksums or cryptographic signatures.**
* A **cryptographic signature** is a way to validate that a piece of data (like firmware) was created by a known source and has not been altered. It uses a combination of hashing and encryption techniques to produce a signature that can be verified by others.
* In a bootloader, **CRC (Cyclic Redundancy Check)** is used to verify the integrity of firmware by detecting accidental changes or errors in the data. Here’s a step-by-step breakdown of how a CRC check is typically performed by a bootloader:

**1. Define CRC Parameters:**

* The polynomial used for CRC calculation, which defines the algorithm.
* The starting value for the CRC computation.
* A table of precomputed CRC values for faster calculation (optional but common in many implementations).
* The value used to finalize the CRC result.

**2. Read Firmware Data:**

* The bootloader reads the firmware data from non-volatile memory (e.g., flash memory) into a specified memory range for CRC calculation.

**3. Compute CRC:**

* Calculate the CRC value based on the firmware data using the defined polynomial and initial value.
* Use either a direct computation method or a table-based method for efficiency.

**4.Compare CRC Values:**

* Compare the computed CRC value with a known good value (expected CRC) to verify data integrity.

**5. Handle Errors:**

* If the CRC values do not match, handle the error by aborting the boot process, entering a recovery mode, or logging an error.

**Example code :**

**bool verify\_firmware\_integrity(void) {**

**uint32\_t calculated\_checksum = calculate\_checksum(RAM\_FIRMWARE\_START, firmware\_size);**

**if (calculated\_checksum == EXPECTED\_CHECKSUM) {**

**return true;**

**} else {**

**return false;**

**}**

**}**

1. **Conditional Operations:**

The bootloader may perform additional conditional operations based on specific criteria, such as checking for firmware updates or alternate boot modes.

* **Check for Firmware Updates:** Look for a new firmware image in a designated location. This designated location" for a new firmware image can vary depending on the system's architecture, configuration, and bootloader design

**Example :** flash memory, external storage like EEPROM,SD cards etc, Clous services

* **Alternate Boot Modes:** Check for user inputs or flags that indicate different boot modes (**e.g.,** recovery mode, diagnostic mode).
  + 1. **Recovery Mode** : Used to recover or reinstall firmware if the main firmware is corrupted or fails to load correctly. Triggered by specific hardware signals, bootloader flags, or user actions (e.g., holding a button during power-up). Loads a minimal or secondary firmware from a backup location or external storage to replace the corrupted primary firmware.
    2. **Diagnostic Mode:** Used to test and diagnose hardware components and system functionality. Helps identify issues such as hardware malfunctions or configuration problems. Accessed through specific sequences, commands, or software interfaces, often during or after boot-up.

**Example code :**

**void perform\_conditional\_operations(void) {**

**if (check\_for\_firmware\_update()) {**

**load\_new\_firmware();**

**}**

**if (check\_alternate\_boot\_mode()) {**

**enter\_alternate\_boot\_mode();**

**}**

**}**

1. **Jump to Application:** Once the firmware is loaded and verified, the bootloader sets up the environment and jumps to the main application code entry point.

* **Set Up Environment:** Ensure all necessary system configurations are set for the application.
* **Jump to Application Entry Point:** Change the program counter to the start address of the application code in RAM.

**Example code :**

**void jump\_to\_application(void) {**

**// Deinitialize peripherals used by bootloader**

**deinit\_hardware();**

**// Set the vector table base address to the application's vector table**

**SCB->VTOR = (uint32\_t)RAM\_FIRMWARE\_START;**

**// Create a function pointer to the application's reset handler**

**void (\*app\_reset\_handler)(void) = (void (\*)(void))(\*((uint32\_t\*) (RAM\_FIRMWARE\_START + 4)));**

**// Set the main stack pointer to the application's stack pointer**

**\_\_set\_MSP(\*((uint32\_t\*)RAM\_FIRMWARE\_START));**

**// Jump to the application's reset handler**

**app\_reset\_handler();**

**}**

**4. Application Code**

* **Further Initialization:** Configures application-specific hardware and settings. Sets up any additional hardware components that are specific to the application, such as sensors, actuators, communication interfaces, or displays. Configures and initializes peripherals like timers, UARTs, I2C/SPI interfaces, ADCs, and GPIOs according to the application’s requirements. Adjusts system parameters such as clock settings, power management configurations, and communication protocols.

**void initialize\_peripherals(void) {**

**// Initialize UART for serial communication**

**uart\_init();**

**// Configure GPIO pins for output**

**gpio\_setup();**

**// Set up timers for periodic tasks**

**timer\_init();**

**}**

* **Main Execution Loop:** Runs core functionality of the system. Executes the primary tasks and processes that define the application’s behaviour. Continuously checks for events or inputs and processes them as required. Manages the state of the system, transitioning between different operational modes or states based on inputs and conditions.

**int main(void) {**

**// Perform initial setup**

**initialize\_peripherals();**

**// Main execution loop**

**while (1) {**

**// Read sensors**

**read\_sensor\_data();**

**// Process data**

**process\_data();**

**// Update outputs**

**update\_outputs();**

**// Optionally include a delay**

**delay\_ms(100);**

**}**

**}**

* **Interrupt Handling:** Manages real-time events. Manages immediate responses to time-critical events that occur asynchronously to the main execution flow. Defines functions that handle specific interrupts, such as external hardware signals or timer expirations. Saves and restores the system state to handle interrupts without disrupting the main execution flow. Manages multiple interrupts with varying priorities to ensure timely processing of critical events.

**// Timer interrupt handler**

**void TIM2\_IRQHandler(void) {**

**if (TIM2->SR & TIM\_SR\_UIF) {**

**TIM2->SR &= ~TIM\_SR\_UIF; // Clear interrupt flag**

**// Perform timer-based tasks**

**timer\_task();**

**}**

**}**

* **Task Management:** Schedules and executes tasks, often managed by an RTOS.
* **Task Scheduling:** Schedules tasks or threads based on priority, timing, and resource availability.
* **Task Switching:** Handles switching between different tasks or threads, allowing the system to perform multiple operations concurrently.
* **Resource Management:** Allocates and manages system resources (e.g., memory, CPU time) among different tasks.

**#include "FreeRTOS.h"**

**#include "task.h"**

**// Task function**

**void vTaskFunction(void \*pvParameters) {**

**while (1) {**

**// Perform task-specific operations**

**task\_operation();**

**// Yield to other tasks**

**vTaskDelay(pdMS\_TO\_TICKS(100));**

**}**

**}**

**int main(void) {**

**// Initialize hardware**

**initialize\_peripherals();**

**// Create tasks**

**xTaskCreate(vTaskFunction, "Task1", 100, NULL, 1, NULL);**

**// Start the scheduler**

**vTaskStartScheduler();**

**while (1) {**

**// Main loop (should never reach here)**

**}**

**}**