**Bootloader Basics**

The bootloader is a very important component in any operating system. A bootloader, also known as a boot program or bootstrap loader. It is **special operating system software** that loads into the working memory of a computer after start-up. So, you must be clear that this is also software like an application. The term **bootloader** is a shortened form of the words **“*bootstrap loader”***.

**What is a Bootloader in Embedded systems?**

Like a normal OS, the bootloader in a microcontroller also serves the same purpose. This is the first piece of code that runs when you press the reset button if you have a bootloader. If you don’t have a bootloader, then directly an application will start running

If you have a bootloader, then before the application, this bootloader starts running and does some process. Once it is done with the operations, the bootloader job is done. So, it gives control to the application. Now, the application does its job based on our product or project.

**What is the need for Bootloader in Microcontroller?**

**Firmware update**

When you don’t have a bootloader, it is very simple. We are not complicating by writing an extra bootloader. Only application is enough. But when you are planning to sell your products to the customers, what will do if you want to update the application/firmware in the device that you sold already? Every time go to the field and connect the JTAG/J-LINK and flash the firmware or application? It is not possible, right? So, If you have your bootloader, then you don’t need to worry about that. You can update the firmware or application without connecting any debugger or flasher.

**Security**

When you have the product which has to be secured, then what will you do when someone overwrites the application or firmware with their customized firmware to hack your product? How do you find it? In this case, we can use the bootloader to check whether the firmware is valid or not. If it is valid, then only we give control to the firmware or application.

**What happens when you have a bootloader?**

When your project has a bootloader, then you will be having two binaries. One is a Bootloader image and another one is an application. The bootloader will be placed in the 0x00000000 with its Vector Table. The application will be placed in another area of the flash memory with its vector table.

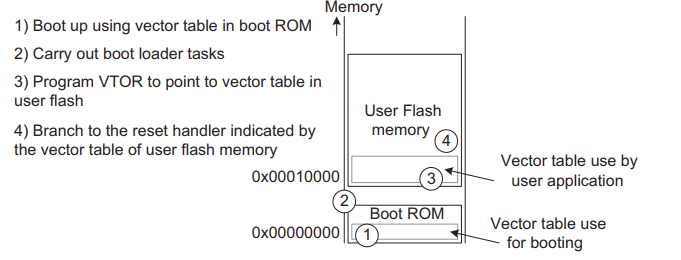
* When you press the reset button, it will start from the bootloader. So, it will copy the Stack pointer to **MSP** (Main Stack Pointer) from the location of 0x00000000.
* Then it moves to the bootloader’s reset handler.
* The bootloader will do some operations based on your need like check for firmware version, upgrade the firmware, etc.
* Once it has done with its operation, it will jump to the application’s vector table.
* Then it will initialize the **MSP** (Main Stack Pointer) again.
* Now we have to tell the controller to use the application’s vector table instead of using the bootloader’s vector table. That will be done using the **Vector Table Offset Register (VTOR)**.
* Then it goes to the application’s reset handler. There it will copy and initialize the data segments to the RAM (Global, Static variables).
* Finally, it moves to the main function of the application.

**Custom Boot Loader**

In some microcontrollers there are multiple program memories: boot ROM and user flash memory. The boot loaders are often pre-programmed in the boot ROM by the microcontroller manufacturer.

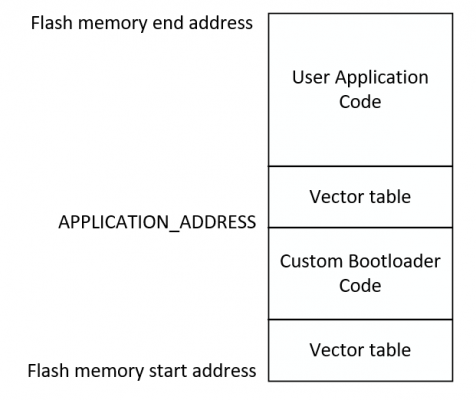
When the Power On Reset occurs:

* the first execute boot loader code in the boot ROM
* then VTOR is programmed to point to 0x0 address of User Flash memory by default
* then final branching to the user application in the flash.



From here, if we want something more specific in booting phase like upgrade firmware or security, we need to build our own boot loader. This boot loader will be similar to boot loader in System Memory, but it is built by yourself and placed in Flash Memory region.

To do that thing, we need two parts in Flash Memory Area:



* The custom boot loader to do some tasks as upgrade image in user application (like System Memory does) or security purpose.
* The user application placed in Flash Memory.

Finally we have the booting scenario:



Note that the VTOR value of Custom Boot Loader will point to start address of Flash, and the VTOR value of User Application will point to APPLICATION\_ADDRESS where it start to flashed into memory.

**Difference between the system bootloader and custom bootloader**

|  |  |
| --- | --- |
| System bootloader | Custom bootloader |
| Definition:   * A **system bootloader**, often referred to as the built-in or ROM bootloader, is a small, fixed program embedded within the microcontroller’s ROM by the manufacturer. * It is typically immutable and provides basic functionalities necessary to initialize the hardware and load subsequent stages of the boot process. | Definition:   * A **custom bootloader** is a user-defined program stored in rewritable non-volatile memory (e.g., internal flash). It is developed and implemented by the end user or application developer. * It provides more advanced and application-specific functionalities compared to the system bootloader. |
| Function :   * **Initial Hardware Initialization:** Initializes basic hardware components such as clocks and memory. * **Boot Source Selection:** Determines the boot source based on pin configurations or other settings (e.g., BOOT pins). * **Firmware Update:** Provides a mechanism to update the firmware via standard communication interfaces (e.g., UART, USB, I2C, SPI). * **Security Checks:** May perform basic security checks, such as verifying the integrity of the code to be executed next. | **Function:**   * **Extended Hardware Initialization:** Performs additional and more complex hardware initialization tailored to the specific application. * **OS/Application Loading:** Loads the operating system or main application from non-volatile storage into RAM and transfers control to it. * **Configuration and Updates:** Provides user interfaces or protocols for configuring system parameters and updating firmware. * **Security Features:** Can implement advanced security features such as encryption, secure boot, and more sophisticated integrity checks. |
| **Characteristics:**   * **Immutable:** Cannot be modified after manufacturing. * **Limited Functionality:** Designed to be small and efficient, providing only essential boot functionality. * **Manufacturer-Specific:** Provided by the microcontroller manufacturer and is common across all devices in a specific family. | **Characteristics:**   * **Modifiable:** Can be updated or replaced as needed to add new features, fix bugs, or improve performance. * **Flexible and Feature-Rich:** Designed to meet specific application requirements, potentially including a wide range of functionalities. * **User-Defined:** Developed by the end user or system integrator, allowing for customization and optimization for the specific application. |

**Boot ROM**

**Boot ROM (Read-Only Memory)** refers to a small, non-volatile memory area embedded within a microcontroller or processor. This memory contains a fixed set of instructions that the CPU executes immediately after a reset or power-on. The code in Boot ROM is typically written by the microcontroller's manufacturer and is not modifiable by the end user

**Functions of Boot ROM**

**1. Initial Hardware Setup**

* **Clock Initialization:** Sets up the system clock sources and frequency.
* The Boot ROM sets the default clock source. For STM32 microcontrollers, this is typically the internal high-speed oscillator (HSI), which runs at 16 MHz.
* The Boot ROM may configure the Phase-Locked Loop (PLL) to achieve a higher system clock frequency if needed. The PLL can take the HSI or an external clock (HSE) as its input and multiply it to generate higher frequencies.
* The Boot ROM configures the number of wait states for Flash memory access based on the system clock frequency to ensure reliable operation.
* The Boot ROM enables the clocks for essential peripherals that are required during the boot process. This might include the GPIO ports, UART for debugging, or other communication interfaces.
* **Peripheral Initialization:** Initializes essential peripherals required for the boot process.

1. **USART/UART Initialization:**
   * **Purpose:** Configures serial communication for debugging or external communication.
   * **Details:** Sets baud rate, word length, parity, stop bits, and enables the UART or USART peripheral.
2. **System Timer Initialization:**
   * **Purpose:** Sets up timers for timekeeping, delays, and scheduling.
   * **Details:** Configures the timer's prescaler, auto-reload value, and interrupt settings.
3. **Flash Memory Initialization:**
   * **Purpose:** Ensures that the Flash memory is ready for reading and writing.
   * **Details:** Configures Flash wait states, enabling read/erase/write operations.
4. **GPIO Initialization:**
   * **Purpose:** Configures general-purpose I/O pins for input or output.
   * **Details:** Sets pin modes (input/output/alternate function), speed, and pull-up/pull-down resistors.
5. **RCC (Reset and Clock Control) Initialization:**
   * **Purpose:** Manages the clock settings for the microcontroller and peripherals.
   * **Details:** Configures clock sources, dividers, and enables peripheral clocks.
6. **Power Management Initialization:**
   * **Purpose:** Manages power settings to ensure stable operation.
   * **Details:** Configures voltage regulators, low-power modes, and power-saving features.

* **Memory Configuration:** Configures memory settings such as memory protection and addressing.

1. **Memory Protection Unit (MPU) Configuration:**
   * **Purpose:** Protects memory regions from unauthorized access or corruption.
   * **Details:** Configures regions with access permissions (read/write/execute), and sets up memory attributes (cacheable, bufferable).
2. **Memory Mapping:**
   * **Purpose:** Defines the layout of memory regions, such as Flash, SRAM, and peripheral addresses.
   * **Details:** Configures base addresses and sizes for different memory regions.
3. **Cache Configuration:**
   * **Purpose:** Manages instruction and data cache settings for performance.
   * **Details:** Configures cache size, policies, and enables/disables caching as needed.
4. **Stack and Heap Initialization:**
   * **Purpose:** Sets up the stack and heap regions for dynamic memory allocation.
   * **Details:** Configures stack size, heap size, and initial pointers.
5. **Boot Configuration:**
   * **Purpose:** Determines how the system starts up and where to find the bootloader or user application.
   * **Details:** Configures boot mode pins or settings that influence the bootloader's behaviour and the start address for execution.

**2. Boot Source Selection**

* **Boot Mode Determination:** Checks the configuration of boot pins (such as BOOT0 and BOOT1) or other settings to determine the boot source.

Based on the boot pins there are three modes

* **Internal Flash:** The microcontroller will start executing code from its internal flash memory if BOOT0 is set to a particular state (e.g., low) and BOOT1 is in a default state.
* **System Memory:** For specific use cases like bootloader execution, the microcontroller may boot from system memory, which contains the factory bootloader code.
* **SRAM:** Some microcontrollers can boot from SRAM, allowing code to be loaded from an external source into SRAM before execution.
* **Source Prioritization:** Selects the appropriate boot source (e.g., internal flash, external flash, system memory, SRAM) based on the determined boot mode.

**3. Firmware Loading**

* **Built-in Bootloader Execution:** If the boot source is system memory, the Boot ROM may contain a built-in bootloader that provides mechanisms to load firmware from various interfaces (e.g., UART, USB, I2C).
* **Jump to Bootloader/Application:** Transfers control to the selected bootloader or directly to the application code in the chosen memory.

**4. Security and Integrity Checks**

* **Authentication:** Verifies the integrity and authenticity of the firmware to be executed next, ensuring it has not been tampered with.
* **Error Handling:** If verification fails, the Boot ROM can trigger error handling routines or fall back to a safe boot mode.

**Boot ROM Code**

**Boot ROM code** refers to the initial set of instructions stored in the Boot ROM. These instructions are designed to perform essential tasks such as:

* Initial hardware setup (e.g., clock initialization, basic peripheral setup)
* Boot source selection (e.g., determining where to fetch the next stage of the boot process based on pin configurations or other settings)
* Basic security checks (e.g., verifying the integrity of the code to be executed next)

**System Bootloader**

The **system bootloader** is a part of the Boot ROM code that provides additional functionality for loading and updating firmware. It is often referred to as the built-in bootloader. The system bootloader typically supports various interfaces for firmware updates and recovery operations, such as UART, USB, I2C, and SPI.

**Relationship Between Boot ROM, Boot ROM Code, and System Bootloader**

* **Boot ROM**: The physical read-only memory area in the microcontroller.
* **Boot ROM Code**: The initial set of instructions stored in the Boot ROM, including the system bootloader.
* **System Bootloader**: A component of the Boot ROM code that facilitates firmware updates and other essential boot operations.

**How They Work Together**

1. **Power-On or Reset:**
   * The CPU begins executing the Boot ROM code stored in the Boot ROM.
2. **Initial Execution (Boot ROM Code):**
   * Performs initial hardware setup.
   * Determines the boot source based on BOOT pin configurations or other settings.
3. **System Bootloader Execution (If Applicable):**
   * If the boot source is configured to boot from system memory, the system bootloader code within the Boot ROM is executed.
   * The system bootloader provides an interface for loading new firmware via supported communication interfaces (e.g., UART, USB).
4. **Transfer of Control:**
   * Depending on the boot mode, the Boot ROM code (or system bootloader) transfers control to the next stage, which could be:
     + Main application code in internal flash memory.
     + Custom bootloader stored in non-volatile memory.
     + Code in SRAM (for debugging purposes).

**Example in STM32F4xx Microcontrollers**

In STM32F4xx microcontrollers, the Boot ROM includes both the Boot ROM code and the system bootloader:

1. Boot ROM Code Execution: Upon reset, the CPU executes the Boot ROM code, which performs initial hardware setup and configuration. This code is responsible for the basic system initialization, including setting up the stack pointer and configuring memory access.
2. System Bootloader Execution: After the Boot ROM code has completed its setup tasks, control may be passed to the system bootloader. The system bootloader then performs additional tasks such as loading the main application or handling firmware updates.

The BootROM (also known as the ROM bootloader) is a small piece of code that is stored in the device's ROM. When the device is powered on, the BootROM is executed first. The BootROM's main task is to load the next stage of the boot process, which is typically a bootloader or operating system.  
  
The BootROM loads images in a number of different ways, depending on the device. Some common methods include:  
  
Direct memory access (DMA): The BootROM uses DMA to transfer the image directly from the storage device to memory. This is the most common method for loading images from flash memory.  
Serial communication: The BootROM uses a serial protocol, such as UART or USB, to transfer the image from a host computer to memory. This method is often used for development and debugging purposes.  
Network boot: The BootROM uses a network protocol, such as DHCP or TFTP, to transfer the image from a network server to memory. This method is often used for embedded devices that are not connected to a host computer. But this method makes a compulsion to have a networked connection.  
  
Once the image has been loaded, the BootROM passes control to it. The next stage of the boot process will then load the operating system and start the device up.

**BOOTROM**

The BOOTROM is the very first piece of instruction to be executed when a device boots up. Or when a system undergoes hard reset (may be due to hardware reset/software reset/ power-on reset or watchdog reset), the control lands up here, in the BOOTROM.

BOOTROM (also known as internal ROM) is a non-modifiable (Read only) area of memory which is written once during manufacturing. Mind you, BOOTROM can never be modified by a developer, and reading the content of the BOOTROM is no easy task too.

There is no general guidelines as to what should be accomplished inside the BOOTROM, but there are some common functionalities attributed to it. Being the very first piece of software to run, the BOOTROM performs certain basic functionalities like initializing the RAM, setting up the hardware PLL, RC oscillators, clocks, initializes a software watchdog (the good old watchdog!), checks the authenticity of the next stage bootloader by validating its RSA signatures, CRCs etc. The quint essential task of the BOOTROM is to load the next stage software into the RAM (done by pointing the PC to this new address). This software is usually an early stage bootloader that is picked up from the flash memory (UFS or emmc), or a USB stick based on the hardware PIN settings on the board.

Manufactures fight the temptation to add too many features into the BOOTROM, as this is one of the most critical sections of software, from the profit point of view. Any bug in BOOTROM requires a factory re-flashing, and hence a total recall. So better be safe than sorry. Apple faced a similar issue form a BOOTROM bug that affected millions of iphones [3]. For more resources on BOOTROM, you can refer to the reference links I have provided below [1][2].

In my next article, I will be talking be about the early stage bootloaders and their core functionalities

The boot flow in an ARM Cortex-M4 microcontroller typically follows these stages:

1. **Power-On Reset (POR):** When the device is powered on, a reset signal initializes the microcontroller. This ensures that all components start from a known state.
2. **Start-Up Code:** After the reset, the processor starts executing the start-up code. This is usually located at the beginning of the flash memory, in a specific vector table that includes the initial stack pointer and the reset vector address.
3. **System Initialization:**
   * **Reset Handler:** The processor jumps to the reset handler defined in the vector table. This handler typically performs essential initialization tasks, such as setting up the system clock and configuring basic hardware settings.
   * **SystemInit Function:** This function is often called early in the boot process to configure the system clock and other essential peripherals.
4. **Bootloader (Optional):** Some systems include a bootloader stage. This code, if present, is responsible for loading the main application firmware. The bootloader may check for firmware updates or load different applications based on certain conditions.
5. **Main Application:** Once the system initialization is complete and any optional bootloader tasks are done, the processor jumps to the main() function of the main application. The startup code typically sets up the vector table to point to the application's entry point.
6. **Initialization of Application:** The main application initializes its own peripherals and system components, sets up tasks (in an RTOS environment), and begins executing its primary functionality.
7. **Normal Operation:** After initialization, the microcontroller enters its normal operational mode, executing the main application code and handling interrupts and other events as defined by the application.

In ARM Cortex-M4, the boot flow is designed to be relatively simple compared to more complex processors, focusing on quick initialization and readiness for application execution. The Cortex-M4’s architecture supports efficient handling of interrupts and real-time tasks, making it well-suited for embedded applications.