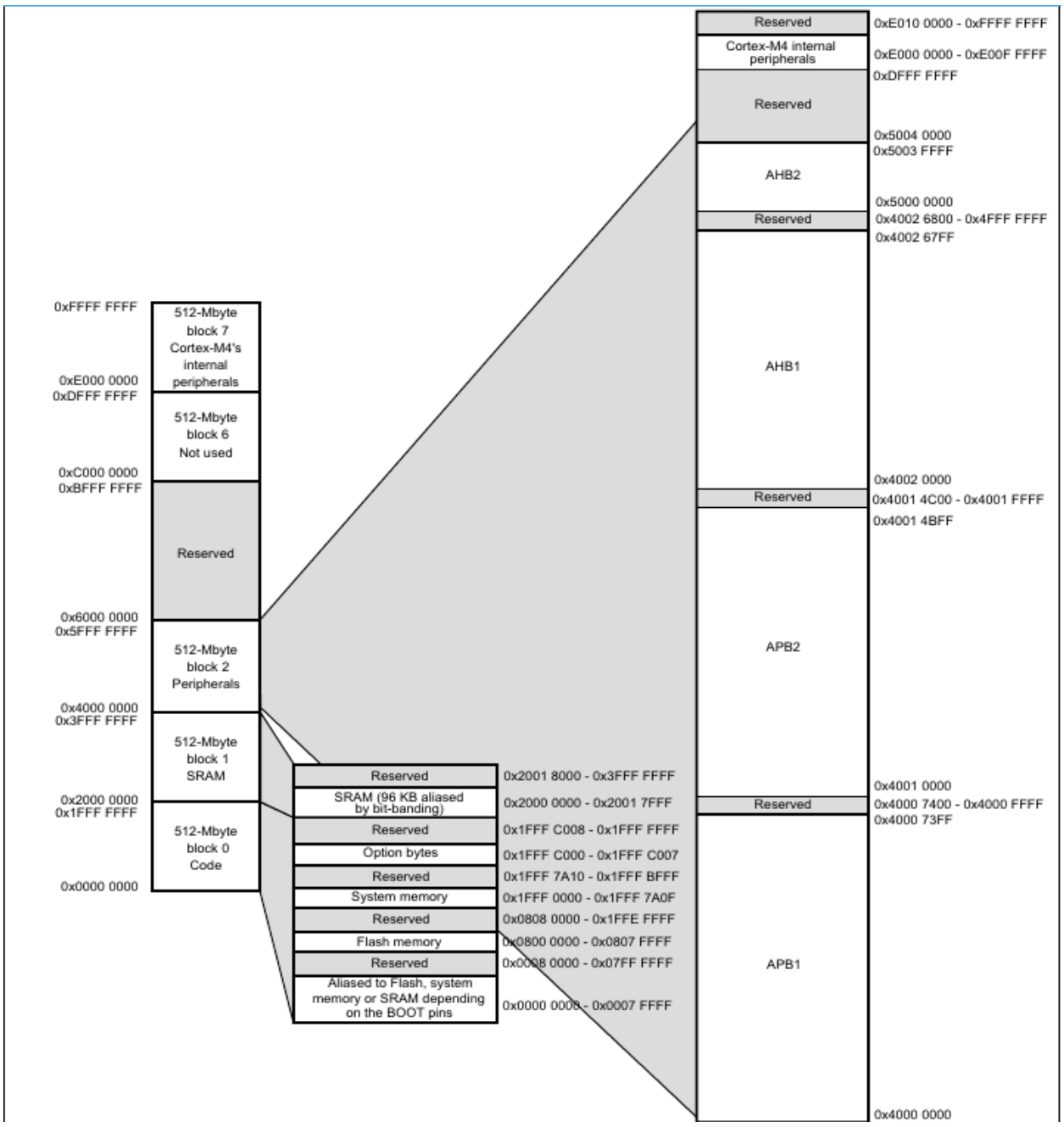
Index

Index..... 1
1. Memory Mapping..... 2
1.1. Code and Memory Space (0x0000 0000 - 0x1FFF FFFF)..... 3
1.2. SRAM (0x2000 0000 - 0x3FFF FFFF)..... 3
1.3. Peripheral Space (0x4000 0000 - 0x5FFF FFFF)..... 3
1.4. External RAM (0x6000 0000 - 0x9FFF FFFF)..... 4
1.5. External Devices (0xA000 0000 - 0xDFFF FFFF)..... 4
1.6. System Control Space (0xE000 0000 - 0xE00F FFFF)..... 4
1.7. Private Peripheral Bus (0xE004 0000 - 0xE004 0FFF)..... 4
1.8. Internal block diagram..... 5
2. Boot ROM Code in the ARM Cortex-M4..... 7
2.1. The boot sequence can be divided into four parts..... 7
2.1.1. Power on Reset (POR)..... 7
2.1.2. Memory Aliasing (Remapping) and Architecture (Hardware Process)..... 8
2.1.3. Firmware Booting (Hardware and Firmware Process)..... 9
2.1.4. Reset_Handler() execution..... 10
2.2. Boot from System Memory..... 11
2.2.1. Why Use the Built-in Bootloader?..... 11
1. Recovery Mode..... 11
2. Firmware Updates..... 11
3. Ease of Use..... 12
2.2.2. How does it know about the firmware corruption?..... 12
2.3. Boot Process After Power Restoration..... 12
2.4. Example..... 13
2.4.1. Flash Memory and Code Location..... 13
2.4.2. Accessing the Boot Process from the Reset_Handler()..... 13
2.4.2.1. Flashing the Code:..... 13
2.4.2.2. Startup File and Reset_Handler():..... 13
2.5. Test Cases for Boot ROM Code..... 14

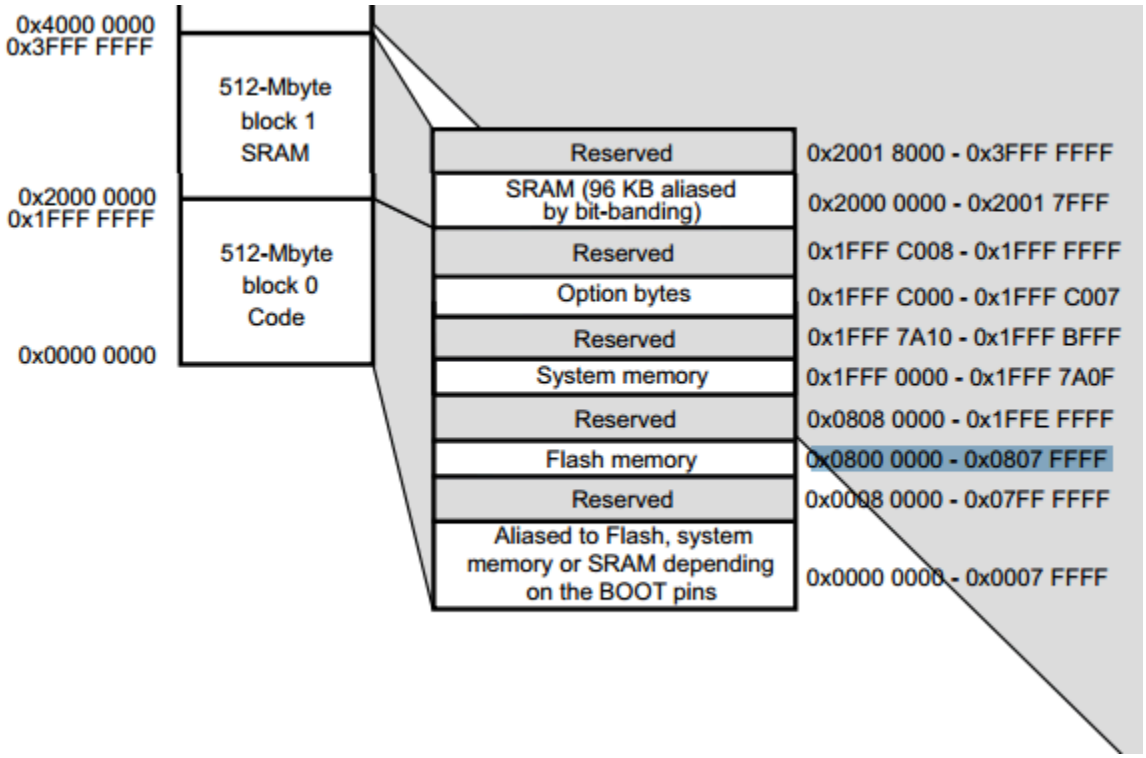
1. Memory Mapping

Memory mapping for the STM32 Nucleo F401RE involves the allocation of specific memory regions for different types of memory and peripheral registers. The ARM Cortex-M4 core used in this microcontroller follows a specific memory map standard.



1.1. Code and Memory Space (0x0000 0000 - 0x1FFF FFFF)

- Reserved for system memory and code and flash memory, which holds the user code. On the STM32F401RE, this space is divided into sectors.
- **0x1FFF 0000 - 0x1FFF 77FF:** System memory (System Bootloader). This region contains the boot ROM code provided by STMicroelectronics for factory programming.



1.2. SRAM (0x2000 0000 - 0x3FFF FFFF)

- **0x2000 0000 - 0x2001 7FFF:** SRAM (96 KB), which is used for data storage during execution.

1.3. Peripheral Space (0x4000 0000 - 0x5FFF FFFF)

- **0x4000 0000 - 0x4002 33FF:** Peripheral registers, including GPIO, USART, I2C, SPI, etc.

1.4. External RAM (0x6000 0000 - 0x9FFF FFFF)

- This region is used for external memory connected through the FSMC (Flexible Static Memory Controller).

1.5. External Devices (0xA000 0000 - 0xDFFF FFFF)

- Reserved for external devices.

1.6. System Control Space (0xE000 0000 - 0xE00F FFFF)

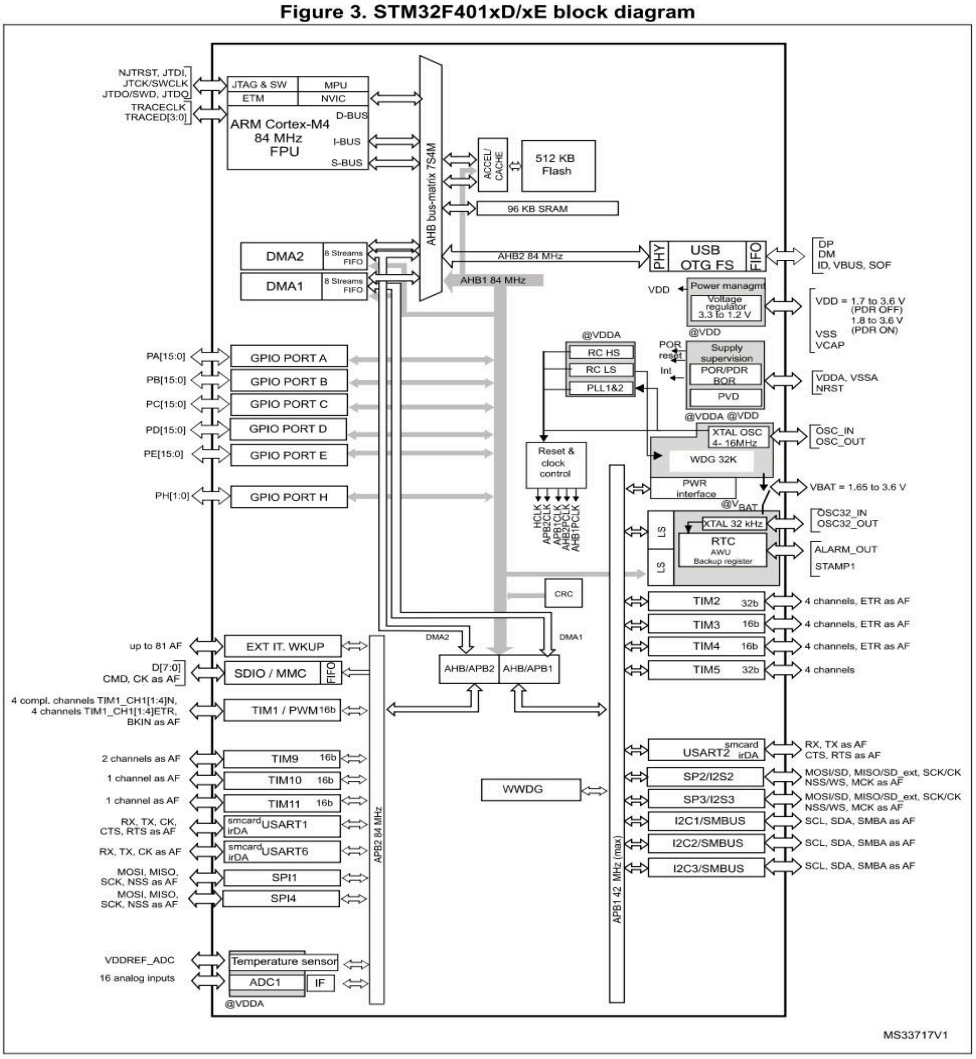
- **0xE000 0000 - 0xE00F FFFF:** System control space, which includes NVIC (Nested Vectored Interrupt Controller), SysTick Timer, and other core peripherals.

1.7. Private Peripheral Bus (0xE004 0000 - 0xE004 0FFF)

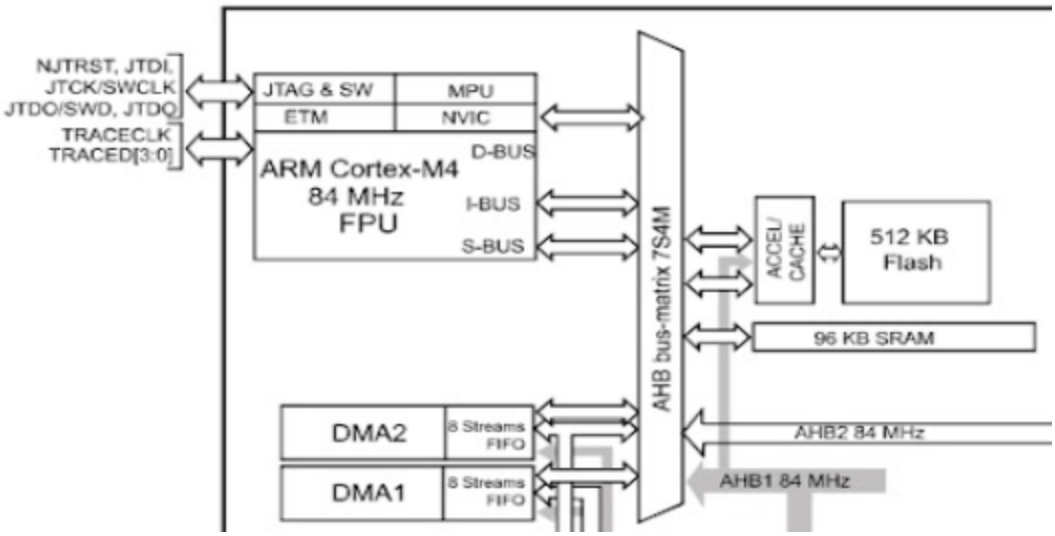
- This region includes additional system control and debug components.

1.8. Internal block diagram

Figure 3. STM32F401xD/xE block diagram



1. The timers connected to APB2 are clocked from TIMxCLK up to 84 MHz, while the timers connected to APB1 are clocked from TIMxCLK up to 42 MHz.


NJTRST, JTDI,
JTCK/SWCLK
JTDO/SWD, JTDO
TRACECLK
TRACED[3:0]
JTAG & SW
MPU
ETM
NVIC
ARM Cortex-M4
84 MHz
FPU
D-BUS
I-BUS
S-BUS
AHB bus-matrix 7S4M
ACCEL/
CACHE
512 KB
Flash
96 KB SRAM
DMA2
8 Streams
FIFO
DMA1
8 Streams
FIFO
AHB2 84 MHz
AHB1 84 MHz

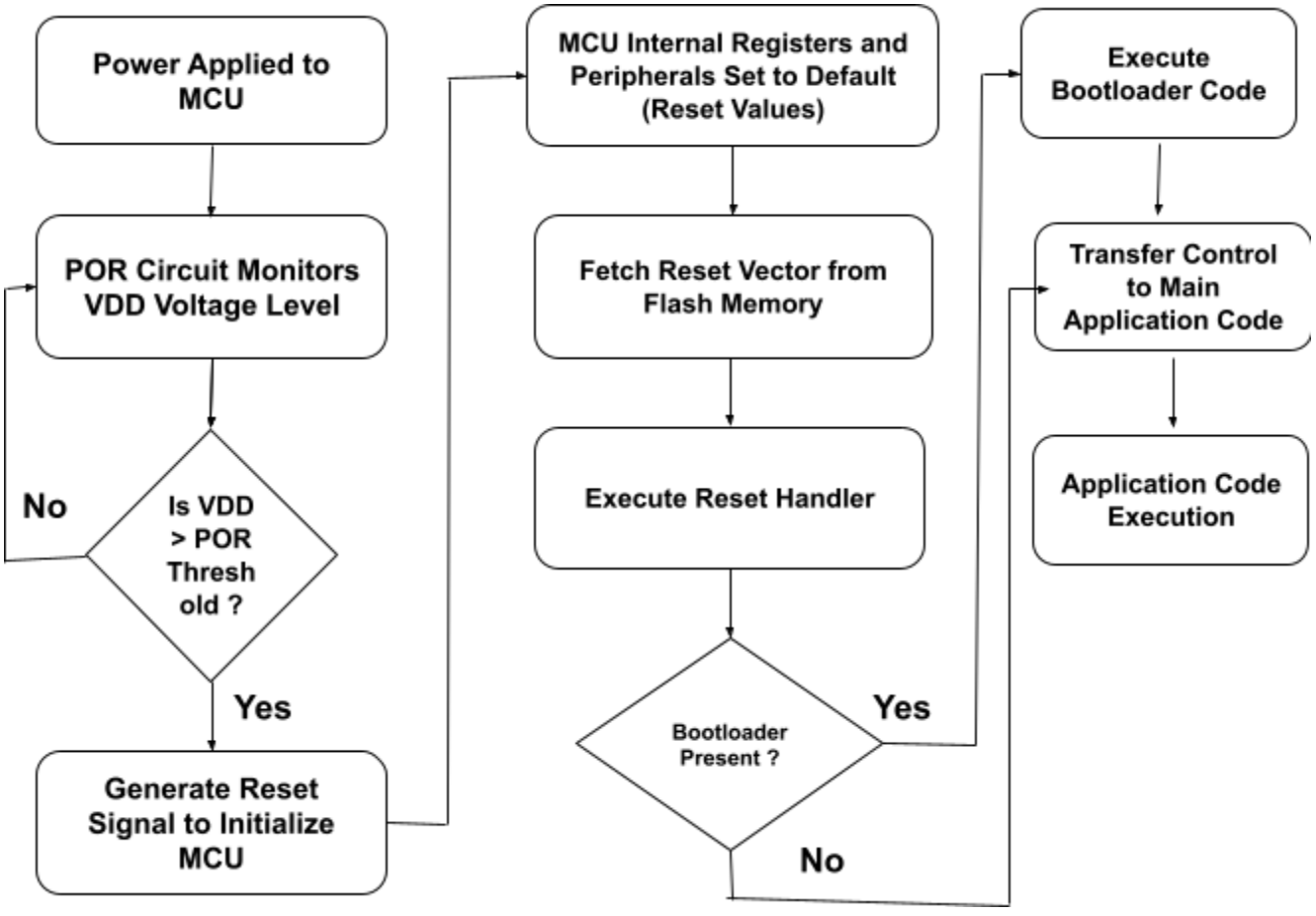
2. Boot ROM Code in the ARM Cortex-M4

2.1. The boot sequence can be divided into four parts

1. Power on Reset (Hardware Process)
2. Memory Aliasing (Remapping) and Architecture (Hardware Process)
3. Firmware Booting (Hardware and Firmware Process)
4. Reset_Handler() execution , for bringing up firmware (Firmware Process)

2.1.1. Power on Reset (POR)

2.1.1.1. Flowchart



2.1.1.2. Reset Triggered

- The boot process starts when the microcontroller is reset, either due to a power-on reset, external reset , or a software reset. The voltage levels rise from 0V to their nominal operating values. The POR circuitry monitors this voltage.

- The POR circuitry includes a voltage detector that ensures the supply voltage reaches a predefined threshold. If the voltage is below this threshold, the reset circuitry keeps the processor in a reset state.
- Once the voltage is stable and above the threshold, the reset signal is generated, which means the processor can start initialising.
    - **Peripheral Reset:** All peripherals are reset to their default states.
    - **CPU Reset:** The CPU and internal buses are reset.
    - **Memory Initialization:** Including Flash and SRAM, is reset and initialised.
- The ARM Cortex-M4 processor starts execution from address **0x00000000.** The processor fetches the initial stack pointer value from this address.

2.1.2. Memory Aliasing (Remapping) and Architecture (Hardware Process)

2.1.2.1. Remapping Mechanism

- **Boot Memory:** The memory remapping is handled by internal hardware based on the BOOT0 pin configuration. This hardware ensures that the correct memory region is accessible at the start of execution.

2.1.2.2. Boot Pin Configuration and Execution

- **Before the processor Executes Code** : As soon as the Microcontroller is reset, the boot pins are checked to determine the boot mode.
- BOOT0 pins on the STM32 microcontroller that can be used to select the boot mode.

- **Boot from Flash memory (BOOT0 = 0):**
    - **Memory Mapping :**
        - This is the default mode for most applications.
        - When BOOT0 is set to 0, Flash memory is mapped to the start (0x00000000) of the memory space at address 0x08000000.
        - The vector table, which includes the initial stack pointer and the reset vector, is located at the beginning of Flash memory.
- **Connect BOOTO to Vcc :**
    - Use a jumper wire or a switch to connect the BOOT0 pin to the Vcc (3.3V) pin.
    - Alternatively, some development boards, like the STM32 Nucleo boards, come with jumpers or switches that can be used to set the BOOT0 pin high.
    - After setting the BOOT0 pin high, reset or power cycle the microcontroller. This will cause it to boot from the system memory (the built-in bootloader).
    - **Memory Mapping:**
        - When BOOT0 is set to 1, system memory is mapped to address 0x1FFF0000.

- **Boot from SRAM :**
    - SRAM is not used for booting because it is volatile and not intended for storing program code. SRAM is used for runtime data and stack, not for storing and executing application code.

- **Internal Logic :**
    - The decision to boot from Flash or system memory is hardcoded into the internal startup logic of the STM32 microcontroller. This logic is part of the chip's startup sequence and is not user-accessible.

2.1.3. Firmware Booting (Hardware and Firmware Process)

- **Fetch Initial Stack Pointer :**
    - The ARM Cortex-M processors have two stack pointers : The main stack Pointer (MSP) and the Process Stack Pointer (PSP). On reset, the MSP is used by default.
    - **If BOOT0 is LOW:**
        - The processor reads the value at address 0x00000000 (which is mapped to Flash memory at 0x08000000). The processor fetches the initial stack pointer from this address.
        - The fetched value is loaded into the main stack pointer (MSP) register.
    - **If BOOT0 is HIGH :**
        - The processor reads the value at address 0x00000000 (which is mapped to System memory at 0x1FFF0000). The processor fetches the initial stack pointer from this address.
        - The fetched value is loaded into the main stack pointer (MSP) register.
- **Stack in SRAM :**
    - The stack pointer (SP) register holds the current value of the stack pointer.
    - By starting the stack at the top of SRAM and growing downward, the stack efficiently uses available memory. As functions are called and local variables are pushed onto the stack, it moves towards lower memory addresses, which leaves the higher memory addresses available for other uses.
    - Downward stack growth simplifies the calculation of stack pointers and memory allocation. The stack pointer (SP) decreases as data is pushed onto the stack and increases as data is popped off.

- **Main Stack Pointer (MSP) :**
    - The MSP is a Special-purpose register in ARM Cortex-M processors.
    - The MSP is the default stack pointer used by the processor in privileged mode, which includes the reset handler, exception handlers, and any privileged code.
    - When an interrupt or exception occurs, the processor automatically uses the MSP to manage the stack frames for these events.
    - This keeps the interrupt stack separate from the main application stack if the Process Stack Pointer (PSP) is used.
    - In many systems, the application code can switch to use the Process Stack Pointer (PSP) for regular code execution, allowing the MSP to be dedicated to handling system-level operations and interrupts.
    - This separation enhances system reliability and security.
    - The initial value of the MSP is set to the top of SRAM (0x20018000 for STM32F401RE with 96 KB of SRAM).

- **Fetch the Reset Vector :**
    - The processor then reads the address of the reset handler from the address immediately following the initial stack pointer in the vector table.
    - **Address :** 0x00000004 of the mapped region (e.g., 0x08000004 or 0x1FFF0004).
    - This address is loaded into the Program Counter (PC), directing the processor to start execution from the reset handler.

2.1.4. Reset_Handler() execution

- The Reset Handler is a function defined in the startup code, which is executed immediately after a reset event. Its primary purpose is to prepare the microcontroller's environment so that the application code can run correctly. This includes setting up the stack, initializing hardware, and calling the main function.

- **Detailed Steps in the Reset Handler**
    - **Stack Pointer Initialization:**
        - The stack pointer is set to the top of the SRAM, which is typically defined by the linker script.
    - **Copy Data Section:**
        - The .data section, which contains initialised global and static variables, is copied from Flash memory to SRAM. This is necessary because Flash memory is read-only, and these variables need to be writable.
    - **Zero Initialize the .bss Section:**
        - The .bss section, which contains uninitialized global and static variables, is zero-initialised in SRAM.
    - **System Initialization:**
        - System-level initialization functions are called. This typically includes setting up the clock system, configuring the Flash memory interface, and other hardware-specific initialization.
    - **Call the SystemInit Function :**
        - SystemInit() is used to configure the system to a specific operational state required by the application.
        - The SystemInit() function in the Reset_Handler() is called after the reset signal to perform additional system-level initializations that are not covered by the hardware reset process. Although the reset signal initializes the hardware peripherals, CPU, and memory to a default state
        - **Clock Configuration:** The hardware reset sets the system clock to a default low-frequency internal oscillator. For many applications, a higher clock frequency is required. SystemInit() configures the system clock (e.g., enabling and configuring the PLL to use an external crystal oscillator).
        - **Memory Configuration:** Configure the Flash latency and prefetch settings according to the new clock speed. Enable caches and other memory features.
        - **Peripheral Initialization:** Enable and configure system peripherals such as GPIO ports, timers, UARTs, and other essential peripherals needed

early in the application. Configure the vector table location, if needed, especially if the application uses an alternative location for the vector table.
        - **System Interrupt Configuration:** Set up priority grouping for system interrupts. Enable necessary system interrupts that need to be active immediately.
        - **Power Configuration:** Adjust power settings for different power modes (e.g., enabling power scaling for high-performance modes or setting up low-power modes).
    - **Call the main Function :**
        - Finally, the Reset Handler calls the main function, where the user application code begins execution.

2.2. Boot from System Memory

Boot from system involves using the built-in bootloader stored in the system memory of the STM32F401RE microcontroller.

2.2.1. Why Use the Built-in Bootloader?

1. Recovery Mode

If the application firmware becomes corrupted, the built-in bootloader provides a way to recover by reprogramming the flash memory.

- **Corrupted Firmware:** If your application firmware gets corrupted or fails to execute correctly, the device may become unresponsive or unable to boot from the main flash memory.
- **Recovery Process:** The built-in bootloader allows you to reprogram the flash memory by entering a recovery mode. This mode can be used to upload a new or corrected firmware image to the microcontroller.
- **How It Works:** When the microcontroller is powered on with the BOOT0 pin set high, the bootloader code in the system memory executes instead of the application code in flash. The bootloader can then communicate with a host (e.g., via UART, USB) to receive and program new firmware into the flash memory.

2. Firmware Updates

The built-in bootloader enables updating the firmware without requiring a debugger or development environment.

- **Ease of Updates:** Using the built-in bootloader simplifies the process of updating the firmware. This can be done in the field without specialized tools or a development environment.
- **How It Works:** The bootloader listens for firmware update commands from a connected device (e.g., a PC or another microcontroller). Once a valid firmware image is received, the bootloader programs it into the flash memory.
- **Methods:** The firmware can be updated via supported communication interfaces such as USART, USB, CAN, etc.

3. Ease of Use

Using a bootloader simplifies the process of deploying new firmware, especially in field updates.

- **User-Friendly:** The built-in bootloader abstracts the complexities of flash programming, providing a simple interface for updating firmware.
- **Field Updates:** In scenarios where devices are deployed in the field, having a built-in bootloader allows for easy and reliable updates without needing to bring the device back to a service centre or use a debugger.

2.2.2. How does it know about the firmware corruption?

Failure to Boot:

- The microcontroller does not start or gets stuck in an infinite reset loop.

Unexpected Behaviour:

- The device behaves erratically, performs incorrect operations, or crashes frequently.

Communication Failures:

- Loss of communication with peripheral devices or external systems.

2.3. Boot Process After Power Restoration

- **Power Restoration:**
  - When power is restored, the microcontroller performs a power-on reset (POR). During this reset, the internal state is reinitialized.
- **Reinitialization:**
  - The microcontroller starts by checking the state of the BOOT0 pin and mapping the appropriate memory region to the start of the address space (0x00000000).
- **Vector Table Access:**
  - The vector table, which contains the initial stack pointer and reset vector, is read from Flash memory or system memory.
- **MSP Setup:**
  - The stack pointer (MSP) is initialized with the value read from the vector table. Since SRAM was cleared when the power was lost, the stack pointer will point to a new stack area in SRAM after power restoration.
- **Execution:**
  - The processor starts executing from the reset handler as per the address provided in the vector table.

2.4. Example

When you flash code onto the STM32F401RE microcontroller using STM32CubeIDE, the code is typically loaded into Flash memory, starting from a specified address.

2.4.1. Flash Memory and Code Location

1. **Code Location:**
    - On STM32F401RE, the user application code is generally placed in the Flash memory starting at address 0x08000000.
    - This address is where the default bootloader looks to find the application code when the microcontroller starts up with the BOOT0 pin low.
2. **File Extensions:**
    - **Binary Files (.bin):** Raw binary files that can be directly written to Flash memory.
    - **Hex Files (.hex):** Intel Hex format files, often used for loading firmware into microcontrollers.
    - **ELF Files (.elf):** Executable and Linkable Format files used during debugging, which include both code and debugging information.

2.4.2. Accessing the Boot Process from the Reset_Handler()

To understand how the boot process works and how it is initiated from the Reset_Handler(), follow these steps:

2.4.2.1. Flashing the Code:

- **Compile and Build:** When you compile your project in STM32CubeIDE, the IDE generates an executable file (e.g., .elf) which is then converted into a binary or hex file for flashing.
- **Flashing Tool:** STM32CubeIDE uses a tool like STM32CubeProgrammer to flash the binary or hex file into the Flash memory of the STM32F401RE.

2.4.2.2. Startup File and Reset_Handler():

- **Startup File:** This file, typically named startup_stm32f401xe.s or startup_stm32f4xx.s, contains the vector table and the Reset_Handler() function. It is responsible for initializing the microcontroller when a reset occurs.
- **Vector Table:** The vector table, located at the beginning of Flash memory (address 0x08000000), includes the address of Reset_Handler() as its second entry. The first entry contains the initial stack pointer value.
- **Reset_Handler() Function:** This function is executed immediately after the microcontroller resets. It is responsible for setting up the system and preparing it for the application code.

Here's how the boot process unfolds from the Reset_Handler():

1. **Initial Setup:**
    - **Stack Pointer:** The Reset_Handler() initializes the stack pointer with the value loaded from the vector table.

    - **System Initialization:** It typically calls the SystemInit() function to configure the system clock and other hardware components.
2. **Code Execution:**
    - **Copy Data Segments:** The Reset_Handler() copies initialized data from Flash to SRAM.
    - **Zero BSS Segments:** It initializes uninitialized global and static variables to zero.
    - **Application Start:** Finally, it calls the main() function, which starts the user application.
3. **Boot from Flash:**
    - When the BOOT0 pin is low, the microcontroller boots from Flash memory at address 0x08000000.
    - The Reset_Handler() is executed from this Flash address, which then leads to the execution of your application code.

2.5. Test Cases for Boot ROM Code

1. **Test Case: Stack Pointer Initialization**
    - **Objective:** to verify that the stack pointer is correctly initialized to the top of the RAM.
    - **Steps:**
        1. Reset the microcontroller.
        2. Allow the Boot ROM code to run.
        3. Check the value of the stack pointer (SP).
    - **Expected Result:** The SP should point to the top of the RAM as defined in the linker script (e.g., _estack).
2. **Test Case: Vector Table Setup**
    - **Objective:** to verify that the interrupt vector table is correctly set up and the reset vector points to the reset handler.
    - **Steps:**
        1. Reset the microcontroller.
        2. Allow the Boot ROM code to run.
        3. Check the address of the reset vector in the vector table.
    - **Expected Result:** The reset vector should point to the Reset_Handler function.
3. **Test Case: Data Section Initialization**
    - **Objective:** to verify that the data section is correctly copied from Flash to RAM.
    - **Steps:**
        1. Define some initialized data variables in the application code.
        2. Reset the microcontroller.
        3. Allow the Boot ROM code to run.
        4. Check the values of the initialized data variables in RAM.
    - **Expected Result:** The initialized data variables in RAM should have the values defined in Flash.
4. **Test Case: BSS Section Zero-Initialization**
    - **Objective:** Verify that the BSS section is correctly zero-initialized.
    - **Steps:**
        1. Define some uninitialized data variables (BSS section) in the application code.
        2. Reset the microcontroller.
        3. Allow the Boot ROM code to run.
        4. Check the values of the uninitialized data variables in RAM.

- **Expected Result:** The uninitialized data variables in RAM should be zero.

5. **Test Case: Jump to Main Application**
   - **Objective:** Verify that the Boot ROM code correctly jumps to the main application.
   - **Steps:**
     1. Implement a simple main function in the application code that sets a known value in a specific memory location.
     2. Reset the microcontroller.
     3. Allow the Boot ROM code to run.
     4. Check the specific memory location for the known value.
   - **Expected Result:** The specific memory location should contain the known value set by the main function.