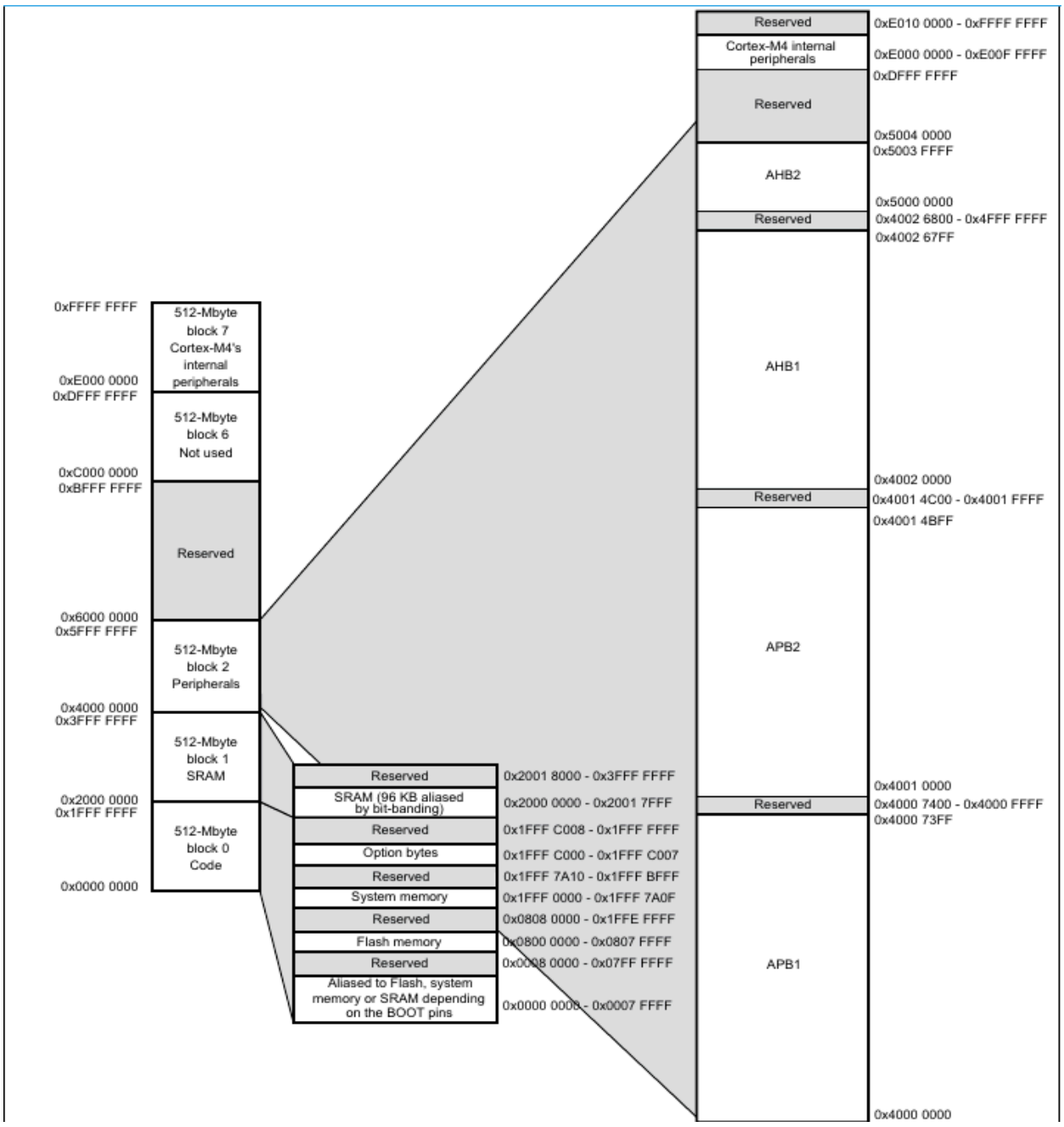
INDEX

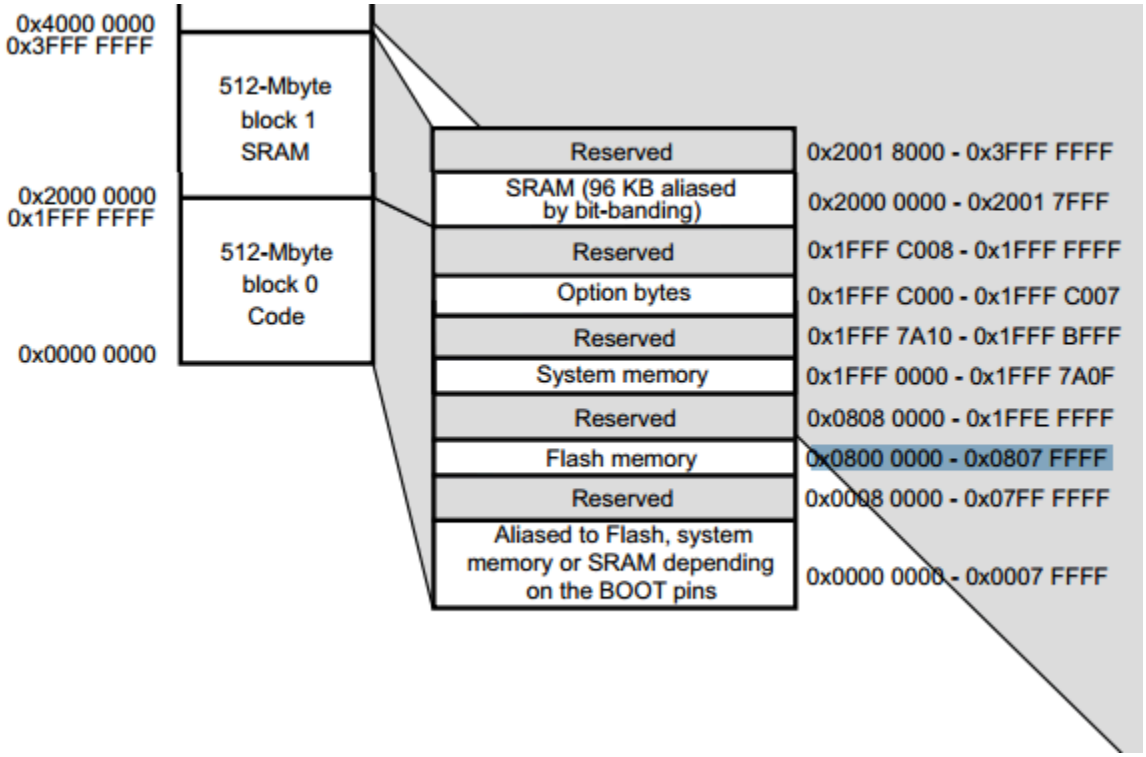
1. Memory Mapping

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)

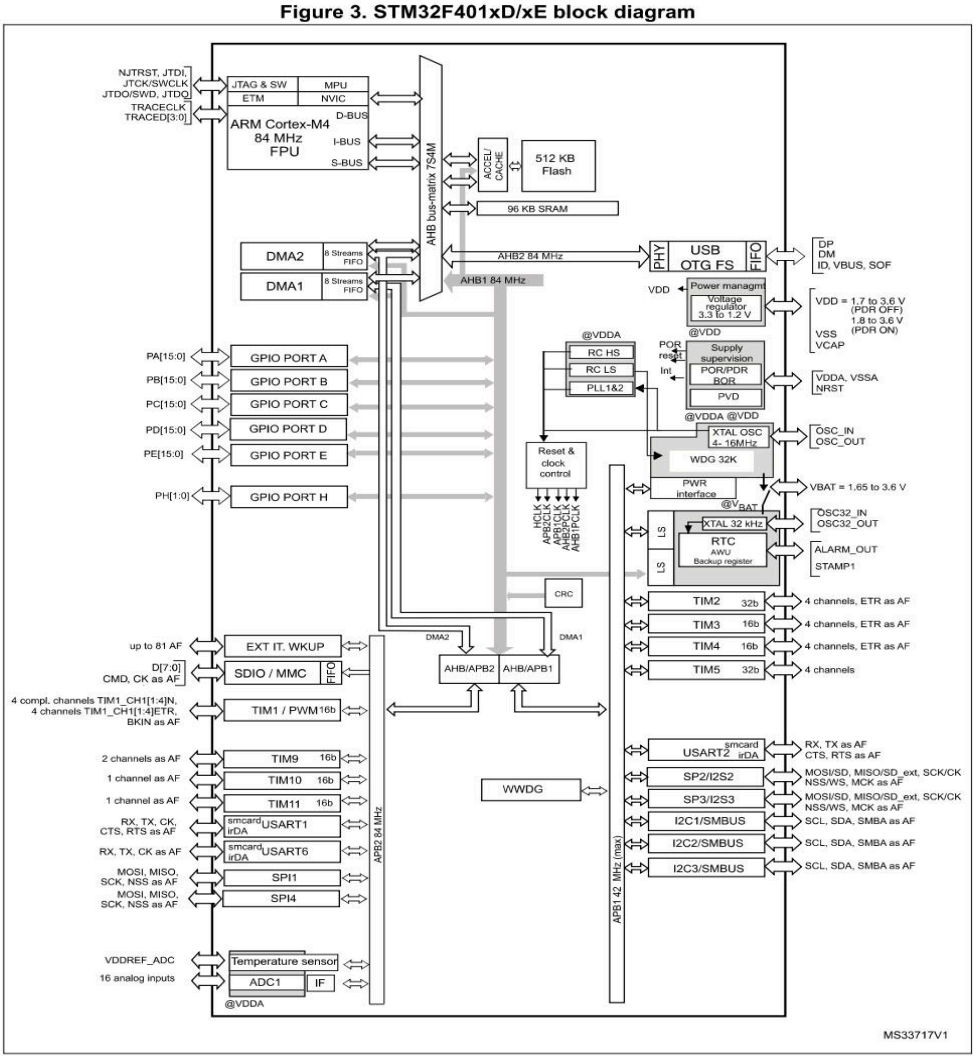
- **0xE000 0000 - 0xE00F FFFF:** System control space, which includes NVIC (Nested Vectored Interrupt Controller), SysTick Timer, and other core peripherals.

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

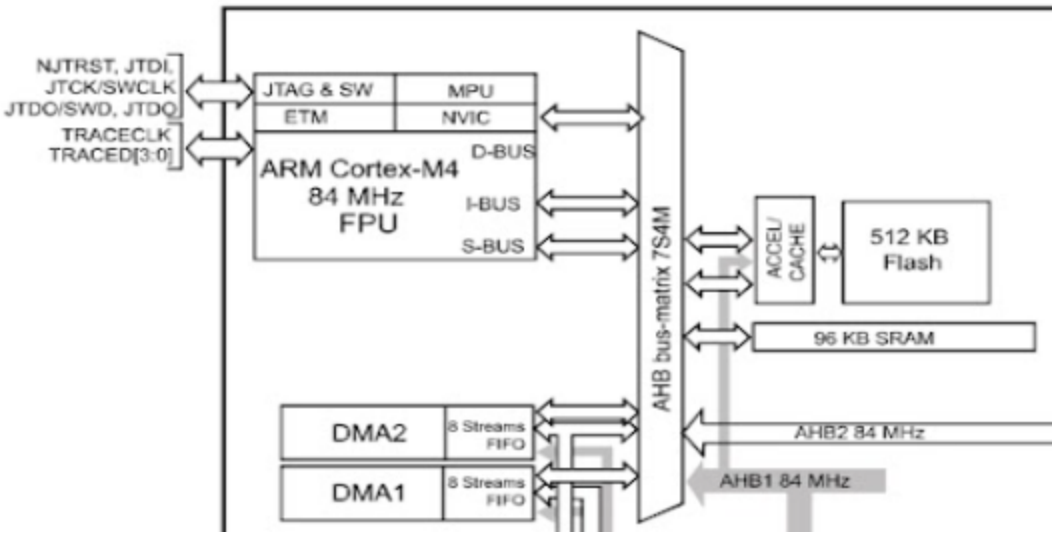
- This region includes additional system control and debug components.

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. Booting Process in the ARM Cortex-M4

Reset :
There are three types of reset

- Power Reset
- System Reset
- Backup domain Reset

A system reset sets all registers to their reset values except the reset flags in the clock controller CSR register and the registers in the Backup domain. A system reset is generated when one of the following events occurs:

1. A low level on the NRST pin (external reset)
2. Window watchdog end of count condition (WWDG reset)
3. Independent watchdog end of count condition (IWDG reset)
4. A software reset (SW reset)
5. Low-power management reset

- If booting from system memory (bootloader mode), the bootloader executes and can perform operations such as programming the Flash memory or other boot-related tasks.
- Once the bootloader completes its tasks, it typically jumps to the application code in Flash memory if such functionality is implemented in the bootloader.

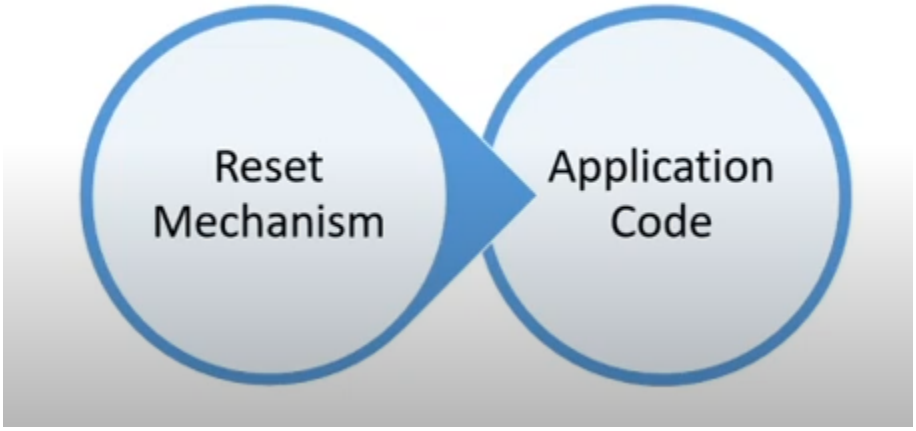
Low-power management reset
There are two ways of generating a low-power management reset:

- Reset generated when entering the Standby mode:
    - This type of reset is enabled by resetting the nRST_STDBY bit in the user option bytes. In this case, whenever a Standby mode entry sequence is successfully executed, the device is reset instead of entering the Standby mode.
- Reset when entering the Stop mode:
    - This type of reset is enabled by resetting the nRST_STOP bit in the user option bytes. In this case, whenever a Stop mode entry sequence is successfully executed, the device is reset instead of entering the Stop mode

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)

Without Bootloader



2.1.1. Power on/Power down Reset (POR/PDR reset)

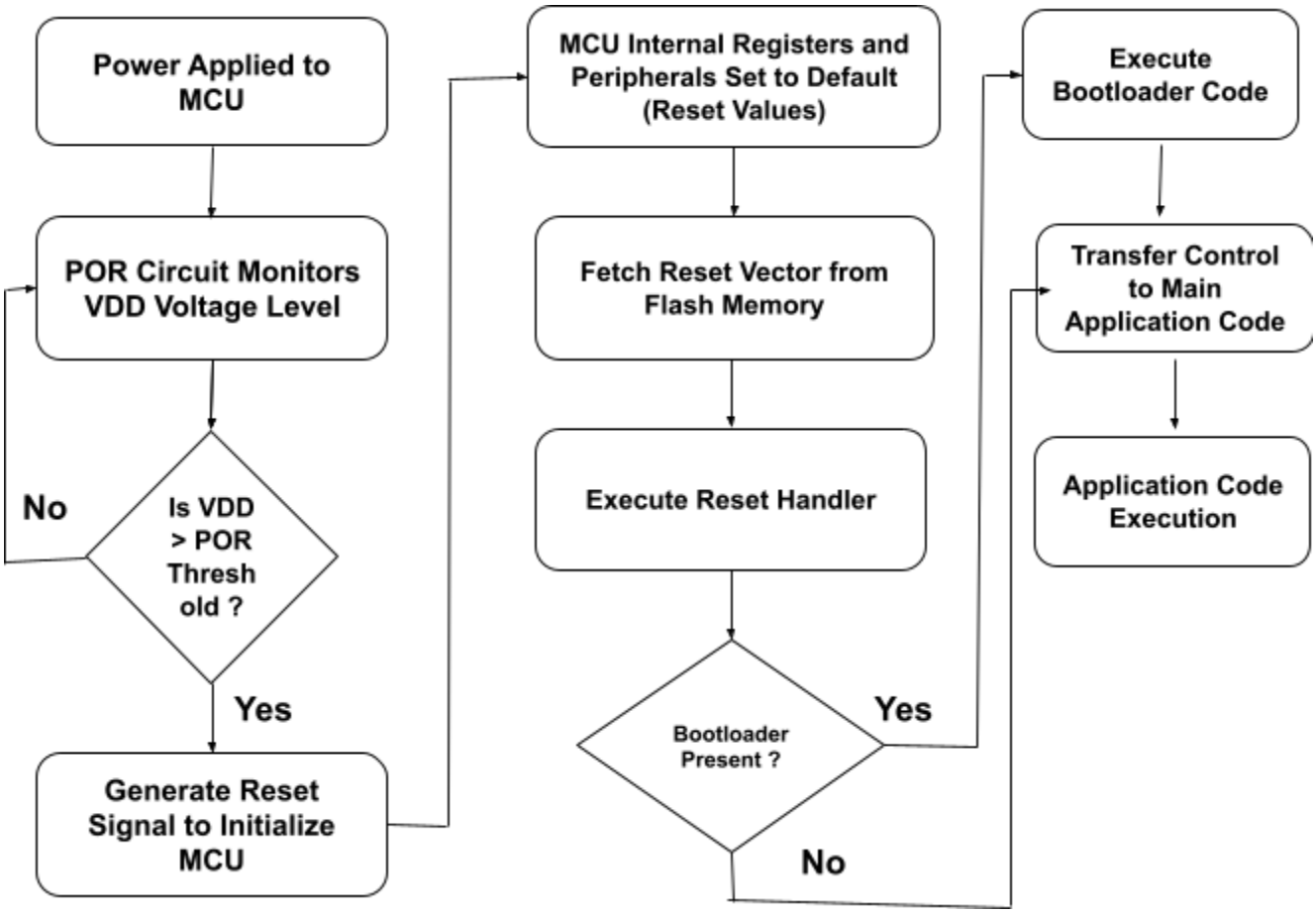
Power-On Reset (POR)

POR occurs when the microcontroller is powered on. This reset ensures that the microcontroller starts in a clean state and initializes correctly. The main purpose of a POR is to reset the entire system, including all registers, peripherals, and memory, so that the microcontroller can begin execution from the reset vector.

Power-Down Reset (PDR)

PDR occurs when the supply voltage drops below a certain threshold. This reset prevents the microcontroller from operating incorrectly due to insufficient voltage, which could cause erratic behaviour, corrupted memory, or other unpredictable actions.

2.1.1.1. Flowchart

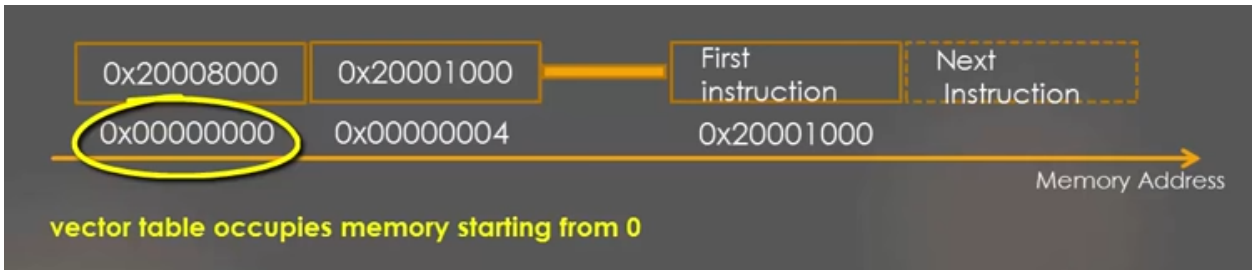


2.1.1.2. Reset Triggered

- 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. The reason for this is that the CPU's instruction fetch (ICode bus) always starts from this base 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.
- The MEM_MODE bits in SYSCFG_MEMRMP are initially set based on the state of the BOOT pins

- **Boot from Flash memory (BOOT0 = 0):**
  - **Memory Mapping :**
    - This is the default mode for most applications.
    - When BOOT0 is set to 0, MEM_MODE will be 00, mapping Main Flash memory to 0x0000 0000.

        - 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.
    - MEM_MODE will be 01, mapping System Flash memory to 0x0000 0000.
    - 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.

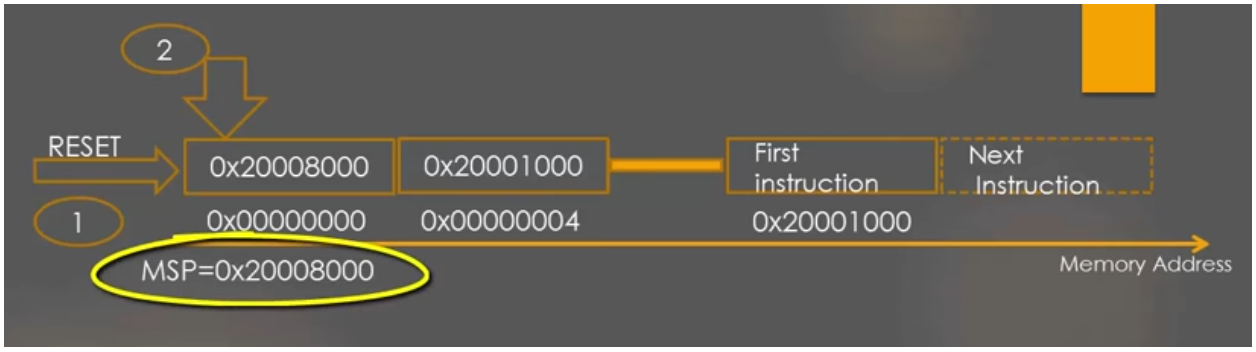
- **Boot from SRAM :**
    - SRAM is not used for booting because it is volatile and not intended for storing program code. SRAM is used for runtime data and stack, not for storing and executing application code.

- **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.



1→ After reset, PC is loaded with the address 0x00000000

2→ Processor reads the value from the 0x00000000 location into MSP.

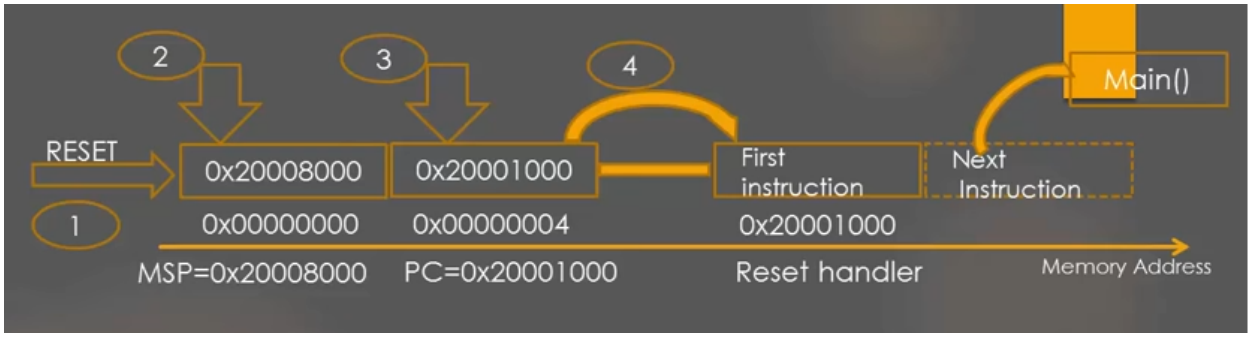
- **Stack in SRAM :**
    - The stack pointer (SP) register holds the current value of the stack pointer.
    - By starting the stack at the top of SRAM and growing downward, the stack efficiently uses available memory. As functions are called and local variables are pushed onto the stack, it moves towards 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



3 → Then the processor reads the address of the reset handler from the location 0x00000004

4 → Then it jumps to reset the handler and start executing the instructions.

5 → Then you can call your main() function from the reset handler.

- 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.

2.4. 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():

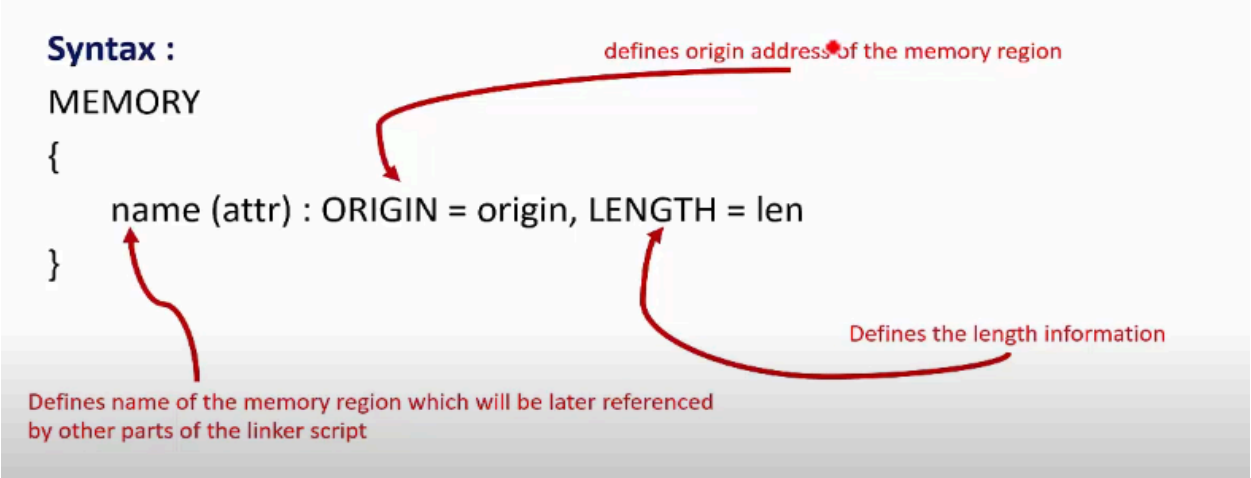
- **Startup File:** This file, typically named startup_stm32f401xe.s or startup_stm32f4xx.s, contains the vector table and the Reset_Handler() function. It is responsible for initializing the microcontroller when a reset occurs.
- **Vector Table:** The vector table, located at the beginning of Flash 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.

2.4.3. Linker Script

- Linker script is a text file which explains how different sections of the object files should be merged to create an output file.
- Linker script also includes the code and data memory address and size information.
- Linker scripts are written using the GNU linker command language.
- GNU linker script has the file extension of .ld

Linker scripts commands

- **ENTRY** :
    - This command is used to set the Entry point address information in the header of the final elf file generated.
    - In our case “Reset_Handler” is the entry point into the application. The first piece of code that executes right after the processor reset.
    - The debugger uses this information to locate the first function to execute.
    - Not a mandatory command to use, but required when you debug the elf file using the debugger (GDB).
    - Syntax : Entry(_symbol_name_)
    - Entry(Reset_Handler)
- **MEMORY** :
    - This command allows you to describe the different memories present in the target and their start address and size information.
    - The linker uses information mentioned in this command to assign addresses to merged sections.
    - The information given under this command also helps the linker to calculate total code and data memory consumed so far and throw an error message if data, code, heap or stack areas cannot fit into available size.
    - Typically one linker script has one memory command.



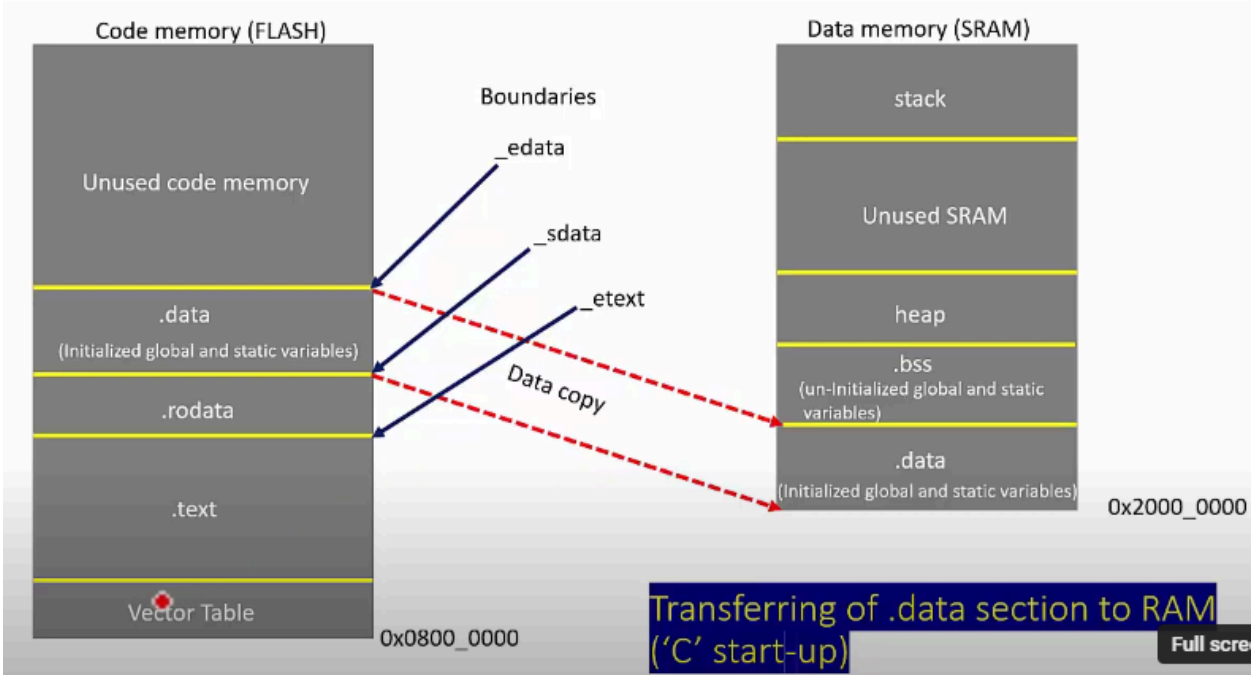
Attributes :

Attribute letter	Meaning
R	Read-only sections
W	Read and write sections
X	Sections containing executable code.
A	Allocated sections
I	Initialized sections.
L	Same as 'I'
!	Invert the sense of any of the following attributes.

- **SECTIONS**
  - SECTIONS command is used to create different output sections in the final elf executable generated.
  - Important command by which you can instruct the linker how to merge the input sections to yield an output section.

- This command also controls the order in which different output sections appear in the elf file generated.
- By using command , you also mention the placement of an action in a memory region. For example, you instruct the linker to place the .text section in the FLASH memory region, which is described by the memory command.

- **Location Counter (.)**
  - This is a special linker symbol denoted by a dot.'
  - This symbol is called "location counter" since linker automatically updates this symbol with location information.
  - You can use this symbol inside the linker script to track and define boundaries of various sections.
  - You can also set the location counter to any specific value while writing a linker script.
  - Location counter should appear only inside the sections commands
  - The location counter is then incremented by the size of the output section



- One for debugging in RAM.
- One for running from Flash.

2.4.3.1. Debug Mode Linker Script (Code in RAM)

1. **Memory Sections:**
    - RAM: Both code and data are placed in RAM. This is advantageous during debugging because RAM is faster.
    - FLASH: Typically not used for storing the program in this configuration.
2. **Sections:**
    - .isr_vector: Startup code, including the interrupt vector table, is placed in RAM.
    - .text: Contains the program code, placed in RAM.
    - .rodata: Read-only data, placed in RAM.
    - .data: Initialized global and static variables, placed in RAM.
    - .bss: Uninitialized global and static variables, placed in RAM.

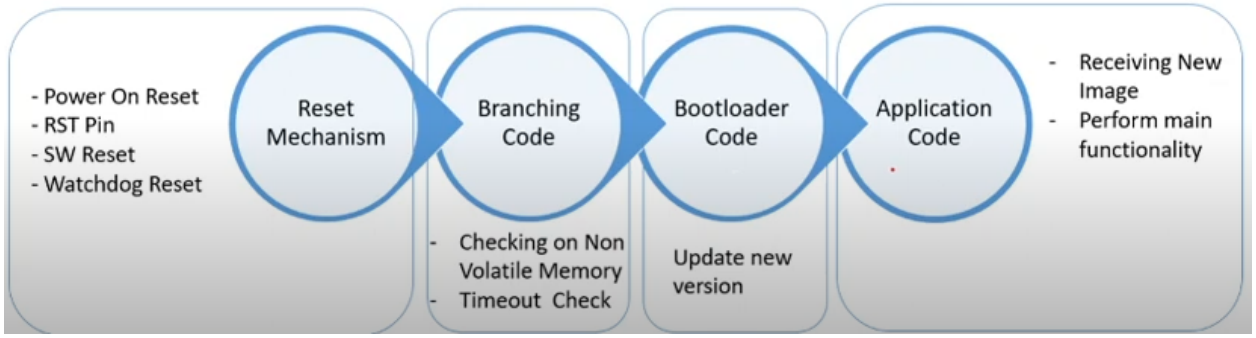
2.4.3.2. Release Mode Linker Script (Code in Flash)

1. **Memory Sections**:
    - **RAM**: Used for data sections such as .data, .bss, heap, and stack.
    - **FLASH**: Used for storing the program code and read-only data.
2. **Sections**:
    - **.isr_vector**: Startup code, including the interrupt vector table, is placed in flash.
    - **.text**: Program code, placed in flash.
    - **.rodata**: Read-only data, placed in flash.
    - **.data**: Initialized global and static variables, initially placed in flash but copied to RAM at startup.
    - **.bss**: Uninitialized global and static variables, placed in RAM.

2.5. Boot from System Memory

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

With Bootloader :



2.5.1. Why Use the Built-in Bootloader?

- If the application firmware becomes corrupted, the built-in bootloader provides a way to recover by reprogramming the flash memory.
- The built-in bootloader enables updating the firmware without requiring a debugger or development environment.
- Bootloader needed to upgrade Product's system when bugs are found.

Entering Bootloader Mode:

- **BOOT0 Pin**: To use the bootloader, you typically set the BOOT0 pin to a high voltage (e.g., VDD) while the microcontroller is powered on or reset. This configuration tells the microcontroller to execute the bootloader code in system memory instead of attempting to execute the potentially corrupted application code from flash memory.

Firmware Reprogramming:

- **Communication**: The bootloader can communicate with a host device (such as a PC) via supported communication interfaces like UART, USB, or CAN. The host device sends the new firmware image to the microcontroller.

- **Programming**: The bootloader receives the new firmware, verifies its integrity, and writes it into the flash memory of the microcontroller.
- **Verification**: After writing the new firmware, the bootloader might verify that the data was correctly programmed into flash memory to ensure the integrity of the new firmware.

Reset and Boot:

- **Final Reset**: After the new firmware has been successfully written and verified, you typically set the BOOT0 pin back to low (ground) and reset the microcontroller.
- **Execution**: The microcontroller will now boot from the newly programmed firmware in flash memory and execute it as usual.

2.5.2. 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 in a way that is not regular, performs incorrect operations, or crashes frequently.

Communication Failures:

- Loss of communication with peripheral devices or external systems.

Backup domain reset

The Backup Domain Reset in STM32 microcontrollers involves resetting all registers related to the Real-Time Clock (RTC) and backup domain.

This reset can be triggered either programmatically through the BDRST bit in the RCC_BDCR register or automatically by restoring power to VDD and VBAT after they have been completely off.

2.5.3. Uses of the Built-In Bootloader

1. **Firmware Update**:
    - **Bootloading**: The bootloader allows for in-system programming (ISP) and updating of firmware via various interfaces such as UART, USB, or CAN. This is useful for updating the firmware without needing external programming tools.
    - **Recovery**: It provides a method to recover from failed firmware updates or other issues that prevent the microcontroller from booting normally.
2. **Ease of Development**:
    - **Development Testing**: The built-in bootloader simplifies the development process by allowing firmware to be loaded and tested quickly. This reduces the need for external programming hardware during early stages of development.
    - **Initial Firmware**: It enables the initial loading of firmware into the microcontroller, especially useful when the system is first powered up or when developing new applications.
3. **Security**:
    - **Protection**: Some STM32 bootloaders have features for verifying the integrity of the firmware being loaded, which can help protect against corrupt or malicious code.
    - **Authentication**: In some configurations, the bootloader can include mechanisms for authenticating the firmware before execution.

2.6. 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.