Reset Mechanism and Reset Sequence

In typical Cortex-M microcontrollers, there are three types of reset mechanisms:

1. Power-On Reset
   - Resets the entire microcontroller, including the processor, debug support component, and peripherals.
2. System Reset
   - Resets only the processor and peripherals, excluding the debug support component.
3. Processor Reset
   - Resets the processor exclusively.

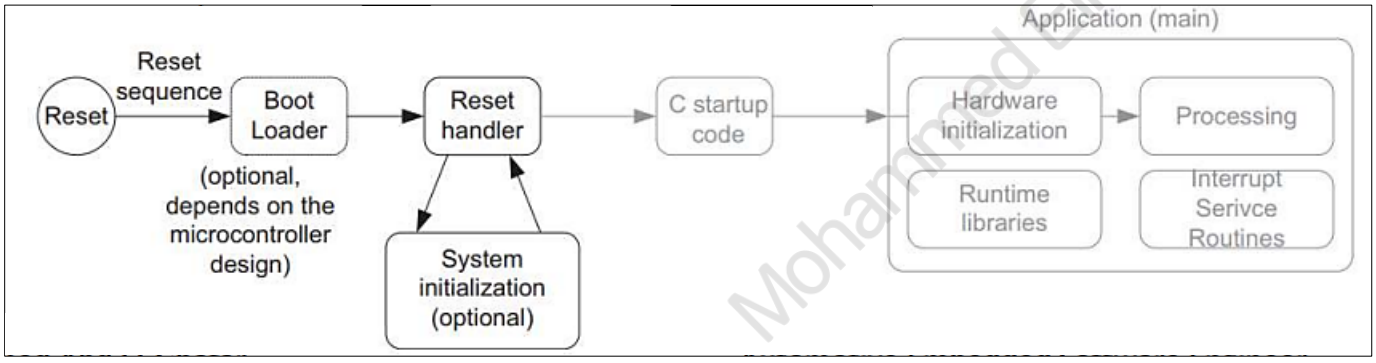
→ During system debug or processor reset operations, the debug components in Cortex-M3 or Cortex-M4 processors remain unaffected (are not reset) to maintain the connection between the debug host and the microcontroller.

→ The debug host can initiate a system reset or processor reset through a register in the System Control Block (SCB).

→ The duration of Power-On Reset and System Reset varies based on the microcontroller design. In some instances, the reset may last several milliseconds as the reset controller waits for a stable clock source like a crystal oscillator.

- **Initialization Process When a Microcontroller Starts**

  When a microcontroller starts, the following sequence of events occurs:
  - Most modern microcontrollers utilize on-chip flash memory to store the compiled program.
  - Some microcontrollers may have a separate boot ROM (It may be also in the same on-chip flash), which contains a boot loader program executed before the user program in the flash memory.
  - Typically, only the program code in the flash memory can be modified, while the boot loader code is fixed by the manufacturer.
  - After programming the flash memory with the code, the processor gains access to the program.
  - Upon reset, the processor executes the reset sequence.



Microcontroller Startup Sequence

During the microcontroller startup sequence:

- The C startup code is executed before entering the main application code.
- The startup code initializes the microcontroller, ensuring a proper startup for the processor:
  1. Disable the global interrupt to avoid being interrupted.
  2. Initializes the stack pointer.
  3. Initializes the .bss segment (SRAM) to “zeros”
  4. Copy the .data segment located in the (Flash) to the .data segment located in the (SRAM).
  5. Initialize the interrupt vector table by our ISR addresses.
  6. Enables the global interrupt again.
  7. Call the main () function

Further Initialization Steps

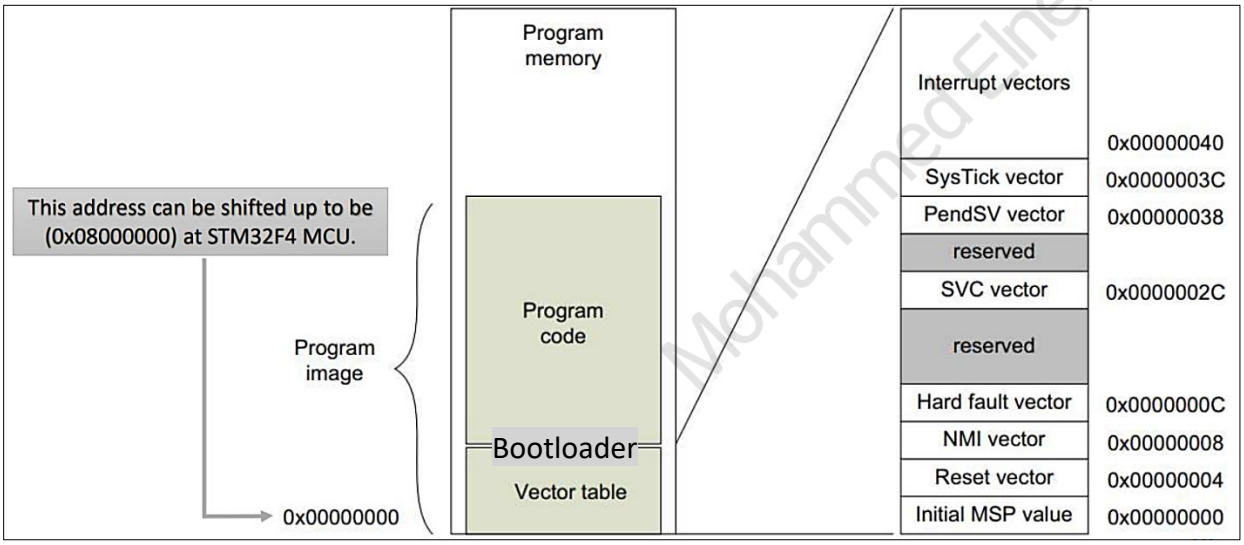
After jumping to the main (), we can perform:

→ The hardware initialization might involve a number of peripherals, some system control registers as well as interrupt control registers.

→ The initialization of the system clock control and the PLL might also take place if this was not carried out in the reset handler.

→ The initialization of the system clock control and the PLL might also take place if this was not carried out in the reset handler.

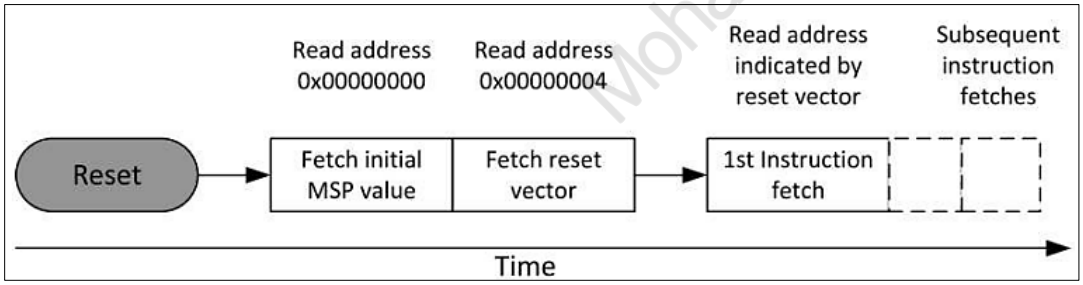
- بنستنتج ان بيكون عندنا في البروجيكت بتاعنا 2 applications والاتنين بيشتغلوا بس مش في نفس الوقت واللي هم
  - ✓ (main application and Bootloader)
  - ✓ الهدف الاساسي لل Bootloader هو عمل تحديث لل main application
  - ✓ وجود Bootloader بيكون مرحلة اختيارية تعتمد علي microcontroller وعند وجوده بيكون اول حاجة بتشتغل عندي في MCU
  - ✓ اول لما Bootloader بيشتغل بيروح يعمل check الاول لل App ويشوف هو Valid ولا لا
  - ✓ لو APP مش Valid ساعتها ال MCU بتاعنا بيفضل في حالة اسمها (stay in boot)
  - ✓ لو App طلع valid ساعتها بيحصل عملية Jump لتنفيذ اوامر application
- لما اجي اقول ان Bootloader او APP ان هو بدأ يشتغل ساعتها اي واحد منهم بيروح ينفذ startup code الخاص به
- ال startup code بيكون مكتوب بلغة السي او الاسمبلي وهو عبارة عن function مهمتها الاساسية ان هي بتنفذ sequence الخاص ب startup code
- واول function بيتم تنفيذها بداخل startup code هي Reset_Handler وفي نهاية startup code بيتم عمل call لل main function
- عادة بيكون مكان Bootloader في بداية Flash memory



- اول لما البروسيسور يجي يشتغل محتاج يعرف مكان Reset_Handler وبيعرف مكانها من vector table

Position	Priority	Type of priority	Acronym	Description	Address
	-	-	-	Reserved	0x0000 0000
	-3	fixed	Reset	Reset	0x0000 0004

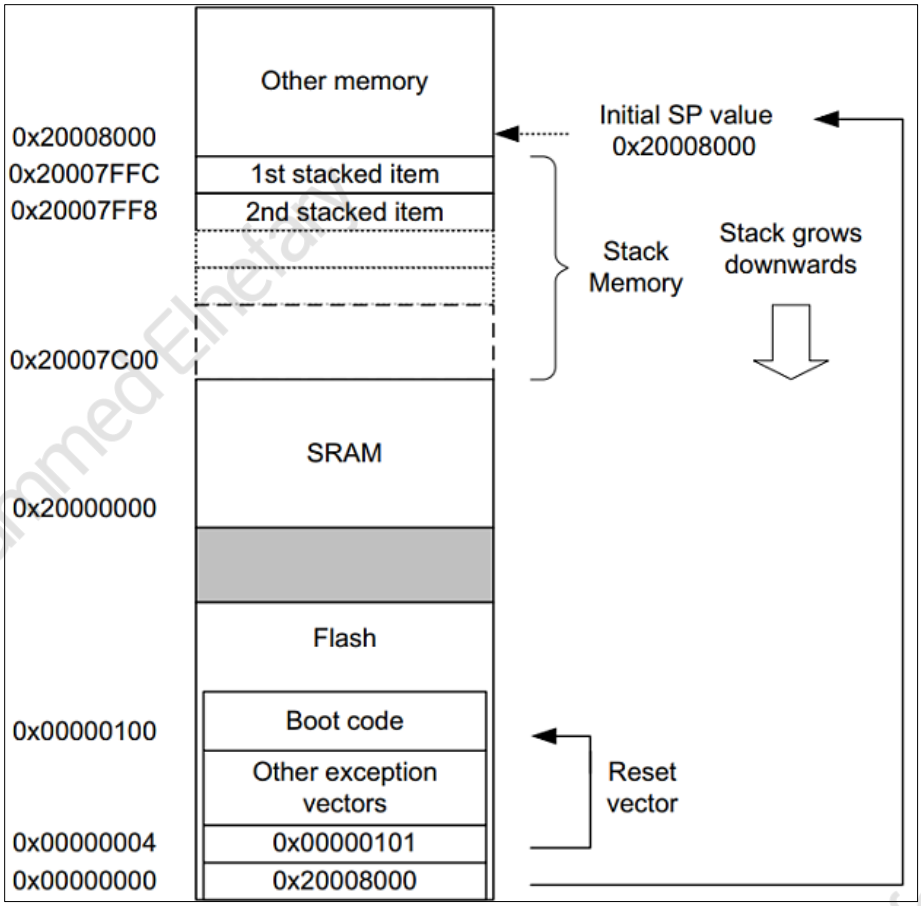
- **The Vector-Table**
  - → Normally, the program image for the Cortex-M based MCUs processors is located from address 0x00000000.
    - This address can be shifted up to be (0x08000000) at STM32F4 MCU.
  - → The beginning of the program image contains the vector table.
  - → It contains the starting addresses (Vectors) of exceptions.
  - → Each vector is located in address of "Exception_Number x 4."
  - → These vectors have LSB set to 1 to indicate that the exceptions handlers are to be executed with Thumb instructions.
  - → The size of the vector table depends on how many interrupts are implemented.
  - → After reset and before the processor starts executing the program, the Cortex-M processors read the first two words from the memory.
  - → The beginning of the memory space contains the vector table.
  - → The first two words in the vector table are the initial value for the Main Stack Pointer (MSP), and the reset vector, which is the starting address of the reset handler
  - → After these two words are read by the processor, the processor then sets up the MSP and the Program Counter (PC) with these values.
  - → The setup of the MSP is necessary because some exceptions such as the NMI or HardFault handler could potentially occur shortly after the reset and the stack memory and hence the MSP will then be needed to push some of the processor status to the stack before exception handling.
  - → Because the stack operations in the Cortex-M3 or Cortex-M4 processors are based on full descending stack (SP decrement before store), the initial SP value should be set to the "Last SRAM address".
  - → Example: If the SRAM starting from address 0x20000000 with length = 128K
    - The last memory location = 0x20000000 + 0x20000 = 0x20020000
    - SP = 0x20020000
  - → Notice that in the Cortex-M processors, vector addresses in the vector table should have their LSB set to 1 to indicate that they are Thumb code.

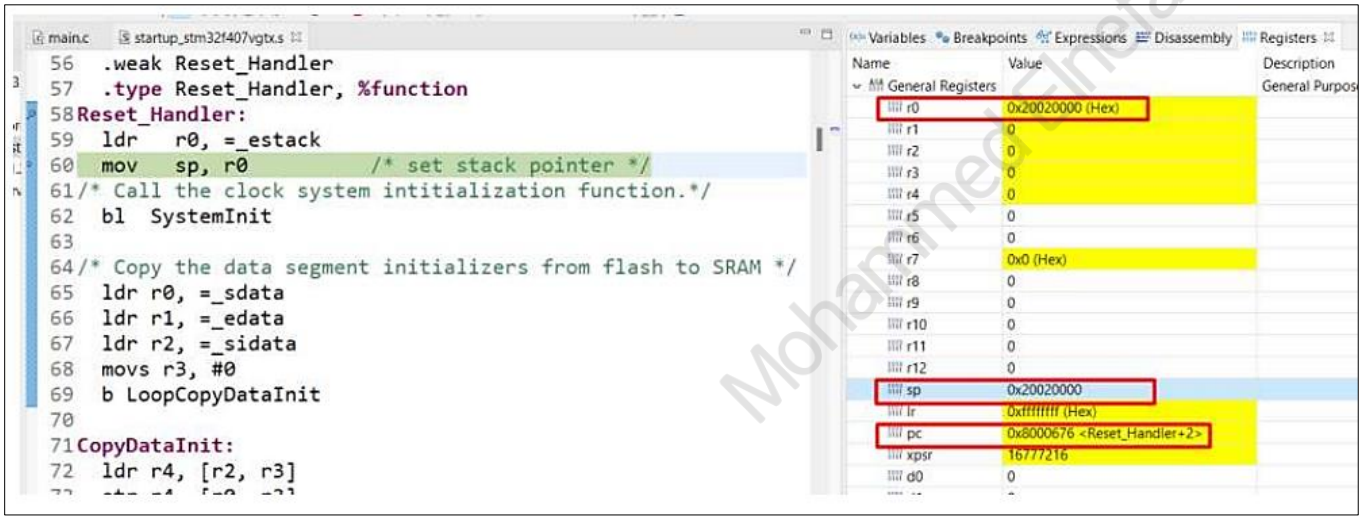



```
.text          0x08000188       0xb0 startup_stm32f407xx.o
               0x08000188                Reset_Handler
               0x0800022c                HASH_RNG_IRQHandler
```

Device Memory @ 0x08000000 : File : startup_stm32f407xx.bin
Target memory, Address range: [0x08000000 0x08000818]

Address	0	4	8	C
0x08000000	20020000	08000189	0800022D	080007F9
0x08000010	080007FF	08000805	0800022D	00000000





- Startup-code
  - ال startup code هو عبارة عن array of addresses او هو عبارة عن array of function pointers وكل عنوان بيمثل start address الخاص ب ISR معينة
  - بيحصل allocation لل startup code اللي همثله ب vector table في اول عنوان في flash memory بداية من 0x0 = address
  - واللي هايساعدنا نعمل عملية allocation الخاصة ب startup code في flash هو linker script

```
uint32_t Vector_Table[] __attribute__ ((section (".isr_vector"))) = {

};
```

- يتم تعريف new IS Section في اول linker script لان التخزين في memory بيكون بالترتيب الخاص بملف linker script

```
/* Sections */
SECTIONS{
  /* The startup code into "FLASH" Rom type memory */
  .isr_vector :
  {
    . = ALIGN(4);
    *(.isr_vector)
    . = ALIGN(4);
  } >FLASH
```

```
typedef unsigned int uint32_t;

extern uint32_t _estack, _etext, _sdata, _edata, _sbss, _ebss;

uint32_t * const MSP_Value = (uint32_t *)&_estack;

uint32_t *Vector_Table[] __attribute__ ((section (".isr_vector"))) = {
    (uint32_t *)MSP_Value
};
```

Extern from linker script

Store the address of stack to MSP

```
/* Vector table for a Cortex M4 */
uint32_t *Vector_Table[] __attribute__ ((section (".isr_vector"))) = {
    (uint32_t *)MSP_Value,
    (uint32_t *)&Reset_Handler,
    (uint32_t *)&NMI_Handler,
    (uint32_t *)&HardFault_Handler

};

void Reset_Handler(void){

}

void NMI_Handler(void){

}

void HardFault_Handler(void){

}
```

Vector table from datasheet

- واحنا بنعمل Build لل application بتاعنا واحنا بنستخدم startup code السابق هايطلعلي linker error لان مفيش definition اتعمل لل ISR الباقية

```
arm-none-eabi-ld: startup_stm32f407xx.o:(.isr_vector+0x10): undefined reference to `MemManage_Handler'
arm-none-eabi-ld: startup_stm32f407xx.o:(.isr_vector+0x14): undefined reference to `BusFault_Handler'
arm-none-eabi-ld: startup_stm32f407xx.o:(.isr_vector+0x18): undefined reference to `UsageFault_Handler'
arm-none-eabi-ld: startup_stm32f407xx.o:(.isr_vector+0x2c): undefined reference to `SVC_Handler'
```

- حاليا واحنا بنكتب الملف مش محتاجين نعمل definition لكل ISR اللي عندي وعلشان نحل المشكلة دي ممكن نعمل Default function ونخلي اي ISR ملهاش definition لو حصل ليها call تروح لل default handler وبنعمل كدة من خلال attribute يسمي alias

```
void Default_Handler(void){

}
```

```
void Reset_Handler                (void);
void NMI_Handler                  (void);
void HardFault_Handler            (void);
void MemManage_Handler            (void) __attribute__ ((alias ("Default_Handler")));
void BusFault_Handler             (void) __attribute__ ((alias ("Default_Handler")));
void UsageFault_Handler           (void) __attribute__ ((alias ("Default_Handler")));
void SVC_Handler                  (void) __attribute__ ((alias ("Default_Handler")));
void DebugMon_Handler             (void) __attribute__ ((alias ("Default_Handler")));
void PendSV_Handler               (void) __attribute__ ((alias ("Default_Handler")));
void SysTick_Handler              (void) __attribute__ ((alias ("Default_Handler")));
void WWDG_IRQHandler              (void) __attribute__ ((alias ("Default_Handler")));
void PVD_IRQHandler               (void) __attribute__ ((alias ("Default_Handler")));
```

These functions have a definition

Device Memory @ 0x08000000 : File : startup_stm32f407xx.bin

Target memory, Address range: [0x08000000 0x080001BC]

Address	0	4	8	C	ASCII
0x08000000	20020000	08000189	08000195	080001A1	... ‰...•...¡...
0x08000010	080001AD	080001AD	080001AD	00000000	-...-...-.......
0x08000020	00000000	00000000	00000000	080001AD	............-...
0x08000030	080001AD	00000000	080001AD	080001AD	-.......-...-...
0x08000040	080001AD	080001AD	080001AD	080001AD	-...-...-...-...
0x08000050	080001AD	080001AD	080001AD	080001AD	-...-...-...-...
0x08000060	080001AD	080001AD	080001AD	080001AD	-...-...-...-...
0x08000070	080001AD	080001AD	080001AD	080001AD	-...-...-...-...

- اول 4 اماكن في memory
- اول حاجة عنوان Stack اللي هايتخزن في MSP
- ال 3 عناوين اللي بعد كدة عناوين ISR اللي احنا عملنا لكل واحدة فيهم definition خاص بيها
- وباقي ISRs ليهم نفس العنوان وهو عنوان default handler
- ولواحتاجنا بعد كدة تعمل implementation لاي function هانحذف alias الخاص بيها وبعد كدة نعمل definition الخاص بيها
- ودلوقتي هانعمل implementation لل reset handler function وبيكون الهدف الاساسي منها 4 حاجات اساسية

1. Copy the .data segment located in the (Flash) to the .data segment located in the (SRAM)
2. Initializes the .bss segment (SRAM) to “zeros”
3. Call the system initialization function
4. Call the main function

- علشان نعمل اول خطوة محتاجين نحدد بداية ونهاية (.data section ➔ flash)
- في linker script file احنا كنا محددين _sdata and _edata ← two location counter
- ولو جينا نشوف عنوان two location counter في ملف map file هنلاقيهم بيبصوا علي بداية ونهاية (.data section) ولكن اللي موجودة في ram وليس ال flash

```
/* .data section, Initialized data sections into "RAM" Ram type memory */
.data :
{
  . = ALIGN(4);
  _sdata = .;        /* Create a global symbol at data start */
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */
  . = ALIGN(4);
  _edata = .;        /* Define a global symbol at data end */
} >RAM AT> FLASH
```

- وعلشان نحل المشكلة دي هانعمل location counter تاني بس هيبقي خارج اي section وهيبقي بعد .rodata في ملف linker script علشان نشوف نقطة نهاية .rodata واللي هاتكون في نفس الوقت نقطة بداية section اللي بعدها عالطول واللي احنا هانحدده وهو flash ➔ .data
- بس المرة دي مش هانستخدم dot location counter (.) ولكن هانستخدم function اسمها LOADADDR(section_name);
- ووظيفة function دي ان هي هاتشوف نهاية section اللي قبلها في linker script file اللي هو (.rodata) وبرضو في نفس الوقت بتعمل return لل LMA(load memory address) الخاص ب section_name اللي بين القوسين يعني هايرجع (flash ➔ .data section) ➔ LMA
- دائما location counter بيعمل track لل VMA الخاص ب ال section وليس LMA
- ولو عملنا location counter عادي في اخر (.rodata) ➔ section هايكون هو نفس العنوان اللي هاترجعه LOADADDR function مع مراعاة ترتيب تعريف كلا منهم
- وبعدين هانستخدم القيمة بتاعة return الخاص ب _sidata ➔ LOADADDR function ونعملها extern في ملف startup-cade

```
/* .rodata section, Constant data into "FLASH" Rom type memory */
.rodata :
{
  . = ALIGN(4);
  *(.rodata)         /* .rodata sections (constants, strings, etc.) */
  *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
  . = ALIGN(4);
  _erodata = .;
} >FLASH

_sidata = LOADADDR(.data);

/* .data section, Initialized data sections into "RAM" Ram type memory */
.data :
{
  . = ALIGN(4);
  _sdata = .;        /* Create a global symbol at data start */
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */
  . = ALIGN(4);
  _edata = .;        /* Define a global symbol at data end */
} >RAM AT> FLASH
```

The same address ➔ end of .rodata
And ➔ .data ➔LMA ➔ flash

.data ➔ RAM

```
*(.rodata*)
                0x080001ac                . = ALIGN (0x4)
                0x080001ac                _erodata = .
                0x080001ac                _sidata = LOADADDR (.data)
```

- وعلشان نعرف برضو حجم الداتا اللي هانقلها من .data segment located in the (Flash) to the .data segment located in the (SRAM) هانستخدم two location counters اللي احنا معرفينهم قبل كدة بداخل (_sdata and _edata) الفرق بين العنوانين هاينتج حجم الداتا اللي انا عايز انقلها ➔ .data section) size of data segment

```
/* 1) Copy the data segment initializers from flash to SRAM */
Section_Size = &_edata - &_sdata;  /* Length of .data segment */
MemSourceAddr = (uint32_t *)&_sidata;  /* Start address of .data sengement (LMA) -> Load Memory Address */
MemDestAddr = (uint32_t *)&_sdata;     /* Start address of .data sengement (VMA) -> Virtual Memory Address */

for(uint32_t MemCounter = 0; MemCounter < Section_Size; MemCounter++){
    *MemDestAddr++ = *MemSourceAddr++;
}
```

- وعلشان نعمل تاني خطوة محتاجين نحدد بداية ونهاية (.bss section)
- وبعدين هانكمل بنفس الخطوات

```
/* 2) Initialize the .bss segment with zero */
 Section_Size = &_ebss- &_sbss;         /* Length of .bss segment */
 MemDestAddr = (uint32_t *)&_sbss;      /* Start address of .bss sengement */

for(uint32_t MemCounter = 0; MemCounter < Section_Size; MemCounter++){
    *MemDestAddr++ = 0;
}
```

- وبعد كدة في الخطوة الثالثة هانعمل Call to the system initialization function والنعمل في function دي اي implementation محتاجينه قبل مانبدأ ننفذ main function زي مثلا تعريف clock الخاصة ب mcu واي configurations تانية احنا عايزنها

```
static void System_Intitialization(void){
    /* Clock intitialization */

}

void Reset_Handler(void){

    /* 3) Call the system intitialization function */
    System_Intitialization();
```

- واخر خطوة هي Call the main function

```
extern int main(void);

void Reset_Handler(void){
    /* 4) Call the main function */
    main();
```

- وممكن نزود خطوة احتياطية وهي exit routine علشان لو الكود خرج من main function مينفذش في garbage values وهتبقي عبارة عن infinite loop

```
/* 4- Exit Routine */
while(1);
```