Embedded Systems

Boot

Ver 1.0

Carlo Brandolese, William Fornaciari

AA 2020-2021

Boot

What is the «boot» of a system?

Several components that interact together

- A sequence of steps that bring a system from power on to its fully operational state

Complexity varies depending on the system

Involves several hardware/software components

- Hardware platform
- Micrcontroller / microprocessor
- BIOS
- Bootloader
- Operating system
- Application

Linux – A very complete example

POLITECNICO MILANO 1863

Hardware

1	**Power on** The system is given power
2	**Reset** The CPU / MCU is kept in reset until a stable state of the power supply is reached
3	**PLL Lock** The CPU / MCU start the oscillators and waits for the frequency output of the PLL is stable

BIOS

4	**POST** Executes from Flash or ROM and performs an initial system self-test
5	**Peripherals** Looks for other peripheral's BIOSs and executes them

5	**Memory, HD, ports** Finds available memory, available communication ports and detects HD settings
6	**Boot sequence** Detemines the boot sequence based on prior user settings
8	**Bootloader** Executes the system-specific bootloader

Bootloader

9	**Loading** Locates and loads the kernel at a fixed physical memory address
10	**Starts execution** Determines the starting address of the kernel and starts executing from that entry point

Linux – A very complete example

POLITECNICO MILANO 1863

Linux

11	**Modules** The Linux kernel load additional modules and sets the kernel services up (IPC, VM, …)
12	**Init** The kernel start the first process
13	**Services** Based on the runleve, init starts all the services
14	**Login** The init process runs the login process, allowing uses to access the system
15	**Shell** Upon succesful login, a shell is started and the user can interact with th system

Application

16	**User application** Finally the user application is started

Where it all begins

Despite the complexity of the process, one step is crucial

- Step 4: The BIOS starts executing
- It is the moment when
  - The hardware startup is finshed
  - The first firmware instruction is executed

Every processor-based system must undergo this step

- The process is the same for all systems
- Differ only with respect to the memories involved

Microcontroller boot sequence

We consider a system with

- A microcontroller
- A flash memory where the binary executable is stored
- A RAM memory where data will reside during execution

We will also assume that

- The binary contains the entire code to run
  - May be the BIOS, a bare-metal application or an application linked with the OS
- The code will be executed directly from flash
  - The case where the code is executed from RAM (mostly in general purpose microprocessor-based systems or larger emebdded systems) is very similar

Microcontroller boot sequence

The flash memory contains «sections»

- TEXT: The executable code
- RODATA: Constants
- DATA: Initialized variables
- BSS: Non-initialized variables

Several binary formats

- Some formats actually define and describe the sections (e.g. ELF)
- Other formats are agnostic and rely on the software itself to identify the sections (e.g. HEX, BIN, S19, …)

Microcontroller boot sequence

The TEXT section contains three logiclly different types of information

Interrupt Vector Table

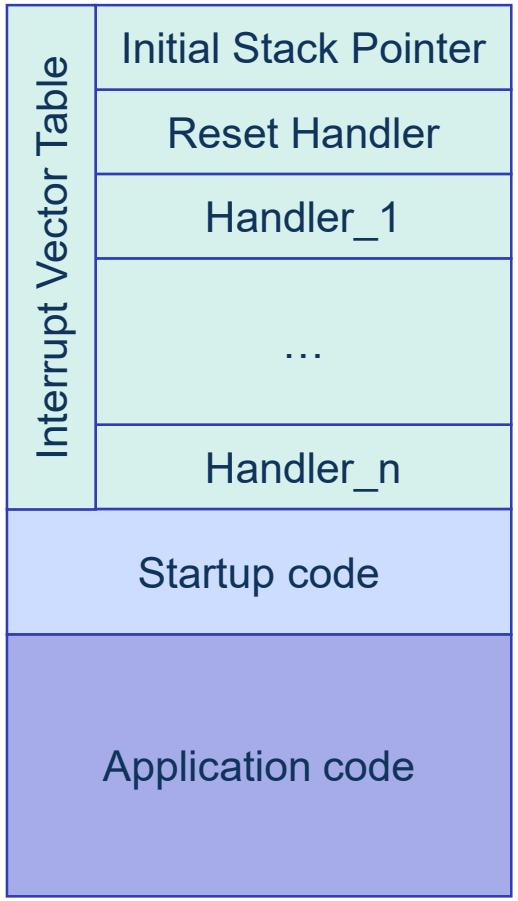
- A fixed-size table containing
  - The initial stack pointer
  - The address of the «reset handler»
  - The addresses of the interrupt service routines

Startup code

- A portion of code that will initialize the system
- Normally available from MCUs vendors
- Written in assembly

Application code

- The binary executable of the application



Microcontroller boot sequence

When the microcontroller exits from reset

- Loads the program counter with the address in the second entry of the interrupt vector table
- Starts executing the code from that address, i.e. executes the reset handler

More in general

- The base of the IVT may not be fixed but stored in a register of the MCU
  - ARM Cortex-M0: Address is fixed to 0x80000000
  - ARM Cortex-M4: Offset from 0x80000000 stored in register VTOR
- The position of the reset handler address in the table depends on the specifc MC
  - Being the second entry is, though, rather common

Microcontroller boot sequence

The reset handler start executing and performs a sequence of operations

Copies the initial stack pointer address into the SP

- Sometimens two stack pointers are used (SSP, MSP)
- From this moment on, the stack exists and functions can be called

Invokes a «system initialization» function

- Usually configures the clocks and waits for PLLs to become stable

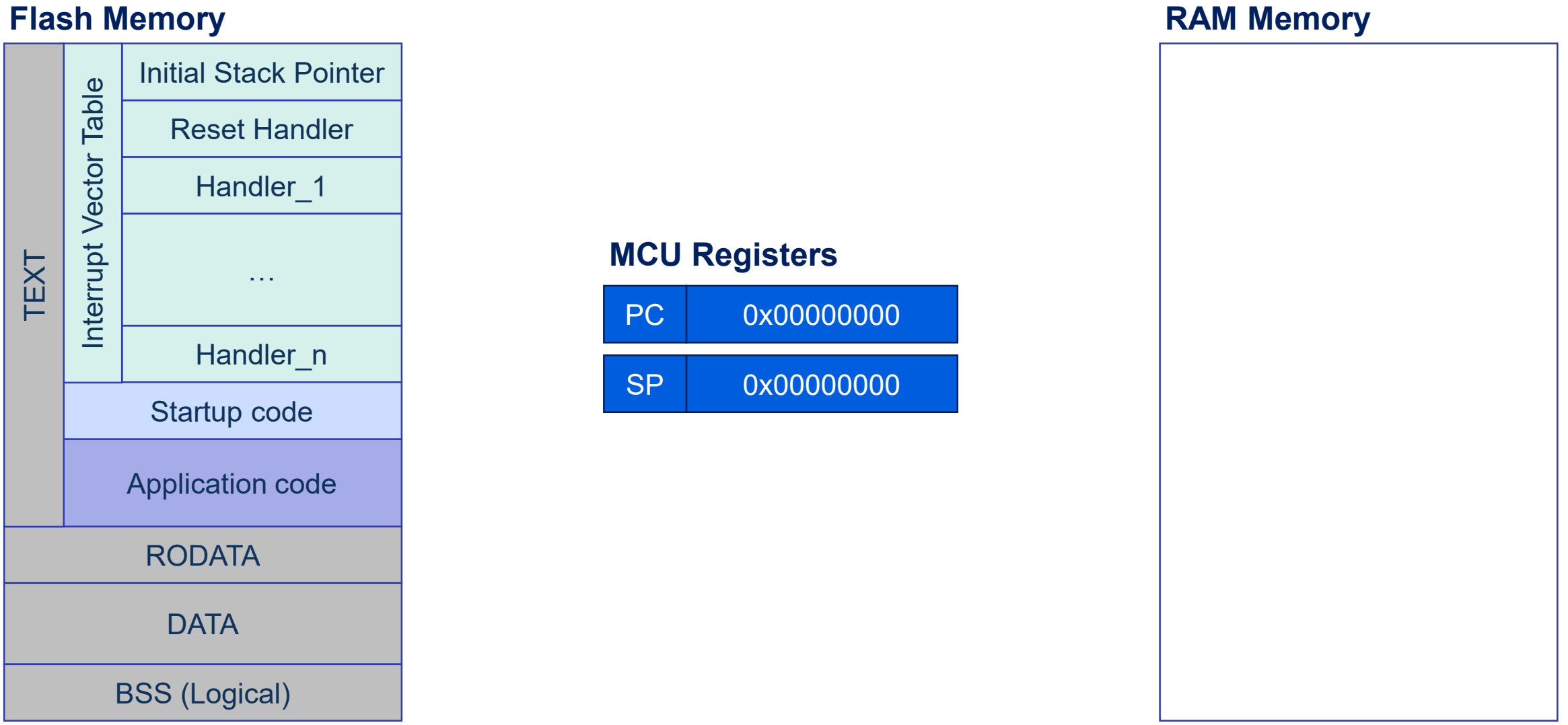
Prepares memory for execution

- Copies DATA section from flash to RAM according to linker script directives
- Zero-fills the area corresponding to the BSS section

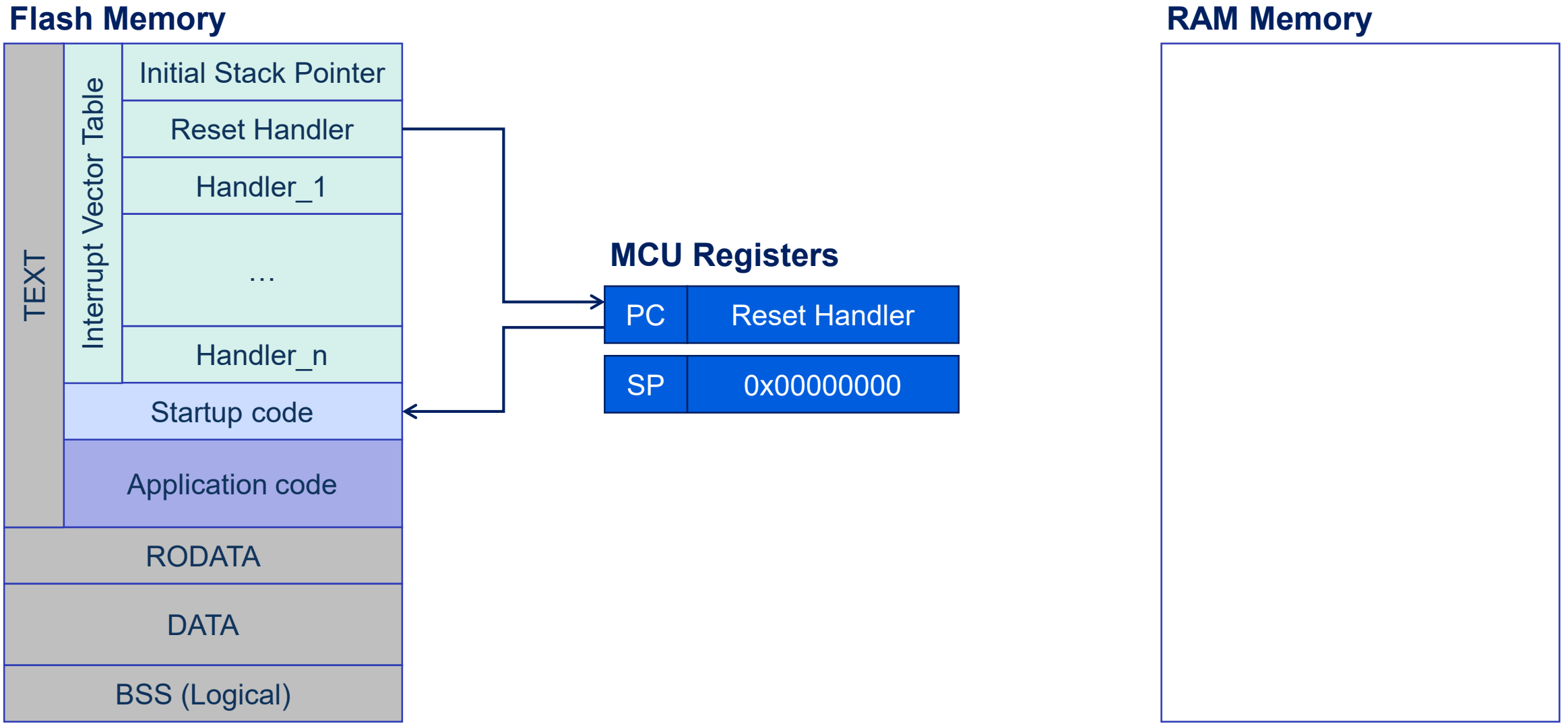
Jumps to main

- Here is the entry point of the application (or application plus OS)
- This is a jump, not a function call

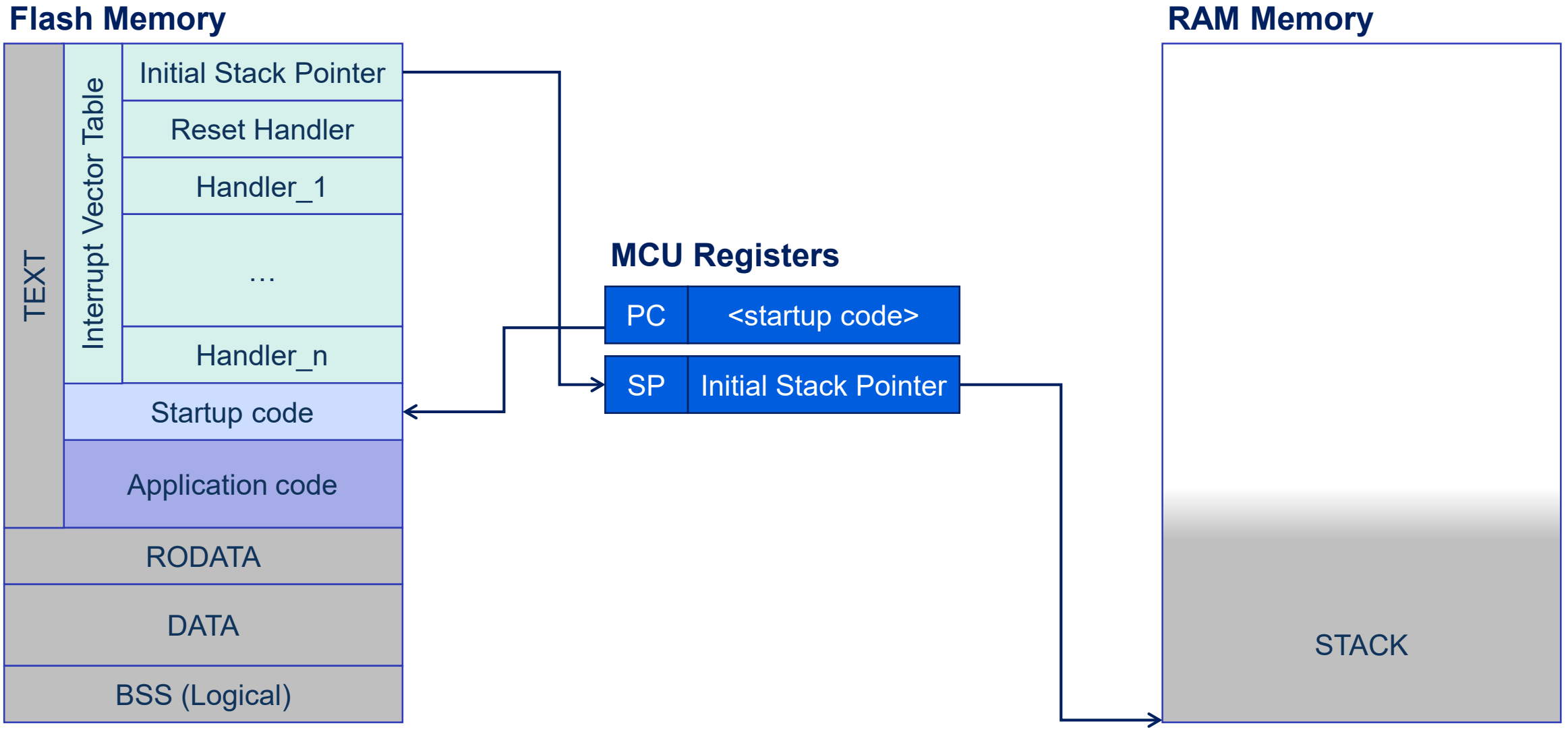
Microcontroller boot-sequence



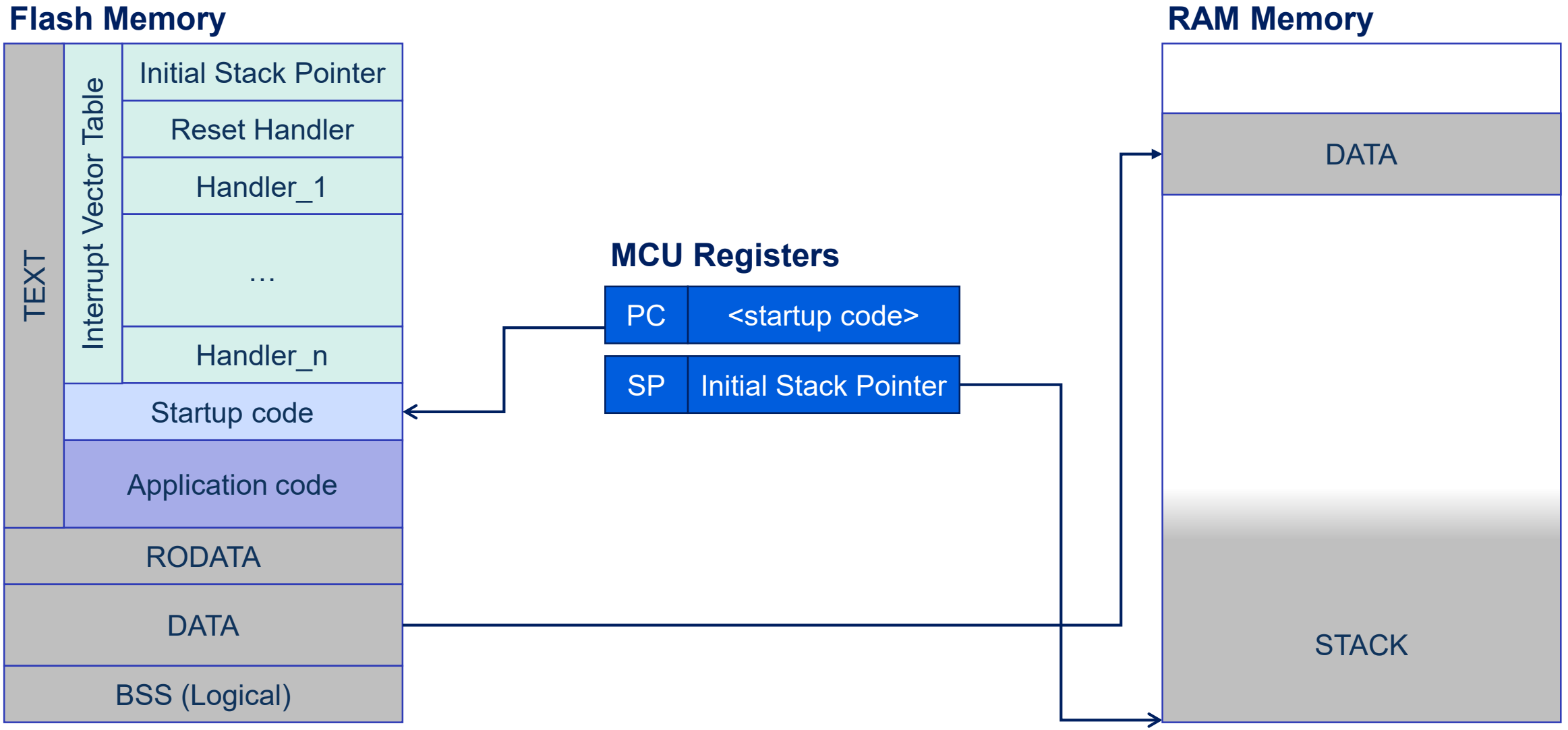
Microcontroller boot-sequence



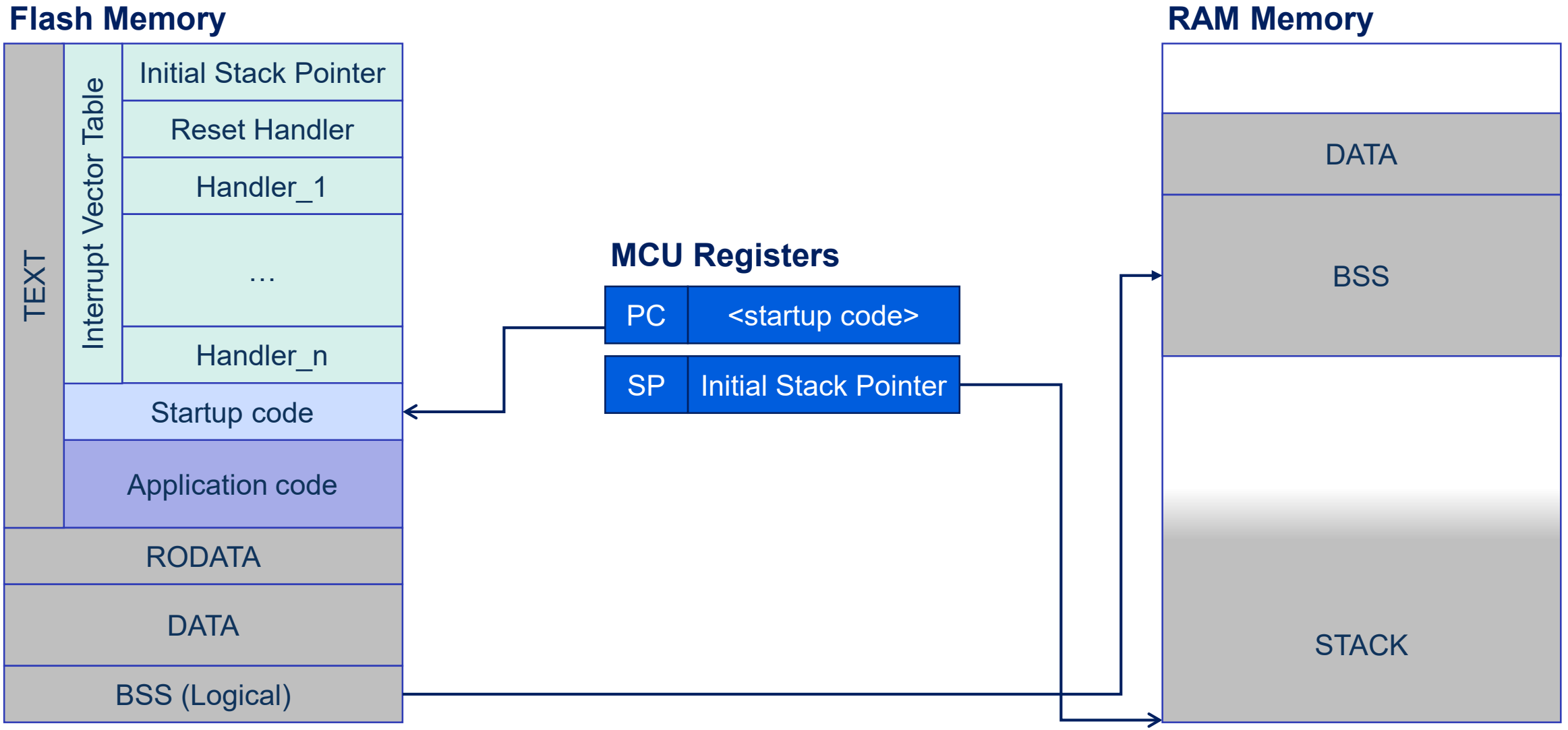
Microcontroller boot-sequence



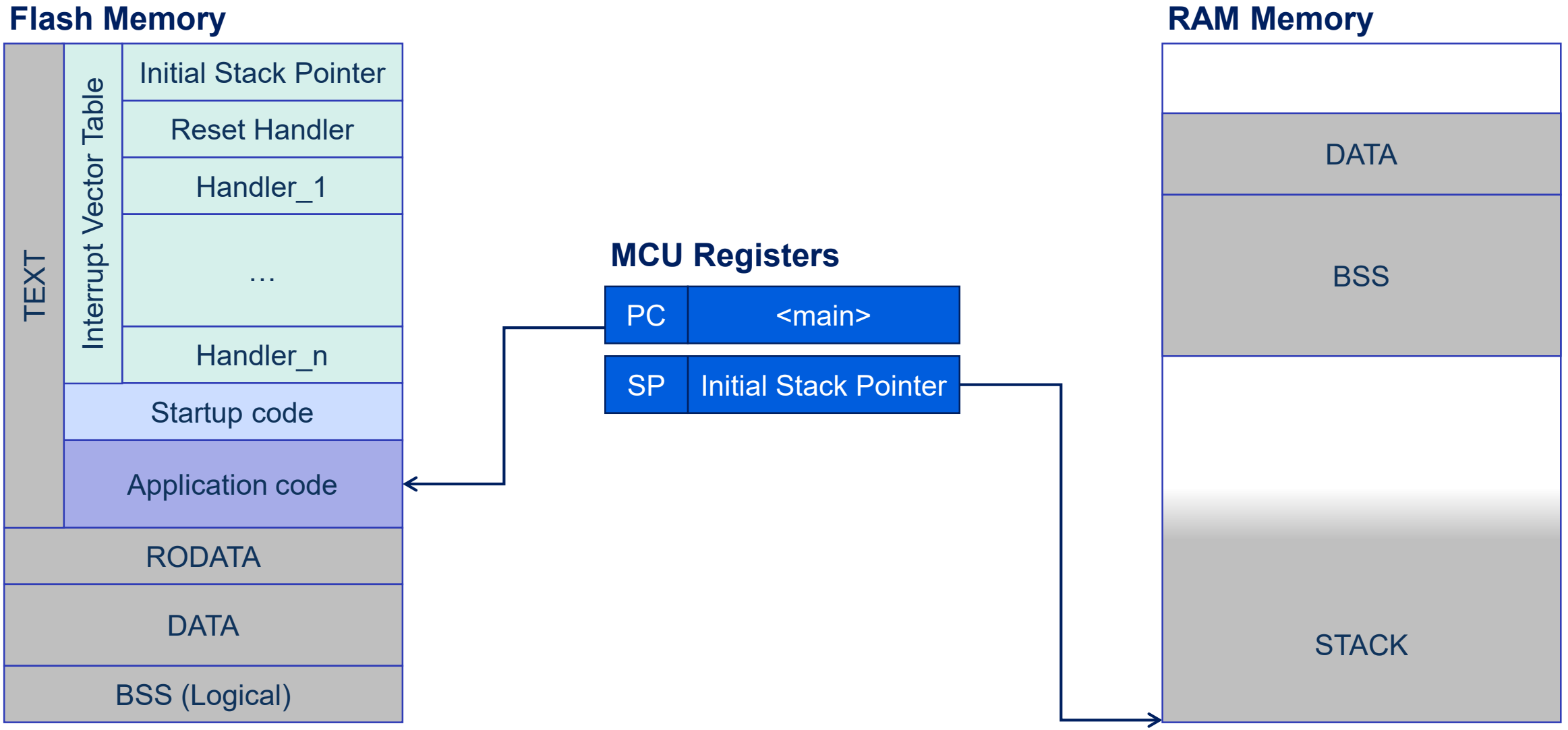
Microcontroller boot-sequence



Microcontroller boot-sequence



Microcontroller boot-sequence



STMicroelectronics startup code

STM32F0 ARM Cortex-M0

Vector table

```
Stack_Size      EQU     0x00000800
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp

                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors
                DCD     __initial_sp                ; Top of Stack
                DCD     Reset_Handler               ; Reset Handler
                DCD     NMI_Handler                 ; NMI Handler
                DCD     HardFault_Handler           ; Hard Fault Handler
                DCD     SVC_Handler                 ; SVCall Handler
                DCD     SysTick_Handler             ; SysTick Handler

                ; External Interrupts
                DCD     WWDG_IRQHandler             ; Window Watchdog
                DCD     OS_Tick_Handler             ; RTC_IRQHandler
                ...
                DCD     FLASH_IRQHandler            ; FLASH
__Vectors_End

__Vectors_Size  EQU  __Vectors_End - __Vectors
```

Initial stack address

Vector table base address

Stack pointer and reset handler

Service call

RTOS system tick handler(s)

Peripheral ISR

Vector table size

Reset handler

```
        AREA    |.text|, CODE, READONLY

Reset_Handler:
        PROC
        LDR     R0, = __initial_sp          ; set stack pointer
        MSR     MSP, R0
```

Initial stack address

DATA section

```
;  Single section scheme.
;
;  The ranges of copy from/to are specified by following symbols
;    __etext: LMA of start of the section to copy from. Usually end of text
;    __data_start__: VMA of start of the section to copy to
;    __data_end__: VMA of end of the section to copy to
;
;  All addresses must be aligned to 4 bytes boundary.

        LDR     R1, = __etext
        LDR     R2, = __data_start__
        LDR     R3, = __data_end__
        SUBS    R3, R2
        BLE     .L_loop1_done

.L_loop1:
        SUBS    R3, #4
        LDR     R0, [R1,R3]
        STR     R0, [R2,R3]
        BGT     .L_loop1

.L_loop1_done:
```

DATA section start in flash

DATA section start/end in RAM

Copies data

DATA section

```
;  Single BSS section scheme.
;
;  The BSS section is specified by following symbols
;    __bss_start__: start of the BSS section.
;    __bss_end__: end of the BSS section.
;
;  Both addresses must be aligned to 4 bytes boundary.

        LDR     R1, = __bss_start__
        LDR     R2, = __bss_end__

        MOVS    R0, 0
        SUBS    R2, R1
        BLE     .L_loop3_done

.L_loop3:
        SUBS    R2, #4
        STR     R0, [R1, R2]
        BGT     .L_loop3

.L_loop3_done:
```

BSS section start/end in RAM

Zeroes data

Application start

```
ApplicationStart
        LDR     R0, = RCC_Init
        BLX     R0
        LDR     R0, = __main
        BX      R0
        ENDP
```

Calls system init function

Calls main()

Infineon startup code

XMC4700 ARM Cortex-M4

Vector table

```
    .align 2
    .globl  __Vectors
    .type   __Vectors, %object

__Vectors:
    .long   __initial_sp                /* Top of Stack                 */
    .long   Reset_Handler               /* Reset Handler                */

    Entry   NMI_Handler                 /* NMI Handler                  */
    Entry   HardFault_Handler           /* Hard Fault Handler           */
    Entry   MemManage_Handler           /* MPU Fault Handler            */
    Entry   SVC_Handler                 /* SVCall Handler               */
    Entry   DebugMon_Handler            /* Debug Monitor Handler        */
    Entry   PendSV_Handler              /* PendSV Handler               */
    Entry   SysTick_Handler             /* SysTick Handler              */

    /* Interrupt Handlers for Service Requests (SR) from XMC4700 Peripherals */
    Entry   SCU_0_IRQHandler            /* Handler name for SR SCU_0    */
    Entry   ERU0_0_IRQHandler           /* Handler name for SR ERU0_0   */
    Entry   ERU0_1_IRQHandler           /* Handler name for SR ERU0_1   */
    Entry   VADC0_C0_0_IRQHandler       /* Handler name for SR VADC0_C0_0 */
    ...
    Entry   VADC0_C0_1_IRQHandler       /* Handler name for SR VADC0_C0_1 */
    Entry   VADC0_C0_2_IRQHandler       /* Handler name for SR VADC0_C0_1 */
    Entry   VADC0_C0_3_IRQHandler       /* Handler name for SR VADC0_C0_3 */

    .size   __Vectors, . - __Vectors
```

Vector table base address

Stack pointer and reset handler

Service call

Debugger support

RTOS system tick handler

Peripheral ISR

Vector table size

Reset handler

```
/* Reset Handler */

    .globl  Reset_Handler
    .type   Reset_Handler, %function

Reset_Handler:
    ldr  sp, = __initial_sp
    ldr  r0, = SystemInit
    blx  r0

/* Default exception Handlers - Users may override this default
 * functionality by defining handlers of the same name in their
 * C source code
 */

    .weak Default_Handler
    .type Default_Handler, %function

Default_Handler:
    b    .
```

Initializes the stack

Calls the system init function

Default handler

DATA section

```
/* Initialize DATA section
 *  Between symbol address __copy_table_start__ and __copy_table_end__,
 *  there are array of triplets, each of which specify:
 *    offset 0: LMA of start of a section to copy from
 *    offset 4: VMA of start of a section to copy to
 *    offset 8: size of the section to copy. Must be multiply of 4
 *  All addresses must be aligned to 4 bytes boundary.
 */
    ldr   r4, = __copy_table_start__
    ldr   r5, = __copy_table_end__
.L_loop0:
    cmp   r4, r5
    bge   .L_loop0_done
    ldr   r1, [r4]
    ldr   r2, [r4, #4]
    ldr   r3, [r4, #8]

.L_loop0_0:
    subs  r3, #4
    ittt  ge
    ldrge r0, [r1, r3]
    strge r0, [r2, r3]
    bge   .L_loop0_0
    adds  r4, #12
    b     .L_loop0

.L_loop0_done:
```

For each sub-section in DATA

Copies the sub-section

BSS section

```
/* Zero initialized data (BSS)
 *  Between symbol address __zero_table_start__ and __zero_table_end__,
 *  there are array of tuples specifying:
 *    offset 0: Start of a BSS section
 *    offset 4: Size of this BSS section. Must be multiply of 4
*/
          ldr       r3, = __zero_table_start__
          ldr       r4, = __zero_table_end__

.L_loop2:
          cmp       r3, r4
          bge       .L_loop2_done
          ldr       r1, [r3]
          ldr       r2, [r3, #4]
          movs      r0, 0

.L_loop2_0:
          subs      r2, #4
          itt       ge
          strge     r0, [r1, r2]
          bge       .L_loop2_0
          adds      r3, #8
          b         .L_loop2

.L_loop2_done:
```

For each sub-section in BSS

Zeroes the sub-section

Application start

```
__Start:
    ldr   r0, = main
    blx   r0

    b     .
```

Jumps to main()

Should never reach here

Bootloader

Simplified description

Bootloader

An application is loaded to the MCU flash with a programmer

- Requires dedicated hardware and software tools
- Can only be done in a controlled environment, e.g. laboratory, EoL

When in-field firmware update is required a bootloader is needed

A bootloader is a program with the following three main functionality

- Receive a firmware binary from some communication system
- Copy the new firmware in flash, replacing the old one
- Executing the new firmware

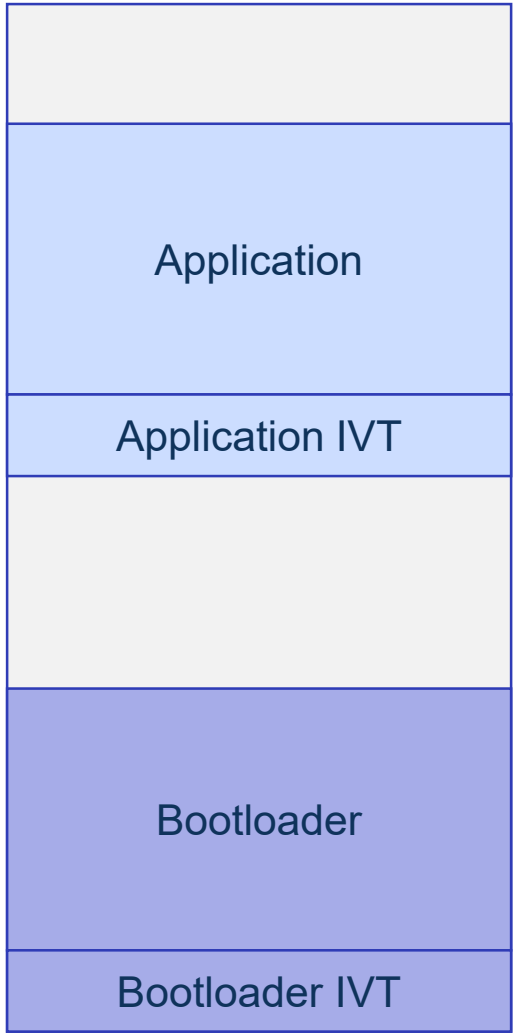
Bootloader

When a bootloader is needed

- It is the first software being executed
- Its vector table is located at the default address
- It has a dedicated flash area
- Coexists with the firmware
- It is a different binary

Critical aspects

- The bootloader and the application have different interrupt vector tables
- The bootloader must know how to execute the application



Bootloader

Suppose that the bootloader

- Has already received a new firmware
- Has copied the new firmware in the correct position
- Has verified that the new firmware is «correct», e.g. by means of CRC checking

The bootloader must now

- Switch to the correct IVT
- Start the execution of the firmware

The bootloader must now

- The MCU has a dedicated register for IVT offset
- The MCU does not have such a register

Bootloader – With IVT offset register

POLITECNICO MILANO 1863

In this case the operations to be performed are the following

Prepare the firmware execution

- Disable all interrupts and clear all pending interrupts, to avoid servicing and interrupt according to the new table while still executing the bootloader code on the bootloader stack
- Wait for all memory operation completion

Execute firmware

- Switch to the new IVT by assigning a new value to the IVT offset register
- Load the stack pointer with the value in the new IVT
- Jump to the reset handler address in the new IVT

Bootloader – Without IVT offset register

In this case the interrupt vector table

- May reside at two fixed addresses, one in Flash and one in RAM
- The bootloader execution assumes that its IVT is in flash

Prepare the firmware execution

- Disable all interrupt and clar pending ones
- Copy the new IVT from Flash to RAM

Execute firmware

- Switch to the new IVT by enabling memory «remapping»
- Load the stack pointer with the value in the new IVT
- Jump to the reset handler address in the new IVT