Fundamentals of Embedded and Real Time Systems

MODULE 08

TAMER AWAD

Review Module 07

Module 08

Boot Process
- Reset Button
- System Reset
- Boot Configuration
- Booting Process

Startup Code
- Before “main”
- Standard startup code
- Linker Map file
- Data initialization
- After Reset
- Vector Table
- Fault Handlers

The Vector Table
- The build process
- ELF files
- Injecting code at startup
- User-defined vector table

System Timer
- SysTick,
- Application
- Use in CMSIS

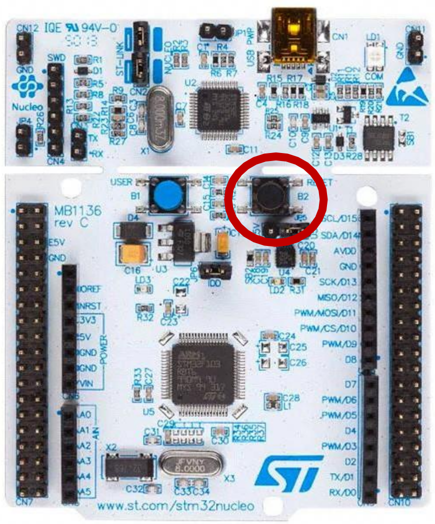
Interrupts Overview
- Polling
- CPU Interrupts
- Demo SysTick interrupt

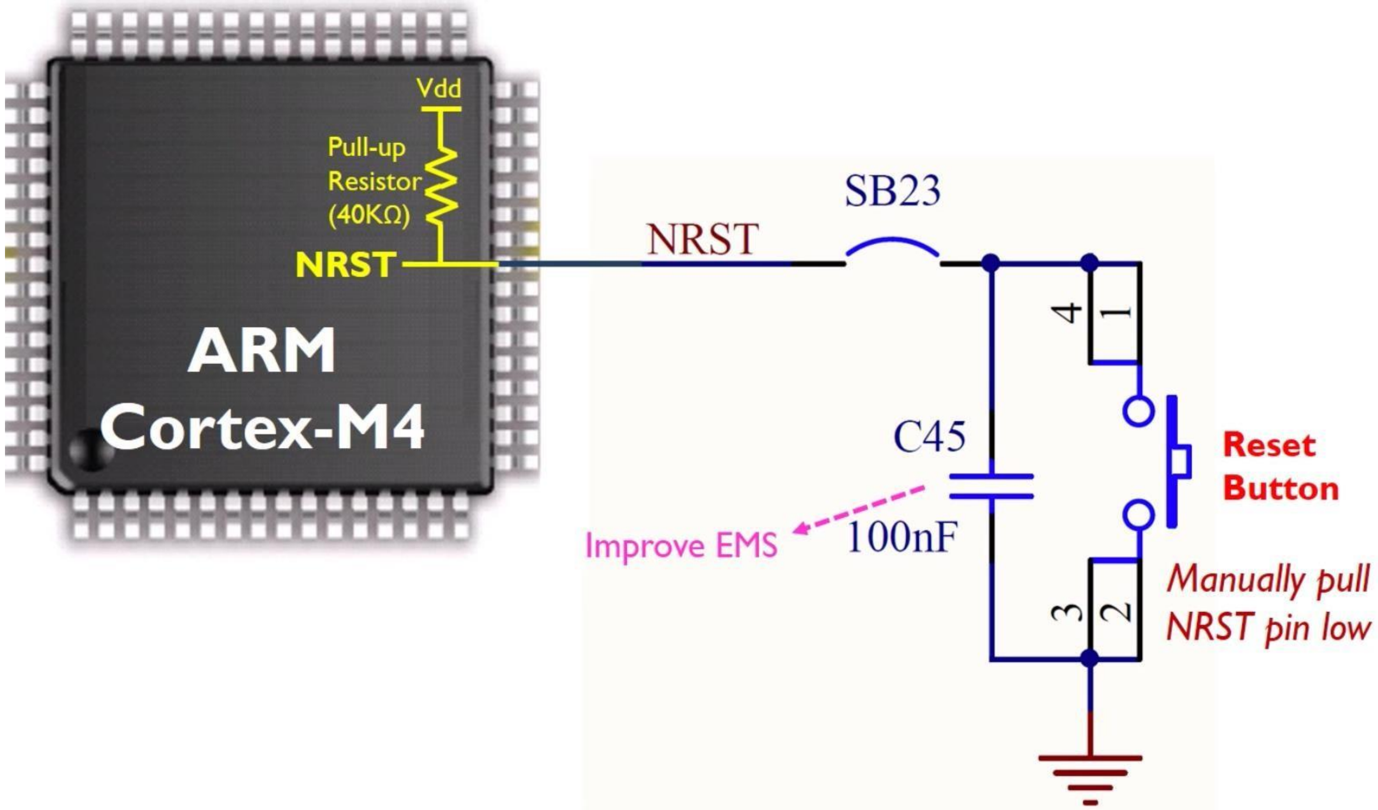
Assignment
- Assignment 07

Boot Process

- Reset Button
- System Reset
- After Reset
- Boot Configuration
- Booting Process

Reset Button





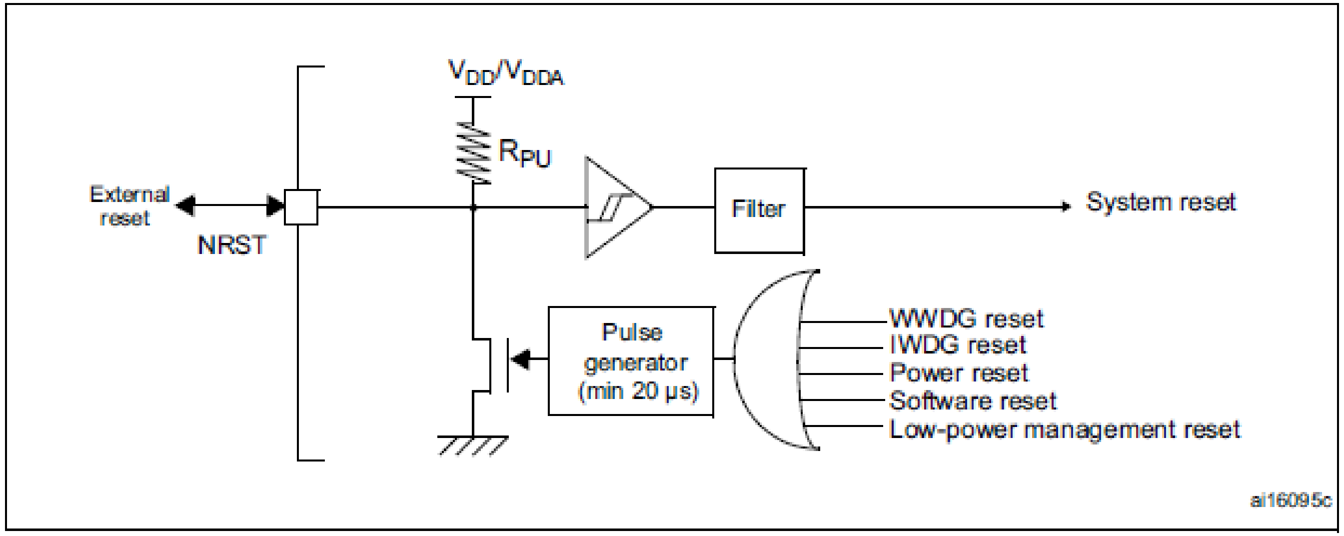
System Reset

A system reset can be generated by hardware or software

A system reset sets all registers to their reset values except for the reset flag in the clock controller register (CSR)

- Source: Reference Manual RM0368 (section 6)

Figure 11. Simplified diagram of the reset circuit

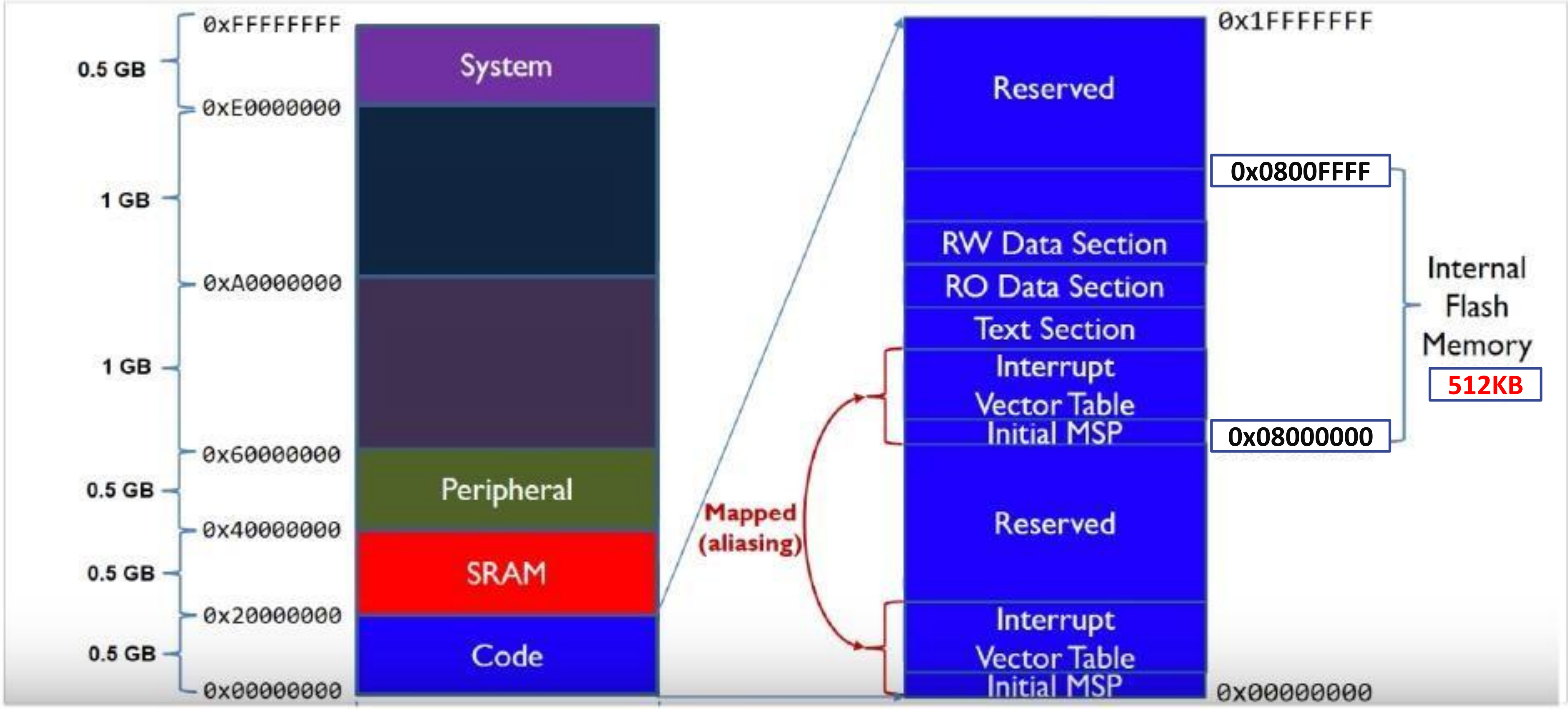


Boot Configuration

- STM32 parts are able to boot into different memory regions via BOOT pins
  - ***BOOT0***: Dedicated pin, by PCB design
  - ***BOOT1***: Shared with a GPIO pin
- Most Cortex processors support at least three different boot modes.
- The processor can boot from:
  - Main flash memory
  - System memory
    - Built in bootloader from manufacturer (protected against write and erase), which can be reprogrammed via USART. Used for developing a custom bootloader.
  - Main SRAM

Boot1	Boot0	Boot Mode
x	0	Boot from main flash memory
0	1	Boot from system memory
1	1	Boot from embedded SRAM

Booting Process



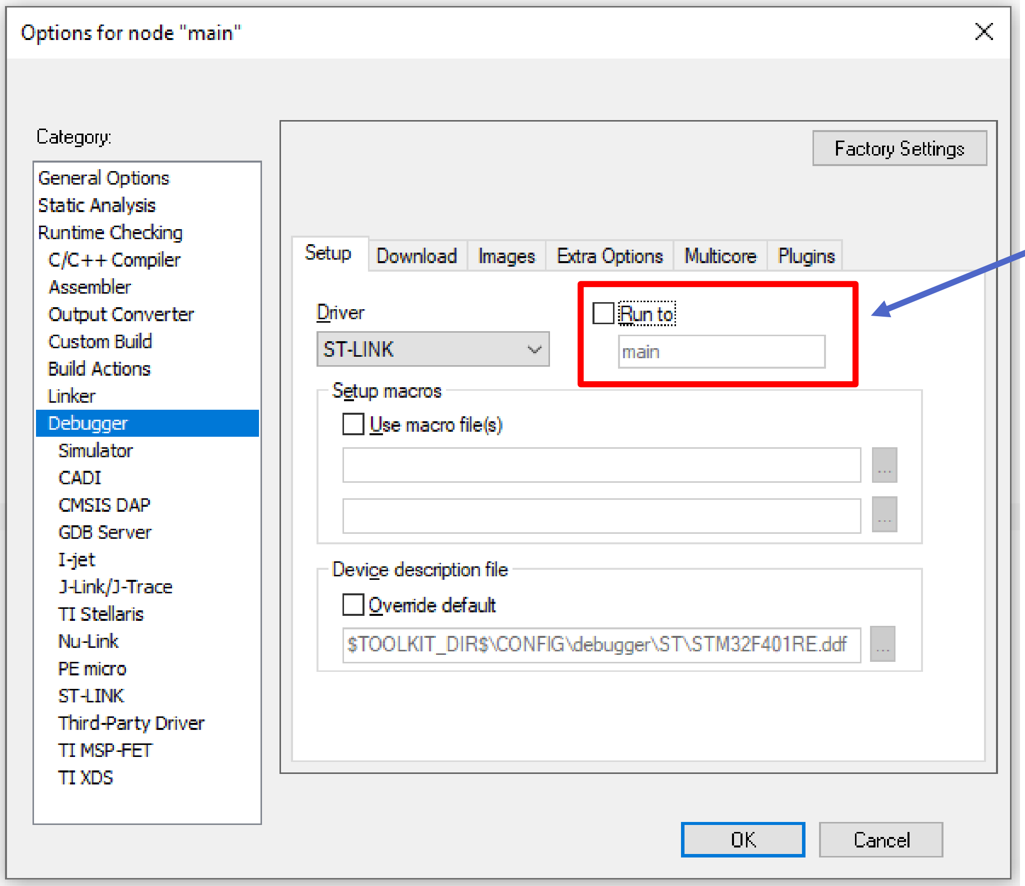
Startup Code

- Before “main”
- Standard startup code
- Linker Map file
- Data initialization
- After Reset
- Vector Table
- Fault Handlers

Demo: Startup Code

- Download Module08 Startup Code demo
- The standard startup code that gets linked with your application from the IAR library
- The various sections in the linker map
- Data initilization
- What happens after reset
- The vector table
- Fault Handlers

Before “main”



Uncheck box to see the world before main.

Standard startup code from IAR

```
__iar_program_start:
    0x800'01a4: 0xf3af 0x8000  NOP.W
    0x800'01a8: 0xf3af 0x8000  NOP.W
⇨   0x800'01ac: 0xf7ff 0xffd6  BL          ?main                  ...
```

```
__cmain:
⇨   0x800'015c: 0xf000 0xf80d  BL          __low_level_init   ...
    0x800'0160: 0x2800         CMP         R0, #0
    0x800'0162: 0xd001         BEQ.N       _call_main         ...
    0x800'0164: 0xf7ff 0xffde  BL          __iar_data_init3   ...
```

```
__low_level_init:
    0x800'017a: 0x2001         MOVS        R0, #1
⇨   0x800'017c: 0x4770         BX          LR
```

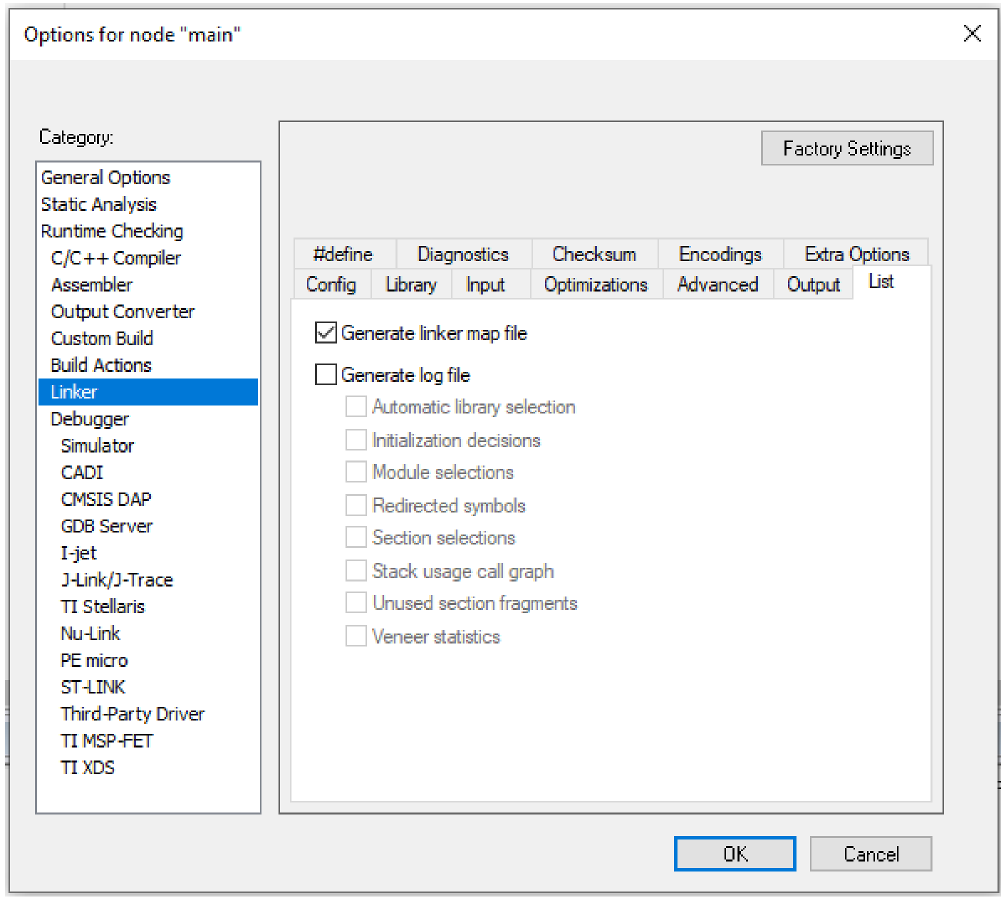
```
__cmain:
    0x800'015c: 0xf000 0xf80d  BL          __low_level_init   ...
    0x800'0160: 0x2800         CMP         R0, #0
    0x800'0162: 0xd001         BEQ.N       _call_main         ...
⇨   0x800'0164: 0xf7ff 0xffde  BL          __iar_data_init3   ...
```

```
_call_main:
    0x800'0168: 0xf3af 0x8000  NOP.W
    0x800'016c: 0x2000         MOVS        R0, #0
    0x800'016e: 0xf3af 0x8000  NOP.W
⇨   0x800'0172: 0xf7ff 0xff65  BL          main                   ...
```

Standard startup code from IAR

- __iar_program_start:
  - Start of code execution
- ?main :
  - This is an IAR specific startup code. IAR decided to call it “?main”.
- __low_level_init:
  - Function intended to perform a customized initialization of the hardware that must occur very early on.
- __iar_data_init3:
  - Function intended to initialize the program data per the C standard.

Linker Map File



Map file sections: MODULE SUMMARY

- Should always know how big your program is in terms of code space in ROM and data space in ROM and RAM. To find out, you scroll to the "Module Summary" section.
- Information broken down by read-only code, read-only data, and read-write data, as well as the object modules
- At the bottom of that section you find the total used.
- The largest contributor is the section “Linker created” which is generated by the linker for the stack.
    - EX: 8192 which is 0x2000, which is what was specified in the project options.

Map File Sections: "Module Summary"

```
*******************************************************************************
*** MODULE SUMMARY
***

    Module               ro code  ro data  rw data
    ------               -------  -------  -------
command line/config:
    -------------------------------------------
    Total:

C:\Users\tameraw\OneDrive\Documents\Education\UW_Embedded_Certi
    delay.o                   10
    main.o                   160                28
    -------------------------------------------
    Total:                   170                28

dl7M_tln.a: [2]
    exit.o                     4
    low_level_init.o           4
    -------------------------------------------
    Total:                     8

rt7M_tl.a: [3]
    cexit.o                   10
    cmain.o                   30
    cstartup_M.o              12
    data_init.o               40
    vector_table_M.o          66
    zero_init3.o              58
    -------------------------------------------
    Total:                   216

shb_l.a: [4]
    exit.o                    20
    -------------------------------------------
    Total:                    20

    Gaps                       2
    Linker created                     16    8'192
-----------------------------------------------
    Grand Total:             416       16    8'220
```

Map file sections: PLACEMENT SUMMARY

- Lists all the program sections
- A program section is a contiguous chunk of memory that has a symbolic name.
  - .intvec: for interrupts and where the vector table is placed (ROM address range)
  - .text: for the code (ROM address range)
  - .rodata: for read-only data (ROM address range)
  - .bss: holds uninitialized data that need to be set to **zero** during the system startup
  - CSTACK: holds the stack and is left **uninitialized** during startup.
- Note: **.a** files == archive files (i.e library file), ex: rt7M_tl.a

Placement Summary section

```
*******************************************************************************
*** PLACEMENT SUMMARY
***

"A0":  place at address 0x800'0000 { ro section .intvec };
"P1":  place in [from 0x800'0000 to 0x807'ffff] { ro };
define block CSTACK with size = 8K, alignment = 8 { };
define block HEAP with size = 8K, alignment = 8 { };
"P2":  place in [from 0x2000'0000 to 0x2001'7fff] {
          rw, block CSTACK, block HEAP };

  Section            Kind         Address    Size  Object
  -------            ----         -------    ----  ------
"A0":                                        0x40
  .intvec            ro code   0x800'0000    0x40  vector_table_M.o [3]
                             - 0x800'0040    0x40

"P1":                                       0x170
  .text              ro code   0x800'0040    0xa0  main.o [1]
  .text              ro code   0x800'00e0     0xa  delay.o [1]
  .text              ro code   0x800'00ea    0x3a  zero_init3.o [3]
  .text              ro code   0x800'0124    0x28  data_init.o [3]
  .iar.init_table    const     0x800'014c    0x10  - Linker created -
  .text              ro code   0x800'015c    0x1e  cmain.o [3]
  .text              ro code   0x800'017a     0x4  low_level_init.o [2]
  .text              ro code   0x800'017e     0x4  exit.o [2]
  .text              ro code   0x800'0182     0x2  vector_table_M.o [3]
  .text              ro code   0x800'0184     0xa  cexit.o [3]
  .text              ro code   0x800'0190    0x14  exit.o [4]
  .text              ro code   0x800'01a4     0xc  cstartup_M.o [3]
  .rodata            const     0x800'01b0     0x0  zero_init3.o [3]
                             - 0x800'01b0   0x170

"P2", part 1 of 2:                           0x1c
  .bss               zero     0x2000'0000     0xc  main.o [1]
  .bss               zero     0x2000'000c     0x8  main.o [1]
  .bss               zero     0x2000'0014     0x4  main.o [1]
  .bss               zero     0x2000'0018     0x4  main.o [1]
                            - 0x2000'001c    0x1c

"P2", part 2 of 2:                         0x2000
  CSTACK                      0x2000'0020  0x2000  <Block>
    CSTACK           uninit   0x2000'0020  0x2000  <Block tail>
                            - 0x2000'2020  0x2000
```

Map file: Data initialization

```
 Section            Kind         Address    Size  Object
 -------            ----         -------    ----  ------
"A0":                                       0x40
 .intvec            ro code   0x800'0000    0x40  vector_table_M.o [3]
                            - 0x800'0040    0x40

"P1":                                      0x184
 .text              ro code   0x800'0040    0xa0  main.o [1]
 .text              ro code   0x800'00e0     0xa  delay.o [1]
 .text              ro code   0x800'00ea    0x2e  copy_init3.o [3]
 .text              ro code   0x800'0118    0x28  data_init.o [3]
 .iar.init_table    const     0x800'0140    0x14  - Linker created -
 .text              ro code   0x800'0154    0x1e  cmain.o [3]
 .text              ro code   0x800'0172     0x4  low_level_init.o [2]
 .text              ro code   0x800'0176     0x4  exit.o [2]
 .text              ro code   0x800'017a     0x2  vector_table_M.o [3]
 .text              ro code   0x800'017c     0xa  cexit.o [3]
 .text              ro code   0x800'0188    0x14  exit.o [4]
 Initializer bytes  const     0x800'019c    0x1c  <for P2-1>
 .text              ro code   0x800'01b8     0xc  cstartup_M.o [3]
 .rodata            const     0x800'01c4     0x0  copy_init3.o [3]
                            - 0x800'01c4   0x184

"P2", part 1 of 2:                          0x1c
 P2-1                        0x2000'0000    0x1c  <Init block>
   .data            inited   0x2000'0000     0x4  main.o [1]
   .data            inited   0x2000'0004     0x8  main.o [1]
   .data            inited   0x2000'000c     0xc  main.o [1]
   .data            inited   0x2000'0018     0x4  main.o [1]
                           - 0x2000'001c    0x1c
```

- “Initializer bytes” section is created in ROM
- “.data” section is created in RAM
- Both of equal size
- The startup code copies the "Initializer bytes“ section from ROM to the “.data” sections in RAM.

DEMO: Data initialization

1. Setup code to contain global variables (structure demo)
2. Initialize some of the global variables
3. Leave some uninitialized
4. Use these global variables inside of main.
5. Step thru the IAR code before main
6. Step into __iar_data_init3
7. In memory window, set all global variables to all FFFFFFFF's
8. Check the global variable values in the "Watch" window
9. Step thru the __iar_data_init3 until the "BLX" instructions
10. Step thru the __iar_packbits_init_single3 while watching the memory window
11. Double check the global variable values in the "Watch" window
12. Remove uninitialized variables and show Map file sections (.bss is gone).

Data initialization

```
__iar_data_init3:
    0x800'0188: 0xb510         PUSH      {R4, LR}
    0x800'018a: 0x4907         LDR.N     R1, [PC, #0x1c]    ...
    0x800'018c: 0x4479         ADD       R1, R1, PC
    0x800'018e: 0x3118         ADDS      R1, R1, #24        ...
    0x800'0190: 0x4c06         LDR.N     R4, [PC, #0x18]    ...
    0x800'0192: 0x447c         ADD       R4, R4, PC
    0x800'0194: 0x3416         ADDS      R4, R4, #22        ...
    0x800'0196: 0xe004         B.N       0x800'01a2
    0x800'0198: 0x680a         LDR       R2, [R1]
    0x800'019a: 0x1d08         ADDS      R0, R1, #4
    0x800'019c: 0x4411         ADD       R1, R1, R2
    0x800'019e: 0x4788         BLX       R1
    0x800'01a0: 0x4601         MOV       R1, R0
    0x800'01a2: 0x42a1         CMP       R1, R4
    0x800'01a4: 0xd1f8         BNE.N     0x800'0198
    0x800'01a6: 0xbd10         POP       {R4, PC}
    0x800'01a8: 0x0000'0008    DC32      0x8
    0x800'01ac: 0x0000'0014    DC32      0x14 (20)
```

```
__iar_packbits_init_single3:
    0x800'014e: 0xb530           PUSH      {R4, R5, LR}
    0x800'0150: 0x6801           LDR       R1, [R0]
    0x800'0152: 0x6884           LDR       R4, [R0, #0x8]
    0x800'0154: 0x1842           ADDS      R2, R0, R1
    0x800'0156: 0x6841           LDR       R1, [R0, #0x4]
    0x800'0158: 0xeb02 0x0351    ADD.W     R3, R2, R1, LSR #1
    0x800'015c: 0x07c9           LSLS      R1, R1, #31
    0x800'015e: 0xd503           BPL.N     0x800'0168
    0x800'0160: 0x444c           ADD       R4, R4, R9
    0x800'0162: 0xe001           B.N       0x800'0168
    0x800'0164: 0x1c49           ADDS      R1, R1, #1
    0x800'0166: 0xd105           BNE.N     0x800'0174
    0x800'0168: 0x429a           CMP       R2, R3
    0x800'016a: 0xd00a           BEQ.N     0x800'0182
    0x800'016c: 0xf912 0x1b01    LDRSB.W   R1, [R2], #0x1
    0x800'0170: 0xf812 0x5b01    LDRB.W    R5, [R2], #0x1
    0x800'0174: 0x2900           CMP       R1, #0
    0x800'0176: 0xf804 0x5b01    STRB.W    R5, [R4], #0x1
    0x800'017a: 0xd4f3           BMI.N     0x800'0164
    0x800'017c: 0x1e49           SUBS      R1, R1, #1
    0x800'017e: 0xd5f7           BPL.N     0x800'0170
    0x800'0180: 0xe7f2           B.N       0x800'0168
    0x800'0182: 0x300c           ADDS      R0, R0, #12        ...
    0x800'0184: 0xbd30           POP       {R4, R5, PC}
```

The C-Standard initialization sequence

- IAR implements standard-compliant startup code.
- By the time main() is called, the C standard requires that:
  - All **initialized** variables get their initial values.
  - And all **uninitialized** variables to be set to zero.
- Some vendors are not compliant with this standard, so should test the start up code to verify.
- If the .bss sections are not cleared, then one might need to explicitly initialize all previously uninitialized variables to zero.
- Note that this would not be optimal:
  - We would be converting the .bss sections to .data sections, which require a matching "Initializer bytes“ section in ROM. In other words, we would be taking space in ROM for a bunch of zeros.

Where does the SP (stack pointer) get its initial value?

Where does the PC (program counter) end up at the function ***__iar_program_start?***

After "Reset"

After “Reset”

- The ARM Cortex-M is hardwired after reset to:
  - Copy the bits from address 0 to the SP register.
  - Copy all bits (except the least-significant-bit) from address 0x4 to the PC register.
- Recall:
  - The LSb of any value loaded to the PC must be one, because this bit indicates Thumb mode which is the only mode supported by Cortex-M.

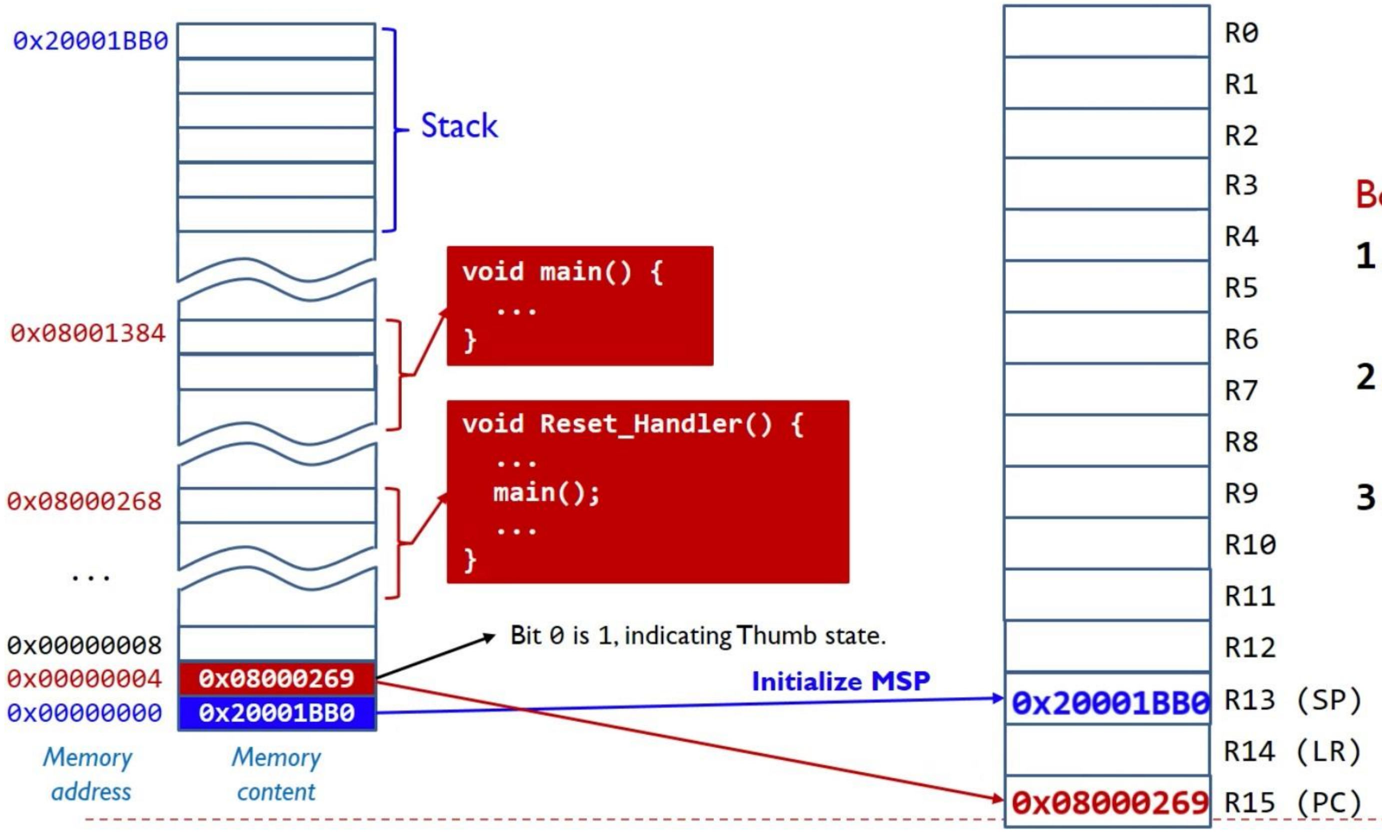
Disassembly

```
   0x800'0000: 0x2000'2020      DC32        CSTACK$$Limit
   0x800'0004: 0x0800'01b9      DC32        __iar_program_start
   0x800'0008: 0x0800'017b      DC32        BusFault_Handler
   0x800'000c: 0x0800'017b      DC32        BusFault_Handler
   0x800'0010: 0x0800'017b      DC32        BusFault_Handler
   0x800'0014: 0x0800'017b      DC32        BusFault_Handler
   0x800'0018: 0x0800'017b      DC32        BusFault_Handler
   0x800'001c: 0x0000'0000      DC32        0x0
   0x800'0020: 0x0000'0000      DC32        0x0
   0x800'0024: 0x0000'0000      DC32        0x0
   0x800'0028: 0x0000'0000      DC32        0x0
   0x800'002c: 0x0800'017b      DC32        BusFault_Handler
   0x800'0030: 0x0800'017b      DC32        BusFault_Handler
   0x800'0034: 0x0000'0000      DC32        0x0
   0x800'0038: 0x0800'017b      DC32        BusFault_Handler
   0x800'003c: 0x0800'017b      DC32        BusFault_Handler
void main(void)
{
main:
   0x800'0040: 0xb538           PUSH        {R3-R5, LR}
    p1.x = sizeof(Point);
```

Name	Value
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
APSR	0x00000000
IPSR	0x00000000
EPSR	0x01000000
PC	0x080001B8
SP	0x20002020
LR	0xFFFFFFFF
PRIMASK	0x00000000
BASEPRI	0x00000000

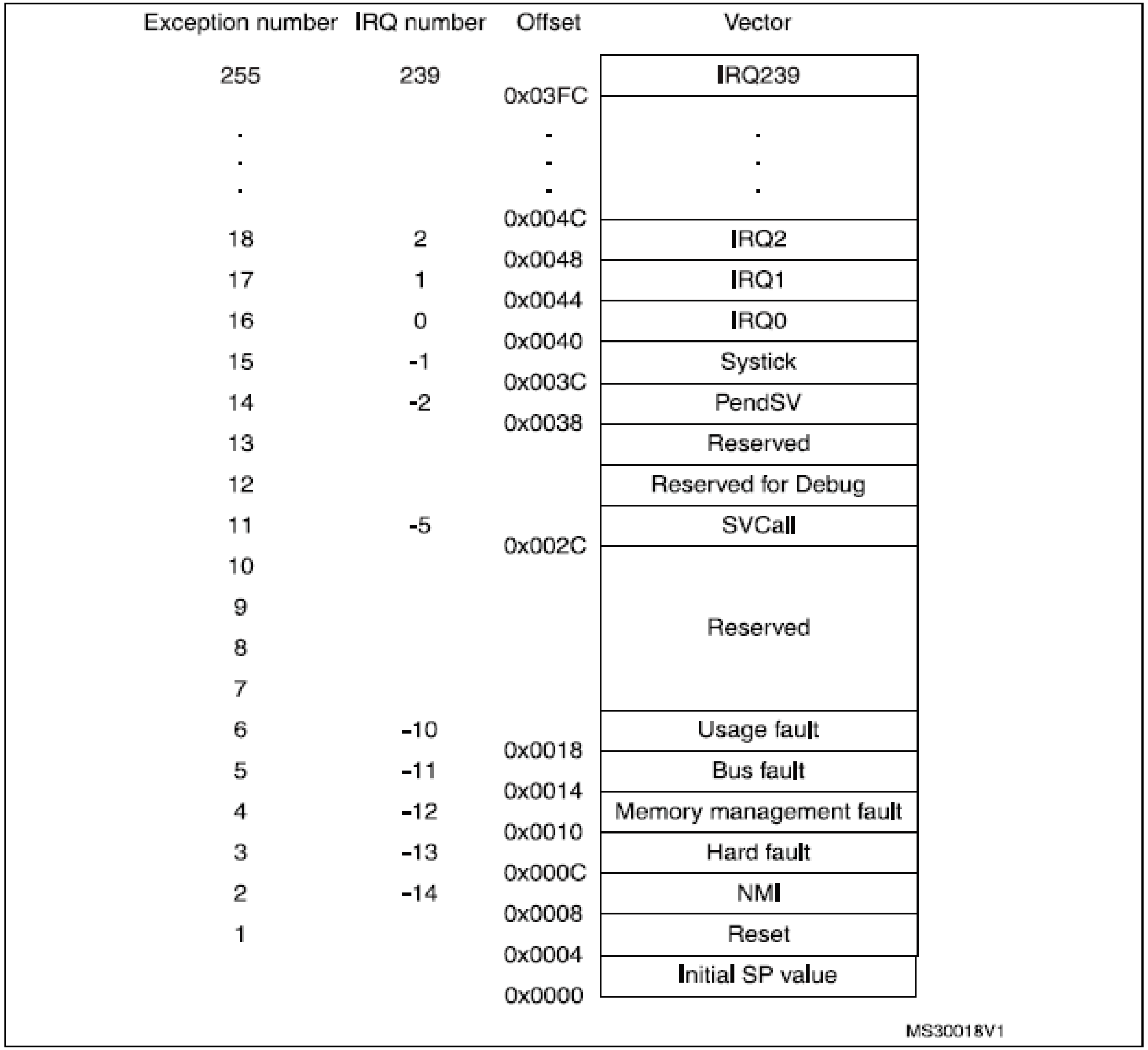
<u>Note:</u> These are not machine instructions. These are simply words in memory.

After “Reset”



Booting Sequence:

1. Read pins BOOT0 and BOOT1 and determine boot mode
2. Fetch MSP from address 0x00000000
3. Fetch PC from address 0x00000004



Vector Table

- Located at address 0x00 in ROM.
- The vector table contains the initial value of the stack pointer SP and the start address of the PC.
- It also contains the exception and interrupt vectors that the processor can handle.
- *Source:*
  - *PM0214-Programming Manual for STM32 Cortex-M4*
  - *Section 2.3.4*

Vector table from IAR vs datasheet

- The Vector Table from the datasheet is much larger than the one from the disassembly view.
- The vectors labeled IRQ0, IRQ1, etc. are not present in the disassembly view.
- The vector table provided by the IAR library is generic. It contains only the standard exception vectors that are defined at the beginning of the table and are common to all Cortex-M microcontrollers.
- The IAR table does not contain interrupt vectors that are specific to a given microcontroller, such as IRQ0, IRQ1, etc., so it cannot really handle any interrupts.
- We will need to replace the generic IAR Vector Table with the specific one that will match exactly the layout defined in the Datasheet of the specific microcontroller.

Fault Handlers

- The IAR startup code defines all exception handlers, such as BusFault, DebugMonitor, HardFault, MemoryManager, and Non-Maskable-Interrupt, but they all point to the same piece of code.
- The IAR code associated with all these exception handlers is a single branch instruction, which jumps to itself.
- So an occurrence of any of the exceptions ends up tying the CPU in a tight endless loop, which is good for debugging, because when you break into the code, you will find it looping inside the exception handler.
- However, this is not good for production as the device will become locked and unresponsive.
- Should consider implementing some built-in recovery mechanism upon hitting these exceptions (reset device for example).

```
BusFault_Handler:
DebugMon_Handler:
HardFault_Handler:
MemManage_Handler:
NMI_Handler... +5 symbols not displayed:
   0x800'017a: 0xe7fe         B.N       BusFault_Handler  ...
```

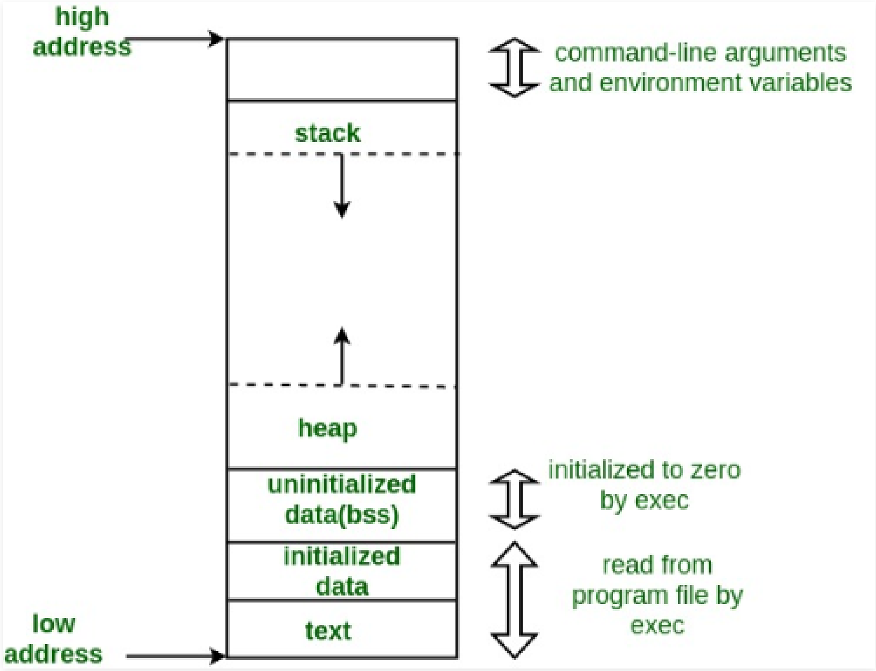
Memory layout of C Programs

https://www.geeksforgeeks.org/memory-layout-of-c-program/

Memory Layout of C Programs

A typical memory representation of C program consists of following sections.

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap



BREAK 1

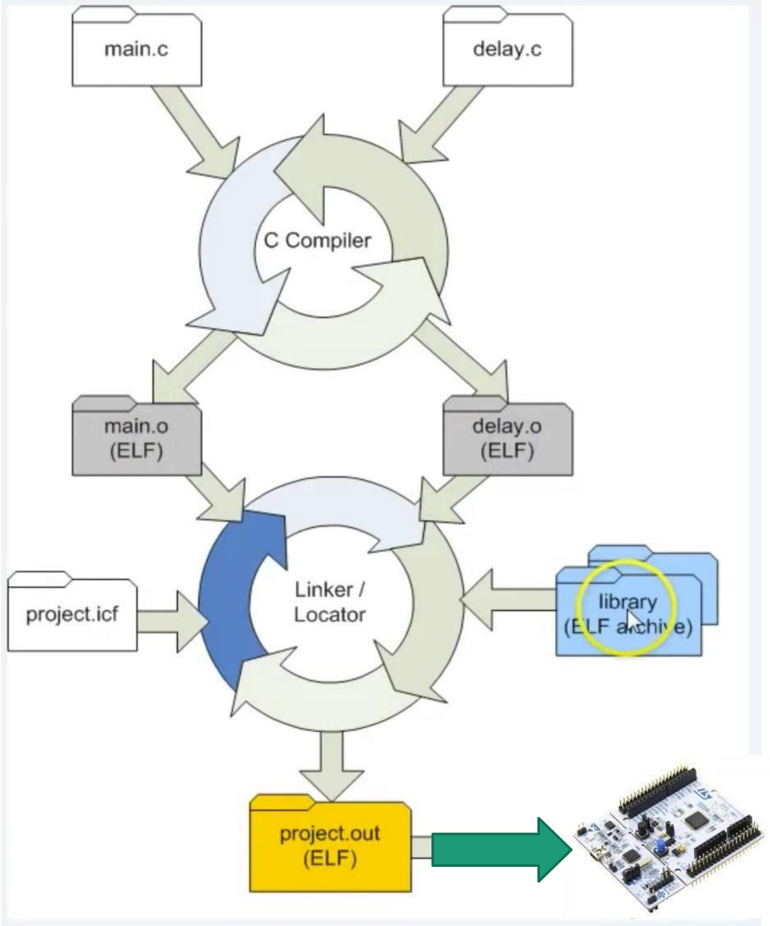
The Vector Table

- The build process
- ELF files
- Relocatable code
- Injecting code at startup
- The “.intvec” section
- Initializing a user-defined vector table

The build process

Recall that an object file contains "relocatable" machine code that is not directly executable because it is not yet committed to any specific address in memory.

It is the job of the linker to combine all the objects, resolve the cross-module references, and fix the addresses.



ELF files

- ELF is a standard Object file format. It stands for "Executable and Linkable Format”, or "Extensible Linking Format".
- The first few ASCII characters of the file spell out 'E', 'L','F'. This is the indicator of the ELF file format.
- ELF is not the only format for object files, but it is one of the most popular formats used by modern development tools.
- Can use viewer installed with the IAR tool set:
  - C:\Program Files (x86)\IAR Systems\Embedded Workbench 8.3\arm\bin**ielfdumparm.exe**
  - Can be used to dump the content of the binary ELF file into human-readable text

ELF files

- The ELF file contains several sections.
- Like the map file, it contains sections such as:
  - .data --> for initialized data
  - .bss --> for uninitialized data
  - .text --> for code
- Sections that hold the symbolic information for the linker.
- Sections that contain debug information for the debugger.

ELF files

- The ELF dump utility provides a quick way to view the disassembly of the generated code, without loading the program into the target and inspecting the disassembly view.
- Can use either the dump utility with any ELF type files: **Object** files as well as **Output** files:
  - <project_path>\Obj\main.o
  - <project_path>\Exe\main.out
  - EX: **ielfdumparm.exe** --all Debug\Obj\main.o > main_obj.txt
- <u>Warning:</u>
  - Should never use the size of the object file to assess the code size generated from a given .c source code. The actual machine code is only a small part among many other parts in that object file.
  - The only reliable source of information about the code size of various modules is the linker map file.

Demo: ELF Files

- Use ielfdumparm.exe utility
- Run against \Obj\main.o
  - ***ielfdumparm --all Obj\main.o > main_obj.txt***
- Run against \Exe\main.out
  - ***ielfdumparm --all Exe\main.out > main_out.txt***
- Open both files side-by-side in IAR
- Find the ".text" section for the "main".
- Look for the BL instruction in both files
  - Why is Op-code different?
- Look at the variable section in both files
  - Why are the addresses different?

Relocatable Code: Object vs Output files

OBJ\MAIN.O

main_obj.txt

```
-----------------------------------------------------------
Section #17 .text:

          $t:
          `.text17`:
          main:
   0x0: 0xb538        PUSH   {R3-R5, LR}
   0x2: 0x482a        LDR.N  R0, [PC, #0xa8] ; p1
   0x4: 0x2104        MOVS   R1, #4
   0x6: 0x8001        STRH   R1, [R0]
   0x8: 0x2143        MOVS   R1, #67        ; 0x43
   0xa: 0x7081        STRB   R1, [R0, #0x2]
   0xc: 0x4928        LDR.N  R1, [PC, #0xa0] ; p3
   0xe: 0x6008        STR    R0, [R1]
  0x10: 0x2004        MOVS   R0, #4
  0x12: 0x680a        LDR    R2, [R1]
  0x14: 0x8010        STRH   R0, [R2]
  0x16: 0x2022        MOVS   R0, #34        ; 0x22
  0x18: 0x6809        LDR    R1, [R1]
  0x1a: 0x7088        STRB   R0, [R1, #0x2]
  0x1c: 0x4825        LDR.N  R0, [PC, #0x94] ; r1
  0x1e: 0x2101        MOVS   R1, #1
  0x20: 0x8001        STRH   R1, [R0]
  0x22: 0x2101        MOVS   R1, #1
  0x24: 0x7081        STRB   R1, [R0, #0x2]
  0x26: 0x2105        MOVS   R1, #5
```

EXE\MAIN.OUT

main_out.text

```
-------------------------------------------------------------------
Section #5 P1 ro:

                $t:
                `.text17`:
                main:
  0x800'0040: 0xb538        PUSH   {R3-R5, LR}
  0x800'0042: 0x482a        LDR.N  R0, [PC, #0xa8]     ; p1
  0x800'0044: 0x2104        MOVS   R1, #4
  0x800'0046: 0x8001        STRH   R1, [R0]
  0x800'0048: 0x2143        MOVS   R1, #67             ; 0x43
  0x800'004a: 0x7081        STRB   R1, [R0, #0x2]
  0x800'004c: 0x4928        LDR.N  R1, [PC, #0xa0]     ; p3
  0x800'004e: 0x6008        STR    R0, [R1]
  0x800'0050: 0x2004        MOVS   R0, #4
  0x800'0052: 0x680a        LDR    R2, [R1]
  0x800'0054: 0x8010        STRH   R0, [R2]
  0x800'0056: 0x2022        MOVS   R0, #34             ; 0x22
  0x800'0058: 0x6809        LDR    R1, [R1]
  0x800'005a: 0x7088        STRB   R0, [R1, #0x2]
  0x800'005c: 0x4825        LDR.N  R0, [PC, #0x94]     ; r1
  0x800'005e: 0x2101        MOVS   R1, #1
  0x800'0060: 0x8001        STRH   R1, [R0]
  0x800'0062: 0x2101        MOVS   R1, #1
  0x800'0064: 0x7081        STRB   R1, [R0, #0x2]
  0x800'0066: 0x2105        MOVS   R1, #5
```

Object vs Output files

OBJ\MAIN.O

main_obj.txt

```
0xb2: 0x0020            MOVS     R0, R4
0xb4: 0xf7ff 0xfffe     BL       delay
0xb8: 0x4d11            LDR.N    R5, [PC, #0x44]
0xba: 0x6828            LDR      R0, [R5]
0xbc: 0xf050 0x0020     ORRS.W   R0, R0, #32
0xc0: 0x6028            STR      R0, [R5]
0xc2: 0x0020            MOVS     R0, R4
0xc4: 0xf7ff 0xfffe     BL       delay
0xc8: 0x6828            LDR      R0, [R5]
0xca: 0xf030 0x0020     BICS.W   R0, R0, #32
0xce: 0x6028            STR      R0, [R5]
0xd0: 0xe7ee            B.N      @b0
0xd2: 0xbf00            NOP
              `$d.32`:
0xd4: 0x0000'0000       DC32     p1
0xd8: 0x0000'0000       DC32     r1
0xdc: 0x0000'0000       DC32     t1
0xe0: 0x0000'0000       DC32     myArray1
0xe4: 0x0000'0000       DC32     p2
0xe8: 0x0000'0000       DC32     r2
0xec: 0x0000'0000       DC32     t2
0xf0: 0x0000'0000       DC32     myArray2
```

EXE\MAIN.OUT

main_out.txt

```
0x800'00f2: 0x0020            MOVS     R0, R4
0x800'00f4: 0xf000 0xf826     BL       delay
0x800'00f8: 0x4d11            LDR.N    R5, [PC, #0x44]
0x800'00fa: 0x6828            LDR      R0, [R5]
0x800'00fc: 0xf050 0x0020     ORRS.W   R0, R0, #32
0x800'0100: 0x6028            STR      R0, [R5]
0x800'0102: 0x0020            MOVS     R0, R4
0x800'0104: 0xf000 0xf81e     BL       delay
0x800'0108: 0x6828            LDR      R0, [R5]
0x800'010a: 0xf030 0x0020     BICS.W   R0, R0, #32
0x800'010e: 0x6028            STR      R0, [R5]
0x800'0110: 0xe7ee            B.N      @80000f0
0x800'0112: 0xbf00            NOP
                    `$d.32`:
0x800'0114: 0x2000'0000       DC32     p1
0x800'0118: 0x2000'0004       DC32     r1
0x800'011c: 0x2000'000c       DC32     t1
0x800'0120: 0x2000'0018       DC32     myArray1
0x800'0124: 0x2000'0024       DC32     p2
0x800'0128: 0x2000'0028       DC32     r2
0x800'012c: 0x2000'0030       DC32     t2
0x800'0130: 0x2000'003c       DC32     myArray2
```

Relocatable Code

- Some instructions have different encoding between the Object and the Output files .
- For example, the 32-bit BL instruction, by which main calls the delay function is encoded as **0xf7ff 0xfffe** in the object file and **0xf000 0xf81e** in the final image.
- BL is a PC-relative instruction, meaning that in order to branch to the delay function the Program Counter (PC) will be incremented by the signed immediate offset encoded in the instruction itself.
- The problem is that the object file does not know where the delay function will end up in memory, so the BL opcode in the object file contains a generic offset 0x7ff.fffe
- This offset is then fixed by the linker, after the linker decides where the delay function will be in memory with respect to main.

Relocatable Code

- The linker also needs to fix the addresses of the variables as well.
- For example, the addresses of the variables p1, r1, t1, …etc. are not known at compile time.
- All these addresses are zero in the object file.
- The linker fixed these addresses in the output file after figuring out where to put the variables p1, r1, t1…etc.
- Note:
  - So the linker must be specific to the target processor, because it must "know" the instructions and how to fix them at the binary opcode level.
  - A linker designed for the x86 processor of your PC cannot be used to link programs for the ARM processor; even though all these tools might be using the ELF file format. We need both a compiler and a linker for the target processor.

Objective

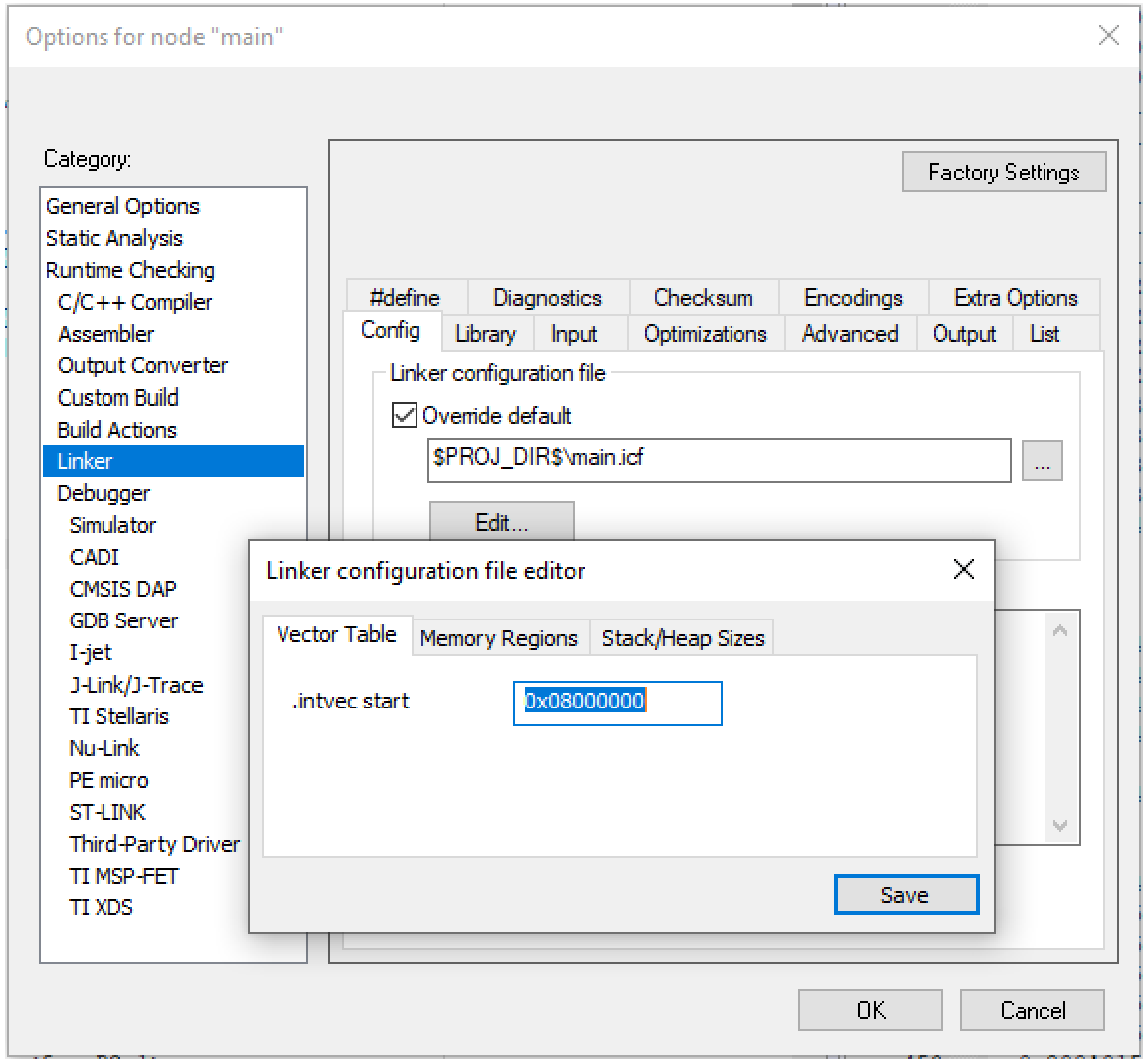
How do we place our own defined vector table in memory at address 0x0 (or 0x800.0000)?

Injecting code at startup

- How can we make a C module accomplish anything at the startup time, when the machine is not ready yet for executing C code.
  - Sack pointer is not set up yet, the initialized data is not copied from ROM to RAM, and the .bss section is not cleared yet either.
- Startup code for most other processors can't be written in C and requires the use of assembly language.
- However, the ARM Cortex-M has been specifically designed to reduce the need for low-level assembly programming.
- But even for Cortex-M, the startup code will require some non-standard language extensions

Injecting code at startup

- To see exactly where the linker has put the original vector table from the library, check the linker map file.
- The default vector table is placed in the **.intvec** section at address zero in ROM.
- To understand how this section is defined, open the linker script file **main.icf**.
- The purpose of the linker script is to tell the linker where to place the various merged program sections in the address space.
- Also, in the map file you will find the entry **"__vector_table**" at 0x800.0000



.intvec section

.intvec section

- There is no standard C syntax to place variables in specific sections.
- IAR provides an extension as follows: @ "<section_name>"
  - Example: @ ".intvec“
  - Search the “Help” content in IAR for details

Declare and place your own sections

To declare new sections—in addition to the ones used by the IAR build tools—to hold specific parts of your code or data, use mechanisms in the compiler and assembler. For example:

```
/* Place a variable in that section. */
const short MyVariable @ "MYOWNSECTION" = 0xF0F0;
```

Demo: Vector Table

- Add new "**startup_stm32f401xe.c**" file
- Define an array "**__vector_table**"
- Add the file to the project
- Build and view the map file
- Our **__vector_table** overrode the one from IAR.
- But no longer in .intvec section.
- Use **@ ".intvec"**
- Build and view the map file.
- Now the ".intvec" is in RAM

Demo: Vector Table

- The compiler accepts the new startup file.
- However, the **.intvec** section is not at address zero as before. It is pushed down to the RAM region.
  - *Why?*
  - *And how can this be resolved?*

Demo: Vector Table

- Why the .intvec section is placed differently?
  - The vector table that we defined is a variable
  - So the compiler can't put it into the ROM (read-only memory).
- How to fix this issue?
  - To force the vector table array into the ROM, we need to define the vector table as a constant (using the “const” keyword that the C language provides).
- Build and view the map file.
  - The .intvec is back in ROM and the __vector_table is at address 0x800.0000
- Run and view the vector table in the disassembly view.

In summary

- We added a new file, named startup_stm32f401xe.c to the project.
- We added the vector table as a constant array of integers.
- Got the new vector table from our startup_stm32f401xe.c to get linked instead of the generic one from the IAR library.
- Got the vector table located in the section ".intvec" at address 0x800.0000
- Now we need to initialize the table properly.

startup_stm32f401xe.c

```c
// Startup code for Nucleo F401RE

int const __vector_table[] @ ".intvec" = {
      0,      //0x800'0000
      0,
};
```

Initializing the vector table

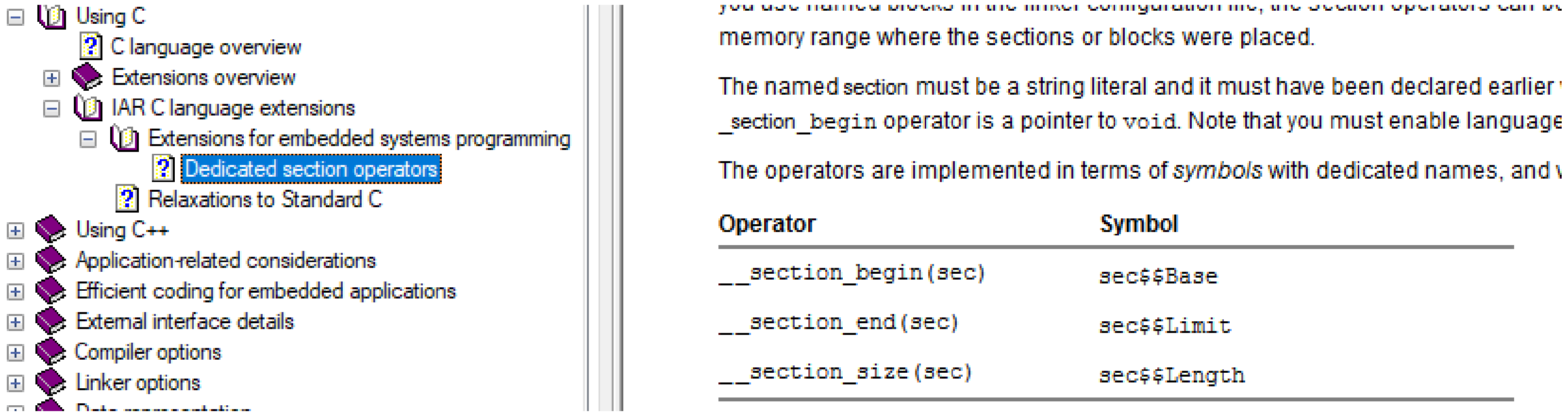
- Ho to initialize the vector table with the correct stack pointer and all interrupts available in our Nucleo F401RE board?
  - Initialize the pointer to the Top of stack without breaking compatibility with the IAR tool.
  - Initialize the pointer to the reset handler
    - The Reset handler is the initial value copied to the PC register when the microcontroller comes out of reset,
    - This is the address in ROM where the ARM Cortex-M processor starts executing code.
- Initialize the standard exception handlers (common to all ARM Cortex-M)

Demo: Vector Table – Initial Stack Pointer

Reverse engineering + Hacking

Comment out our __vector_table and view in the map file how the CSTACK limits are presented.

- CSTACK$$Limit is the entry seen in the map file.
- This indicates that the symbol CSTACK$$Limit is known to the linker.
- Searching the IAR help, will find that the linker generates these symbols more info about the section limits.
- Use the address of **CSTACK$$Limit** variable in element [0] of the array.
- **View disassembly and the SP register to confirm correctness.**



Demo: Vector Table – Reset handler

- The Reset handler is the initial value copied to the PC register when the microcontroller comes out of reset
- This is the address in ROM where the ARM Cortex-M processor starts executing code.
- Looking at the default initialization of the Reset handler from the standard IAR library, we see that the reset handler is set to the "**__iar_program_start**".
- Let's use the use the address of the function "**__iar_program_start**" in element[1] of the array.
- Similar to the CSTACK$$Limit symbol, we need to declare the symbol **__iar_program_start** as a function, by providing its prototype.
- "__iar_program_start" is a function that takes no arguments and returns no arguments.
- **View the disassembly and the PC register and run the program to confirm correctness.**

Demo: Vector Table – Standard Handlers

- These entries in the vector table are common to all ARM Cortex-M processors and are for handling exceptions.
  - Note that they are not arranged contiguously in the vector table as there are gaps in the table marked as "Reserved". So we need to preserve this layout in our custom vector table.
- Initialize the standard exceptions similar to the Reset Handler.
- The names of exception and interrupt handlers are part of the CMSIS standard (refer to names in the “**stm32f4xx_it.h**” file generated by the STM32CubeMX tool).
- Unlike the __iar_program_start Reset handler, which was taken from the standard IAR library, **we need to write the code for these exception handlers**.
- The standard way to code an exception handler is to **use an endless loop** that ties up the CPU when the corresponding exception occurs.

Demo: Vector Table – Test Handlers

- Build and run the blinking LED program
- Break before the call to the “delay” function
- Step into “delay”
- Modify the LR register to be an “**even**” address value.
- Run program and verify that we’re inside one of our handlers.

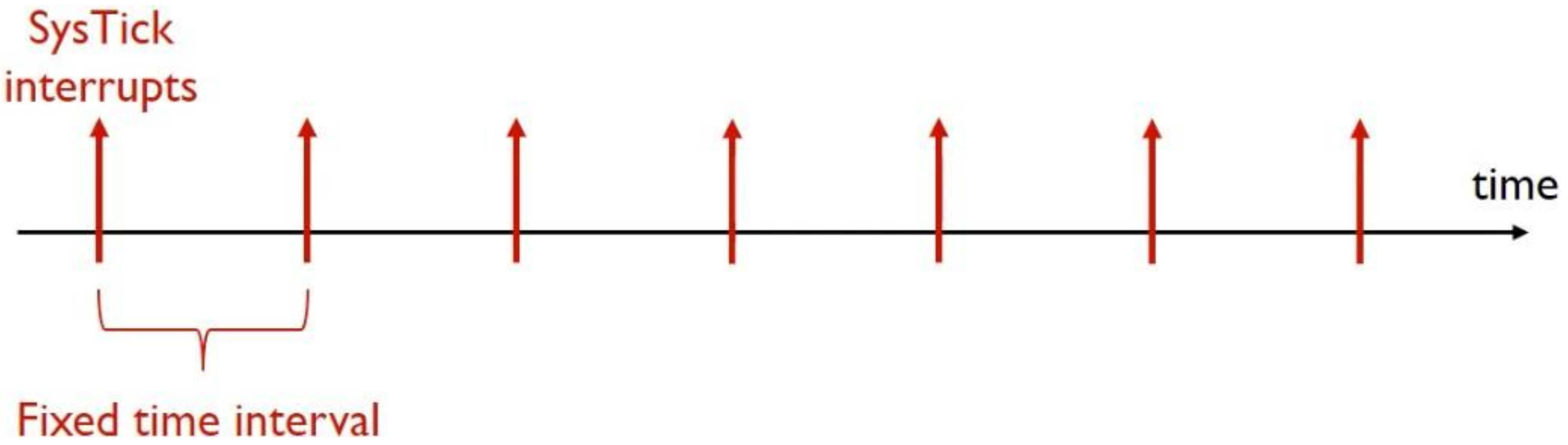
BREAK 2

System Timer

- SysTick,
- Application
- Hardware

System Timer SysTick

A basic timer that can be used to generate interrupts at regular time intervals



System Timer SysTick

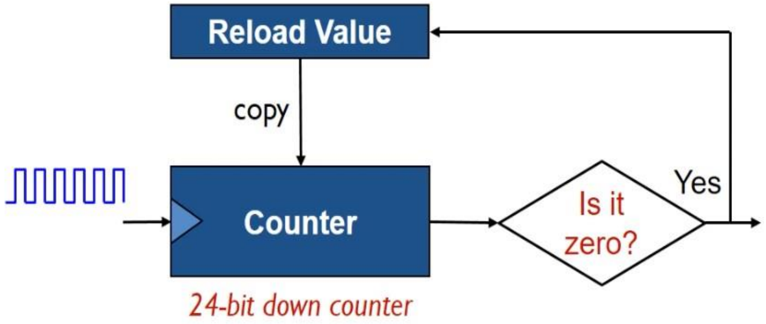
- The SysTick is a hardware peripheral.
- It is a separate hardware block on the MCU silicon.
- Clocked by the CPU clock
- Consists of three registers:
  - STK_VAL register (SysTick->VAL in CMSIS)
    - A 24-bit down-counter that decrements by one at every CPU clock cycle.
    - When the counter reaches zero, it generates an interrupt to the CPU.
  - STK_LOAD (SysTick->LOAD)
    - The LOAD register specifies the start value to load into the STK_VAL register
  - STK_CTRL (SysTick->CTRL)
    - Enables the SystTick features
- *Source: PM0214-Programming Manual for STM32 Cortex-M4 (Section 4.5)*

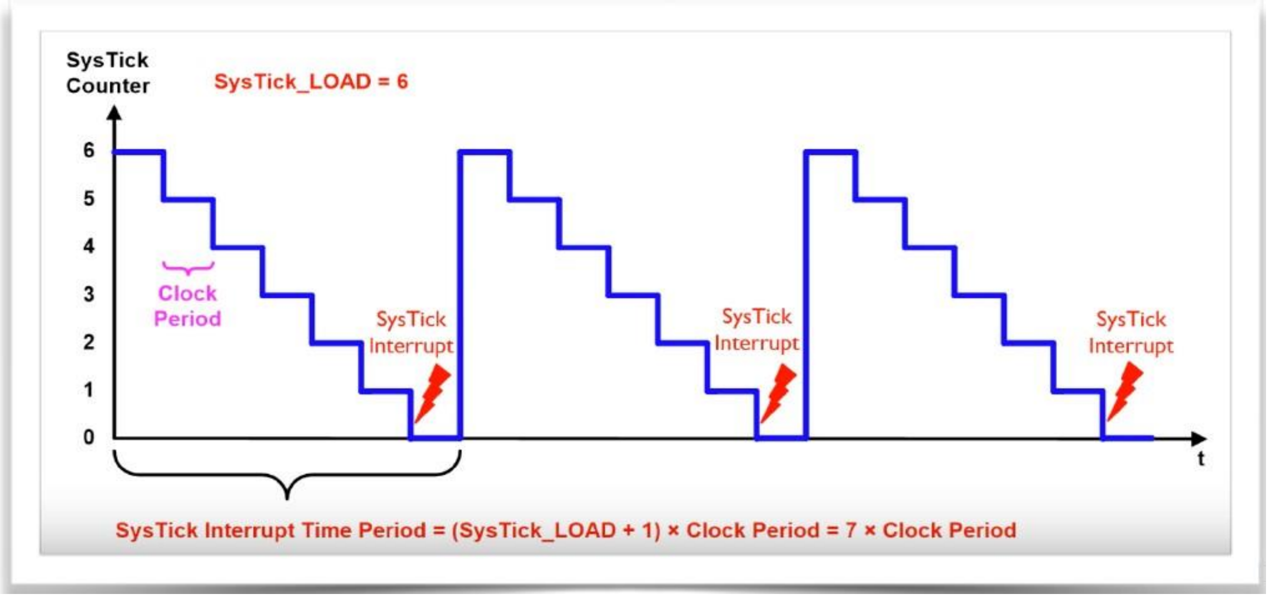
System Timer Application

- Software can use it to implement a time delay function (**assignment**)
- Execute some specific tasks software periodically polling to check peripheral status or read external inputs
- Operating System rely on the System Timer to invoke scheduler in order to support multitasking and to improve the CPU utilization

Timers & Counters: Hardware

- Timer is a hardware component built within the processor chip.
- If enabled, it counts upward or downward driven via a clock source.





4.5.2 SysTick reload value register (STK_LOAD)

Address offset: 0x04

Reset value: 0x0000 0000

Required privilege: Privileged

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								RELOAD[23:16]							
								rw	rw	rw	rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RELOAD[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:24 Reserved, must be kept cleared.

Bits 23:0 **RELOAD:** RELOAD value

The LOAD register specifies the start value to load into the STK_VAL register when the counter is enabled and when it reaches 0.

Calculating the RELOAD value

The RELOAD value can be any value in the range 0x00000001-0x00FFFFFF. A start value of 0 is possible, but has no effect because the SysTick exception request and COUNTFLAG are activated when counting from 1 to 0.

The RELOAD value is calculated according to its use:

- To generate a multi-shot timer with a period of N processor clock cycles, use a RELOAD value of N-1. For example, if the SysTick interrupt is required every 100 clock pulses, set RELOAD to 99.
- To deliver a single SysTick interrupt after a delay of N processor clock cycles, use a RELOAD of value N. For example, if a SysTick interrupt is required after 100 clock pulses, set RELOAD to 99.

Configure SysTick timer for interrupts

STK_LOAD (Programming Manual)

SysTick->LOAD (CMSIS)

Configure SysTick timer for interrupts

STK_VAL (programming manual)

SysTick->VAL (CMSIS)

4.5.3 SysTick current value register (STK_VAL)

Address offset: 0x08

Reset value: 0x0000 0000

Required privilege: Privileged

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								CURRENT[23:16]							
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CURRENT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:24 Reserved, must be kept cleared.

Bits 23:0 **CURRENT:** Current counter value

The VAL register contains the current value of the SysTick counter.

Reads return the current value of the SysTick counter.

A write of any value clears the field to 0, and also clears the COUNTFLAG bit in the STK_CTRL register to 0.

4.5.1 SysTick control and status register (STK_CTRL)

Address offset: 0x00

Reset value: 0x0000 0000

Required privilege: Privileged

The SysTick CTRL register enables the SysTick features.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															COUNT FLAG
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													CLKSO URCE	TICK INT	EN ABLE
													rw	rw	rw

Bits 31:17 Reserved, must be kept cleared.

Bit 16 COUNTFLAG:

Returns 1 if timer counted to 0 since last time this was read.

Bits 15:3 Reserved, must be kept cleared.

Bit 2 **CLKSOURCE:** Clock source selection

Selects the clock source.

0: AHB/8

1: Processor clock (AHB)

Bit 1 **TICKINT**: SysTick exception request enable

0: Counting down to zero does not assert the SysTick exception request

1: Counting down to zero to asserts the SysTick exception request.

Note: Software can use COUNTFLAG to determine if SysTick has ever counted to zero.

Bit 0 **ENABLE**: Counter enable

Enables the counter. When ENABLE is set to 1, the counter loads the RELOAD value from the LOAD register and then counts down. On reaching 0, it sets the COUNTFLAG to 1 and optionally asserts the SysTick depending on the value of TICKINT. It then loads the RELOAD value again, and begins counting.

0: Counter disabled

1: Counter enabled

Configure SysTick timer for interrupts

STK_CTRL (Programming Manual)

SysTick->CTRL (CMSIS)

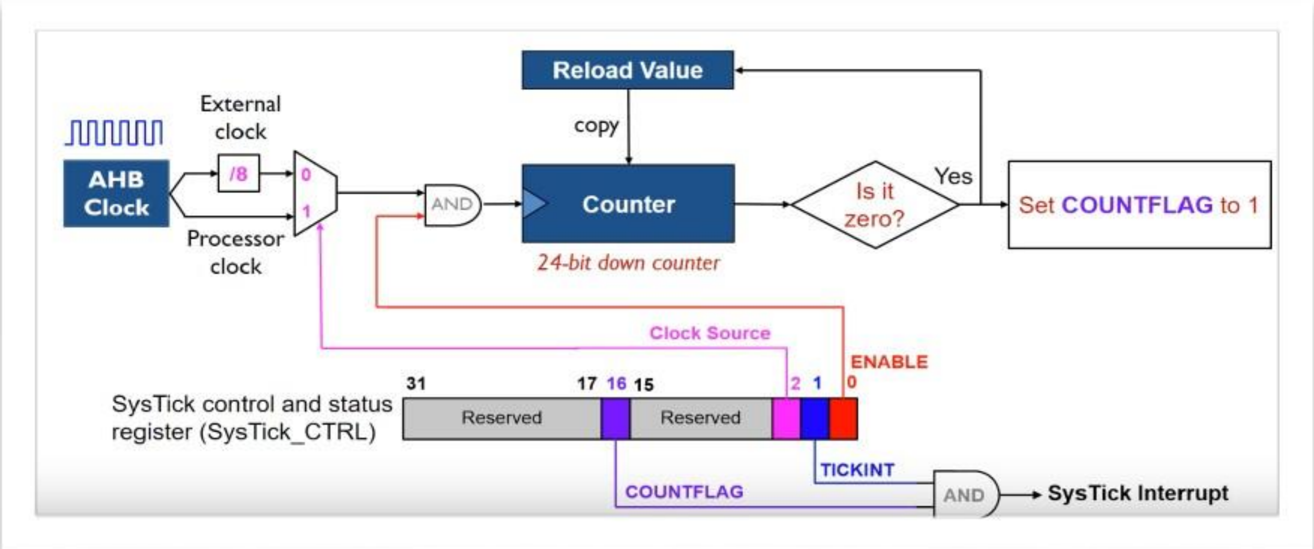
- *Clock-Source*
- *Interrupt-Enable*
- *Counter enable*

Use in CMSIS

```
__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)
{
  if ((ticks - 1) > SysTick_LOAD_RELOAD_Msk)  return (1);      /* Reload value impossible */

  SysTick->LOAD  = ticks - 1;                                  /* set reload register */
  NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);  /* set Priority for Systick Interrupt */
  SysTick->VAL   = 0;                                          /* Load the SysTick Counter Value */
  SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                   SysTick_CTRL_TICKINT_Msk   |
                   SysTick_CTRL_ENABLE_Msk;                    /* Enable SysTick IRQ and SysTick Timer */
  return (0);                                                  /* Function successful */
}
```

Source: core_cm4.h



Interrupts Overview

- Polling
- CPU Interrupts
- Demo SysTick interrupt

Polling

- The delay function busy-waits for million iterations, which happens to take about one second.
- Busy-waits means that the CPU is doing nothing else, but constantly checking whether the iteration counter has dropped all the way to zero.
- In programming this is called “**polling**” and is an approach, in which the program keeps checking for a certain condition to occur in order to do something in response.
- This “polling” example inside delay() ties up the CPU and renders it unavailable for any other work.

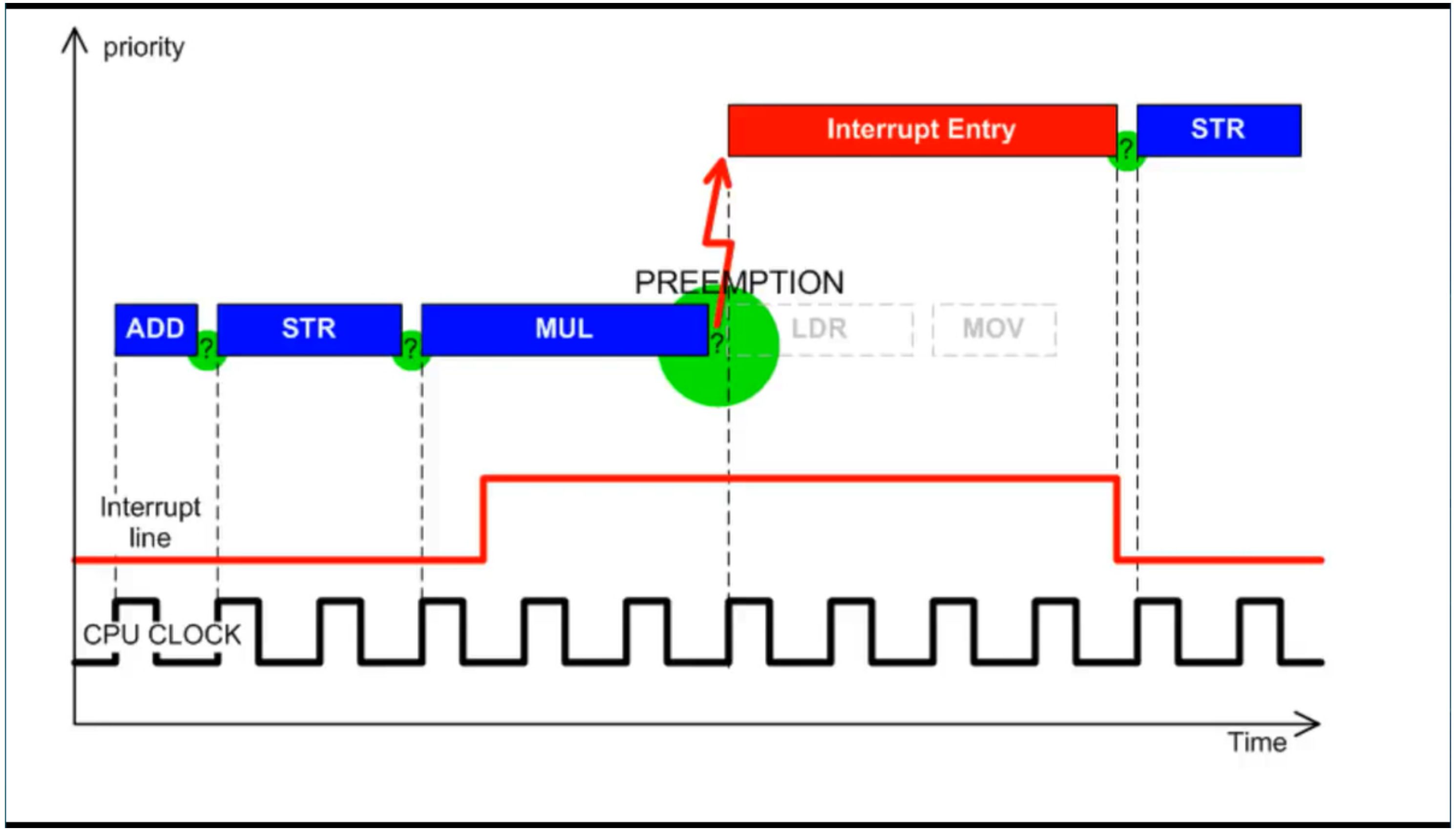
```
// main.c
delay(1000000);


// delay.c
#include "delay.h"
void delay(int volatile iteration) {
   while (iteration > 0) {
      iteration--;
   }
}
```

CPU Interrupts

- The CPU "listens" to the interrupts coming in through interrupt lines.
- The CPU has a special built-in hardware that samples the status of the interrupt line after every instruction.
- As long as the interrupt line is low, the CPU fetches the next instruction in the pipeline.
- However, when the interrupt line is high, the special CPU hardware forces the CPU to branch to a different code path (ISR == Interrupt Service Routine)
- This process is called “PREEMPTION”.
- While the instructions are strictly synchronized to the CPU clock, the interrupt line is generally not.
- The interrupt line can change its status at any time and typically in the middle of an instruction, completely “ASYNCHRONOUS” to the instruction execution.

Interrupts are asynchronous



SysTick usage

4.4.5 SysTick usage hints and tips

Some implementations stop all the processor clock signals during deep sleep mode. If this happens, the SysTick counter stops.

Ensure software uses aligned word accesses to access the SysTick registers.

The SysTick counter reload and current value are not initialized by hardware. This means the correct initialization sequence for the SysTick counter is:

1. Program reload value.
2. Clear current value.
3. Program Control and Status register.

Source: Cortex-M4 Generic User Guide

Demo: SysTick usage

- Build on demo with custom vector_table
- Remove all code from main except for the blinking LED part.
- Enable the use of CMSIS
  - In project settings
  - And copy files “stm32f401xe.h“ & “system_stm32f4xx.h” to the project directory
- Setup SysTick registers
- Setup Load value to 1 Sec (16,000,000 cycles)
- Blink LED within SysTick_Handler()

Demo: How to determine the interval

- To set the interval to one second, we need to know the speed of the CPU clock in terms of cycles per second.
- For our board = 16MHz (Default, see section 3.11 of datasheet).
- 16,000,000 cycles per second, which is 0x00F4.2400
  - Ensure it does not overflow SysTick->LOAD
  - Max value for SysTick->LOAD is 0x00FF.FFFF
- View SysTick->LOAD value in “Symbolic Memory” window (at address 0xE000E014)
- Set breakpoint inside SysTick_Handler()
- Remove delay code from main
- Add LED toggle code inside Systick_Handler()
  - **GPIOA->ODR ^= GPIO_ODR_OD5;**

Assignment 07

Suggested Reading

- ***"The Definitive Guide to ARM Cortex M3 & M4" by Joseph Yiu (Third Edition)***
  - Chapter 4.5: Exceptions and interrupts.
  - Chapter 7.1, 7.2: Exceptions & Interrupts overview and types.
  - Chapter 8.1: Exception Handling.
- ***"An Embedded Software Primer" by David E. Simon***
  - Chapter 4: Interrupts
- **RM0368 Reference Manual**
  - Section 2.4: Boot Configuration
- **PM0214 Programming Manual:**
  - Section 4.5: SysTick Timer