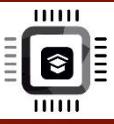
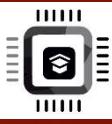


Embedded System Programming on ARM Cortex-m3/m4

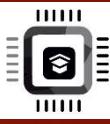


Overview



What will you learn at the end of the course ??

- Architecture
- Programming model
- Memory Architecture
- Interrupt Management
- Exception handling
- Buttons, LEDs
- Programming and Debugging using KEIL
- Debugging using Logic analyzers
- Lab Sessions with lots of Code implementations



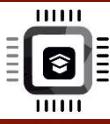
Lab Sessions

I Believe in the quote by

“TALK IS CHEAP, SHOW ME THE CODE”

– Linus Torvalds

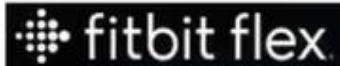
Will Code Lots of Assignments to Understand
things better !!



So, Fasten your Seat Belt
Lets Get Started !!!

Motivation to learn ARM-Cortex-M Processors .

- ▶ It's an embedded processor used in most of the Microcontrollers you see today and used in wide ranges of Embedded applications.
 - ▶ Battery powered devices like health monitoring and fitness tracking applications , Medical Meters,etc
 - ▶ Automotive applications
 - ▶ IOT Applications
 - ▶ Mobile and Home Appliances
 - ▶ Home/Building automation
 - ▶ Toys and Consumer products
 - ▶ PC and Mobile Accessories
 - ▶ Test and measurement devices



Wear + Life

Water-resistant wristband with a 5-day battery life
(Battery life and charge cycles vary with use,
settings and other factors. Actual results will vary)



Wireless Syncing

Sync stats wirelessly & automatically
to computers and leading
smartphones



Embedded

Academy

Microcontroller : STM32L151C6 by STMicro

Processor : ARM Cortex M3

Application Type : Ultra Low power

TomTom Spark 3 GPS Multisport Fitness Watch



Copyright © 2024 Fastbit Embedded Brain Academy

Microcontroller : SAMSx (Atmel SMART ARM Cortex-M7 Microcontrollers)

Processor : ARM Cortex M7

Application Type : Ultra Low power

Motivation to learn ARM-Cortex-M Processors .

- ▶ Most of the famous MCU manufactures produce microcontrollers based on ARM Cortex M processors.
 - ▶ TI (Low power battery based applications)
 - ▶ STMicro (High + medium + low performance MCUs)
 - ▶ Toshiba (measuring Equipment's + metering)
 - ▶ NXP
 - ▶ Microchip
 - ▶ Broadcom (Wireless Connectivity , IOT)
 - ▶ There are many....



Cortex
M3/M4

Micro-Controller



Copyright © 2024 Fast Bit Lab



Added Brain Academy



Many More . . .



Motivation to learn ARM-Cortex-M Processors .

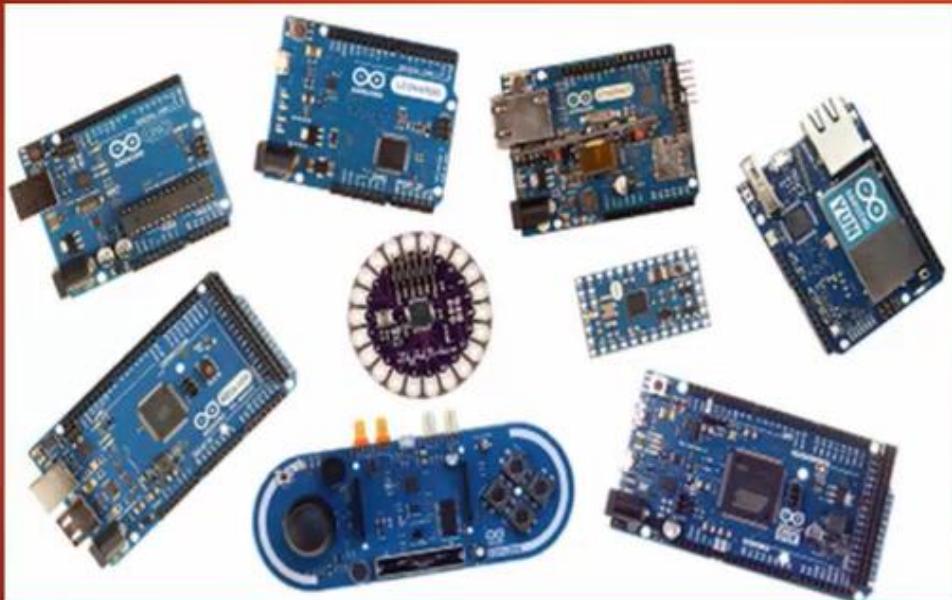
- ▶ Most of the manufacturers love to use ARM Cortex M processor in their design because of its minimal cost , minimal power and minimal silicon area.
- ▶ Its a 32 bit processor which will surely boost the computational performance of an application and it comes with almost same the price of 8bit or 16 bit traditional processor
- ▶ You can use this processor based MCUs in ultra low power to high performance based applications.
- ▶ Processor is customizable to include Floating point unit, DSP unit, MPU , etc.
- ▶ Very powerful and easy to use interrupt controller which supports 240 external interrupts

Motivation to learn ARM-Cortex-M Processors .

- ▶ RTOS friendly . That means it provides some exceptions , processor operational modes and access level configuration which helps to develop secured RTOS related applications
- ▶ Its instruction set is rich and memory efficient. It uses Thumb instruction set which is collection of 16bit as well as 32 bit instructions. Cortex M processor cannot execute the ARM instruction set instructions. It uses Thumb instruction set which gives the same 32bit ARM instruction performance but in 16bit format.
- ▶ ARM Provides lots of documentations to learn more about this processor.

Major Competitors

- ▶ 1. AVR based Microcontrollers (8/16/32 bit) by Microchip(Atmel)

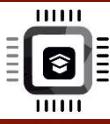


Most of the Arduino boards use Microcontrollers which are based on AVR Processor Core Of 8bit/16bit/32 bit architecture

Major Competitors

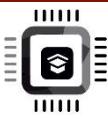
- ▶ 2. MSP430 Microcontrollers (16bit) by TI



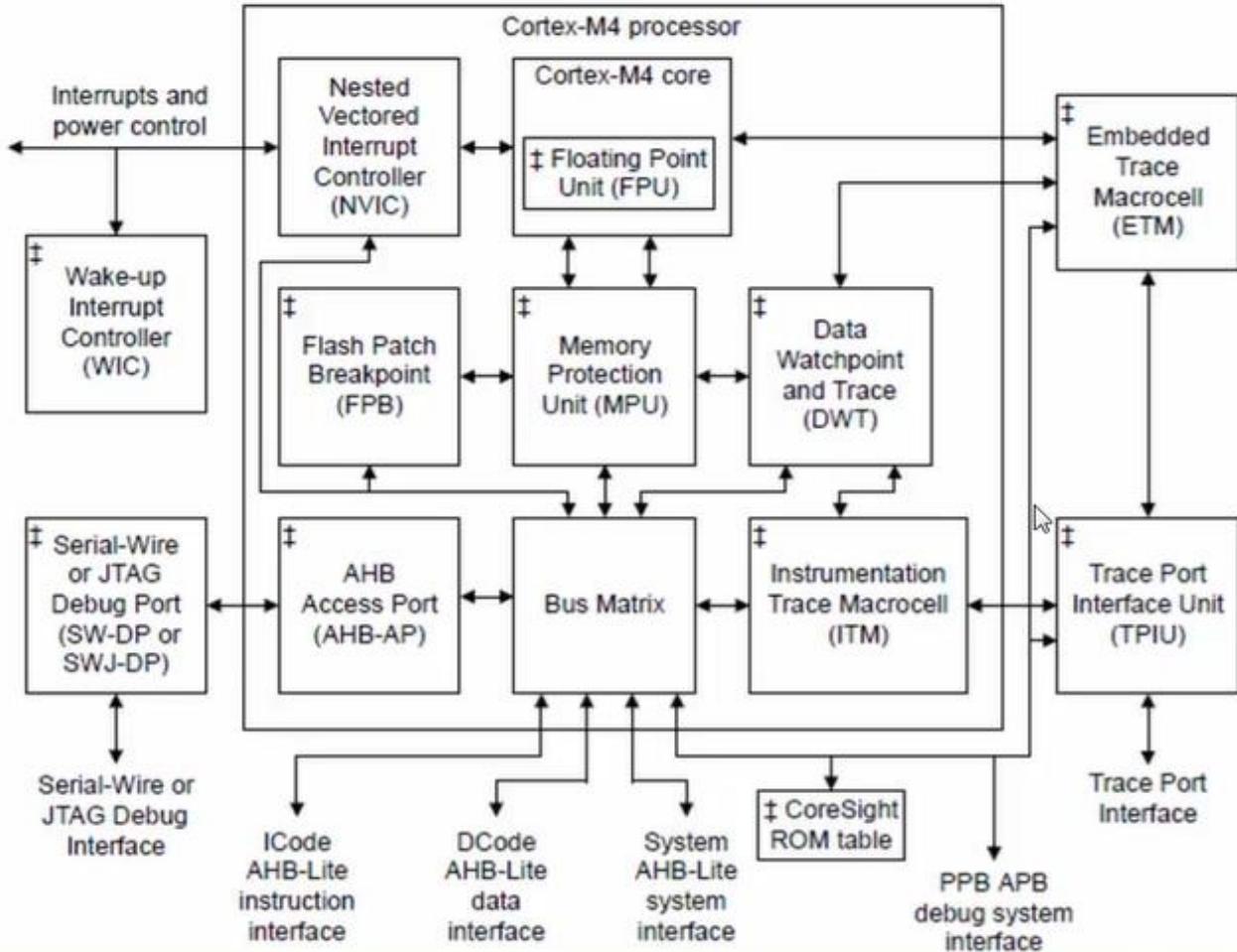


Motivation to learn ARM-Cortex-M3/M4

- It's really cheap
- High speed
- Less Power
- More Code Density
- Better Interrupt Management
- Many more



Block diagram showing the structure of the Cortex-M4 processor.



Installing an Integrated Development Environment (IDE)

- IDE is a software that contains all the essential tools to develop, compile, link, and debug your code.
- In some cases, you may have to integrate compiler and debugger tools to the IDE manually
- Throughout this course we will be using Eclipse-based STM32CubeIDE which is developed by ST Microelectronics to write, compile, debug applications on STM32 ARM-based microcontrollers.
- STM32CubeIDE is an eclipse IDE with STM32 related customization

Embedded Target

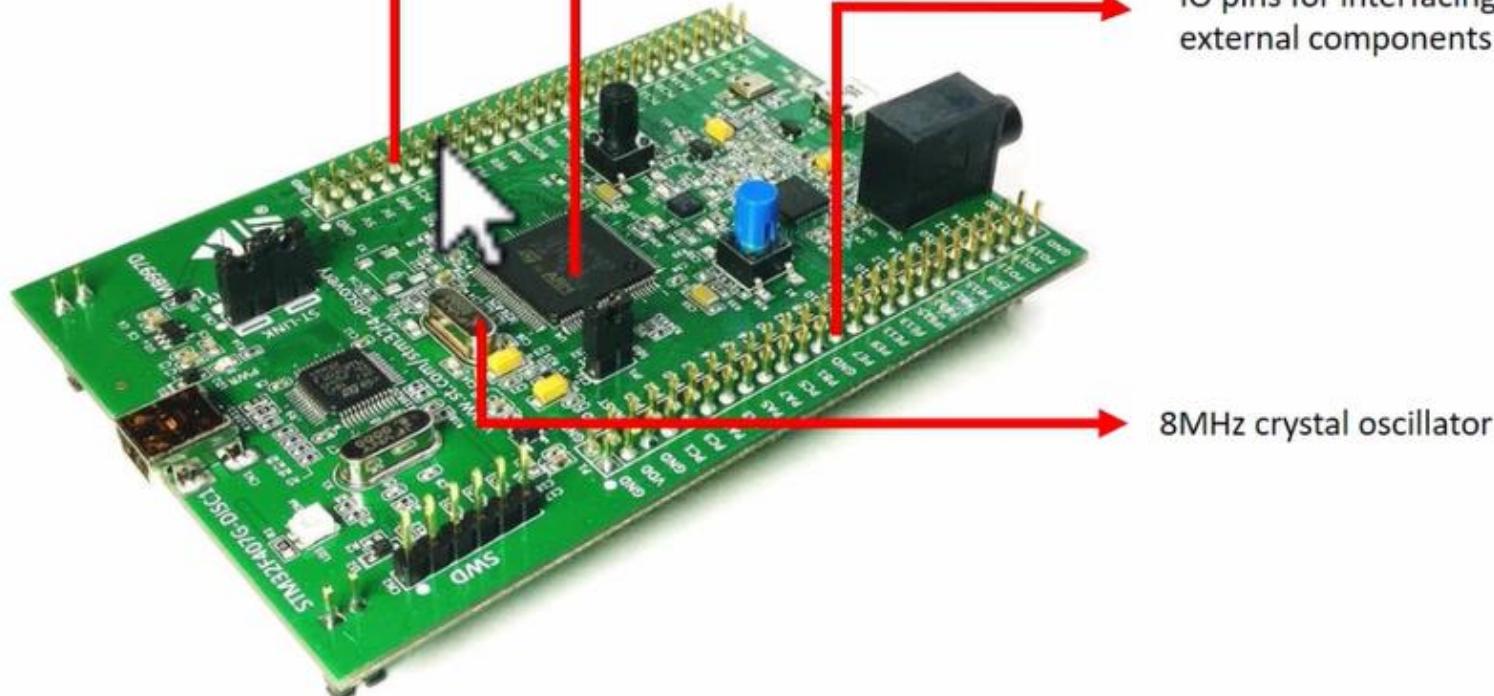
- In this course, the STM32F407VGT6 MCU based DISCOVERY Board is used as the target



IO pins for interfacing
external components

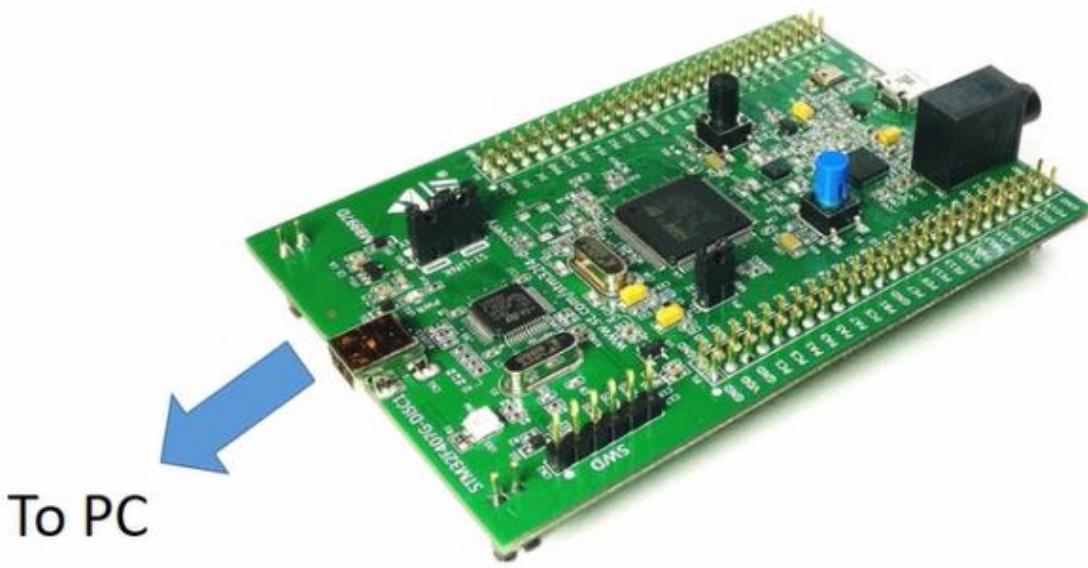
STM32F407VGT6 32 bit Microcontroller
Processor : ARM® Cortex® -M4 with FPU

IO pins for interfacing
external components



8MHz crystal oscillator

Board Connection to PC



USB Cable, Type A Plug to Mini Type B Plug
(This cable doesn't come with board)

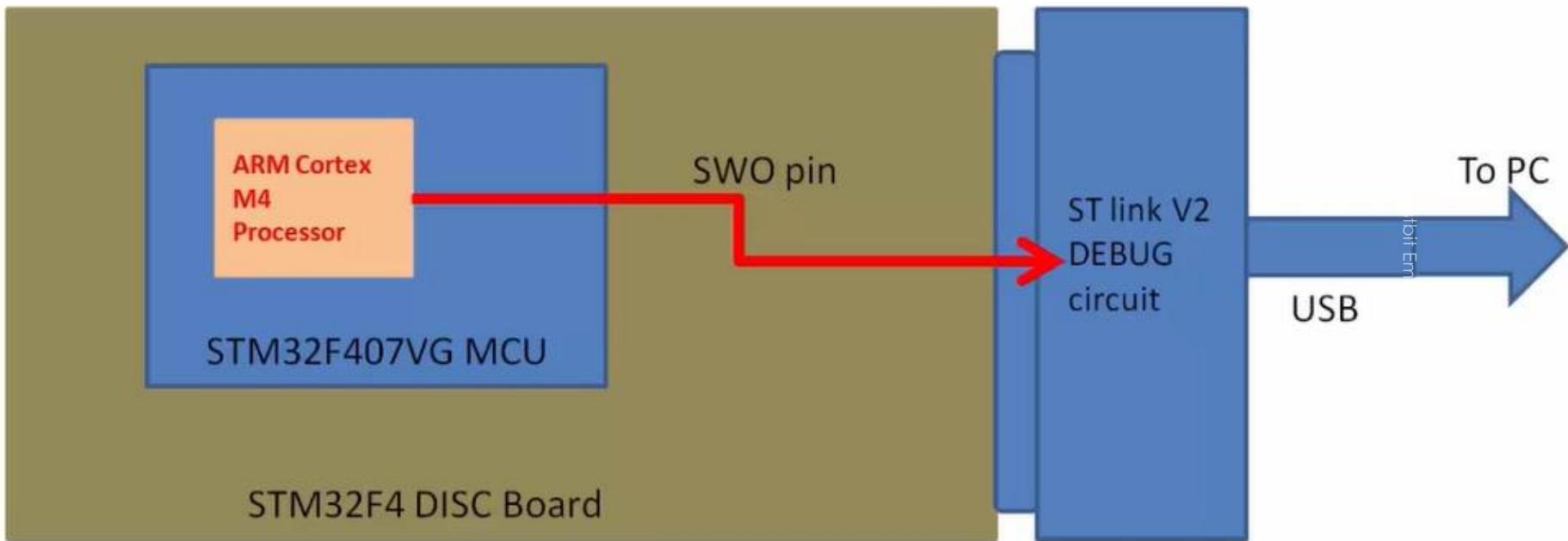


Embedded – ‘Hello World’

How are we going to see the message output without a display device connected to our board?

Using printf outputs on ARM Cortex M3/M4/M7 based MCUs

- This discussion is only applicable to MCUs based on ARM Cortex M3/M4/M7 or higher processors.
- printf works over SWO pin(Serial Wire Output) of SWD interface



ARM Cortex M4 Processor



ITM unit

Debug connector(SWD)

SWD(Serial Wire Debug)

2 pin (debug) + 1 pin (Trace)

Instrumentation Trace Macrocell Unit

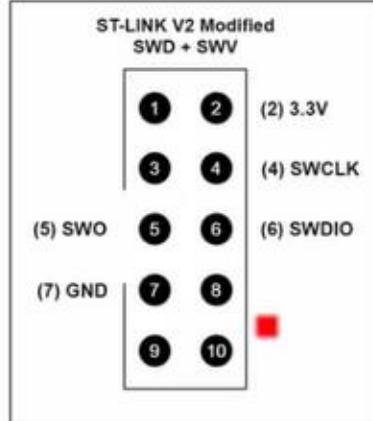
The ITM is an optional application-driven trace source that supports printf style debugging to trace operating system and application events, and generates diagnostic system information

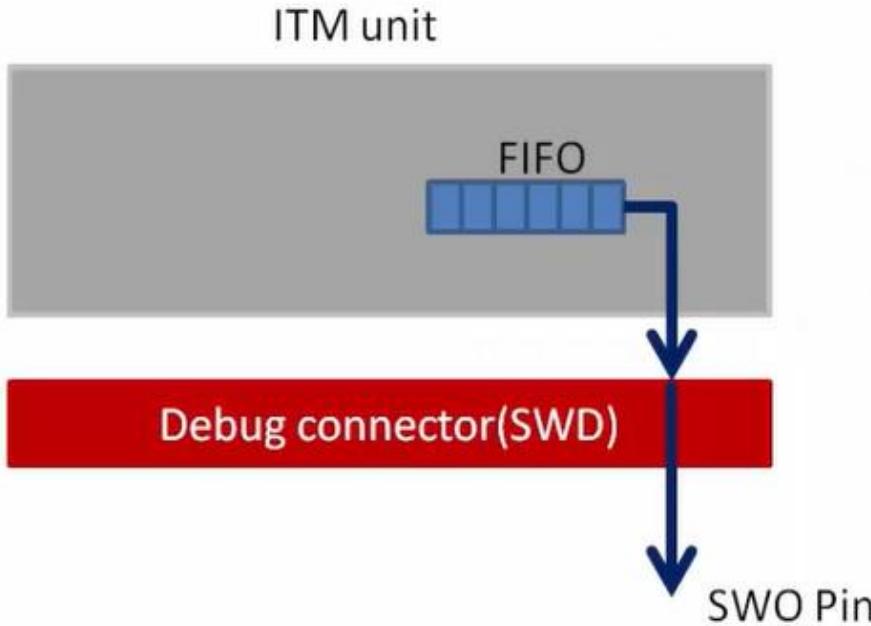
SWD

- ✓ Serial Wire Debug (SWD) is a two-wire protocol for accessing the ARM debug interface
- ✓ It is part of the ARM Debug Interface Specification v5 and is an alternative to JTAG
- ✓ The physical layer of SWD consists of two lines
 - SWDIO: a bidirectional data line
 - SWCLK: a clock driven by the host
- ✓ By using SWD interface should be able to program MCUs internal flash , you can access memory regions , add breakpoints, stop/run CPU.
- ✓ The other good thing about SWD is you can use the serial wire viewer for your printf statements for debugging.

SWD and JTAG

JTAG was the traditional mechanism for debug connections for ARM7/9 family, but with the Cortex-M family, ARM introduced the Serial Wire Debug (SWD) Interface. SWD is designed to reduce the pin count required for debug from the 4 used by JTAG (excluding GND) down to 2. In addition, SWD interface provides one more pin called SWO(Serial Wire Output) which is used for Single Wire Viewing (SWV), which is a low cost tracing technology





This SWO pin is connected to ST link circuitry of the board and can be captured using our debug software (IDE)

Printf implementation in std. library

```
printf( )
```

```
{  
    __write();  
}
```

Your project

```
printf();
```

Your project

```
__write()  
{  
    ITM_sendChar();  
    LCD_sendChar();  
}
```

OpenOCD Debugger and Semi-hosting

- Set the linker arguments
 - `-specs=rdimon.specs -lc -lrdimon`
- Add semi-hosting run command
 - `monitor arm semihosting enable`
- Add the below function call to main.c
 - `extern void initialise_monitor_handles(void);`
 - `initialise_monitor_handles(void);`

Caller

```
void fun1(void)  
{
```

```
    fun1_ins_1;
```

```
    fun2();
```

```
    fun1_ins_2;
```

```
}
```

PC jumps to fun2 address

Callee

```
void fun2(void)  
{
```

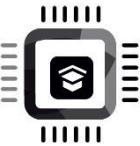
```
    fun2_ins_1;  
    fun2_ins_2;
```

```
}
```

PC jumps back to resume fun1

LR = return address (address of
next instruction)

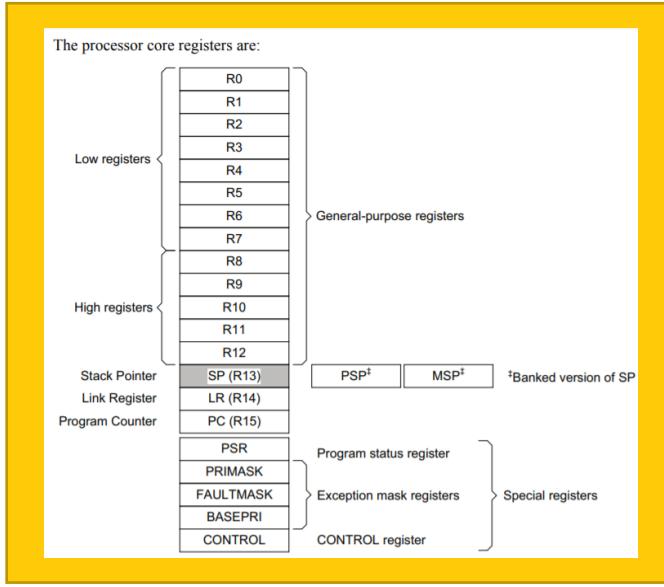
Function return
PC = LR



Operation modes and access levels

- Operational Modes of the Cortex Mx Processor:
Demonstration
- Access Levels of the Cortex Mx Processor:
Demonstration

Non-memory mapped registers

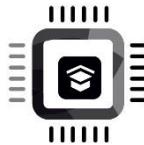


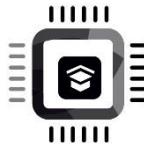
- The registers do not have unique addresses to access them. Hence they are not part of the processor memory map.
- You cannot access these registers in a 'C' program using address dereferencing.
- To access these registers, you have to use assembly instructions.

Memory mapped registers

Registers of the processor specific peripherals (NVIC,MPU,SCB,DEBUG,etc)

Registers of the Microcontroller specific peripherals (RTC,I2C,TIMER,CAN,USB,etc)



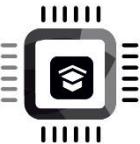


ARM GCC inline assembly code usage

- Inline assembly code is used to write pure assembly code inside a ‘C’ program.
- GCC inline assembly code syntax as shown below

Assembly instruction : MOV R0,R1

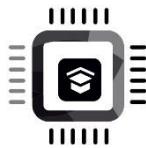
Inline assembly statement : __asm volatile("MOV R0,R1");



- LDR R0,[R1]
- LDR R1,[R2]
- ADD R1,R0
- STR R1,[R3]

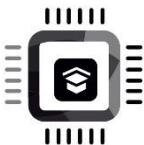
```
void fun_add(void)
{
    __asm volatile ("LDR R0,[R1]");
    __asm volatile ("LDR R1,[R2]");
    __asm volatile ("ADD R1,R0");
    __asm volatile ("STR R1,[R3]");

    __asm volatile (
        "LDR R0,[R1]\n\t"
        "LDR R1,[R2]\n\t"
        "ADD R1,R0\n\t"
        "STR R1,[R3]\n\t"
    );
}
```



'C' variable and inline assembly

- Move the content of 'C' variable 'data' to ARM register R0
- Move the content of the CONTROL register to the 'C' variable "control_reg."



General form of an inline assembly statement

```
_asm volatile (code : output operand list : input operand list : clobber list);
```

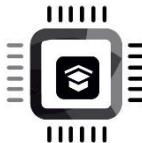
Volatile attribute to the asm statement to instruct the compiler not to optimize the assembler code

Clobber list is mainly used to tell the compiler about modifications done by the assembler code

Assembly mnemonic defined as a single string

A list of output operands, separated by commas.

A list of input operands, separated by commas.



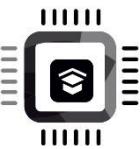
General form of an inline assembler statement

```
_asm volatile (code : output operand list : input operand list : clobber list);
```



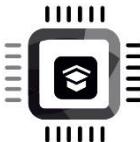
```
__asm volatile("MOV R0,R1");
```

```
__asm volatile("MOV R0,R1": :  
:);
```



Exercise

Load 2 values from memory , add them and store the result back to the memory using inline assembly statements.



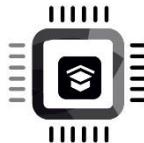
Input/output operands and Constraint string

Each input and output operand is described by a constraint string followed by a C expression in parentheses.

Input/output operand format :

“<Constraint string>” (< ‘C’ expression>)

Constraint string = constraint character + constraint modifier

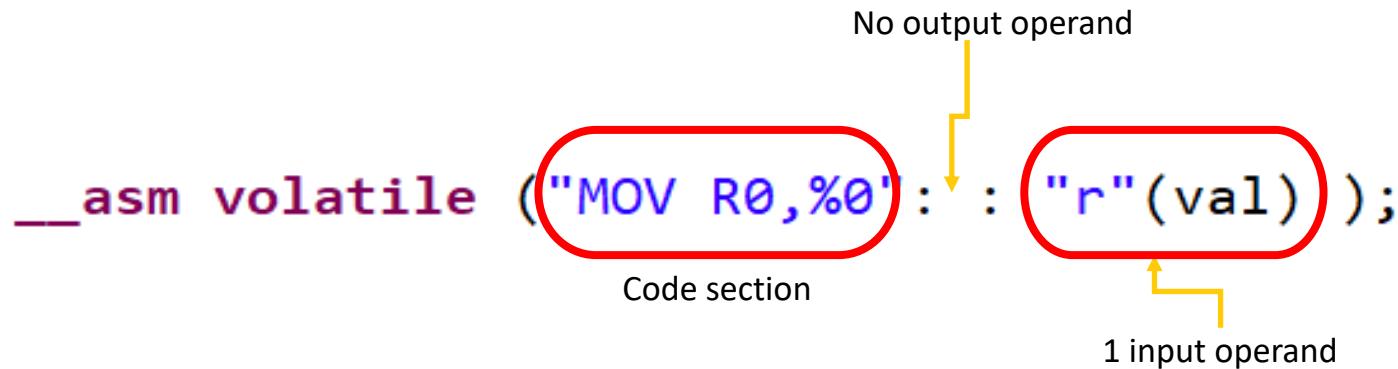


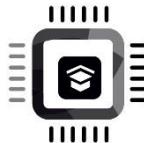
Example 1 : Move the content of 'C' variable 'val' to ARM register R0

Instruction ==> MOV

Source ==> a 'C' variable 'val' (INPUT)

Destination ==> R0 (ARM core register)





Example 1 : Move the content of 'C' variable 'val' to ARM register R0

Instruction ==> MOV

Source ==> a 'C' variable 'val' (INPUT)

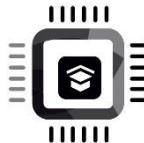
Destination ==> R0 (ARM core register)

```
_asm volatile ("MOV R0,%0": : "r"(val) );
```

No output operand

Code section

1 input operand



Example 1 : Move the content of 'C' variable 'val' to ARM register R0

Instruction ==> MOV

Source ==> a 'C' variable 'val' (INPUT)

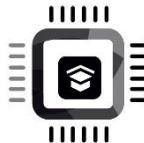
Destination ==> R0 (ARM core register)

```
_asm volatile ("MOV R0,%0": : "r"(val) );
```

No output operand

Code section

1 input operand



Example 1 : Move the content of 'C' variable 'val' to ARM register R0

Instruction ==> MOV

Source ==> a 'C' variable 'val' (INPUT)

Destination ==> R0 (ARM core register)

Constraint string
('r' is a constraint character)

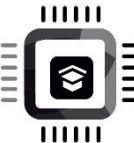
```
_asm volatile ("MOV R0,%0": : "r"(val));
```

Operand indexing using % sign followed by a digit

%0 refers to the first operand

%1 refers to the second and so forth

A 'C' variable



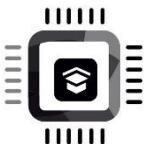
For ARM processors, GCC 4 provides the following constraints.

Constraint	Usage in ARM state	Usage in Thumb state
f	Floating point registers f0 .. f7	Not available
h	Not available	Registers r8..r15
G	Immediate floating point constant	Not available
H	Same as G, but negated	Not available
I	Immediate value in data processing instructions e.g. ORR R0, R0, #operand	Constant in the range 0 .. 255 e.g. SWI operand
J	Indexing constants -4095 .. 4095 e.g. LDR R1, [PC, #operand]	Constant in the range -255 .. -1 e.g. SUB R0, R0, #operand
K	Same as I, but inverted	Same as I, but shifted
L	Same as I, but negated	Constant in the range -7 .. 7 e.g. SUB R0, R1, #operand
I	Same as r	Registers r0..r7 e.g. PUSH operand
M	Constant in the range of 0 .. 32 or a power of 2 e.g. MOV R2, R1, ROR #operand	Constant that is a multiple of 4 in the range of 0 .. 1020 e.g. ADD R0, SP, #operand
m	Any valid memory address	
N	Not available	Constant in the range of 0 .. 31 e.g. LSL R0, R1, #operand
O	Not available	Constant that is a multiple of 4 in the range of -508 .. 508 e.g. ADD SP, #operand
r	General register r0 .. r15 e.g. SUB operand1, operand2, operand3	Not available
w	Vector floating point registers s0 .. s31	Not available
X	Any operand	

constraint character

constraint modifier

Modifier	Specifies
=	Write-only operand, usually used for all output operands
+	Read-write operand, must be listed as an output operand
&	A register that should be used for output only



Example 2 : Move the content of CONTROL register to 'C' variable "control_reg"

CONTROL register is a special register of the ARM core

To read CONTROL register you have to use MRS instruction

Instruction ==> MRS

Source ==> CONTROL register

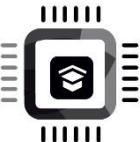
Destination ==> A 'C' variable "control_reg" (OUTPUT operand)

```
uint32_t control_reg;  
  
__asm volatile("MRS %0,CONTROL": "=r"(control_reg)::);
```

Code section

No input operand

1 output operand



Example 3 : Copy the content of 'C' variable var1 to var2

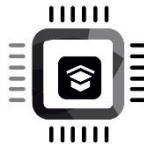
Instruction ==> MOV

Source ==> a 'C' variable 'var1' (INPUT operand)

Destination ==> a 'C' variable 'var2' (OUTPUT operand)

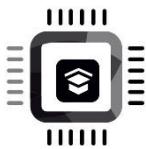
```
int var1=10, var2;
```

```
__asm ("MOV %0,%1": "=r"(var2): "r"(var1));
```

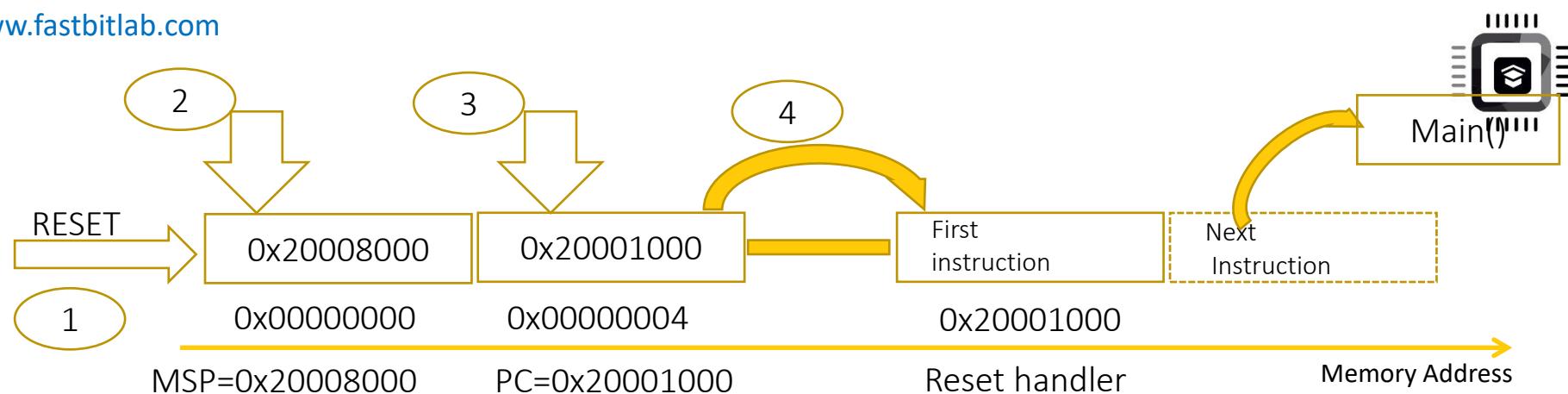


Example 4 : copy the contents of a pointer into another variable

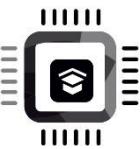
```
int p1, *p2;  
  
p2 = (int*)0x20000008;  
  
__asm volatile("LDR %0,[%1]": "=r"(p1): "r"(p2)); //p1 = *p2
```



Processor Reset Sequence

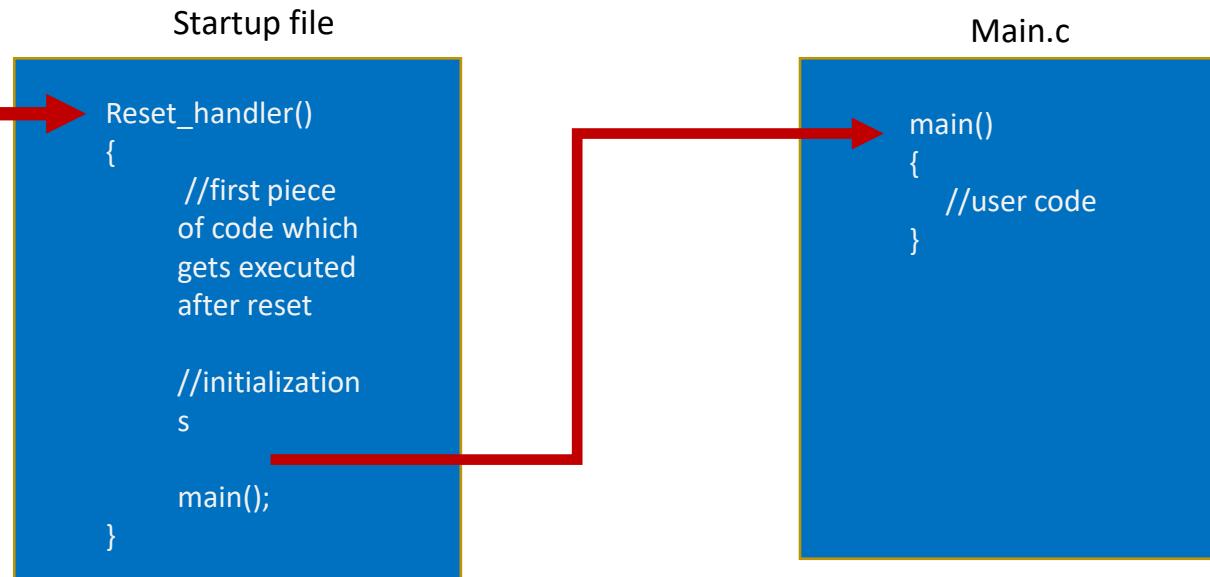


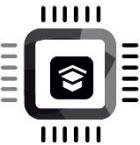
1. After reset, PC is loaded with the address 0x00000000
2. Processor read the value from 0x00000000 location in to MSP
3. Then processor reads the address of the reset handler from the location 0x00000004
4. Then it jumps to reset handler and start executing the instructions
5. Then you can call your `main()` function from reset handler.



Reset sequence

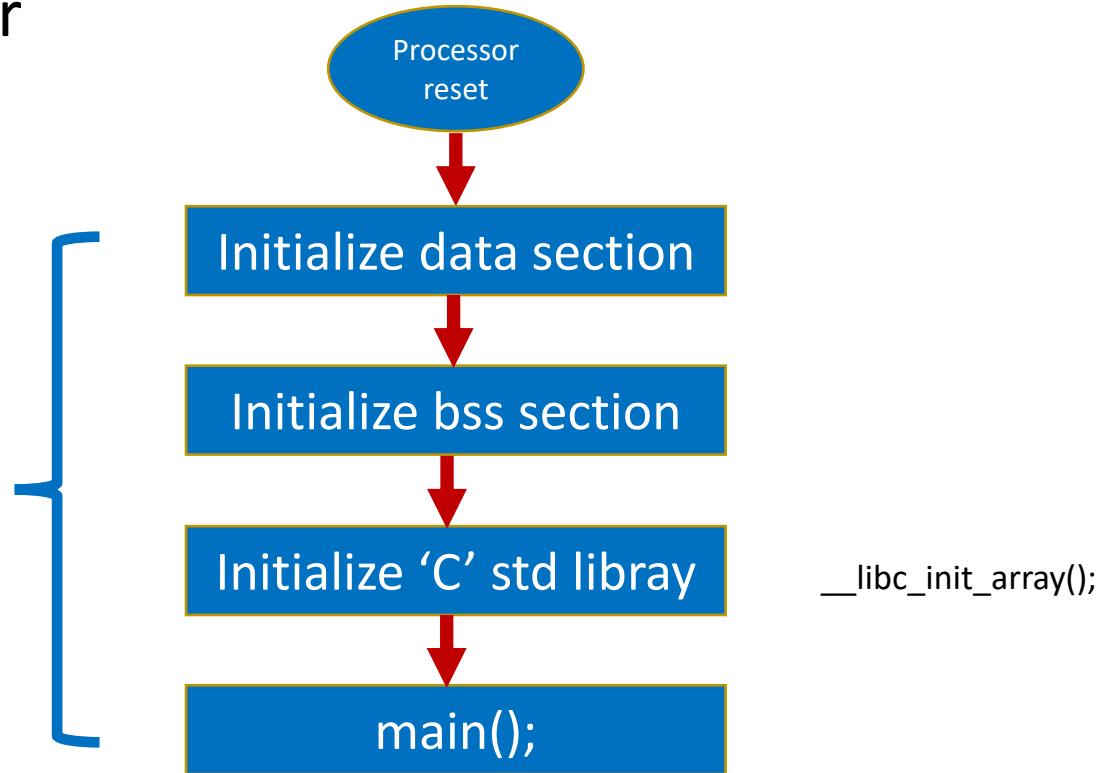
Vector table directs
processor to execute Reset
handler after reset

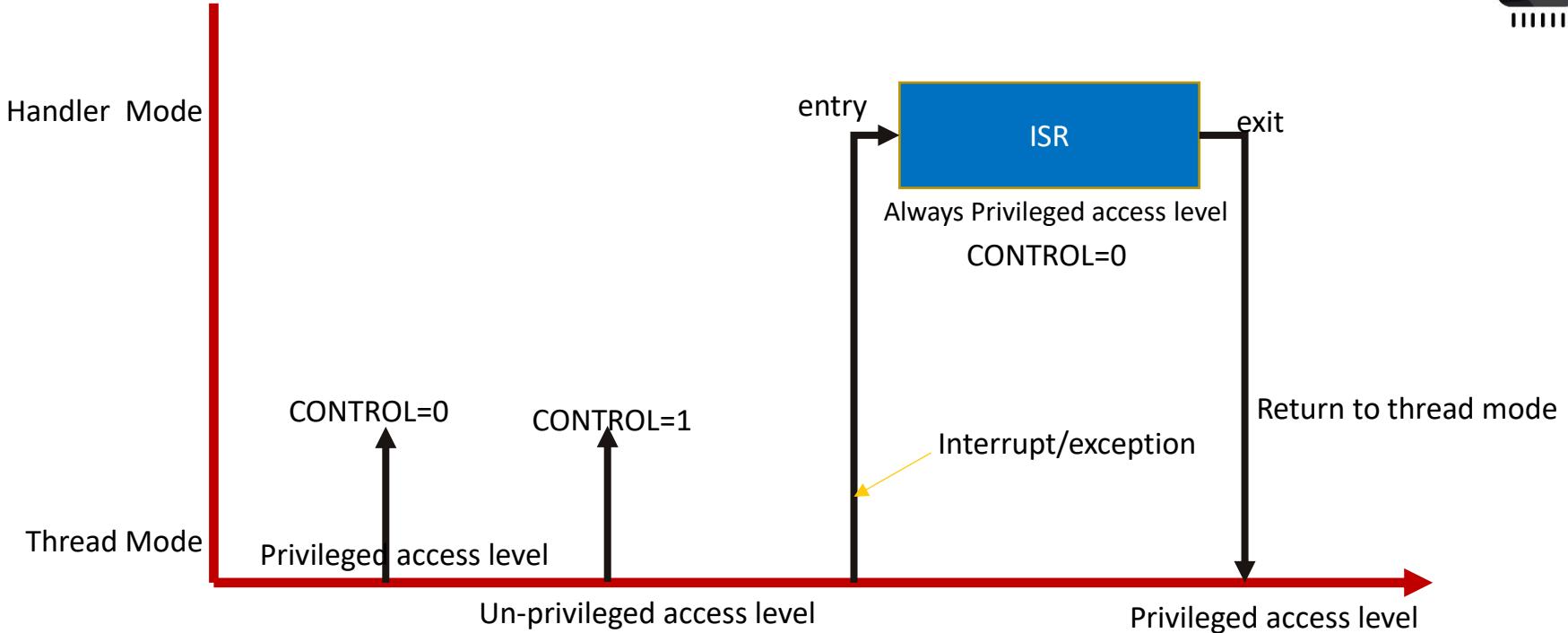
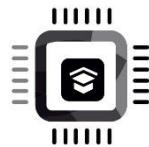




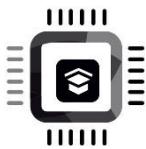
Reset handler

Reset handler responsibilities before calling main

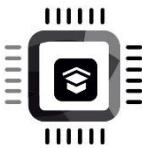




Thread mode code returning to Privileged access level from Unprivileged access level

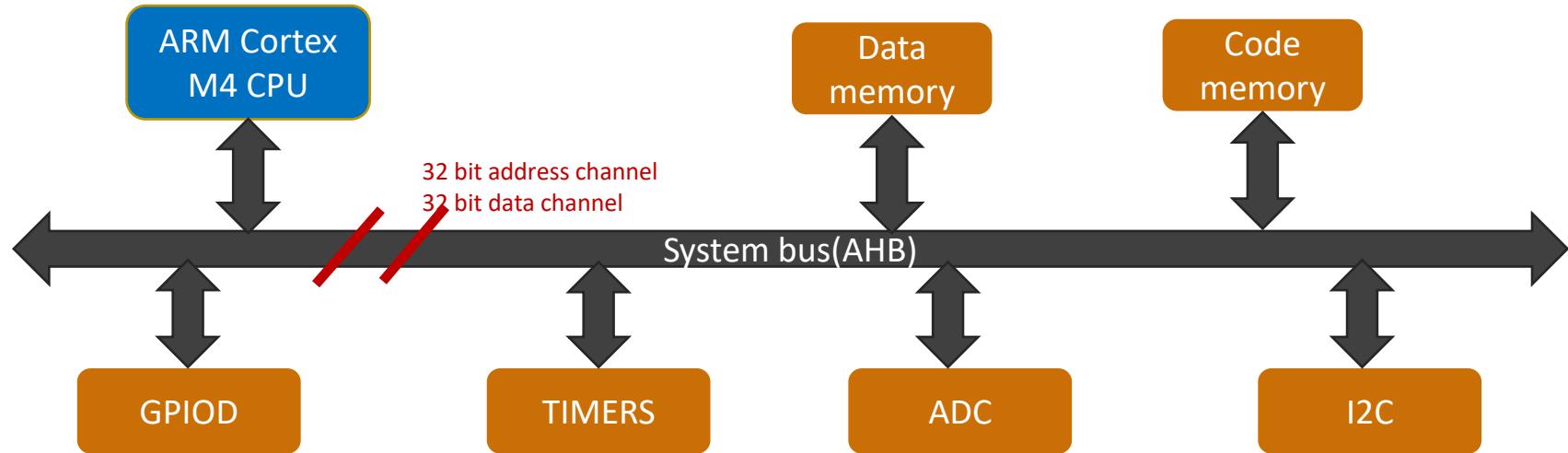
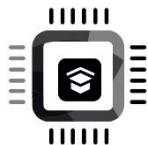


Memory Map and Bus interfaces



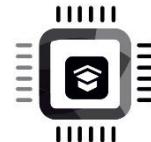
Memory Map of the processor

- Memory map explains mapping of different peripheral registers and memories in the processor addressable memory location range
- The processor, addressable memory location range, depends upon the size of the address bus.
- The mapping of different regions in the addressable memory location range is called ‘memory map’



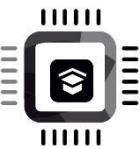
The processor has a fixed default memory map that provides up to 4GB of addressable memory

*To know exact processor, memory and peripheral interconnection refer MCU reference manuals or datasheet

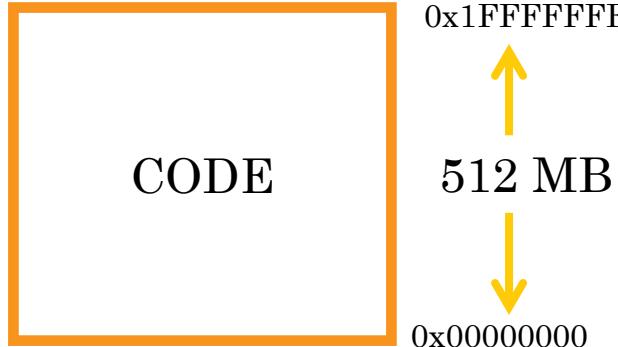


Memory map of the ARM Cortex Mx processor

Vendor-specific memory	511MB	0xFFFFFFFF
Private peripheral bus	1.0MB	0xE0100000 0xE00FFFFF 0xE0000000 0xDFFFFFFF
External device	1.0GB	0xA0000000 0x9FFFFFFF
External RAM	1.0GB	0x60000000 0x5FFFFFFF
Peripheral	0.5GB	0x40000000 0x3FFFFFFF
SRAM	0.5GB	0x20000000 0x1FFFFFFF
Code	0.5GB	0x00000000

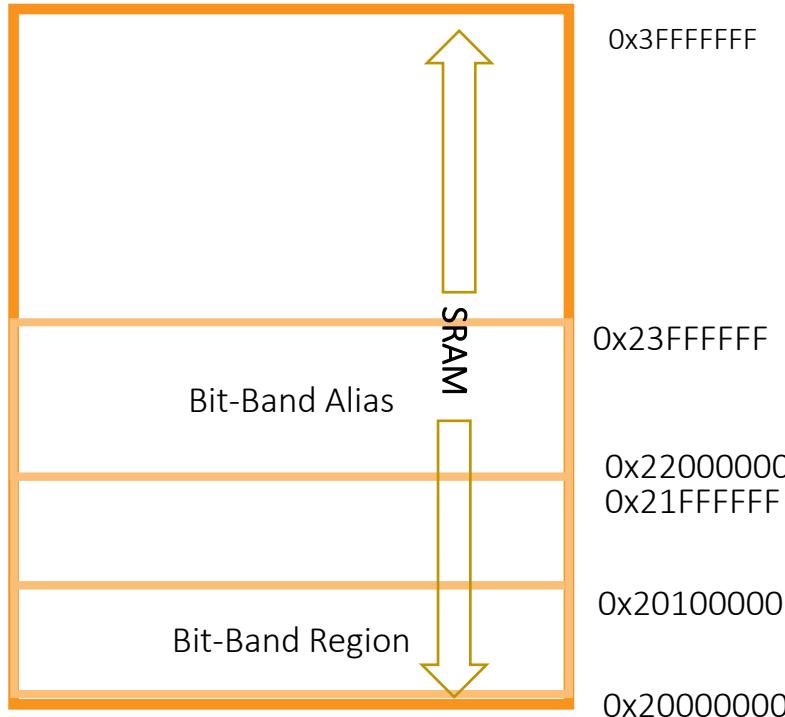
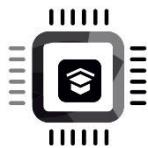


CODE Region



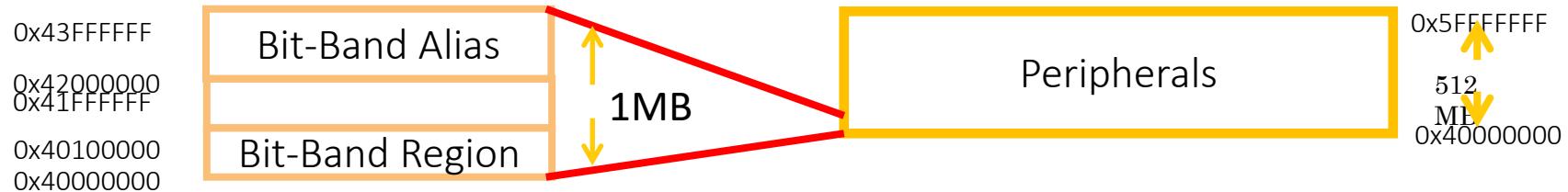
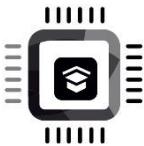
- ✓ This is the region where the MCU vendors should connect CODE memory.
- ✓ Different type of Code memories are: Embedded flash, ROM, OTP, EEPROM, etc
- ✓ Processor by default fetches vector table information from this region right after reset.

SRAM Region

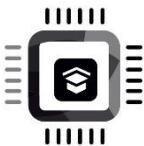


- The SRAM(Static-RAM) region is in the next 512 MB of memory space after the CODE region.
- It is primarily for connecting SRAM, mostly on-chip SRAM.
- The first 1 MB of the SRAM region is a bit addressable.
- You can also execute program code from this region

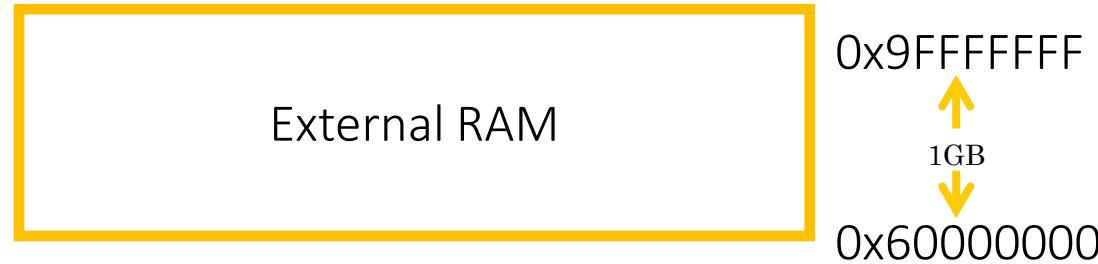
Peripherals Region



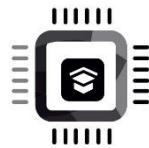
- The peripheral memory region also has the size of 512MB,
- Used mostly for on-chip peripherals.
- Like the SRAM region, the first 1MB of the peripheral region is bit addressable if the optional bit-band feature is included.
- This is an eXecute Never (XN) region
- Trying to execute code from this region will trigger fault exception



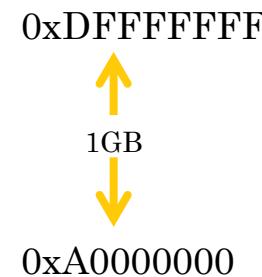
External RAM Region



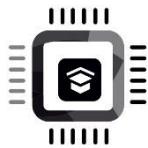
- This region is intended for either on-chip or off-chip memory
- you can execute code in this region.
- E.g., connecting external SDRAM



External Device Region



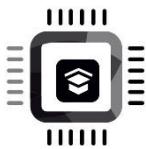
- This region is intended for external devices and/or shared memory
- This is a eXecute Never (XN) region



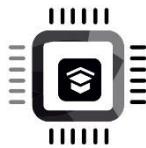
Private Peripheral Bus Region



- This region includes the NVIC, System timer, and system control block
- This is a eXecute Never (XN) region

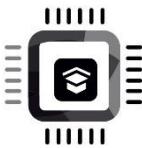


Bus Protocols and Bus interfaces



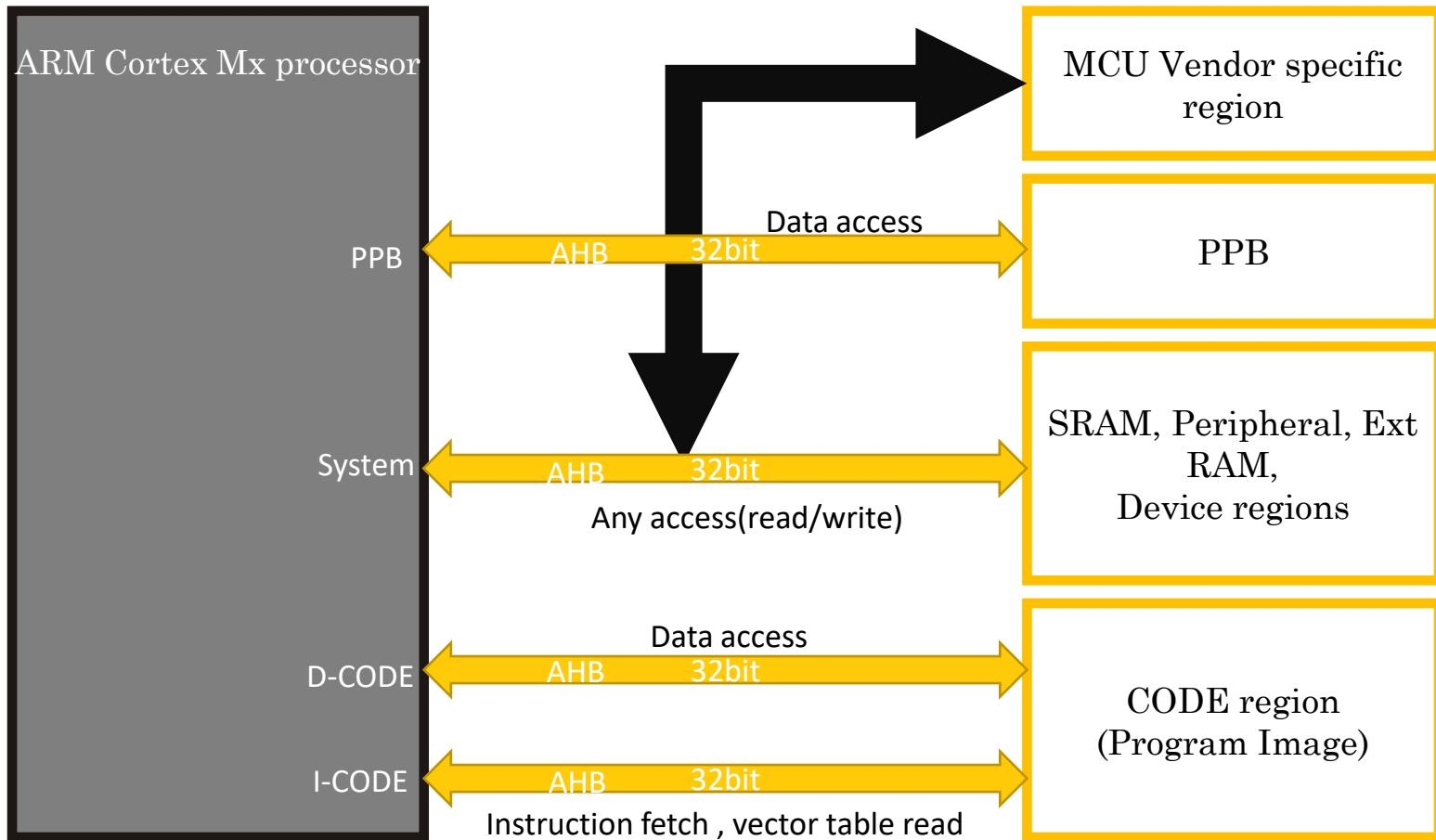
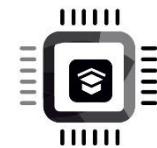
Bus Protocols and Bus interfaces

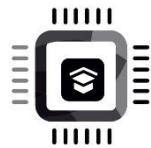
- In Cortex Mx processors the bus interfaces are based on advanced microcontroller bus architecture (AMBA) specification
- AMBA is a specification designed by ARM which governs standard for on-chip communication inside the system on chip
- AMBA specification supports several bus protocols.
 - ✓ AHB Lite(AMBA High-performance Bus)
 - ✓ APB (AMBA Peripheral Bus)



AHB and APB

- AHB Lite bus is mainly used for the main bus interfaces
- APB bus is used for PPB access and some on-chip peripheral access using an AHB-APB bridge
- AHB Lite bus majorly used for high-speed communication with peripherals that demand high operation speed.
- APB bus is used for low-speed communication compared to AHB. Most of the peripherals which don't require high operation speed are connected to this bus.

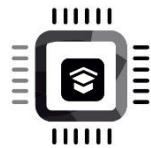




Private Peripheral Bus Region

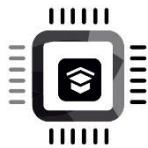


- This region includes the NVIC, System timer, and system control block
- This is a eXecute Never (XN) region



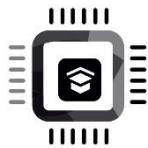
STM32F40xxx block diagram

- Refer to the datasheet of STM32F407VG MCU.



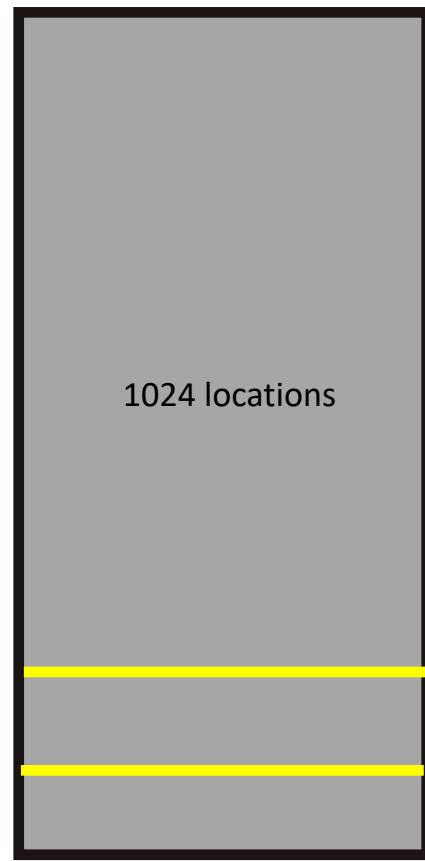
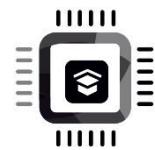
Bit band and bit band alias addresses

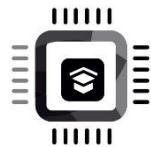
- The regions for SRAM and peripherals include optional bit-band regions.
- Bit-banding provides atomic operations to bit data.



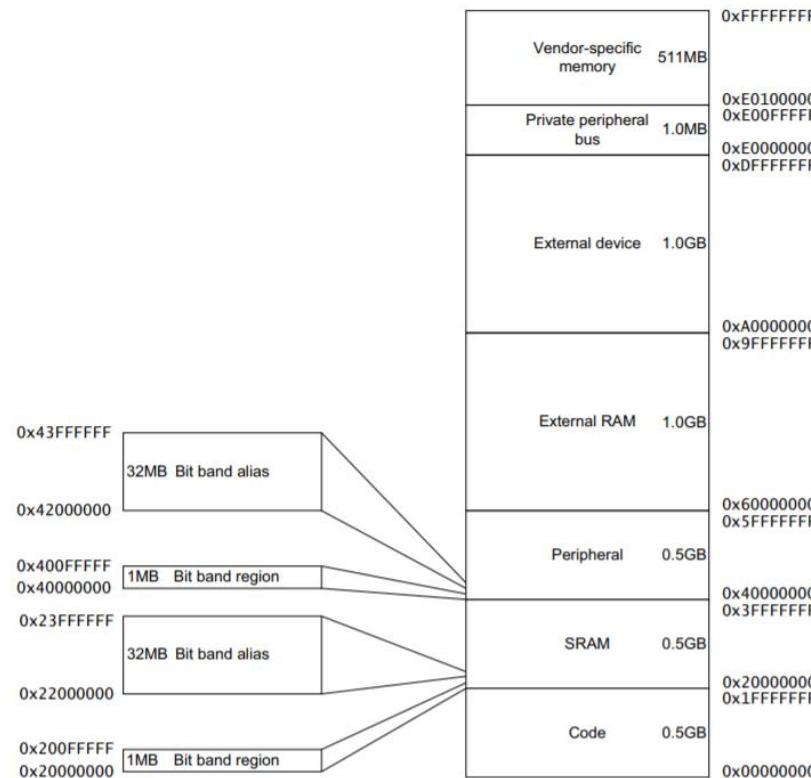
What is bit-banding ?

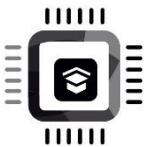
- It is the capability to address a single bit of a memory address.
- This feature is optional. i.e., MCU manufacturer supports it or many not support this feature. Refer to the reference manual





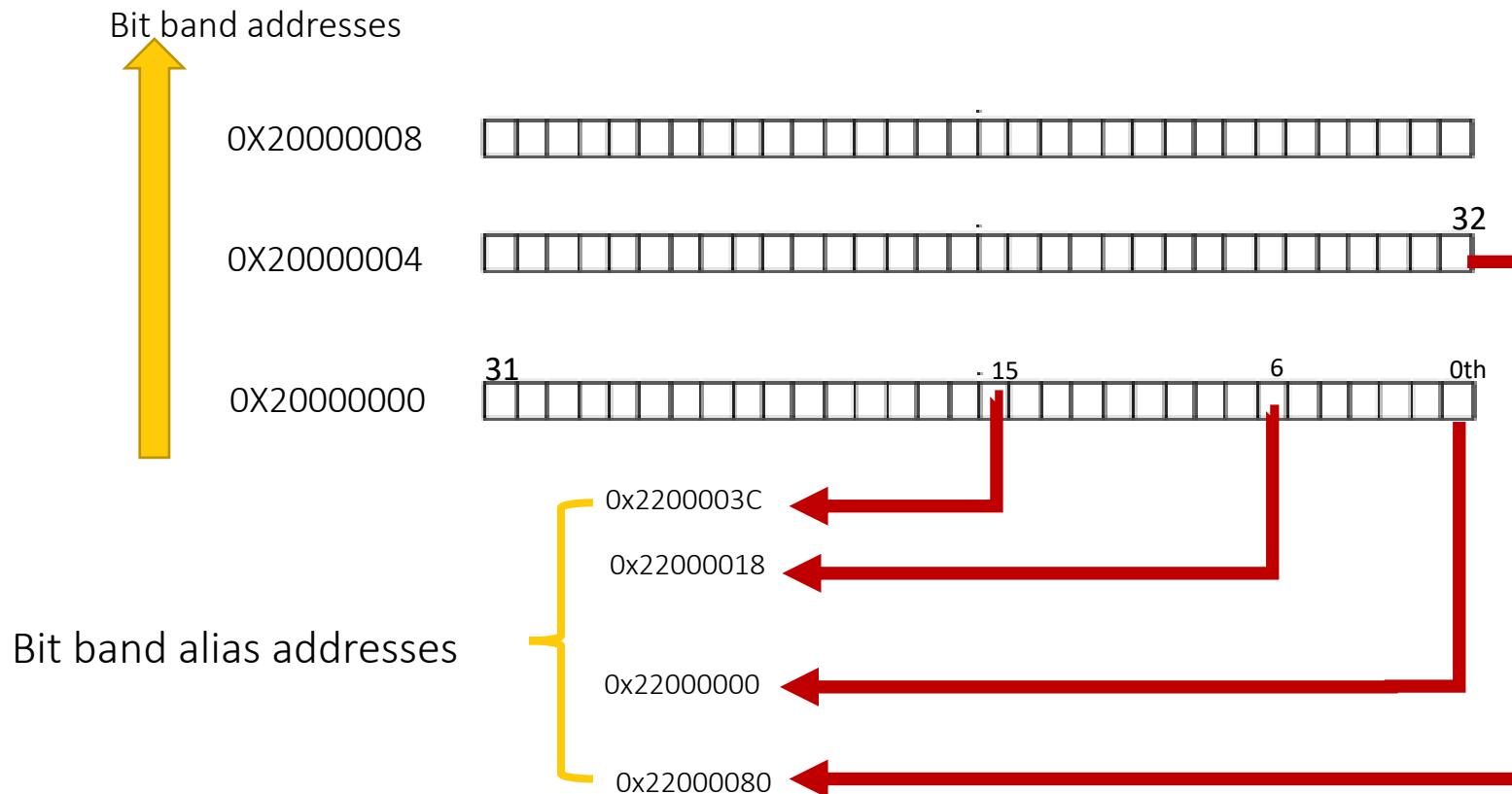
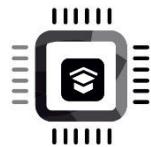
Bit band and bit band alias addresses



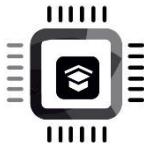


How this works ??



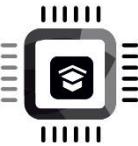


Exercise



Modify the content of the memory location 0x2000_0200 using usual and bit banding method and analyze the difference

- First store the value 0xff into the memory location 0x2000_0200
- Make the 7th-bit position of the value to 0



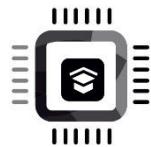
Calculation of bit band alias address

- Calculate the bit band alias address for given bit band memory address and bit position
- 7th bit position of the memory location 0x2000_0200 using its alias address

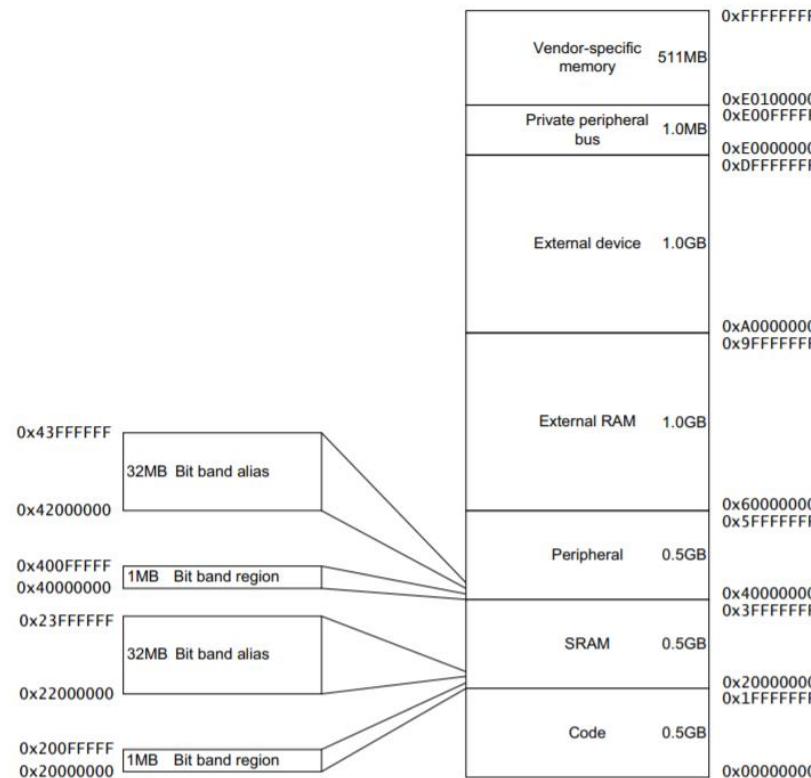
General formula :

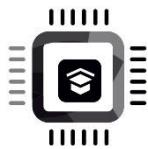
Alias address =

alias_base + (32 * (bit_band_memory_addr - bit_band_base)) + bit * 4

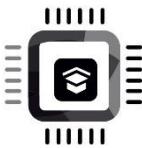


Bit band and bit band alias addresses



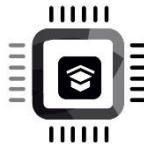


Stack Memory



Stack memory

- Stack memory is part of the main memory(Internal RAM or external RAM) reserved for the temporary storage of data (transient data)
- Mainly used during function, interrupt/exception handling
- Stack memory is accessed in last in first out fashion (LIFO)
- The stack can be accessed using PUSH and POP instructions or using any memory manipulation instructions(LD, STR)
- The stack is traced using a stack pointer(SP) register. PUSH and POP instructions affect(decrement or increment) stack pointer register (SP, R13)



Stack memory uses

1. The temporary storage of processor register values
2. The temporary storage of local variables of the function
3. During system exception or interrupt, stack memory will be used to save the context (some general-purpose registers, processor status register, return address) of the currently executing code

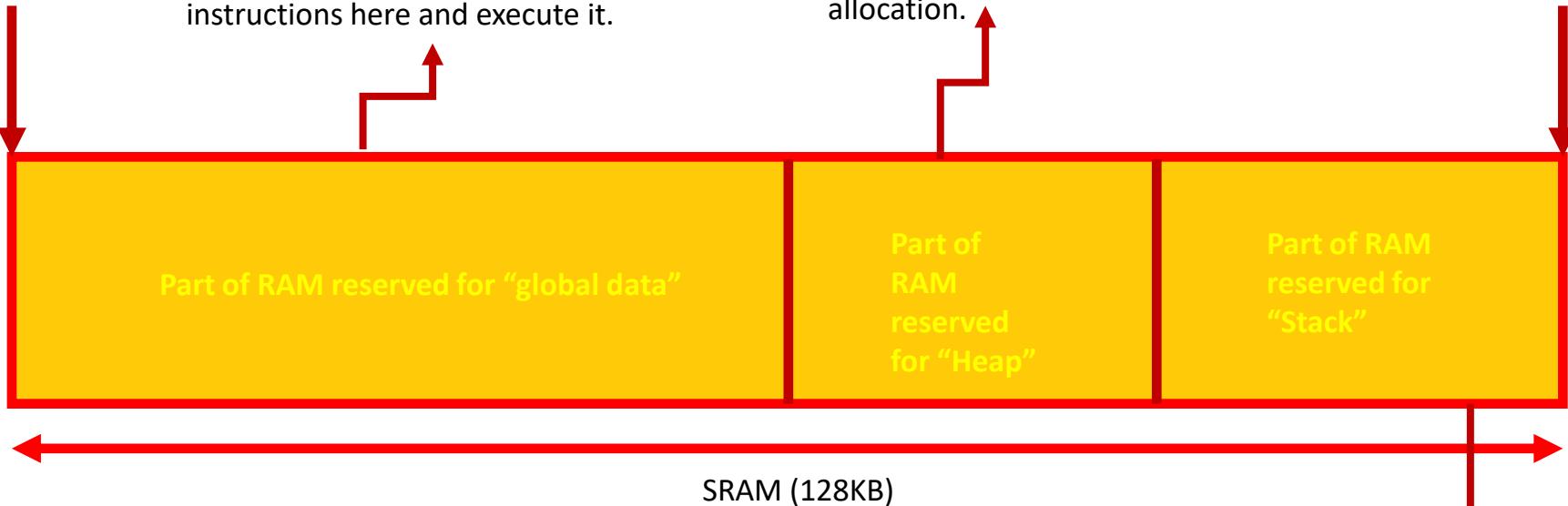


RAM_START

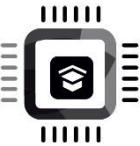
Will be utilized when the program contains global data and static local variables. Even you can store instructions here and execute it.

Will be utilized during dynamic memory allocation.

RAM_END

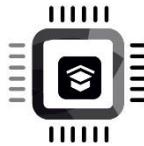


Will be utilized during function call to save temporary data, Temporary storage of local variables of the function, temporary storage of stack frames during interrupts and exceptions.



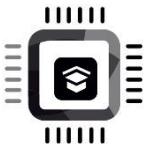
Stack operation model

In ARM Cortex Mx processor stack consumption model is **Full Descending(FD)**



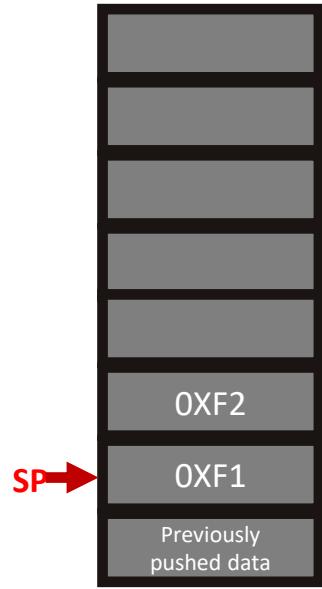
Different Stack operation models

- Full Ascending stack (FA)
- **Full Descending stack(FD)** (ARM Cortex Mx processors use this)
- Empty Ascending stack(EA)
- Empty Descending stack(ED)

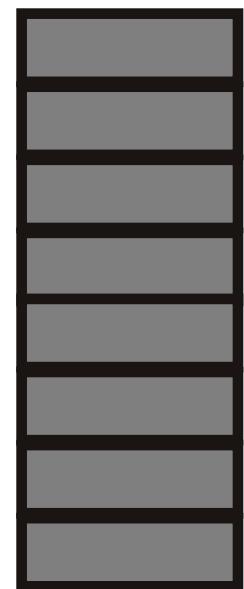
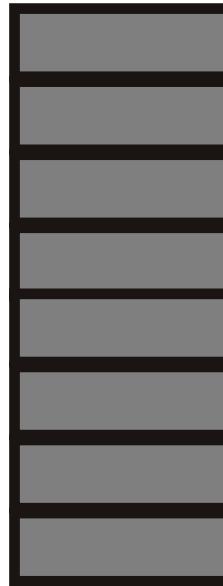


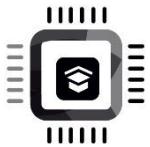
Different stack model operation

Increasing memory addresses ↑



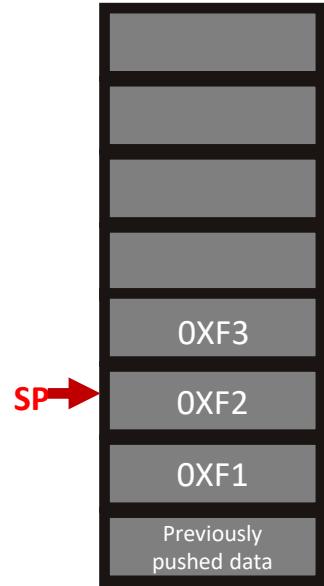
Full ascending stack



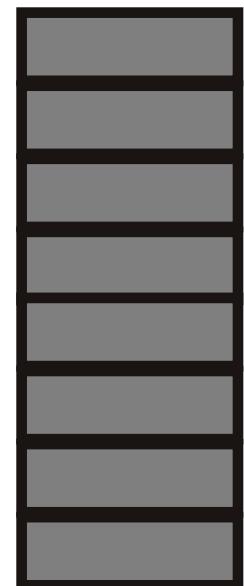


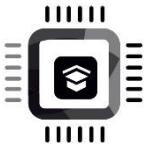
Different stack model operation

Increasing memory addresses ↑



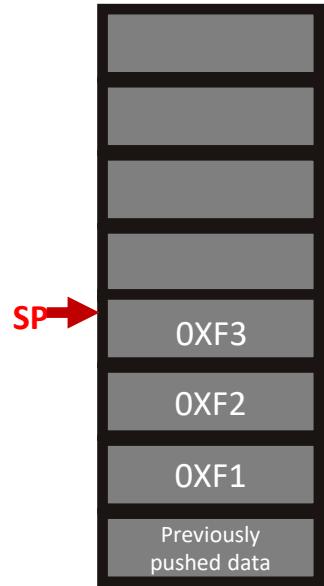
Full ascending stack



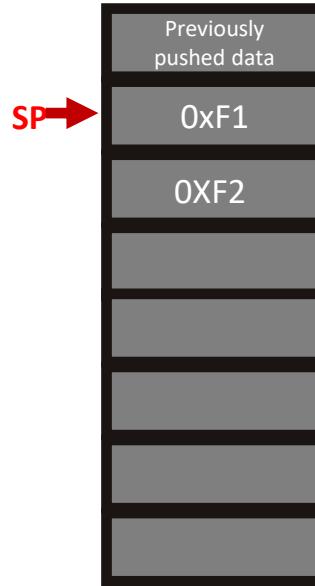


Different stack model operation

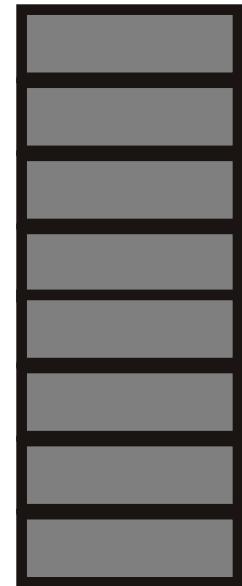
Increasing memory addresses ↑

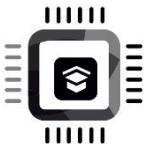


Full ascending stack



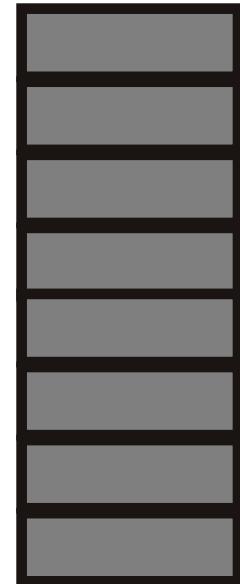
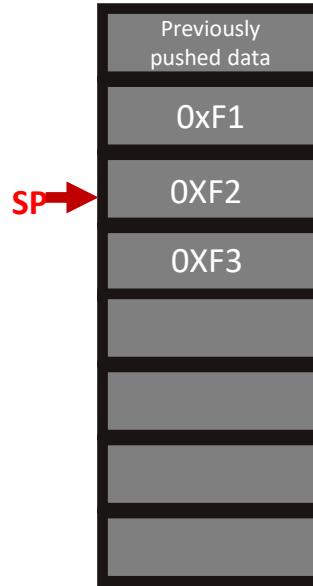
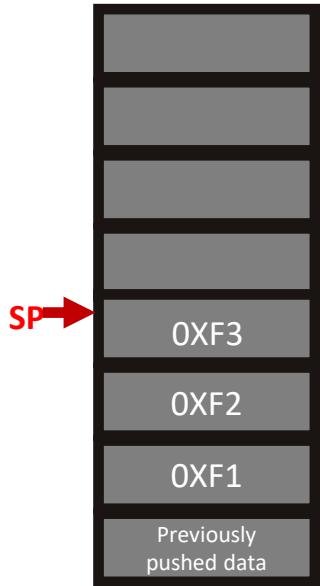
Full descending stack

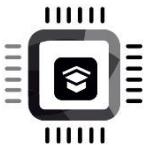




Different stack model operation

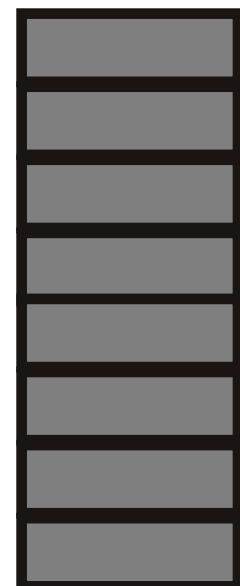
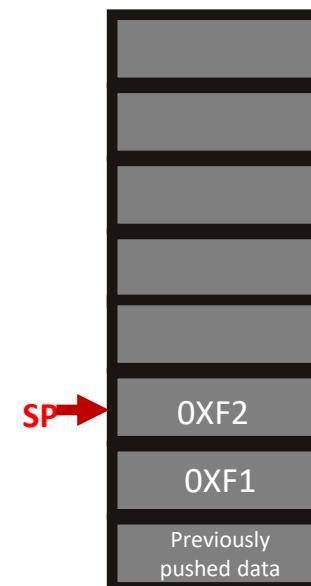
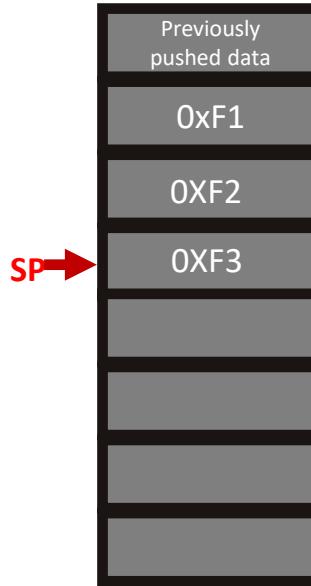
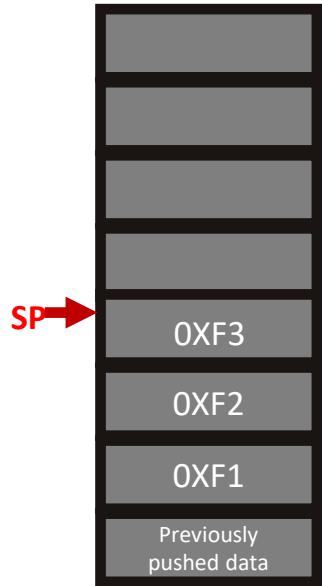
Increasing memory addresses ↑

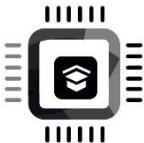




Different stack model operation

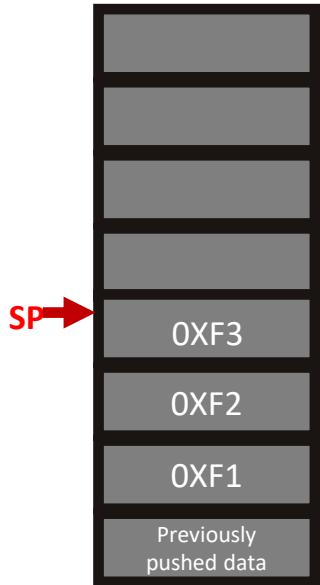
Increasing memory addresses ↑



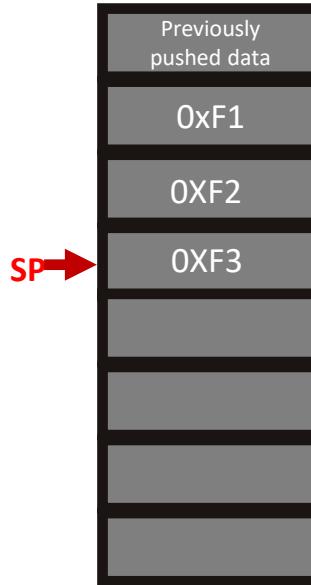


Different stack model operation

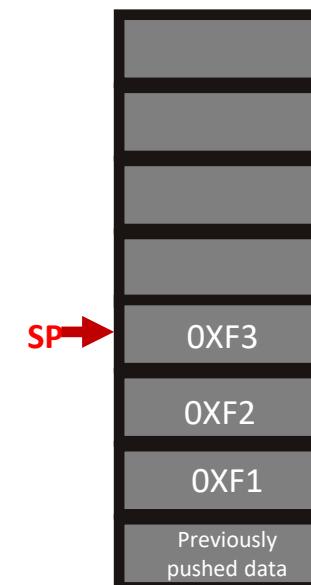
Increasing memory addresses ↑



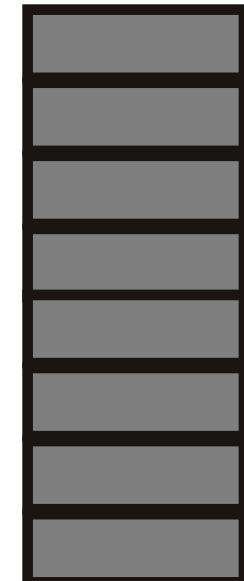
Full ascending stack

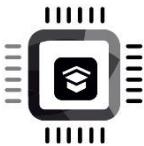


Full descending stack



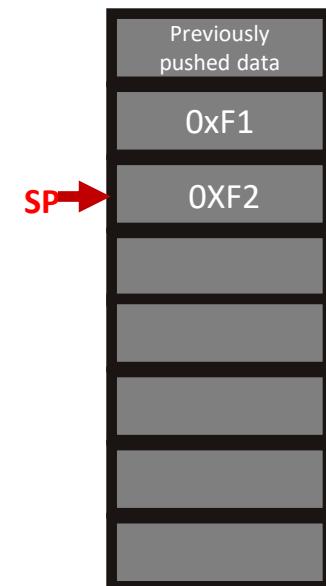
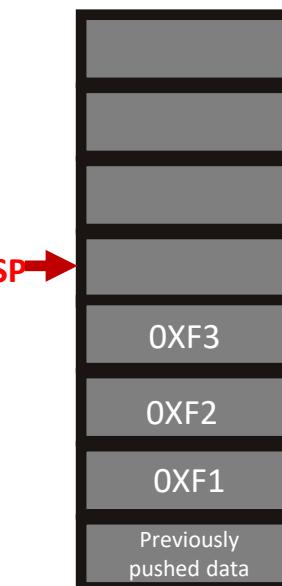
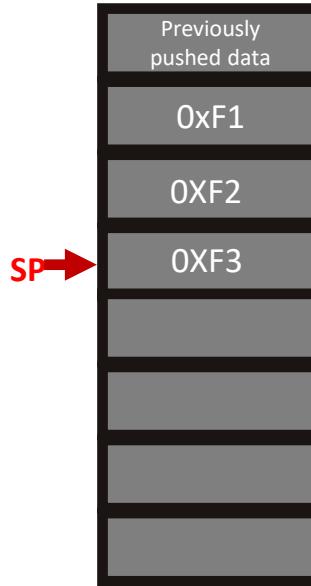
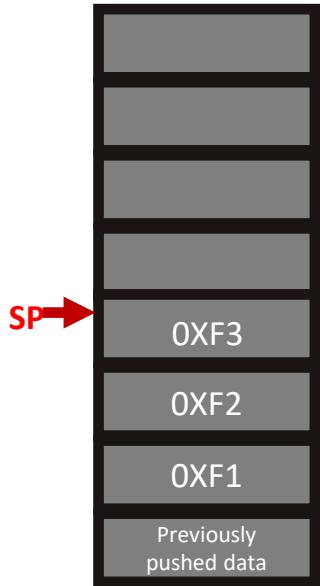
Empty ascending stack

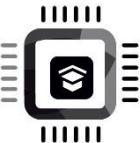




Different stack model operation

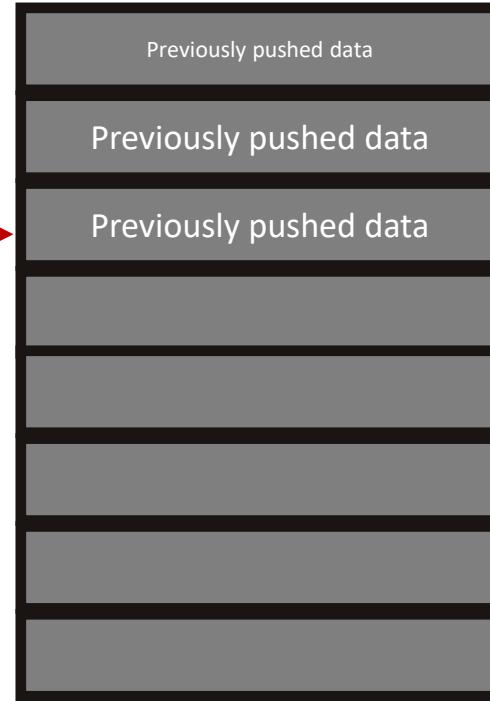
Increasing memory addresses ↑





PUSH LR
PUSH R0
PUSH R1
POP R2
POP R3
POP PC

SP →



Increasing memory addresses ↑

Core registers of the processor

R0 = 0xFFBC

R1 = 0xAB11

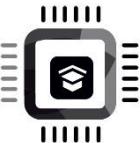
LR = 0x800211CD

R2=0

R3=5

PC=0x80010000

Stack operation in ARM Cortex Mx processor (FD)



PUSH LR

PUSH R0

PUSH R1

POP R2

POP R3

POP PC

Core registers of the processor

R0 = 0xFFBC

R1 = 0xAB11

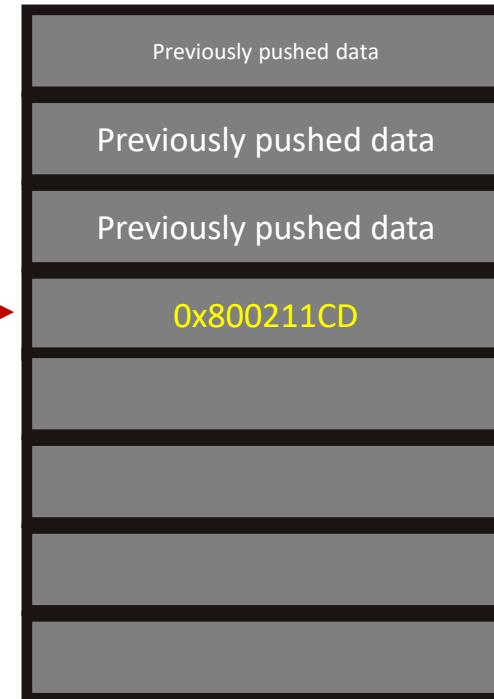
LR = 0x800211CD

R2=0

R3=5

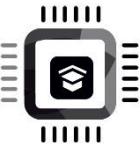
PC=0x80010000

SP



Increasing memory addresses ↑

Stack operation in ARM Cortex Mx processor (FD)



PUSH LR
PUSH R0
PUSH R1
POP R2
POP R3
POP PC

Core registers of the processor

R0 = 0xFFBC

R1 = 0xAB11

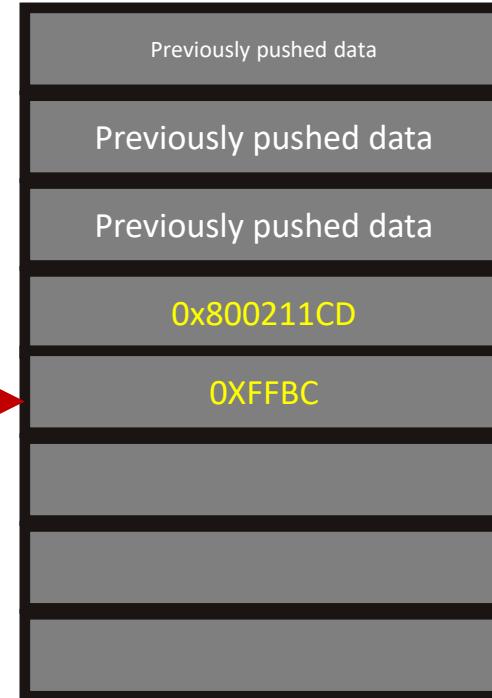
LR = 0x800211CD

R2=0

R3=5

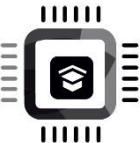
PC=0x80010000

SP



Increasing memory addresses ↑

Stack operation in ARM Cortex Mx processor (FD)

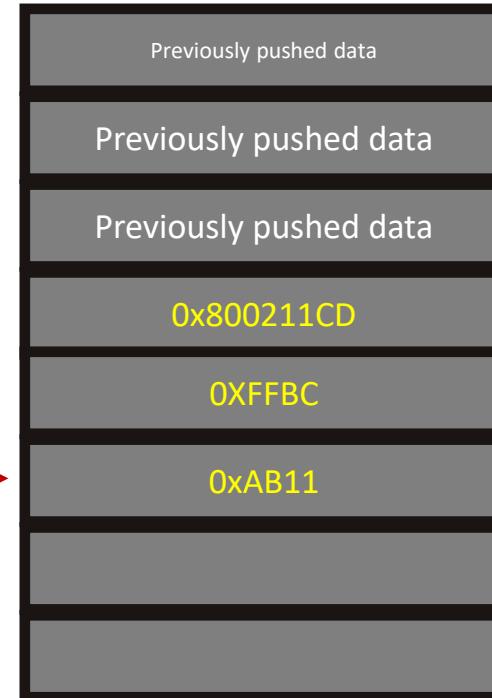


PUSH LR
PUSH R0
PUSH R1
POP R2
POP R3
POP PC

Core registers of the processor

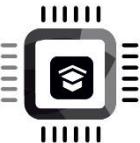
R0 = 0xFFBC
R1 = 0xAB11
LR = 0x800211CD
R2=0
R3=5
PC=0x80010000

SP →



Increasing memory addresses ↑

Stack operation in ARM Cortex Mx processor (FD)



PUSH LR
PUSH R0
PUSH R1
POP R2
POP R3
POP PC

Core registers of the processor

R0 = 0xFFBC

R1 = 0xAB11

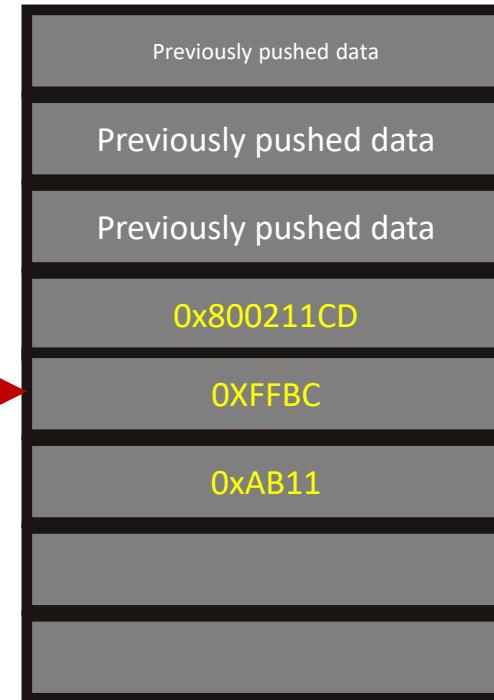
LR = 0x800211CD

R2=0xAB11

R3= 5

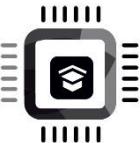
PC=0x80010000

SP



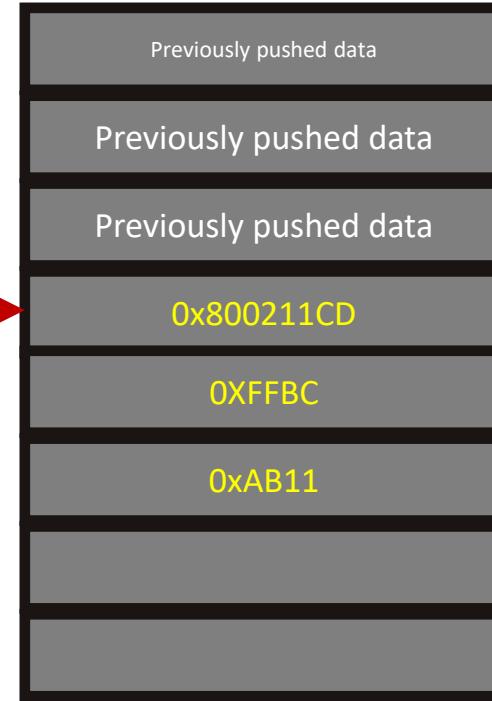
Increasing memory addresses

Stack operation in ARM Cortex Mx processor (FD)



PUSH LR
PUSH R0
PUSH R1
POP R2
POP R3
POP PC

SP →



Increasing memory addresses ↑

Core registers of the processor

R0 = 0xFFBC

R1 = 0xAB11

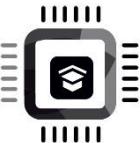
LR = 0x800211CD

R2=0xAB11

R3= 0xFFBC

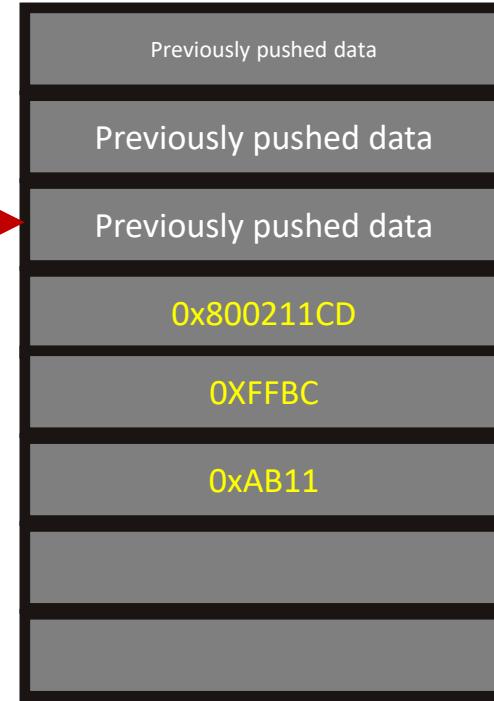
PC=0x80010000

Stack operation in ARM Cortex Mx processor (FD)



PUSH LR
PUSH R0
PUSH R1
POP R2
POP R3
POP PC

SP →



Increasing memory addresses ↑

Core registers of the processor

R0 = 0xFFBC

R1 = 0xAB11

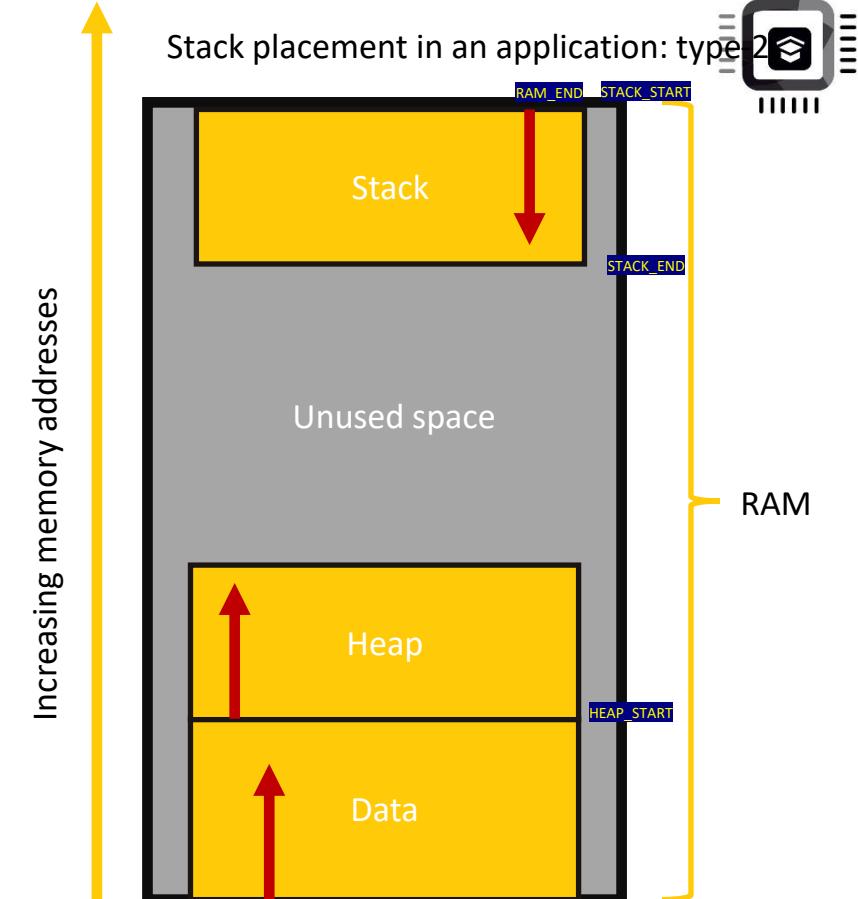
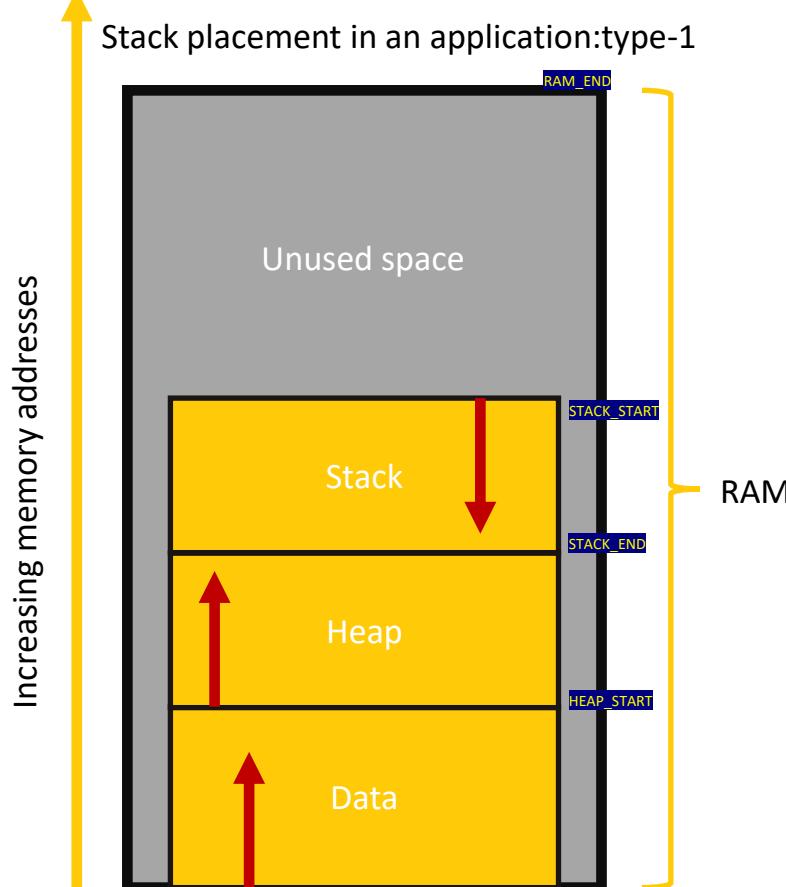
LR = 0x800211CD

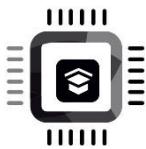
R2=0xAB11

R3= 0xFFBC

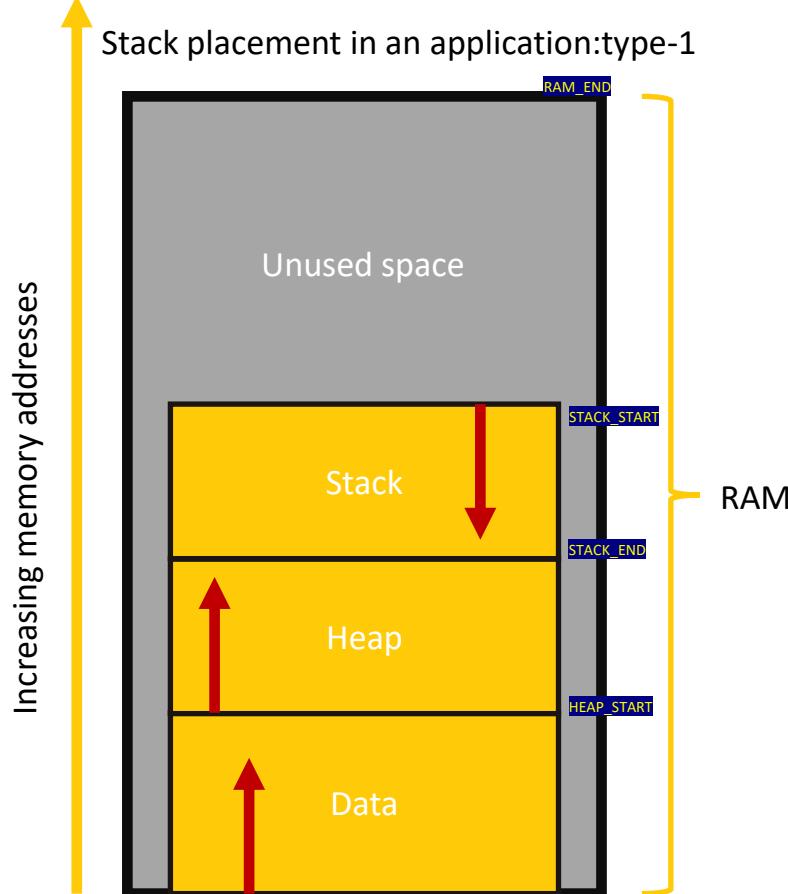
PC= 0x800211CD

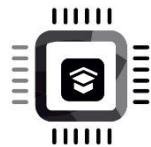
Stack operation in ARM Cortex Mx processor (FD)



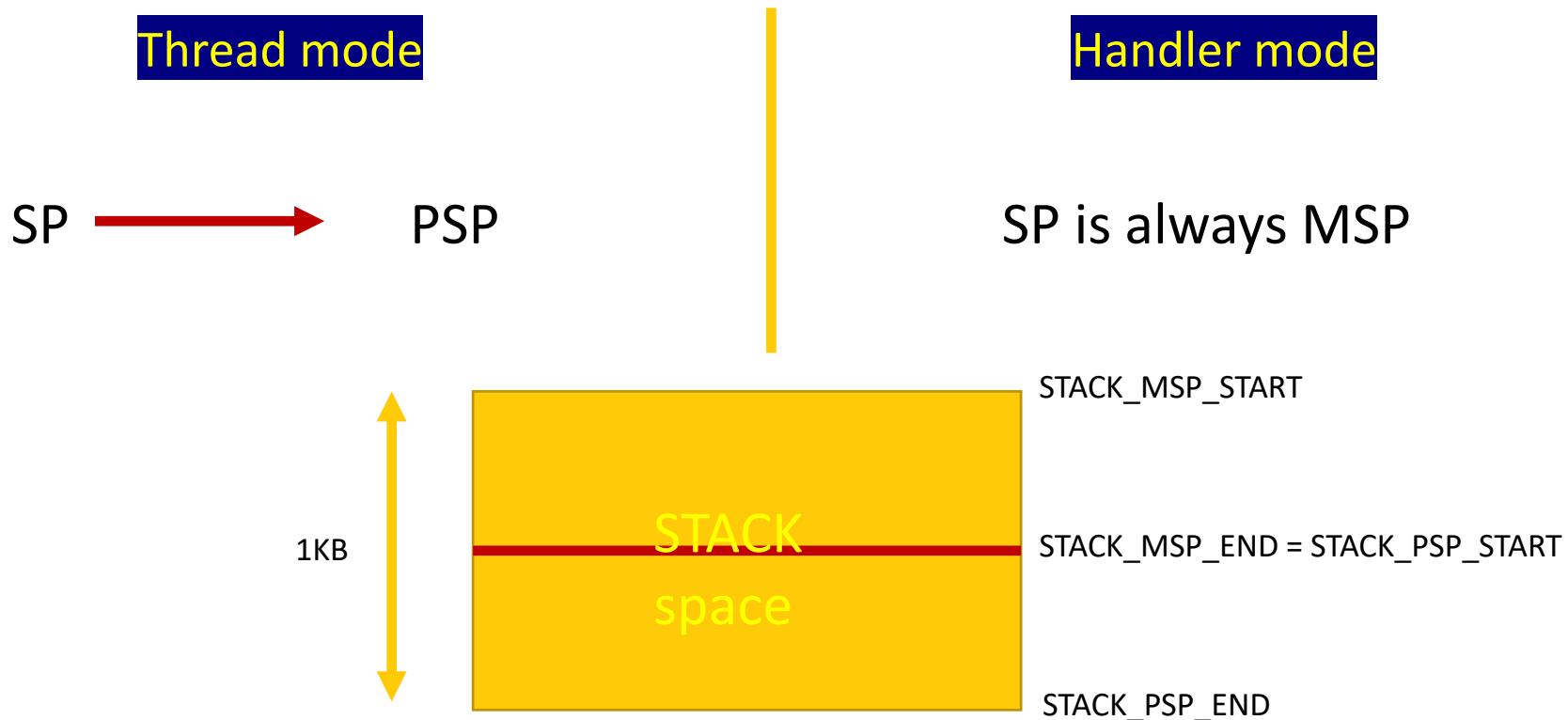


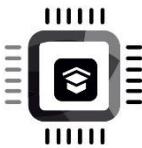
Stack placement





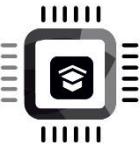
Change SP to PSP for thread mode





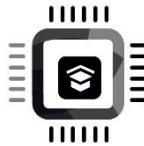
MSP , PSP summary

1. Physically there are 2 stack pointer registers in Cortex-M processors
2. Main Stack Pointer(MSP): This is the default stack pointer used after reset, and is used for all exception/interrupt handlers and for codes which run in thread mode
3. Process Stack Pointer(PSP): This is an alternate stack pointer that can only be used in thread mode. It is usually used for application task in embedded systems and embedded OS
4. After power-up, the processor automatically initializes the MSP by reading the first location of the vector table.



Changing SP

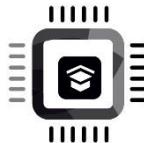
- To access MSP and PSP in assembly code, you can use the MSR and MRS instructions
- In a ‘C’ program you can write a naked function (‘C’ like assembly function which doesn’t have epilogue and prologue sequences) to change the currently selected stack pointer



Function call and AAPCS standard

```
/* this is "caller" */  
void fun_x(void)  
{  
    int ret;  
    ret = fun_y(1,2,4,5);  
}  
  
/* this is "callee" */  
int fun_y(int a, int b , int c, int d)  
{  
    return (a+b+c+d);  
}
```





Procedure Call Standard for the Arm Architecture (AAPCS)

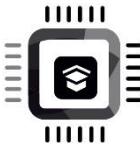
- The AAPCS standard describes procedure call standard used by the application binary interface(ABI) for ARM architecture

SCOPE

The AAPCS defines how subroutines can be separately written, separately compiled, and separately assembled to work together. It describes a contract between a calling routine and a called routine that defines:

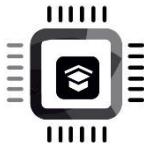
- Obligations on the caller to create a program state in which the called routine may start to execute.
- Obligations on the called routine to preserve the program state of the caller across the call.
- The rights of the called routine to alter the program state of its caller.





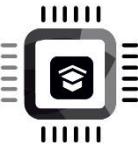
Procedure Call Standard for the Arm Architecture (AAPCS)

- When a 'C' compiler compiles code for the ARM architecture, it should follow the AAPCS specification to generate code
- According to this standard, a 'C' function can modify the registers R0, R1, R2, R3, R14(LR) and PSR and it's not the responsibility of the function to save these registers before any modification
- If a function wants to make use of R4 to R11 registers, then it's the responsibility of the function to save its previous contents before modifying those registers and retrieve it back before exiting the function



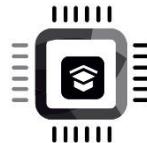
Procedure Call Standard for the Arm Architecture (AAPCS)

- R0, R1, R2, R3, R12, R14(LR) registers are called "caller saved registers," it's the responsibility of the caller to save these registers on stack before calling the function if those values will still be needed after the function call and retrieve it back once the called function returns. Register values that are not required after the function call don't have to be saved.
- R4 to R11 are called "callee saved registers." The function or subroutine being called needs to make sure that, contents of these registers will be unaltered before exiting the function
- According to this standard, caller function uses R0, R1, R2, R3 registers to send input arguments to the callee function.
- The callee function uses registers R0 and R1 to send the result back to the caller function.



Procedure Call Standard for the Arm Architecture (AAPCS)

- R0, R1, R2, R3, R12, R14(LR) registers are called "caller saved registers," it's the responsibility of the caller to save these registers on stack before calling the function if those values will still be needed after the function call and retrieve it back once the called function returns. Register values that are not required after the function call don't have to be saved.
- R4 to R11 are called "callee saved registers." The function or subroutine being called needs to make sure that, contents of these registers will be unaltered before exiting the function
- According to this standard, caller function uses R0, R1, R2, R3 registers to send input arguments to the callee function.
- The callee function uses registers R0 and R1 to send the result back to the caller function.



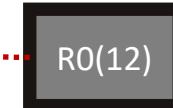
ARM Cortex Mx Core registers

```
/* this is "caller" */
void fun_x(void)
{
    int ret;
    ret = fun_y(1, 2, 4, 5);
}
```

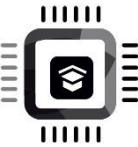


```
/* this is "callee" */
int fun_y(int a, int b, int c, int d)
{
    int result = a+b+c+d;

    return result;
}
```

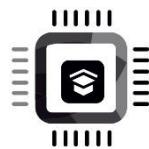


Parameter and
result passing during
a function call as per
AAPCS standard

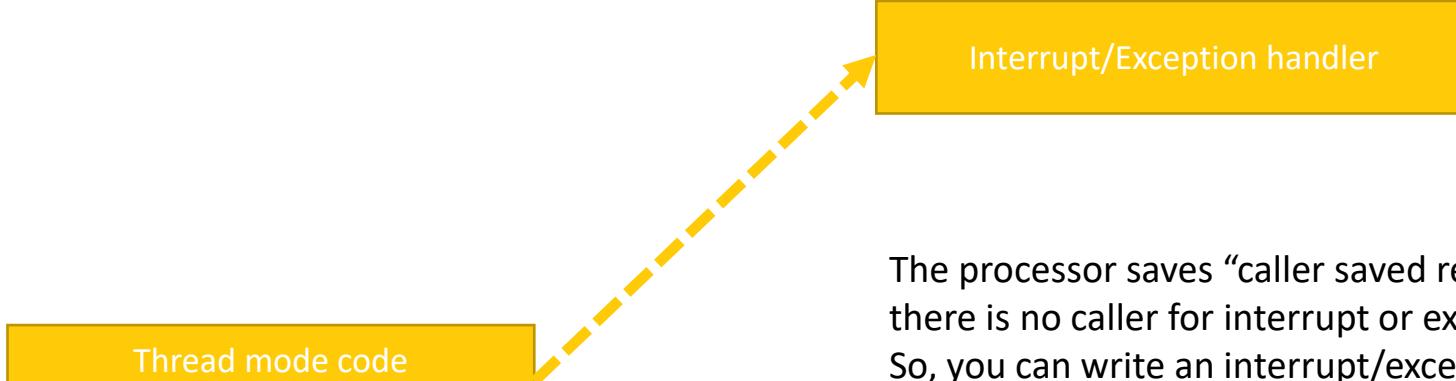


Stack activities during interrupt and exception

- To allow a ‘C’ function to be used as an exception/interrupt handler, the exception mechanism needs to save R0 to R3, R12, LR, and XPSR at exception entrance automatically and restore them at exception exit under the control of the processor hardware.
- In this way, when returned to the interrupted program, all the registers would have the same value as when the interrupt entry sequence started.



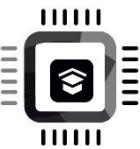
Thread mode code
(user code)



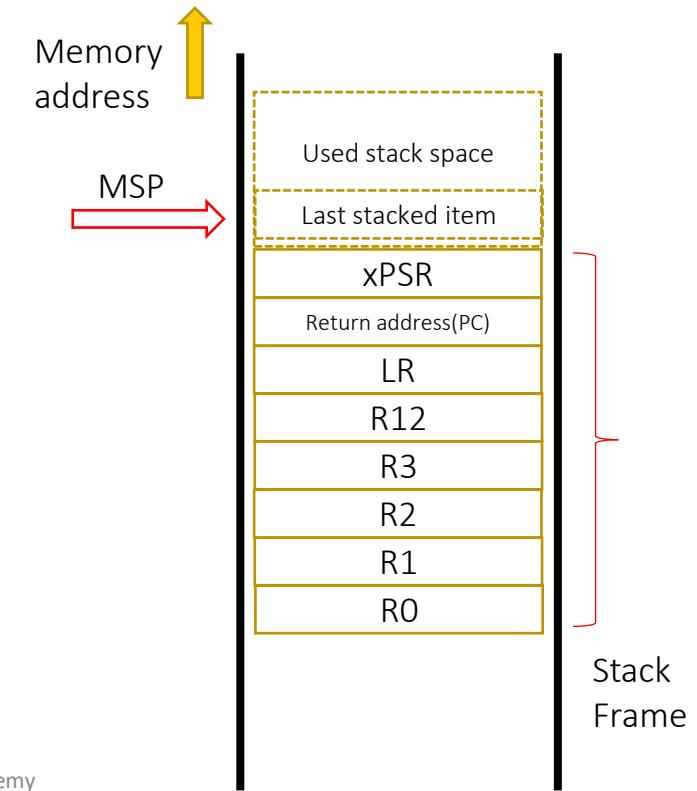
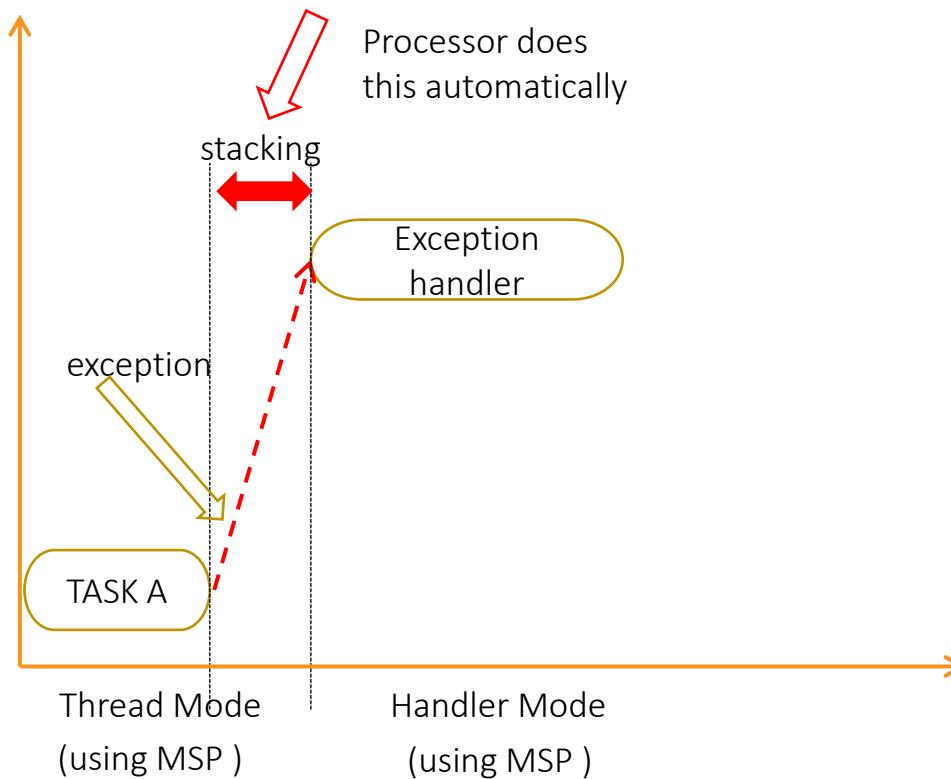
Interrupt/Exception handler

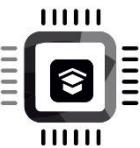
The processor saves “caller saved registers” since there is no caller for interrupt or exception handler. So, you can write an interrupt/exception handler as normal ‘c’ function without worrying about AAPCS rules.



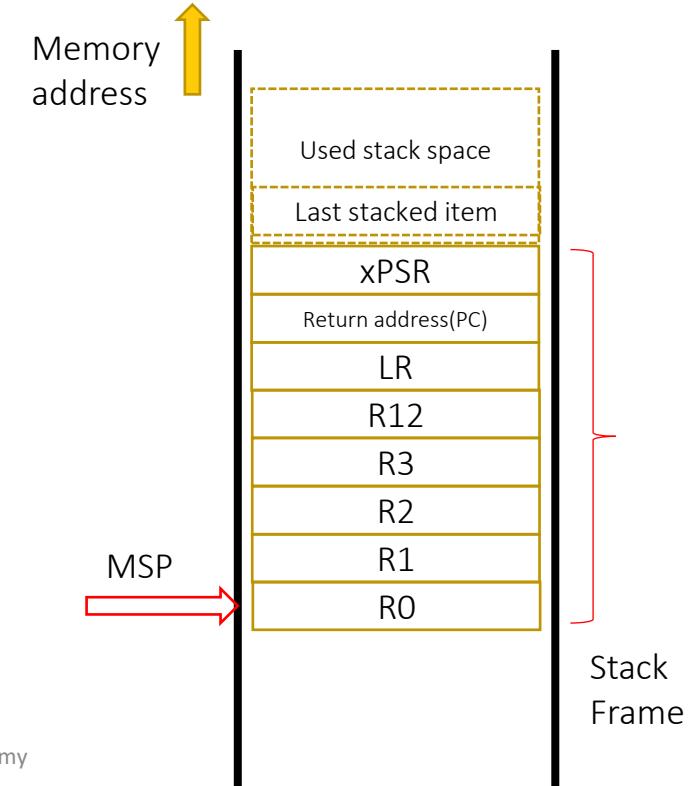
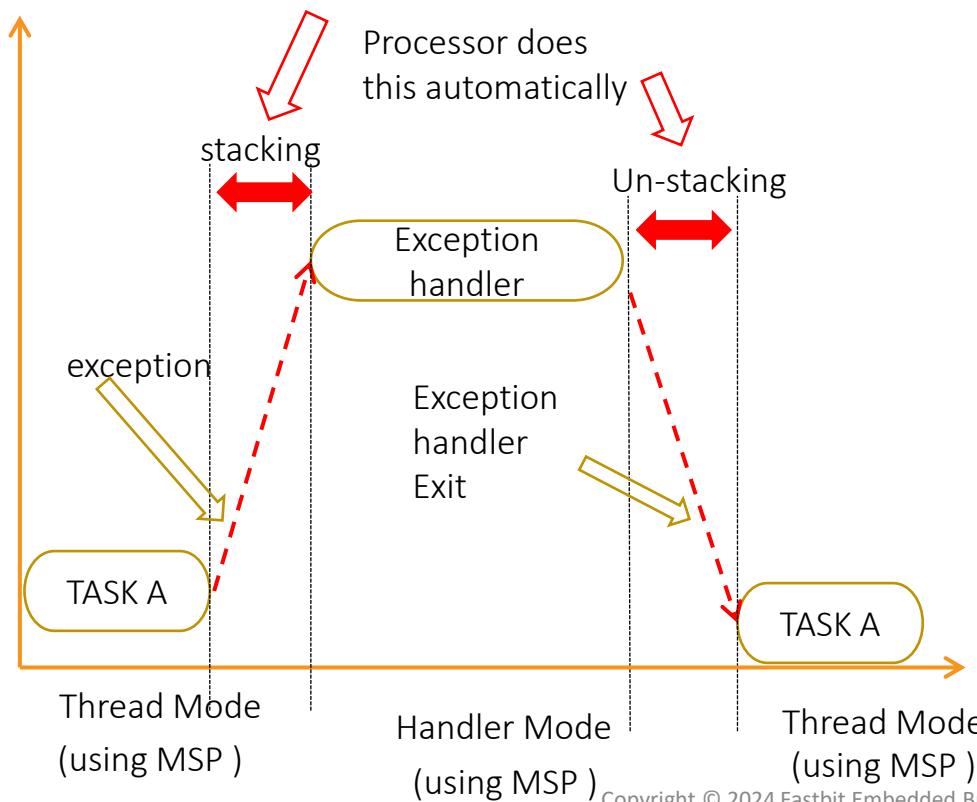


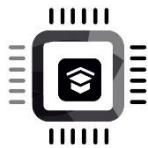
Stacking





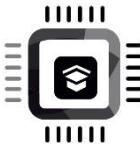
Un-stacking





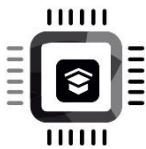
Stack initialization

- Before reaching main
- After reaching the main function, you may again reinitialize the stack pointer

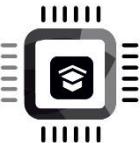


Stack initialization tips

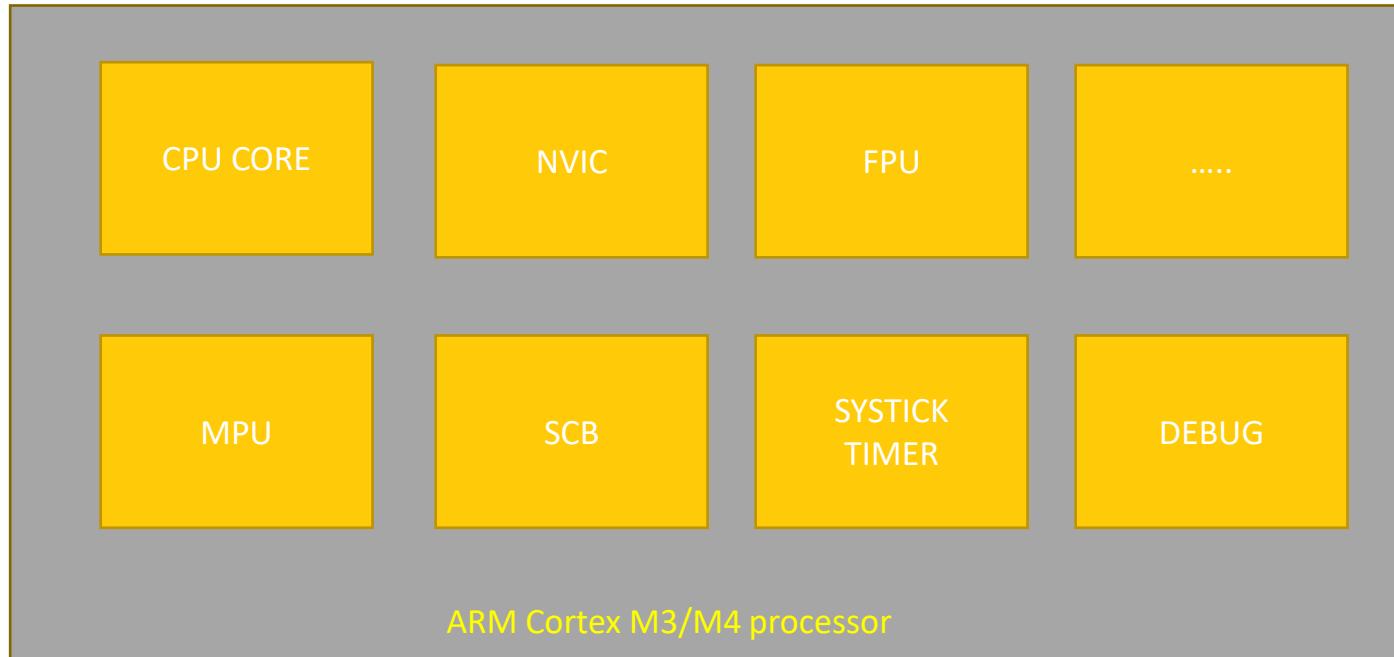
- 1) Evaluate your targeted application. Decide the amount of stack that would be needed for the worst-case scenario of your application run time
- 2) Know your processor's stack consumption model (FD, FA, ED, EA)
- 3) Decide stack placement in the RAM (middle, end, external memory)
- 4) In many applications, there may be second stage stack init. For example, if you want to allocate stack in external SDRAM then first start with internal RAM, in the main or startup code initialize the SDRAM then change the stack pointer to point to SDRAM
- 5) If you are using the ARM cortex Mx processor, make sure that the first location of the vector table contains the initial stack address (MSP). The startup code of the project usually does this.
- 6) You may also use the linker script to decide the stack, heap and other RAM area boundaries. Startup code usually fetches boundary information from linker scripts.
- 7) In an RTOS scenario, the kernel code may use MSP to trace its own stack and configure PSP for user task's stack

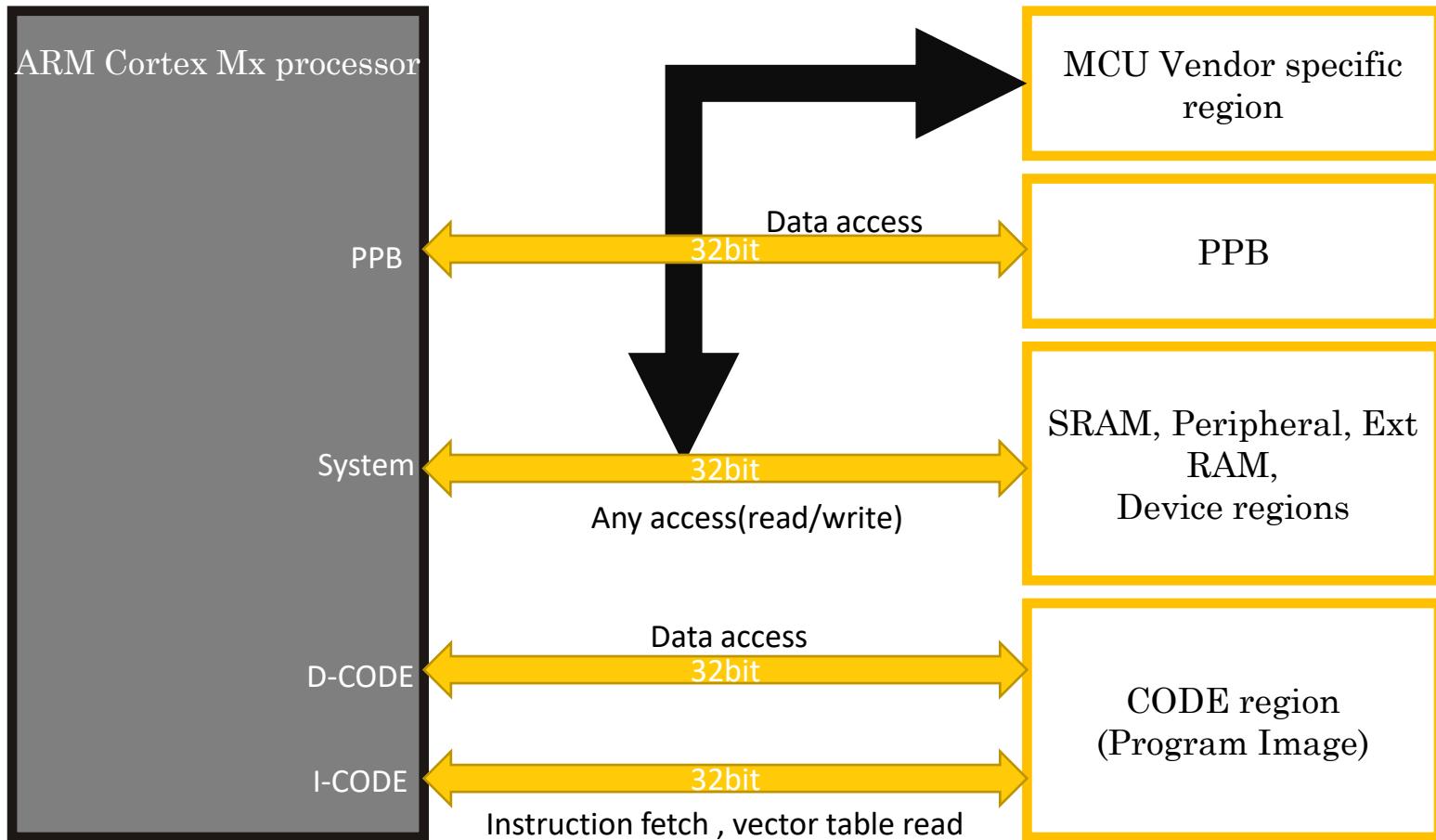
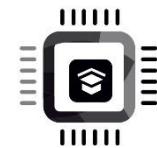


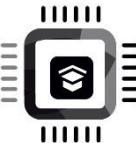
System exception control registers



ARM Cortex-M3/M4 processor peripherals







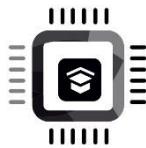
The address map of the Private Peripheral Bus(PPB)

The address map of the *Private Peripheral Bus* (PPB) is:

Table 4-1 Core peripheral register regions

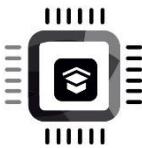
Address	Core peripheral	Description
0xE000E008-0xE000E00F	SyStem Control Block	Table 4-12 on page 4-11
0xE000E010-0xE000E01F	System timer	Table 4-32 on page 4-33
0xE000E100-0xE000E4EF	Nested Vectored Interrupt Controller	Table 4-2 on page 4-3
0xE000ED00-0xE000ED3F	System Control Block	Table 4-12 on page 4-11
0xE000ED90-0xE000ED93	MPU Type Register	Reads as zero, indicating MPU is not implemented ^a
0xE000ED90-0xE000EDB8	Memory Protection Unit	Table 4-38 on page 4-38
0xE000EF00-0xE000EF03	Nested Vectored Interrupt Controller	Table 4-2 on page 4-3
0xE000EF30-0xE000EF44	Floating Point Unit	Table 4-49 on page 4-48

a. Software can read the MPU Type Register at 0xE000ED90 to test for the presence of a *Memory Protection Unit* (MPU).



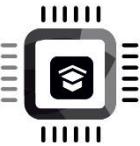
System Control Block(SCB)

- The System Control Block (SCB) provides system implementation information and system control. This includes configuration, control, and reporting of the system exceptions.
- Explore registers from ARM Cortex M4 generic user guide



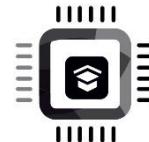
SCB registers

- You can enable fault handlers
- Get pending status of the fault exceptions
- Trap processor for divide by zero and unaligned data access attempts
- Control sleep and sleep wakeup settings
- Configure the priority of system exceptions
- Systick timer control and status

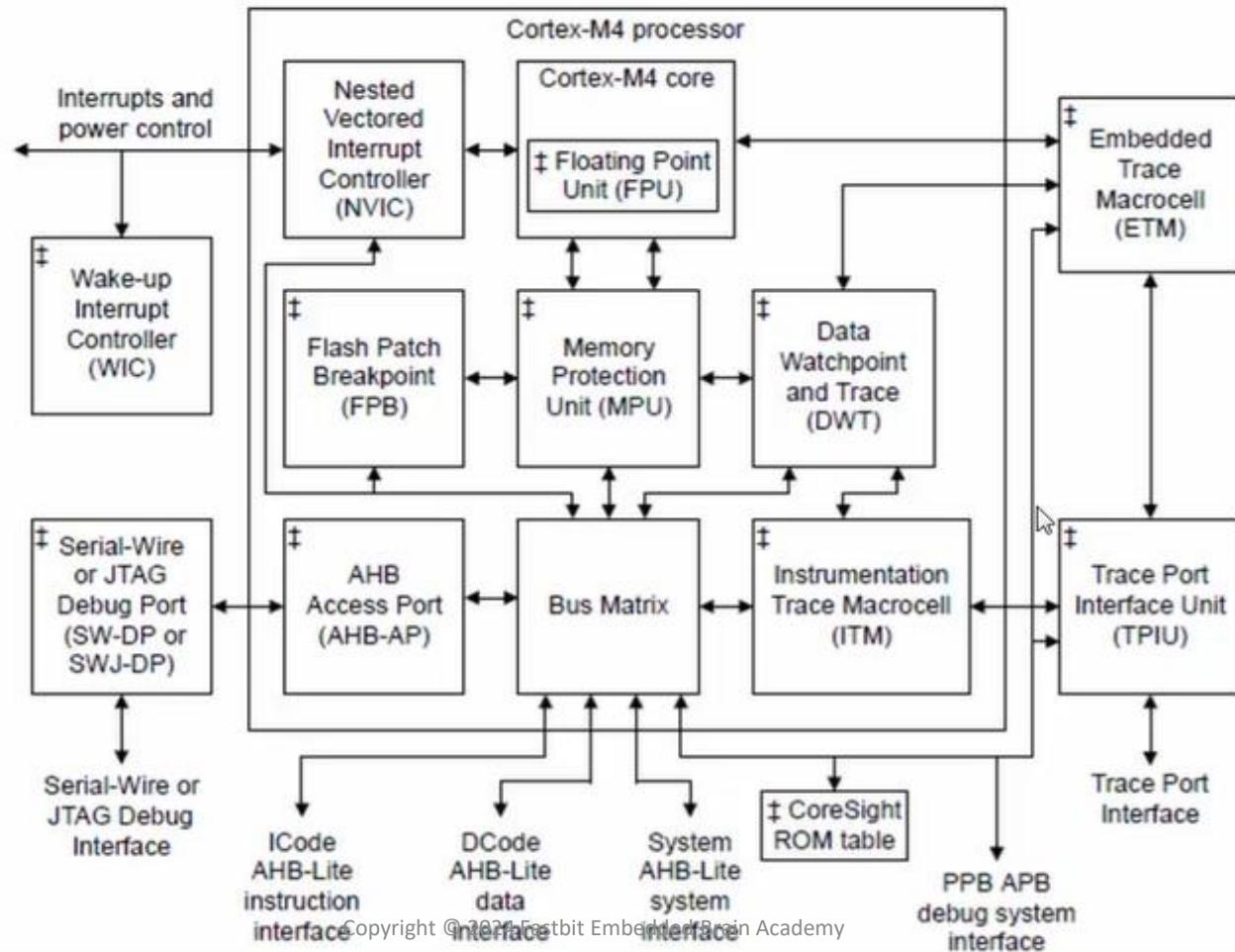


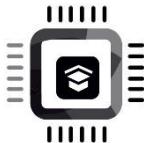
Default system exception status

Exception	Default state
Hard fault	Always enable by default , can be masked
NMI	Always enabled , cannot be masked
Usage fault	Disabled by default
Mem manage fault	Disabled by default
Bus fault	Disabled by default
Systick exception	Disabled by default and triggers whenever systick timer is enabled and expires
SVC exception	Triggers only when the svc instruction is executed
PendSV exception	Disabled by default
Debug monitor exception	Disabled by default



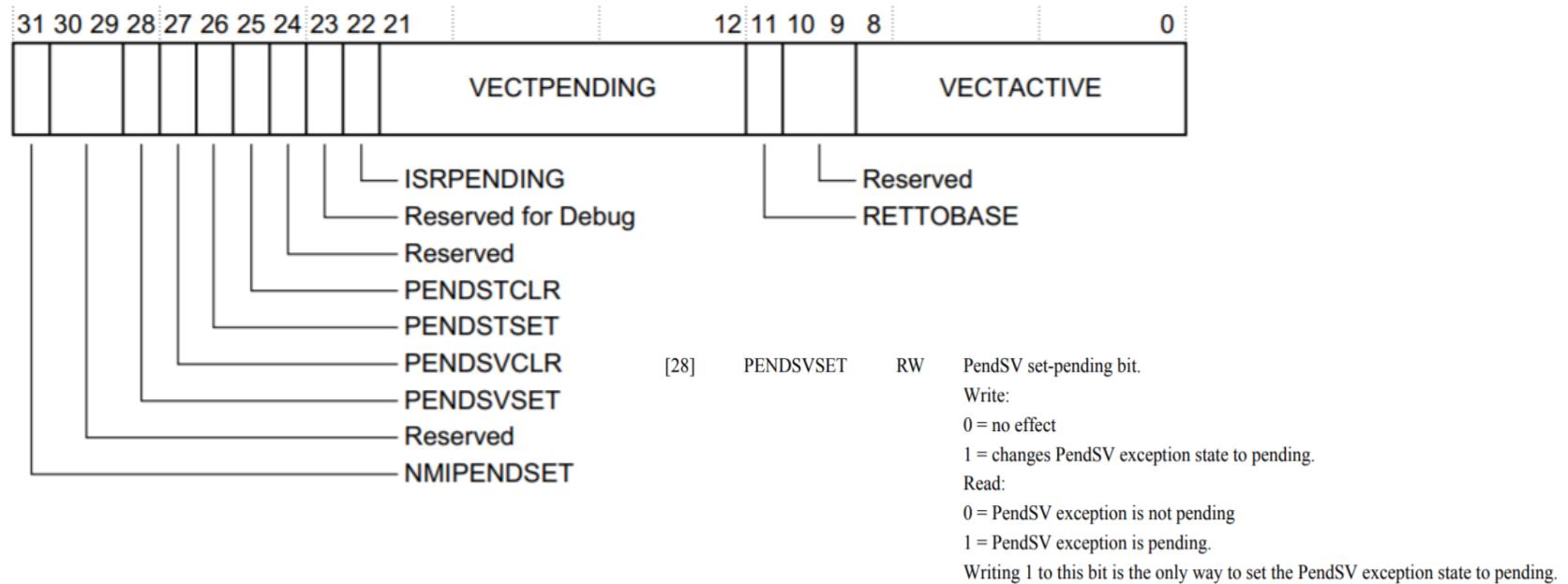
Block diagram showing the structure of the Cortex-M4 processor.

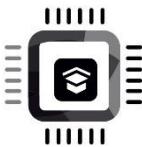




Enable PendSV Exception

- Interrupt Control and State Register





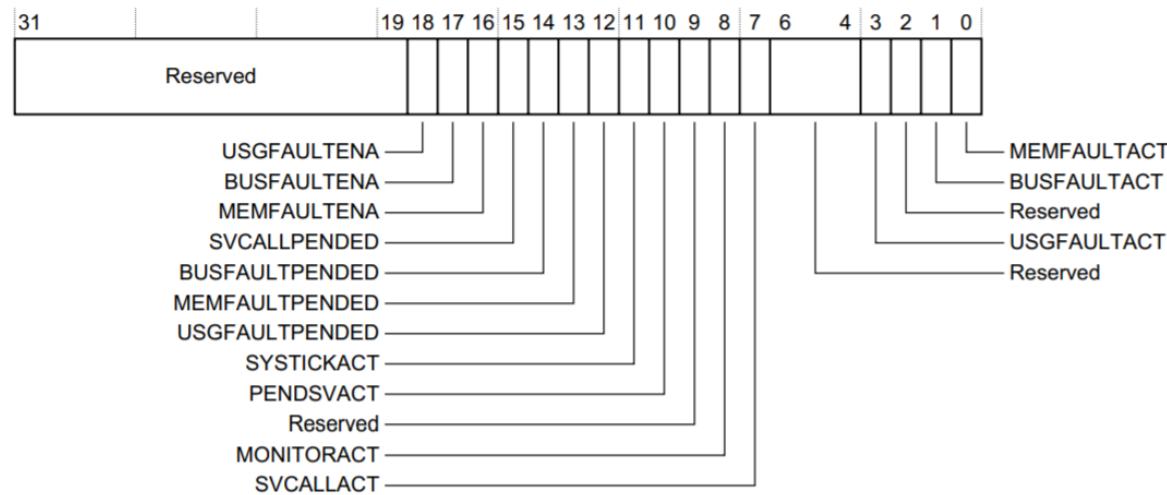
Enable fault exceptions

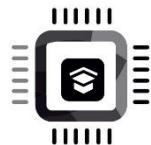
4.3.9 System Handler Control and State Register

The SHCSR enables the system handlers, and indicates:

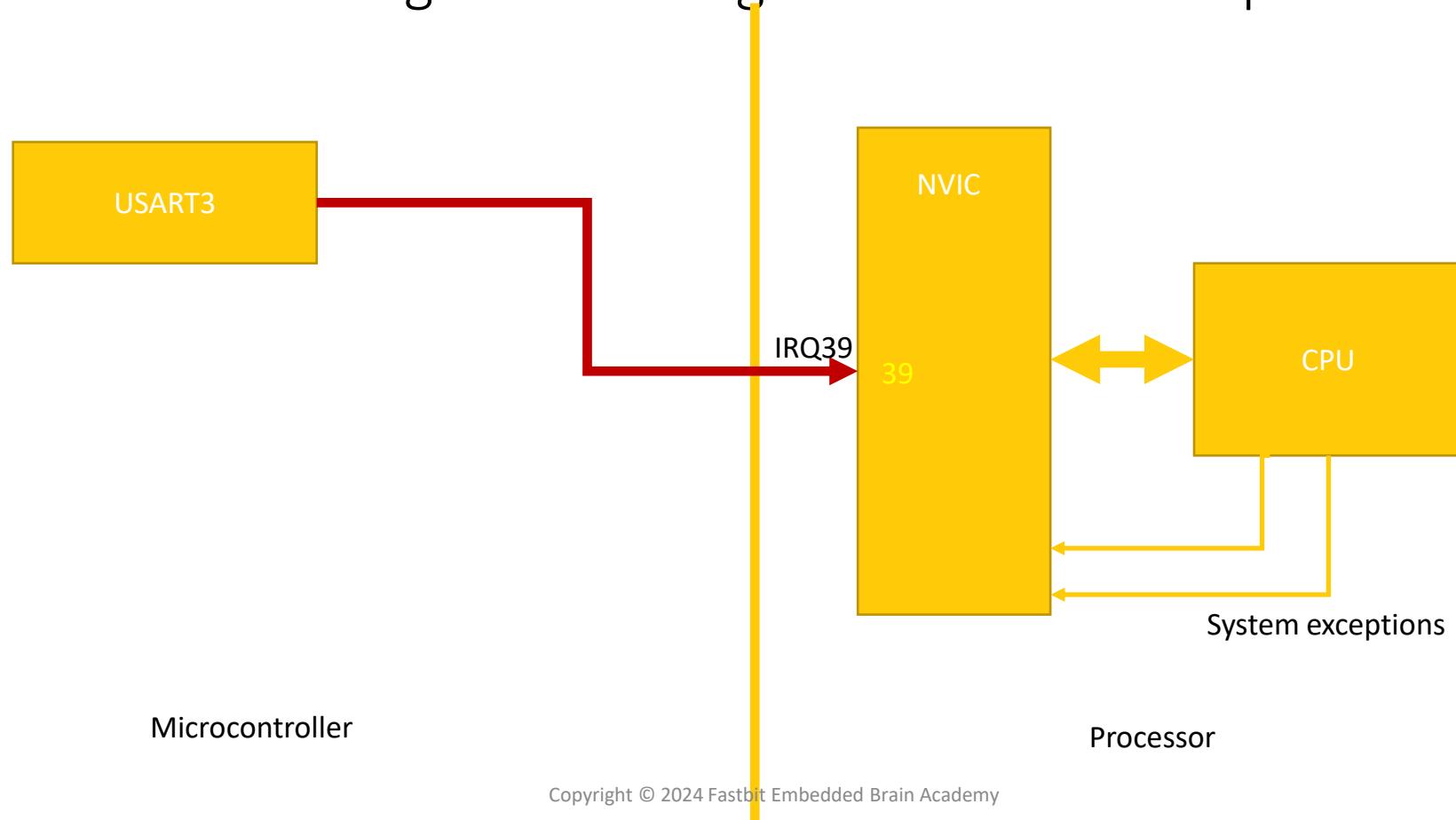
- the pending status of the BusFault, MemManage fault, and SVC exceptions
- the active status of the system handlers.

See the register summary in [Table 4-12 on page 4-11](#) for the SHCSR attributes. The bit assignments are:

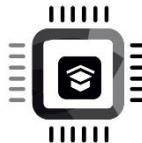




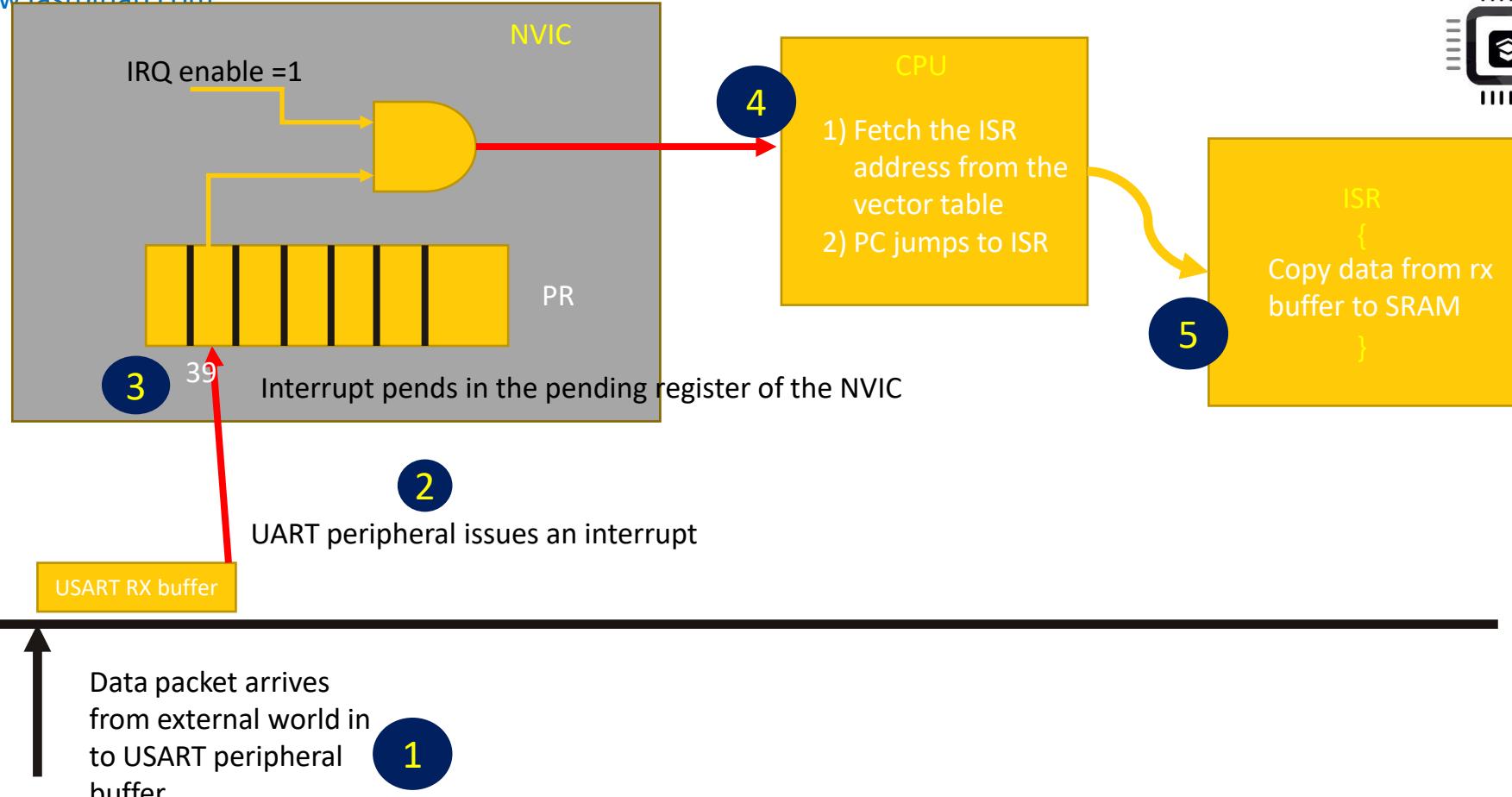
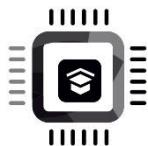
Exercise-Enabling and Pending of USART3 Interrupt

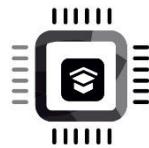


Steps to program an MCU peripheral interrupt



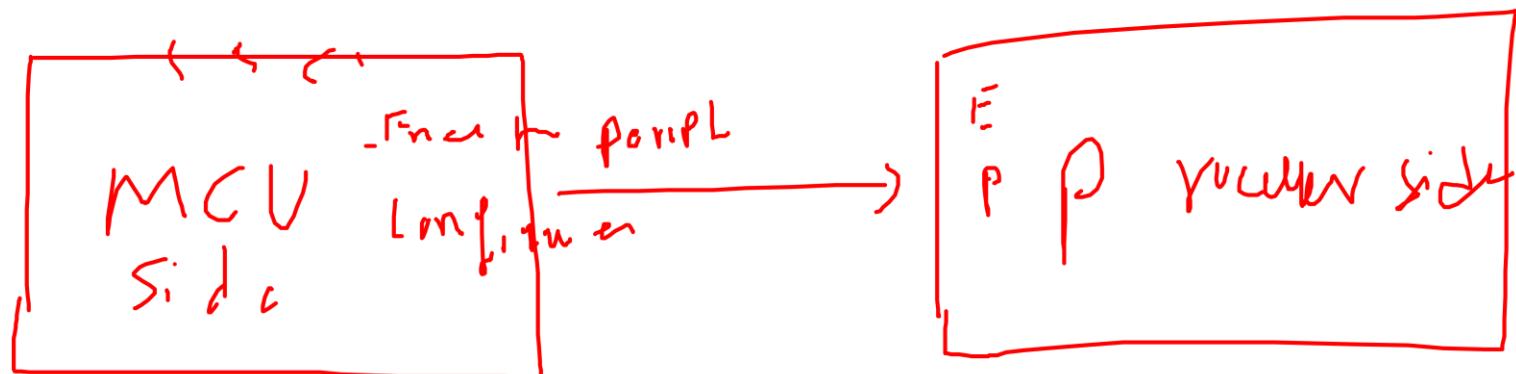
- Identify the IRQ number of the peripheral by referring to the MCU vector table. IRQ numbers are vendor-specific
- Program the Processor register to enable that IRQ (only when you enable the IRQ, the processor will accept the interrupt over that line). Set the priority (optional)
- Configure the peripheral (USART3) using its peripheral configuration register. For example, in the case of USART3, whenever a packet is received, it will automatically issue an interrupt on the IRQ line 39.
- When the interrupt is issued on the IRQ line, it will first get pended in the pending register of the processor
- NVIC will allow the IRQ handler associated with the IRQ number to run only if the priority of the new interrupt is higher than the currently executing interrupt handler. Otherwise newly arrived interrupt will stay in pending state.
- Please note that if peripheral issues an interrupt when the IRQ number is disabled (not activated from the processor side), then still interrupt will get pended in the pending register of the NVIC. As soon as IRQ is enabled, it will trigger the execution of the ISR if the priority is higher than the currently active ISR.

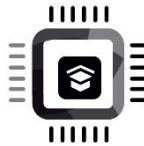




MCU peripheral configuration

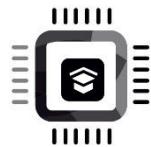
- It has 2 configuration parts



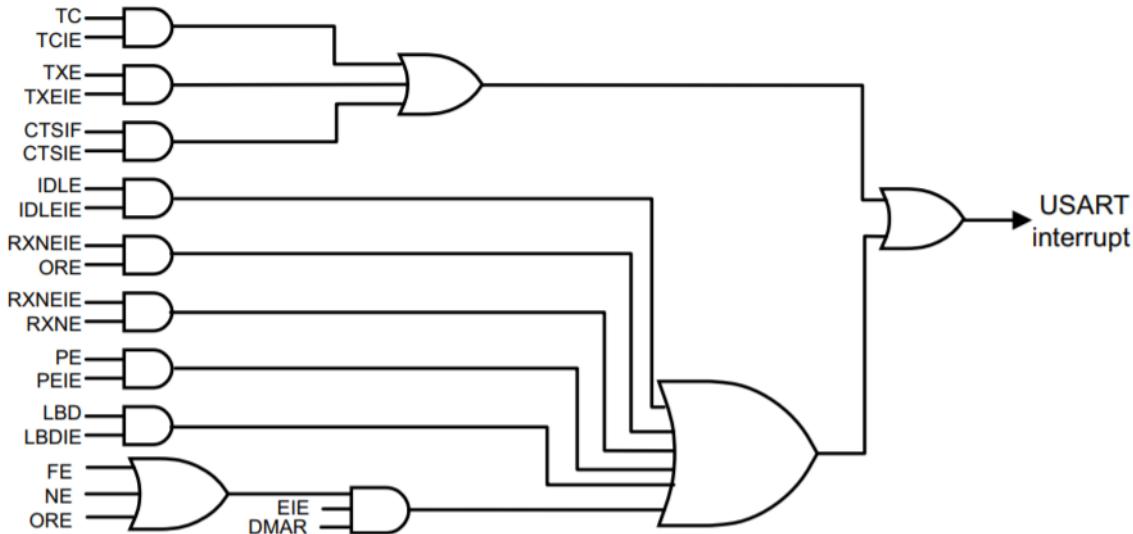


About peripheral programming

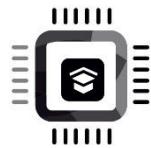
- Peripheral programming such as UART, SPI, TIMER is out of the scope of this course.
- This is covered in MCU1 and MCU2 courses.



USART interrupt mapping diagram

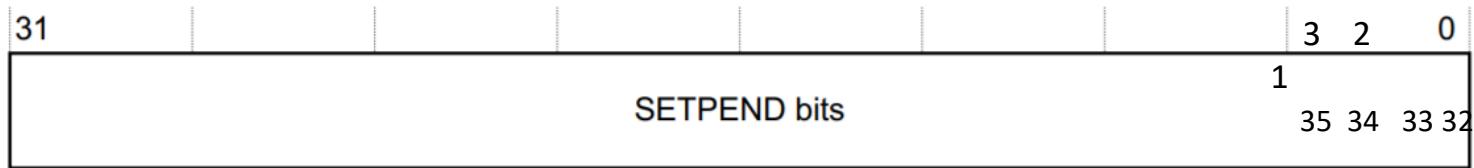


MSv42089V1

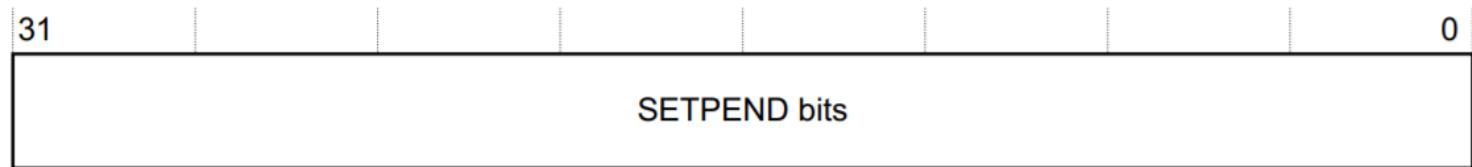


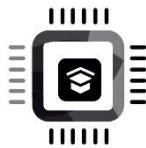
Interrupt Set Pending Registers

NVIC_ISPR1



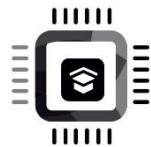
NVIC_ISPR0





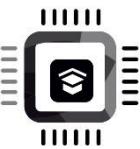
Interrupt priority and configuration

- What is priority ?
- What is priority value ?
- Different programmable priority levels
- Explore priority registers for interrupts (IRQs)
- Explore priority registers for system exceptions
- Pre-empt priority and sub priority



What is priority ?



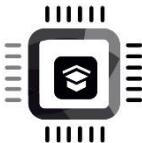


What is priority ?

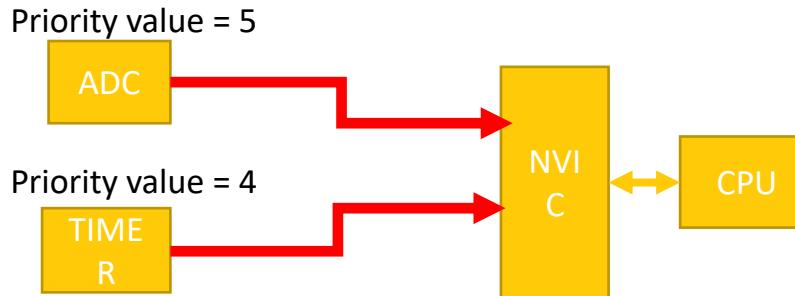
Priority means urgency

What is priority value ?

The priority value is a measure of urgency (Decides how urgent from others)



Relation between priority value & priority

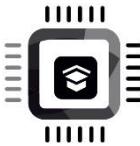


For ARM cortex Mx processor
Lesser the priority value , higher
the priority (urgency)

- In this case, Timer peripheral priority value is lesser than the priority value of ADC; hence TIMER interrupt is more URGENT than ADC interrupt.
- So, we say TIMER priority is HIGHER than ADC priority
- If both interrupts hit the NVIC at the same time, NVIC allows TIMER interrupt first. so, TIMER ISR will be executed first by the processor.

Note:

Sometimes we use the term "priority" synonymous with the word "priority value". Instead of saying priority value of IRQ0 is 4, we just say priority of IRQ0 is 4



Different priority levels

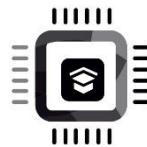
- Priority values are also called as priority levels.

How many different priority levels are there in ARM cortex Mx processor?

It depends on the Interrupt Priority Register implementation by the MCU vendors.

STM32F4x MCU has 16 different priority levels

TI TM4C123Gx has 8 different priority levels



Interrupt Priority register of the ARM Cortex Mx processor

- Interrupt Priority register is part of NVIC register set

Note :

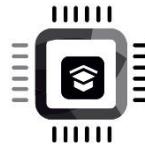
By Using these registers you can configure priority levels for interrupts(IRQs) only.

Table 4-2 NVIC register summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000E100-0xE000E11C	NVIC_ISER0-NVIC_ISER7	RW	Privileged	0x00000000	<i>Interrupt Set-enable Registers on page 4-4</i>
0xE000E180-0xE000E19C	NVIC_ICER0-NVIC_ICER7	RW	Privileged	0x00000000	<i>Interrupt Clear-enable Registers on page 4-5</i>
0xE000E200-0xE000E21C	NVIC_ISPR0-NVIC_ISPR7	RW	Privileged	0x00000000	<i>Interrupt Set-pending Registers on page 4-5</i>
0xE000E280-0xE000E29C	NVIC_ICPR0-NVIC_ICPR7	RW	Privileged	0x00000000	<i>Interrupt Clear-pending Registers on page 4-6</i>
0xE000E300-0xF000E31C	NVIC_IABR0-NVIC_IABR7	RW	Privileged	0x00000000	<i>Interrupt Active Bit Registers on page 4-7</i>
0xE000E400-0xE000E4EF	NVIC_IPR0-NVIC_IPR59	RW	Privileged	0x00000000	<i>Interrupt Priority Registers on page 4-7</i>
0xE000EF00	STIR	WO	Configurable ^a	0x00000000	<i>Software Trigger Interrupt Register on page 4-8</i>

a. See the register description for more information.

60 Interrupt priority registers



Interrupt Priority Register of the ARM Cortex Mx processor

4.2.7 Interrupt Priority Registers

The NVIC_IPR0-NVIC_IPR59 registers provide an 8-bit priority field for each interrupt and each register holds four priority fields. These registers are byte-accessible. See the register summary in [Table 4-2 on page 4-3](#) for their attributes. Each register holds four priority fields as shown:

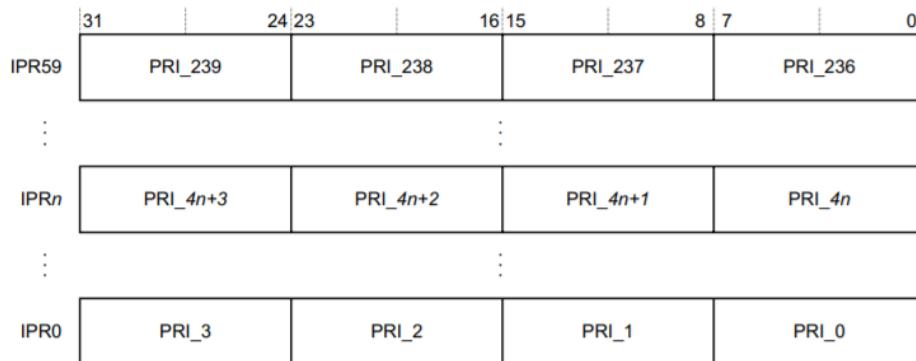
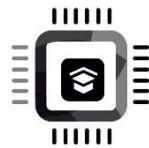
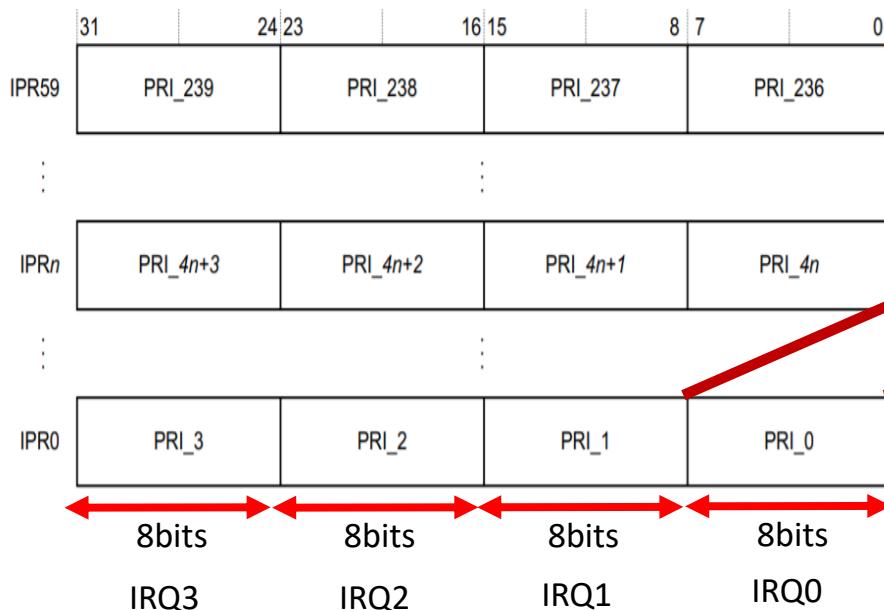


Table 4-9 IPR bit assignments

Bits	Name	Function
[31:24]	Priority, byte offset 3	Each implementation-defined priority field can hold a priority value, 0-255. The lower the value, the greater the priority of the corresponding interrupt. Register priority value fields are eight bits wide, and non-implemented low-order bits read as zero and ignore writes.
[23:16]	Priority, byte offset 2	
[15:8]	Priority, byte offset 1	
[7:0]	Priority, byte offset 0	

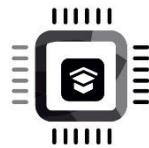


Interrupt Priority Register implementation



Real implementation by the MCU vendors

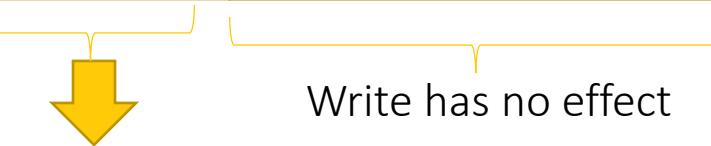
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented				Not implemented			



Interrupt Priority Register implementation

Microcontroller Vendor XXX

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented	Not implemented						

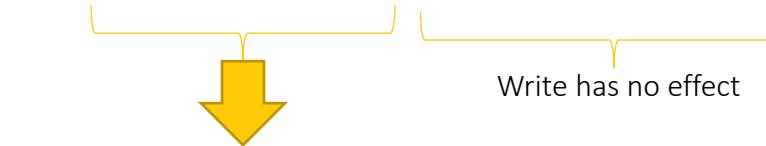


8 priority levels

0x00,0x20,0x40,0x60,
0x80,0xA0,0xC0, 0xE0

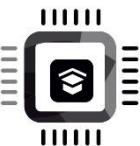
Microcontroller Vendor YYY

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented	Not implemented						

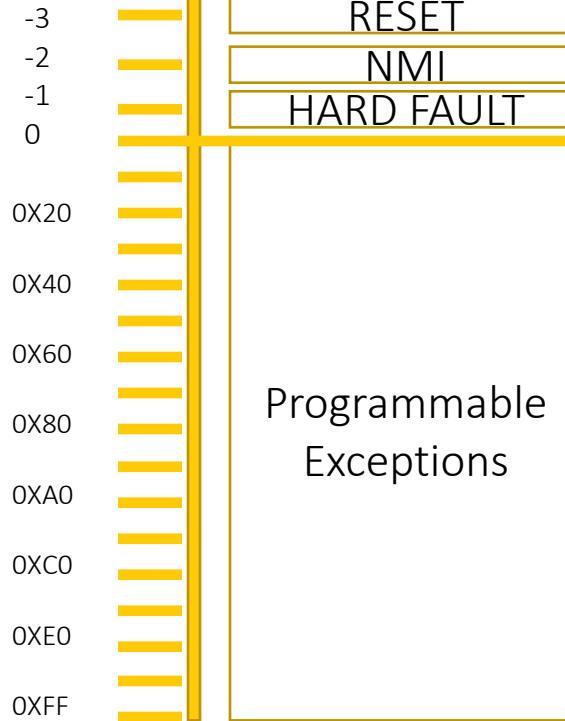


16 priority levels

0x00,0x10,0x20,0x30,0x40,0x50,
0x60,0x70,0x80,0x90,0xa0,0xb0,0xc0,0x
d0,0xe0,0xf0

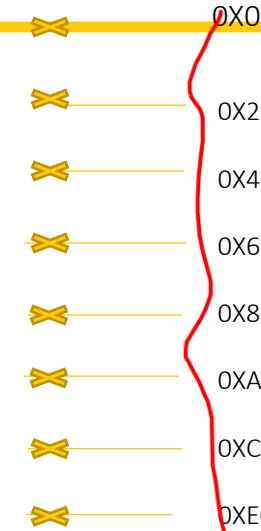


Highest Priority

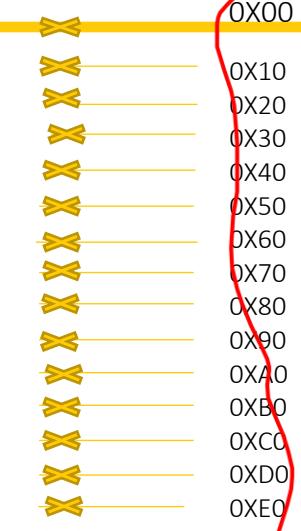


lowest Priority

3-bits priority width
In interrupt priority register



4 bits priority width
In interrupt priority register





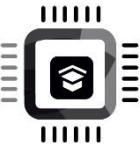
4.3 System control block

The *System Control Block* (SCB) provides system implementation information, and system control. This includes configuration, control, and reporting of the system exceptions. The system control block registers are:

Table 4-12 Summary of the system control block registers

Address	Name	Type	Required privilege	Reset value	Description
0xE000E008	ACTLR	RW	Privileged	0x00000000	<i>Auxiliary Control Register</i>
0xE000ED00	CPUID	RO	Privileged	0x410FC240	<i>CPUID Base Register</i> on page 4-13
0xE000ED04	ICSR	RW ^a	Privileged	0x00000000	<i>Interrupt Control and State Register</i> on page 4-13
0xE000ED08	VTOR	RW	Privileged	0x00000000	<i>Vector Table Offset Register</i> on page 4-16
0xE000ED0C	AIRCR	RW ^a	Privileged	0xFA050000	<i>Application Interrupt and Reset Control Register</i> on page 4-16
0xE000ED10	SCR	RW	Privileged	0x00000000	<i>System Control Register</i> on page 4-19
0xE000ED14	CCR	RW	Privileged	0x00000200	<i>Configuration and Control Register</i> on page 4-19
0xE000ED18	SHPR1	RW	Privileged	0x00000000	<i>System Handler Priority Register 1</i> on page 4-21
0xE000ED1C	SHPR2	RW	Privileged	0x00000000	<i>System Handler Priority Register 2</i> on page 4-22
0xE000ED20	SHPR3	RW	Privileged	0x00000000	<i>System Handler Priority Register 3</i> on page 4-22
0xE000ED24	SHCRS	RW	Privileged	0x00000000	<i>System Handler Control and State Register</i> on page 4-23
0xE000ED28	CFSR	RW	Privileged	0x00000000	<i>Configurable Fault Status Register</i> on page 4-24
0xE000ED28	MMSR ^b	RW	Privileged	0x00	<i>MemManage Fault Status Register</i> on page 4-25
0xE000ED29	BFSR ^b	RW	Privileged	0x00	<i>BusFault Status Register</i> on page 4-26
0xE000ED2A	UFSR ^b	RW	Privileged	0x0000	<i>UsageFault Status Register</i> on page 4-28
0xE000ED2C	HFSR	RW	Privileged	0x00000000	<i>HardFault Status Register</i> on page 4-30
0xE000ED34	MMAR	RW	Privileged	Unknown	<i>MemManage Fault Address Register</i> on page 4-30
0xE000ED38	BFAR	RW	Privileged	Unknown	<i>BusFault Address Register</i> on page 4-31
0xE000ED3C	AFSR	RW	Privileged	0x00000000	<i>Auxiliary Fault Status Register</i> on page 4-31

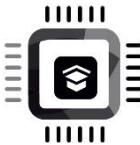
Priority
Registers
for system
exceptions



Pre-empt priority and sub priority

What if two interrupts of the same priority hit the processor at the same time?

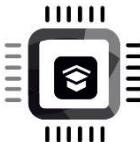
The sub priority value of the interrupts will be checked to resolve the conflict. An interrupt with lower sub priority value will be allowed first.



Priority Grouping

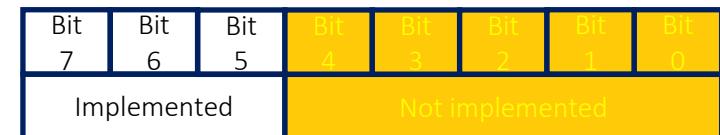
Pre-Empt Priority: When the processor is running interrupt handler, and another interrupt appears, then the pre-empt priority values will be compared, and interrupt with higher pre-empt priority(less in value) will be allowed to run.

Sub Priority: This value is used only when two interrupts with the same pre-empt priority values occur at the same time. In this case, the interrupt with higher sub-priority(less in value) will be handled first.

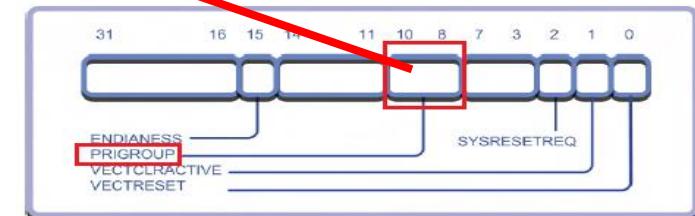


Priority Grouping

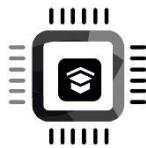
Priority Group	Pre-empt priority field	sub-priority field
0(default)	Bit[7:1]	Bit[0]
1	Bit[7:2]	Bit[1:0]
2	Bit[7:3]	Bit[2:0]
3	Bit[7:4]	Bit[3:0]
4	Bit[7:5]	Bit[4:0]
5	Bit[7:6]	Bit[5:0]
6	Bit[7:7]	Bit[6:0]
7	None	Bit[7:0]



Interrupt priority register



Application Interrupt And Reset Control Register



Priority Grouping Case Study

Case 1 :

when the Priority group = 0,

As per the table we have,

pre-empt priority width = 7 bits (128 programmable priority levels)

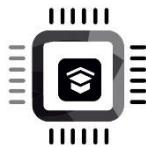
But only 3 bits are implemented, so ,8 programmable priority levels

Sub-priority width = 1 (2 programmable sub priority levels)

Bit 0 is not implemented so no sub priority levels

Not implemented							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Pre-empt priority			Pre-empt priority			Sub priority	

priority level register



Priority Grouping Case Study

Case 2 :

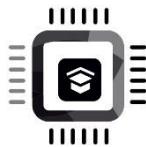
when the priority group = 5,

pre-empt priority width = 2 bits (4 programmable priority levels)

Not implemented							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preempt priority	Sub pri	Not implemented					

Sub-priority width = 6 (64 programmable sub priority levels)

Since only 1 bit is implemented , only 2 programmable sub priority levels



Priority Grouping Case Study

Case 2 :

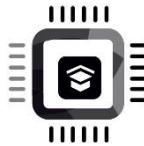
when the priority group = 5,

pre-empt priority width = 2 bits (4 programmable priority levels)

Not implemented							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preempt priority	Sub pri	Not implemented					

Sub-priority width = 6 (64 programmable sub priority levels)

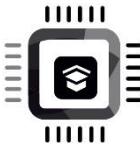
Since only 1 bit is implemented , only 2 programmable sub priority levels



Pre-empt priority and sub priority

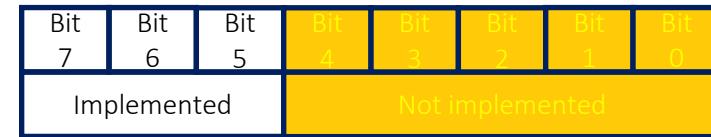
What if two interrupts of the same pre-empt priority and sub priority hit the processor at the same time?

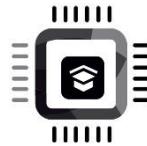
Interrupt with the lowest IRQ number will be allowed first.



Case of same pre-empt and sub priority

Priority Group	Pre-empt priority field	sub-priority field
0(default)	Bit[7:1]	Bit[0]
1	Bit[7:2]	Bit[1:0]
2	Bit[7:3]	Bit[2:0]
3	Bit[7:4]	Bit[3:0]
4	Bit[7:5]	Bit[4:0]
5	Bit[7:6]	Bit[5:0]
6	Bit[7:7]	Bit[6:0]
7	None	Bit[7:0]



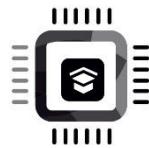


Interrupt priority configuration : Exercise

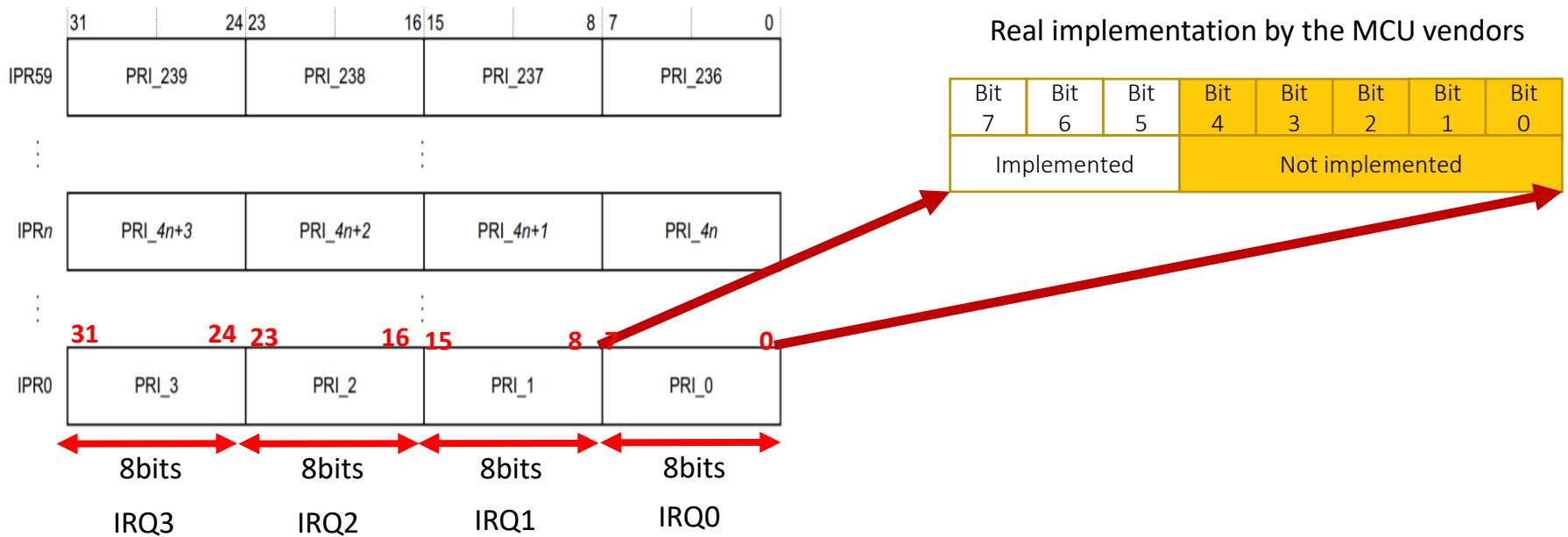
Generate the below peripheral interrupts using NVIC interrupt pending register and observe the execution of ISRs when priorities are same and different

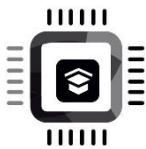
TIM2 global interrupt

I2C1 event interrupt

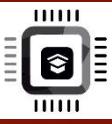


Interrupt Priority Register implementation

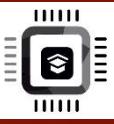




Pending Interrupt Behavior

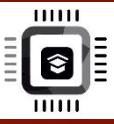


Exception Entry/Exit Sequence



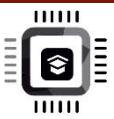
Exception Entry Sequence

1. Pending bit set
2. Stacking and Vector fetch.
3. Entry into the handler and Active bit set
4. clears the pending status(processor does it automatically)
5. Now processor mode changed to handler mode.
6. Now handler code is executing .
7. The MSP will be used for any stack operations inside the handler.



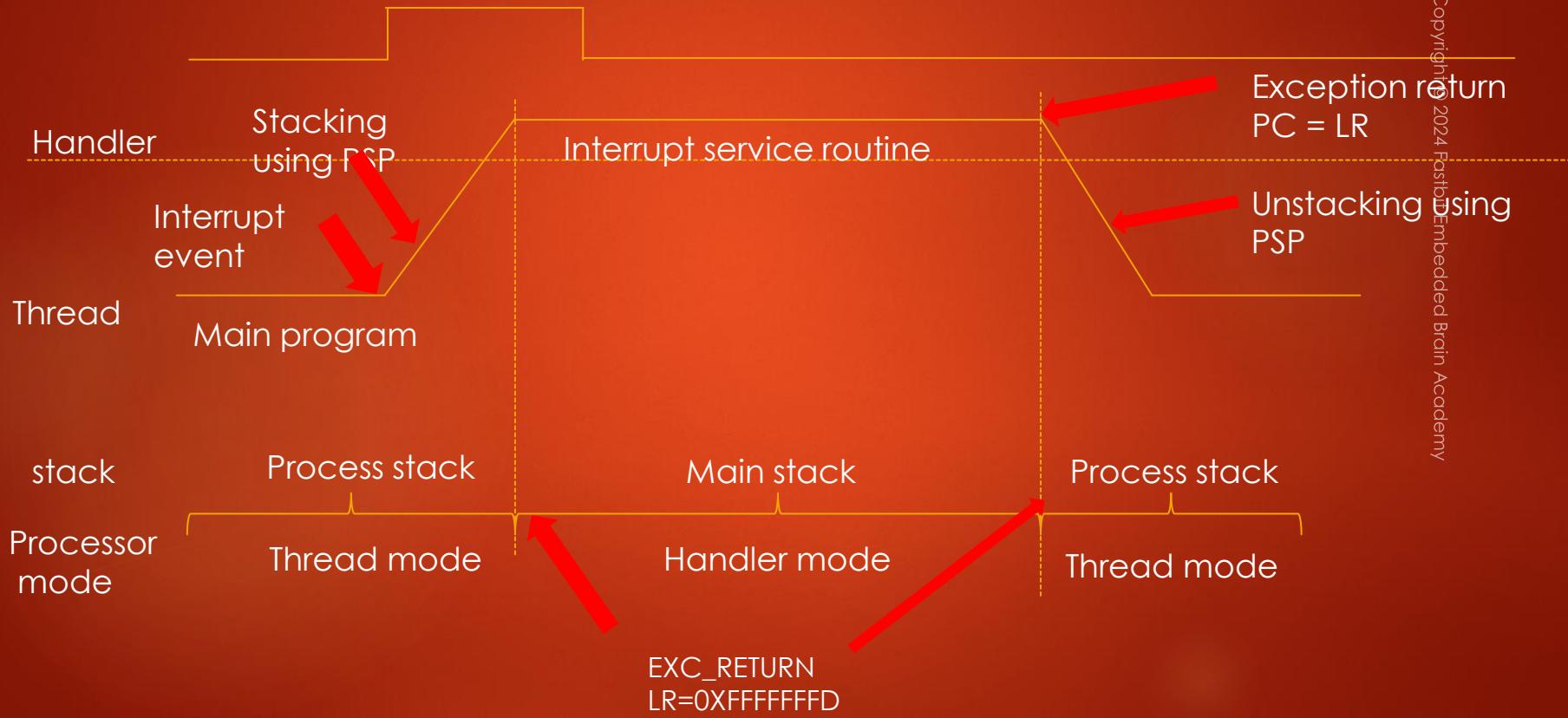
Exception Exit sequence

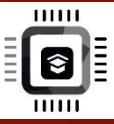
- ✓ In Cortex-M3/M4 processors the exception return mechanism is triggered using a special return address called EXC_RETURN.
- ✓ EXC_RETURN is generated during exception entry and is stored in the LR.
- ✓ When EXC_RETURN is written to PC it triggers the exception return.



Exception Entry/Exit Sequence

Copyright © 2024 FastBitLab Embedded Brain Academy

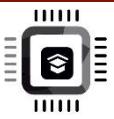




EXC_RETURN

When it is generated ??

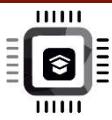
During an exception handler entry , the value of the return address(PC) Is not stored in the LR as it is done during calling of a normal C function. Instead The exception mechanism stores the special value called **EXC_RETURN in LR**.



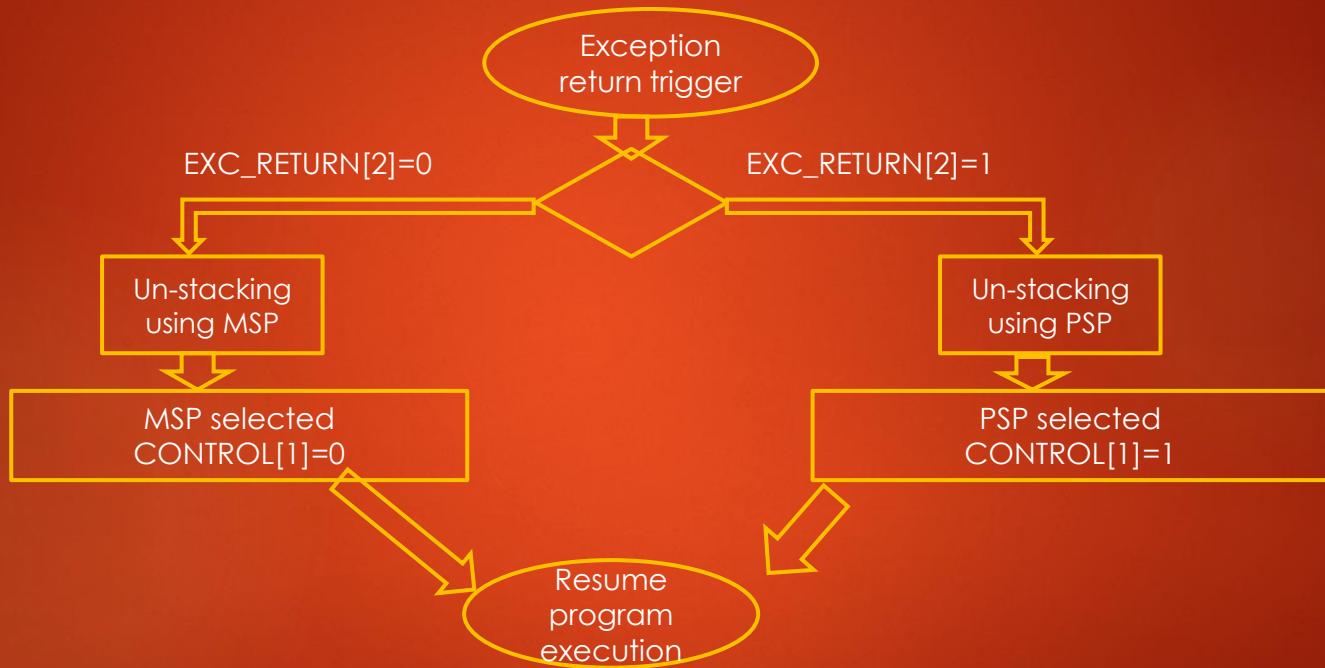
EXC_RETURN Contd.

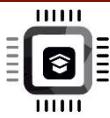
Decoding EXC_RETURN value

Bits	Descriptions	Values
31:28	EXC_RETURN indicator	0xF
27:5	Reserved(all 1)	0xFFFFFFFF
4	Stack frame type	always 1 when floating point unit is not available.
3	Return mode	1= return to thread mode 0 = return to handler mode
2	Return stack	1= return with PSP 0=return with MSP
1	Reserved	0
0	Reserved	1



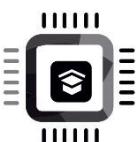
EXC RETURN Contd.





EXC_RETURN Possible Values

EXC_RETURN	Description
0xFFFFFFFF1	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
0xFFFFFFFF9	Return to Thread mode. Exception Return get state from the main stack. Execution uses MSP after return.
0xFFFFFFFFD	Return to Thread mode. Exception return gets state from the process stack. Execution uses PSP after return.
All other values	Reserved.



ARM cortex M3/M4 fault handlers

What is a fault?

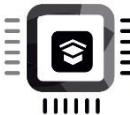
The fault is an exception generated by the processor(system exception) to indicate an error.



ARM cortex M3/M4 fault handlers

Why fault happens?

- ✓ faults happen because of programmers handling processor by violating the design rules or may be due to interfaces with which the processor deals
- ✓ Whenever a fault happens, internal processor registers will be updated to record the type of fault, the address of instruction at which the fault happened, and if an associated exception is enabled, the exception handler will be called by the processor
- ✓ In the exception handler programmers may implement the code to report, resolve, or recover from the fault.
- ✓ For example, if your code tries to divide a number by zero, then divide by 0 fault will be raised from the hardware, which will invoke usage fault exception handler (if enabled). In the exception handler, you may make certain decisions to get rid of the problem, like closing the task. Etc.
- ✓ Most of the time, fault happens by programmer's code not adhering to processor programming guidelines.



Summary of ARM cortex Mx system exceptions

Table 2-16 Properties of the different exception types

Exception number ^a	IRQ number ^a	Exception type	Priority	Vector address or offset ^b	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	-
4	-12	MemManage	Configurable ^c	0x00000010	Synchronous
5	-11	BusFault	Configurable ^c	0x00000014	Synchronous when precise, asynchronous when imprecise
6	-10	UsageFault	Configurable ^c	0x00000018	Synchronous
7-10	-	Reserved	-	-	-



Summary of ARM cortex Mx system exceptions

Table 2-16 Properties of the different exception types (continued)

Exception number ^a	IRQ number ^a	Exception type	Priority	Vector address or offset ^b	Activation
11	-5	SVCall	Configurable ^c	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable ^c	0x00000038	Asynchronous
15	-1	SysTick	Configurable ^c	0x0000003C	Asynchronous
16	0	Interrupt (IRQ)	Configurable ^d	0x00000040 ^e	Asynchronous

- a. To simplify the software layer, the CMSIS only uses IRQ numbers and therefore uses negative values for exceptions other than interrupts. The IPSR returns the Exception number, see *Interrupt Program Status Register* on page 2-6.
- b. See *Vector table* for more information.
- c. See *System Handler Priority Registers* on page 4-21.
- d. See *Interrupt Priority Registers* on page 4-7.
- e. Increasing in steps of 4.



Different types of fault exceptions in Cortex Mx processor

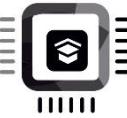
- Hard fault exception* {Enabled by default , non-configurable priority}
 - Usage fault exception
 - Mem manage fault exception
 - Bus fault exception
- 
- {Disabled by default , configurable priority}

* Hard-fault exception can be disabled by code using FAULTMASK register



Causes of fault

- ✓ Divide by zero (if enabled)
- ✓ Undefined instruction
- ✓ Attempt to execute code from memory region which is marked as execute never (XN) to prevent code injection
- ✓ MPU guarded memory region access violation by the code
- ✓ Unaligned data access (if enabled)
- ✓ Returning to thread mode keeping active interrupt alive
- ✓ Bus error (example no response from memory device (e.g., SDRAM))
- ✓ Executing SVC instruction inside SVC handler or calling a function in SVC handler which eventually execute hidden SVC instruction
- ✓ Debug monitor settings and related exceptions

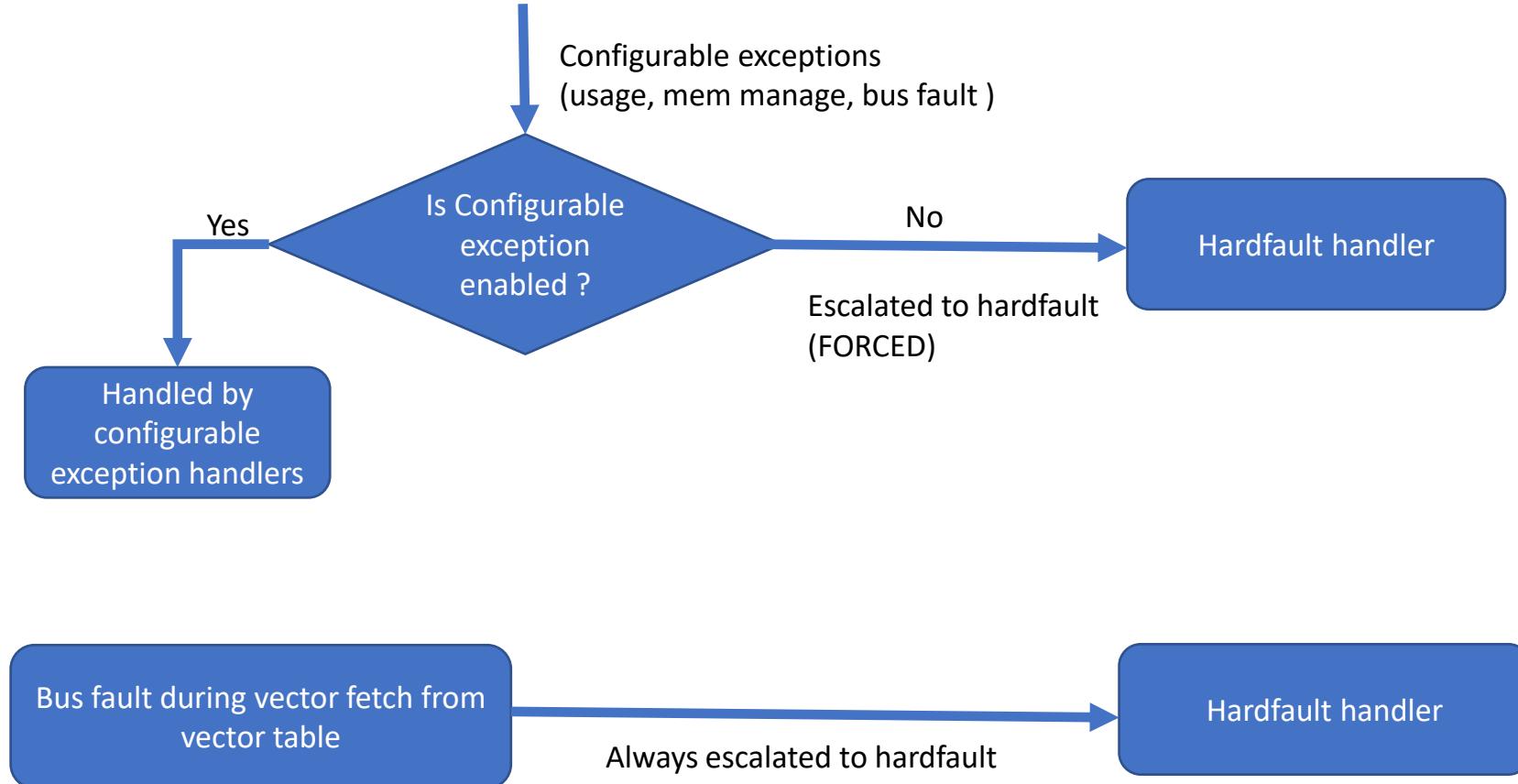


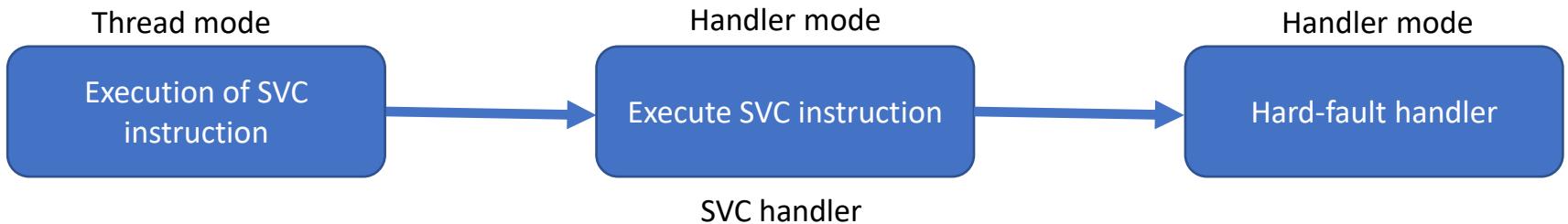
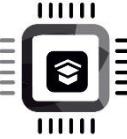
Hard-fault exception

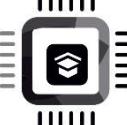
A HardFault is an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism. It has 3rd highest fixed priority (-1) after reset and NMI meaning it has higher priority than any exception with configurable priority

Causes

- 1) Escalation of configurable fault exceptions
- 2) Bus error returned during a vector fetch
- 3) Execution of break point instruction when both halt mode and debug monitor is disabled
- 4) Executing SVC instruction inside SVC handler







Escalation of configurable fault exceptions

- We know that Usage fault exception, Mem manage fault exception, Bus fault exception is configurable exceptions
- When these exceptions are disabled or if the priority of these exceptions is same or lower than the current priority and if any fault events happen which are related to these exceptions, then it will be forced or escalated as hard fault exceptions
- The HFSR gives information about events that activate the HardFault handler



4.3.11 HardFault Status Register

The HFSR gives information about events that activate the HardFault handler. See the register summary in [Table 4-12 on page 4-11](#) for its attributes.

This register is read, write to clear. This means that bits in the register read normally, but writing 1 to any bit clears that bit to 0. The bit assignments are:

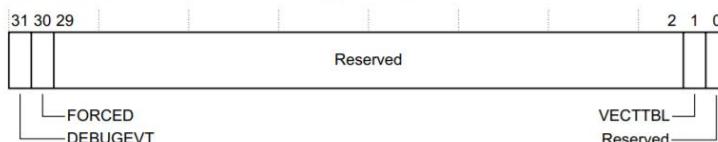


Table 4-28 HFSR bit assignments

Bits	Name	Function
[31]	DEBUGEV'T	Reserved for Debug use. When writing to the register you must write 0 to this bit, otherwise behavior is Unpredictable.
[30]	FORCED	Indicates a forced hard fault, generated by escalation of a fault with configurable priority that cannot be handles, either because of priority or because it is disabled: 0 = no forced HardFault 1 = forced HardFault. When this bit is set to 1, the HardFault handler must read the other fault status registers to find the cause of the fault.
[29:2]	-	Reserved.
[1]	VECTTBL	Indicates a BusFault on a vector table read during exception processing: 0 = no BusFault on vector table read 1 = BusFault on vector table read. This error is always handled by the hard fault handler. When this bit is set to 1, the PC value stacked for the exception return points to the instruction that was preempted by the exception.
[0]	-	Reserved.



Mem manage fault exception

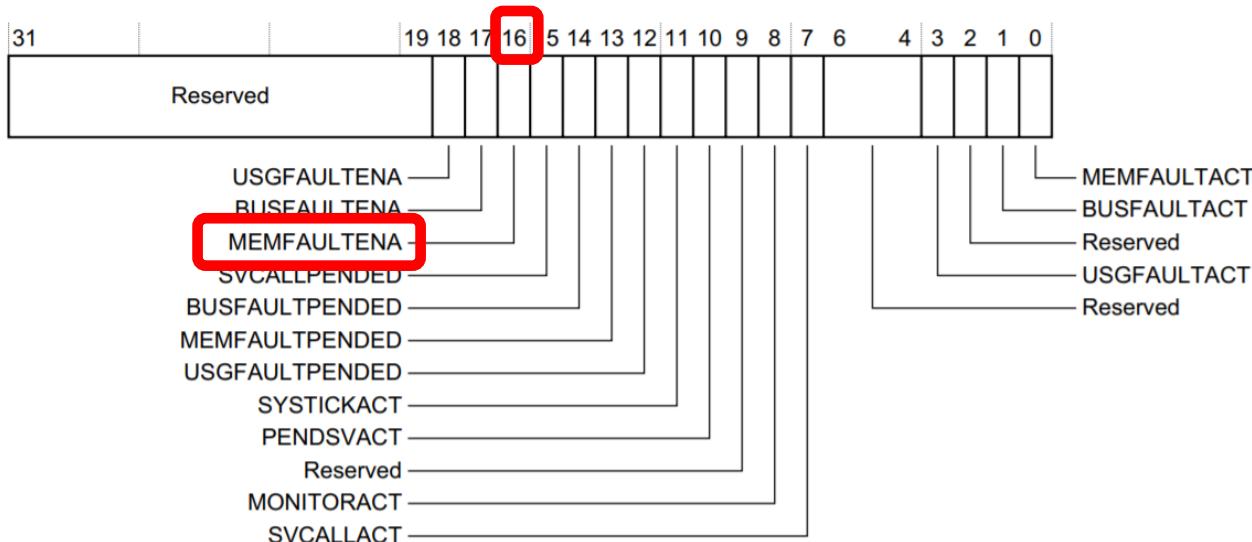
- This is a configurable fault exception. Disabled by default
- You can enable this exception by configuring the processor register “System Handler Control and State Register(SHCSR)”
- When mem mange fault happens mem manage fault exception handler will be executed by the processor
- Priority of this fault exception is configurable

4.3.9 System Handler Control and State Register

The SHCSR enables the system handlers, and indicates:

- the pending status of the BusFault, MemManage fault, and SVC exceptions
- the active status of the system handlers.

See the register summary in [Table 4-12 on page 4-11](#) for the SHCSR attributes. The bit assignments are:





Mem manage fault exception

Causes :

1. As its name indicates this fault exception triggers when memory access violation is detected (access permission by the processor or MPU)
2. Unprivileged thread mode code (such as user application or RTOS task) tries to access a memory region which is marked as “privileged access only ” by the MPU
3. Writing to memory regions which are marked as read-only by the MPU4.
4. This fault can also be triggered when trying to execute program code from “peripheral ” memory regions. Peripheral memory regions are marked as XN(eXecute Never) regions by the processor design to avoid code injection attacks through peripherals.



2.2.3 Behavior of memory accesses

The behavior of accesses to each region in the memory map is:

Table 2-11 Memory access behavior

Address range	Memory region	Memory type ^a	XN ^a	Description
0x00000000-0x1FFFFFFF	Code	Normal	-	Executable region for program code. You can also put data here.
0x20000000-0x3FFFFFFF	SRAM	Normal	-	Executable region for data. You can also put code here. This region includes bit band and bit band alias areas, see Table 2-13 on page 2-16 .
0x40000000-0x5FFFFFFF	Peripheral	Device	XN	This region includes bit band and bit band alias areas, see Table 2-14 on page 2-16 .
0x60000000-0x9FFFFFFF	External RAM	Normal	-	Executable region for data.
0xA0000000-0xDFFFFFFF	External device	Device	XN	External Device memory.
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly-ordered	XN	This region includes the NVIC, System timer, and system control block.
0xE0100000-0xFFFFFFFF	Device	Device	XN	Implementation-specific.

a. See [Memory regions, types and attributes on page 2-12](#) for more information.

- **eXecute Never(XN)** : The processor prevents instruction accesses. A fault exception is generated only on execution of an instruction from an XN region



Bus-fault exception

- This is a configurable fault exception. Disabled by default
- You can enable this exception by configuring the processor register “System Handler Control and State Register(SHCSR)”
- When Bus-fault happens, the processor executes bus fault exception handler
- The priority of this fault exception is configurable

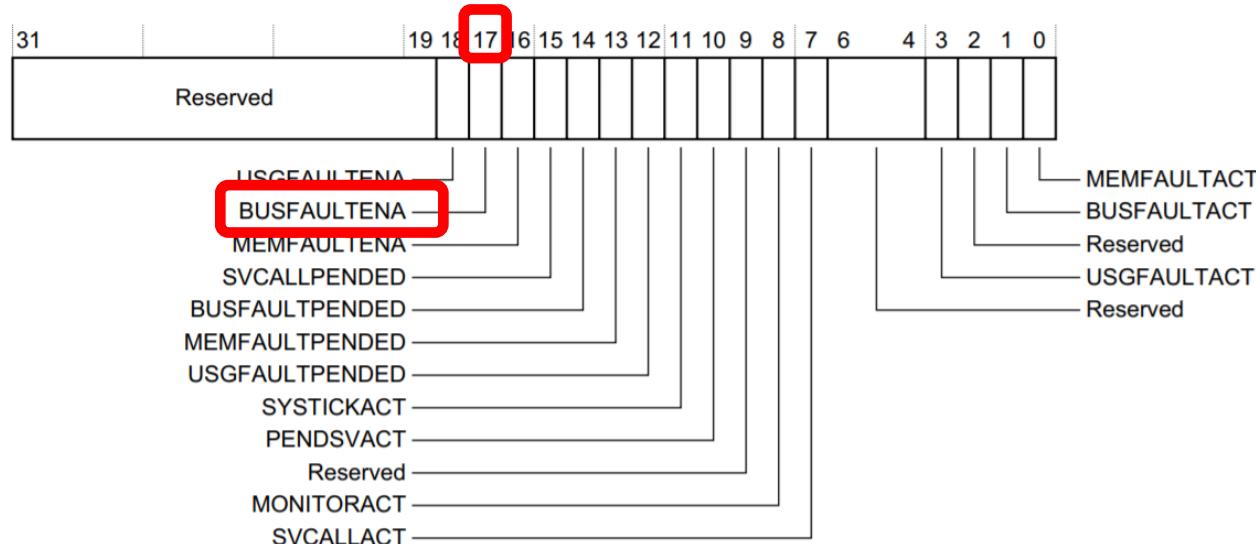


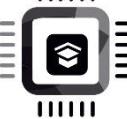
4.3.9 System Handler Control and State Register

The SHCSR enables the system handlers, and indicates:

- the pending status of the BusFault, MemManage fault, and SVC exceptions
- the active status of the system handlers.

See the register summary in [Table 4-12 on page 4-11](#) for the SHCSR attributes. The bit assignments are:

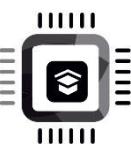




Bus-fault exception

Causes

- ✓ Due to error response returned by the processor bus interfaces during access to memory devices
 - During instruction fetch
 - During data read or write to memory devices
- ✓ If bus error happens during vector fetch, it will be escalated to a hard fault even if bus fault exception is enabled
- ✓ Memory device sends error response when the processor bus interface tries to access invalid or restricted memory locations which could generate a bus fault
- ✓ When the device is not ready to accept memory transfer
- ✓ You may encounter such issues when you play with external memories such as SDRAM connected via DRAM controllers
- ✓ Unprivileged access to the private peripheral bus



Usage fault exception

- This is a configurable fault exception. Disabled by default
- You can enable this exception by configuring the processor register “System Handler Control and State Register(SHCSR)”
- When usage fault happens, the processor executes usage fault exception handler
- Priority of this fault exception is configurable

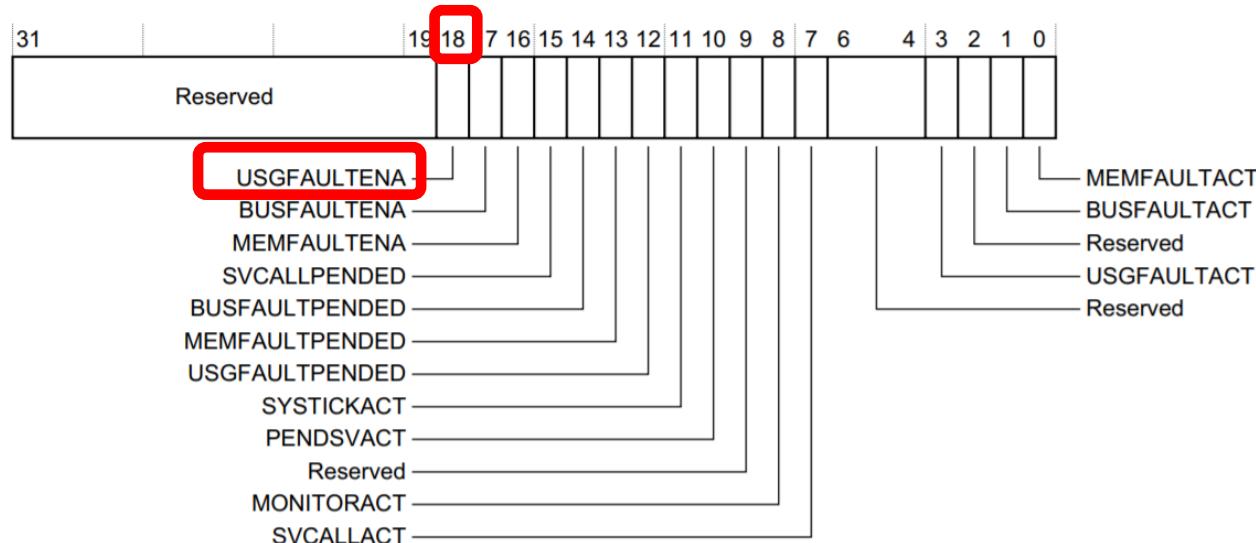


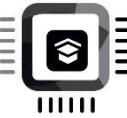
4.3.9 System Handler Control and State Register

The SHCSR enables the system handlers, and indicates:

- the pending status of the BusFault, MemManage fault, and SVC exceptions
- the active status of the system handlers.

See the register summary in [Table 4-12 on page 4-11](#) for the SHCSR attributes. The bit assignments are:

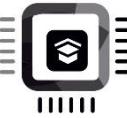




Usage fault exception

Causes :

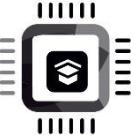
- ✓ Execution of undefined instruction (Cortex M4 supports only thumb ISA, so executing any instruction outside this ISA(like ARM ISA) would result in a fault)
- ✓ Executing floating point instruction keeping floating point unit disabled
- ✓ Trying to switch to ARM state to execute ARM ISA instructions. The T bit of the processor decides ARM state or THUMB state. For cortex M it should be maintained at 1. Making T bit 0 (may happen during function call using function pointers whose 0th bit is not maintained as 1) would result in a fault
- ✓ Trying to return to thread mode when an exception/interrupt is still active
- ✓ Unaligned memory access with multiple load or multiple store instructions
- ✓ Attempt to divide by zero (if enabled, by default divide by zero results in zero)
- ✓ For all unaligned data access from memory (only if enabled, otherwise cortex m supports unaligned data access)



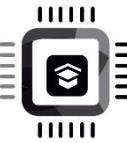
Exercise

Write a program to enable all configurable fault exceptions , implement the fault exception handlers and cause the fault by following method

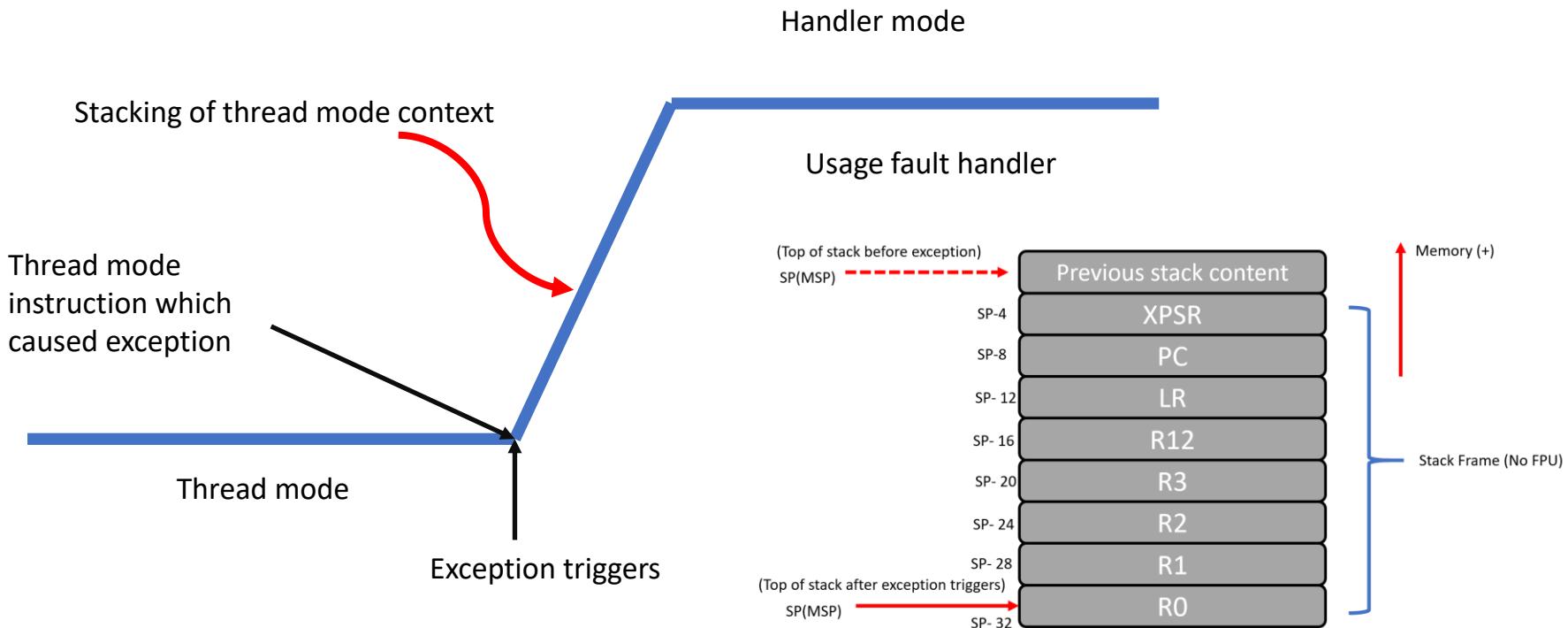
1. Execute an undefined instruction
2. Divide by zero
3. Try executing instruction from peripheral region
4. Executing SVC inside the SVC handler
5. Executing SVC instruction inside the interrupt handler whose priority is same or lesser than SVC handler

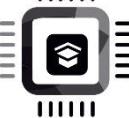


Detecting cause of a fault



Analyzing stack frame

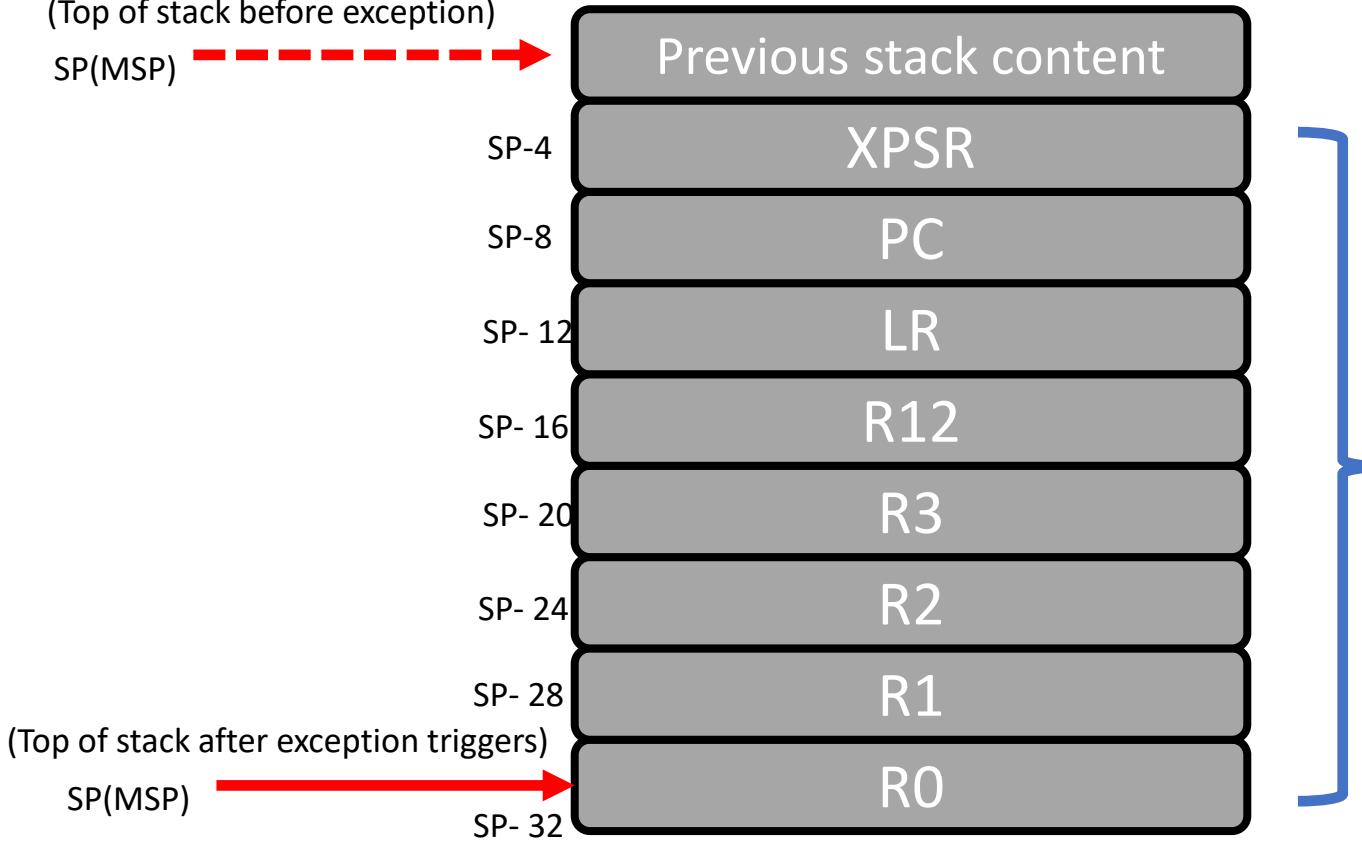




(Top of stack before exception)

SP(MSP)

Memory (+)





__attribute__((naked)) functions

- This attribute tells the compiler that the function is an embedded assembly function. You can write the body of the function entirely in assembly code using `__asm` statements.
- The compiler does not generate prologue and epilogue sequences for functions with `__attribute__((naked))`.
- Use naked functions only to write some assembly instructions(`__asm` statements) . Mixing ‘C’ code might not work properly

Ref : <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0774g/jhg1476893564298.html>



Table 6.1: Table 2, Core registers and AAPCS usage

Regis- ter	Syn- onym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable-register 4.
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.



Fault status and address information

- When a fault happens, inside the fault handler, you can check a couple of fault status and address information registers to get more details about the fault and the instruction address at which the fault happened. This will be helpful during debugging.

Table 2-19 Fault status and fault address registers

Handler	Status register name	Address register name	Register description
HardFault	HFSR	-	<i>HardFault Status Register</i> on page 4-30
MemManage	MMFSR	MMFAR	<i>MemManage Fault Status Register</i> on page 4-25 <i>MemManage Fault Address Register</i> on page 4-30
BusFault	BFSR	BFAR	<i>BusFault Status Register</i> on page 4-26 <i>BusFault Address Register</i> on page 4-31
UsageFault	UFSR	-	<i>UsageFault Status Register</i> on page 4-28



Fault	Handler	Bit name	Fault status register
Bus error on a vector read	HardFault	VECTTBL	<i>HardFault Status Register on page 4-30</i>
Fault escalated to a hard fault		FORCED	
MPU or default memory map mismatch:	MemManage	-	-
on instruction access		IACCVIOL ^a	<i>MemManage Fault Address Register on page 4-30</i>
on data access		DACCVIOL	
during exception stacking		MSTKERR	
during exception unstacking		MUNSKERR	
during lazy floating-point state preservation		MLSPERR	
Bus error:	BusFault	-	-
during exception stacking		STKERR	<i>BusFault Status Register on page 4-26</i>
during exception unstacking		UNSTKERR	
during instruction prefetch		IBUSERR	
during lazy floating-point state preservation		LSPERR	
Precise data bus error		PRECISERR	
Imprecise data bus error		IMPRECISERR	

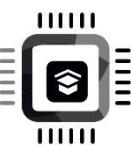


Table 2-18 Faults (continued)

Fault	Handler	Bit name	Fault status register
Attempt to access a coprocessor	UsageFault	NOCP	<i>UsageFault Status Register</i> on page 4-28
Undefined instruction		UNDEFINSTR	
Attempt to enter an invalid instruction set state ^b		INVSTATE	
Invalid EXC_RETURN value		INVPC	
Illegal unaligned load or store		UNALIGNED	
Divide By 0		DIVBYZERO	

- a. Occurs on an access to an XN region even if the processor does not include an MPU or if the MPU is disabled.
- b. Attempting to use an instruction set other than the Thumb instruction set or returns to a non load/store-multiple instruction with ICI continuation.

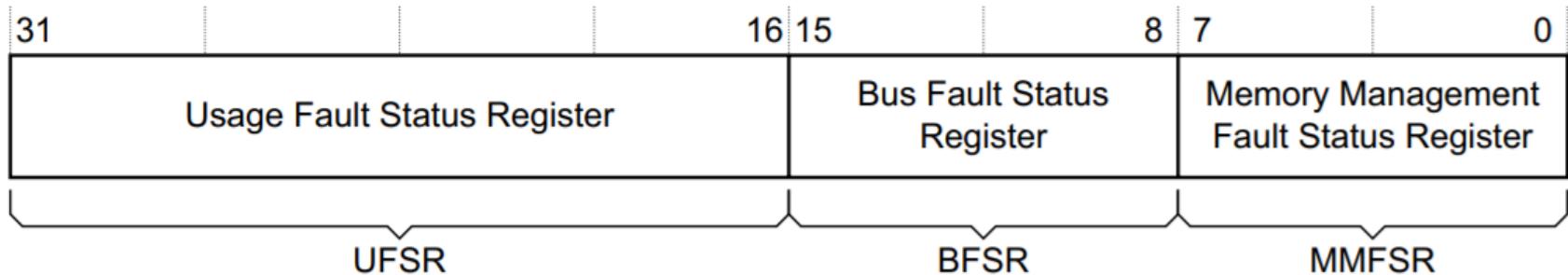
4.3.11 HardFault Status Register

The HFSR gives information about events that activate the HardFault handler. See the register summary in [Table 4-12 on page 4-11](#) for its attributes.

This register is read, write to clear. This means that bits in the register read normally, but writing 1 to any bit clears that bit to 0. The bit assignments are:



Hard Fault Status Register (HFSR)



Configurable Fault Status Register(CFSR)

This register indicates the cause of a MemManage fault, BusFault, or UsageFault



Fault address information registers

- These registers contain address of the location that generated the fault

Table 2-19 Fault status and fault address registers

Handler	Status register name	Address register name	Register description
HardFault	HFSR	-	<i>HardFault Status Register</i> on page 4-30
MemManage	MMFSR	MMFAR	<i>MemManage Fault Status Register</i> on page 4-25 <i>MemManage Fault Address Register</i> on page 4-30
BusFault	BFSR	BFAR	<i>BusFault Status Register</i> on page 4-26 <i>BusFault Address Register</i> on page 4-31
UsageFault	UFSR	-	<i>UsageFault Status Register</i> on page 4-28



Mem Manage Fault Address Register

The MMFAR contains the address of the location that generated a MemManage fault. See the register summary in [Table 4-12 on page 4-11](#) for its attributes. The bit assignments are:

Table 4-29 MMFAR bit assignments

Bits	Name	Function
[31:0]	ADDRESS	When the MMARVALID bit of the MMFSR is set to 1, this field holds the address of the location that generated the MemManage fault



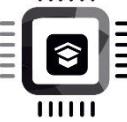
Bus Fault Address Register (BFAR)

BusFault Address Register

The BFAR contains the address of the location that generated a BusFault. See the register summary in [Table 4-12 on page 4-11](#) for its attributes. The bit assignments are:

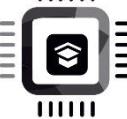
Table 4-30 BFAR bit assignments

Bits	Name	Function
[31:0]	ADDRESS	When the BFARVALID bit of the BCSR is set to 1, this field holds the address of the location that generated the BusFault



Fault handling and Analysis

- Implement a hard fault handler , analyze and print the cause of the hard fault exception
- Print the required status and address information register contents and print the last stacked stack frame



Error reporting when fault happens

- Implement the handler which takes some remedial actions
- Implement a user call back to report errors
- Reset the microcontroller/Processor
- For an OS environment , the task that triggered the fault can be terminated and restarted
- Report the fault status register and fault address register values
- Report additional information of stack frame through debug interface such as printf



Exceptions for system-level services

- ARM cortex Mx processor supports 2 important system-level service exceptions. **SVC(SuperVisor Call) and PendSV(Pendable Service)**
- Supervisory calls are typically used to request privileged operations or access to system resources from an operating system.
- SVC exception is mainly used in an OS environment. For example, A less privileged user task can trigger SVC exception to get system-level services(like accessing device drivers, peripherals) from the kernel of the OS
- PendSV is mainly used in an OS environment to carry out context switching between 2 or more tasks when no other exceptions are active in the system.

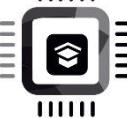


Summary of ARM cortex Mx system exceptions

Table 2-16 Properties of the different exception types (continued)

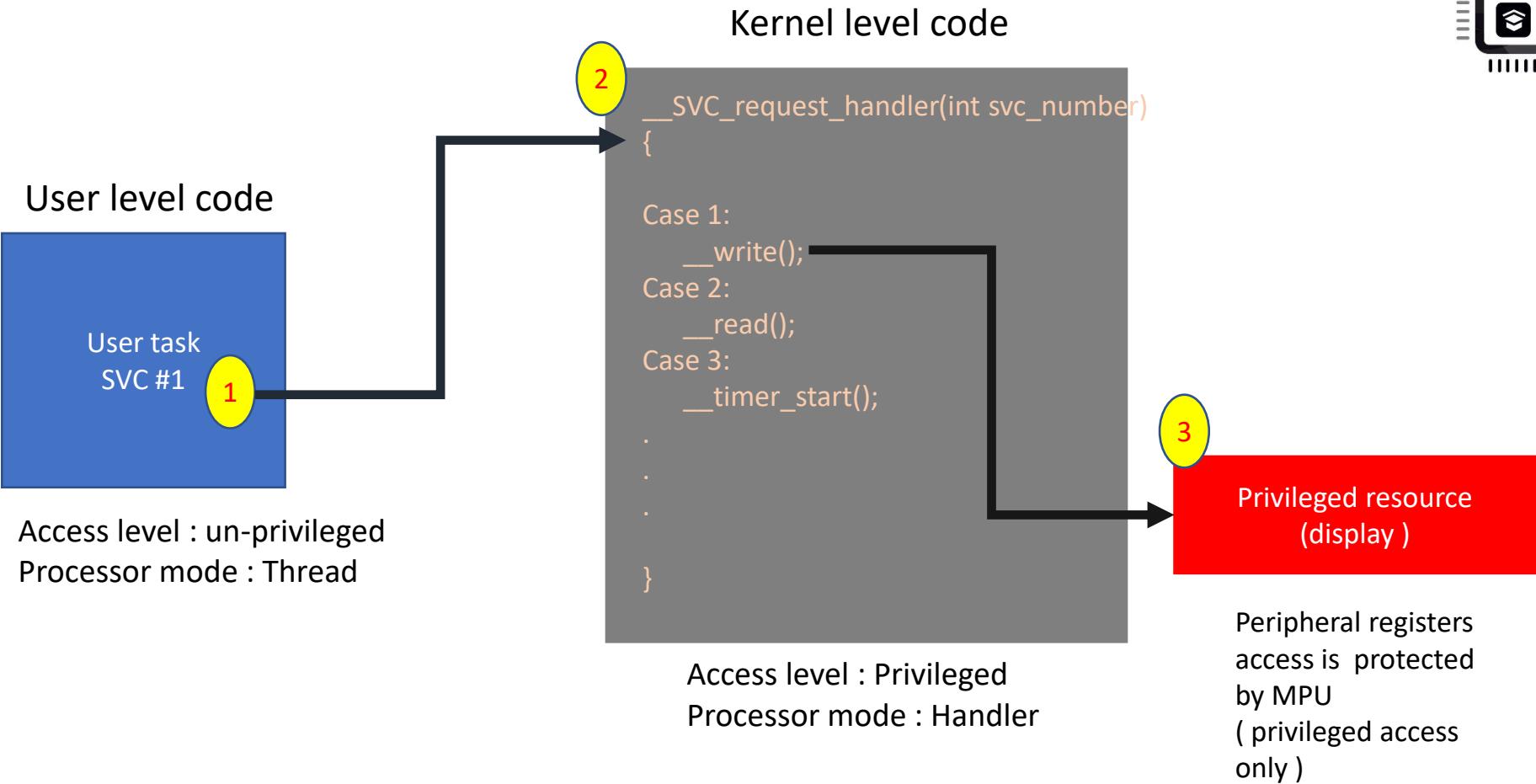
Exception number ^a	IRQ number ^a	Exception type	Priority	Vector address or offset ^b	Activation
11	-5	SVCall	Configurable ^c	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable ^c	0x00000038	Asynchronous
15	-1	SysTick	Configurable ^c	0x0000003C	Asynchronous
16	0	Interrupt (IRQ)	Configurable ^d	0x00000040 ^e	Asynchronous

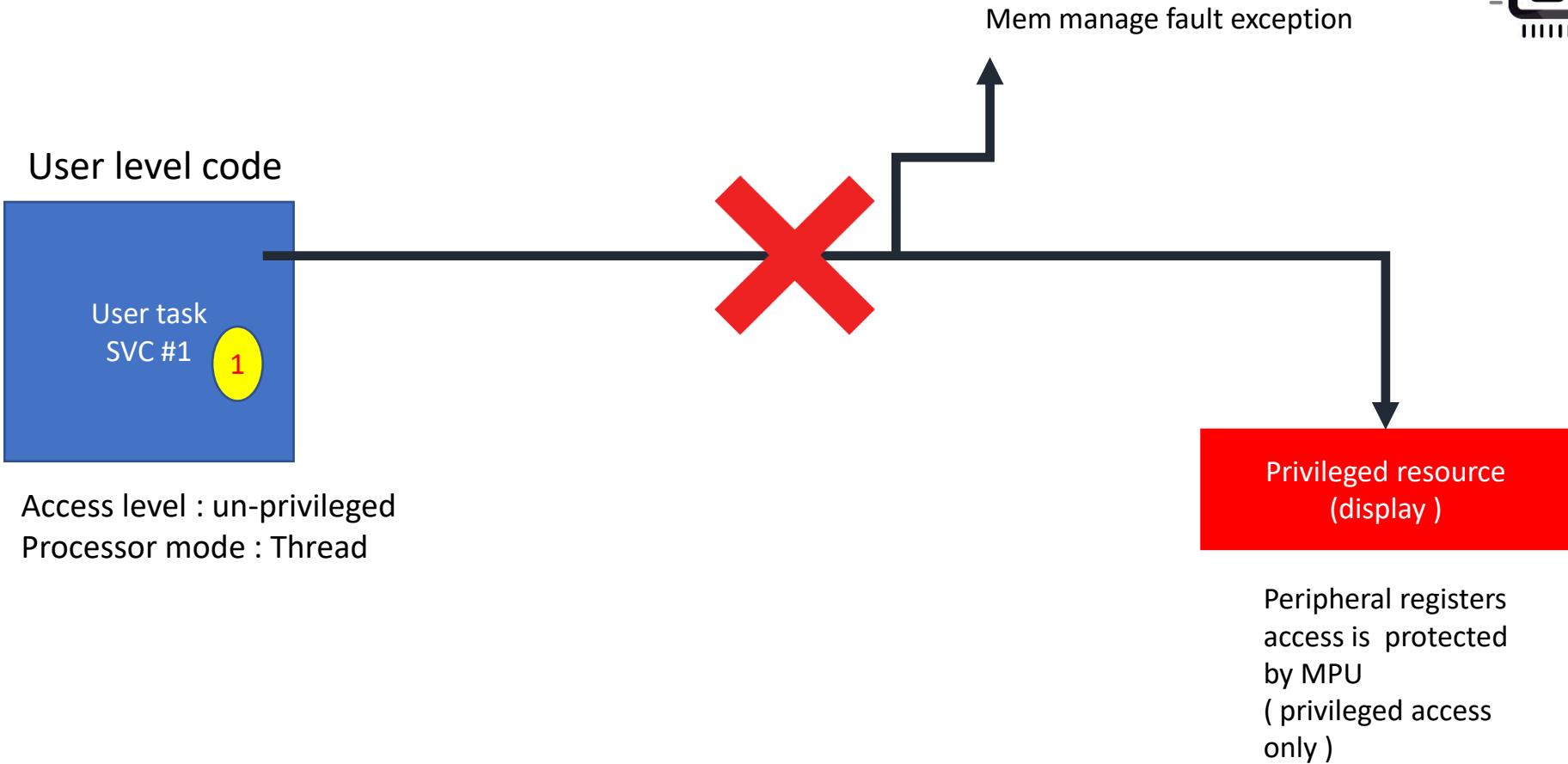
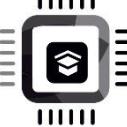
- a. To simplify the software layer, the CMSIS only uses IRQ numbers and therefore uses negative values for exceptions other than interrupts. The IPSR returns the Exception number, see *Interrupt Program Status Register* on page 2-6.
- b. See *Vector table* for more information.
- c. See *System Handler Priority Registers* on page 4-21.
- d. See *Interrupt Priority Registers* on page 4-7.
- e. Increasing in steps of 4.

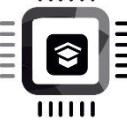


SVC (Supervisor Call) instruction

- SVC is a thumb ISA instruction which causes SVC exception
- In an RTOS scenario, user tasks can execute SVC instruction with an associated argument to make supervisory calls to seek privileged resources from the kernel code
- Unprivileged user tasks use the SVC instruction to change the processor mode to privileged mode to access privileged resources like peripherals
- SVC instruction is always used along with a number, which can be used to identify the request type by the kernel code
- The svc handler executes right after the SVC instruction(no delay. Unless a higher priority exception arrives at the same time)



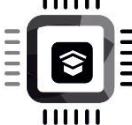




Methods to trigger SVC exception

There are two ways

- 1) Direct execution of SVC instruction with an immediate value
Example : ‘**SVC #0x04**’ in assembly (Using SVC instruction is very efficient in terms of latency)
- 2) Setting the exception pending bit in “**System Handler Control and State Register**” (uncommon method)

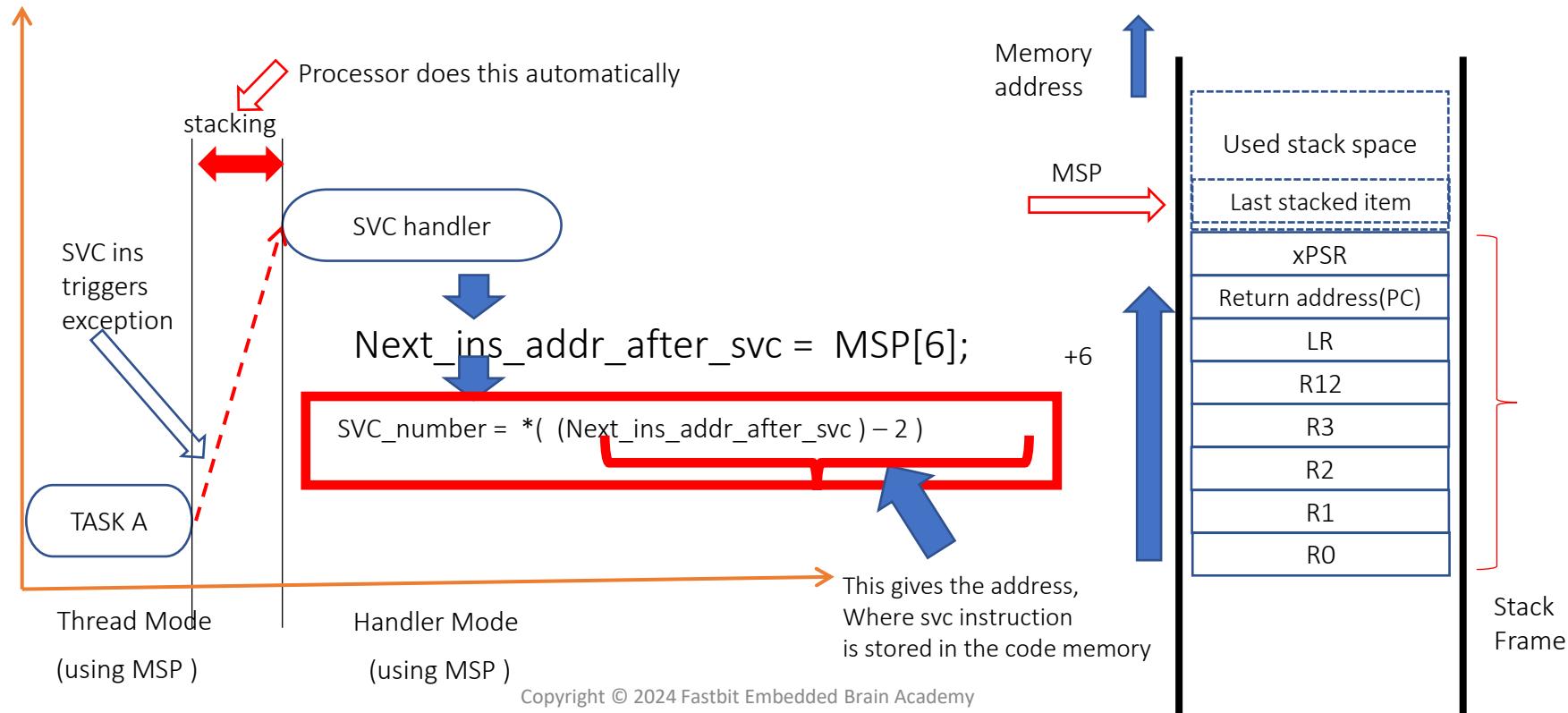


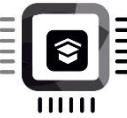
How to extract the SVC number

- The SVC instruction has a number embedded within it, often referred to as the SVC number
- In the SVC handler, you should fetch the opcode of the SVC instruction and then extract the SVC number
- To fetch the opcode of SVC instruction from program memory, we should have the value of PC(return address) where the user code had interrupted while triggering the SVC exception
- The value of the PC(Return address) where the user code had interrupted is stored in the stack as a part of exception entry sequence by the processor.



How To Extract The SVC Number





Exercise

Write a program to execute an SVC instruction from thread mode, implement the svc handler to print the SVC number used. Also, increment the SVC number by 4 and return it to the thread mode code and print it.

Hints :

- 1) Write a main() function where you should execute the SVC instruction with an argument. let's say SVC #0x5
- 2) Implement the SVC handler
- 3) In the SVC handler extract the SVC number and print it using printf
- 4) Increment the SVC number by 4 and return it to the thread mode

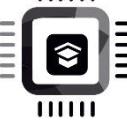


Exercise

Write a program to add, subtract, multiply, and divide 2 operands using SVC handler and return the result to the thread mode code and print the result. Thread mode code should pass 2 operands via the stack frame

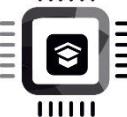
Use the SVC number to decide the operation

Service number	Operation
36	Addition
37	Subtraction
38	Multiplication
39	Division



PendSV Exception

- It is an exception type 14 and has a programmable priority level.
- This exception is triggered by setting its pending status by writing to the **“Interrupt Control and State Register”** of processor
- Triggering a pendSV system exception is a way of invoking the pre-emptive kernel to carry out the context switch in an OS environment
- In an OS environment, PendSV handler is set to the lowest priority level, and the PendSV handler carries out the context switch operation



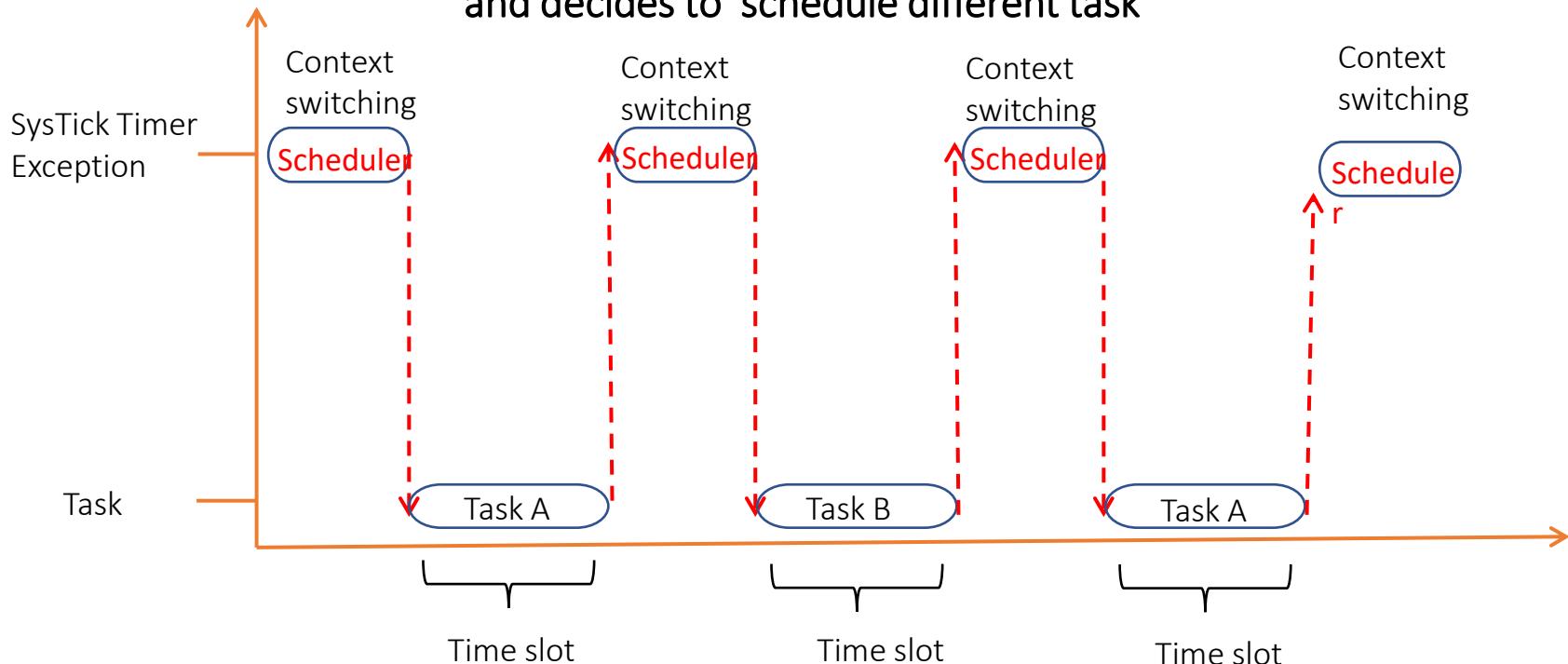
Typical use of PendSV

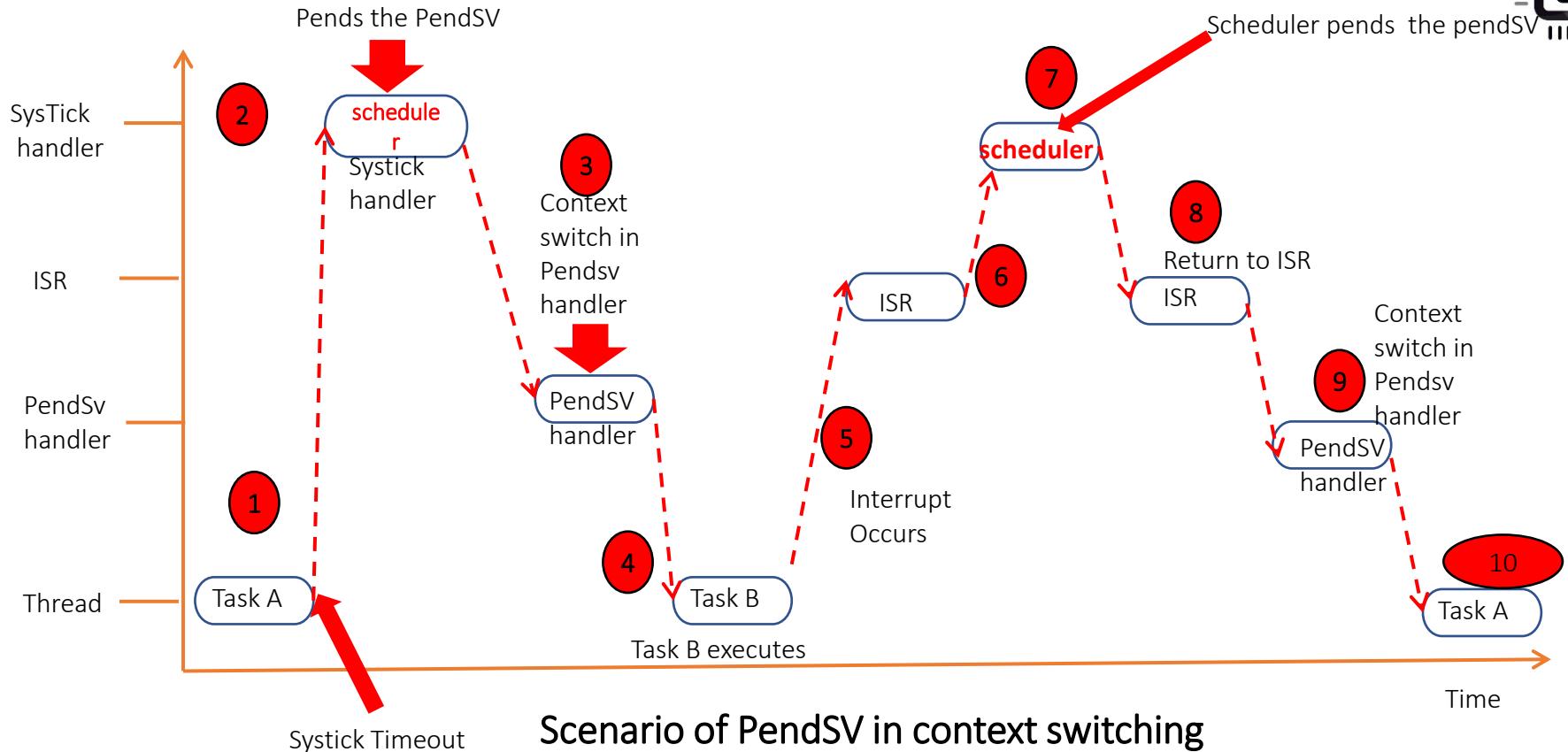
- ✓ Typically this exception is triggered inside a higher priority exception handler, and it gets executed when the higher priority handler finishes.
- ✓ Using this characteristic, we can schedule the PendSV exception handler to be executed after all the other interrupt processing tasks are done.
- ✓ This is very useful for a context switching operation, which is a crucial operation in various OS design.
- ✓ Using PendSV in context switching will be more efficient in an interrupt noisy environment.
- ✓ In an interrupt noisy environment, and we need to delay the context switching until all IRQ are executed.



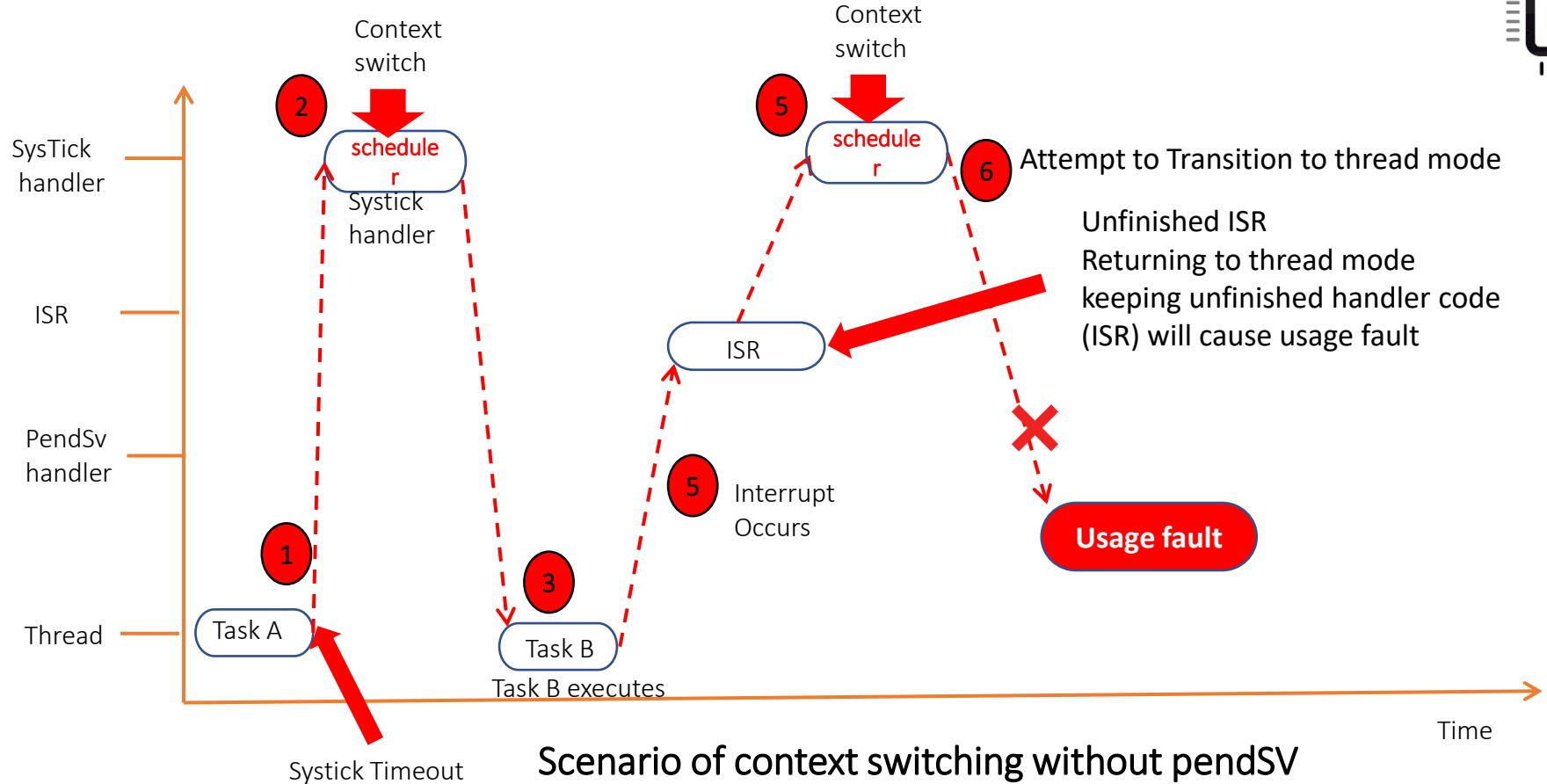
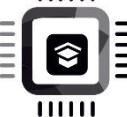
Context Switching

OS code runs on each systick timer exception
and decides to schedule different task





Scenario of PendSV in context switching





PendSV other use cases

- Offloading Interrupt processing
- If a higher priority handler is doing time-consuming work, then the other lower priority interrupts will suffer, and systems responsiveness may reduce. This can be solved using a combination of ISR and pendSV handler

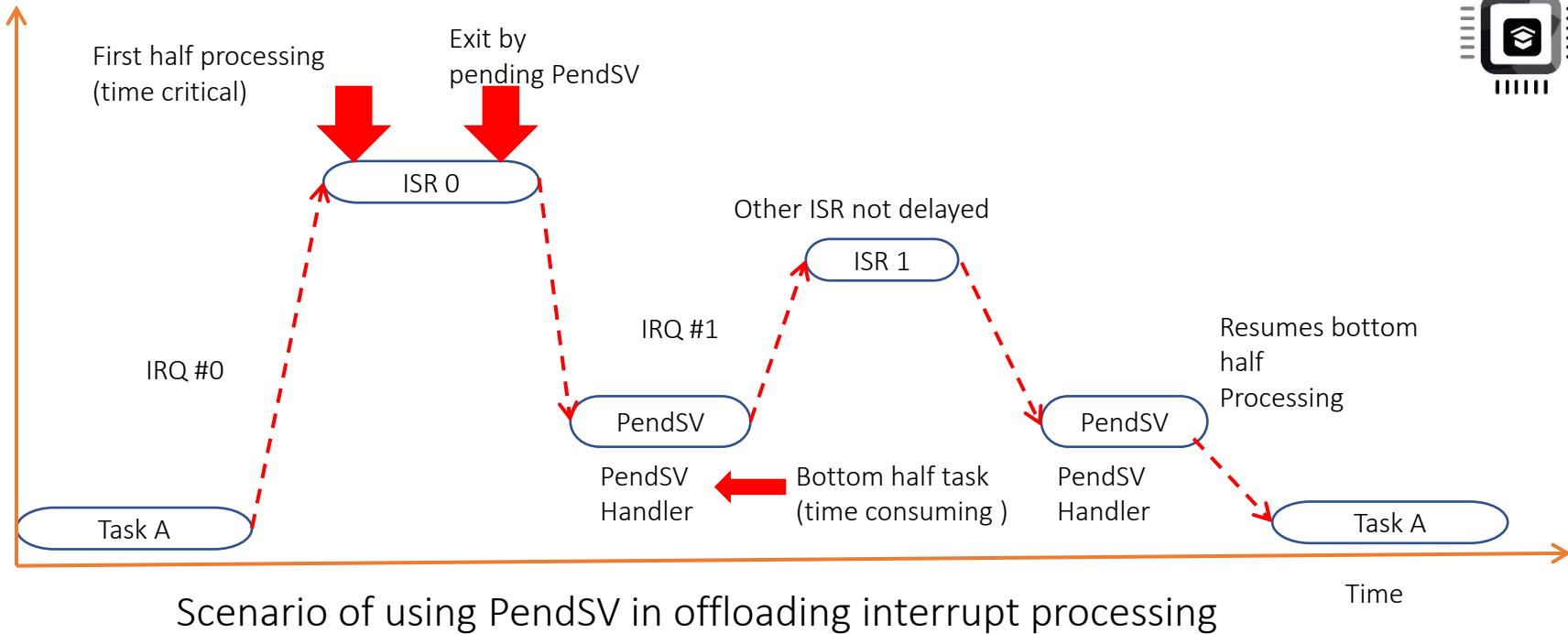
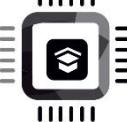


Offloading Interrupt Processing Using Pendsv

Interrupts may be serviced in 2 halves.

- 1) The first half is the time critical part that needs to be executed as a part of ISR.
- 2) The second half is called **bottom half**, is basically delayed execution where rest of the time-consuming work will be done .

So, PendSV can be used in these cases, to handle the second half execution by triggering it in the first half.





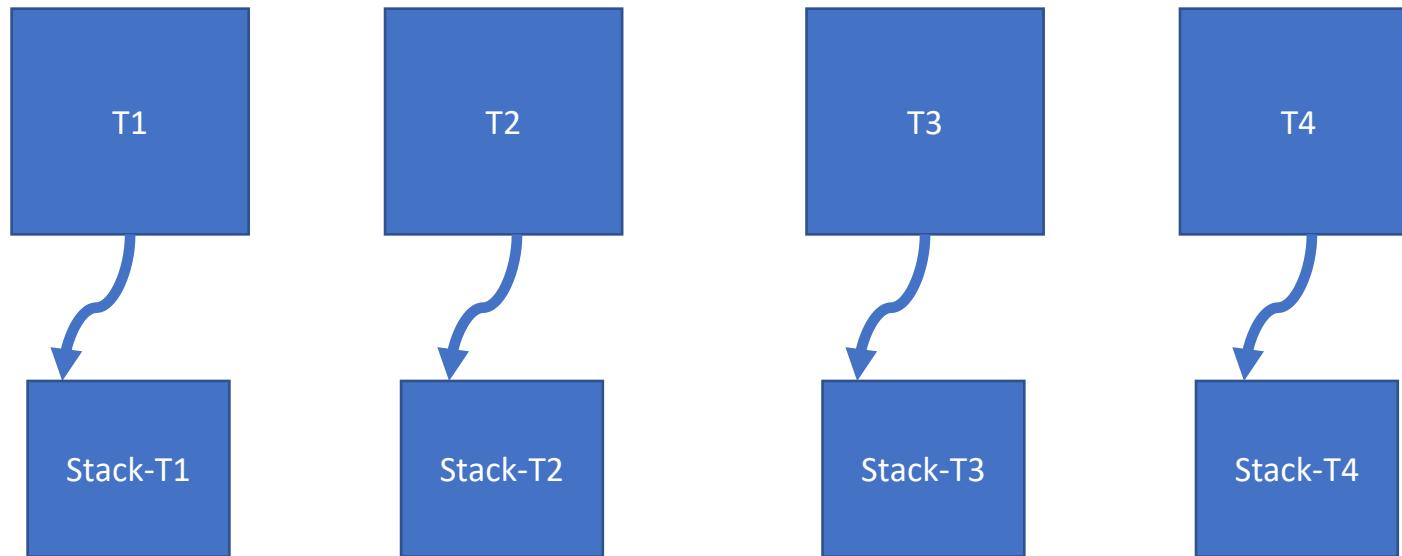
Implementing a scheduler

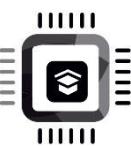
- Let's implement a scheduler which schedules multiple user tasks in a round-robin fashion by carrying out the context switch operation
- Round robin scheduling method is , time slices are assigned to each task in equal portions and in circular order
- First will use systick handler to carry out the context switch operation between multiple tasks
- Later will we change the code using pendSV handler



User tasks

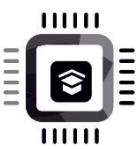
We will be using 4 user tasks in this application





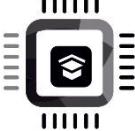
What is a task ?

- A task is nothing but a piece of code, or you can call it a ('C') function, which does a specific job when it is allowed to run on the CPU.
- A task has its own stack to create its local variables when it runs on the CPU. Also when scheduler decides to remove a task from CPU , scheduler first saves the context(state) of the task in task's private stack
- So, in summary, a piece code or a function is called a task when it is schedulable and never loses its 'state' unless it is deleted permanently.

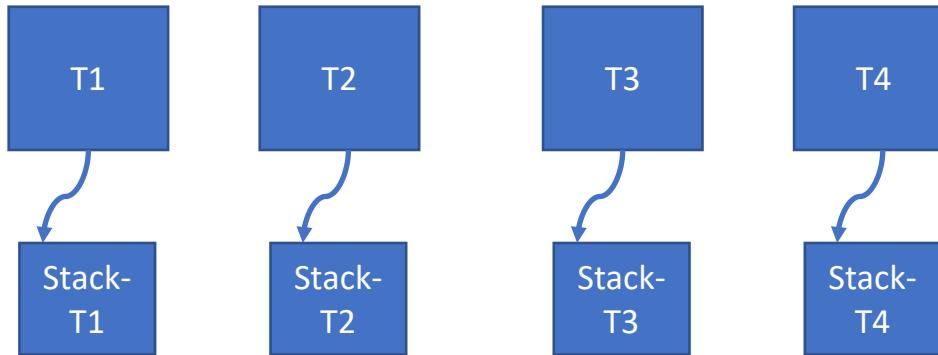


Let's code

Create 4 user tasks (basically never returning C functions)



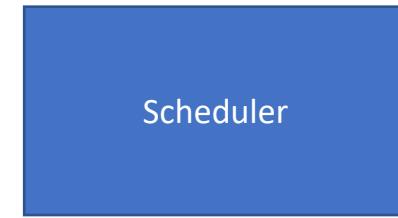
Stack pointer selection



PSP

Thread mode

(Systick handler / PendSV handler)



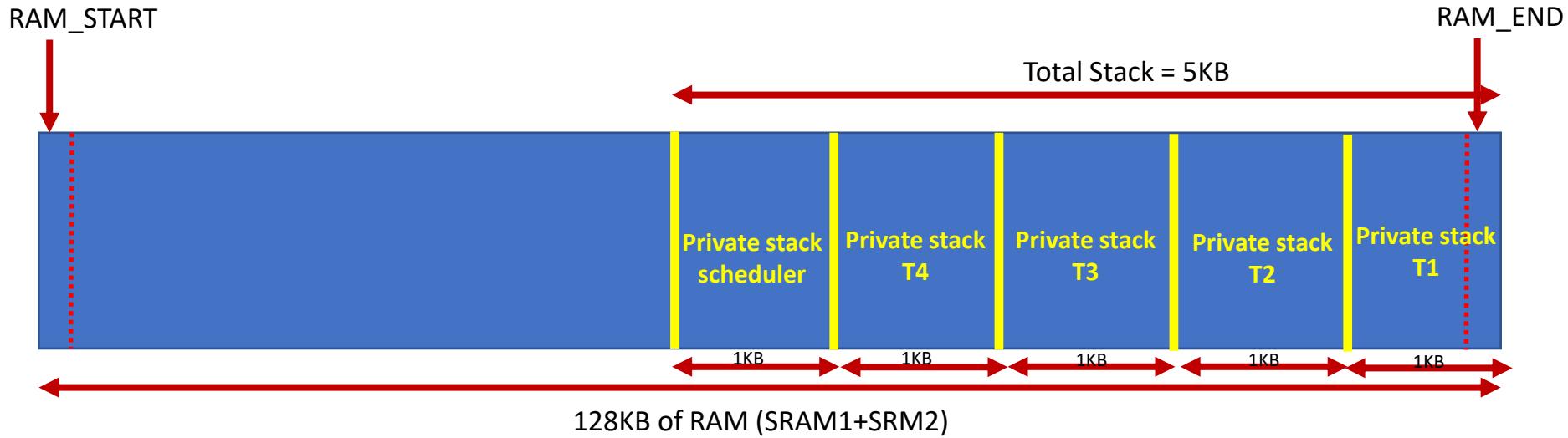
Scheduler

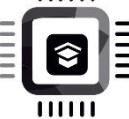
MSP

Handler mode

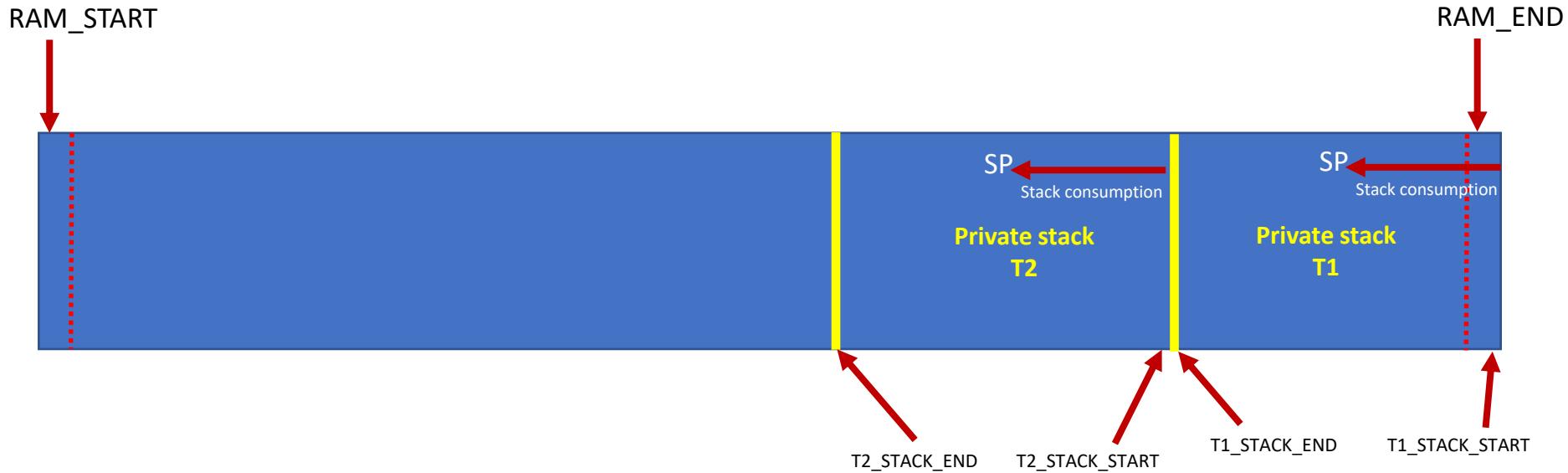


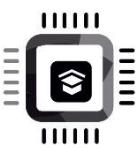
Stack assessment





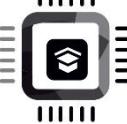
Stack assessment





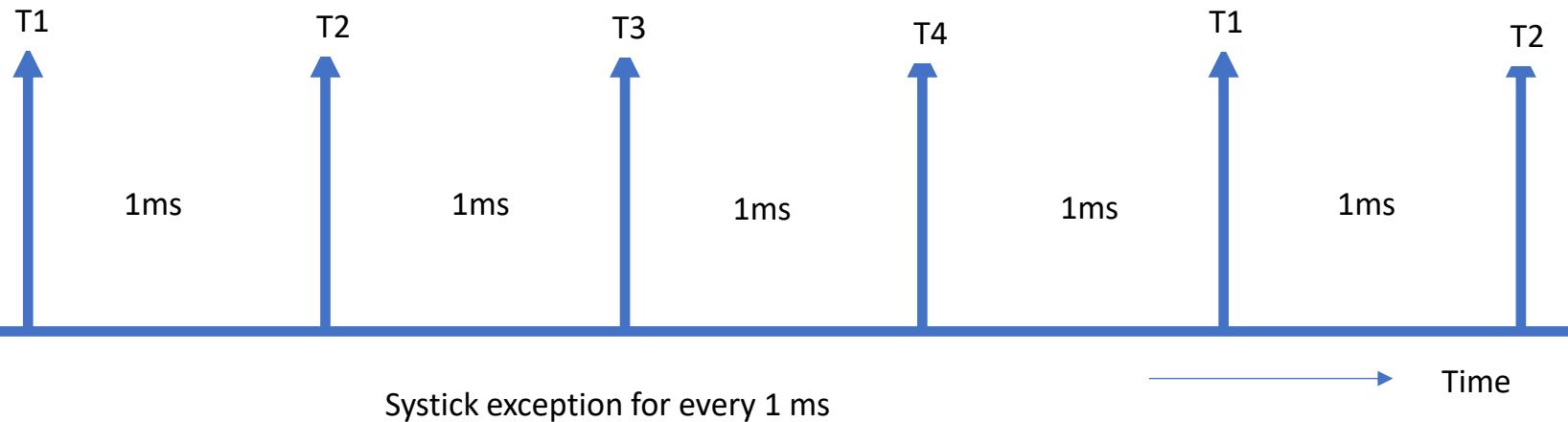
Let's code

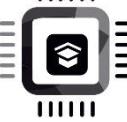
Reserve stack areas for all the tasks and scheduler



Scheduling policy selection

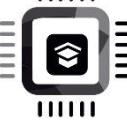
- We will be using round robin pre-emptive scheduling
- No task priority
- We will use SysTick timer to generate exception for every 1ms to run the scheduler code





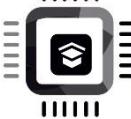
What is scheduling ?

- Scheduling is an algorithm which takes the decision of pre-empting a running task from the CPU and takes the decision about which task should run on the CPU next
- The decision could be based on many factors such as system load, the priority of tasks, shared resource access, or a simple round-robin method

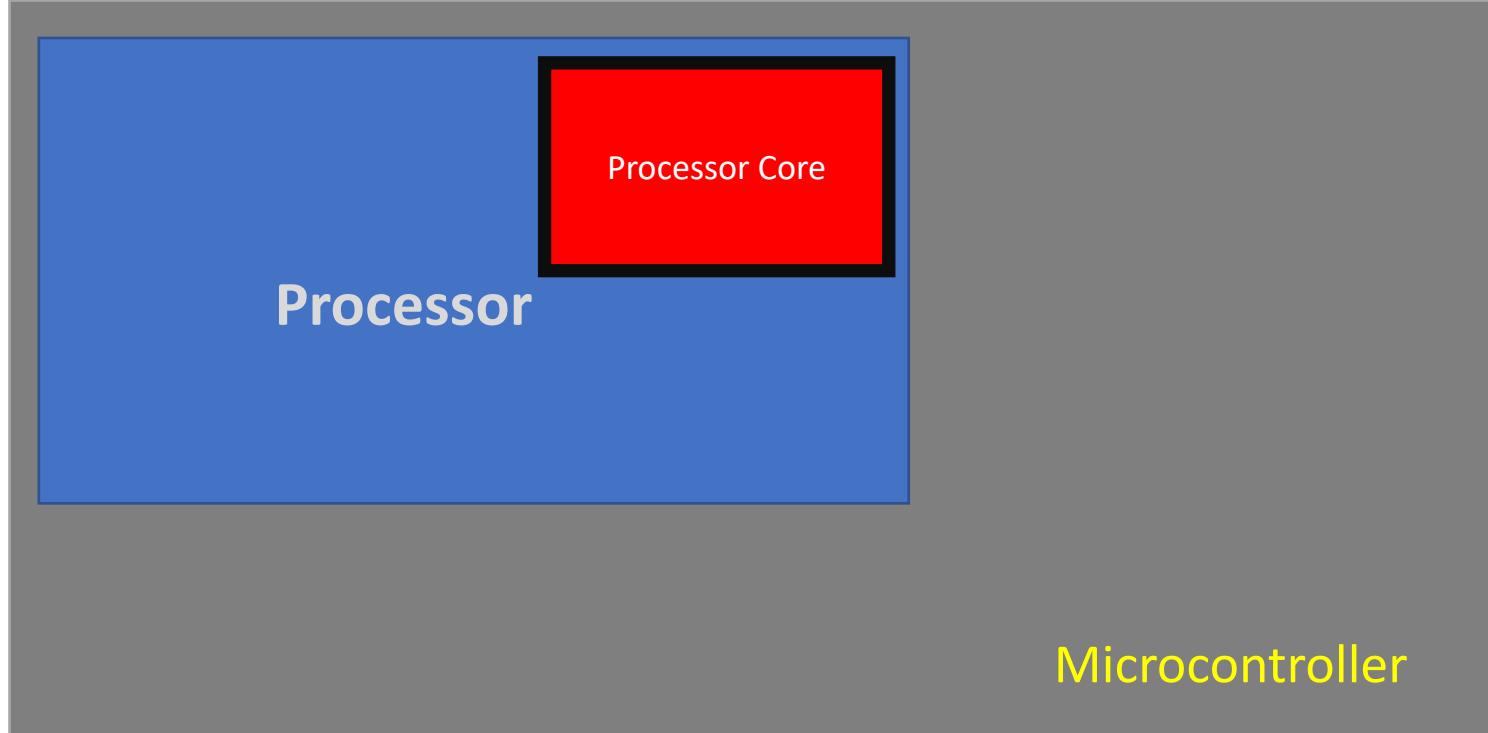


What is context switching ?

- Context switching is the procedure of **switching out** of the currently running task from the CPU after saving the task's execution context or state and **switching in** the next task's to run on the CPU by retrieving the past execution context or state of the task.

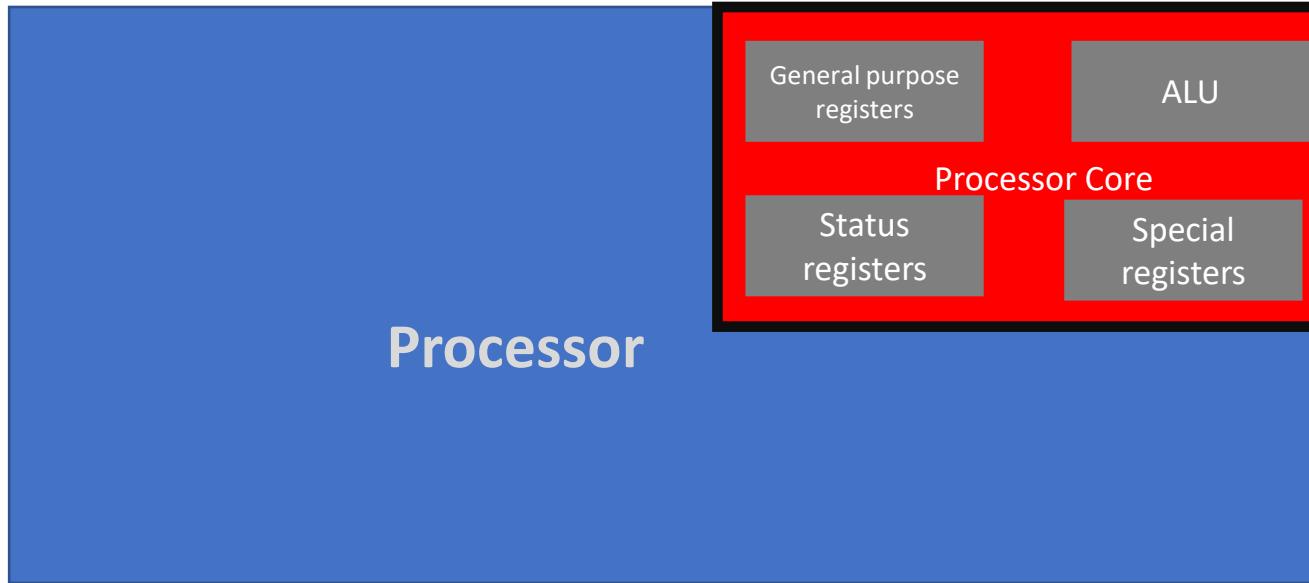


What is execution context or state of a task ?



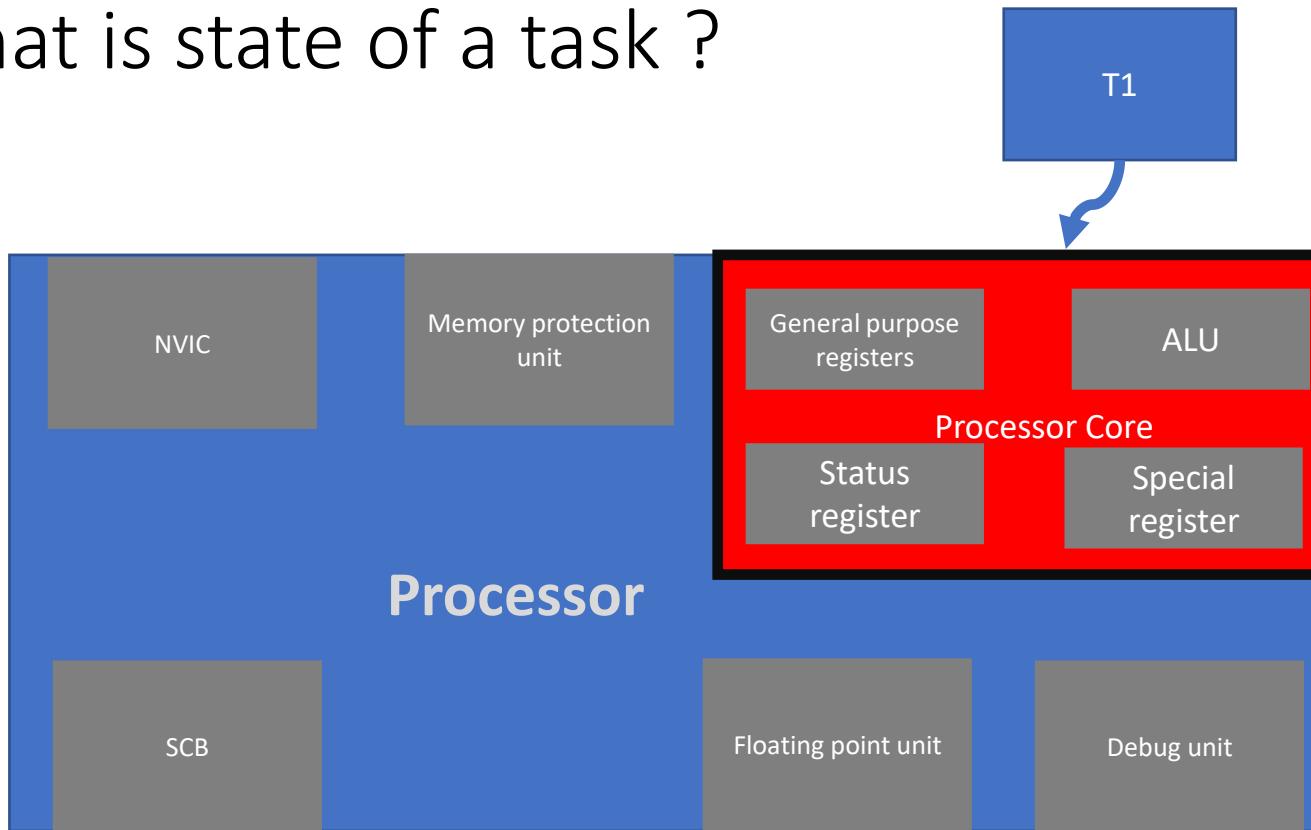


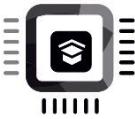
What is state of a task ?



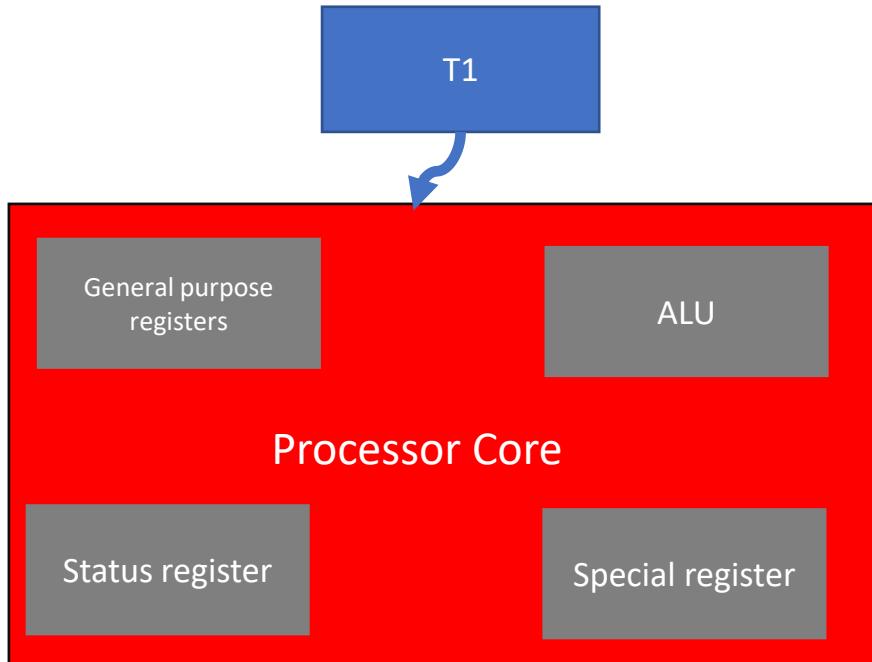


What is state of a task ?



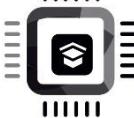


State of a task

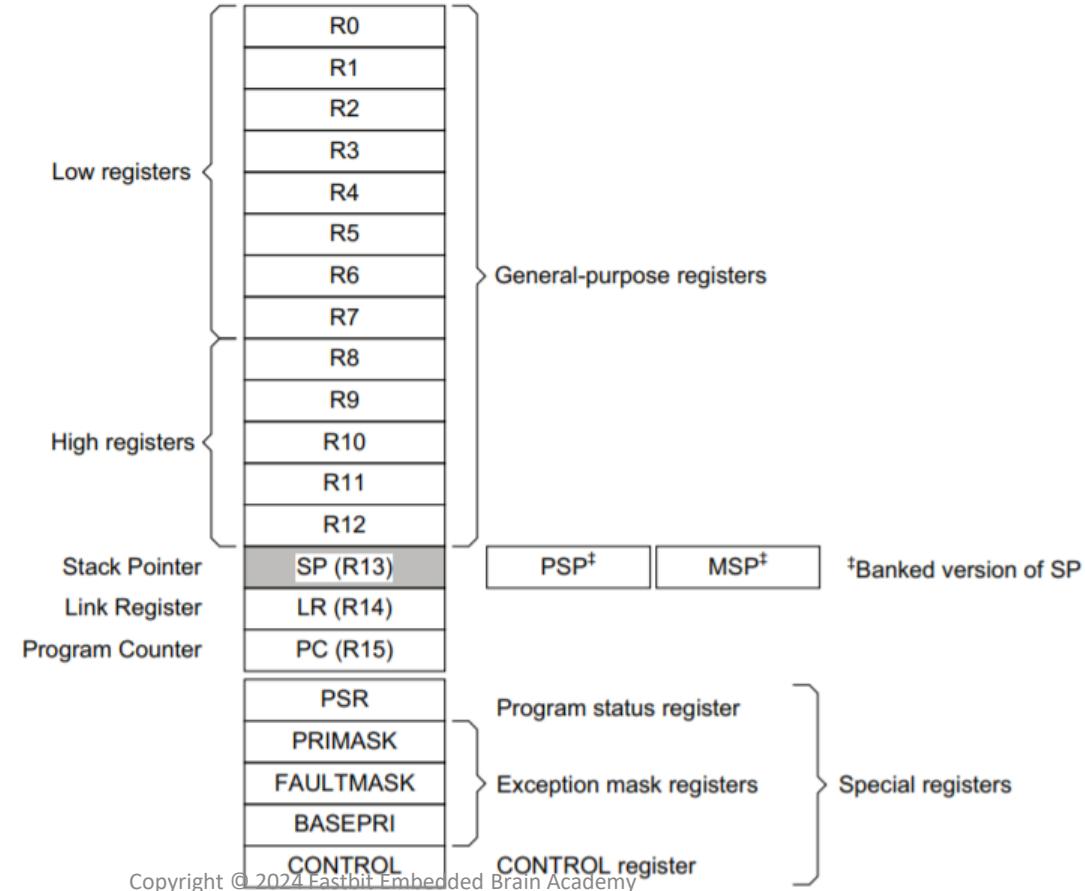


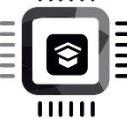
General purpose
registers
+
Some special registers
+
Status register

2.1.3 Core registers

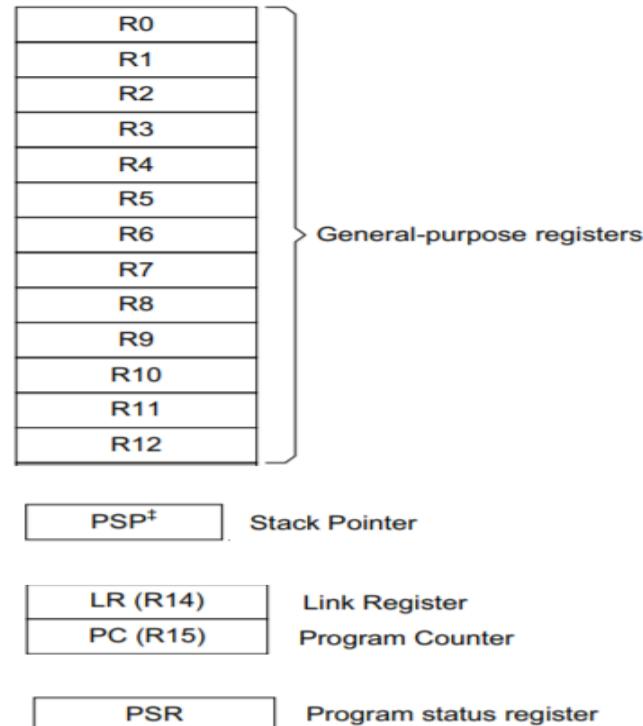


The processor core registers are:



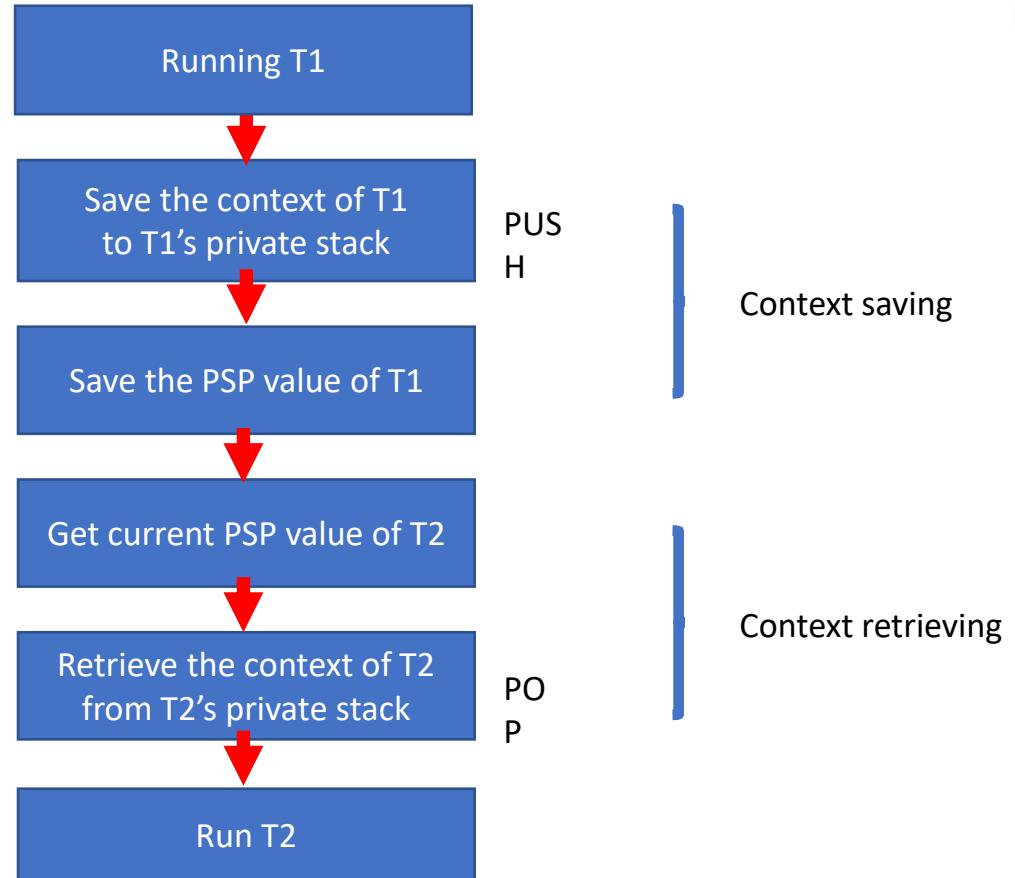


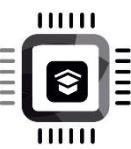
Summary: State of a task





Case of T1 switching out, T2 switching in

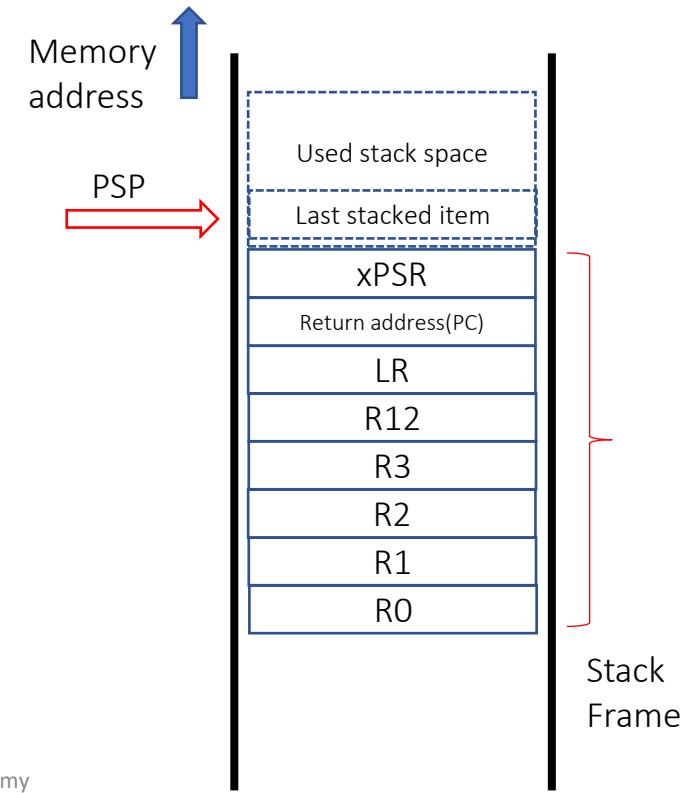
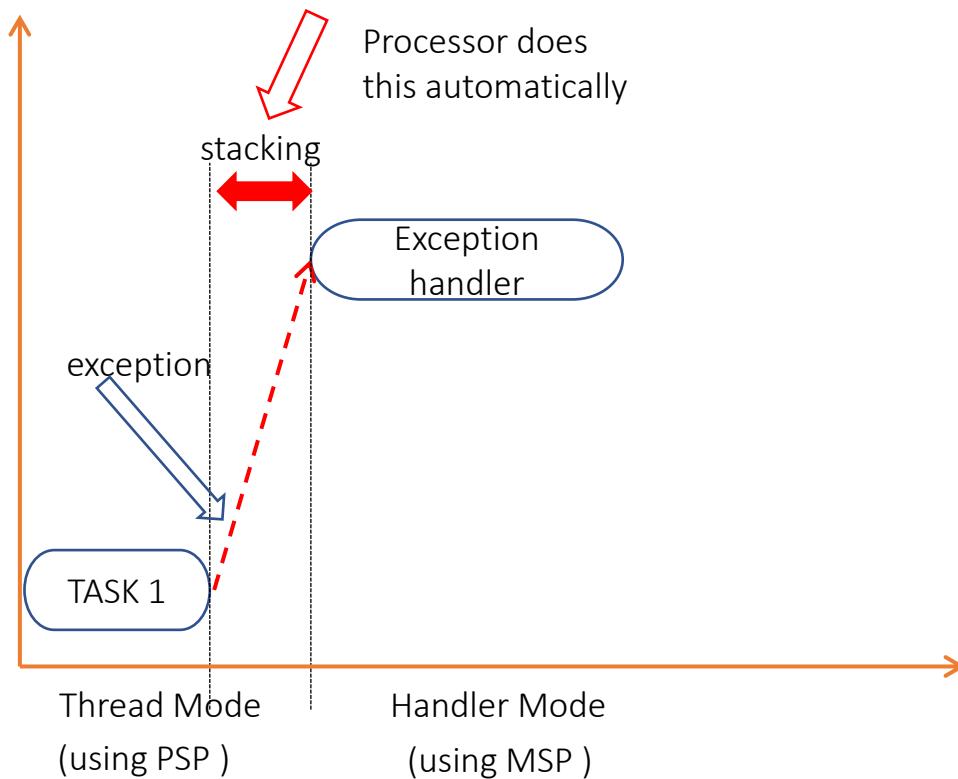




Stacking and Un-stacking during Exception

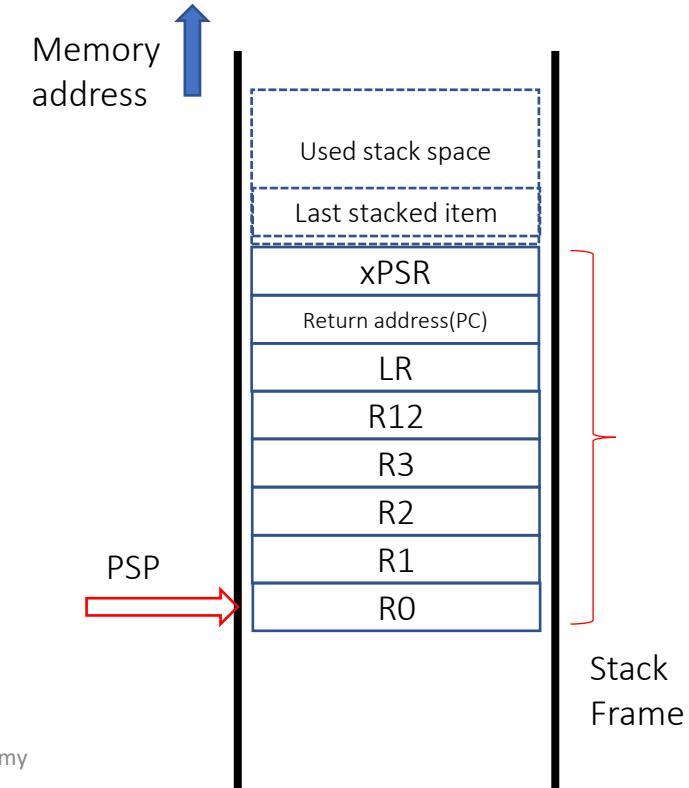
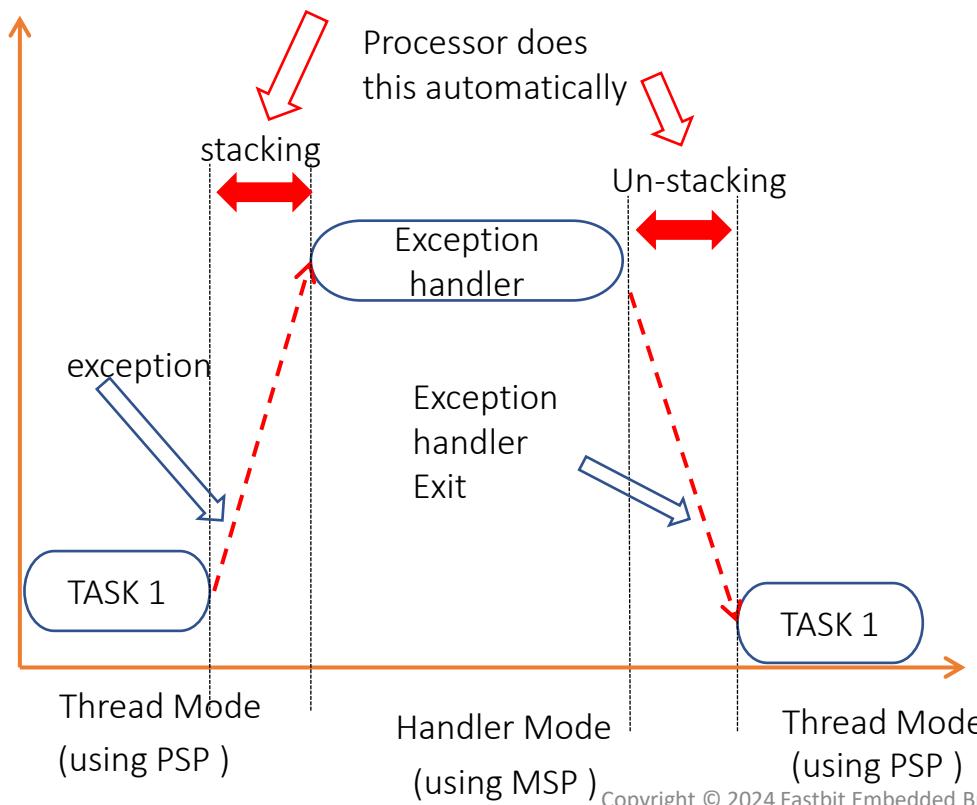


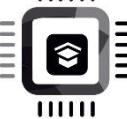
Stacking



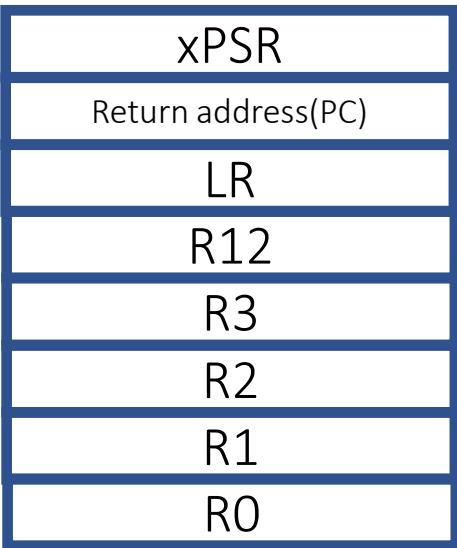


Un-stacking

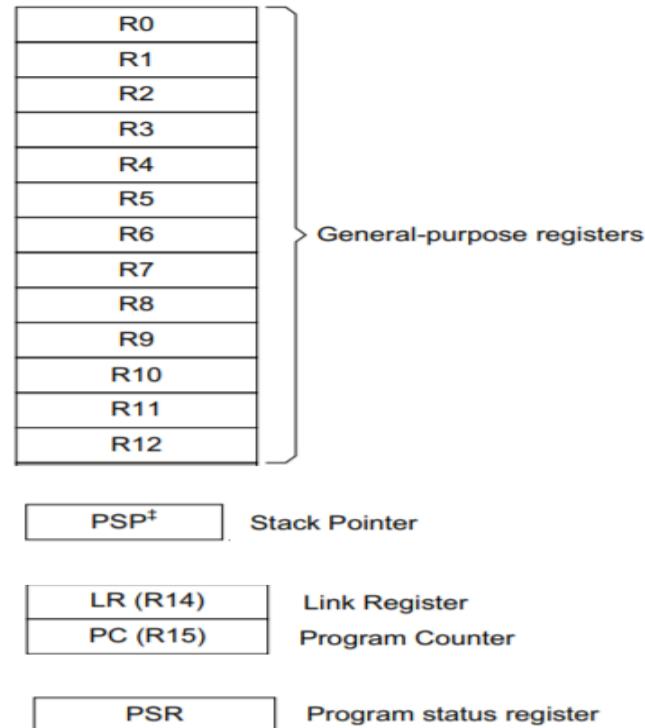


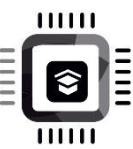


Stack frame



Summary: State of a task





Let's code

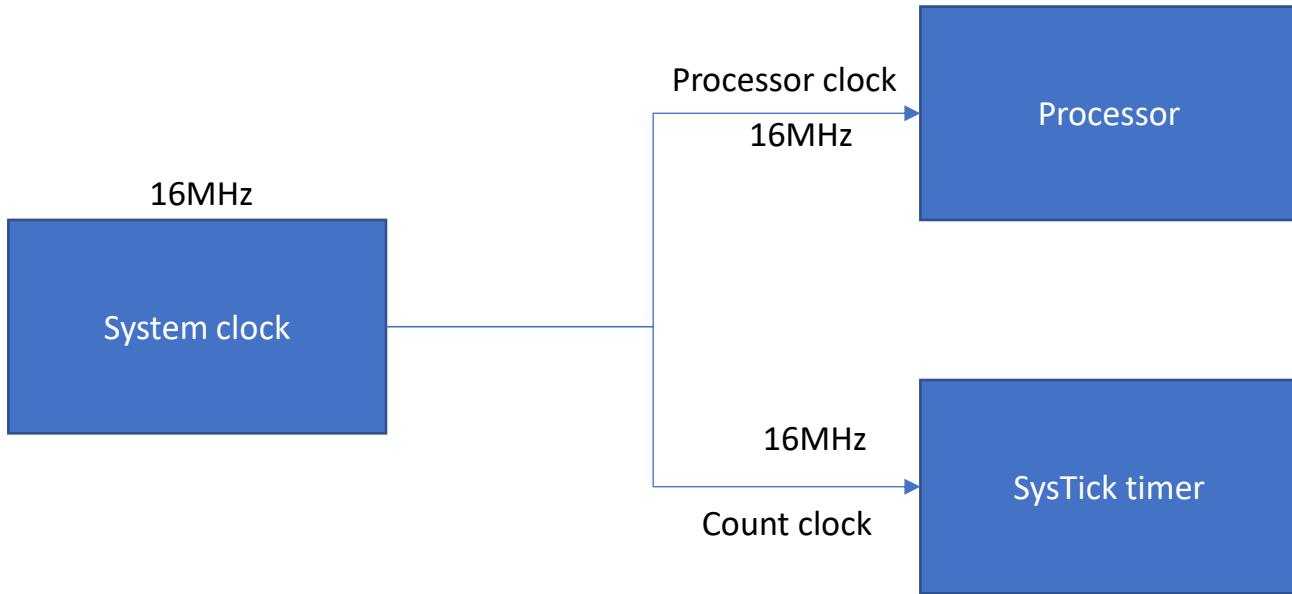
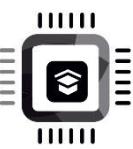
- Configure the systick timer to produce exception for every 1ms

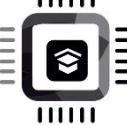


SysTick count value calculation

- Processor Clock = 16MHz
- SysTick timer count clock = 16MHz
- 1ms is 1KHz in frequency domain
- So, to bring down SysTick timer count clock from 16MHz to 1KHz use a divisor (reload value)
- Reload value = 16000

$$\frac{16000000 \text{ Hz}}{16000 \text{ count value}} = 1000 \text{ Hz} \quad (\text{TICK_HZ Desired exception frequency})$$





SysTick timer count clock = 16MHz

For 1 count it takes 0.0625 micro seconds

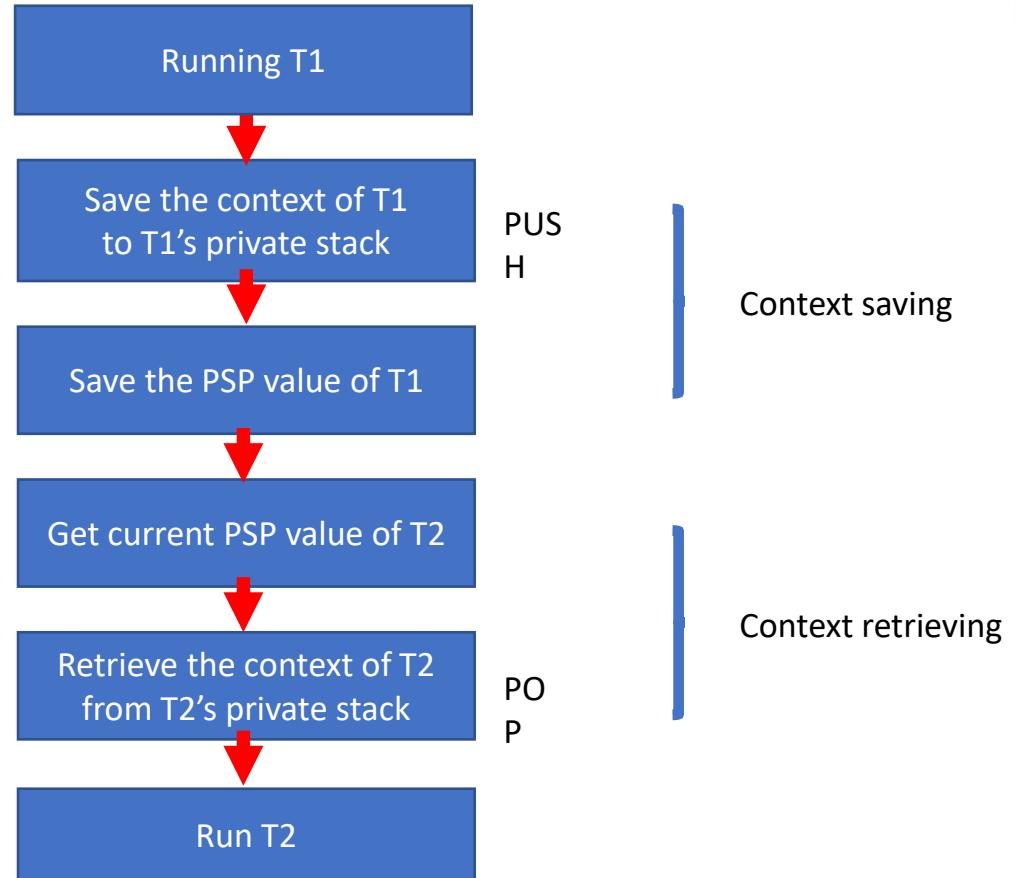
0.0625 μ s delay  1 count

1 μ s delay  16 count

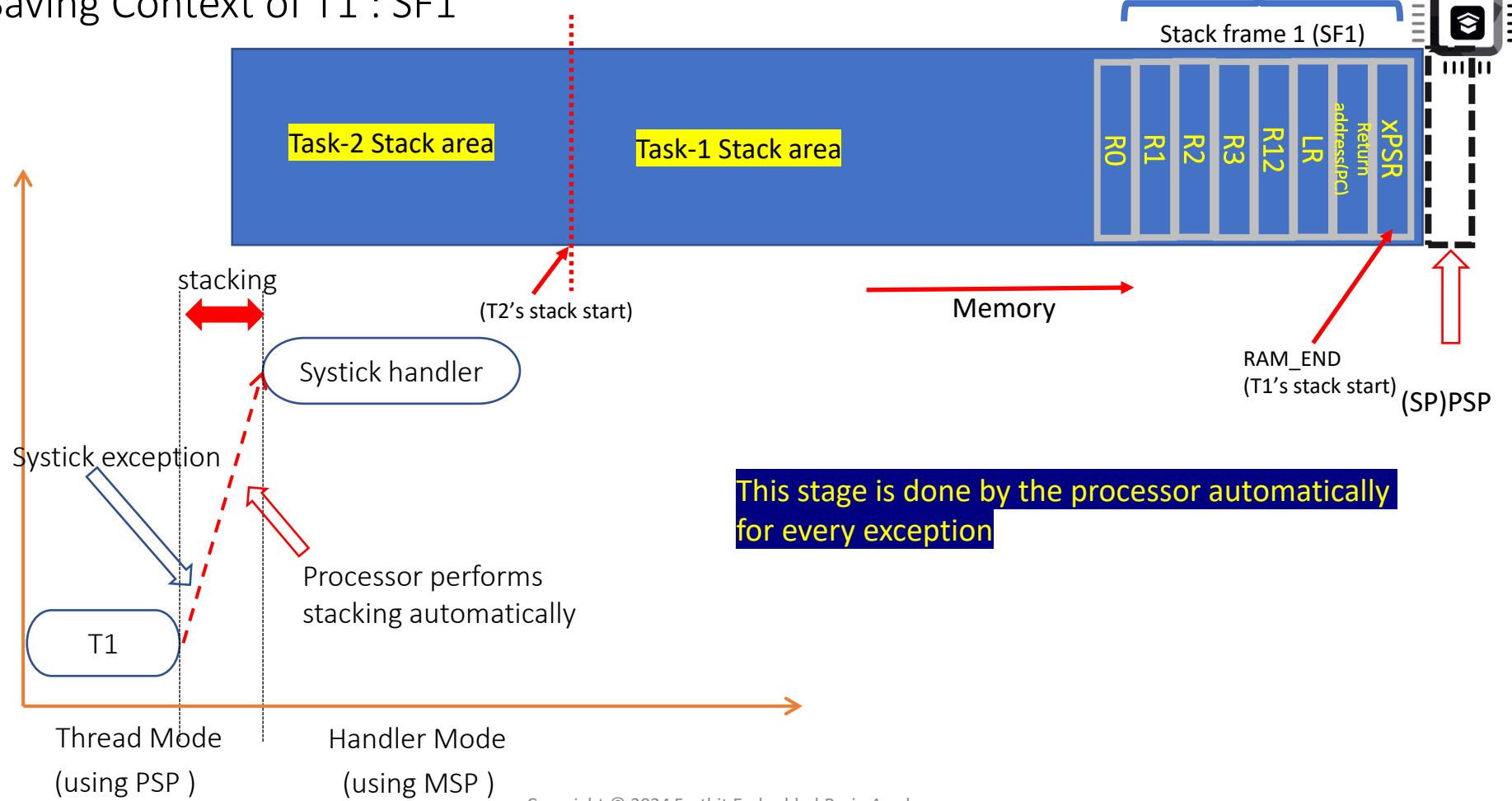
1 ms delay  16000 count

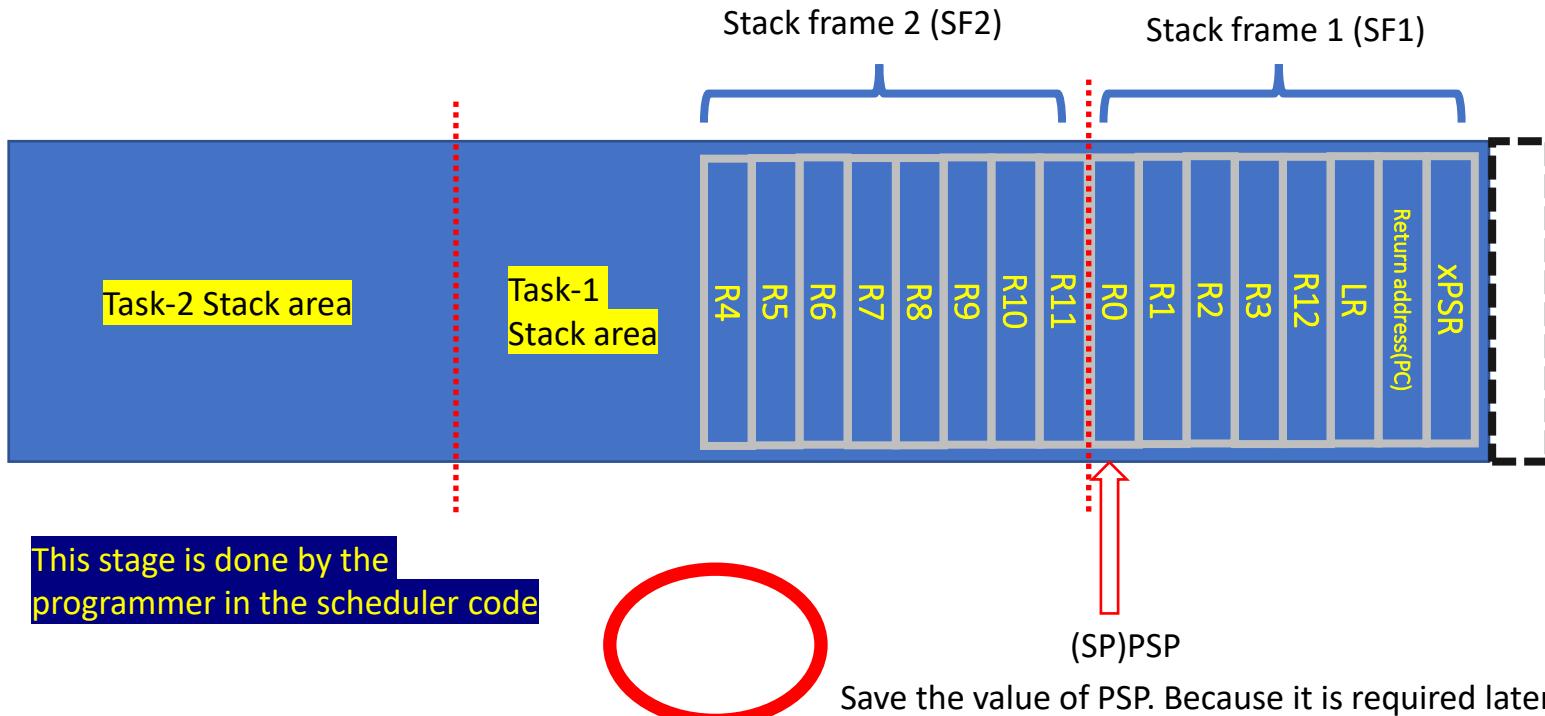
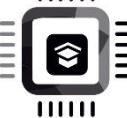


Case of T1 switching out, T2 switching in

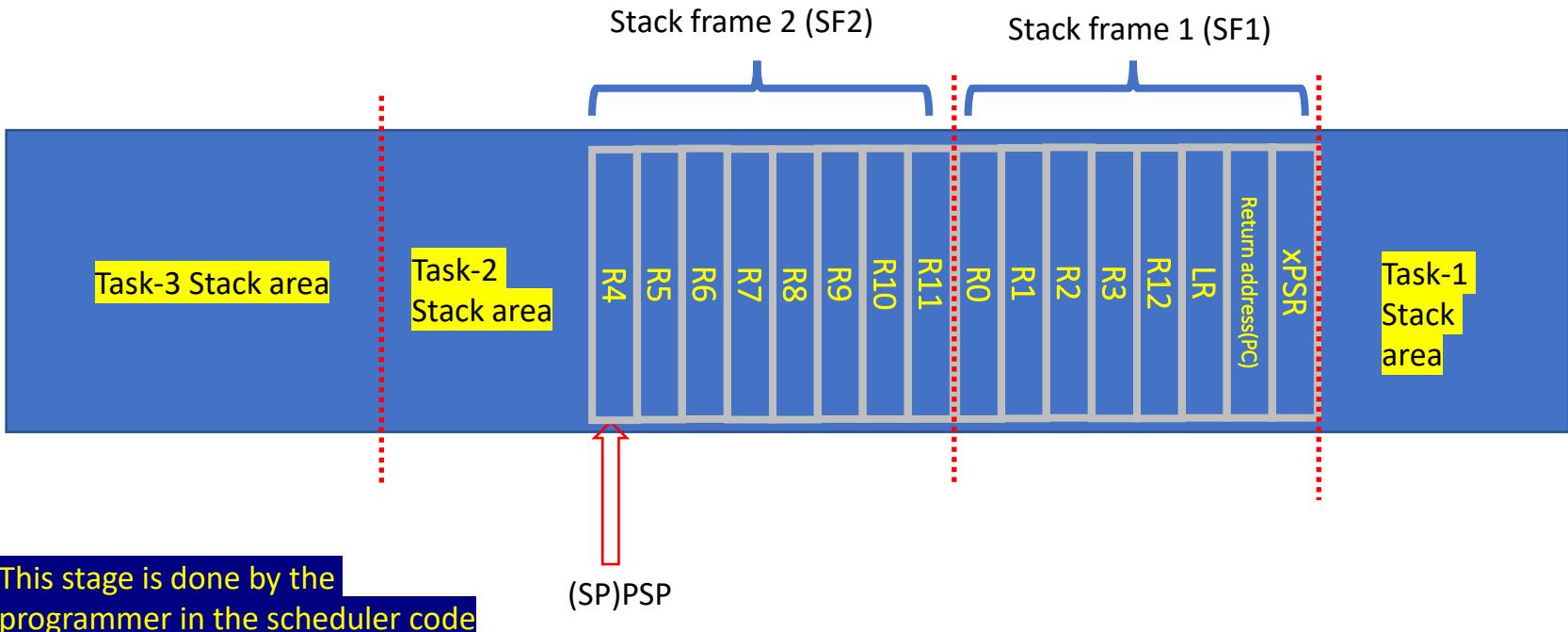
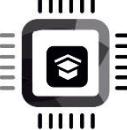


Saving Context of T1 : SF1

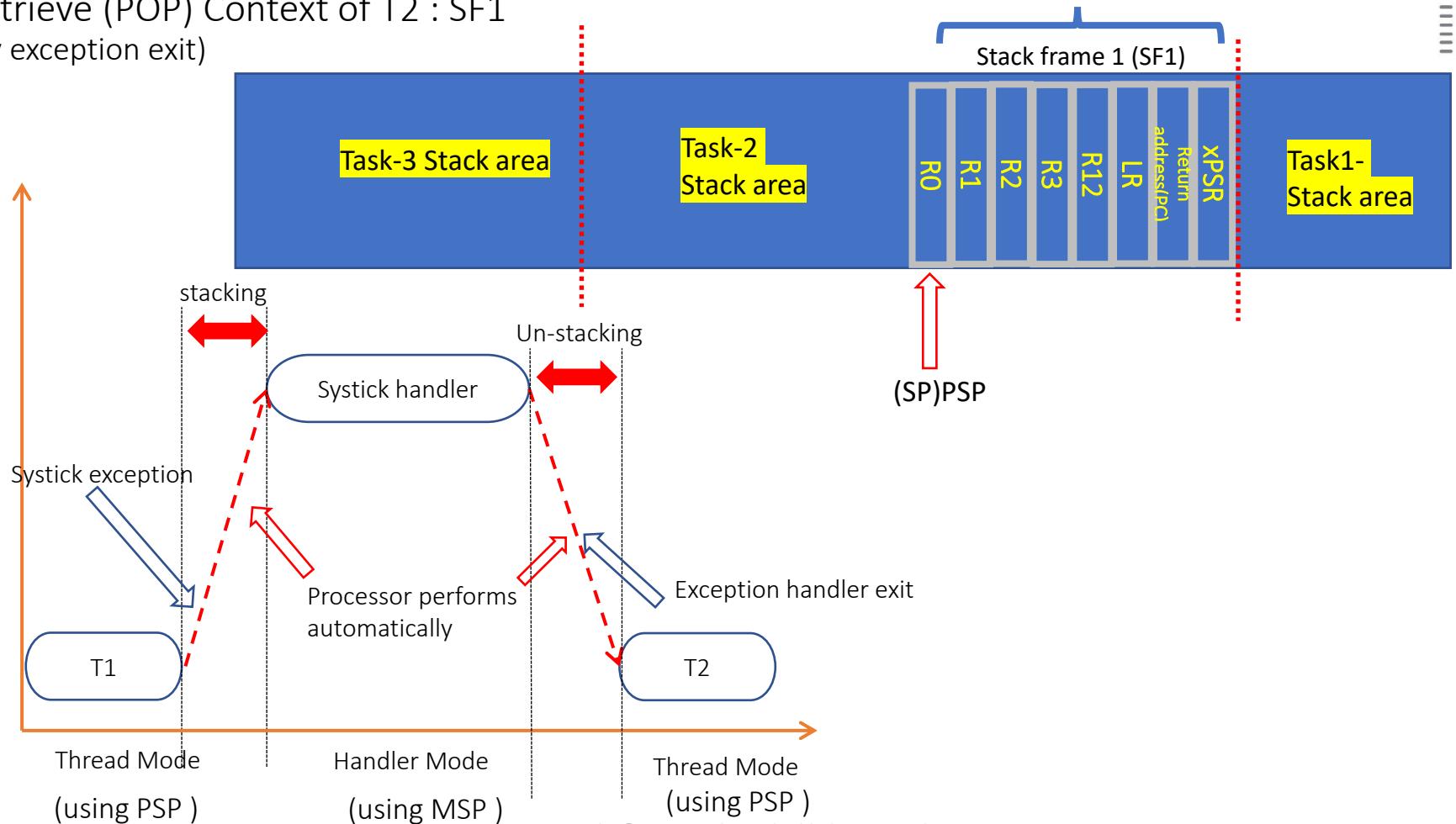


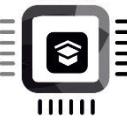


Save the value of PSP. Because it is required later when processor needs to resume the execution of T1 by retrieving its saved state .



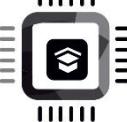
Retrieve (POP) Context of T2 : SF1 (By exception exit)



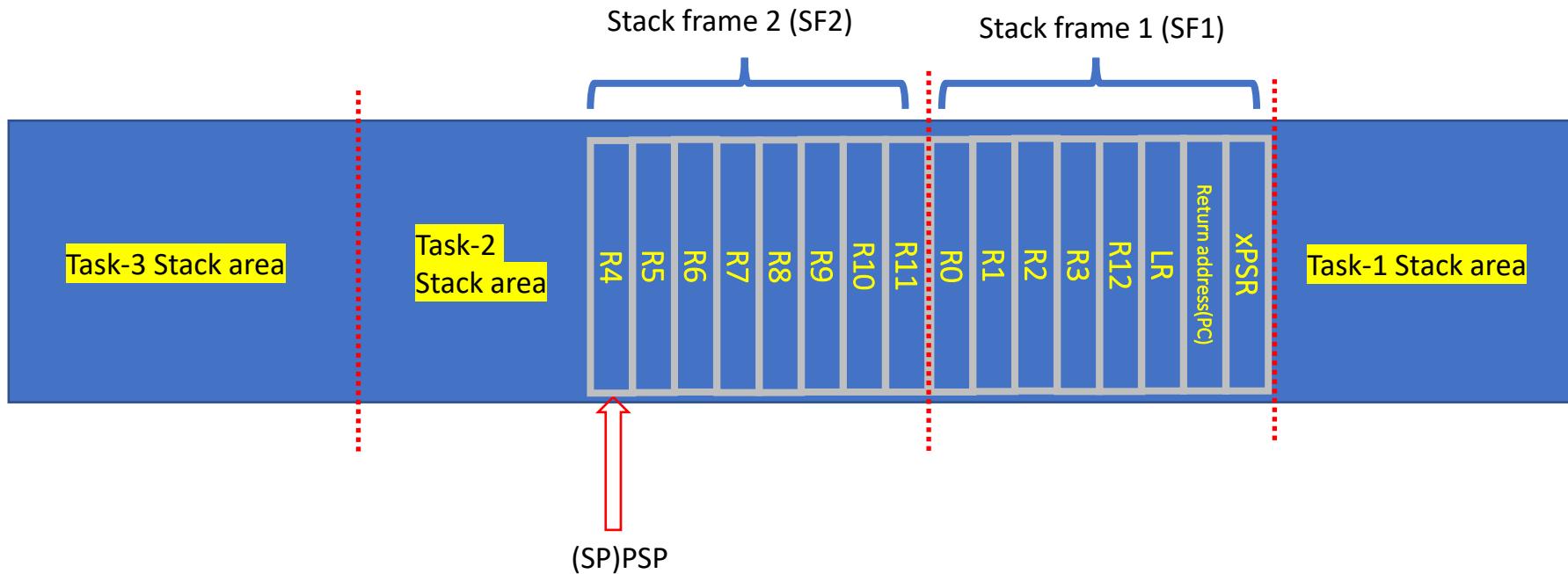


Task's stack area init and storing of dummy SF

- Each task can consume a maximum of 1KB of memory as a private stack
- This stack is used to hold tasks local variables and context(SF1+SF2)
- When a Task is getting scheduled for the very first time, it doesn't have any context. So, the programmer should store dummy SF1 and SF2 in Task's stack area as a part of "task initialization" sequence before launching the scheduler.



Dummy initial context of a Task

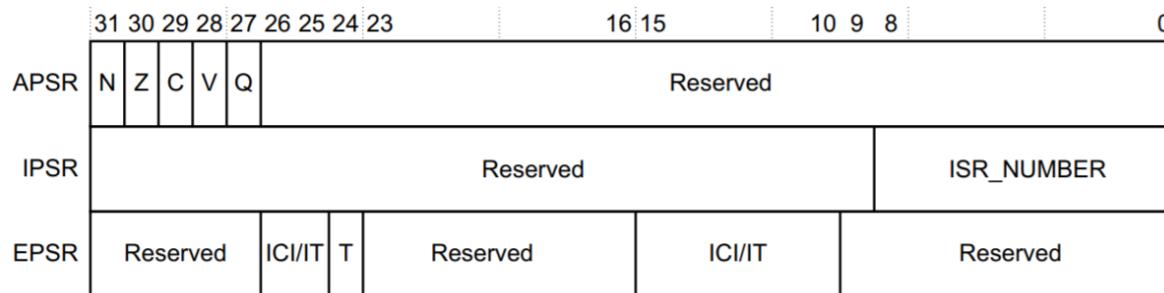




The *Program Status Register* (PSR) combines:

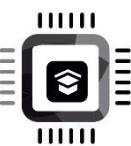
- *Application Program Status Register* (APSR)
- *Interrupt Program Status Register* (IPSR)
- *Execution Program Status Register* (EPSR).

These registers are mutually exclusive bitfields in the 32-bit PSR. The bit assignments are:



Access these registers individually or as a combination of any two or all three registers, using the register name as an argument to the MSR or MRS instructions. For example:

- read all of the registers using PSR with the MRS instruction
- write to the APSR N, Z, C, V, and Q bits using APSR_nzcvq with the MSR instruction.

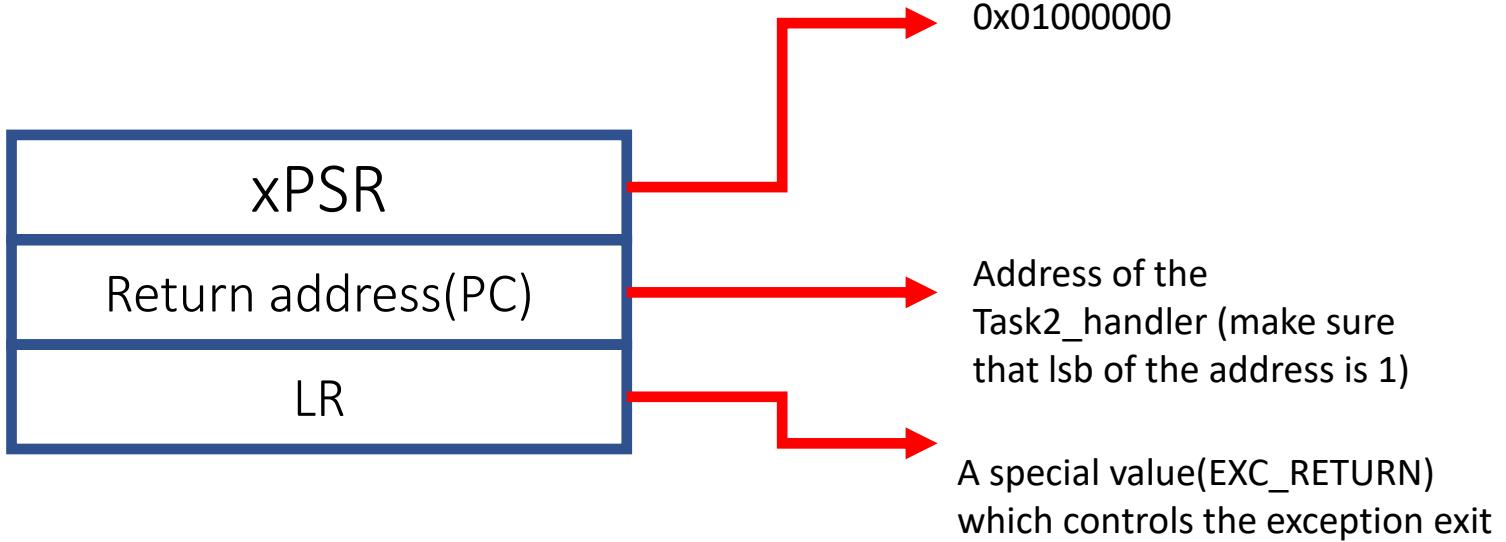
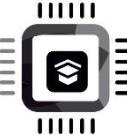


Thumb state

The Cortex-M4 processor only supports execution of instructions in Thumb state. The following can clear the T bit to 0:

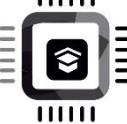
- instructions BLX, BX and POP{PC}
- restoration from the stacked xPSR value on an exception return
- bit[0] of the vector value on an exception entry or reset.

Attempting to execute instructions when the T bit is 0 results in a fault or lockup. See [Lockup on page 2-31](#) for more information.

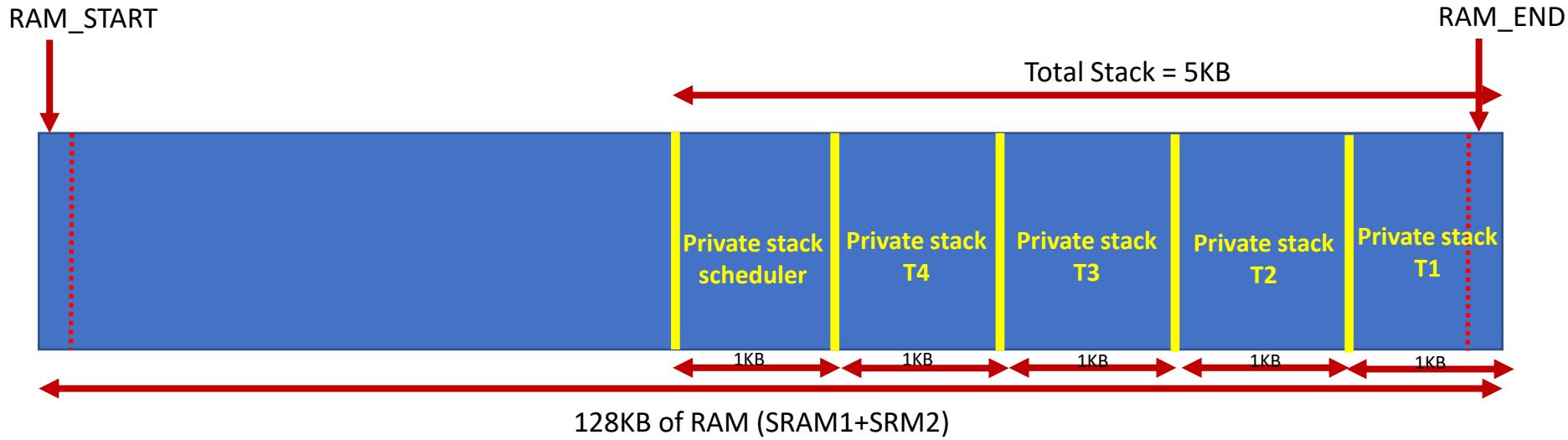


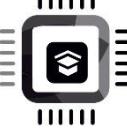
**Table 2-17 Exception return behavior**

EXC_RETURN[31:0]	Description
0xFFFFFFFF1	Return to Handler mode, exception return uses non-floating-point state from the MSP and execution uses MSP after return.
0xFFFFFFFF9	Return to Thread mode, exception return uses non-floating-point state from MSP and execution uses MSP after return.
0xFFFFFFFFD	Return to Thread mode, exception return uses non-floating-point state from the PSP and execution uses PSP after return.
0xFFFFFE1	Return to Handler mode, exception return uses floating-point-state from MSP and execution uses MSP after return.
0xFFFFFE9	Return to Thread mode, exception return uses floating-point state from MSP and execution uses MSP after return.
0xFFFFFED	Return to Thread mode, exception return uses floating-point state from PSP and execution uses PSP after return.



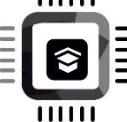
Initialize scheduler stack pointer (MSP)



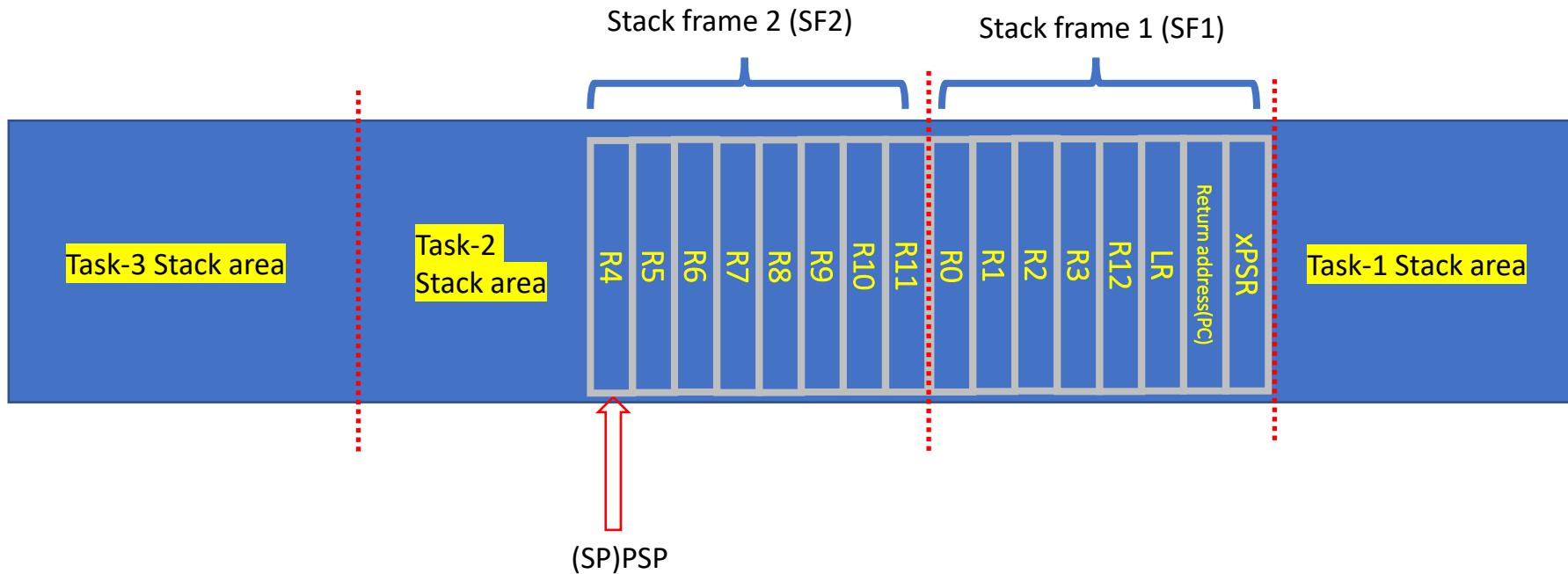


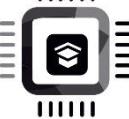
Init tasks stack memory

- Store dummy SF1 and SF2 in stack memory of each task

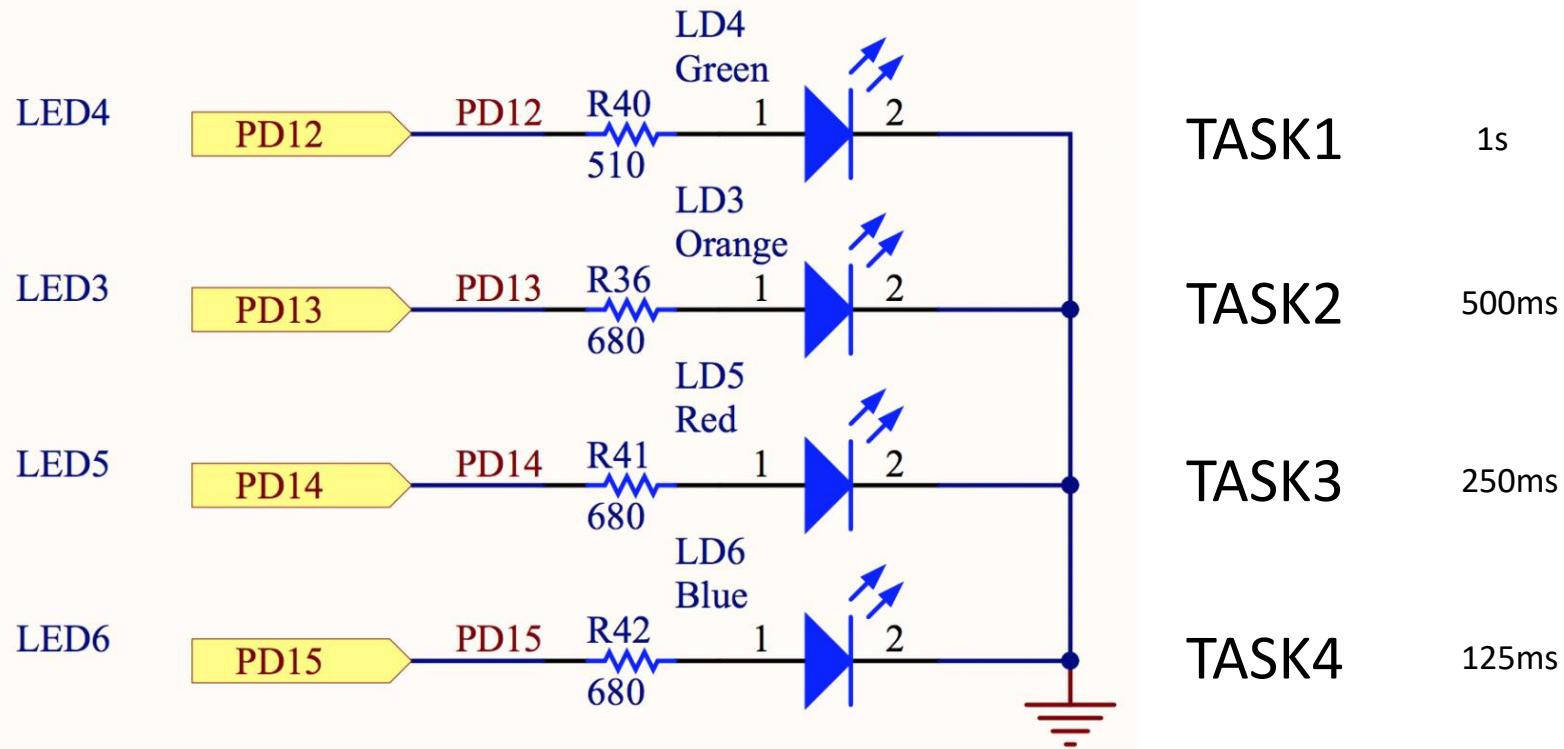


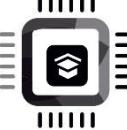
Dummy initial context of a Task





LED toggling using multiple tasks

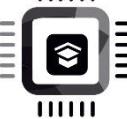




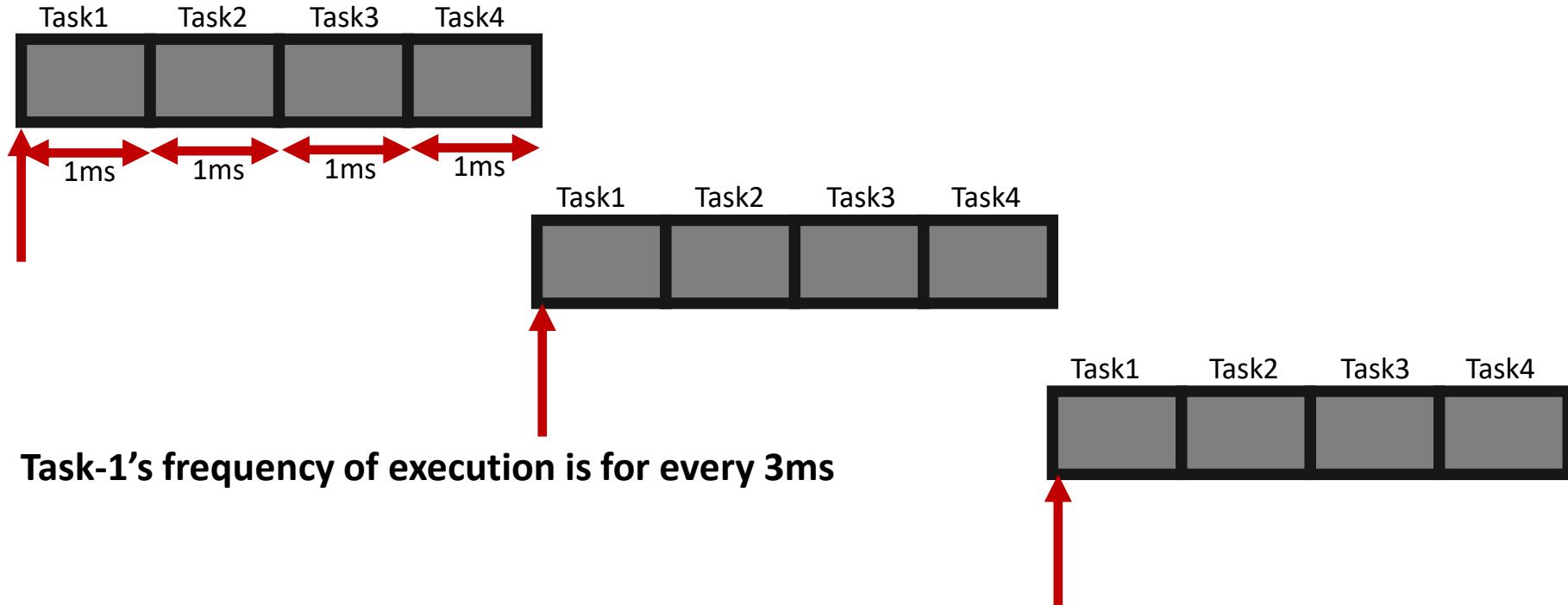
Implementation

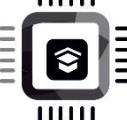
Add the giving led.c and led.h files to the project

Use software delay (for loop) to toggle the LEDs



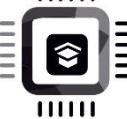
LED toggle behavior





Introducing blocking state for tasks

- When a task has got nothing to do, it should simply call a delay function which should put the task into the blocked state from running state until the specified delay is elapsed
- We should now maintain 2 states for a task. Running and Blocked
- The scheduler should schedule only those tasks which are in Running state
- The scheduler also should unblock the blocked tasks if their blocking period is over and put them back to running state.



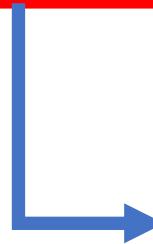
TCB(Task Control Block)

```
void init_systick_timer(uint32_t tick_hz);
__attribute__((naked)) void init_scheduler_stack(uint32_t sched_top_of_stack);
void init_tasks_stack(void);
void enable_processor_faults(void);
__attribute__((naked)) void switch_sp_to_psp(void);
uint32_t get_psp_value(void);

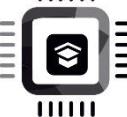
uint32_t psp_of_tasks[MAX_TASKS] = {T1_STACK_START, T2_STACK_START, T3_STACK_START, T4_STACK_START};

uint32_t task_handlers[MAX_TASKS];

uint8_t current_task = 0; //task1 is running
```

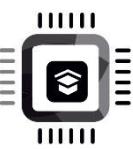


```
typedef struct{
    uint32_t psp_value;
    uint32_t block_count;
    uint8_t run_state;
    void (*task_handler) (void);
} TCB_t;
```



Block a task for a given number of ticks.

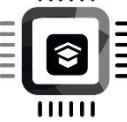
- Let's introduce a function called “**task_delay**” which puts the calling task to the blocked state for a given number of ticks
- E.g., **task_delay(1000)**; if a task calls this function then `task_delay` function puts the task into blocked state and allows the next task to run on the CPU
- Here, the number 1000 denotes a block period in terms of ticks, the task who calls this function is going to block for 1000 ticks (systick exceptions), i.e., for 1000ms since each tick happens for every 1ms.
- The scheduler should check elapsed block period of each blocked task and put them back to running state if the block period is over



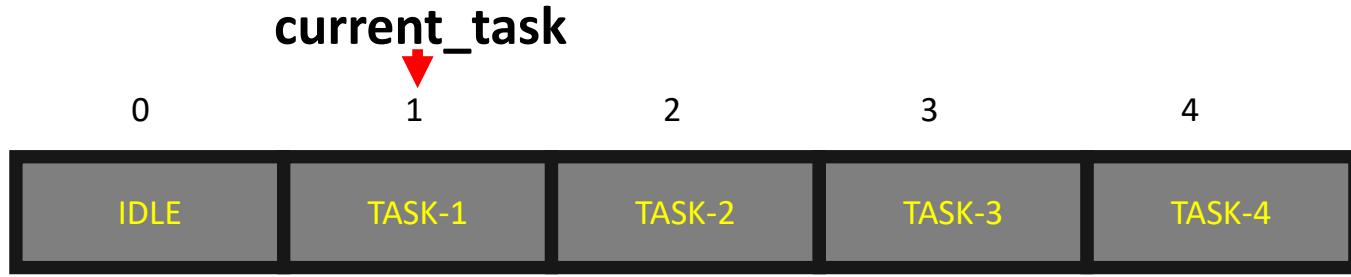
Idle task

What if all the tasks are blocked? Who is going to run on the CPU?

We will use the idle task to run on the CPU if all the tasks are blocked. The idle task is like user tasks but only runs when all user tasks are blocked, and you can put the CPU to sleep.

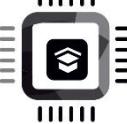


Deciding next task to run

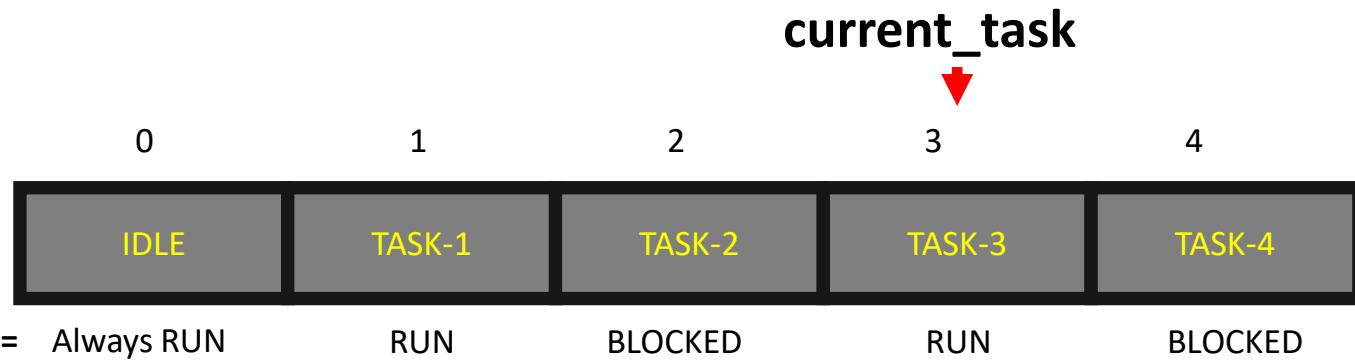


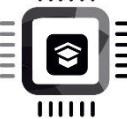
Current_task_State = Always RUN RUN RUN RUN RUN

Deciding which task to run next depends on “state” of the next task.

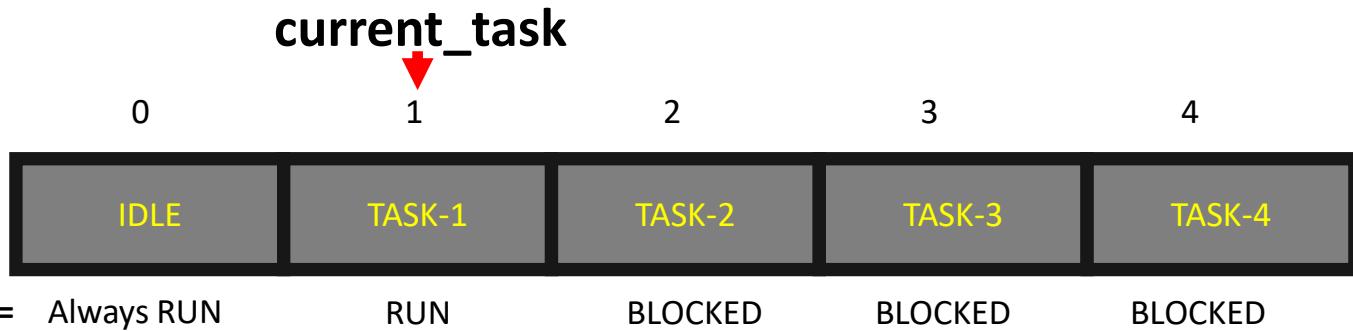


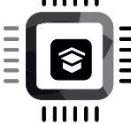
Deciding next task to run





Deciding next task to run





Deciding next task to run

current_task



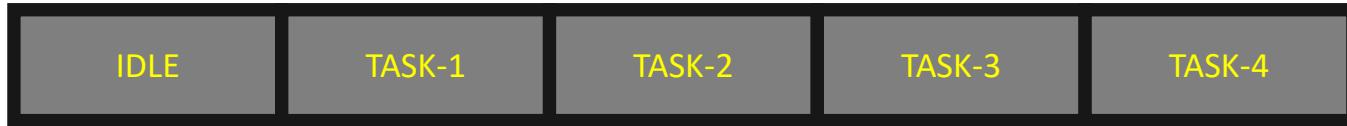
0

1

2

3

4



Current_task_State = Always RUN BLOCKED BLOCKED BLOCKED BLOCKED



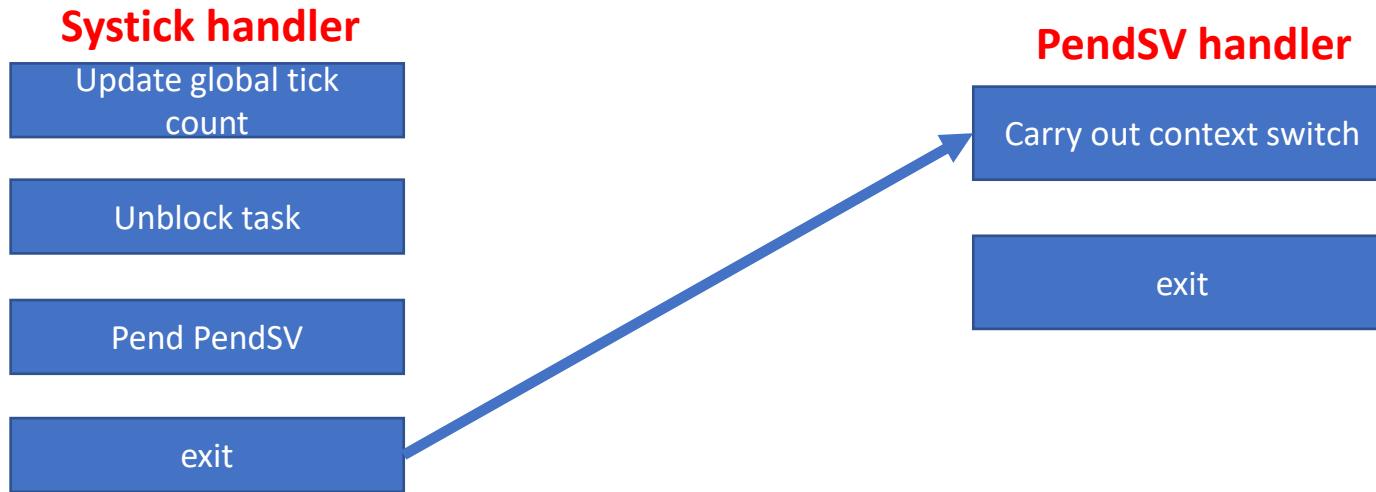
Global tick count

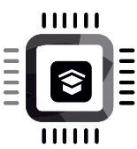
- How does the scheduler decide when to put the blocked state tasks (blocked using task_delay function) back to the running state?
- It has to compare the task's delay tick count with a global tick count
- So, scheduler should maintain a global tick count and update it for every systick exception



PendSV handler

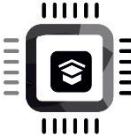
- We will use pendSV handler to carry out the context switch operation instead of systick handler



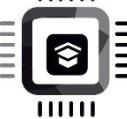


Exercise

- Analyse stack activities by doing instruction level debugging

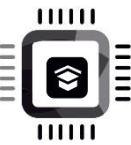


Building and running bare metal executables for ARM target using GNU tools

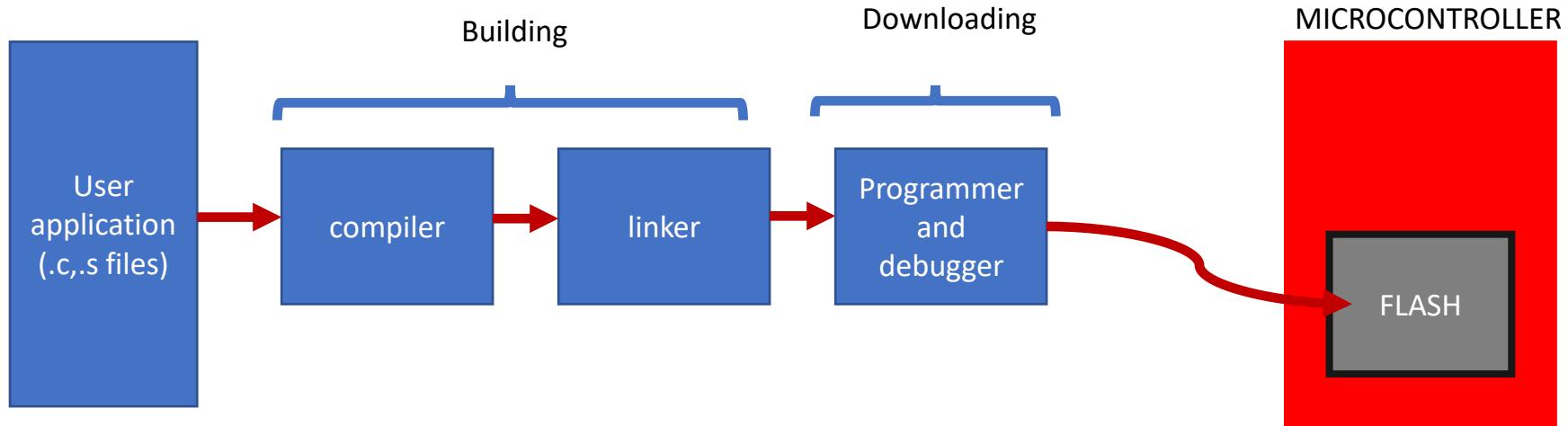


Section deliverables

- Toolchain installation
- Understand compiling a ‘C’ program for an embedded target without using an IDE
- Writing microcontroller startup file for STM32F4 MCU
- Writing your own ‘C’ startup code(code which runs before main())
- Understanding different sections of the relocatable object file(.o files)
- Writing linker script file from scratch and understanding section placements
- Linking multiple .o files using linker script and generating application executable (.elf, bin, hex)
- Loading the final executable on the target using OpenOCD and GDB client



Running





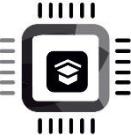
Sample ‘C’ program

Let's take one of the sample programs we wrote in the previous lectures.

Project : task_scheduler

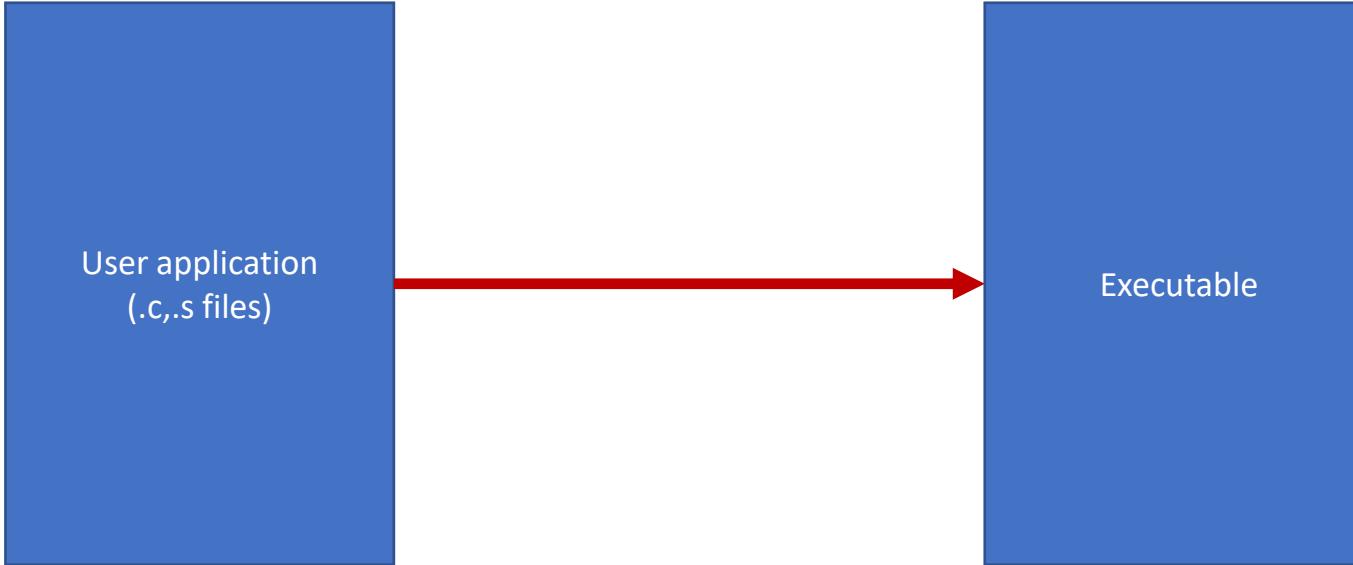
The screenshot shows a file explorer window with the following path: OneDrive > ofc > courses > my_workspace. The table below lists the contents of this workspace.

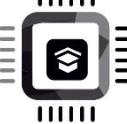
	Name	Status	Type	Size
	led	↻	C Source File	2 KB
;	led	↻	C Header Source F...	1 KB
;	main	↻	C Source File	11 KB
;	main	↻	C Header Source F...	2 KB



(High level language)

(Embedded target specific machine codes)





Cross compilation and Toolchains

What is cross compilation ?

Cross-compilation is a process in which the cross-toolchain runs on the host machine(PC) and creates executables that run on different machine(ARM)



Cross compilation and Toolchains

Cross-compilation toolchains

- ✓ Toolchain or a cross-compilation toolchain is a collection of binaries which allows you to compile, assemble, link your applications.
- ✓ It also contains binaries to debug the application on the target
- ✓ Toolchain also comes with other binaries which help you to analyze the executables
 - dissect different sections of the executable
 - disassemble
 - extract symbol and size information
 - convert executable to other formats such as bin, ihex
 - Provides 'C' standard libraries



Cross compilation and Toolchains

Popular Tool-chains:

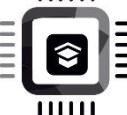
1. GNU Tools(GCC) for ARM Embedded Processors (free and open-source)
2. armcc from ARM Ltd. (ships with KEIL , code restriction version, requires licensing)

**Throughout this course, we have been using,
GNU's Compiler Collections(GCC) Toolchain**



Downloading GCC Toolchain for ARM embedded processors

- If you have installed STM32CubeIDE, then this toolchain would have already installed as a part of installation
- Official website to download GCC toolchain for ARM
- <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm>



Cross toolchain important binaries

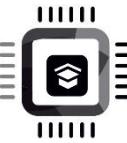
Compiler, linker, assembler
arm-none-eabi-gcc

linker
arm-none-eabi-ld

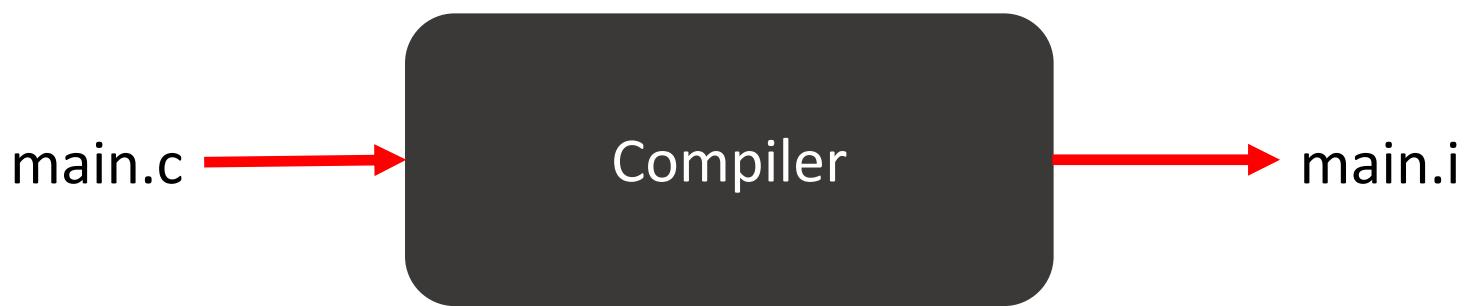
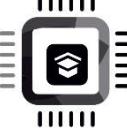
Assembler
arm-none-eabi-as

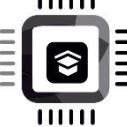
Elf file analyzer
arm-none-eabi-objdump
arm-none-eabi-readelf
arm-none-eabi-nm

Format converter
arm-none-eabi-objcopy

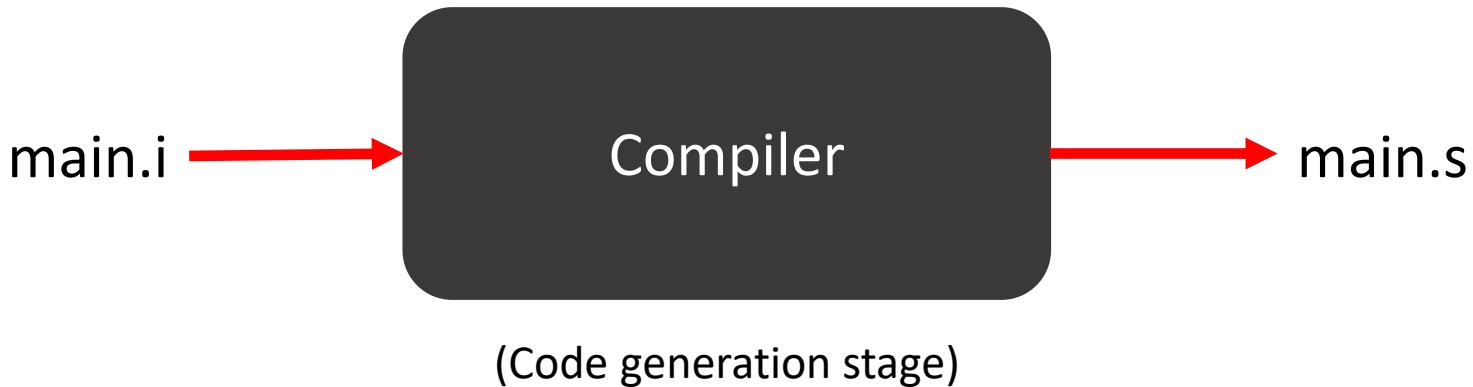


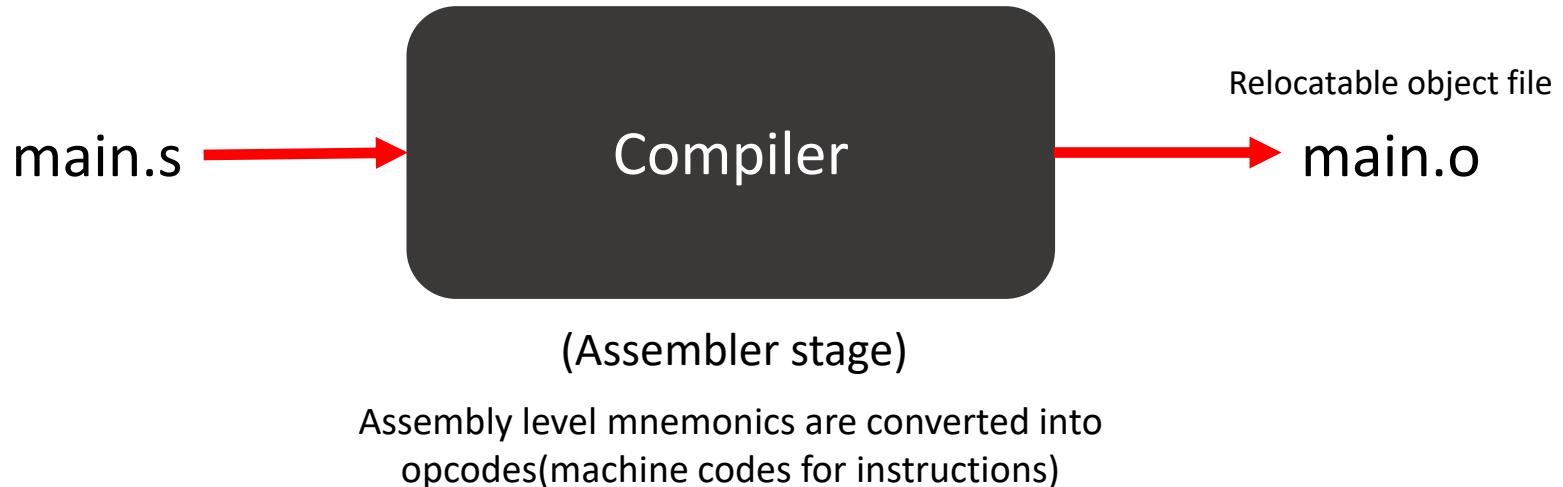
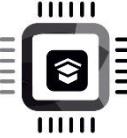
Build process

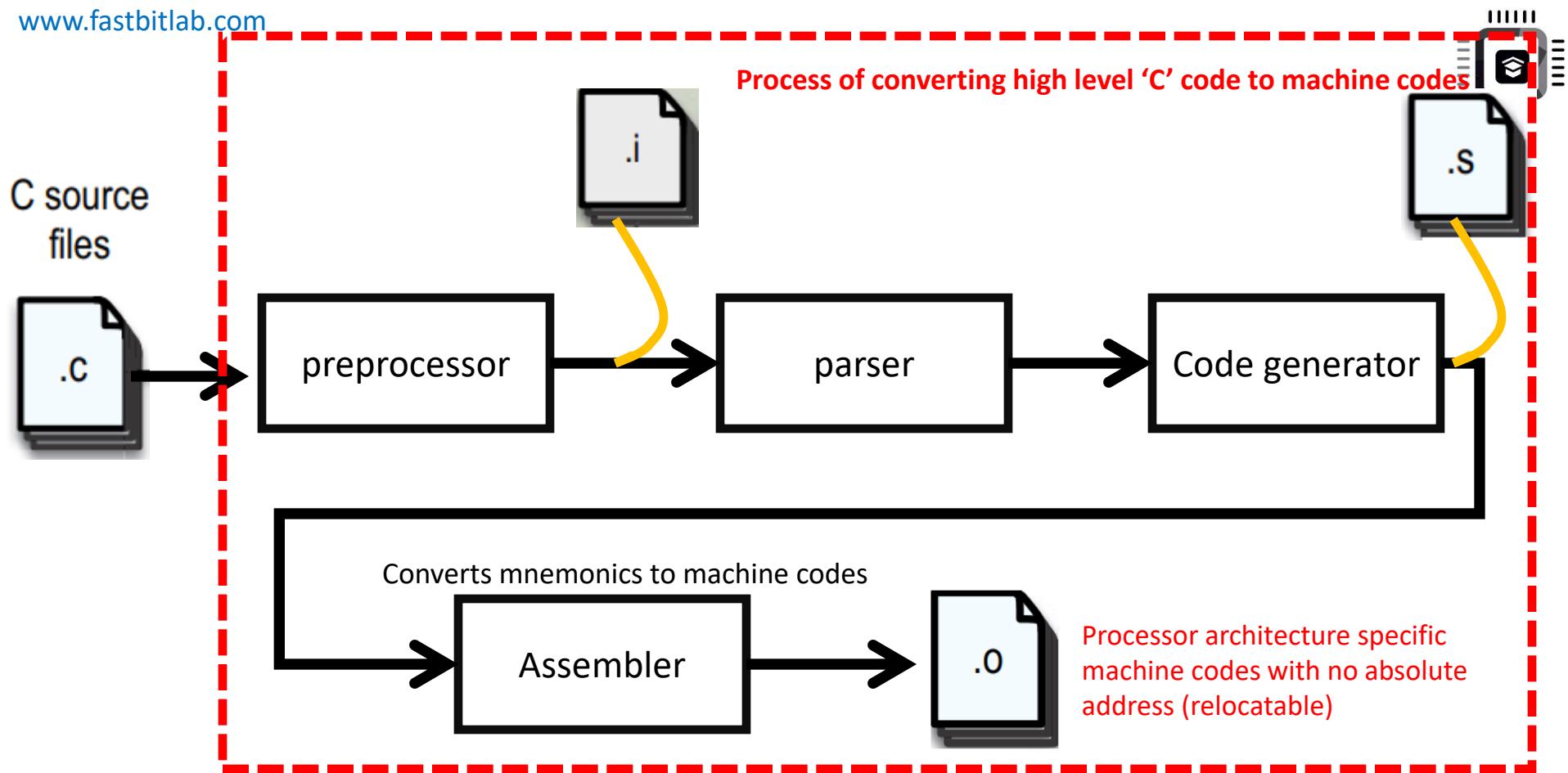


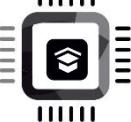


translating a source file into assembly language







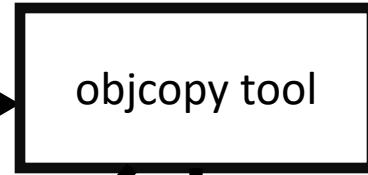


Linking stage of the build

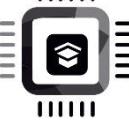
Re-locatable
object files



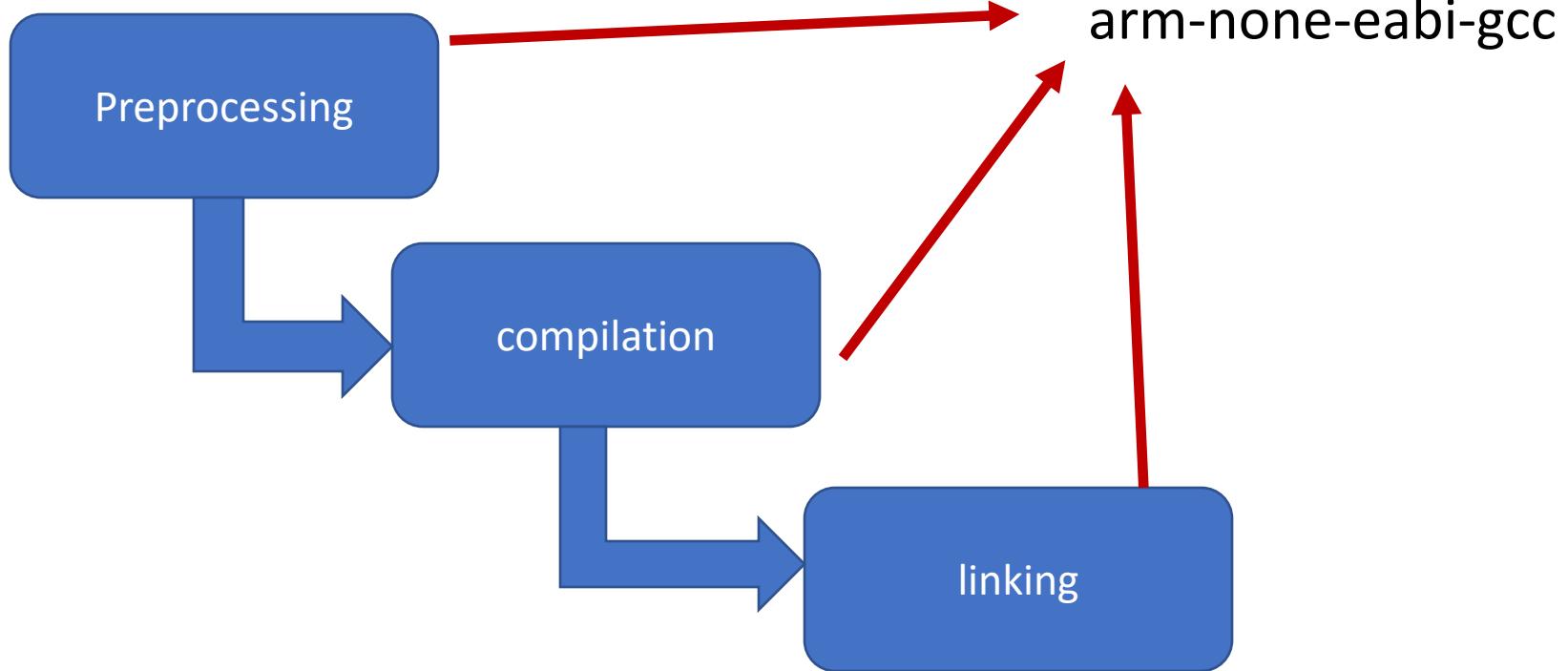
Executable and debug file



Other libraries
(std and/or third party . Ex. libc)



Summary of build process





Compiler options

-c Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix ‘.c’, ‘.i’, ‘.s’, etc., with ‘.o’.

Unrecognized input files, not requiring compilation or assembly, are ignored.

Ref : <https://gcc.gnu.org/onlinedocs/gcc/index.html>



- S Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified. By default, the assembler file name for a source file is made by replacing the suffix ‘.c’, ‘.i’, etc., with ‘.s’. Input files that don’t require compilation are ignored.
- E Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output. Input files that don’t require preprocessing are ignored.
- o *file* Place output in file *file*. This applies to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.
If ‘-o’ is not specified, the default is to put an executable file in ‘a.out’, the object file for ‘source.suffix’ in ‘source.o’, its assembler file in ‘source.s’, a precompiled header file in ‘source.suffix.gch’, and all preprocessed C source on standard output.

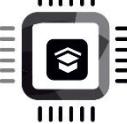
**-x language**

Specify explicitly the *language* for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next ‘-x’ option. Possible values for *language* are:

```
c  c-header  cpp-output
c++  c++-header  c++-cpp-output
objective-c  objective-c-header  objective-c-cpp-output
objective-c++  objective-c++-header  objective-c++-cpp-output
assembler  assembler-with-cpp
ada
f77  f77-cpp-input  f95  f95-cpp-input
go
brig
```

- x none** Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes (as they are if ‘-x’ has not been used at all).

If you only want some of the stages of compilation, you can use ‘-x’ (or filename suffixes) to tell `gcc` where to start, and one of the options ‘-c’, ‘-S’, or ‘-E’ to say where `gcc` is to stop. Note that some combinations (for example, ‘-x cpp-output -E’) instruct `gcc` to do nothing at all.



Compilation

- <compiler> <source file name .s or .c> -o <filename.o>

Important flags

-mcpu=cortex-m4

-mthumb

-c



-mthumb
-marm

Select between generating code that executes in ARM and Thumb states. The default for most configurations is to generate code that executes in ARM state, but the default can be changed by configuring GCC with the `--with-mode=state` configure option.

You can also override the ARM and Thumb mode for each function by using the `target("thumb")` and `target("arm")` function attributes (see [ARM Function Attributes](#)) or pragmas (see [Function Specific Option Pragmas](#)).

-mcpu=name[+extension...]

This specifies the name of the target ARM processor. GCC uses this name to derive the name of the target ARM architecture (as if specified by `-march`) and the ARM processor type for which to tune for performance (as if specified by `-mtune`). Where this option is used in conjunction with `-march` or `-mtune`, those options take precedence over the appropriate part of this option.

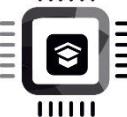
Ref : <https://gcc.gnu.org/onlinedocs/gcc/index.html>



Ref : <https://gcc.gnu.org/onlinedocs/gcc/index.html>

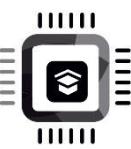
-mtune=*name*

This option specifies the name of the target ARM processor for which GCC should tune the performance of the code. For some ARM implementations better performance can be obtained by using this option. Permissible names are: ‘arm7tdmi’, ‘arm7tdmi-s’, ‘arm710t’, ‘arm720t’, ‘arm740t’, ‘strongarm’, ‘strongarm110’, ‘strongarm1100’, ‘strongarm1110’, ‘arm8’, ‘arm810’, ‘arm9’, ‘arm9e’, ‘arm920’, ‘arm920t’, ‘arm922t’, ‘arm946e-s’, ‘arm966e-s’, ‘arm968e-s’, ‘arm926ej-s’, ‘arm940t’, ‘arm9tdmi’, ‘arm10tdmi’, ‘arm1020t’, ‘arm1026ej-s’, ‘arm10e’, ‘arm1020e’, ‘arm1022e’, ‘arm1136j-s’, ‘arm1136jf-s’, ‘mpcore’, ‘mpcorenovfp’, ‘arm1156t2-s’, ‘arm1156t2f-s’, ‘arm1176jz-s’, ‘arm1176jzf-s’, ‘generic-armv7-a’, ‘cortex-a5’, ‘cortex-a7’, ‘cortex-a8’, ‘cortex-a9’, ‘cortex-a12’, ‘cortex-a15’, ‘cortex-a17’, ‘cortex-a32’, ‘cortex-a35’, ‘cortex-a53’, ‘cortex-a55’, ‘cortex-a57’, ‘cortex-a72’, ‘cortex-a73’, ‘cortex-a75’, ‘cortex-a76’, ‘cortex-a76ae’, ‘cortex-a77’, ‘ares’, ‘cortex-r4’, ‘cortex-r4f’, ‘cortex-r5’, ‘cortex-r7’, ‘cortex-r8’, ‘cortex-r52’, ‘cortex-m0’, ‘cortex-m0plus’, ‘cortex-m1’, ‘cortex-m3’, ‘cortex-m4’, ‘cortex-m7’, ‘cortex-m23’, ‘cortex-m33’, ‘cortex-m35p’, ‘cortex-m1.small-multiply’, ‘cortex-m0.small-multiply’, ‘cortex-m0plus.small-multiply’, ‘exynos-m1’, ‘marvell-pj4’, ‘neoverse-n1’, ‘xscale’, ‘iwmmxt’, ‘iwmmxt2’, ‘ep9312’, ‘fa526’, ‘fa626’, ‘fa606te’, ‘fa626te’, ‘fmp626’, ‘fa726te’, ‘xgene1’.



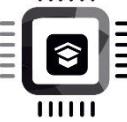
Analyzing .o files (Relocatable object files)

- main.o is in elf format(Executable and linkable format)
- ELF is a standard file format for object files and executable files when you use GCC
- A file format standard describes a way of organizing various elements(data, read-only data, code, uninitialized data, etc.) of a program in different sections.



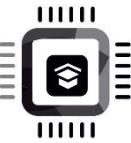
Other file formats

- The Common Object File Format (COFF) : Introduced by Unix System V
- Arm Image Format (AIF) : Introduced by ARM
- SRECORD: Introduced by Motorola

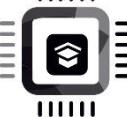


Objdump(displays information from object files.)

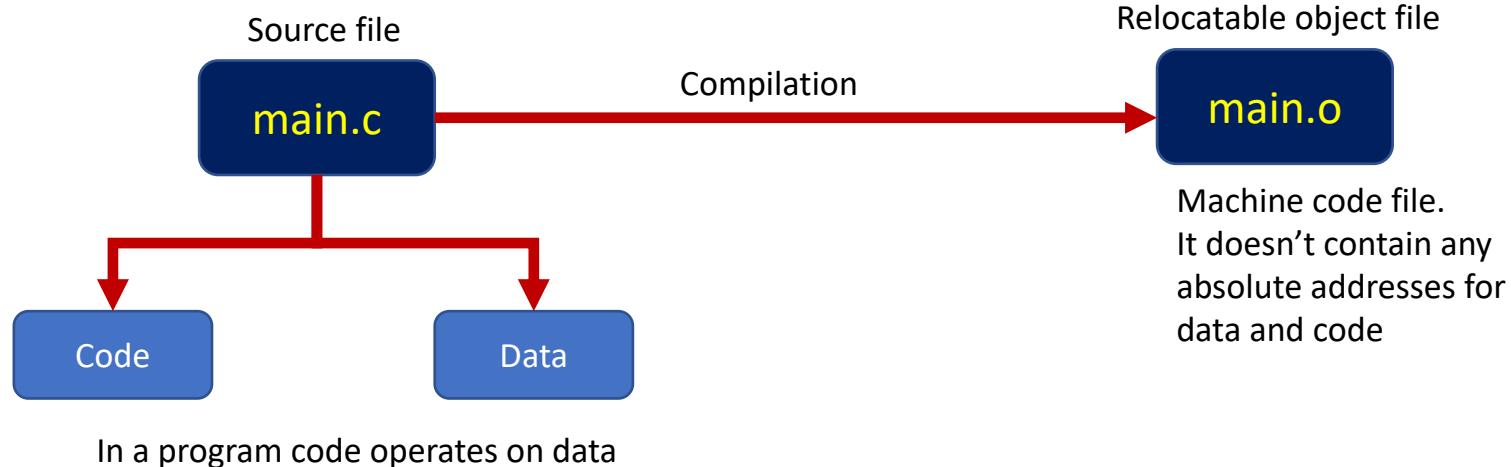
- Arm-none-eabi-objdump –d main.o
- Arm-none-eabi-objdump –D main.o
- Arm-none-eabi-objdump –h main.o
- arm-none-eabi-readelf.exe -p .comment main.o
- arm-none-eabi-objdump.exe -s main.o

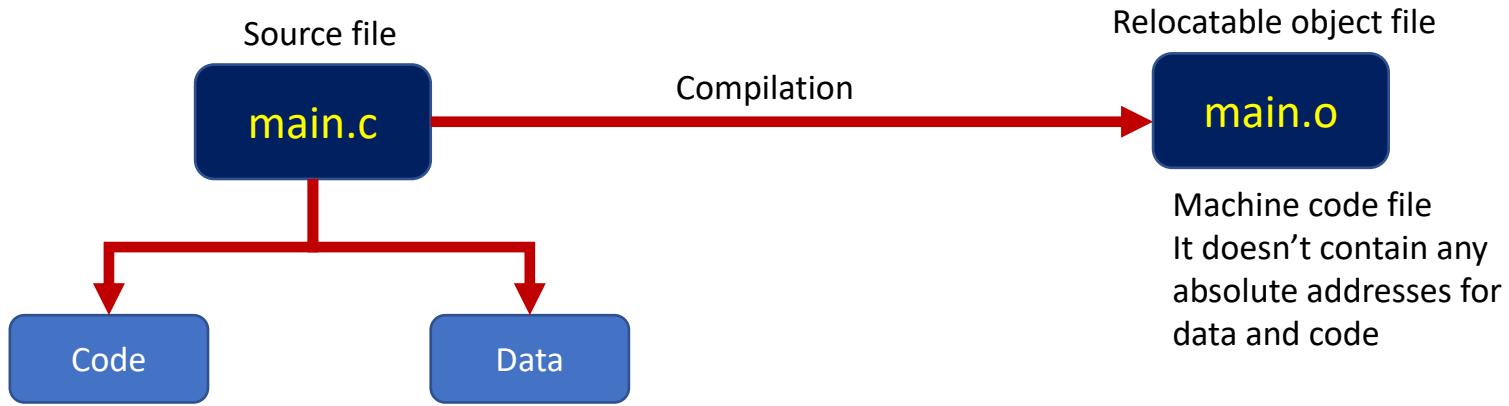
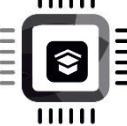


Code and data



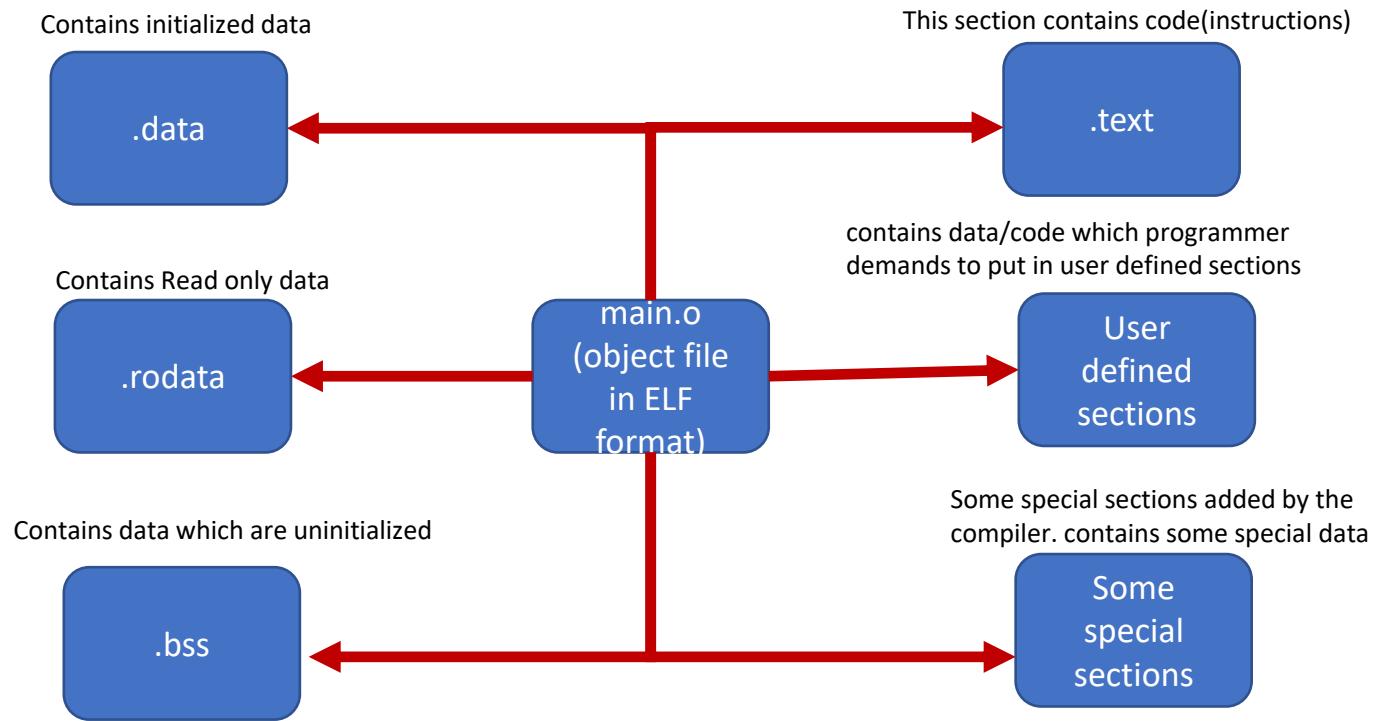
Relocatable object files



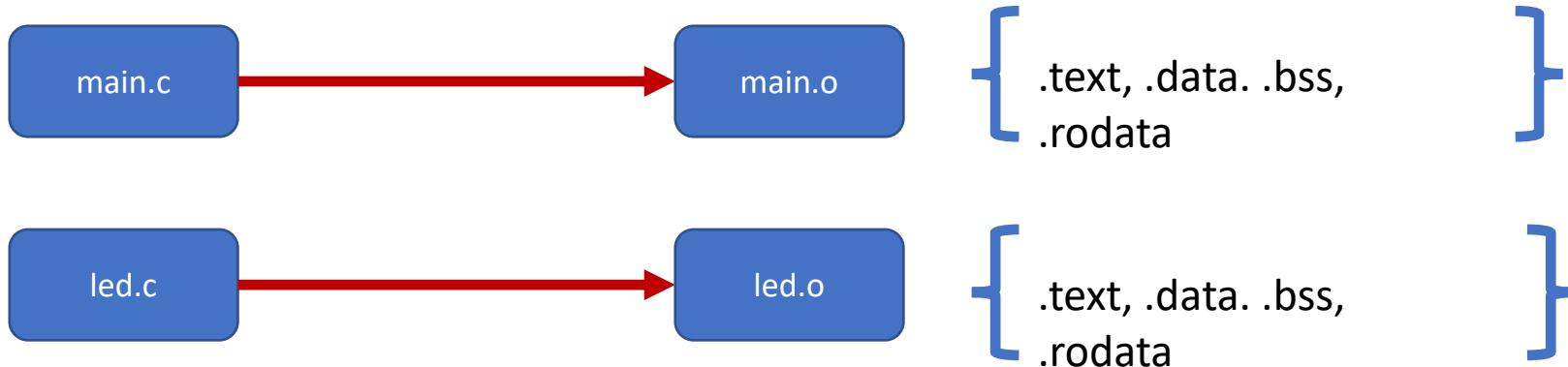
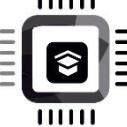


In a program code operates on data

LDR R0,[R1]
LDR R1,[R2]
ADD R1,R0,R1

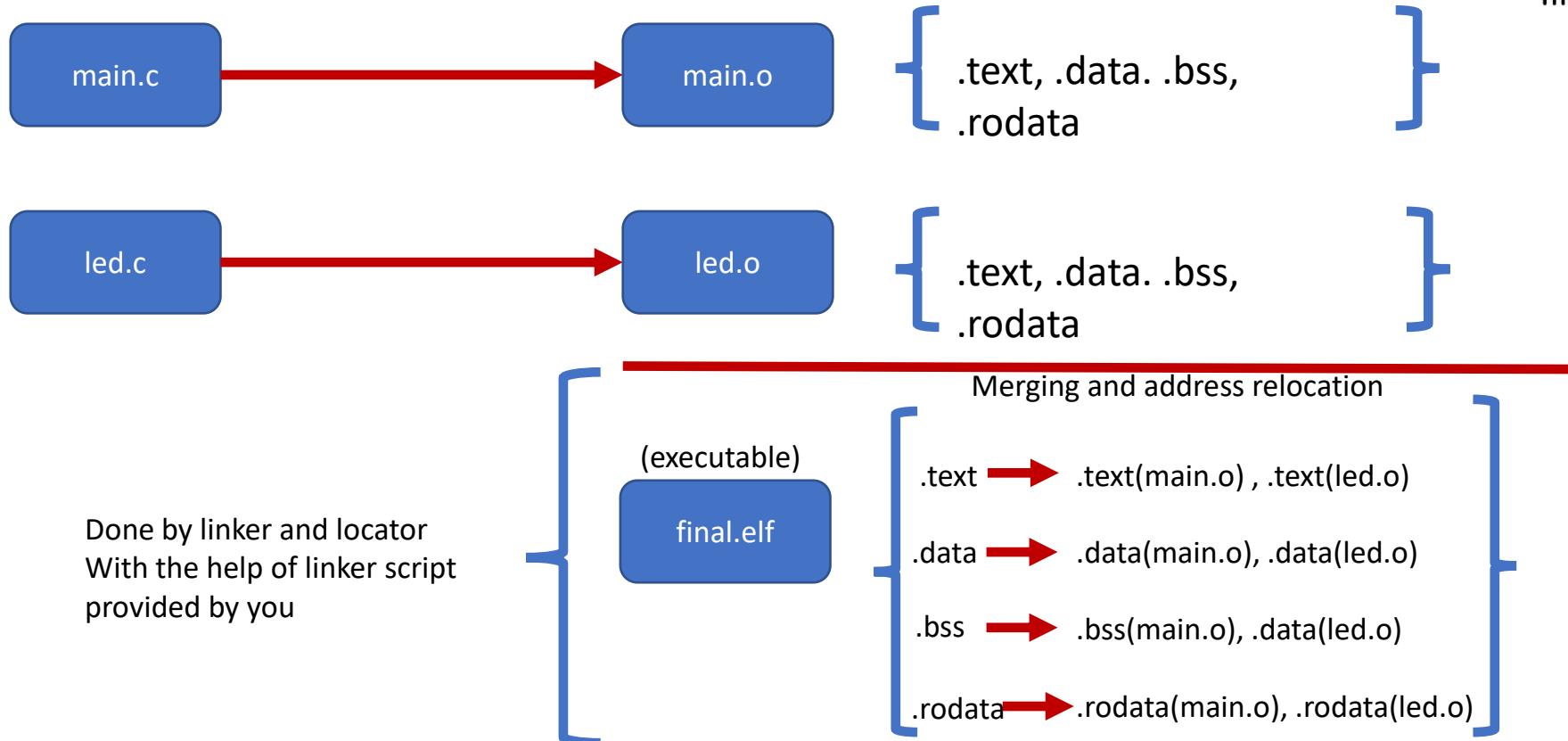
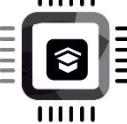


Different sections of the program in an ELF format



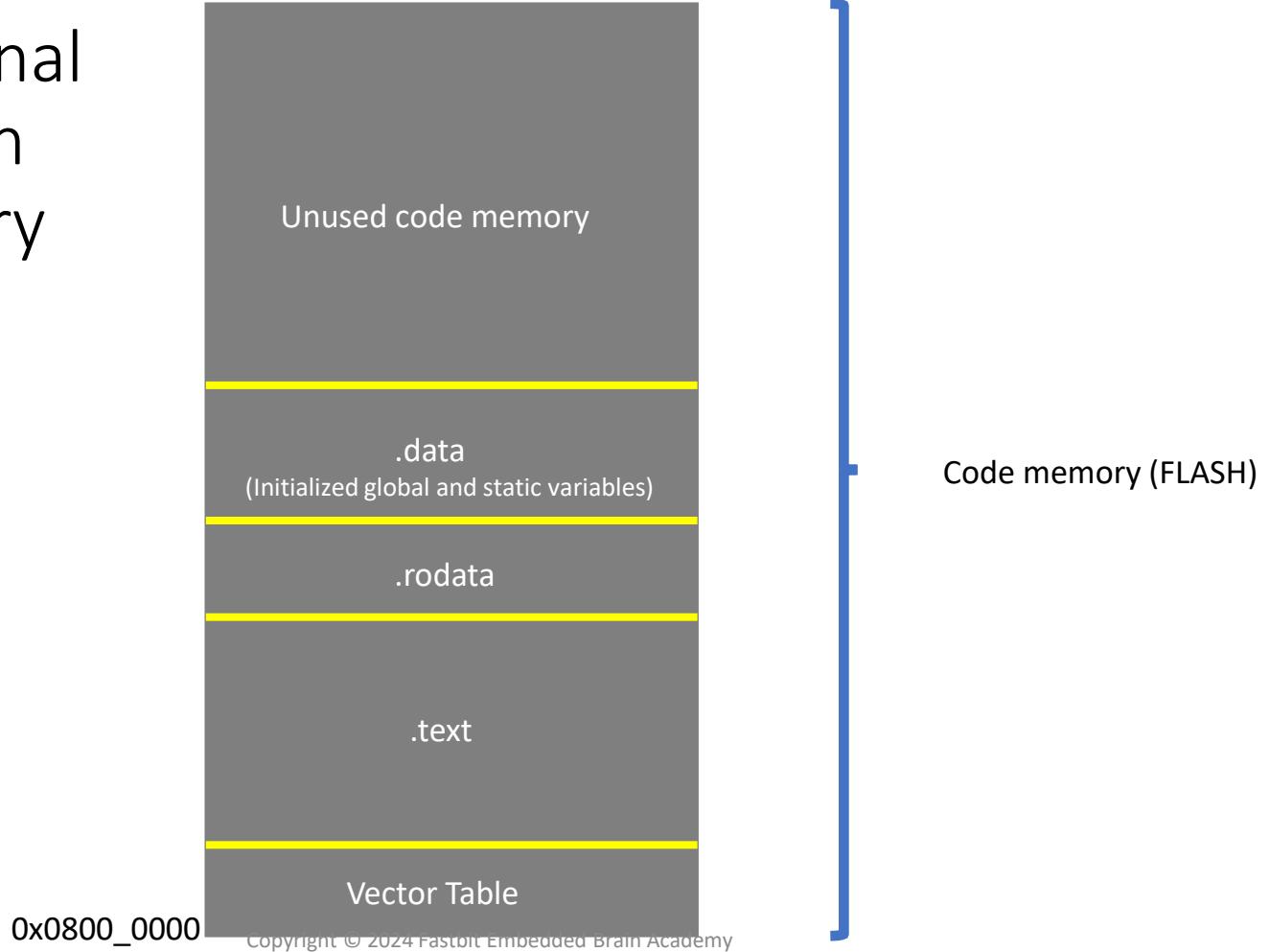
Linker and Locator

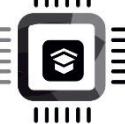
- ✓ Use the linker to merge similar sections of different object files and to resolve all undefined symbols of different object files.
- ✓ Locator (part of linker) takes the help of a linker script to understand how you wish to merge different sections and assigns mentioned addresses to different sections.



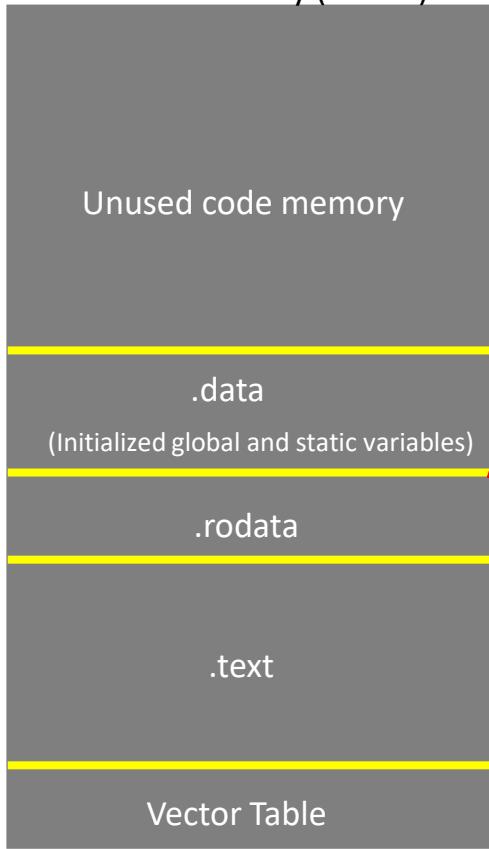


Storage of final executable in code memory

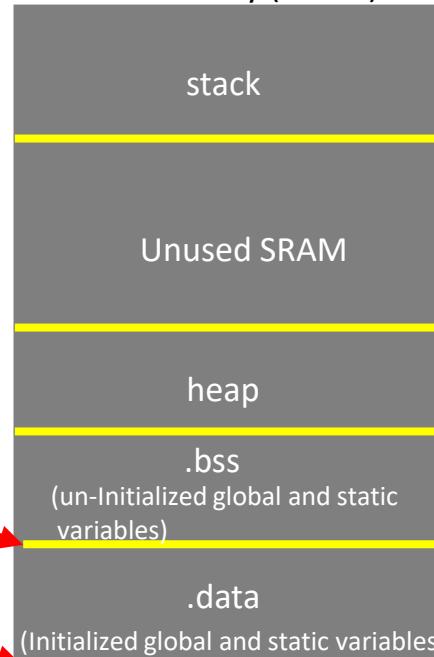




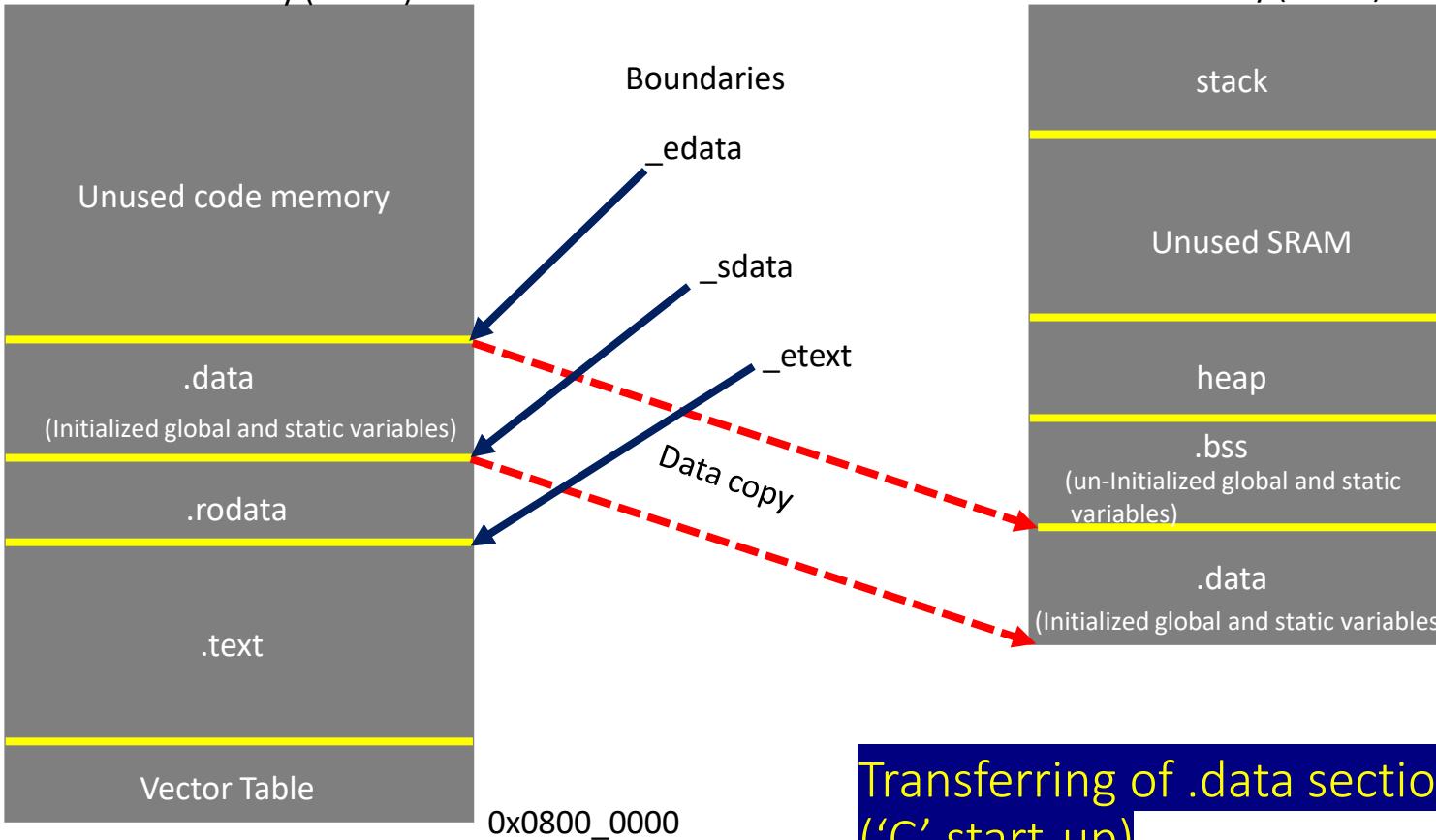
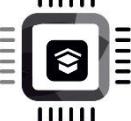
Code memory (FLASH)



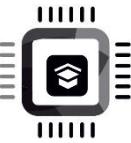
Data memory (SRAM)



Transferring of .data section to RAM
(‘C’ start-up)



Transferring of .data section to RAM
(‘C’ start-up)

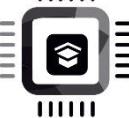


Startup file of a project

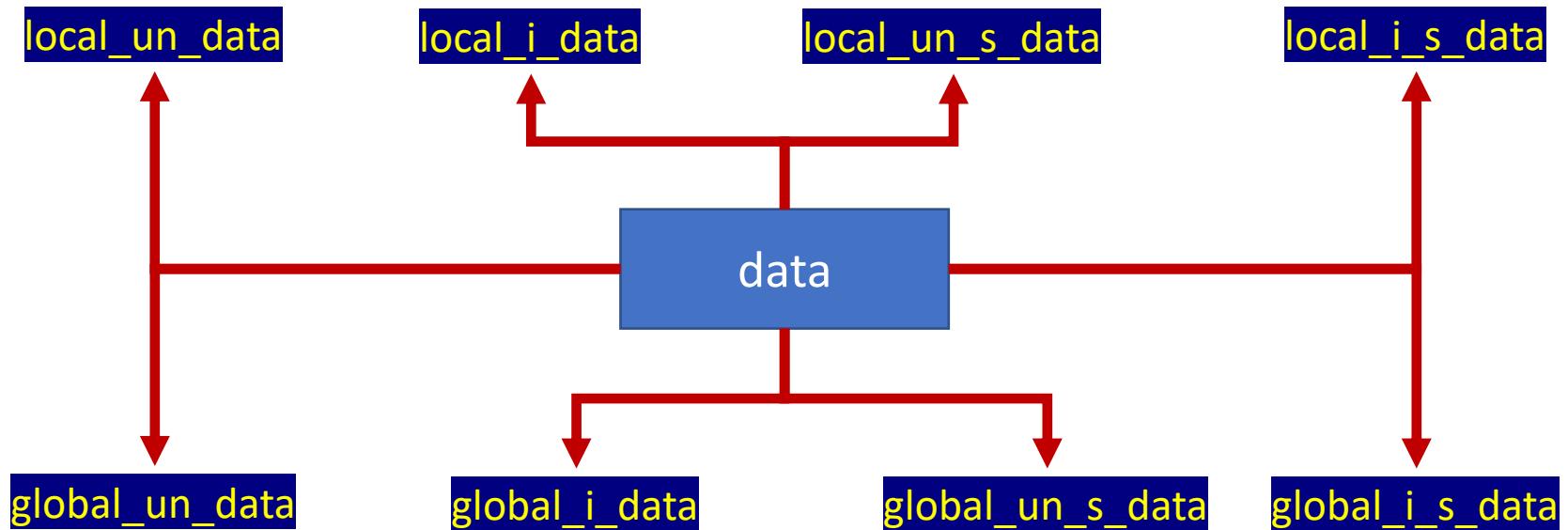


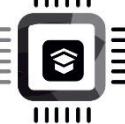
Importance of start-up file

- The startup file is responsible for setting up the right environment for the main user code to run
- Code written in startup file runs before main(). So, you can say startup file calls main()
- Some part of the startup code file is the target (Processor) dependent
- Startup code takes care of vector table placement in code memory as required by the ARM cortex Mx processor
- Startup code may also take care of stack reinitialization
- Startup code is responsible of .data, .bss section initialization in main memory

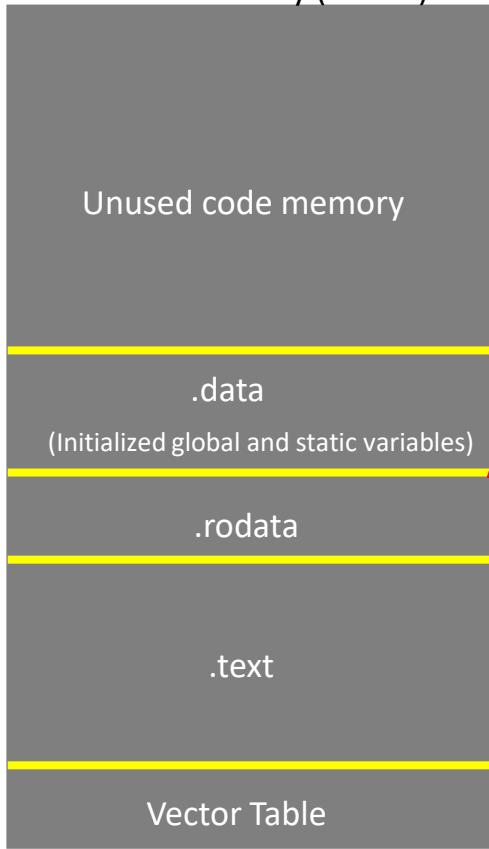


Different data of a program and related sections

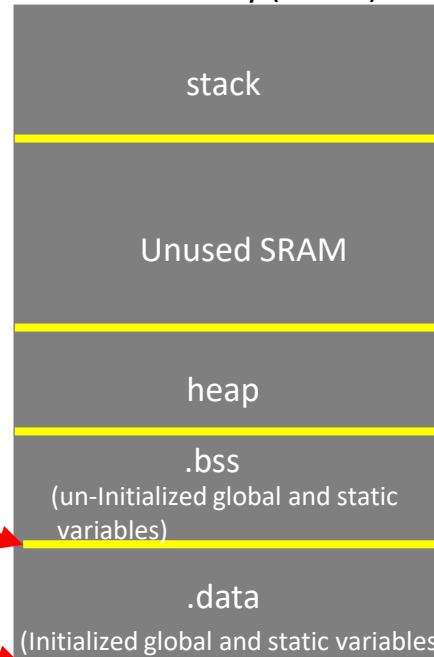




Code memory (FLASH)



Data memory (SRAM)



Transferring of .data section to RAM
(‘C’ start-up)



Variable (Data)	LOAD time	RUN time	Section	Note
Global initialized	FLASH	RAM	.data	Should be copied from flash to ram by startup code
Global uninitialized	----	RAM	.bss	startup code reserves space for this data in RAM and initializes to zero
Global static initialized	FLASH	RAM	.data	Should be copied from flash to ram by startup code
Global static uninitialized	----	RAM	.bss	startup code reserves space for this data in RAM and initializes to zero
Local initialized	----	STACK(RAM)	----	Consumed at run time
Local uninitialized	----	STACK(RAM)	----	Consumed at run time
Local static initialized	FLASH	RAM	.data	Should be copied from flash to ram by startup code
Local static uninitialized	----	RAM	.bss	startup code reserves space for this data in RAM and initializes to zero
All global const	FLASH	----	.rodata	
All local const	----	STACK(RAM)	----	Treated as locals



.bss(block started by symbol) and .data section

- All the uninitialized global variables and uninitialized static variables are stored in the **.bss** section.
- Since those variables do not have any initial values, they are not required to be stored in the **.data** section since the **.data** section consumes FLASH space. Imagine what would happen if there is a large global uninitialized array in the program, and if that is kept in the **.data** section, it would unnecessarily consume flash space yet carries no useful information at all.
- **.bss** section doesn't consume any FLASH space unlike **.data** section
- You must reserve RAM space for the **.bss** section by knowing its size and initialize those memory space to zero. This is typically done in startup code
- Linker helps you to determine the final size of the **.bss** section. So, obtain the size information from a linker script symbols.



```
#include<stdint.h>

uint32_t d_count = 300000;

int g_un_data1;
int g_un_data2 = 0;
int g_i_data = 0x55;//.data
static int g_un_s_data;
static int g_i_s_data = 0x44; //.data

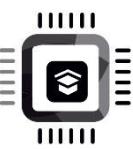
int main(void){

    printf("data = %d,%d,%d,%d,%d\n",g_un_data1,g_un_data2,g_i_data,g_un_s_data,g_i_s_data);

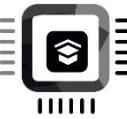
    int l_un_data;
    int l_i_data = 0x98;
    static int l_un_s_data;
    static int l_i_s_data = 0x24; //.data

    printf("sum = %d\n",l_un_data+l_i_data+l_un_s_data+l_i_s_data);

}
```

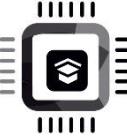


Linker scripts



Linker scripts

- Linker script is a text file which explains how different sections of the object files should be merged to create an output file
- Linker and locator combination assigns unique absolute addresses to different sections of the output file by referring to address information mentioned in the linker script
- 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
- You must supply linker script at the linking phase to the linker using –T option.



Linker scripts commands

- ENTRY
- MEMORY
- SECTIONS
- KEEP
- ALIGN
- AT>



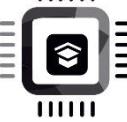
ENTRY command

- This command is used to set the "***Entry point address***" information in the header of 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 command

- 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 is 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
- By using memory command, you can fine-tune various memories available in your target and allow different sections to occupy different memory areas
- Typically one linker script has one memory command



Memory command

Syntax :

```
MEMORY
```

```
{
```

```
    name (attr) : ORIGIN = origin, LENGTH = len
```

```
}
```

defines origin address of the memory region

Defines name of the memory region which will be later referenced by other parts of the linker script

Defines the length information



Memory command

Syntax :

MEMORY

{

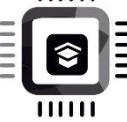
name (attr) : ORIGIN = origin, LENGTH = len

}

Defines the attribute list of the memory region

Valid attribute lists must be made up of the characters
"ALIRWX" that match section 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.



Memory command

Syntax :

```
MEMORY
{
    name (attr) : ORIGIN = origin, LENGTH = len
}
```

Microcontroller	FLASH Size (in KB)	SRAM1 size (in KB)	SRAM2 size(in KB)
STM32F4VGT6	1024	112	16



Sections command

- 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 this command, you also mention the placement of a section 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.

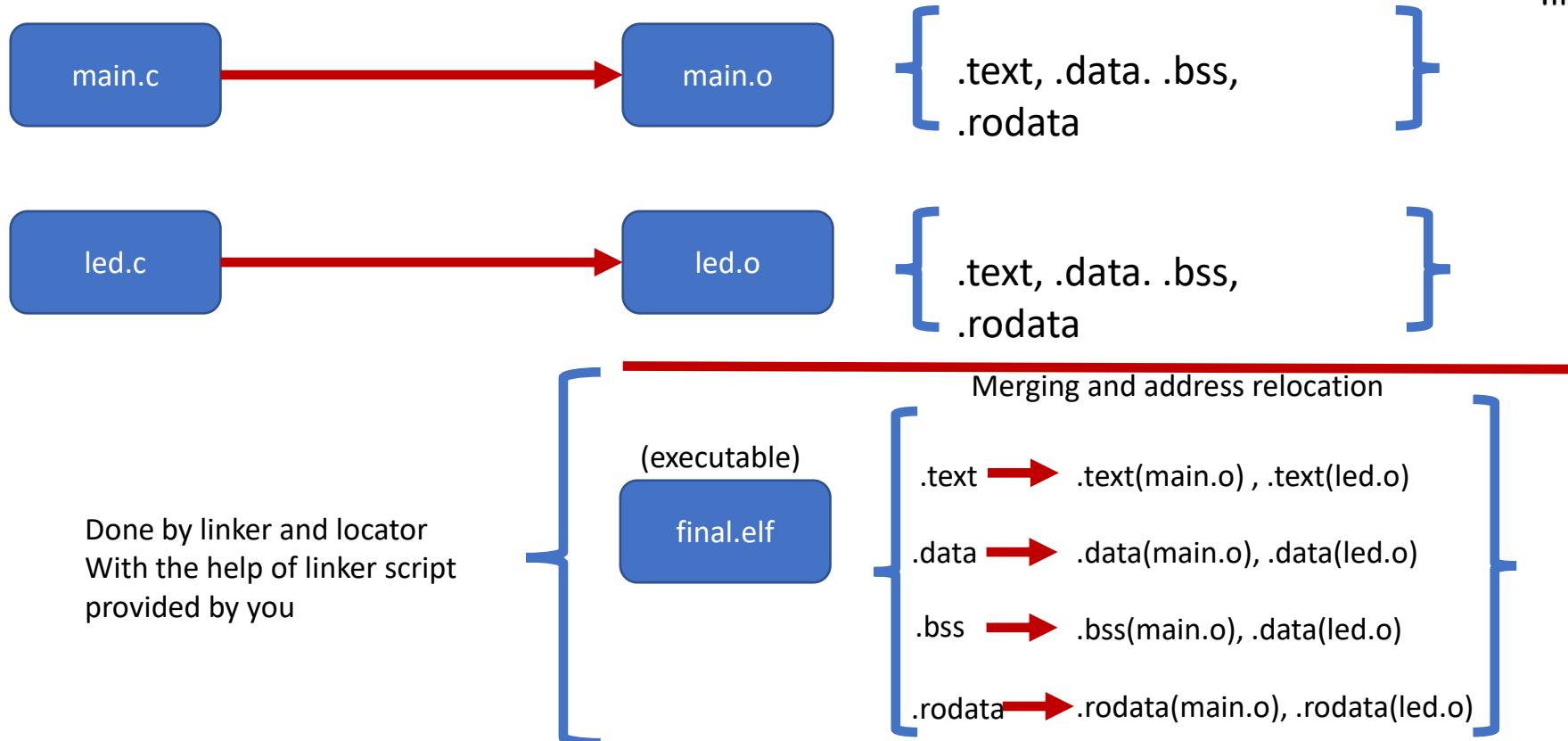
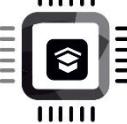


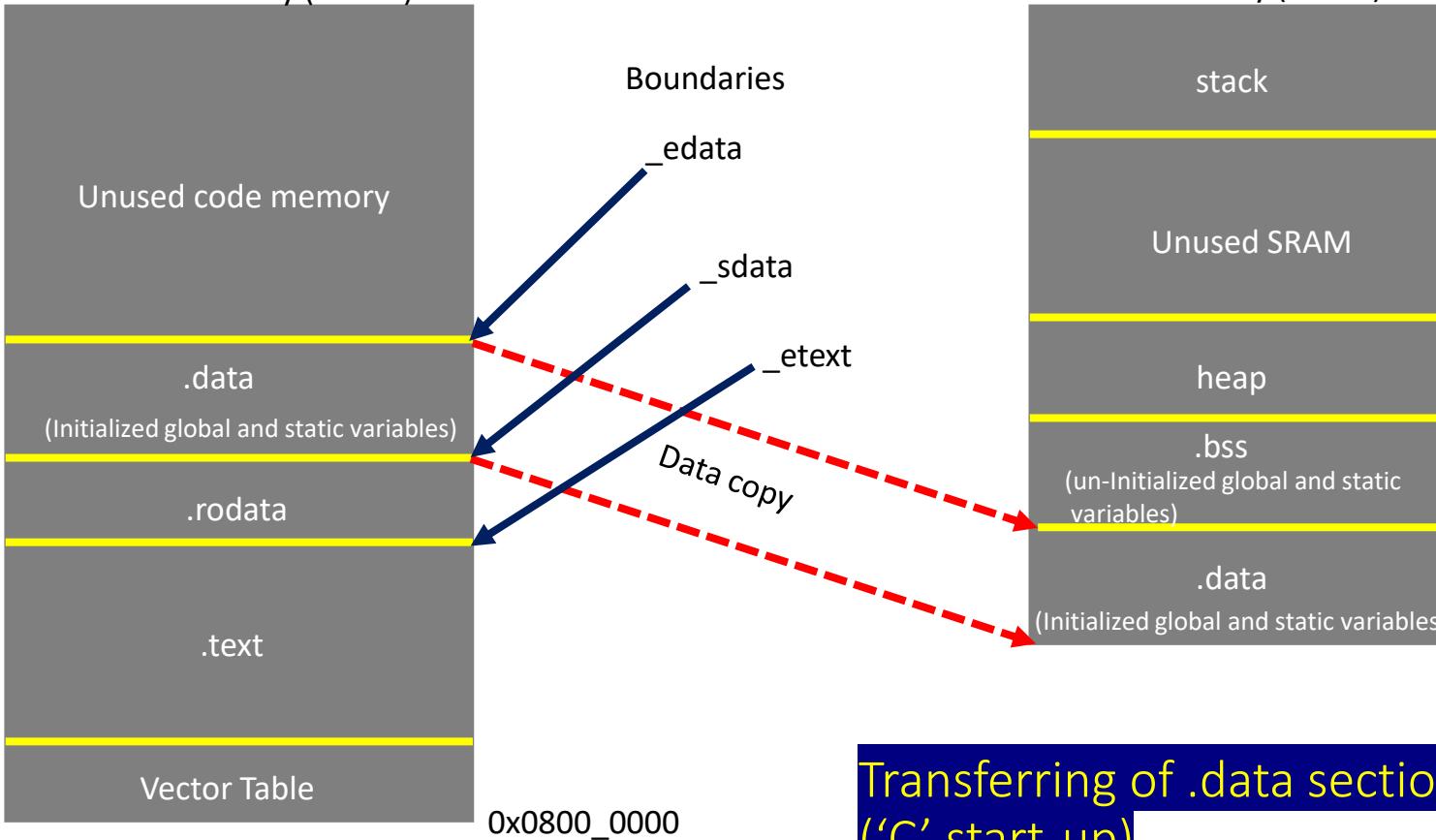
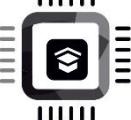
```
/* Sections */
SECTIONS
{
    /* This section should include .text section of all input files */
    .text :
    {
        //merge all .isr_vector section of all input files
        //merge all .text section of all input files
        //merge all .rodata section of all input files

    } >(vma) AT>(lma)

    /* This section should include .data section of all input files */
    .data :
    {
        //here merge all .data section of all input files

    } >(vma) AT>(lma)
}
```





Transferring of .data section to RAM
(‘C’ start-up)



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(address) information
- You can use this symbol inside the linker script to track and define boundaries of various sections
- You can also set location counter to any specific value while writing linker script
- Location counter should appear only inside the **SECTIONS** command
- The location counter is incremented by the size of the output section



Linker script symbol

- A symbol is the name of an address
- A symbol declaration is not equivalent to a variable declaration what you do in your 'C' application

main.c

```
int my_value = 100;  
(global variable)  
  
my_value = 50;  
  
void fun1 (void)  
{ ... }
```



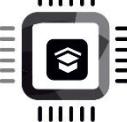
main.o

Symbol table

(address)	(symbol)
0x2000_000	my_value
0	
0x0800_000	fun1
0	



```
1 ENTRY(Reset_Handler)
2
3 MEMORY
4 {
5   FLASH(rx):ORIGIN=0x08000000,LENGTH=1024K
6   SRAM(rwx):ORIGIN=0x20000000,LENGTH=128K
7 }
8
9 __max_heap_size = 0x400; /* A symbol declaration . Not a variable !!! */
10 __max_stack_size = 0x200; /* A symbol declaration . Not a variable !!! */
11
12 SECTIONS
13 {
14   .text :
15   {
16     *(.isr_vector)
17     *(.text)
18     *(.rodata)
19     end_of_text = .; /* Store the updated location counter value in to a symbol 'end_of_text' */
20   }> FLASH
21
22 .data
23 {
24   start_of_data = 0x20000000; /* Assign a value to a symbol 'start_of_data' */
25   *(.data)
26 }> SRAM AT> FLASH
27
```



Linker Script Symbols

- You can create symbols inside the linker script and assign any values
- Symbols created inside can be accessed by a ‘C’ program using ‘extern’ keyword
- Symbols can be created anywhere in the linker script but the special symbol ‘.’(location counter) can only be used inside the SECTIONS command



ALIGN

- Used to align the location counter with the boundary specified.
- Typical usage `. = ALIGN(exp) //exp = 4, 8, 16...`
- Return the result of the current location counter (.) aligned to the next `exp` boundary. `exp` must be an expression whose value is a power of two

Example : `. = ALIGN(4)`

Let's say previous value of . is 0x2000_0001

Updated value of . Will be : 0x2000_0004



Location counter (.)

- If you do not specify the address of an output section in some other way (other ways are described later), the address is set from the current value of the location counter. The location counter is then incremented by the size of the output section.
- At the start of the 'SECTIONS' command, the location counter has the value '0'

```
SECTIONS
{
    . = 0x10000;
    .text : { *(.text) }
    . = 0x8000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

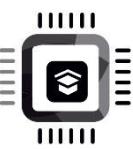
Since the location counter is '0x10000' when the output section `text` is defined, the linker will set the address of the `text` section in the output file to be '0x10000'.

The remaining lines define the `data` and `bss` sections in the output file. The linker will place the `data` output section at address '0x8000000'. After the linker places the `data` output section, the value of the location counter will be '0x8000000' plus the size of the `data` output section. The effect is that the linker will place the `bss` output section immediately after the `data` output section in memory.



Linker script symbols

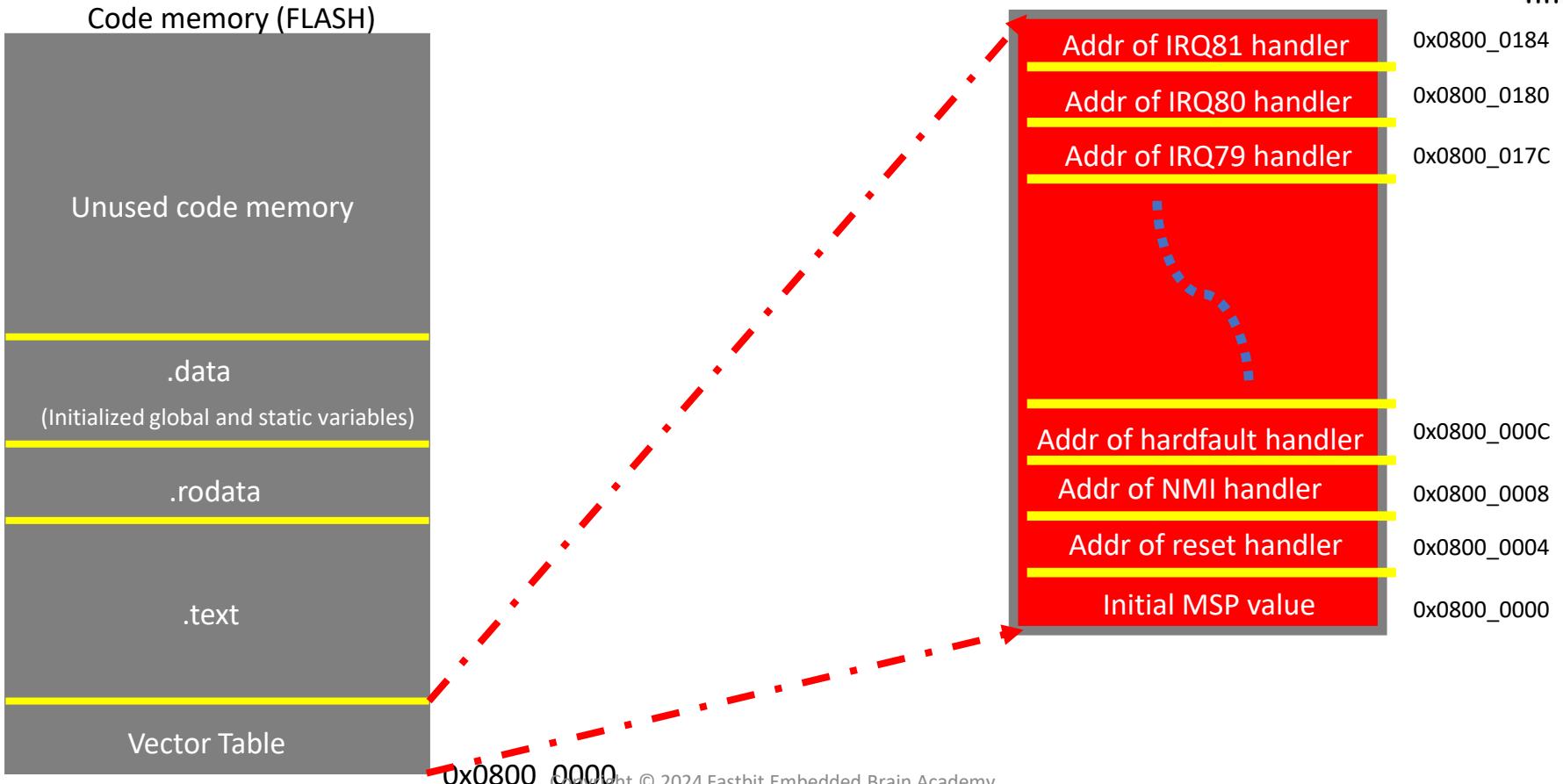
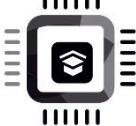
- Inside the linker script you can create some symbols to store some data such as boundary values of different sections.
- These symbols can be read by the C code



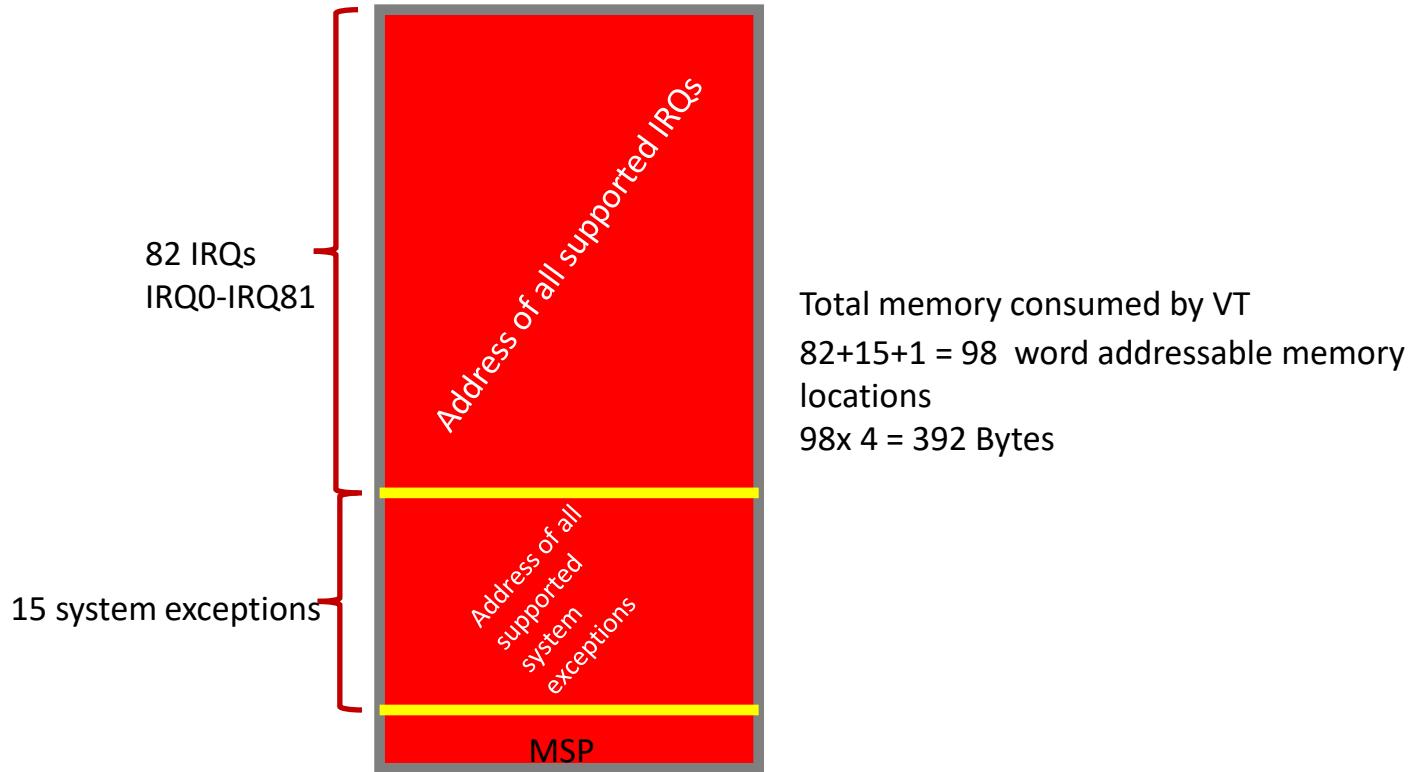
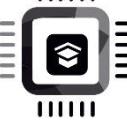
Start-up file

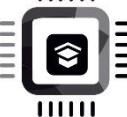
1. Create a vector table for your microcontroller . Vector tables are MCU specific
2. Write a start-up code which initializes .data and .bss section in SRAM
3. Call main()

Vector table placement



Exceptions and interrupts in STM32F4VGTx MCU





Creating a Vector Table

- Create an array to hold MSP and handlers addresses.

```
uint32_t vectors[] = {store MSP and address of various handlers here};
```

- Instruct the compiler not to include the above array in `.data` section but in a different user defined section



variable attribute

__attribute__((section("name")))

- The section attribute specifies that a variable must be placed in a special section mentioned by the programmer
- Normally, the compiler places the data it generates in sections like **.data** and **.bss**. However, you might require additional data sections, or you might want a variable to appear in a special section, for example, to map to special hardware.



Function attribute : weak and alias

Weak :

Lets programmer to override already defined weak function(dummy) with the same function name

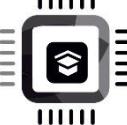
Alias :

Lets programmer to give alias name for a function

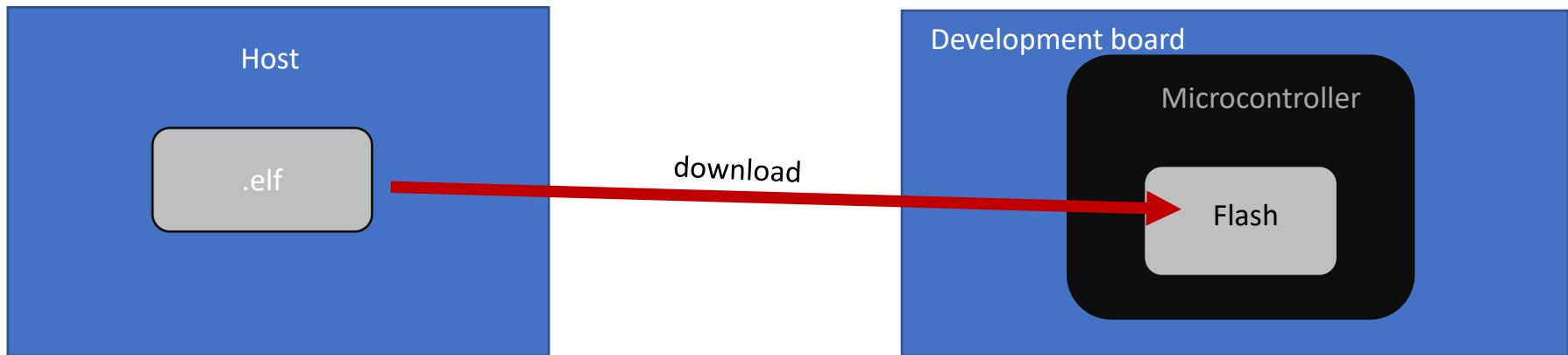


Linker symbol

- <https://stackoverflow.com/questions/8398755/access-symbols-defined-in-the-linker-script-by-application>
- A symbol is a name for an address
- In particular a linker script symbol is not equivalent to a variable declaration in a high level language. it is instead a symbol that does not have a value.
- Linker scripts symbol declarations, by contrast, create an entry in the symbol table but do not assign any memory to them. Thus they are an address without a value. So for example the linker script definition:
- foo = 1000; here we are setting an address for the symbol ‘foo’
- Hence when you are using a linker script defined symbol in source code you should always take the address of the symbol, and never attempt to use its value.



Downloading and debugging executable





Downloading executable to Target





OpenOCD(Open On Chip Debugger)

- The Open On-Chip Debugger (OpenOCD) aims to provide debugging, in-system programming, and boundary-scan testing for embedded target devices.
- Its free and open-source host application allows you to program, debug, and analyze your applications using GDB
- It supports various target boards based on different processor architecture
- OpenOCD currently supports many types of debug adapters: USB-based, parallel port-based, and other standalone boxes that run OpenOCD internally
- GDB Debug: It allows ARM7 (ARM7TDMI and ARM720t), ARM9 (ARM920T, ARM922T, ARM926EJ-S, ARM966E-S), XScale (PXA25x, IXP42x), Cortex-M3 (Stellaris LM3, ST STM32, and Energy Micro EFM32) and Intel Quark (x10xx) based cores to be debugged via the GDB protocol.
- Flash Programming: Flash writing is supported for external CFI-compatible NOR flashes (Intel and AMD/Spansion command set) and several internal flashes (LPC1700, LPC1800, LPC2000, LPC4300, AT91SAM7, AT91SAM3U, STR7x, STR9x, LM3, STM32x, and EFM32). Preliminary support for various NAND flash controllers (LPC3180, Orion, S3C24xx, more) is included



Programming adapters

- Programming adapters are used to get access to the debug interface of the target with native protocol signaling such as SWD or JTAG since HOST doesn't support such interfaces.
- It does protocol conversion. For example, commands and messages coming from host application in the form of USB packets will be converted to equivalent debug interface signaling (SWD or JTAG) and vice versa
- Mainly debug adapter helps you to download and debug the code
- Some advanced debug adapters will also help you to capture trace events such as on the fly instruction trace and profiling information



Programming adapters

- *debug adapter*, which is a small hardware module which helps provide the right kind of electrical signaling to the target being debugged
- These are required since the debug host (on which OpenOCD runs) won't usually have native support for such signaling, or the connector needed to hook up to the target.
- *JTAG Adapter* supports JTAG signaling, and is used to communicate with JTAG (IEEE 1149.1) compliant TAPs on your target board. A *TAP* is a “Test Access Port”, a module which processes special instructions and data. TAPs are daisy-chained within and between chips and boards. JTAG supports debugging and boundary scan operations.
- There are also *SWD Adapters* that support Serial Wire Debug (SWD) signaling to communicate with some newer ARM cores, as well as debug adapters which support both JTAG and SWD transports. SWD supports only debugging, whereas JTAG also supports boundary scan operations.
- **2.2.3. JTAG and SWD interface**
- The external JTAG interface has four mandatory pins, **TCK**, **TMS**, **TDI**, and **TDO**, and an optional reset, **nTRST**. JTAG-DP and SW-DP also require a separate power-on reset, **nPOTRST**.
- The external SWD interface requires two pins:
- a bidirectional **SWDIO** signal
- a clock, **SWCLK**, which can be input or output from the device.



Some of the popular debug adapters



SEgger J-Link EDU - JTAG/SWD Debugger

Multiple CPUs supported—8051, PIC32, RX, ARM7/9/11, Cortex-M/R/A, RISC-V

Download speed up to 1 MByte/s

Debug Protocol : JTAG/SWD

Target Interface : 20-pin



Some of the popular debug adapters



provides extended on-the-fly debug capabilities for Cortex-M devices. You can control the processor, set breakpoints, and read/write memory contents, all while the processor is running at full speed. High-Speed data and instruction trace are streamed directly to your PC enabling you to analyze detailed program behavior.

- Target Connectors 10-pin (0.05") - Cortex Debug Connector
- 20-pin (0.10") - ARM Standard JTAG Connector
- 20-pin (0.05") - Cortex Debug+ETM Connector



Some of the popular debug adapters



The ST-LINK/V2 is an in-circuit debugger and programmer for the STM8 and STM32 microcontroller families.

The single wire interface module (SWIM) and JTAG/serial wire debugging (SWD) interfaces are used to communicate with any STM8 or STM32 microcontroller located on an application board.



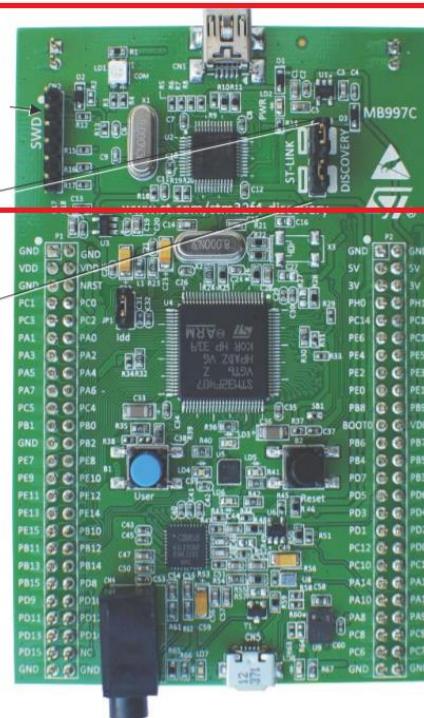
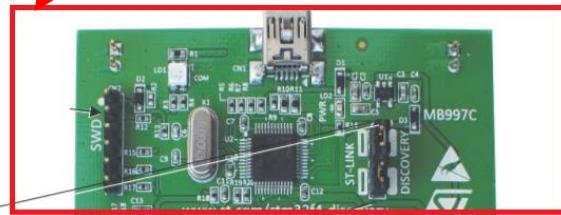
STM32F4-DISC1 board

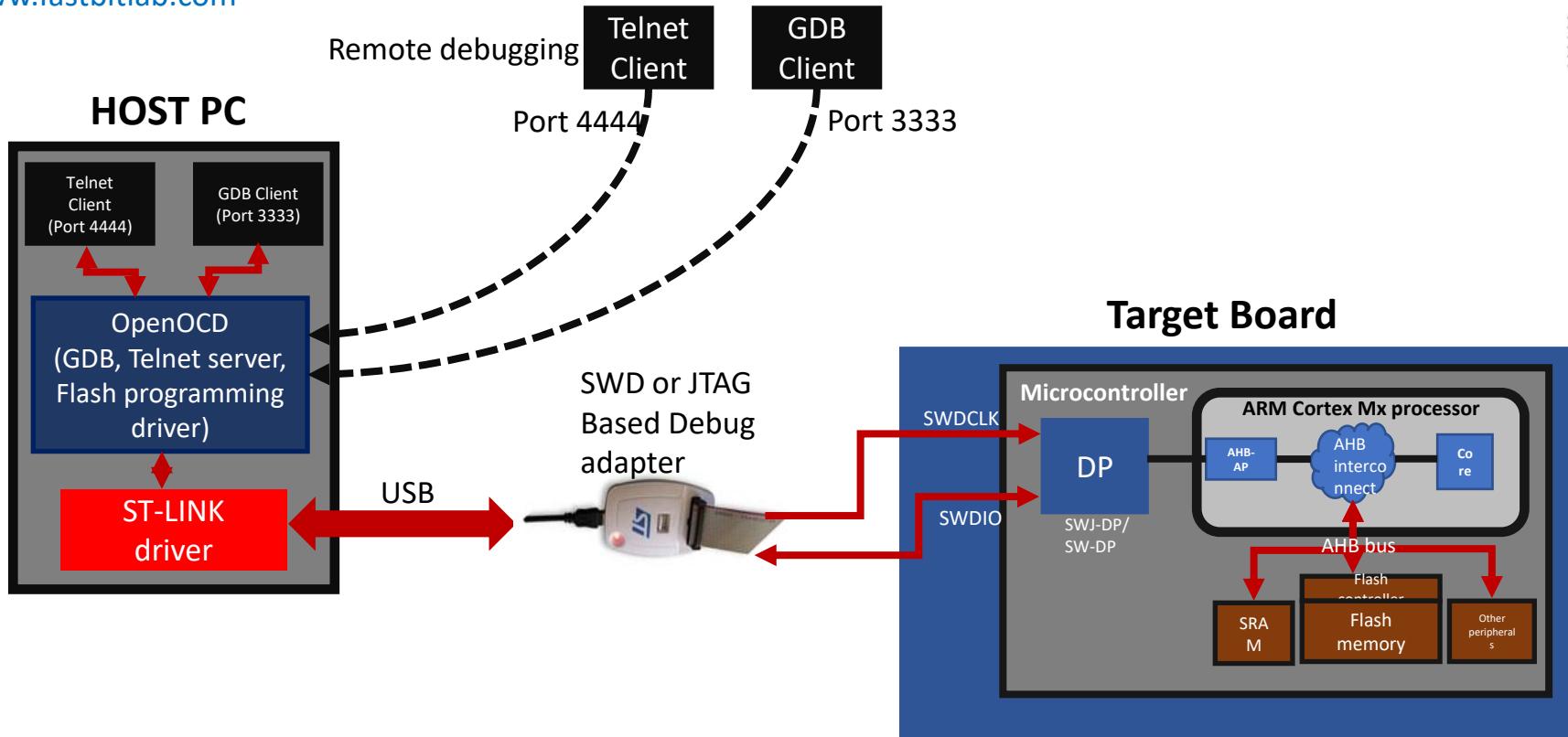
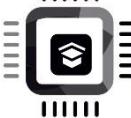
- It has embedded ST-LINK-V2A in-circuit programmer and debugger
- Supports only SWD debug protocol

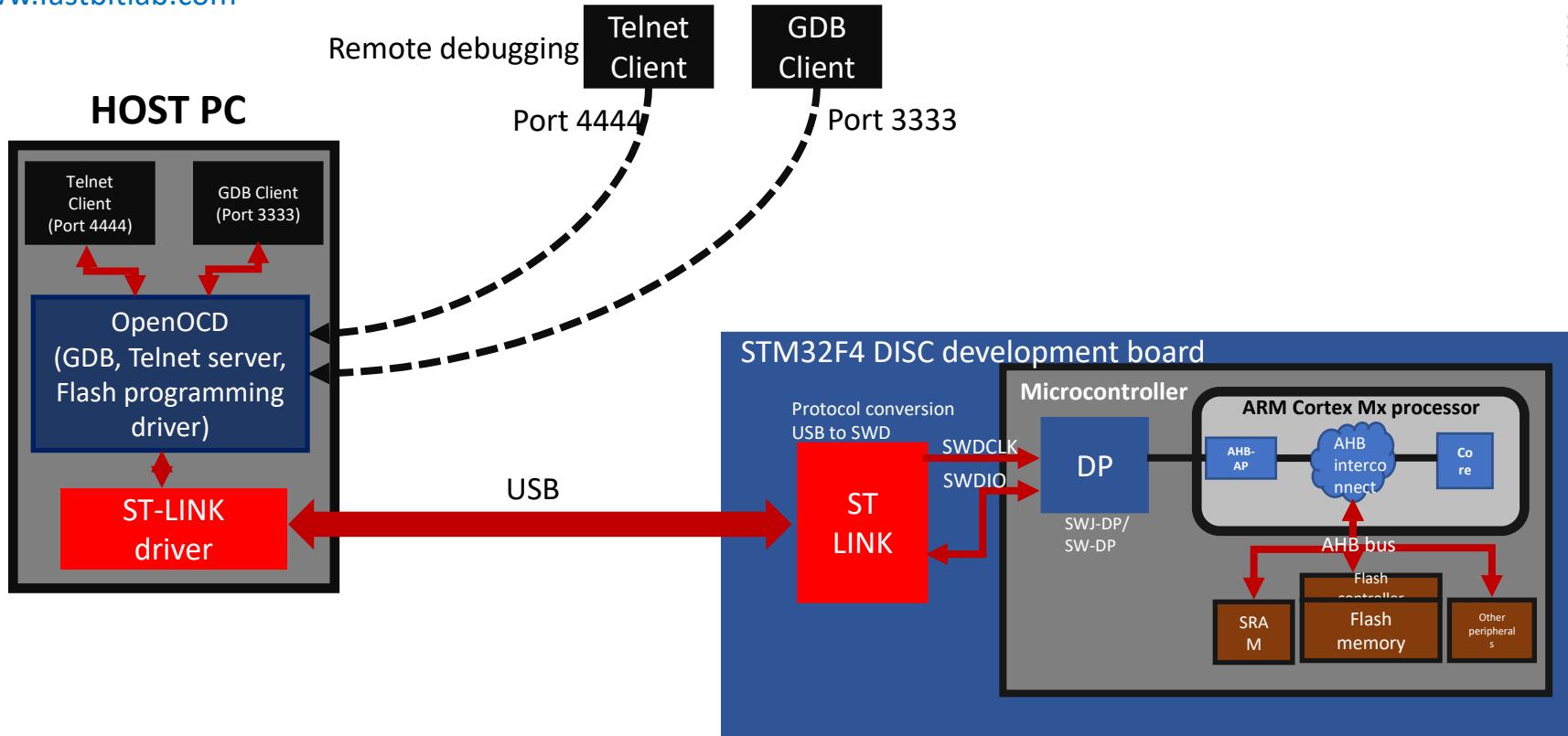


CN3
Jumpers OFF

Embedded ST-LINK-V2A
in-circuit programmer and
debugger



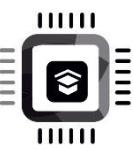






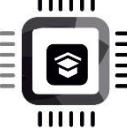
Steps to download the code using OpenOCD

1. Download and install OpenOCD
2. Install Telnet client (for windows you can use PuTTY software)
 - If you cannot use Telnet application you can also use “GDB Client”
3. Run OpenOCD with the board configuration file
4. Connect to the OpenOCD via Telnet Client or GDB client
5. Issue commands over Telnet or GDB Client to OpenOCD to download and debug the code



AHB-AP

- The **Advanced High-performance Bus (AHB)** is a bus present on ARM devices that interconnects the memory and peripherals present on the MCU. The **AHB-AP** exposes access to this bus via several registers. The most common registers used in the **AHB-AP** are:

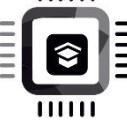


‘C’ standard library newlib & newlib-nano



Newlib

- Newlib is a ‘C’ standard library implementation intended for use on embedded systems, and it is introduced by Cygnus Solutions (now Red Hat)
- “Newlib” is written as a Glibc(GNU libc) replacement for embedded systems. It can be used with no OS (“bare metal”) or with a lightweight RTOS
- Newlib ships with gnu ARM toolchain installation as the default C standard library
- GNU libc (glibc) includes ISO C, POSIX, System V, and XPG interfaces. uClibc provides ISO C, POSIX and System V, while Newlib provides only ISO C



Newlib-nano

- Due to the increased feature set in newlib, it has become too bloated to use on the systems where the amount of memory is very much limited.
- To provide a C library with a minimal memory footprint, suited for use with micro-controllers, ARM introduced newlib-nano based on newlib



Locating newlib and newlib nano

Acer (C:) > Program Files (x86) > GNU Tools Arm Embedded > 9 2019-q4-major > arm-none-eabi > lib >

Newlib 'C' standard library

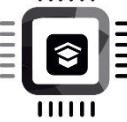
Newlib-nano 'C' standard library

semihosting library
that is used for console
IO operations like
printf

Spec files

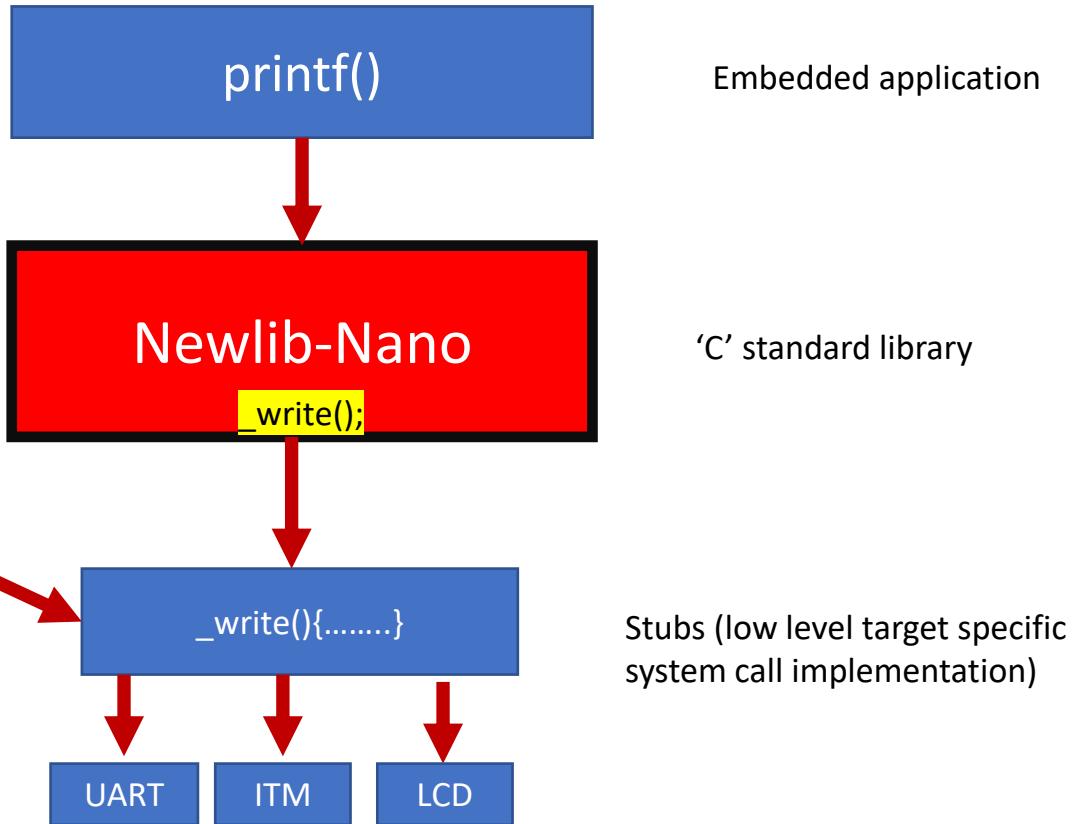
Spec files

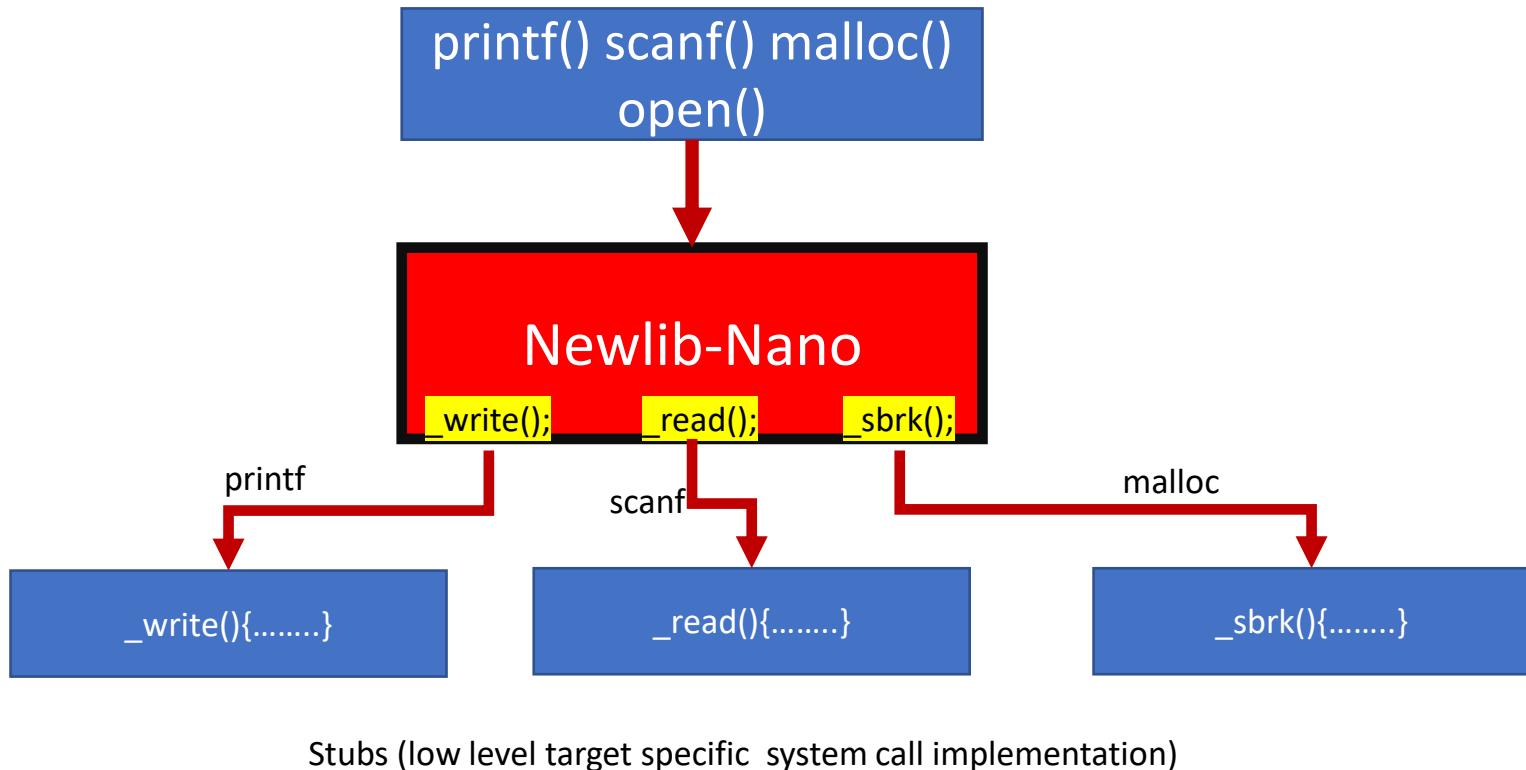
- arm
- cpu-init
- ldscripts
- thumb
- aprofile-validation.specs
- aprofile-validation-v2m.specs
- aprofile-ve.specs
- aprofile-ve-v2m.specs
- crt0.o
- iq80310.specs
- libc.a
- libc_nano.a
- libg.a
- libg_nano.a
- libgloss-linux.a
- libm.a
- libnosys.a
- librdimon.a
- librdimon_nano.a
- librdimon-v2m.a
- librdpmon.a
- libstdc++.a
- libstdc++.a-gdb
- libstdc++_nano.a
- libsopc++.a
- libsopc+_nano.a
- linux.specs
- linux-crt0.o
- nano.specs
- nosys.specs
- pid.specs
- rdimon.specs
- rdimon-crt0.o
- rdimon-crt0-v2m.o
- rdimon-v2m.specs
- rdpmon.specs
- rdpmon-crt0.o
- redboot.id
- redboot.specs
- redboot-crt0.o
- redboot-syscalls.o

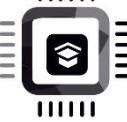


Low level system Calls

- The idea of Newlib is to implement the hardware-independent parts of the standard C library and rely on a few low-level system calls that must be implemented with the target hardware in mind.
- When you are using newlib , you must implement the system calls appropriately to support devices, file-systems, and memory management.







System Calls

- Please download the system calls implementation file **syscalls.c** attached with this lecture and place it in your workspace

**-specs=*file***

Process *file* after the compiler reads in the standard ‘specs’ file, in order to override the defaults which the `gcc` driver program uses when determining what switches to pass to `cc1`, `cc1plus`, `as`, `ld`, etc. More than one ‘**-specs=*file***’ can be specified on the command line, and they are processed in order, from left to right. See [Section 3.19 \[Spec Files\], page 428](#), for information about the format of the *file*.



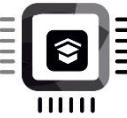
Newlib

- Newlib is just a C standard library implementation intended for use on embedded systems
- useful to have the standard C functions we all learn during our first programming course: printf, malloc, getchar, strncpy, ... A common way to have them is using Newlib.
- Newlib is an implementation of the standard C library that is specifically thought to run somewhere with low resources and undefined hardware
- Newlib is a C library that can be used on embedded systems. However, as the feature set has been increased it has become too bloated for use on the very smallest systems where the amount of memory can be very limited. In order to provide a C library with a very small footprint, suited for use with micro-controllers, we have created newlib-nano based on newlib 1.19.0



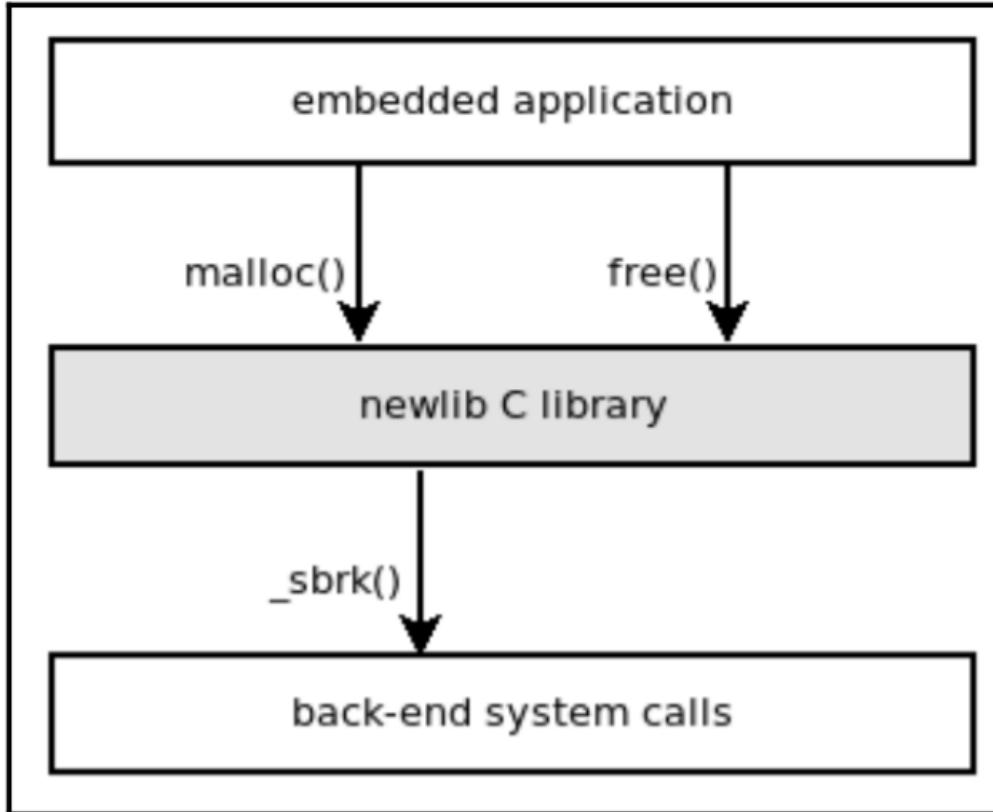
Newlib

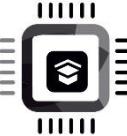
- The idea of Newlib is to implement the hardware-independent parts of the standard C library and rely on few low-level system calls that must be implemented with the target hardware in mind.
- The requirements for each stub are fully documented in newlib's libc.info file, in the section called Syscalls. The key to a successfully ported newlib is providing stubs that bridge the gap between the functionality newlib needs, and what your target system can provide.
- main difference is the size of the library, uclibc and newlib are focused on embedded systems so they want to be small and fast, while glibc is focused in full functionality.
- GNU libc (glibc) includes ISO C, POSIX, System V, and XPG interfaces. uClibc provides ISO C, POSIX and System V, while Newlib provides only ISO C.
- “Newlib” is written as a glibc replacement for embedded systems. It can be used with no OS (“bare metal”) or with a lightweight RTOS
- Newlib is the default library for embedded GCC distributions. Besides being smaller than regular glibc, it is also more friendly toward embedded developers with its licensing terms



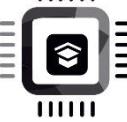
Nano-lib

- To use newlib-nano, users should provide additional gcc compile and link time option:
 - `--specs=nano.specs`
- Please note that `--specs=nano.specs` is a both a compiler and linker option. Be sure to include in both compiler and linker options if compiling and linking are separated.
- Users can choose to use or not use semihosting by following instructions. ** semihosting If you need semihosting, linking like:
- `$ arm-none-eabi-gcc --specs=rdimon.specs $(OTHER_LINK_OPTIONS)`





- ** non-semihosting/retarget If you are using retarget, linking like:
- \$ arm-none-eabi-gcc --specs=nosys.specs \$(OTHER_LINK_OPTIONS)



--gc-collections

- `-Wl,--gc-sections` flag tells the linker to garbage collect unused sections. That is, any sections that are not referenced are removed from the resulting binary, which can shrink the binary.
- `-Wl,-Map=tut.map` tells the linker to generate a map file and stick it into `tut.map`. The map file is helpful for debugging but is informational only.

```
nieki@LAPTOP-08FKS4E3 MINGW64 ~/OneDrive/ofc/courses/my_workspace
$ arm-none-eabi-gcc *.o -o final.elf
c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none-eabi/bin
/ld.exe: c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none
-eabi/lib\libc.a(lib_a-exit.o): in function `exit':
exit.c:(.text.exit+0x2c): undefined reference to `_exit'
c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none-eabi/bin
/ld.exe: c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none
-eabi/lib\libc.a(lib_a-sbrkr.o): in function `_sbrk_r':
sbrkr.c:(.text._sbrk_r+0x18): undefined reference to `_sbrk'
c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none-eabi/bin
/ld.exe: c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none
-eabi/lib\libc.a(lib_a-writer.o): in function `_write_r':
writer.c:(.text._write_r+0x28): undefined reference to `_write'
c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none-eabi/bin
/ld.exe: c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none
-eabi/lib\libc.a(lib_a-closer.o): in function `_close_r':
closer.c:(.text._close_r+0x18): undefined reference to `_close'
c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none-eabi/bin
/ld.exe: c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none
-eabi/lib\libc.a(lib_a_lseekr.o): in function `_lseek_r':
lseekr.c:(.text._lseek_r+0x28): undefined reference to `_lseek'
c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none-eabi/bin
/ld.exe: c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none
-eabi/lib\libc.a(lib_a-readr.o): in function `_read_r':
readr.c:(.text._read_r+0x28): undefined reference to `_read'
c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none-eabi/bin
/ld.exe: c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none
-eabi/lib\libc.a(lib_a-fstatr.o): in function `_fstat_r':
fstatr.c:(.text._fstat_r+0x20): undefined reference to `_fstat'
c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none-eabi/bin
/ld.exe: c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/..../lib/gcc/arm-none-eabi/9.2.1/..../..../arm-none
-eabi/lib\libc.a(lib_a-isatty_r.o): in function `_isatty_r':
isatty_r.c:(.text._isatty_r+0x18): undefined reference to `_isatty'
collect2.exe: error: ld returned 1 exit status
```



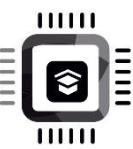
semihosting

- By default, the C library in Arm Compiler uses special functions to access the input and output interfaces on the host computer. These functions implement a feature called semihosting. Semihosting is useful when the input and output on the hardware is not available during the early stages of application development.
- When you want your application to use the input and output interfaces on the hardware, you must retarget the required semihosting functions in the C library.
- The semihosting feature enables your bare-metal application, running on an Arm processor, to use the input and output interface on a host computer. This feature requires the use of a debugger that supports semihosting, for example Arm DS-5 Debugger, on the host computer.
- A bare-metal application that uses semihosting does not use the input and output interface of the development platform. When the input and output interfaces on the development platform are available, you must reimplement the necessary semihosting functions to use the input and output interfaces on the development platform.
- For more information, see how to use the libraries in [semihosting](#) and [nonsemihosting](#) environments.



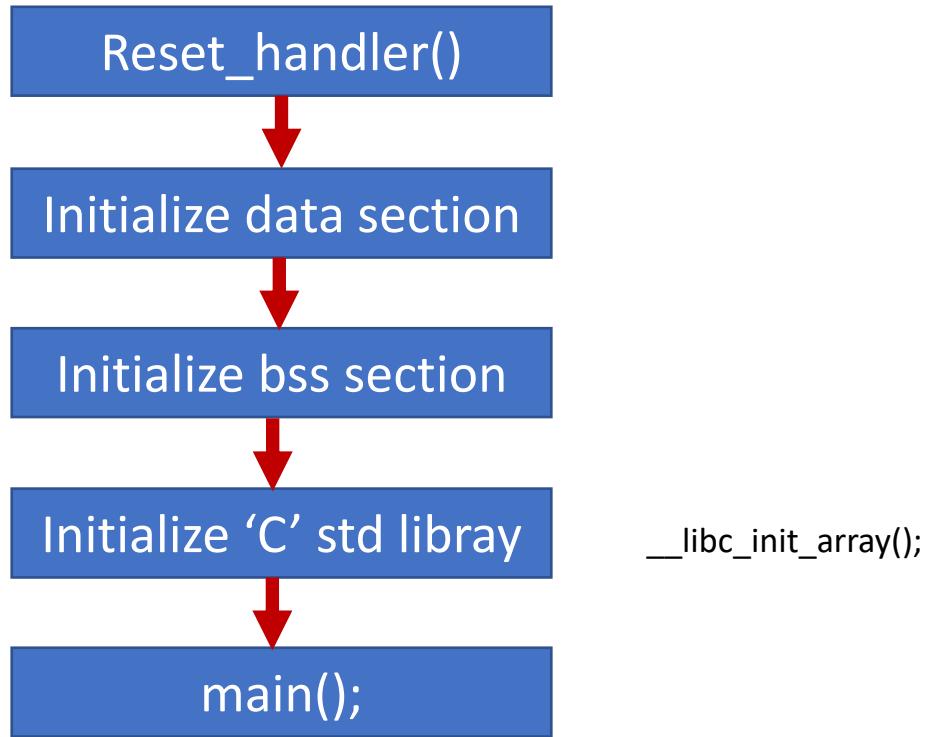
semihosting

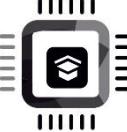
- If you are developing an application to run in a semihosted environment for debugging, you must have an execution environment that supports A32 or T32 semihosting trap instructions for AArch32 state or A64 semihosting trap instruction for AArch64 state.
- The Arm debug agents support semihosting
- Arm Compiler supports semihosting by generating trap instructions such as HLT, SVC, or BKPT depending on the architecture or profile. Debug agents can trap these instructions to perform semihosting operations on the host.
- <https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/semihosting-a-life-saver-during-soc-and-board-bring-up>



`__libc_init_array`

- `__libc_init_array` calls the constructors in the `.preinit_array` and `.init_array` sections. If `.init` is also present for an architecture, the constructors stored in that section will also be invoked.





Thank you

For more courses on embedded systems please visit
www.fastbitlab.com