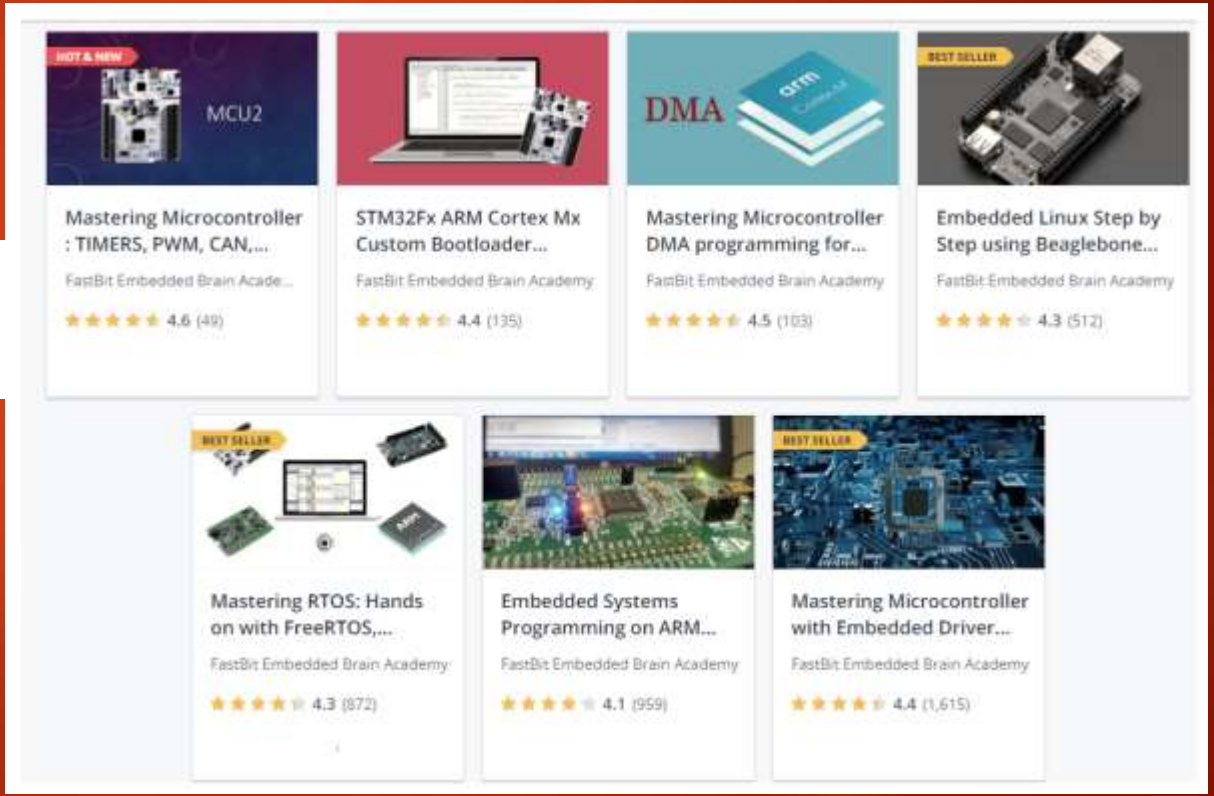
Embedded System Programming on ARM Cortex-m3/m4

For full video course on Microcontroller and RTOS programming please visit : www.fastbitlab.com

All courses are hosted on Udemy.com

Total students	Courses	Reviews
16,226	7	4,245

Udemy



Gift this course

Mastering Microcontroller DMA programming for Beginners

Direct Memory Access Demystified with STM32 Peripherals (ADC, SRAM,UART,M2M,M2P,P2M) and Embedded C code Exercises

4.5 (103 ratings) 1,240 students enrolled

Created by FastBit Embedded Brain Academy Last updated 10/2018

English English [Auto-generated]

Click here to enroll FREE !!!!

http://bit.ly/2P47IeX

Includes

- 9.5 hours on-demand video
- 8 articles
- Full lifetime access
- Access on mobile and TV
- Certificate of Completion

Gift this course

Mastering Microcontroller with Embedded Driver Development

Learn from Scratch Microcontroller & Peripheral Driver Development for STM32 GPIO,I2C,SPI,USART using Embedded C

BESTSELLER 4.3 (1,609 ratings) 9,101 students enrolled

Created by FastBit Embedded Brain Academy Last updated 10/2018

English English [Auto-generated], Portuguese [Auto-generated], 1 more

Click here to watch free preview and enroll

http://bit.ly/2QaW0M9

Includes

- 18 hours on-demand video
- 11 articles
- Full lifetime access
- Access on mobile and TV
- Certificate of Completion

Interactive Features

- 4 downloadable resources

Gift this course

Mastering Microcontroller : TIMERS, PWM, CAN, RTC,LOW POWER

learn STM32 TIMERS, CAN,RTC, PLL. LOW POWER modes work and program them using STM32 Device HAL APIs STEP by STEP

HOT & NEW ★★★★½ 4.5 (46 ratings) 463 students enrolled

Created by FastBit Embedded Brain Academy, Bharati Software Last updated 10/2018

English English [Auto-generated]

Click here to watch free preview and enroll

http://bit.ly/2SA2uFO

Includes

- 21.5 hours on-demand video
- 5 articles
- Full lifetime access
- Access on mobile and TV
- Certificate of Completion

Interactive Features

- 2 downloadable resources
- Assignments

Gift this course

Mastering RTOS: Hands on with FreeRTOS, Arduino and STM32Fx

Learn Running/Porting FreeRTOS Real Time Operating System on Arduino, STM32F4x and ARM cortex M based Mircocontrollers

BESTSELLER ★★★★★ 4.2 (866 ratings) 5,480 students enrolled

Created by FastBit Embedded Brain Academy Last updated 10/2018

English English [Auto-generated], Portuguese [Auto-generated], 1 more

Click here to watch free preview and enroll

http://bit.ly/2PAQRQh

Includes

- 16 hours on-demand video
- 28 articles
- Full lifetime access
- Access on mobile and TV
- Certificate of Completion

Interactive Features

- 17 downloadable resources

Gift this course

Embedded Linux Step by Step using Beaglebone Black

Learn ARM Linux systems, Embedded Linux building blocks , Beaglebone interfacing Projects and much more

BESTSELLER 4.3 (509 ratings) 3,084 students enrolled

Created by FastBit Embedded Brain Academy Last updated 10/2018

English English [Auto-generated]

Click here to watch free preview and enroll

http://bit.ly/2AEcneA

Includes

- 15.5 hours on-demand video
- 24 articles
- Full lifetime access
- Access on mobile and TV
- Certificate of Completion

Interactive Features

- 9 downloadable resources

Gift this course

STM32Fx ARM Cortex Mx Custom Bootloader Development

Learn fundamentals of Bootloader Development for your ARM Cortex Mx based STM32 Microcontroller

4.5 (135 ratings) 1,105 students enrolled

Created by FastBit Embedded Brain Academy Last updated 10/2018

English English [Auto-generated]

Click here to watch free preview and enroll

http://bit.ly/2zhqDrX

Includes

- 7.5 hours on-demand video
- 6 articles
- Full lifetime access
- Access on mobile and TV
- Certificate of Completion

Interactive Features

- 1 downloadable resource

Embedded Systems Programming on ARM Cortex-M3/M4 Processor

With hands on Coding using C Programming and assembly on ARM Cortex M Processor based Microcontroller

4.1 (958 ratings) 6,257 students enrolled

Created by FastBit Embedded Brain Academy Last updated 10/2018

English English [Auto-generated], Spanish [Auto-generated]

Click here to watch free preview and enroll

http://bit.ly/2OjgFf7

Includes

- 11.5 hours on-demand video
- 12 articles
- Full lifetime access
- Access on mobile and TV
- Certificate of Completion

Interactive Features

- 2 downloadable resources

Overview

Motivation to learn ARM-Cortex-M3/M4

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

Cortex
M3/M4

Micro-Controller

freescale semiconductor

FUJITSU

NXP

TEXAS INSTRUMENTS

Many More . . . .

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

Cortex M3/M4 Processor Architecture

KIRAN NAYAK | SECTION-1

Cortex-M3/M4 in S\W developers point of view

What the firmware/Embedded Developers should be knowing ???

- ✓ Programming model
- ✓ How exceptions are handled
- ✓ The memory map
- ✓ Peripheral interfacing
- ✓ How to use the S/W driver libraries from Microcontroller vendors

Programmer's Model: Operational Modes & Access Levels

Operational model

How does Operational model of the processor look like ?

Operational modes

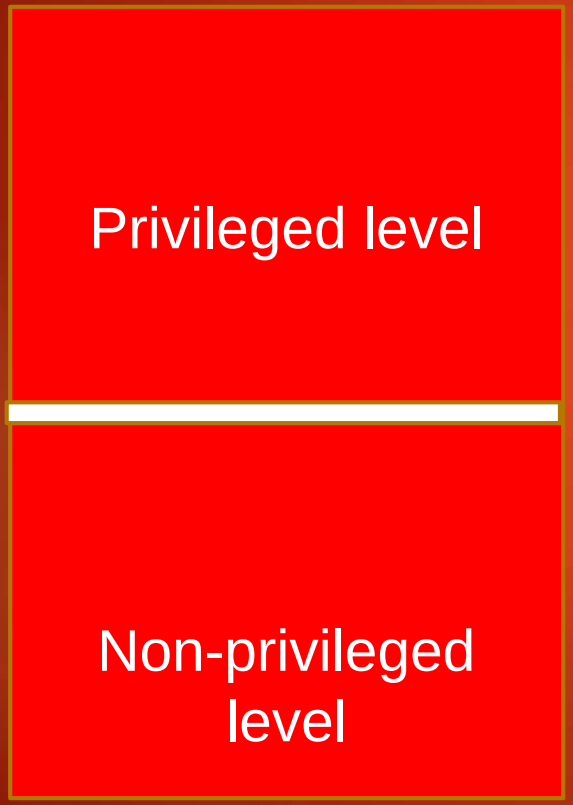
Thread mode

- ✓ Processor executes the normal application code
- ✓ Privileged level or un-privileged level
- ✓ On reset processor enters to thread mode

Handler Mode

- ✓ Enters during System exceptions /interrupts
- ✓ Code executing is privileged
- ✓ Only way user can put the processor to handler mode is to raise an system exception or interrupt

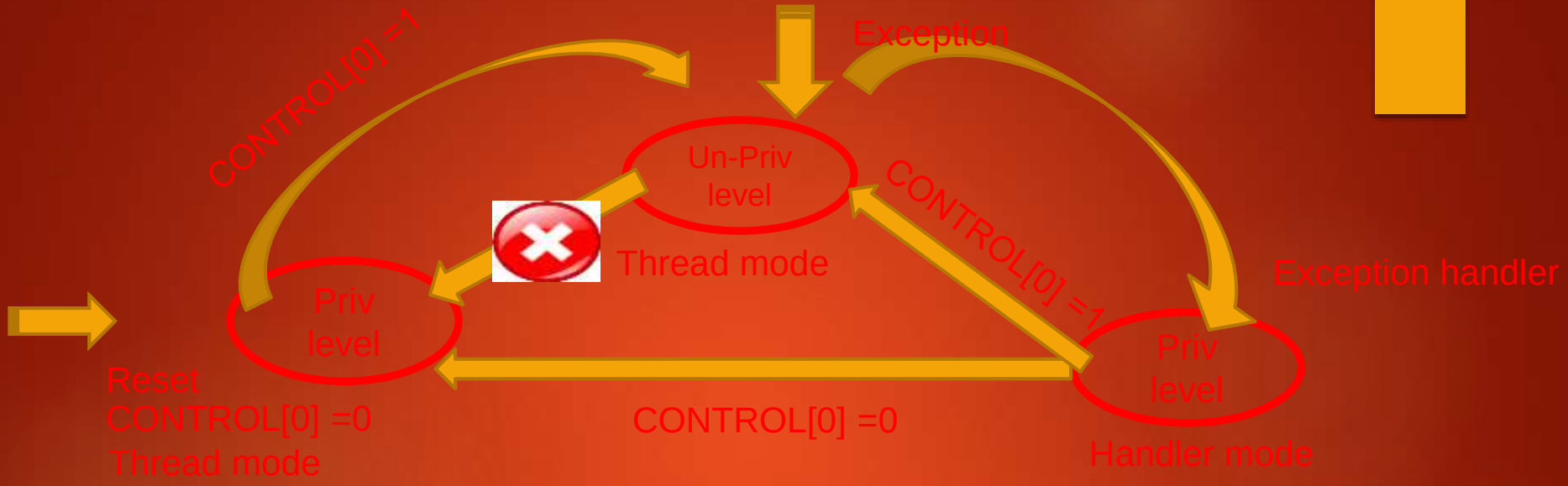
Access Levels



- ✓ Full access to processor resources
- ✓ Handler mode is always privileged
- ✓ Thread mode is by default privileged , but can be changed to un-privileged

- ✓ Restricted access .
- ✓ You can make thread mode to run in this level .

Switching between Privileged and Un-privileged Access Level



1. First Processor start with thread mode and privileged access level
2. Changing CONTROL[0]=1 move the processor into un-privileged mode
3. Its not possible to come back to priv level at this stage
4. Issue a processor exception
5. Processor executes exception handler in handler mode with privileged access level
6. making CONTROL[0]=0 makes return to privileged level
7. Otherwise processor return to un-privileged level

Application of switching between Access levels:

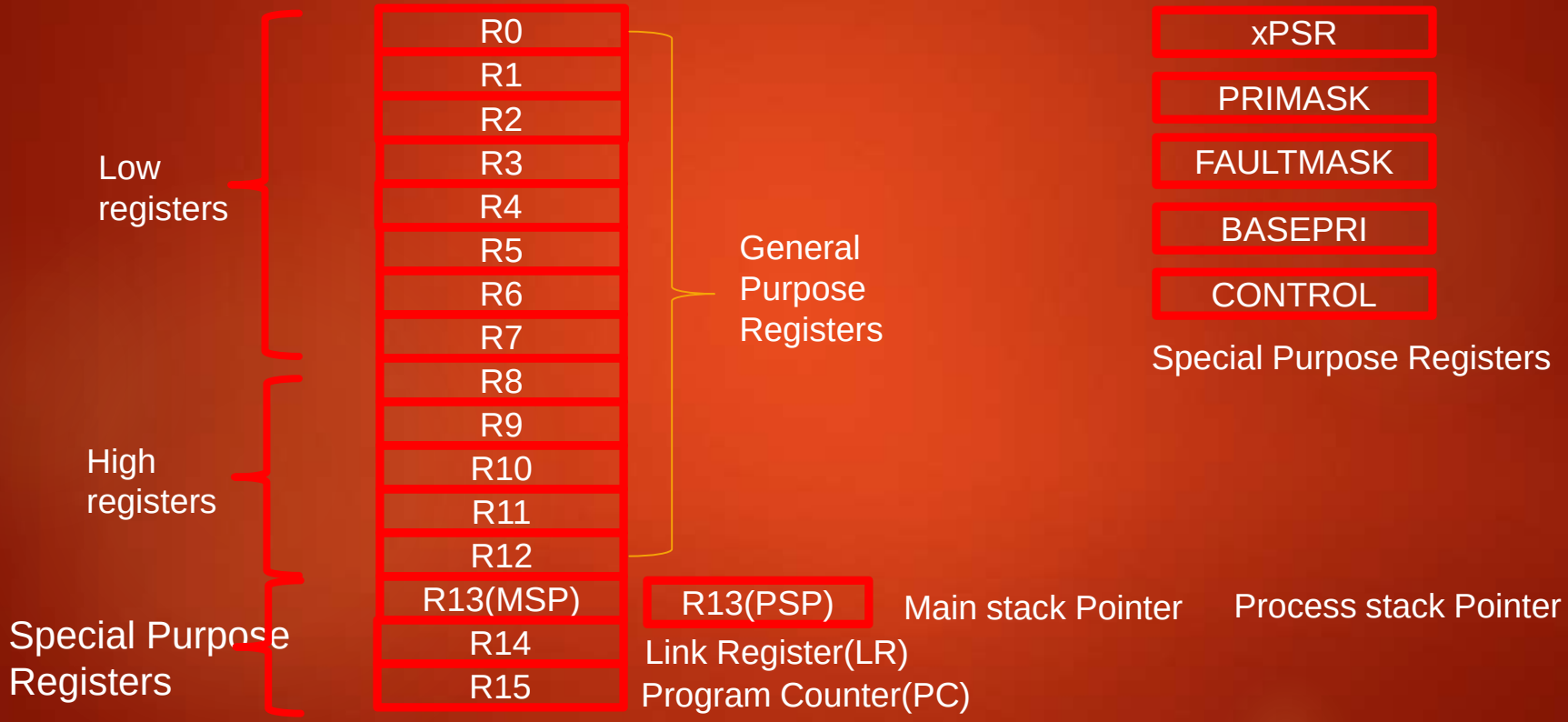
In Embedded Operating system Design

The kernel of a RTOS or embedded OS can change the control register to make an application task to run in unprivileged mode when it gets scheduled to run.

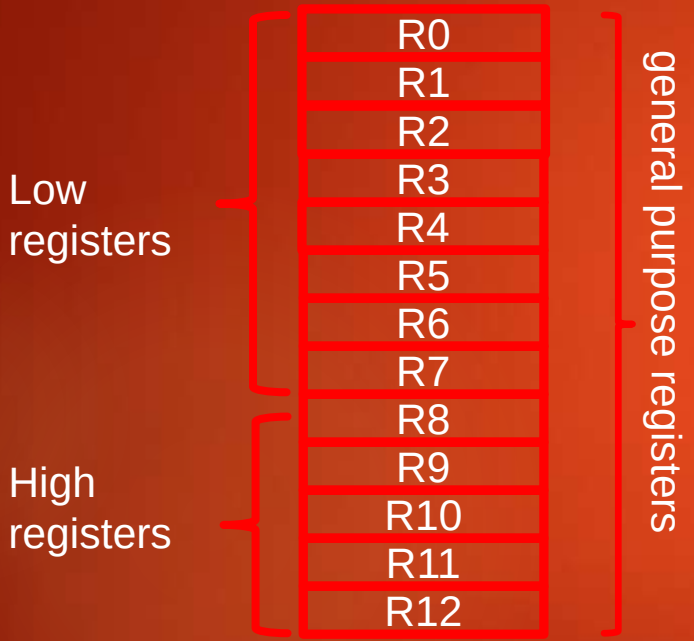
It helps to design more secured and robust applications

Programmer's Model: Register Model

Register model



General purpose Registers



- ✓ Registers R0-R12 are general purpose registers
- ✓ R0-R7 are called low registers because, many 16 bit instructions can only access low registers.
- ✓ All 32 bit instructions and very few 16 bit instructions can access high registers i.e R8-R12
- ✓ The initial values of R0 to R12 are undefined.

R13 stack pointer(SP)

Main Stack Pointer

R13(MSP)

Process Stack Pointer

R13(PSP)

- ✓ Physical there are 2 stack pointer exist. The MSP and PSP.
- ✓ MSP is the default stack pointer.
- ✓ R13 just points to the value of the currently selected stack pointer .
- ✓ You can even change the stack pointer to PSP, by writing to CONTROL register of the processor
- ✓ When the processor is executing exception handler, it always uses the MSP .

R14,Link register(LR)

Link Register(LR)

R15,Program Counter(PC)

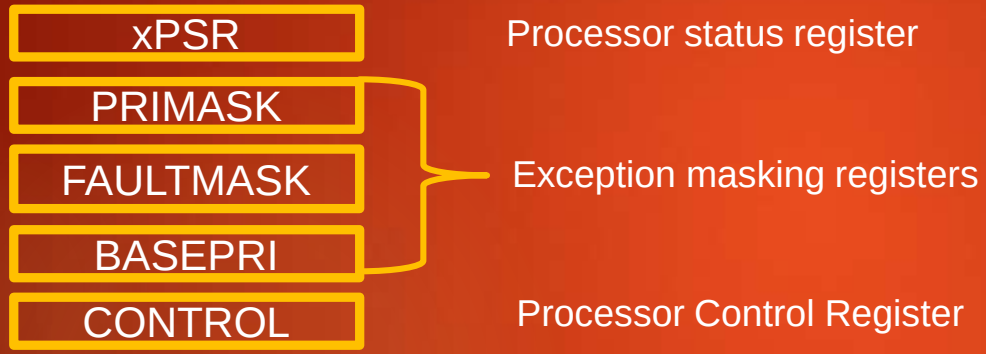
R15

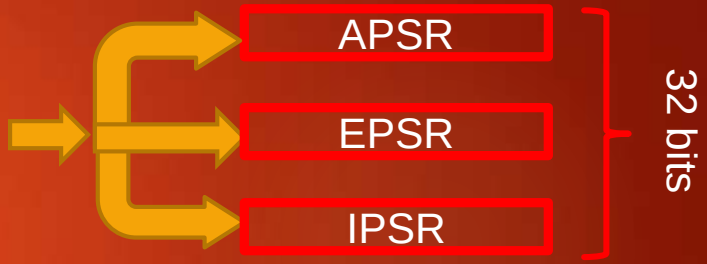
Program Counter(PC)

- ✓ Always points to next instruction to be executed
- ✓ Always increments by 4 (word size)
- ✓ Points to word aligned address in the code memory .

Programmer's Model: Special Purpose Registers

Special Purpose Registers

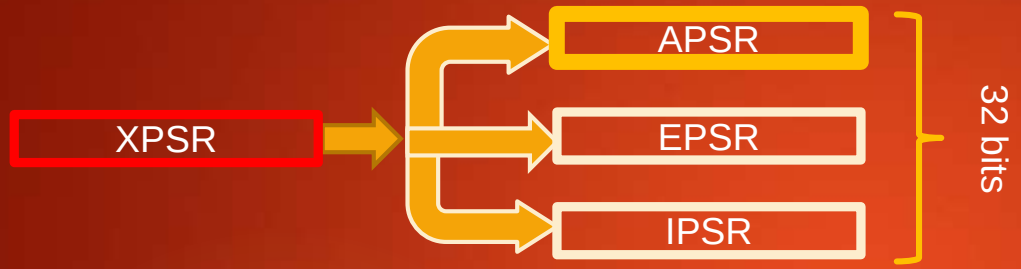


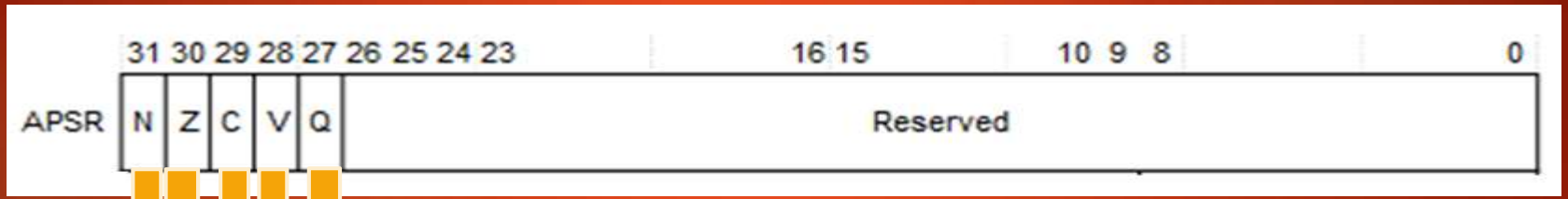


Quiz -1

So , How do you access General Purpose/Special Purpose registers in a ‘C’ program when they are not memory mapped ??

APSR





Saturation Flag

Overflow Flag

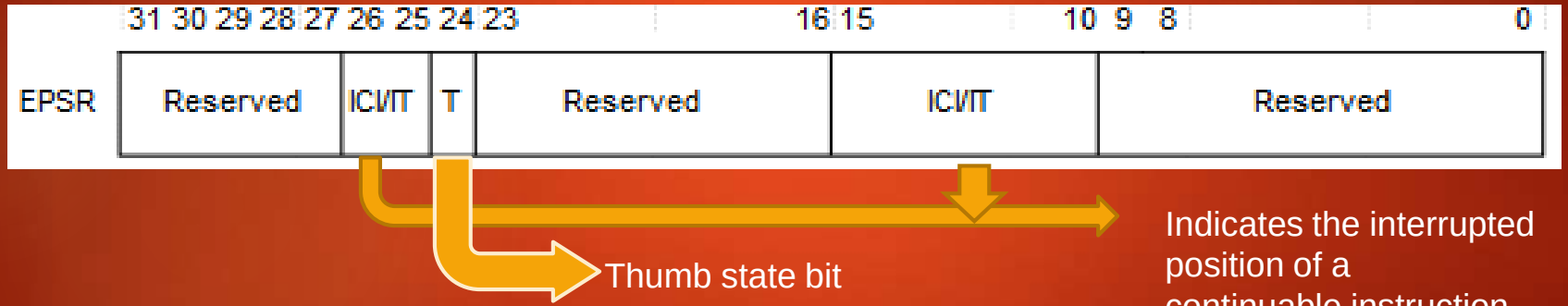
Carry/Borrow Flag

Zero Flag

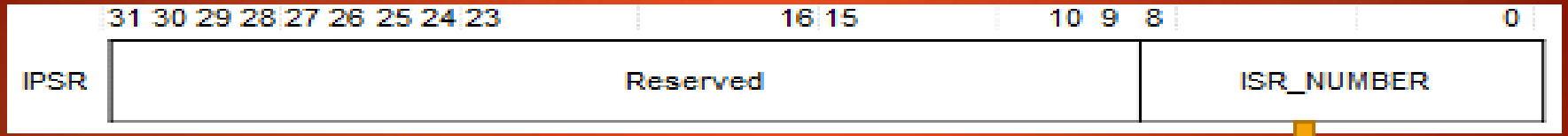
Negative Flag

EPSR


XPSR
APSR
EPSR
IPSR
32 bits


31 30 29 28 27 26 25 24 23
16 15
10 9 8
0
EPSR
Reserved
ICI/IT
T
Reserved
ICI/IT
Reserved
Thumb state bit
Indicates the interrupted position of a continuable instruction

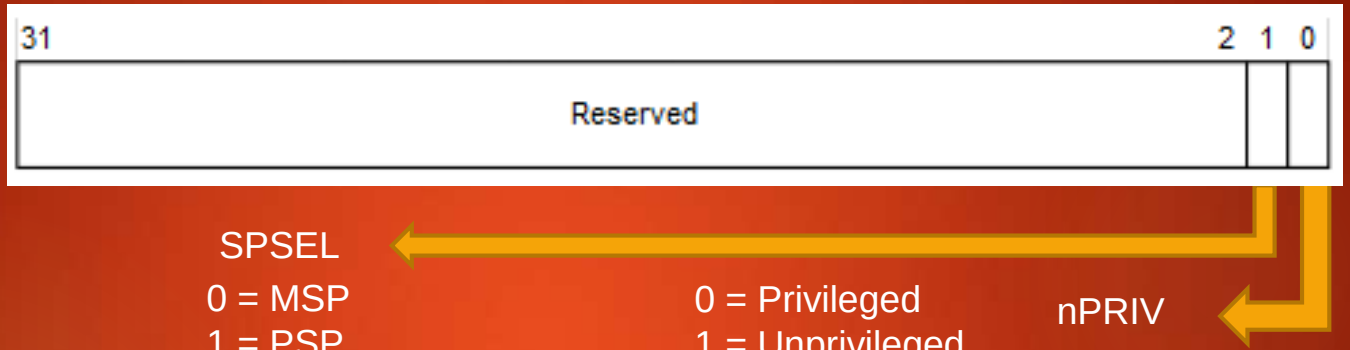
IPSR



This is the number of the current exception
0 = Thread mode
1 = Reserved
2 = NMI
3 = HardFault
.
.
16 = IRQ0
n+15 = IRQ(n-1)

Exception masking registers

CONTROL Register

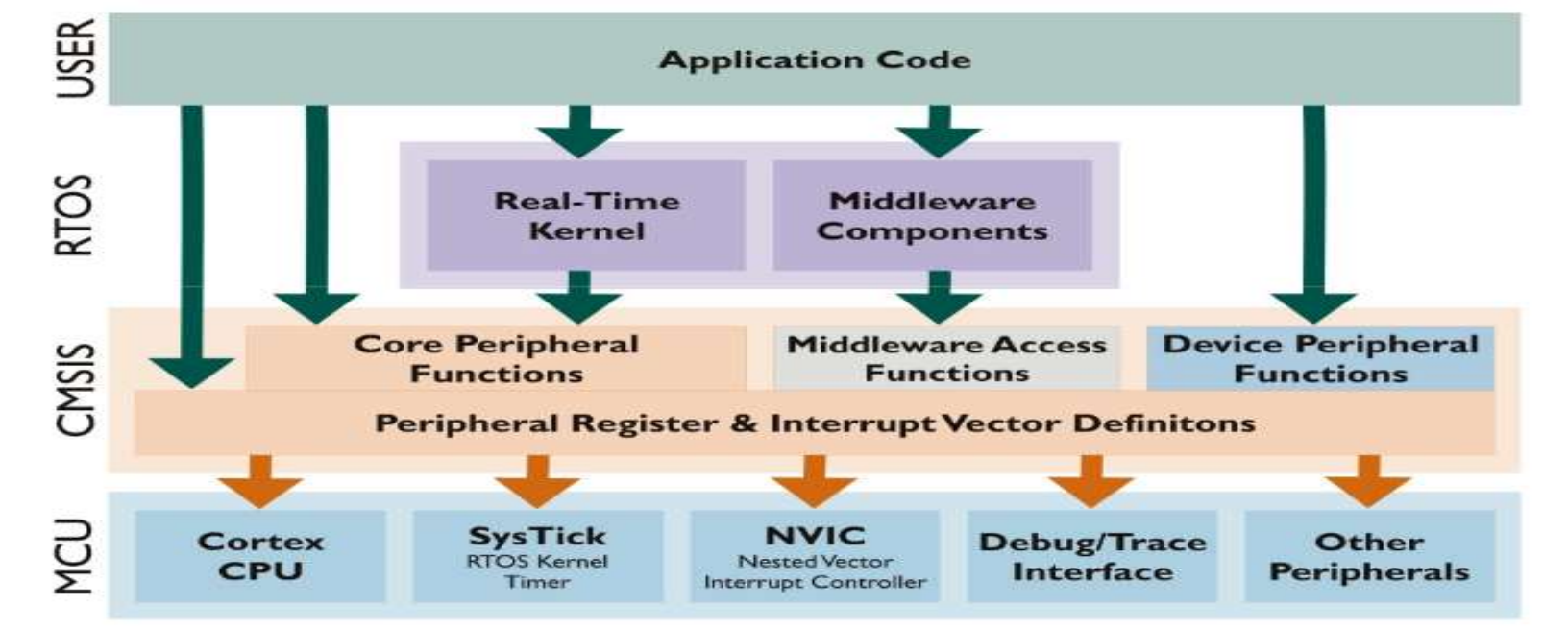


CONTROL Register

- After reset, the CONTORL register is 0.
- This means , by default, Thread mode uses MSP as the stack pointer and code runs in the privileged access level.
- Programs in privileged Thread mode can switch the stack pointer selection or switch to unprivileged access level by writing to CONTROL.
- Once “nPriv” is set to 1, the program running in thread mode can no longer access the CONTROL register.

Cortex Microcontroller Software Interface Standard (CMSIS)

CMSIS


USER
Application Code
RTOS
Real-Time Kernel
Middleware Components
CMSIS
Core Peripheral Functions
Middleware Access Functions
Device Peripheral Functions
Peripheral Register & Interrupt Vector Definitons
MCU
Cortex CPU
SysTick
RTOS Kernel Timer
NVIC
Nested Vector Interrupt Controller
Debug/Trace Interface
Other Peripherals

Why to use CMSIS ?

- CMSIS is a *Hardware Abstraction Layer* for Cortex-M series microcontrollers.
- It supports developers and vendors in creating *reusable software components* for ARM Cortex-M based systems.
- If a library is CMSIS compliant, it's *vendor-independent,* and its easier to swap different families like Cortex M0 to Cortex M3.

Why to use CMSIS ? Contd.

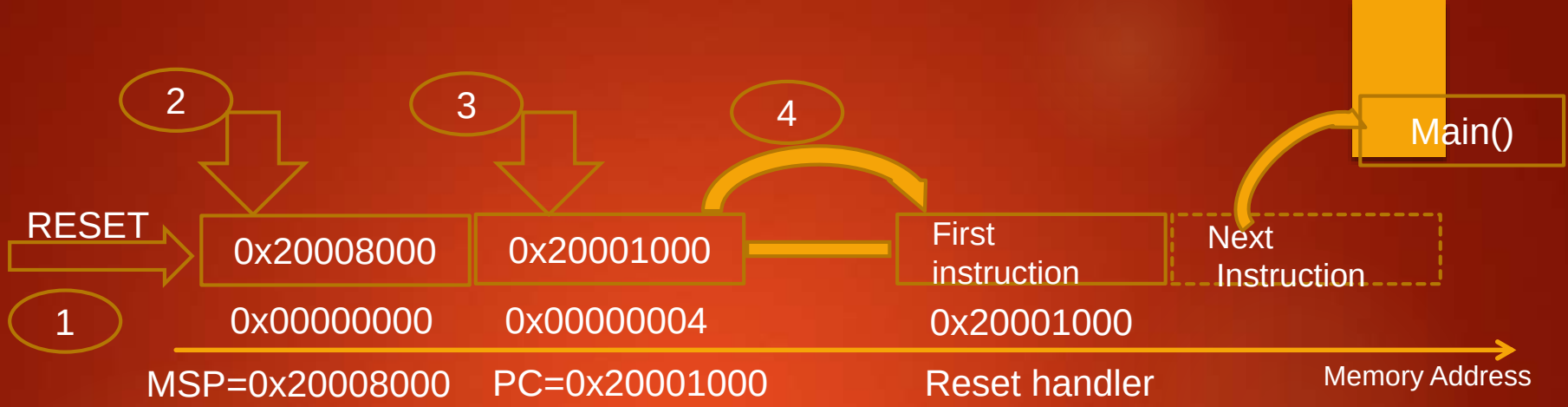
- One of the attractive aspects of the ARM Cortex environment with CMSIS is the ability to *change platforms without a whole bunch of sweat.*

- If you pick a platform that doesn't buy into the CMSIS structure, you may not be able to move around as conveniently.

Why to use CMSIS ? Contd.

- ✓ Simplifying software re-use and software portability . i.e you will quickly able to port code written for Cortex M3 to Cortex M4 when you want to migrate.
- ✓ Reducing the learning curve for microcontroller developers
- ✓ Reducing the time to market for new devices

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.

Switching between Privileged and Un-Privileged Access level of Execution

Points To Remember !!

- ✓ Processor always starts with Privileged Access level
- ✓ In privileged level you can access any Restricted Register
- ✓ From Privileged Level you can go to Un-Privileged access level
- ✓ In unprivileged access level you can not access Restricted Registers
- ✓ From Un-Privileged access level you can not come back to Privileged access level
- ✓ You can only change the access level to privileged ,from an exception/interrupt handler

Lets Code and Understand !!

Memory System Architecture

KIRAN NAYAK| SECTION-4

Session Overview

At the end of the session you will be able to understand

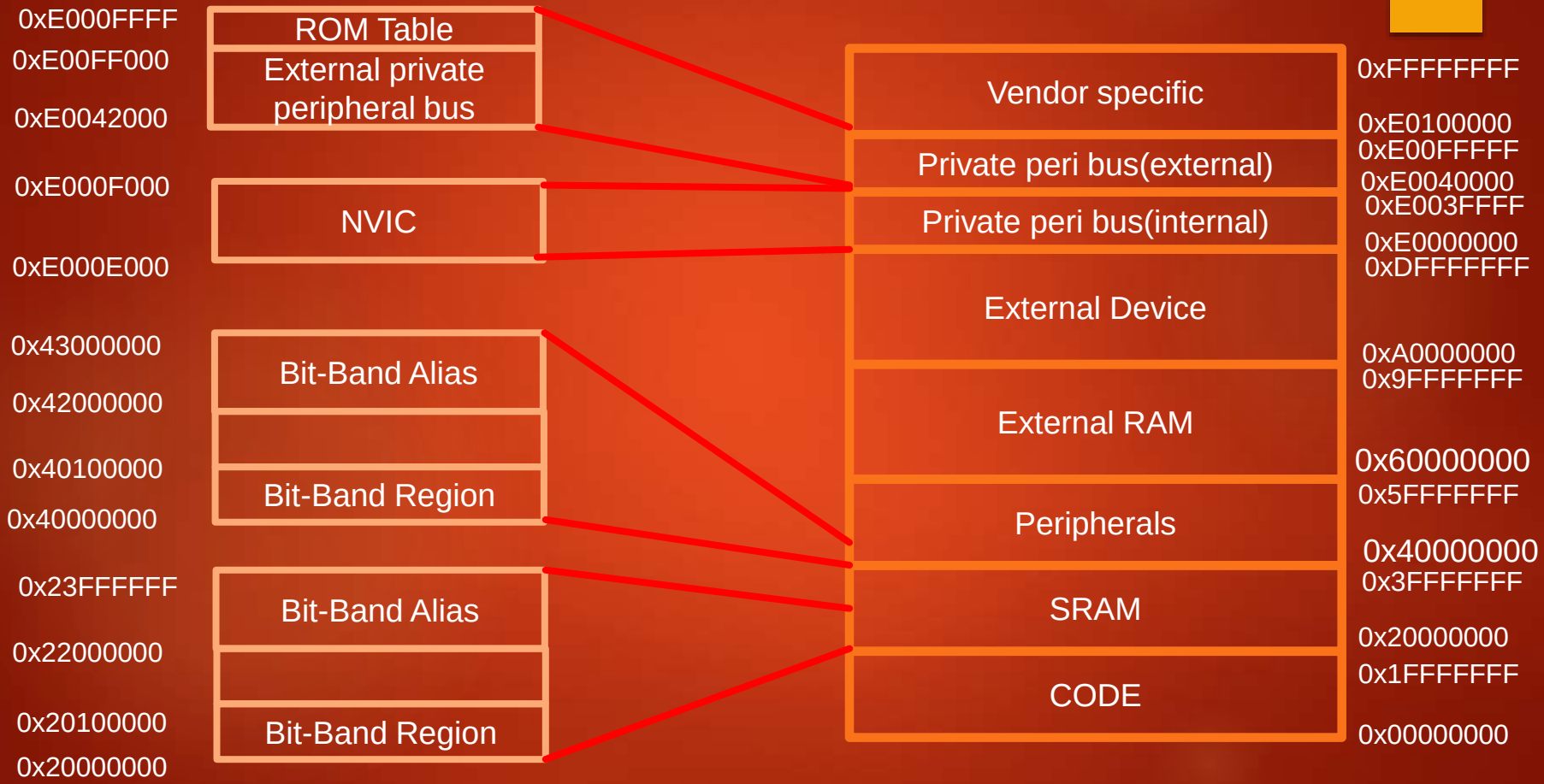
- ✓ Memory system features
- ✓ Memory map of the processor
- ✓ Bus interfaces
- ✓ Bus protocols
- ✓ Aligned and un-aligned data transfer
- ✓ Bit banding and its advantages
- ✓ Code example

Memory System Features

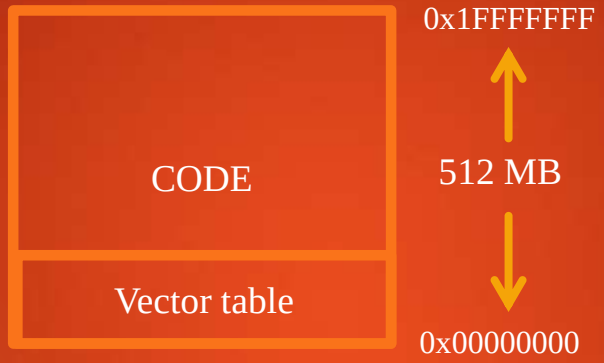
Features

- ✓ All the Cortex-M processors have 32 bit memory addressing, as a result there is 4GB of addressable memory space
- ✓ The memory is one unified space which is shared by code space, data space and peripheral space.
- ✓ Harvard bus architecture . It means concurrent instruction and data accesses using multiple bus interfaces.
- ✓ Support both little endian and big endian memory systems.
- ✓ Support for unaligned data transfers.
- ✓ Bit addressable memory spaces(bit-banding)
- ✓ MPU support

Memory map

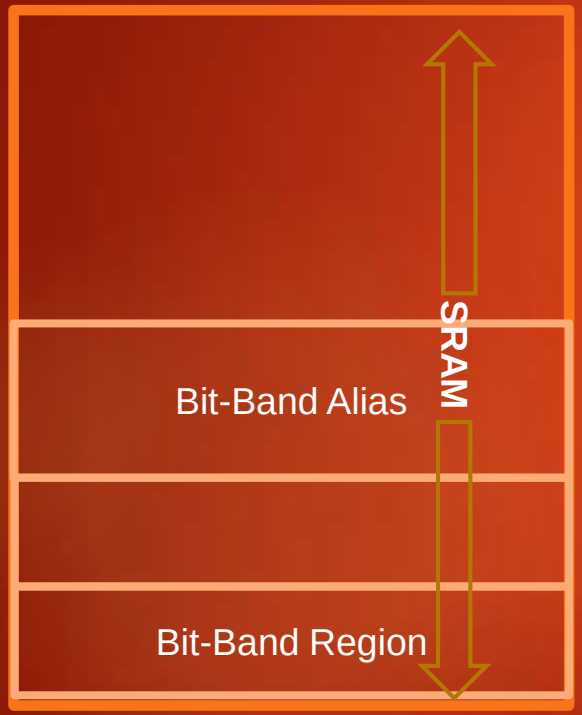

0xE000FFFF
ROM Table
0xE00FF000
External private peripheral bus
0xE0042000
0xE000F000
NVIC
0xE000E000
0x43000000
Bit-Band Alias
0x42000000
0x40100000
Bit-Band Region
0x40000000
0x23FFFFFF
Bit-Band Alias
0x22000000
0x20100000
Bit-Band Region
0x20000000
Vendor specific
Private peri bus(external)
Private peri bus(internal)
External Device
External RAM
Peripherals
SRAM
CODE
0xFFFFFFFF
0xE0100000
0xE00FFFFF
0xE0040000
0xE003FFFF
0xE0000000
0xDFFFFFFF
0xA0000000
0x9FFFFFFF
0x60000000
0x5FFFFFFF
0x40000000
0x3FFFFFFF
0x20000000
0x1FFFFFFF
0x00000000

CODE Region



Range : 0x00000000–0x1FFFFFFF

SRAM Region



0x3FFFFFFF

0x23FFFFFF

0x22000000
0x21FFFFFF

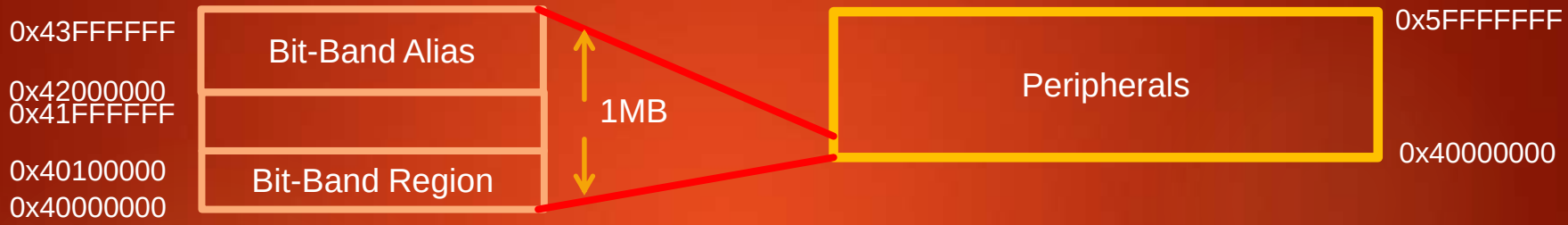
0x20100000

0x20000000

Range : 0x20000000–0x3FFFFFFF

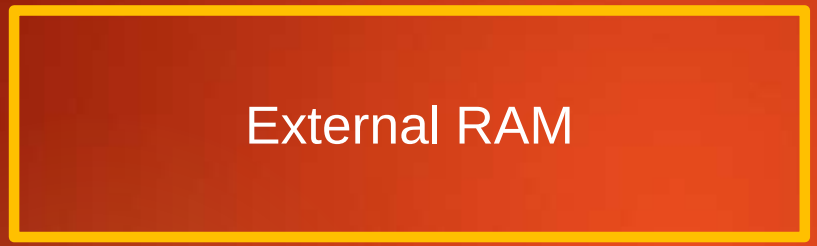
- ✓ The SRAM(Static-RAM) region is located in the next 512 MB of memory space after CODE region
- ✓ It is primarily for connecting SRAM, mostly on chip SRAM.
- ✓ The first 1 MB of the SRAM region is bit addressable.
- ✓ You can also execute program code from this region

Peripherals Region



Address Range : 0x40000000–0x5FFFFFFF

External RAM Region



0x9FFFFFFF

0x60000000

Address range : 0x60000000 to 0x9FFFFFFF

- ✓ This region is intended for either on-chip or off-chip memory
- ✓ you can execute code in this region.

External Device Region

Range : 0xA0000000 to 0xDFFFFFFF

- ✓ This region is intended for external devices and/or shared memory
- ✓ It is a non-executable region.

Bus Protocols & Bus Interfaces

BUS Protocols

- AHB Lite (Main System Bus)
- APB(Peripheral bus)

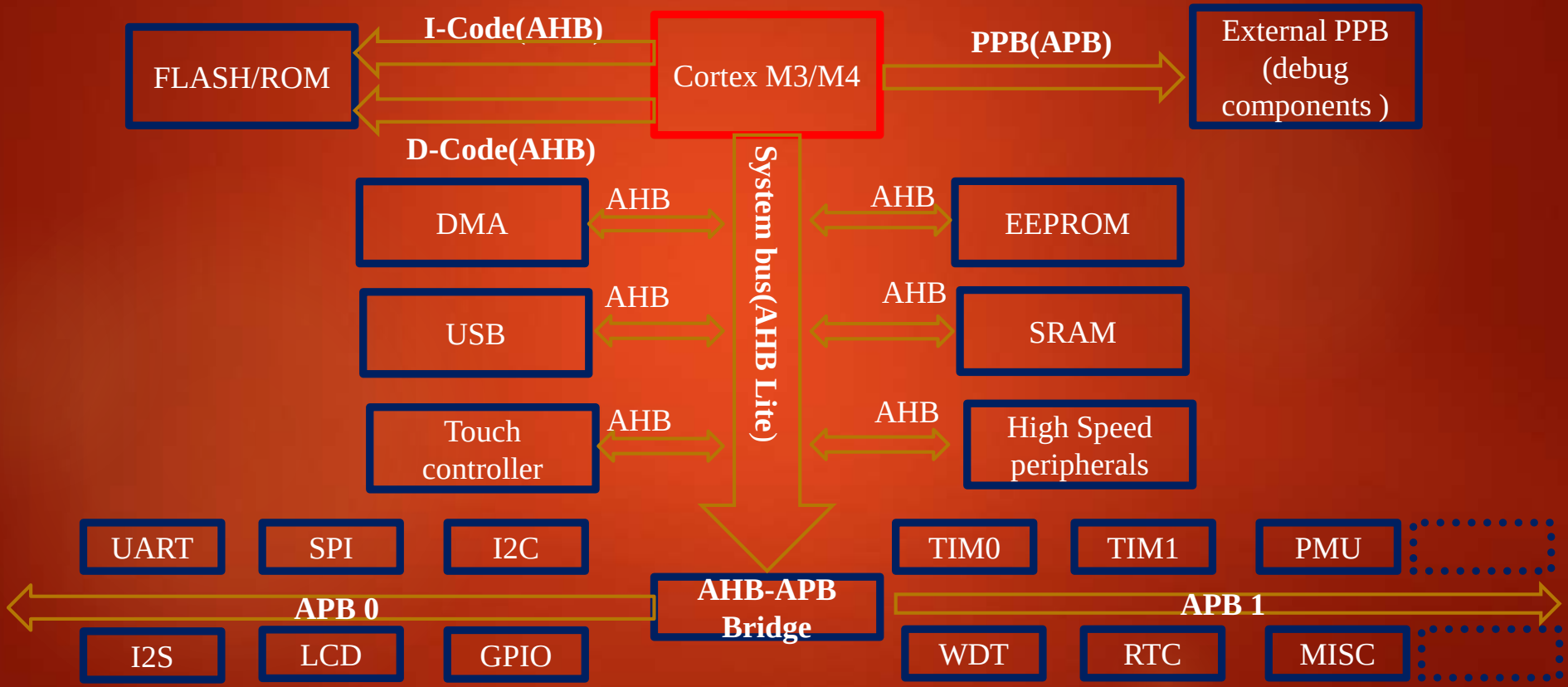
BUS Protocols Contd.

- AHB Lite (Main System Bus)
  - In The cortex m3 and m4 processors , the AHB lite protocol used for the main bus interfaces
  - AHB lite stands for **A**MBA **H**igh Performance **B**us which is derived form *AMBA(Advanced Microcontroller Bus Architecture)* specification

BUS Protocols Contd.

- APB(Peripheral bus)
  - The APB is an AMBA-compliant bus optimized for minimum power and reduced interface complexity
  - The AHB-APB bridge is an AHB slave that provides an interface between the high- speed AHB domain and the low-power APB domain.

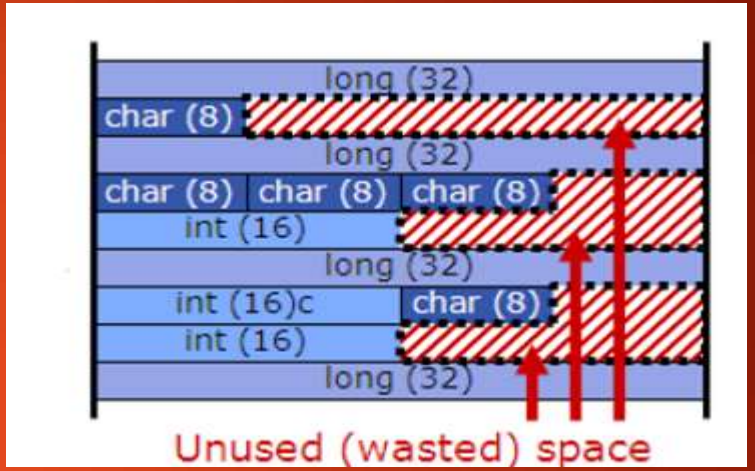
BUS interfaces



Aligned and Un-aligned data transfer

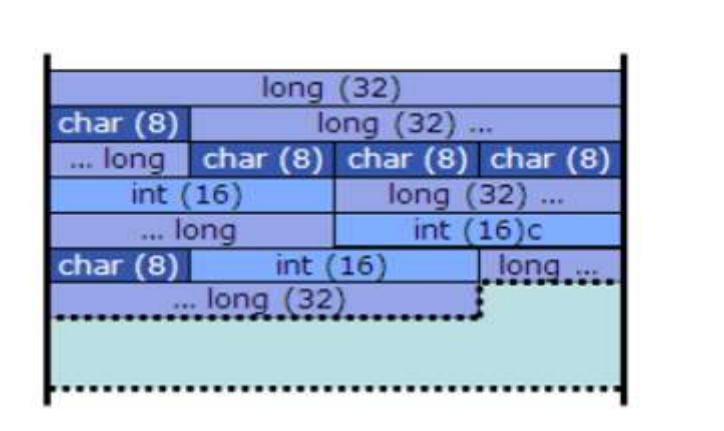
Aligned Data Transfer

```
Struct mydata
 {
      Unsigned long data1;
      Char                 data2
      Unsigned long   data3
      Char                 array[3]
      Short                data4;
      Unsigned long     data5
}
```



What Is Un-Aligned Data Transfer ?

```
Struct  __packed  mydata
      {
          Unsigned long data1;
          Char                 data2
          Unsigned long   data3
          Char                 array[3]
          Short Int            data4;
          Unsigned long     data5
      }
```



How Unaligned Data Access Can Result ?

- ✓ Direct manipulation of pointers
- ✓ Accessing data structure with __**packed** attributes that result in unaligned data
- ✓ Inline Assembly code

Instruction alignment in memory

```
0x080002DC 4812       LDR             r0,[pc,#72]

0x080002DE 6800       LDR             r0,[r0,#0x00]

0x080002E0 F0400001   ORR             r0,r0,#0x01

0x080002E4 4910       LDR             r1,[pc,#64]

0x080002E6 6008       STR             r0,[r1,#0x00]
```

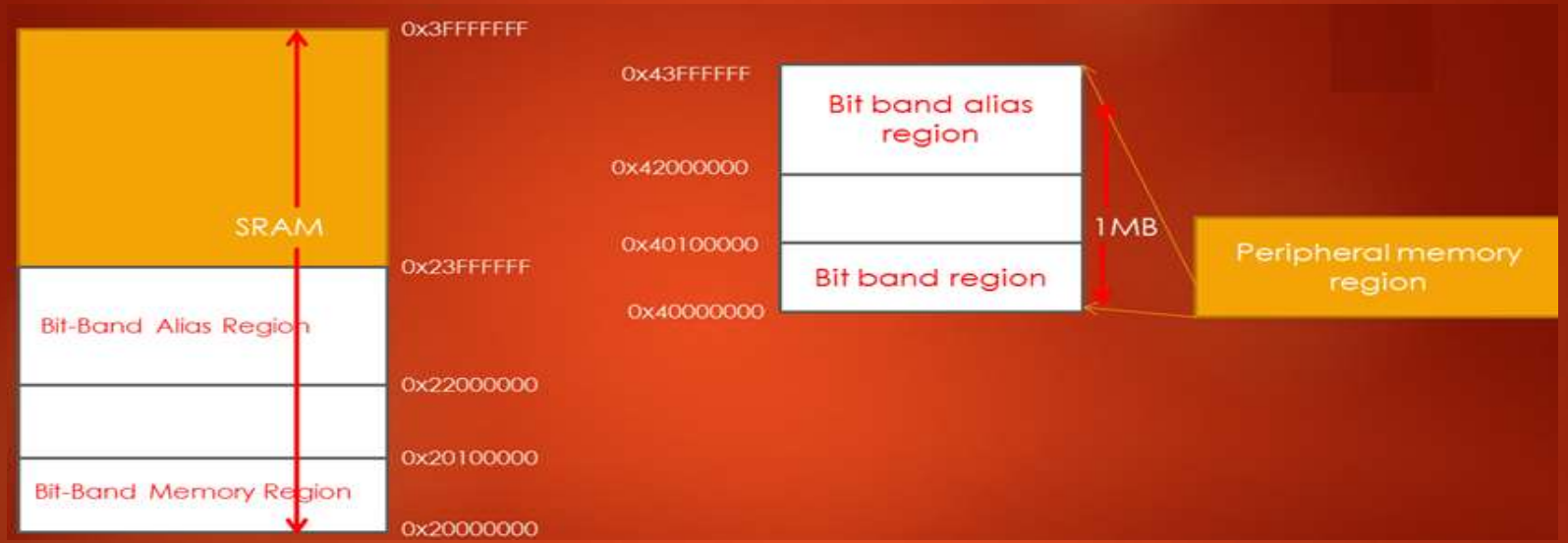
BIT-BANDING

What is bit-banding ?

- ✓ It is a capability to address a single bit of a memory address
- ✓ This feature is optional

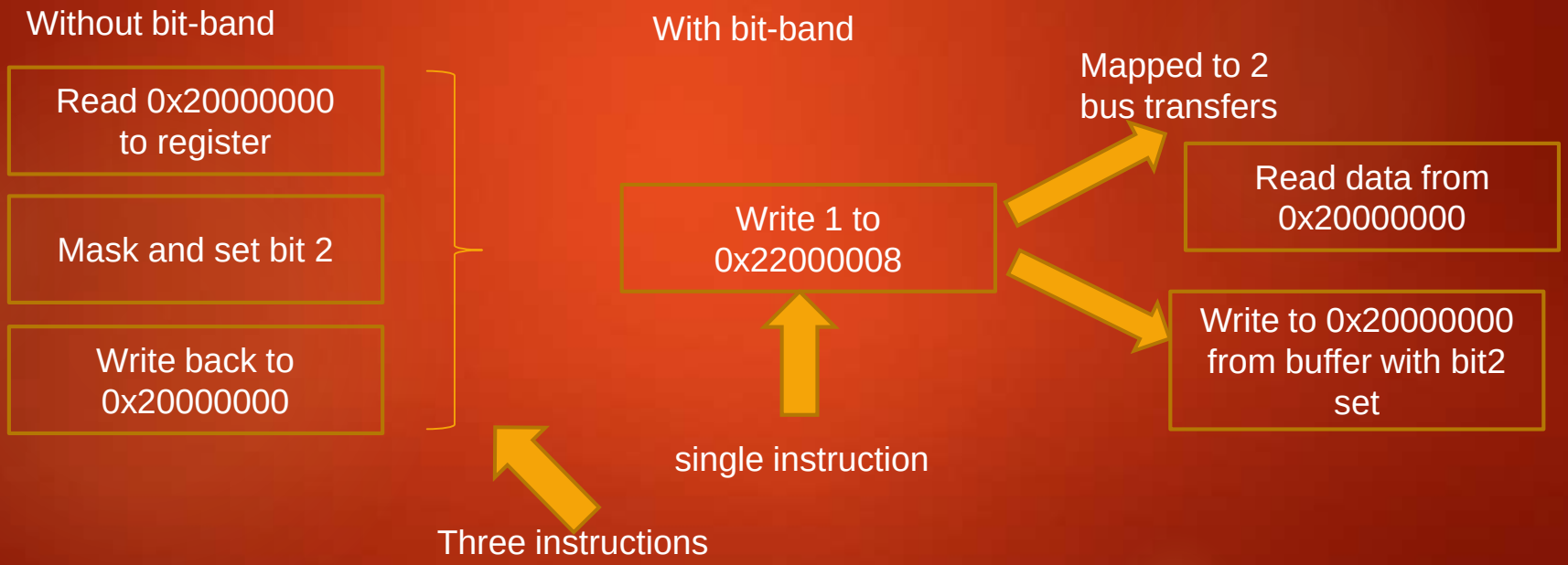

0x3FFFFFFF
SRAM
0x23FFFFFF
Bit-Band Alias Region
0x22000000
0x20100000
Bit-Band Memory Region
0x20000000
0x43FFFFFF
Bit band alias region
0x42000000
0x40100000
Bit band region
0x40000000
1MB
Peripheral memory region

Bit-Band memory regions

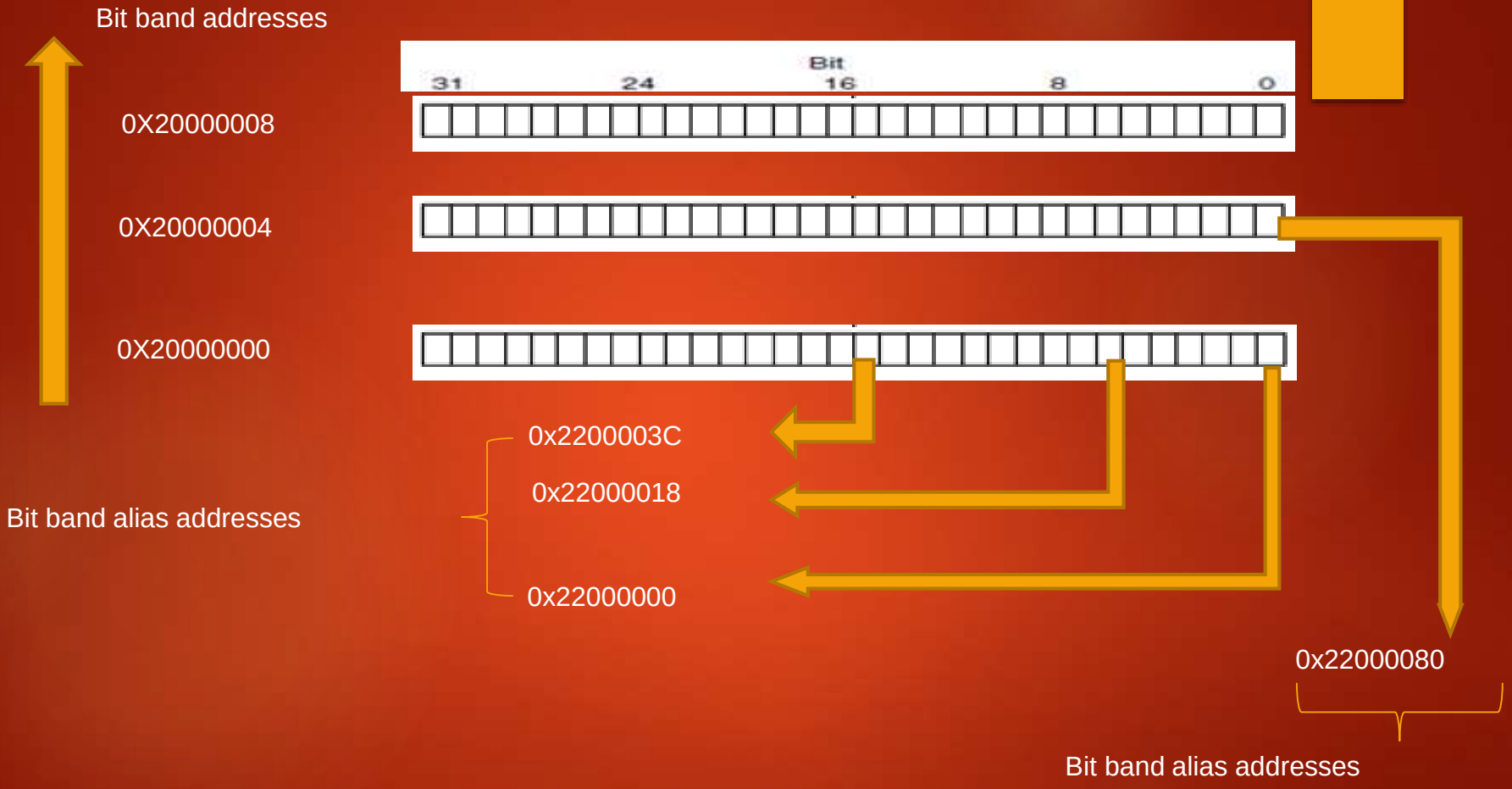


Example

Lets assume we want to set the value of the 2nd bit in the address 0x20000000



How this works ??


Bit band addresses
Bit
31
24
16
8
0
0X20000008
0X20000004
0X20000000
0x2200003C
0x22000018
0x22000000
Bit band alias addresses
0x22000080
Bit band alias addresses

Quiz -2

What is the value of bit-band alias address to address the 6th bit of the address 0x20000004 ??

0x20000004 bit[6]  Alias address = ??

Advantages of bit band regions

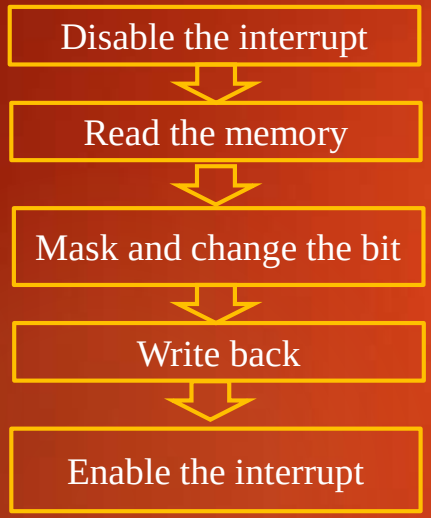
Instead of doing this

1. reading the whole register
2. Mask and check bit
3. Compare and branch

Simply do this

1. Read bit using bit-band alias address
2. Compare and branch.

Instead of doing this



Simply do this

Change the value of memory region using its alias address

Demonstrating Bit-Band Operations

Points To Remember !!

- Bit band Operations can only be operated on 2 memory regions. This is called bit-band region
  - 0x20000000 to 0x20100000 (1 MB)
  - 0x40000000 to 0x40100000(1 MB)

- To Set/Reset any bit field in the Bit band region , we have to use Bit band alias address in the Bit band alias region
  - 0x22000000 to 0x23ffffff (31 MB)
  - 0x42000000 to 0x43ffffff(31 MB)

Memory Remapping

Bit-Band Region		Alias Equivalent
0x20000000 bit[0]	→	0x22000000 bit[0]
0x20000000 bit[1]	→	0x22000004 bit[0]
0x20000000 bit[2]	→	0x22000008 bit[0]
…..		…..
0x20000000 bit[31]	→	0x2200007C bit[0]
0x20000004 bit[0]	→	0x22000080 bit[0]
0x20000004 bit[1]	→	0x22000084 bit[0]
…..		…..

In this Program we will change the value of **BIT0, BIT1 and BIT2** of the data stored in the memory address **0x20000000** using bit-band operations

Lets Code and Understand

Congratulations !! ☺

You Have Learnt

- How to Carry-out Bit-Band Operations in C

Stacks

KIRAN NAYAK| SECTION-6

Stack memory

- Stack is a kind of memory usage mechanism that allows a portion of memory , typically RAM to be used as Last In First Out(LIFO) data storage buffer.
- ARM Processors have the PUSH instruction to store data in stack and the POP instruction to retrieve data from stack.
- ALL stack operations are word aligned

Why stack is used ?

- ✓ Temporary storage of original data when a function being executed needs to use registers for data processing.
- ✓ Passing of information to functions or subroutines
- ✓ For storing local variables
- ✓ To hold processor status and register values in the case of exception such as an interrupt

Stack pointers

Physically there are 2 stack pointers in the Cortex-M3/M4 processors.

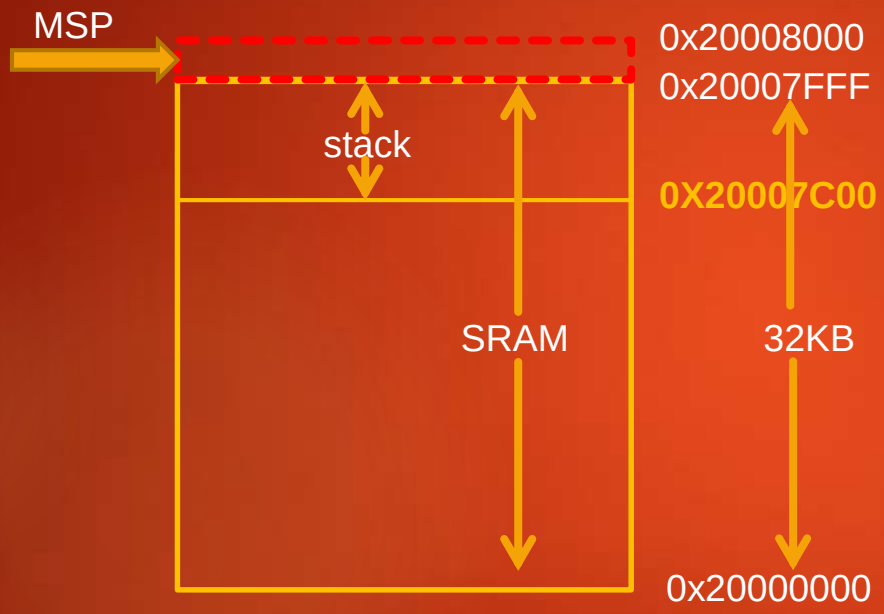
1. **Main stack Pointer(MSP):**
   - ✓ This is the default stack pointer used after reset.
   - ✓ used for all exception handlers.
   - ✓ after power up, the processor hardware automatically **initializes the MSP by reading the vector table.**
2. **Process Stack Pointer(PSP)**
   - ✓ This is an alternate stack point that can only be used in Thread mode
   - ✓ The PSP is not initialized after power up and must be initialized by the software before being used.

Stack pointers

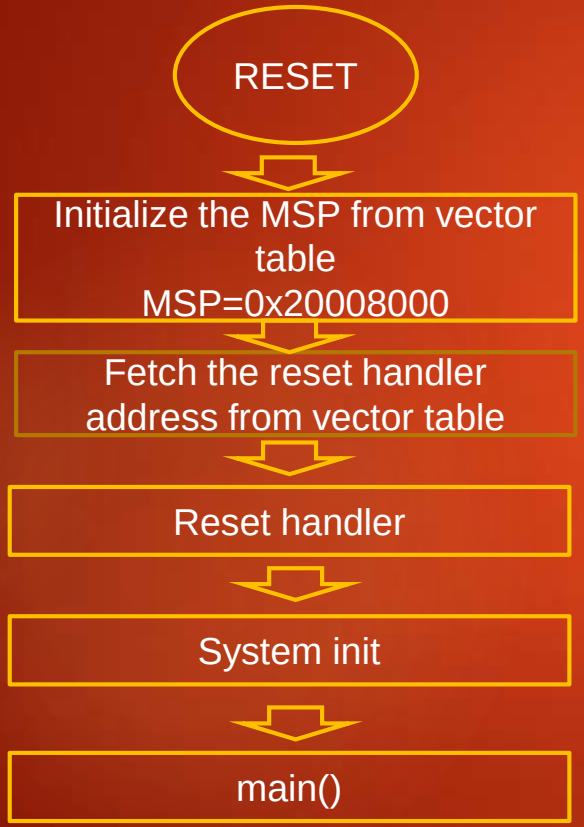
As mentioned Previously the selection between MSP and PSP can be controlled by the value of SPSEL bit in the CONTROL register.

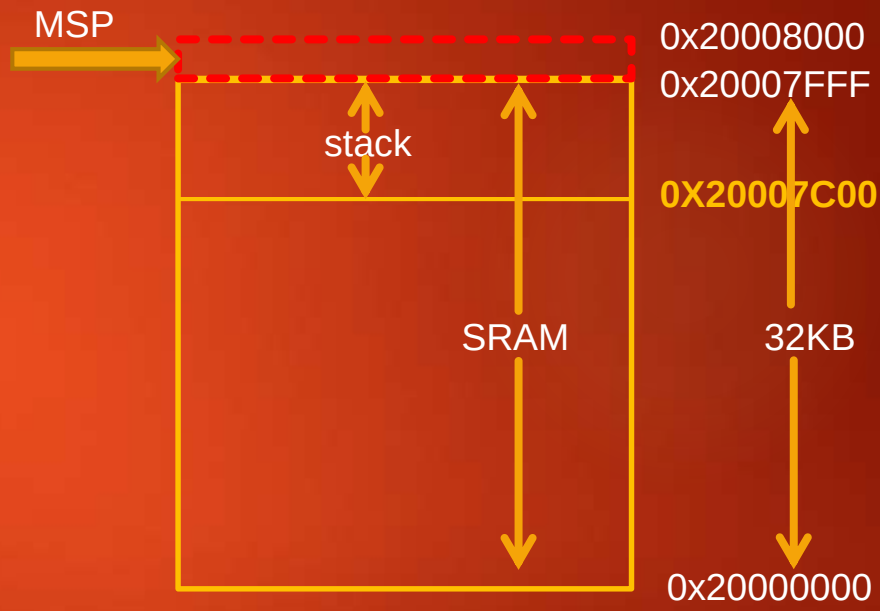
Stack Memory Model

STACK MEMORY MODEL



Initial Stack Init





Subroutine and Stack

SUBROUTINE AND STACK

As per the Procedure Call Standard Of Arm Architecture.

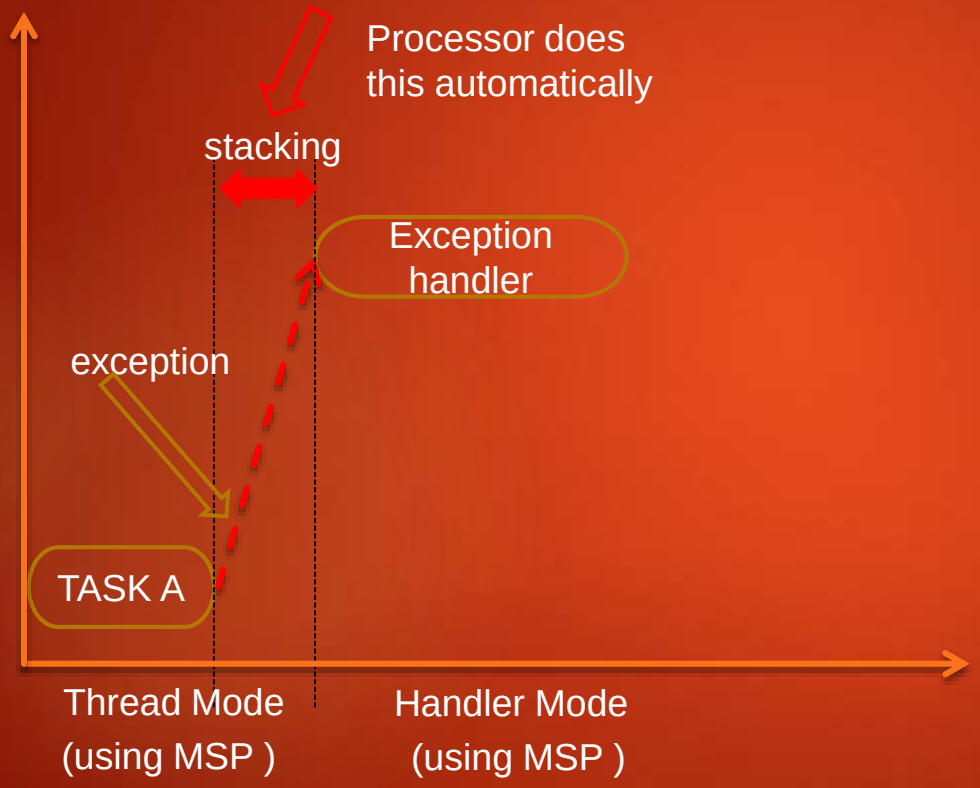
1. when a function is called, as per below table, registers are used for parameter passing.

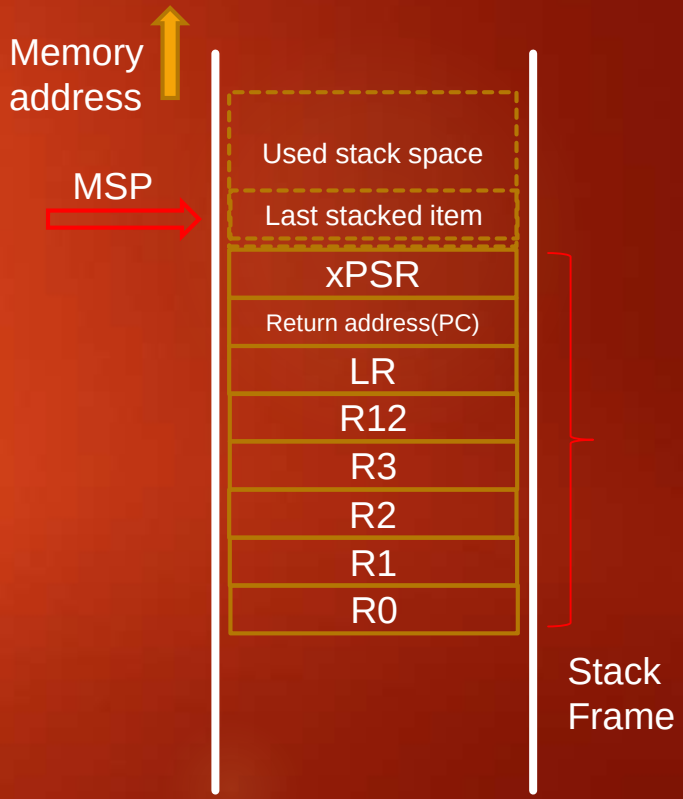
2. It's the callee "function" responsibility to push the contents of R4-R11,R13,R14 if the function is going to change these registers(compiler takes care when coded in c)

Register	Input Parameter	Return value
R0	First input parameter	Function return value
R1	Second input parameter	Function return value if size is 64bit
R2	Third input parameter	
R3	Fourth input parameter	

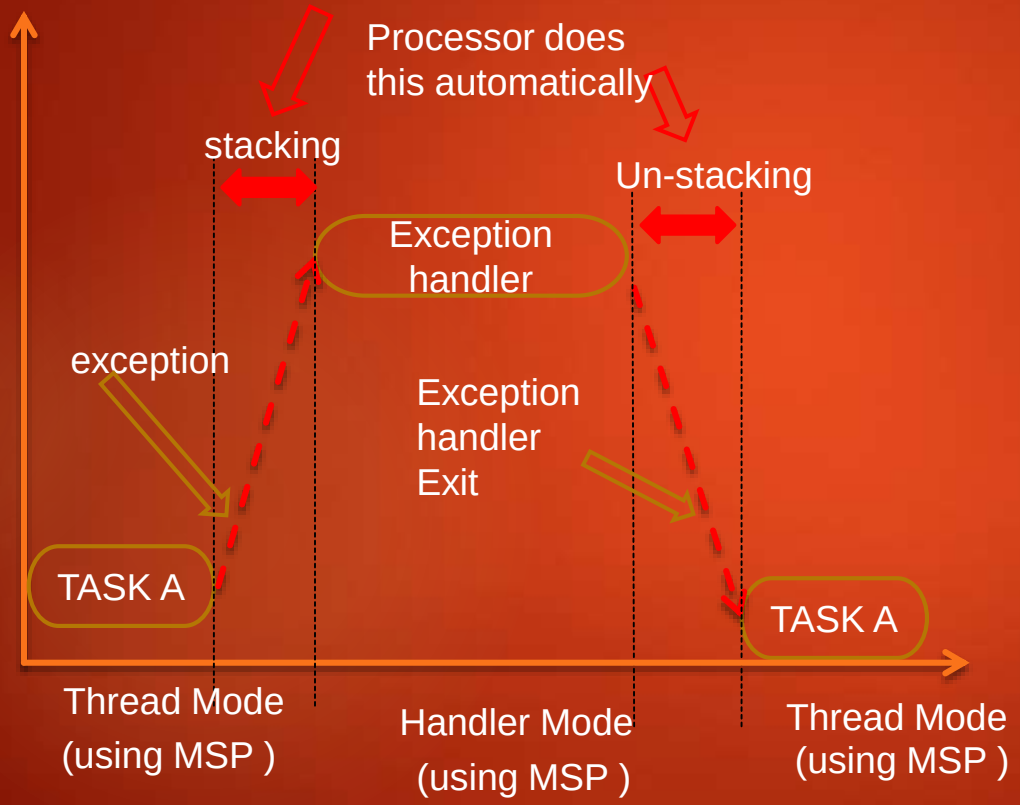
Stacking and Un-stacking during Exception

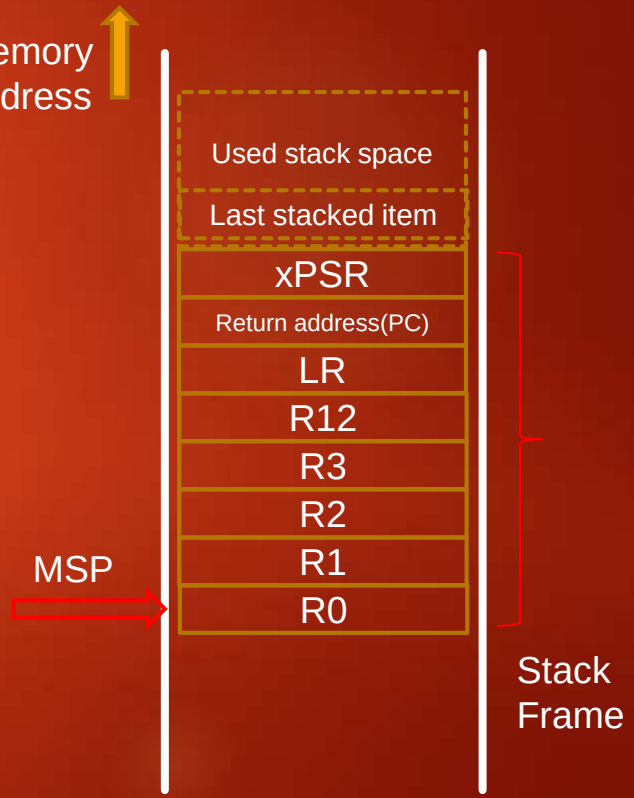
Stacking


Processor does this automatically
stacking
Exception handler
exception
TASK A
Thread Mode (using MSP)
Handler Mode (using MSP)


Memory address
MSP
Used stack space
Last stacked item
xPSR
Return address(PC)
LR
R12
R3
R2
R1
R0
Stack Frame

Un-stacking


Processor does this automatically
stacking
Un-stacking
Exception handler
exception
Exception handler Exit
TASK A
TASK A
Thread Mode (using MSP)
Handler Mode (using MSP)
Thread Mode (using MSP)


Memory address
Used stack space
Last stacked item
xPSR
Return address(PC)
LR
R12
R3
R2
R1
R0
MSP
Stack Frame

Demonstration of Stack Operations Using Different Stack Pointers(MSP/PSP)

Write a program to PUSH the contents of R0,R1,R2 Registers using MSP as a stack pointer, and then POP the contents back using PSP as a stack pointer

Points to Remember !!

- ✓ Processor always start in Privileged level and uses MSP as a stack pointer
- ✓ If you wish, you can change the stack pointer to PSP by writing to CONTROL register
- ✓ In C program , if you want to access MSP,PSP,CONTROL register then you have to use either CMSIS APIs or assembly programming.

Lets Code and Understand

System Exceptions & Interrupts

KIRAN NAYAK| SECTION-8

What Are Exceptions ??

- ✓ Exceptions Are events, which are generated asynchronously, either from external world or from internal system .

- ✓ When generated processor changes normal program flow to provide the service for the generated exceptions

“In a simple way, anything which disturbs the normal flow of execution of the processor is an exception “

What Are Interrupts ?

- ✓ **Remember Interrupt is also an exception !!**
- ✓ We use a word **“interrupt”** for the **exception from external world**. E.g.: peripheral like timers, RTC,I2C,NMI, I/Os. External means “**external to the processor core**”
- ✓ Courtex-M3/M4 Supports 240 interrupts

Exception types

- ✓ Exceptions are divided into 2 types
  - System exceptions(internal to the processor)
  - External exceptions(as already said, name for this is **interrupt**)

- ✓ **15 system exceptions**, numbered from 1 to 15.

- ✓ **240 interrupts**, numbered from 16 to 255

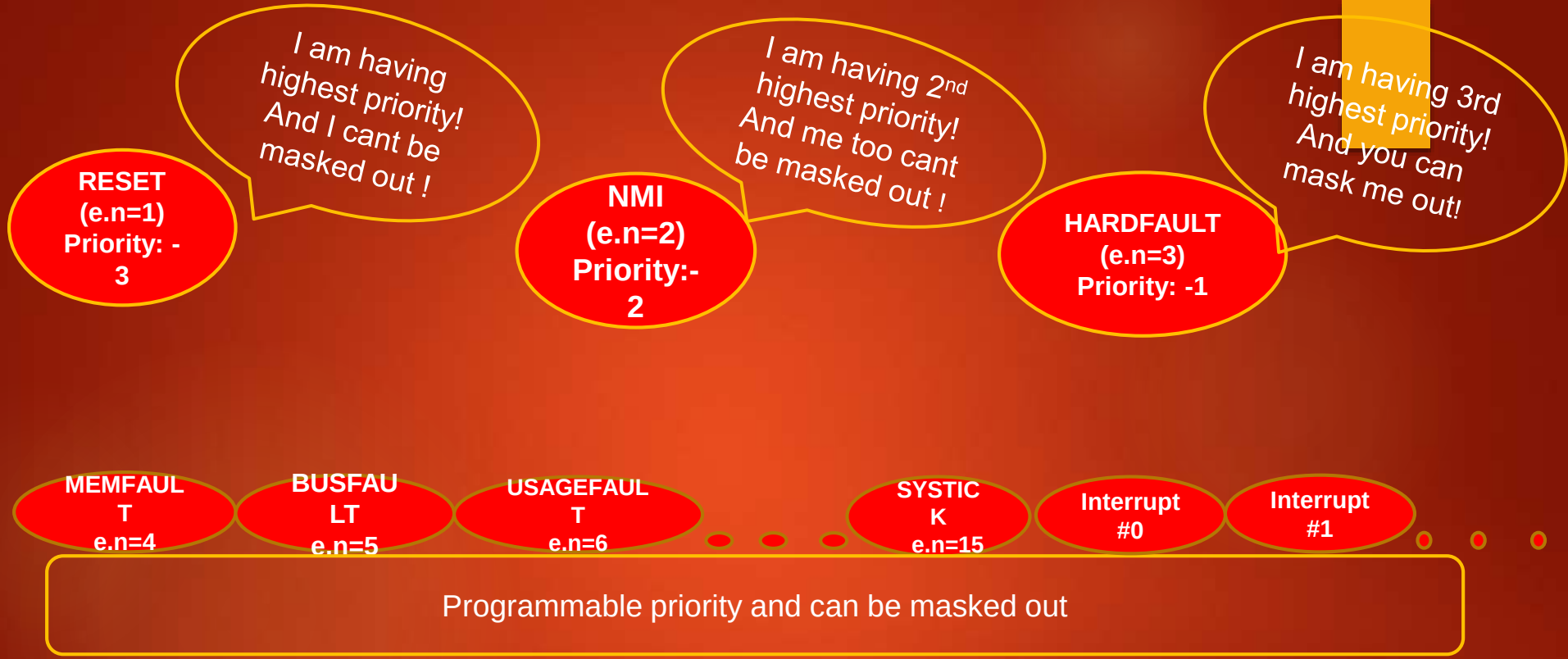
- ✓ **So for e.g. the 16th exception is an interrupt#0**

List Of System Exceptions

Exception number	Exception type	Priority	Description
1	Reset	-3(Highest)	Reset. Pressing a reset button causes this
2	NMI	-2	Non Maskable Interrupt.
3	Hard Fault	-1	all fault conditions, if the corresponding fault handler is not enabled
4	Mem Manage Fault	Programmable	memory manager fault: happens when try to access illegal memory areas, MPU Violation
5	Bus Fault	Programmable	Bus Error. Prefetch abort or data abort on AHB bus
6	Usage Fault	Programmable	invalid usage of some features by the program
7-10	Reserved	---	
11	svc	Programmable	supervisor call
12	Debug monitor	Programmable	debug monitor(breakpoints, watchpoints)
13	Reserved	--	
14	PendSV	Programmable	Pendable service call
15	SysTick	Programmable	System Tick Timer

List of Interrupts

Exception Number	interrupt number	Exception Type	Priority
16	0	external interrupt 0	programmable
17	1	external interrupt 1	programmable
--	--	--	--
--	--	--	--
255	240	external interrupt 240	programmable

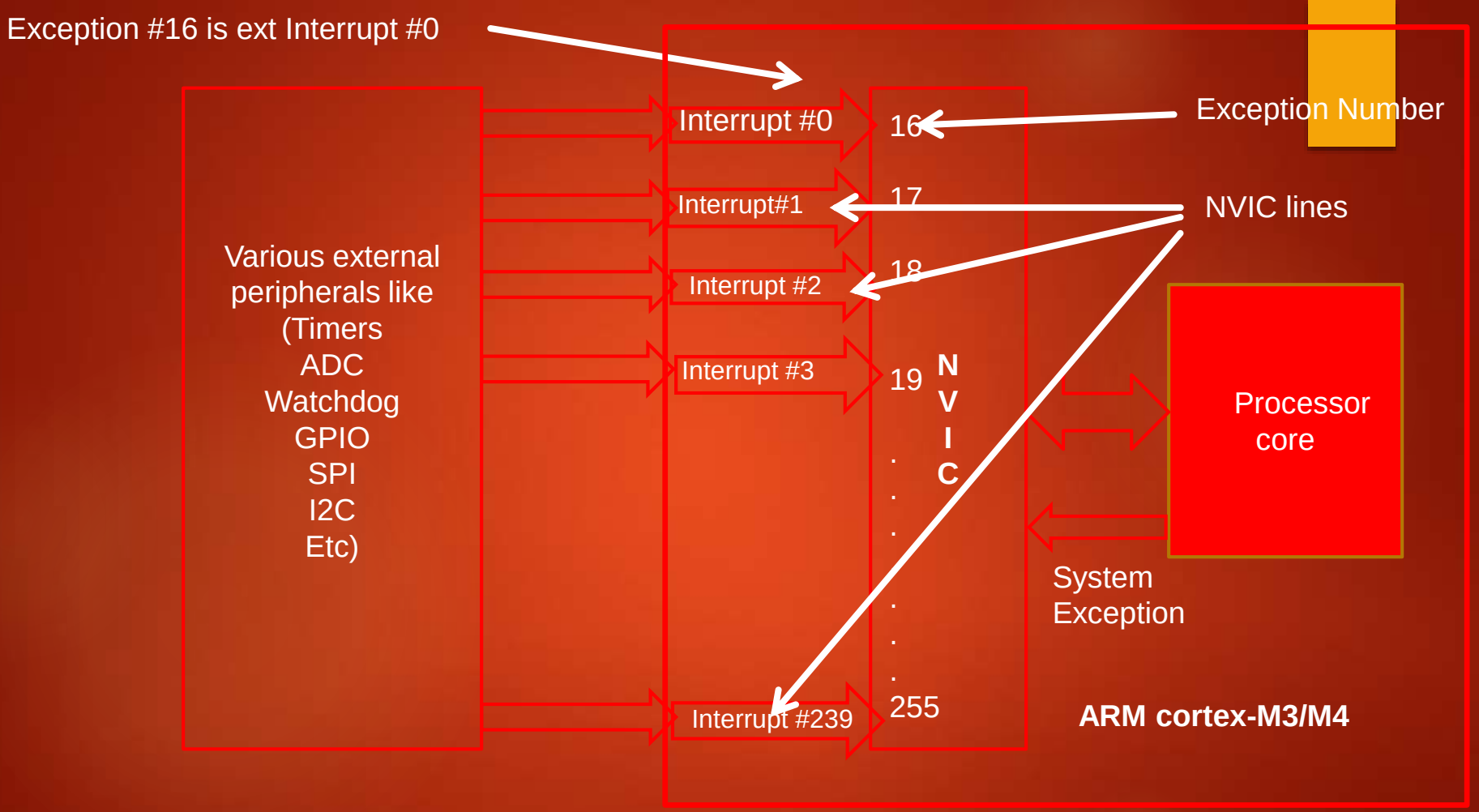


e.n: exception number

Nested **V**ectored **I**nterrupt **C**ontroller (NVIC)

“This is the HW block, which receives and manages exceptions from various sources and delivers to the processor core as per priority ! ”

Lets Understand how System Exceptions and Interrupts are connected to the Processor


Exception #16 is ext Interrupt #0
Various external peripherals like (Timers ADC Watchdog GPIO SPI I2C Etc)
Interrupt #0
Interrupt#1
Interrupt #2
Interrupt #3
Interrupt #239
16
17
18
19
255
NVIC
Exception Number
NVIC lines
Processor core
System Exception
ARM cortex-M3/M4

Points to Remember

- The cortex M processors have a number of programmable registers for managing the interrupts .
- Most of these registers are inside NVIC block (and System Control Block).
- There are 2 ways to configure and mange the interrupt
  - Directly access the NVIC registers
  - User CMSIS core APIs
- The NVIC registers can only be accessed in privileged access level.

Points to remember Contd.

- After reset , all interrupts are disabled and given a priority level value of 0. So before using any interrupts you need to .
  - Set up the priority level of the required interrupt (optional)
  - Enable the interrupt in the NVIC interrupt enable register.
- When the interrupt triggers, the corresponding ISR will execute (you might need to clear the interrupt request from the peripheral with in the handler).
- The name of the ISR can be found inside the vector table inside the startup code. Which is also provided by the microcontroller vendor.

Interrupt Priority

Interrupt Priority

- When an interrupt can be accepted by the processor and get its handler executed is dependent on the priority of the interrupt.
- A higher priority interrupt can pre-empt a lower priority interrupt. This is also called nesting of interrupts
- Some of the exceptions(reset , NMI ,and HardFault) have fixed priority levels. Their priority level are represented in negative numbers to indicate that they are of higher priority than other exceptions.
- Remember , lower numbers for priority indicates higher priority

Interrupt Priority Levels

- Different microcontroller have different levels of priority, for example 8 levels of priority , 16 levels of priority ,etc.
- The microcontroller vendors are free to choose how many levels of priority level they want.
- There is a register called "Priority level register". which determines how many priority level actually supported in the microcontroller implementation

Interrupt Priority Level Register

Microcontroller Vendor XXX

8 Levels of Priority level

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

Microcontroller Vendor YYY

16 Levels of Priority level

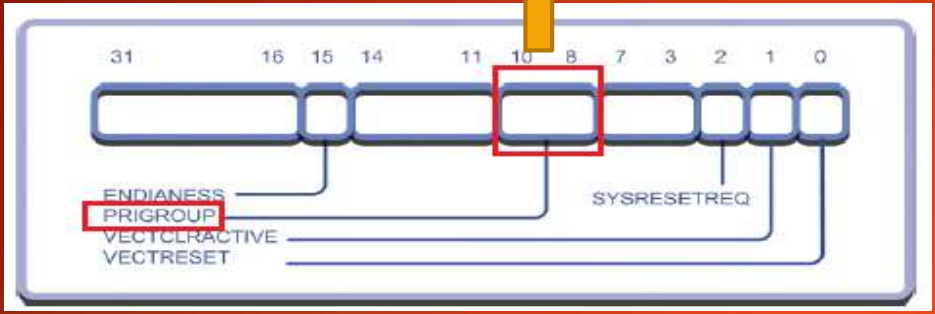
0x00,ox10,0x20,0x30,0x40,0x50,
0x60,0x70,0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0


Highest Priority
-3
-2
-1
0
0X20
0X40
0X60
0X80
0XA0
0XC0
0XE0
0XFF
RESET
NMI
HARD FAULT
Programmable Exceptions
lowest Priority
3-bits priority width In priority level register
0X00
0X20
0X40
0X60
0X80
0XA0
0XC0
0XE0
4 bits priority width In priority level register
0X00
0X10
0X20
0X30
0X40
0X50
0X60
0X70
0X80
0X90
0XA0
0XB0
0XC0
0XD0
0XE0
0XF0

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]

Priority Grouping



Application Interrupt And Reset Control Register

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]

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 exception with higher pre-empt priority(less in number) will be allowed to run.

Sub Priority : this value is used **only when two exceptions with same pre-empt priority level occur at the same time.** In this case , the exception with higher sub-priority(less in number) will be handled first .

Priority Grouping Case Study

Case 1 :

when the Priority group = 0,

As per the table ,

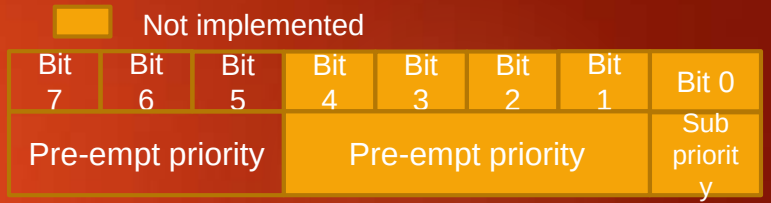
we have

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

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

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

Bit 0 is not implemented so no sub priority level



priority level register

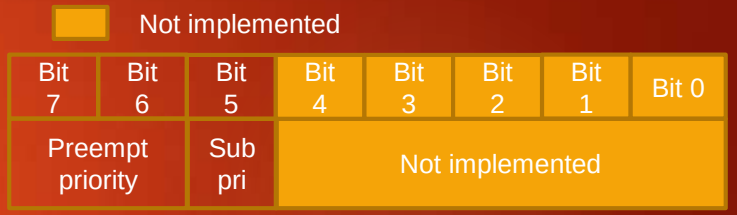
Priority Grouping Case Study

Case 2 : when the Priority_group = 5,

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

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

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



QUIZ-3

When the Priority group = 7, find out

pre-empt priority width = ??

Sub-priority width = ??

And give a conclusion,

How pre-emption works in the system ?

Configuring Interrupt Priority

Function	**void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)**
Description	Sets the priority grouping field using the required unlock sequence. The parameter PriorityGroup is assigned to the field SCB->AIRCR [10:8] PRIGROUP field.
Parameter	priority group value (0...7)
Return	none

Function	**uint32_t NVIC_GetPriorityGrouping**
Description	Reads the priority grouping field from the NVIC Interrupt Controller.
Parameter	none
Return	Priority grouping field (SCB->AIRCR [10:8] PRIGROUP field)

Configuring Interrupt Priority

Function	**uint32_t NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t PreemptPriority, uint32_t SubPriority)**
Description	Encodes the priority for an interrupt with the given priority group, preemptive priority value, and subpriority value..
Parameter	PriorityGroup Used priority group. PreemptPriority Preemptive priority value (starting from 0). SubPriority Subpriority value (starting from 0).
Return	Encoded priority. Value can be used in the function NVIC_SetPriority().

Function	**void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)**
Description	Sets the priority of an interrupt.
Parameter	IRQn Interrupt number. priority Priority to set.
Return	none

Priority Grouping Case Study

Case 2 :

when the Priority_group = 7,

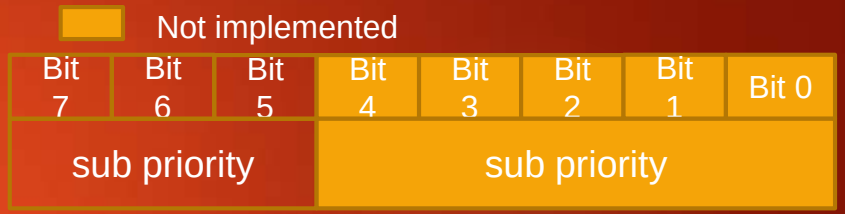
As per the table , in this case

pre-empt priority width = none

Sub-priority width = 8 (256 programmable

sub priority level)

Since only 3 bits are implemented , only 8 programmable sub priority levels



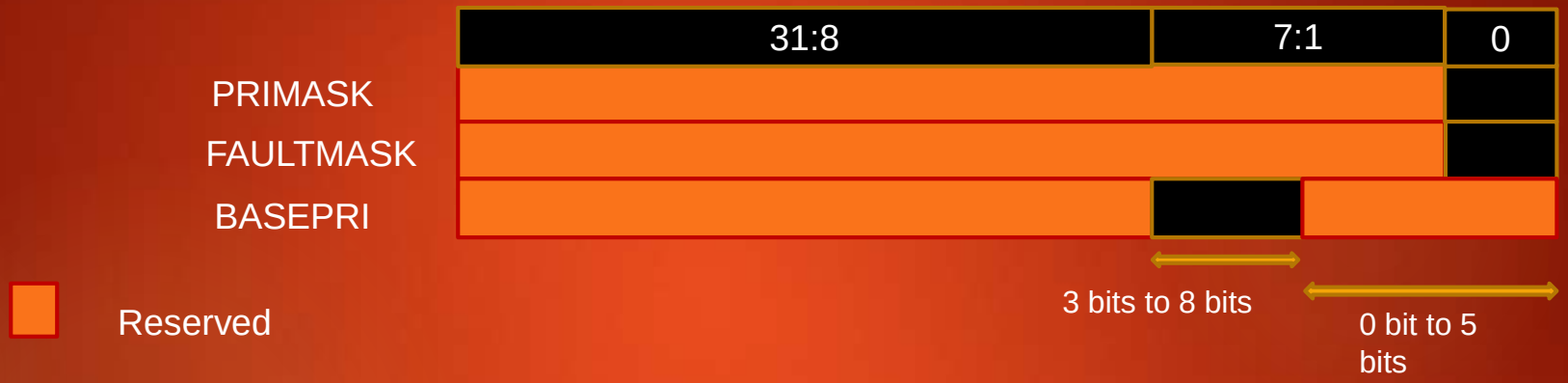
Exception Activation and De-Activation

Interrupt Activation

Function	**void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)**
Description	Sets the priority of an interrupt.
parameter	IRQn Interrupt number. priority Priority to set.
return	none

Function	**void NVIC_EnableIRQ(IRQn_Type IRQn)**
Description	Enables a device-specific interrupt in the NVIC interrupt controller.
parameter	IRQn External interrupt number. Value cannot be negative.
return	none

Global Exception Enable/Disable



CMSIS APIs to handle PRIMASK register

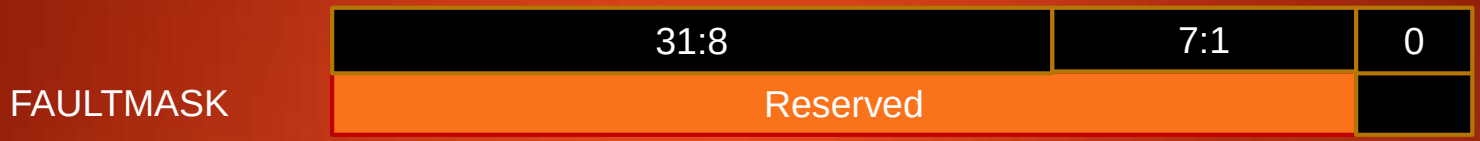
void __enable_irq(); // Clear PRIMASK

void __disable_irq(); // Set PRIMASK

void __set_PRIMASK(uint32_t priMask); // Set PRIMASK to value

uint32_t __get_PRIMASK(void); // Read the PRIMASK value

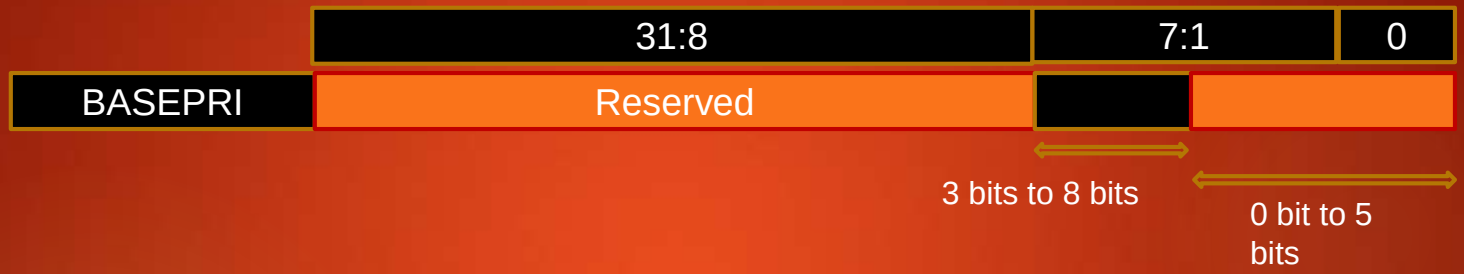
Global Exception Enable/Disable



CMSIS APIs to handle FAULTMASK register are

void __set_FAULTMASK(uint32_t faultMask);
uint32_t __get_FAULTMASK(void);

Global Exception Enable/Disable



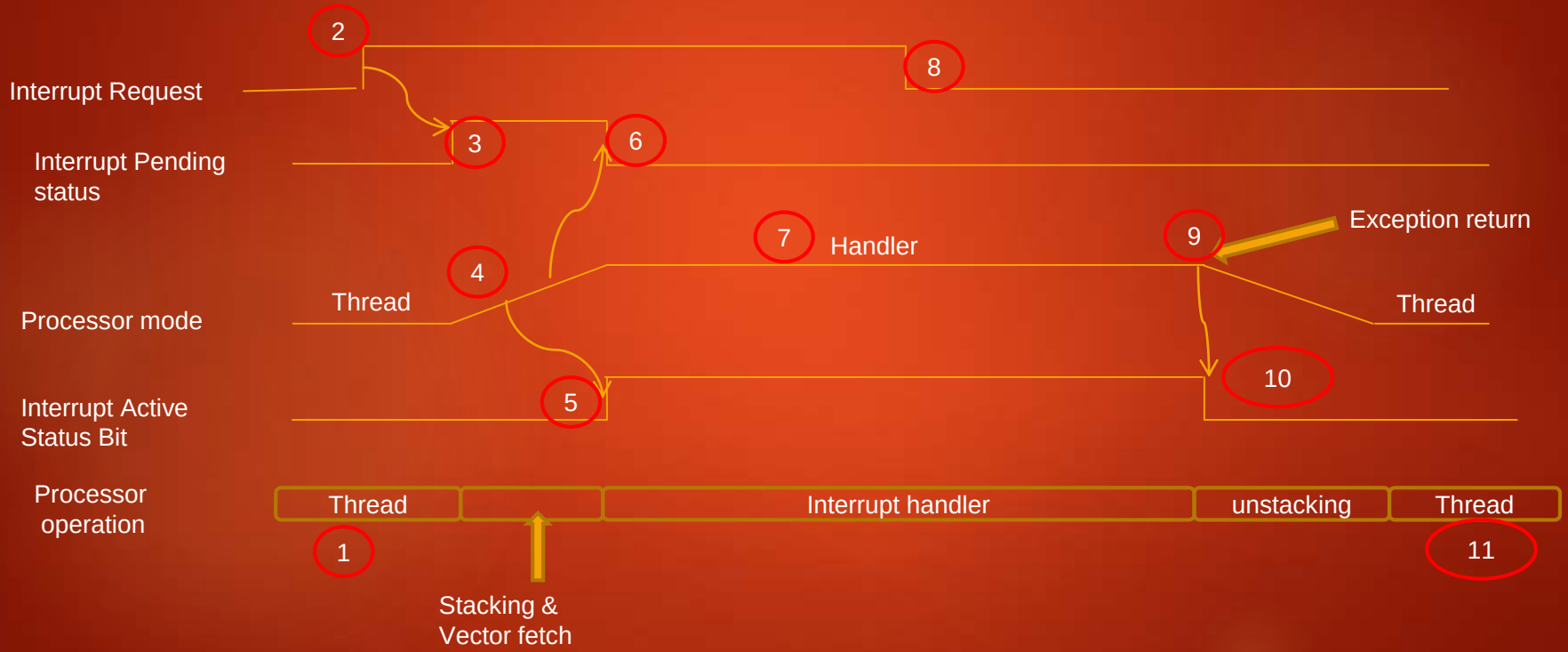
For example, if you want to block all exceptions with priority level equal to or lower than 0x60 , you can use this CMSIS function.

__set_BASEPRI(0x60); // Disables interrupts with priority 0x60 to 0xFF

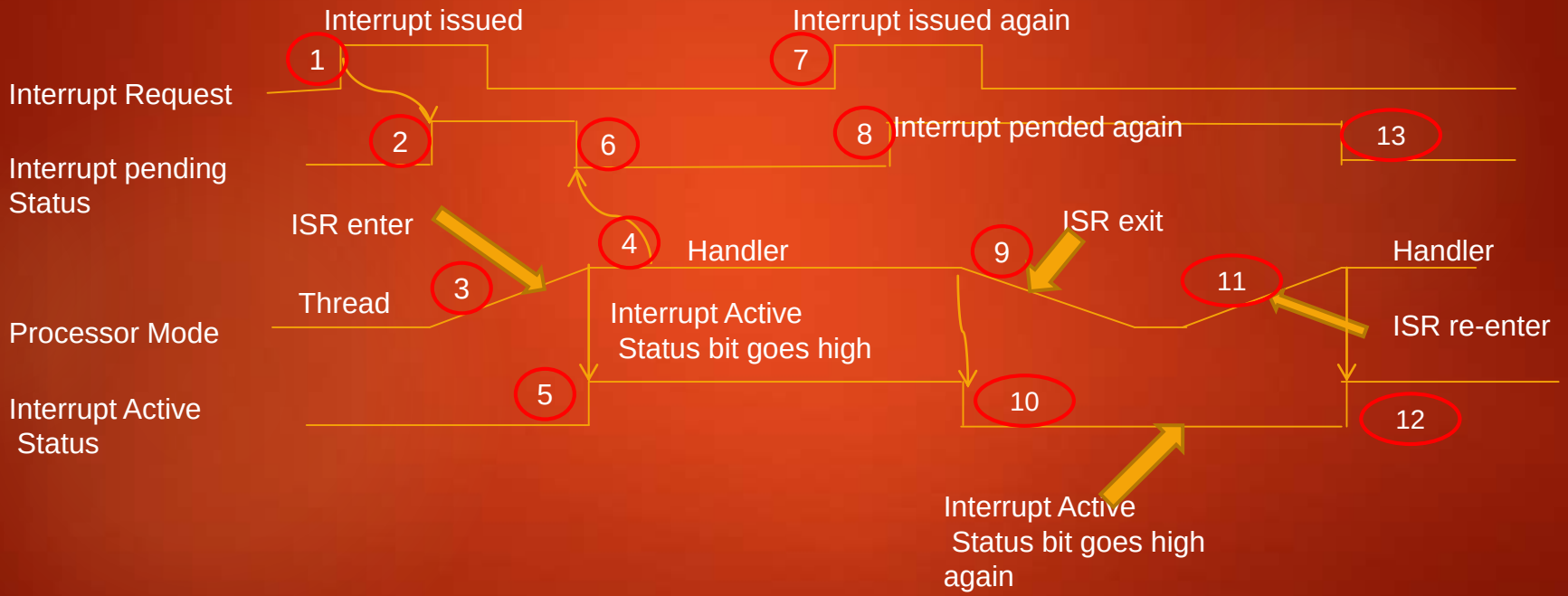
You can also read back the value of BASEPRI:
x = __get_BASEPRI(void); // Read value of BASEPRI

Pending Interrupt Behavior

Case1: Single Pended Interrupt



Case2: Double Pended Interrupt



Quiz 4

What happens when you masked out or disabled an interrupt from particular peripheral, but the peripheral still issues an interrupt ?

Demonstrating Enable/Disable Exceptions using PRIMASK and BASEPRI registers

PRIMASK

BASEPRI

Available Priority levels for 4bit priority-level-register

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

Priority Level Resister implementation in STM32F4xx

0x00, 0x10, 0x20, 0x30, 0x40

0x50, 0x60, 0x70, 0x80, 0x90

0xA0, 0xB0, 0xC0, 0xD0, 0xE0, 0xf0

Congratulations ☺

You have learnt

- How to disable and enable interrupts/Exception using PRIMASK and BASEPRI Registers

Getting Started with USB-Logic Analyzer

Hardware & Software

8 Channels(Digital)
2 GND pins
Max 100 MS/s
(Million Samples/sec)

Software Download for Windows/MAC/Linux

https://www.saleae.com/downloads

Soft

USB

CH1

GND

Embedded Board

GPIO (LED)

GND

Demonstrating Interrupt Priority and Pre-Emption By Configuring Two Interrupts

In this program we will use Systick Timer exception and Button interrupt to understand how pre-emption works due to changing priorities.

SysTick Timer

- The arm cortex M3 and M4 processor has a 24-bit system timer called sysTick.
- The counter inside the sysTick is 24 bit decrement counter. when you start the counter by loading some value, it starts decrementing for every processor clock cycle.
- If it reaches zero , then it will raise a sysTick timer exception and then again reloads the value and continue.
- SysTick Timer can be used for time keeping, time measurement, or as an interrupt source for tasks that need to be executed regularly.

First Case

Interrupt	Priority	Note
SysTick Timer	0xF0	Low
Button	0x00	High

Lets Code and Understand !!

Systick Timing Calculation

We are using 16Mhz internal RC oscillator

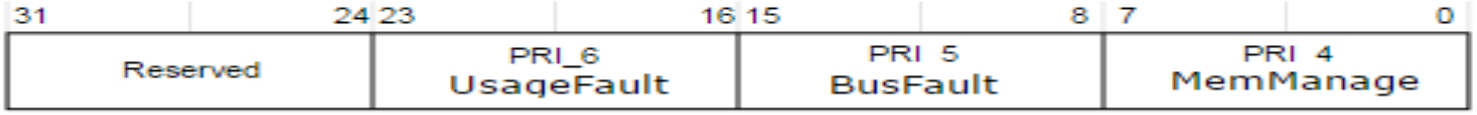
1 tick takes ➔ 1 processor cycle (1/16Mhz)

So, time for

1 tick → 0.0000000625 seconds
2000 ticks ➔ 0.000125 seconds

System Handler Priority Register 1

The bit assignments are:



SHP[3] SHP[2] SHP[1] SHP[0]

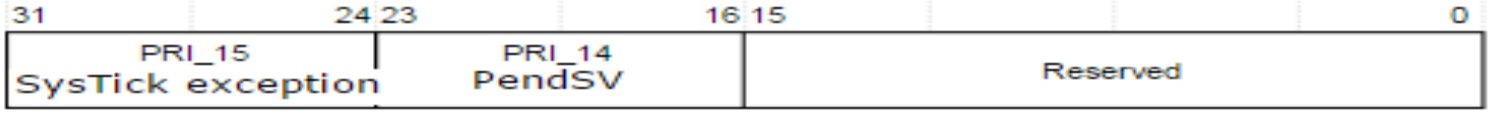
System Handler Priority Register 2

The bit assignments are:

SHP[7] SHP[6] - SHP[4]

System Handler Priority Register 3

The bit assignments are:



SHP[B] SHP[A] SHP[9] - SHP[8]

System Control Block

Second Case

Interrupt	Priority	Note
SysTick Timer	0x00	High
Button	0xF0	Low

Congratulations !! ☺

You Have Learnt

- Configuring interrupt priority
- Interrupt pre-emption in action
- Debugging interrupts by using GPIOs
- How to use USB logic analyzers

NVIC Registers

NVIC Registers for interrupt control

- ✓ Interrupt Enable Registers
- ✓ Pending State Registers
- ✓ Active State Registers
- ✓ Interrupt Mask And Unmask Registers
- ✓ Interrupt Priority Registers

NVIC Register Summary

NVIC register summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000E100-0xE000E11C	NVIC_ISER0-NVIC_ISER7	RW	Privileged	0x00000000	*Interrupt Set-enable Registers*
0XE000E180-0xE000E19C	NVIC_ICER0-NVIC_ICER7	RW	Privileged	0x00000000	*Interrupt Clear-enable Registers*
0XE000E200-0xE000E21C	NVIC_ISPR0-NVIC_ISPR7	RW	Privileged	0x00000000	*Interrupt Set-pending Registers*
0XE000E280-0xE000E29C	NVIC_ICPR0-NVIC_ICPR7	RW	Privileged	0x00000000	*Interrupt Clear-pending Registers*
0xE000E300-0xE000E31C	NVIC_IABR0-NVIC_IABR7	RW	Privileged	0x00000000	*Interrupt Active Bit Registers*
0xE000E400-0xE000E4EF	NVIC_IPR0-NVIC_IPR59	RW	Privileged	0x00000000	*Interrupt Priority Registers*
0xE000EF00	STIR	WO	Configurable[a]	0x00000000	*Software Trigger Interrupt Register*

Interrupt Enable/Disable Registers

To enable the interrupt , the bit corresponding to IRQ number must be set in the NVIC_ISERn (n = 0 to 7) register

Address	Name	Type	Reset value	Description
0XE000E100	NVIC_ISER0	R/W	0	Enable the external interrupt #0 to #31 bit[0]=1 to enable interrupt #0, 0 has no effect bit[1]=1 to enable interrupt #1, 0 has no effect --- bit[31]=1 to enable interrupt #31, 0 has no effect
0XE000E104	NVIC_ISER1	R/W	0	Enable the external interrupt #32 to #63 bit[0]=1 to enable interrupt #32, 0 has no effect bit[1]=1 to enable interrupt #33, 0 has no effect --- bit[31]=1 to enable interrupt #63, 0 has no effect

Interrupt Enable/Disable Registers

To disable the interrupt ,corresponding bit in the NVIC_ICERn (n = 0 to 7) register must be set

Address	Name	Type	Reset value	Description
0XE000E180	NVIC_ICER0	R/W	0	Disable the external interrupt #0 to #31 bit[0]=1 to disable interrupt #0, write 0 has no effect bit[1]=1 to disable interrupt #1, write 0 has no effect --- bit[31]=1 to disable interrupt #31, write 0 has no effect
0XE000E184	NVIC_ICER1	R/W	0	Disable the external interrupt #32 to #63 bit[0]=1 to disable interrupt #32, write 0 has no effect bit[1]=1 to disable interrupt #33, write 0 has no effect --- bit[31]=1 to disable interrupt #63, write 0 has no effect

Pending State Registers

There are two sets of registers to manage pending of interrupts

1. **ISPRn(n= 0 to 7)(Interrupt Set Pending Register)**
    - whenever any interrupt occurs , bit corresponding to its irq number will be set in the "ISPR" register
    - you can force the triggering of interrupt by setting a bit corresponding to interrupt number
2. **ICPRn(n = 0 to 7) (Interrupt Clear-Pending Register)**

    you can clear any pending interrupts by using this register

Interrupt Set Pending Register

Address	Name	Type	Reset value	Description
0XE000E200	NVIC_ISPR0	R/W	0	Pending for external interrupt #0 to #31 bit[0]=1 to pend interrupt #0, write 0 has no effect bit[1]=1 to pend interrupt #1, write 0 has no effect --- bit[31]=1 to pend interrupt #31, write 0 has no effect Read value indicates the current status
0XE000E204	NVIC_ISPR1	R/W	0	Pending for external interrupt #32 to #63 bit[0]=1 to pend interrupt #32, write 0 has no effect bit[1]=1 to pend interrupt #33, write 0 has no effect --- bit[31]=1 to pend interrupt #63, write 0 has no effect Read value indicates the current status

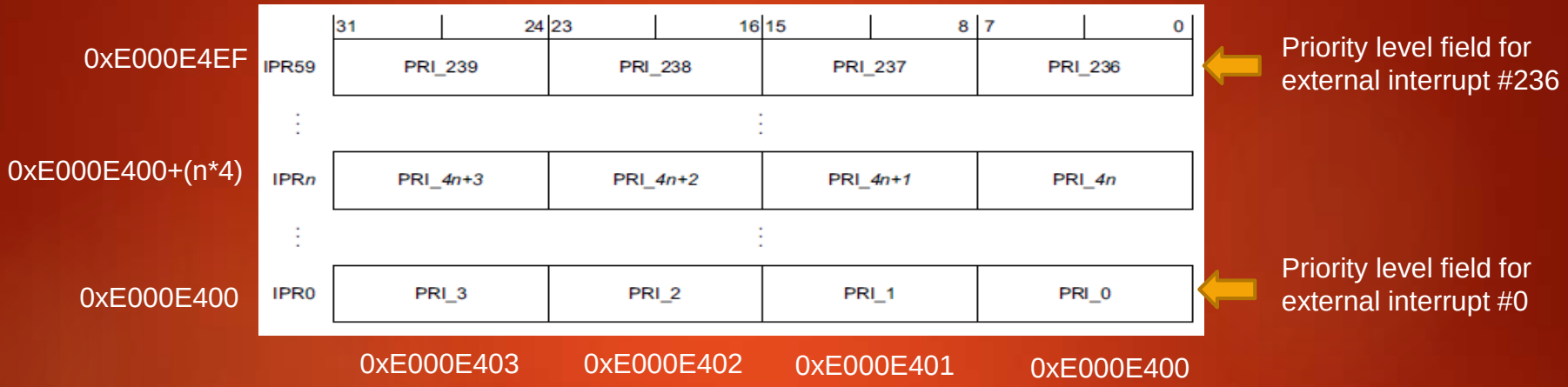
Interrupt Clear Pending Register

Address	Name	Type	Reset value	Description
0XE000E280	NVIC_ICPR0	R/W	0	Clear Pending for external interrupt #0 to #31 bit[0]=1 to Clear pend interrupt #0, write 0 has no effect bit[1]=1 to Clear pend interrupt #1, write 0 has no effect --- bit[31]=1 to Clear pend interrupt #31, write 0 has no effect Read value indicates the current pending status
0XE000E284	NVIC_ICPR1	R/W	0	clear Pending for external interrupt #32 to #63 bit[0]=1 to clear pend interrupt #32, write 0 has no effect bit[1]=1 to clear pend interrupt #33, write 0 has no effect --- bit[31]=1 to clear pend interrupt #63, write 0 has no effect Read value indicates the current pending status

Interrupt Active Bit registers

Address	Name	Type	Reset value	Description
0XE000E300	NVIC_IABR0	R	0	Active status for external interrupt #0 to #31 bit[0] is 1 automatically when interrupt #0 is being serviced bit[1] is 1 automatically when interrupt #1 is being serviced --- bit[31] is 1 automatically when interrupt #31 is being serviced Read value indicates the current active status of interrupts
0XE000E304	NVIC_IABR1	R	0	Active status for external interrupt #32 to #63 bit[0] is 1 automatically when interrupt #32 is being serviced bit[1] is 1 automatically when interrupt #33 is being serviced --- bit[31] is 1 automatically when interrupt #63 is being serviced Read value indicates the current active status of interrupts

Interrupt Priority Registers



Priority level field for external interrupt #236

Priority level field for external interrupt #0

Exception Vector Table

Whenever there is a system exception or interrupts, how does processor come to your ISR code to handle that interrupt or system exception ?

What is Vector Table ?

- The vector table contains the initial value of the Main Stack Pointer(MSP), and addresses of handlers for different system exceptions and external interrupts

- When you reset the processor, processor expect the vector table to be located in the code memory starting from address 0x00000000

Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.	.	.	.
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5		SVCall
10		0x002C	
9			Reserved
8			
7			
6	-10		Usage fault
5	-11	0x0018	Bus fault
4	-12	0x0014	Memory management fault
3	-13	0x0010	Hard fault
2	-14	0x000C	NMI
1		0x0008	Reset
		0x0004	Initial SP value
		0x0000	

Example Of a Vector Table Code From STM32F4XX

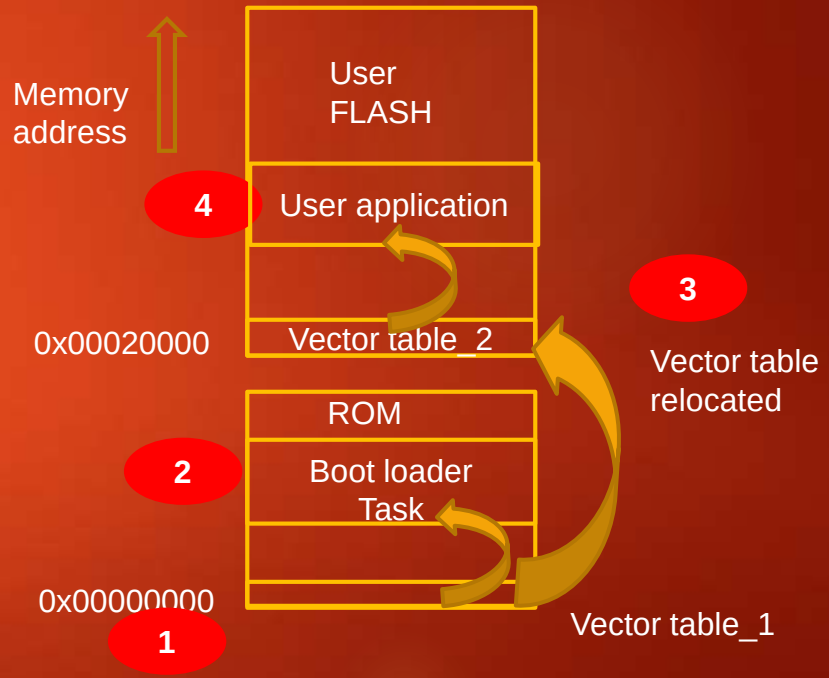
```
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     MemManage_Handler         ; MPU Fault Handler
                DCD     BusFault_Handler          ; Bus Fault Handler
                DCD     UsageFault_Handler        ; Usage Fault Handler
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
                DCD     SVC_Handler               ; SVCall Handler
                DCD     DebugMon_Handler          ; Debug Monitor Handler
                DCD     0                         ; Reserved
                DCD     PendSV_Handler            ; PendSV Handler
                DCD     SysTick_Handler           ; SysTick Handler

                ; External Interrupts
                DCD     WWDG_IRQHandler           ; Window WatchDog
                DCD     PVD_IRQHandler            ; PVD through EXTI Line detection
                DCD     TAMP_STAMP_IRQHandler     ; Tamper and TimeStamps through the
                DCD     RTC_WKUP_IRQHandler       ; RTC Wakeup through the EXTI line
                DCD     FLASH_IRQHandler          ; FLASH
```

Vector Table Relocating

Vector Table Offset Register(VTOR), address 0xE000ED08

Bit 31:30	Bit 29	Bit 28:7	Bit 6:0
TBLOFF(Vector Table Base Offset)			Reserved

Vector Able Reallocation Feature Case Study



Exception Entry/Exit Sequence

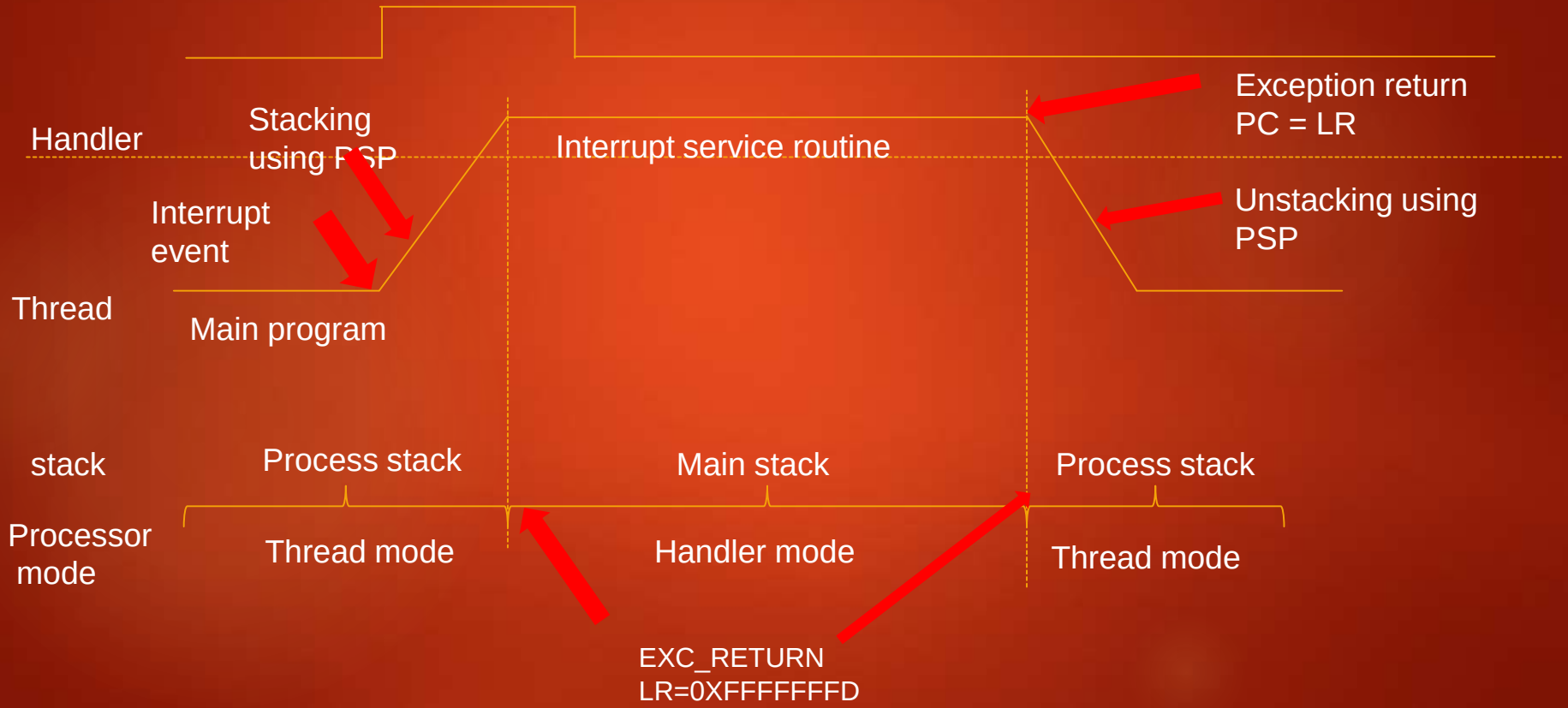
Exception Entry Sequence

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



EXC_RETURN

When it is generated ??

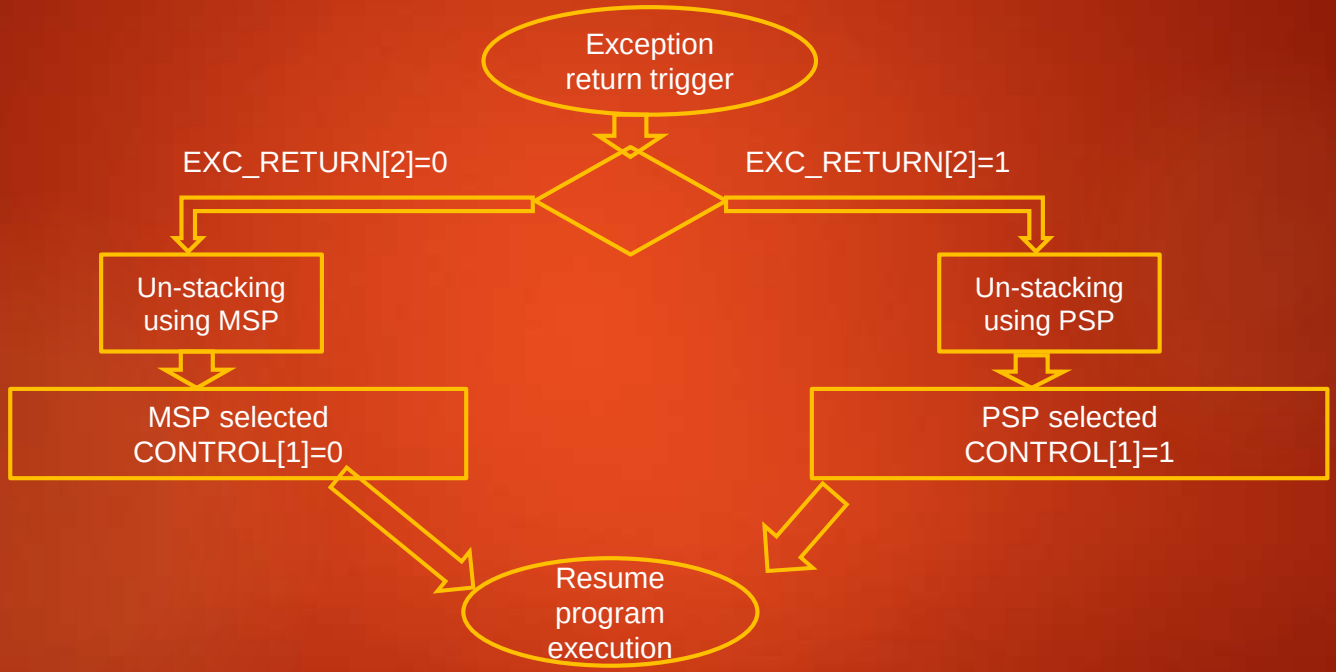
During an exception handler entry , the value of the return address(PC) Is not stored in the LR as it is done during calling of a normal C function. Instead The exception mechanism stores the special value called **EXC_RETURN in LR.**

EXC_RETURN Contd.

Decoding EXC_RETURN value

Bits	Descriptions	Values
31:28	EXC_RETURN indicator	0xF
27:5	Reserved(all 1)	0xEFFFFF
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
0xFFFFFFF1	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode. Exception Return get state from the main stack. Execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode. Exception return gets state from the process stack. Execution uses PSP after return.
All other values	Reserved.

Cortex M3/M4 OS Features

KIRAN NAYAK| SECTION-12

Session Overview

At the end of the session you will be able to understand

- ✓ Use shadowed stack pointer in OS
- ✓ SVC Exception and its uses
- ✓ PendSV Exception and its uses

How CORTEX-M3/M4 Helps OS ?

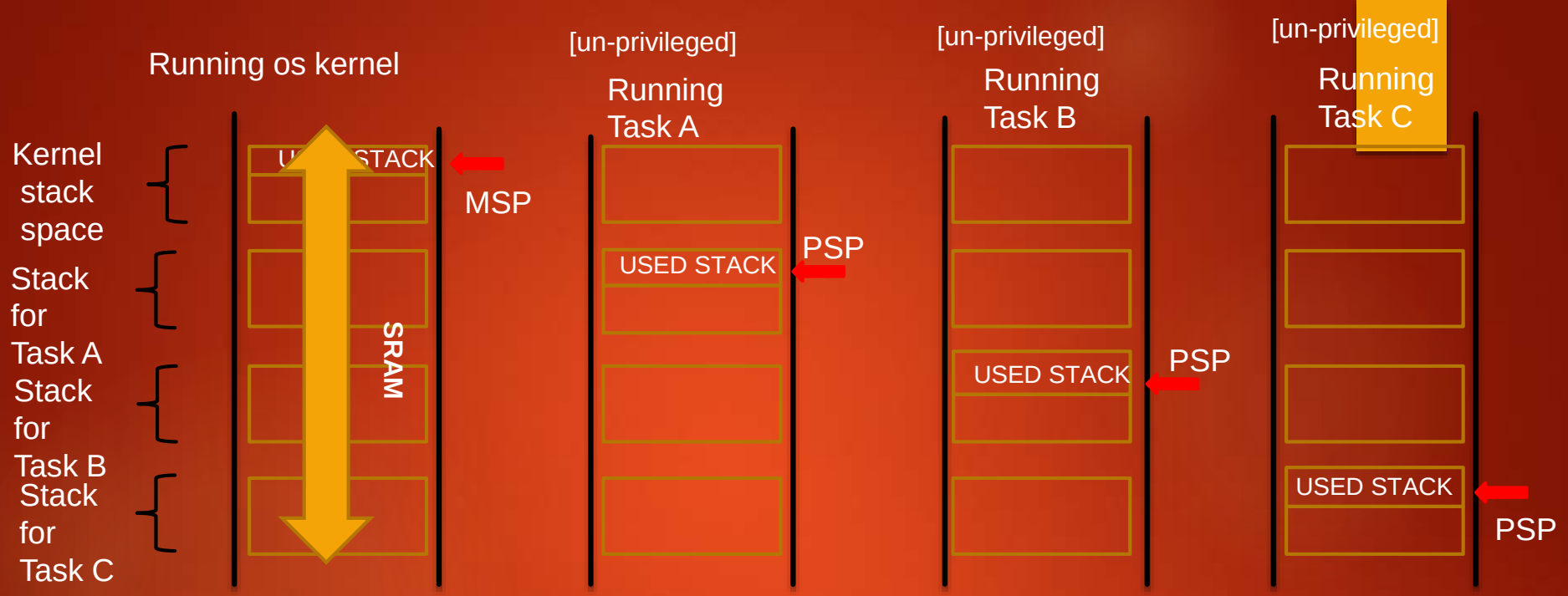
Because of the OS friendly features like:

- Shadowed stack pointers
- SysTick Timer
- SVC and PendSV exceptions

Shadowed Stack Pointer

- Physically 2 stack pointers are there in cortex-M3/M4
- The SP(R13), Which is called Stack Pointer, points to the currently selected stack pointer .
- Value of SPSEL bit in the CONTROL register determines which stack is currently active and used.

How OS Can Benefit From Shadowed Stack Pointers ?


Running os kernel
[un-privileged]
Running
Task A
[un-privileged]
Running
Task B
[un-privileged]
Running
Task C
Kernel
stack
space
Stack
for
Task A
Stack
for
Task B
Stack
for
Task C
STACK
MSP
SRAM
USED STACK
PSP
USED STACK
PSP
USED STACK
PSP

SVC System Exception

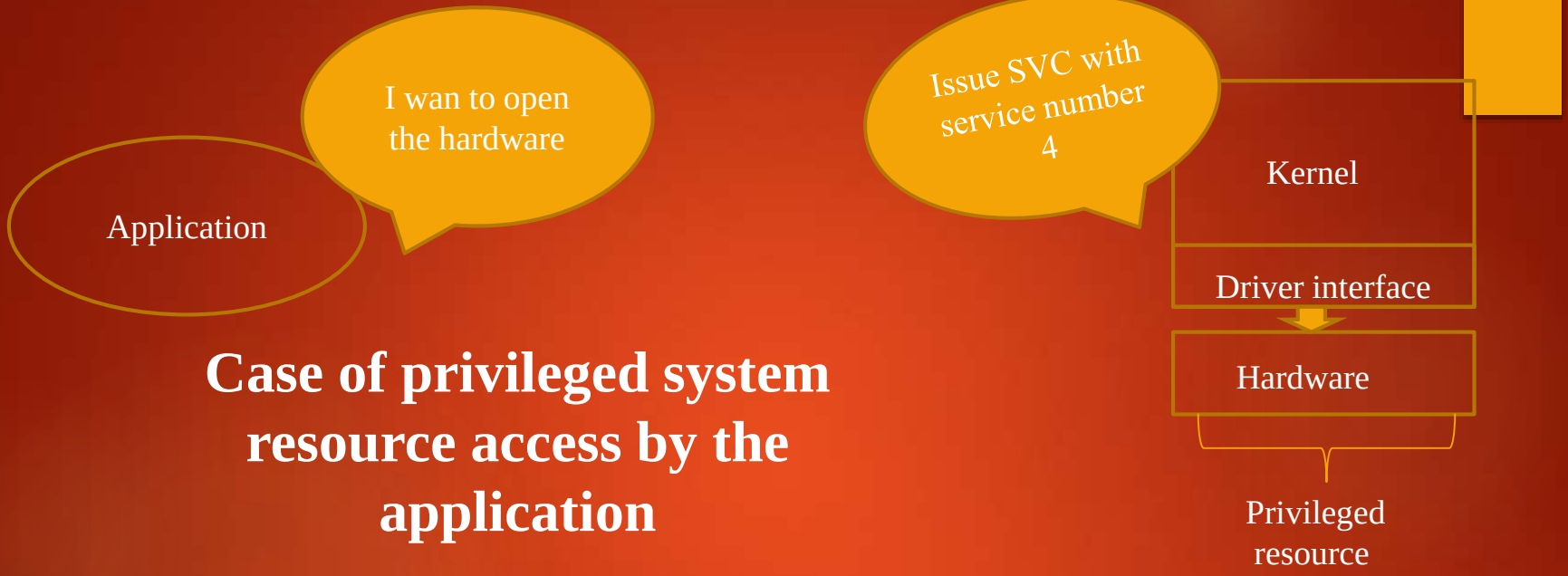
SVC Exception

- SVC stands for Supervisory Call
- This is triggered by SVC instruction
- The svc handler will execute right after the svc instruction(no delay !! Unless a higher priority exception arrive at the same time)

Advantages of SVC Exception

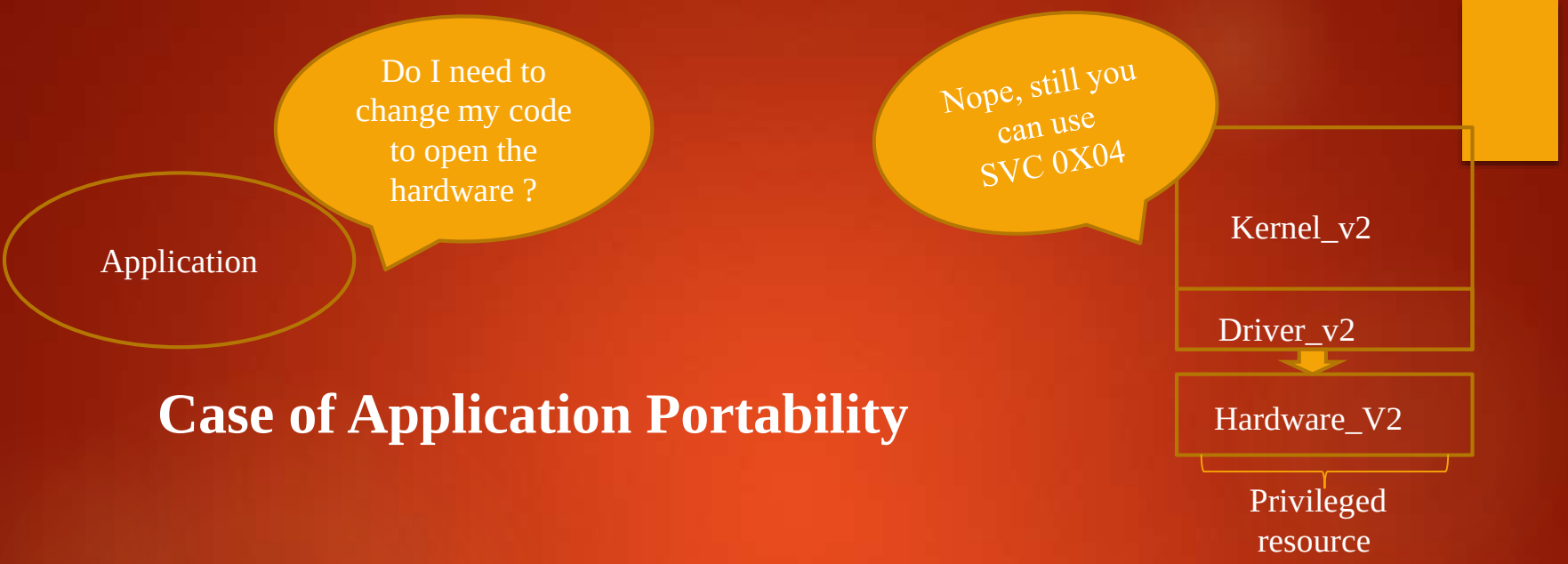
Case of privileged system resource access by the application

Case of privileged system resource access by the application



Case of Application Portability

Case of Application Portability



Method To Trigger SVC Exception

There are two ways

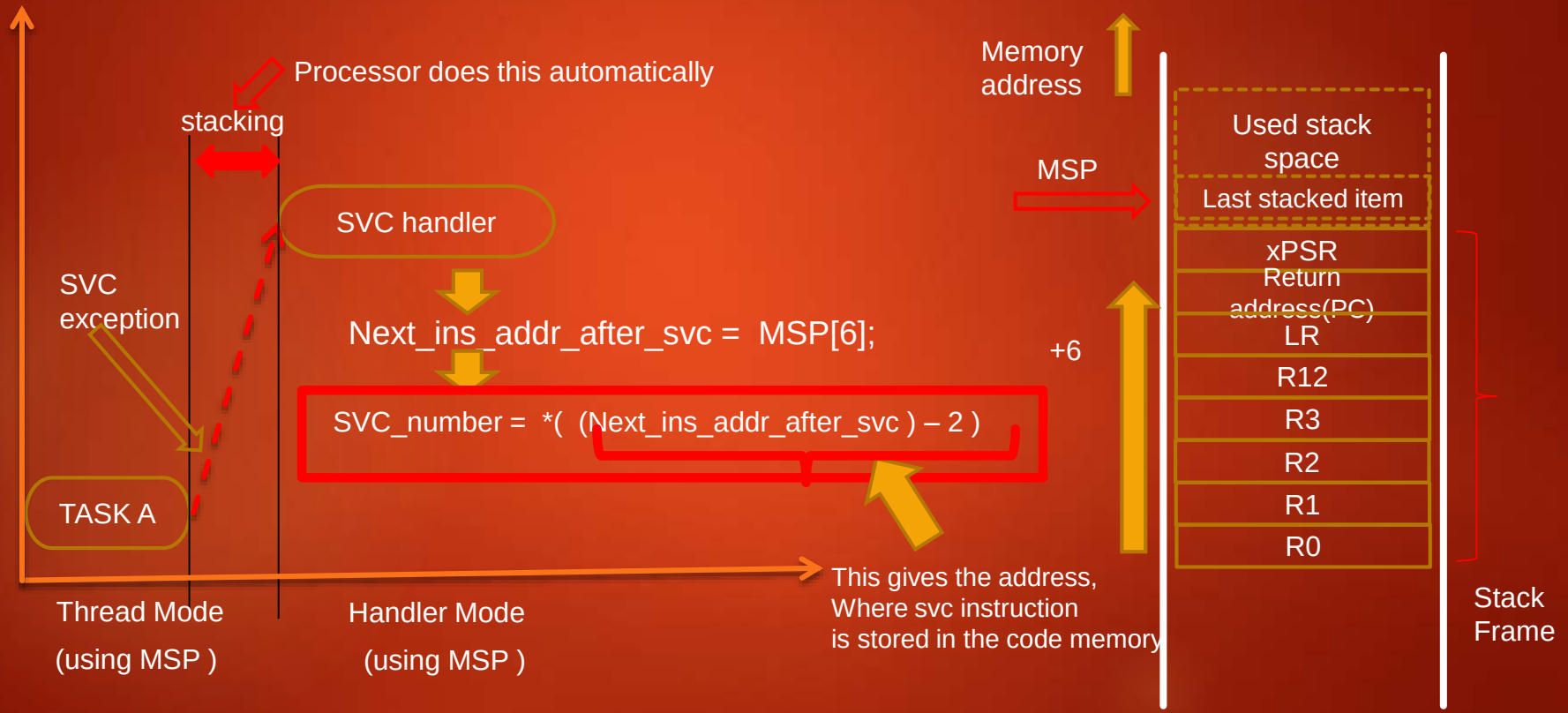
1) Direct execution of svc instruction with an immediate value
   - Example : SVC 0x04 in assembly
   - I will show you in the lab session how to issue through C program
   - Use SVC instruction is very efficient in terms of latency
2) Setting the exception pending bit in “**System Handler Control and State Register”**
   - **This method is not preferred and there is no reason why to use it**

How To Extract The SVC Number

- When svc instruction is executed, the associated immediate value(Service Number) will not be passed to SVC Exception Handler.

- The SVC handler need to **extract the number by using the PC value which was stored on to the stack** , prior coming to the exception handler.

How To Extract The SVC Number



PendSV System Exception

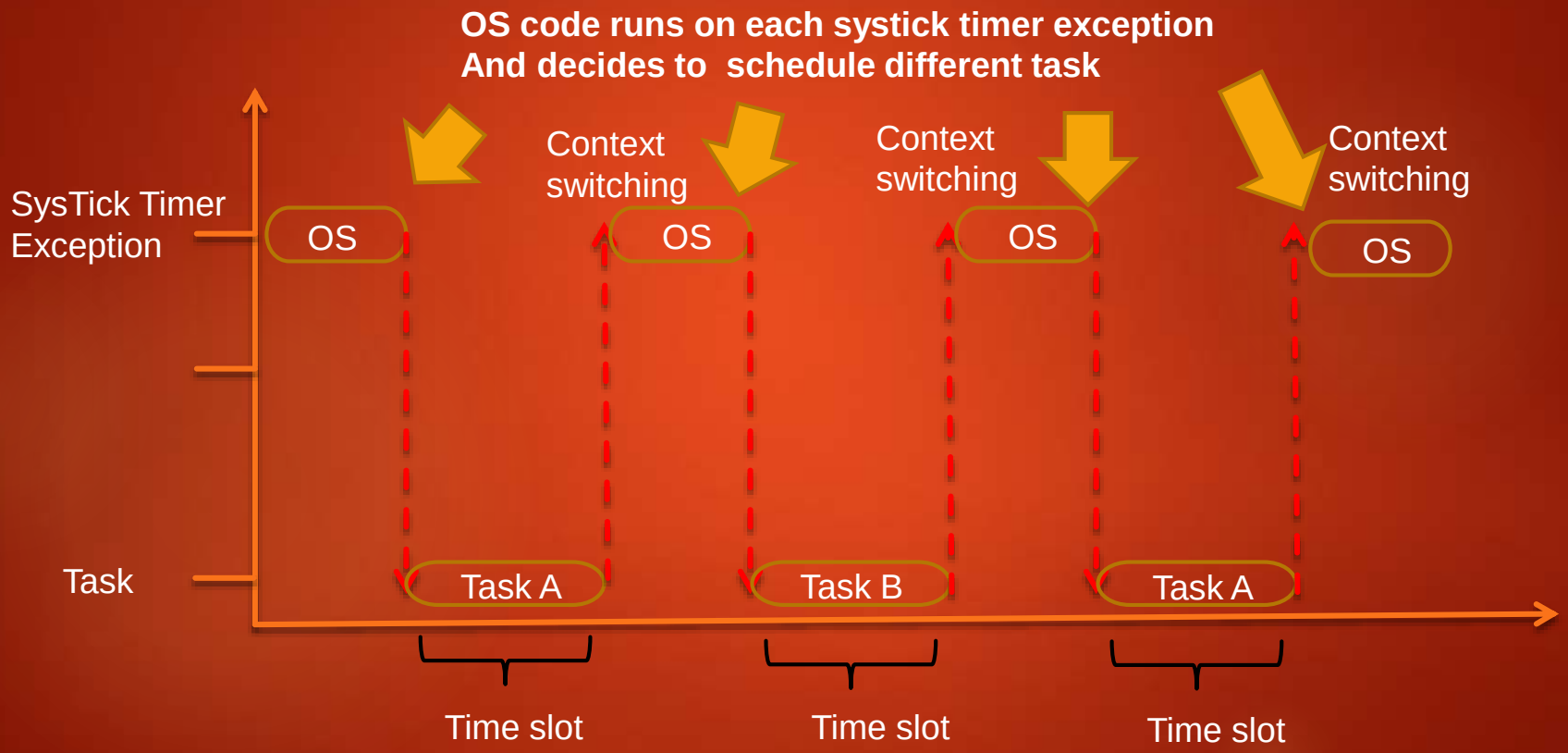
PendSV Exception

- In OS designs, we need to switch between different tasks to support multitasking .This is typically called context switching .
- Context switching is usually carried out in the PendSV exception handler
- It is exception type 14 and has a programmable priority level.
- It is basically set to lowest priority possible
- This exception is triggered by setting its pending status by writing to the “**Interrupt Control and State Register**”

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 key operation in various os design.

Context Switching

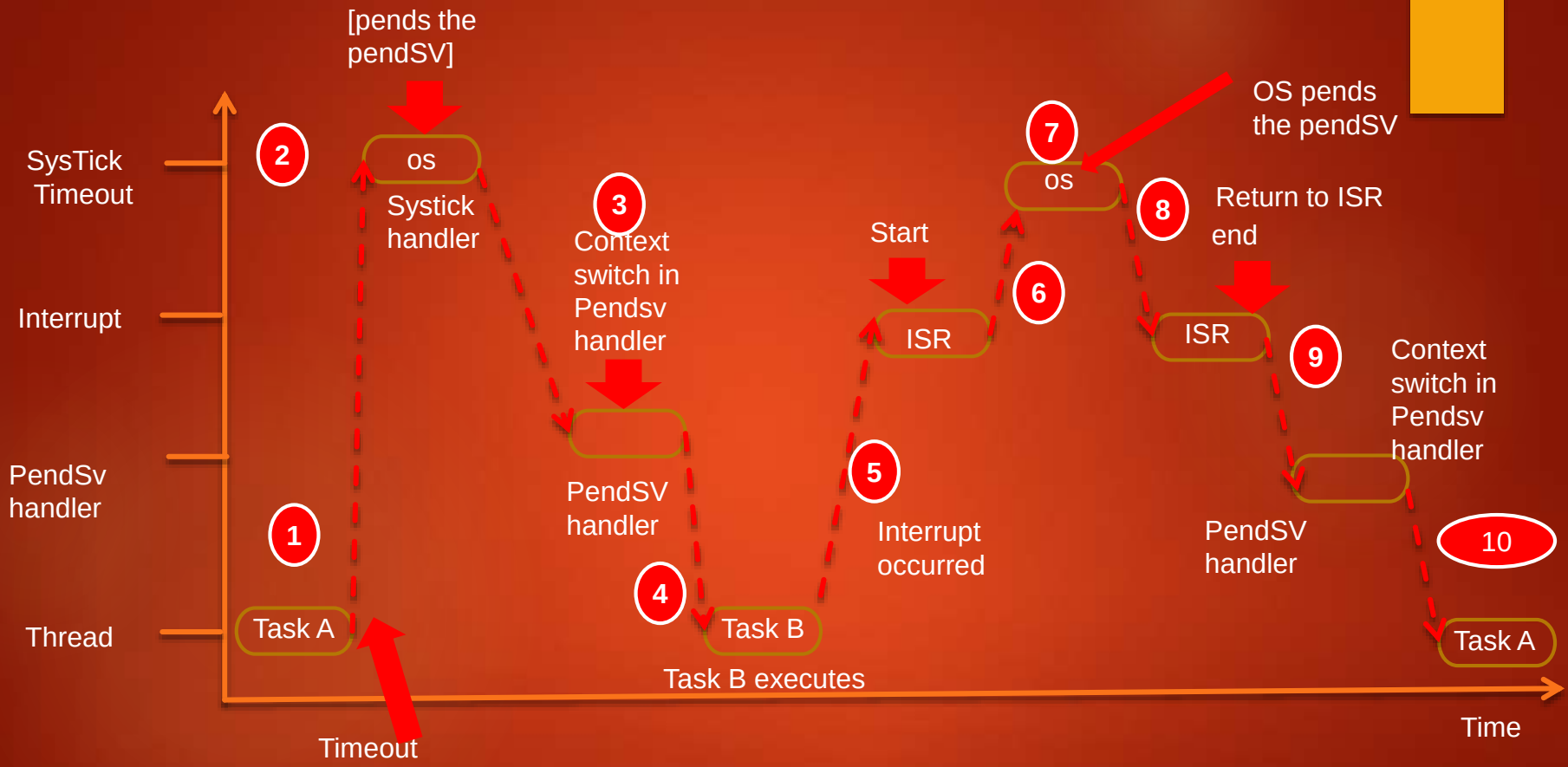


Pendsv In Context Switching

- In typical OS design, the context switching operation is carried out inside the PendSV exception handler.
- using PendSV in context switching will be more efficient in a interrupt noisy environment.
- In a interrupt noisy environment we need to delay the context switching until all IRQ are executed.

Pendsv In Context Switching

- To do this , the PendSV is programmed as the lowest priority exception.
- If the OS decides that the context switching is needed, it sets the pending status of the PendSV , and carries out the context switching within the PendSV exception.



Scenario of PendSV in context switching

Offloading Interrupt processing using PendSV

If a higher priority handlers doing time consuming work, then the other lower priority interrupts will suffer and systems responsiveness may reduce .

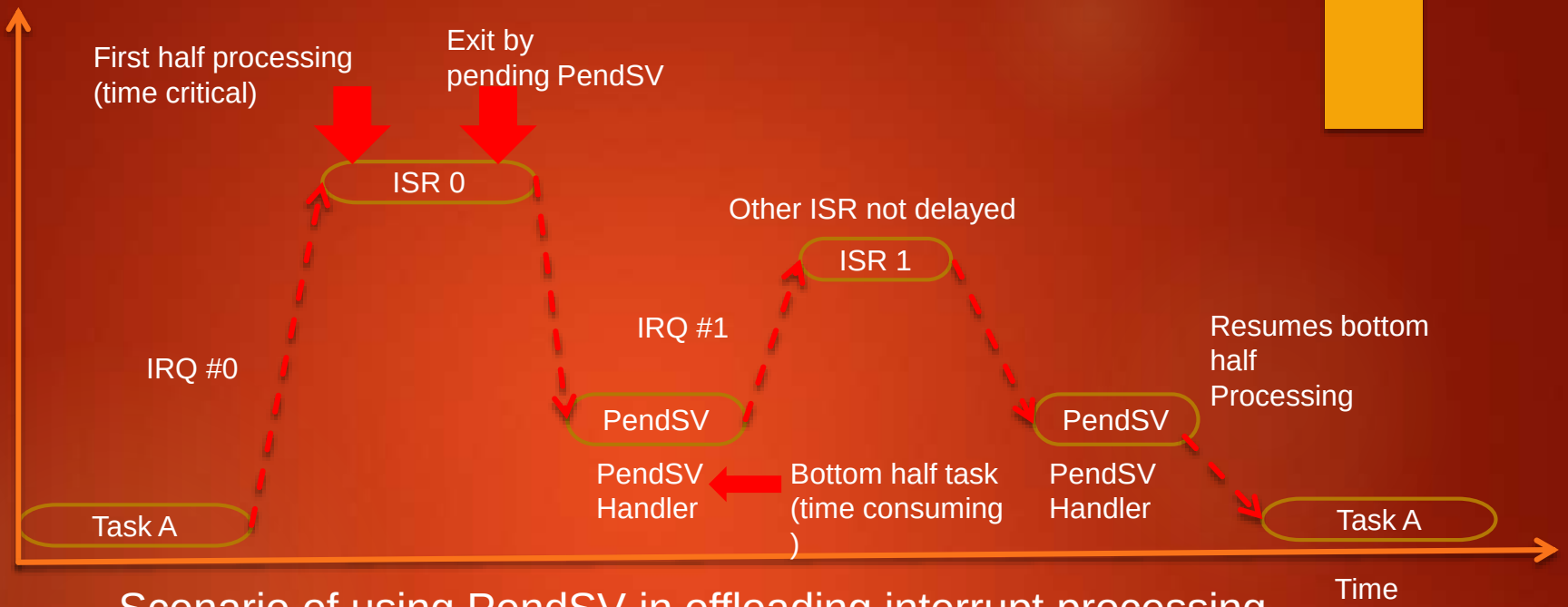
Offloading Interrupt Processing Using Pendsv

Typically interrupts are serviced in 2 halves.

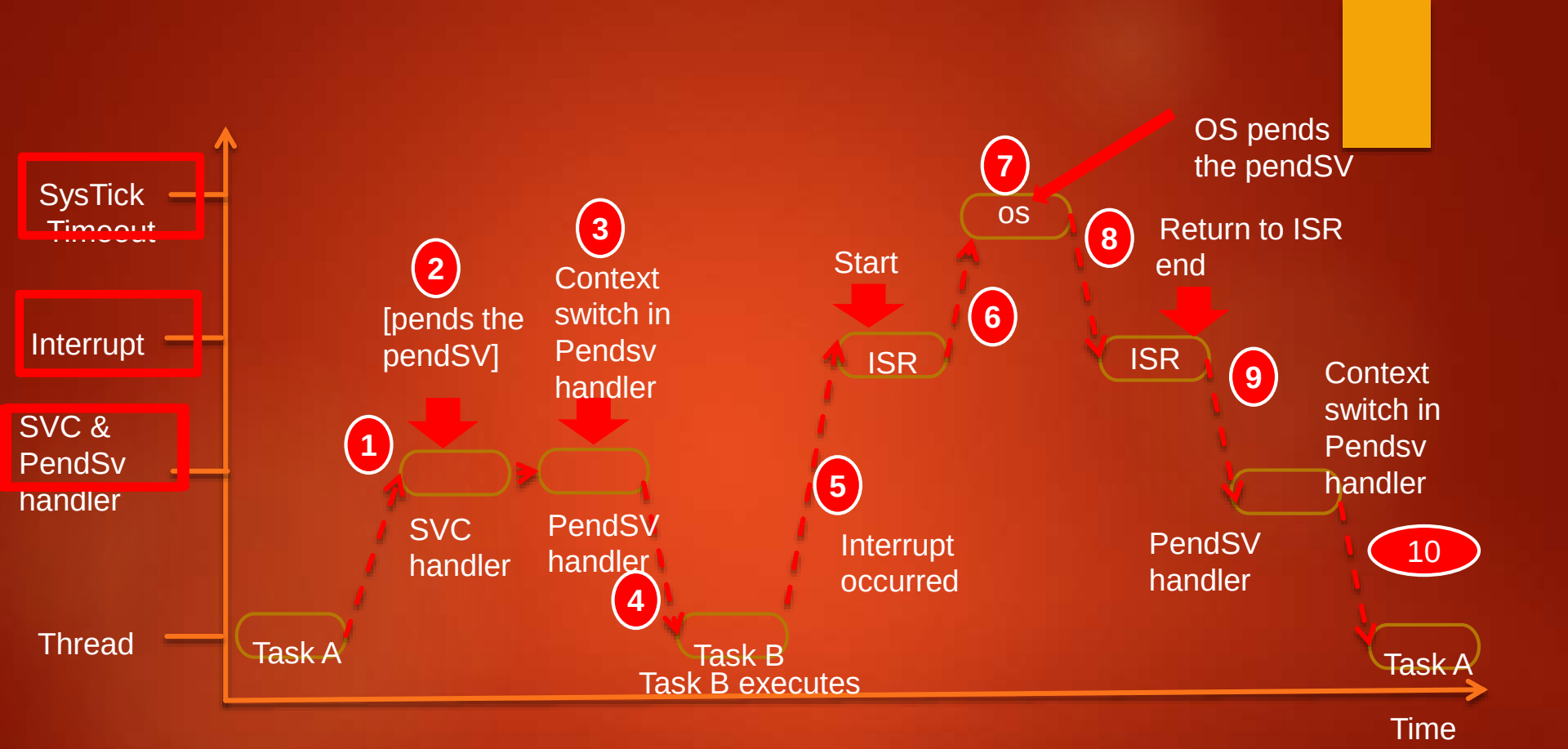
1) The first half is the time critical part that needs to be executed as a part of ISR.

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.



Scenario of using PendSV in offloading interrupt processing



Scenario of PendSV in context switching

My Udemy Online Courses on Microcontroller and RTOS programming

We have several online video courses on ,

Microcontroller Programming,

RTOS Programming

Microcontroller programming with Driver Development

Embedded linux

Microcontroller DMA Programming

Mirocontroller Bootloader development

Please visit www.fastbitlab.com to enroll for the courses.

Or you can check my udemy profile here :

www.fastbitlab.com

Total students	Courses	Reviews
16,226	7	4,245