

# Interrupts (1)

Lecture 9

Yeongpil Cho

Hanyang University

# Topics

- Interrupt Handling Basics
- Priority Management

# Interrupt Handling Basics

# Interrupts

- Motivations
  - Inform processors of some external events timely
    - Polling vs Interrupt
  - Implement multi-tasking with priority support

# Polling vs Interrupt

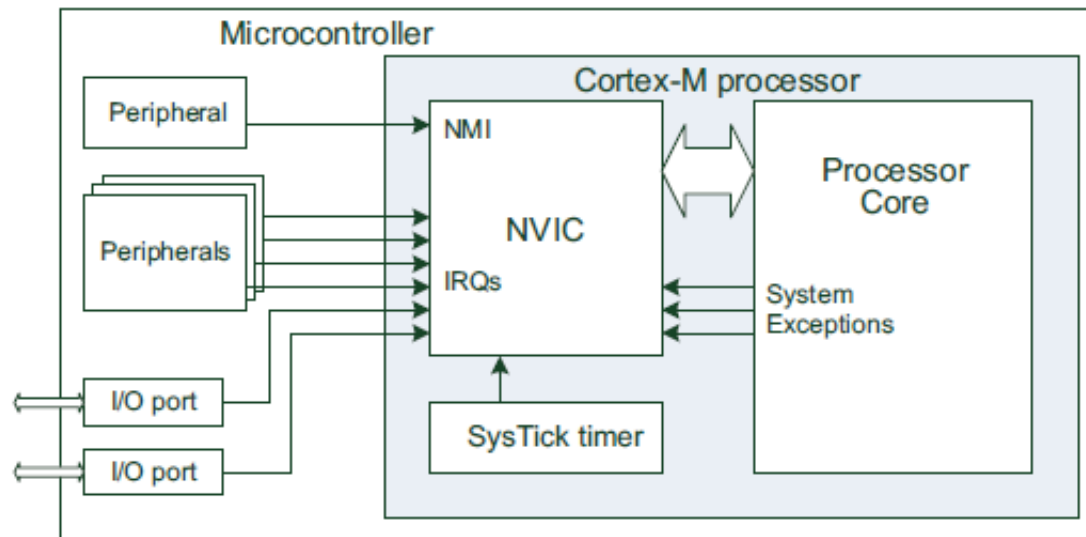
- Polling:
  - You pick up the phone every three seconds to check whether you are getting a call.
- Interrupt:
  - Do whatever you should do and pick up the phone when it rings.



www.Vecto.rs · 22705

# ARMv7-M Interrupt Handling

- One Non-Maskable Interrupt (NMI) supported
- Up to 512 (496 interrupts and 16 exceptions) prioritizable interrupts/exceptions supported
  - Interrupts can be masked
  - Implementation option selects number of interrupts supported
- Nested Vectored Interrupt Controller (NVIC) is tightly coupled with processor core



# Interrupt Service Routine Vector Table

- First entry contains initial Main SP
- All other entries are addresses for
  - exception/interrupt handlers
  - Must always have LSBit = 1 (for Thumb)
- Table can be relocated
  - Use Vector Table Offset Register
  - Still require minimal table entries at 0x0 for booting the core
- Table can be generated using C code

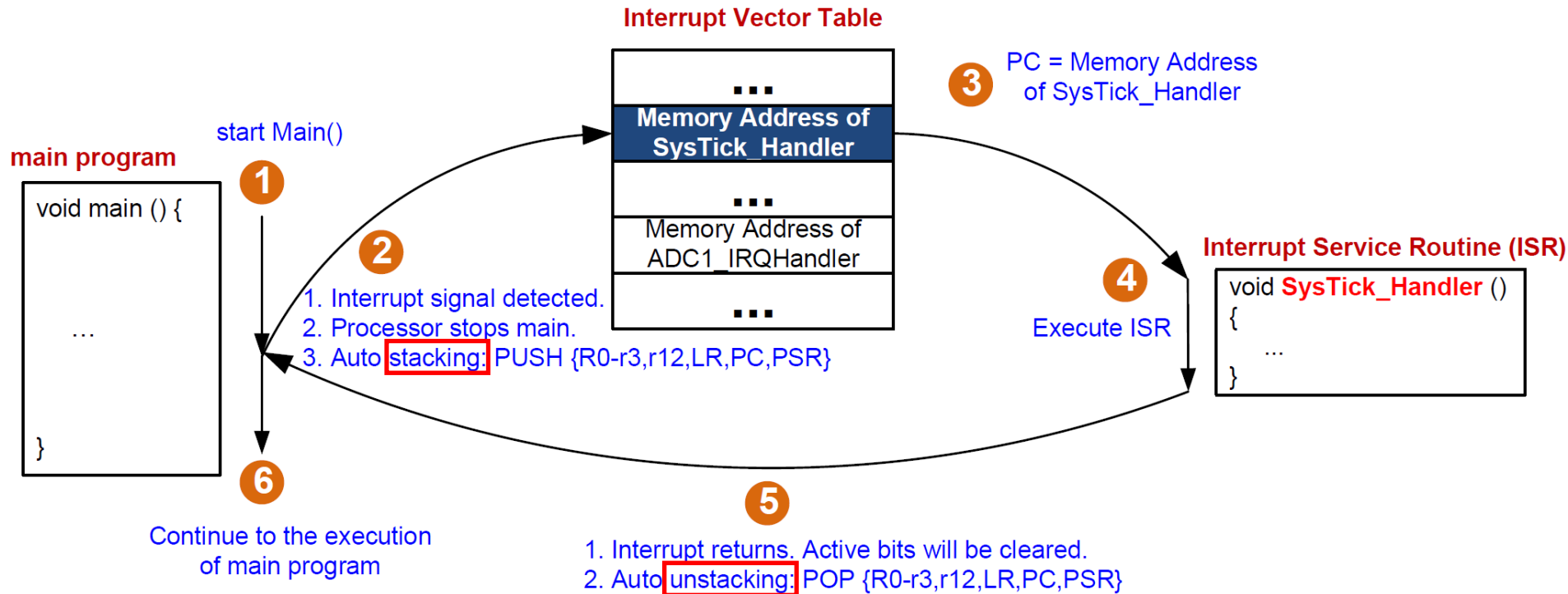
Address	
$0x40 + 4*N$	<b>External N</b>
...	...
0x40	<b>External 0</b>
0x3C	<b>SysTick</b>
0x38	<b>PendSV</b>
0x34	<b>Reserved</b>
0x30	<b>Debug Monitor</b>
0x2C	<b>SVC</b>
0x1C to 0x28	<b>Reserved (x4)</b>
0x18	<b>Usage Fault</b>
0x14	<b>Bus Fault</b>
0x10	<b>Mem Manage Fault</b>
0x0C	<b>Hard Fault</b>
0x08	<b>NMI</b>
0x04	<b>Reset</b>
0x00	<b>Initial Main SP</b>

# Interrupt Service Routine Vector Table of Cortex-M processors

	Exception	Name	Priority	Descriptions
Fault Mode & Start-up Handlers	1	Reset	-3 (Highest)	Reset
	2	NMI	-2	Non-Maskable Interrupt
	3	Hard Fault	-1	Default fault if other handler not implemented
	4	MemManage Fault	Programmable	MPU violation or access to illegal locations
	5	Bus Fault	Programmable	Fault if AHB interface receives error
	6	Usage Fault	Programmable	Exceptions due to program errors
System Handlers	11	SVCall	Programmable	System Service call
	12	Debug Monitor	Programmable	Break points, watch points, external debug
	14	PendSV	Programmable	Pendable Service request for System Device
	15	Systick	Programmable	System Tick Timer
Custom Handlers	16	Interrupt #0	Programmable	External Interrupt #0
	...	...	...	...
	...	...	...	...
	...	...	...	...
	255	Interrupt #239	Programmable	External Interrupt #239

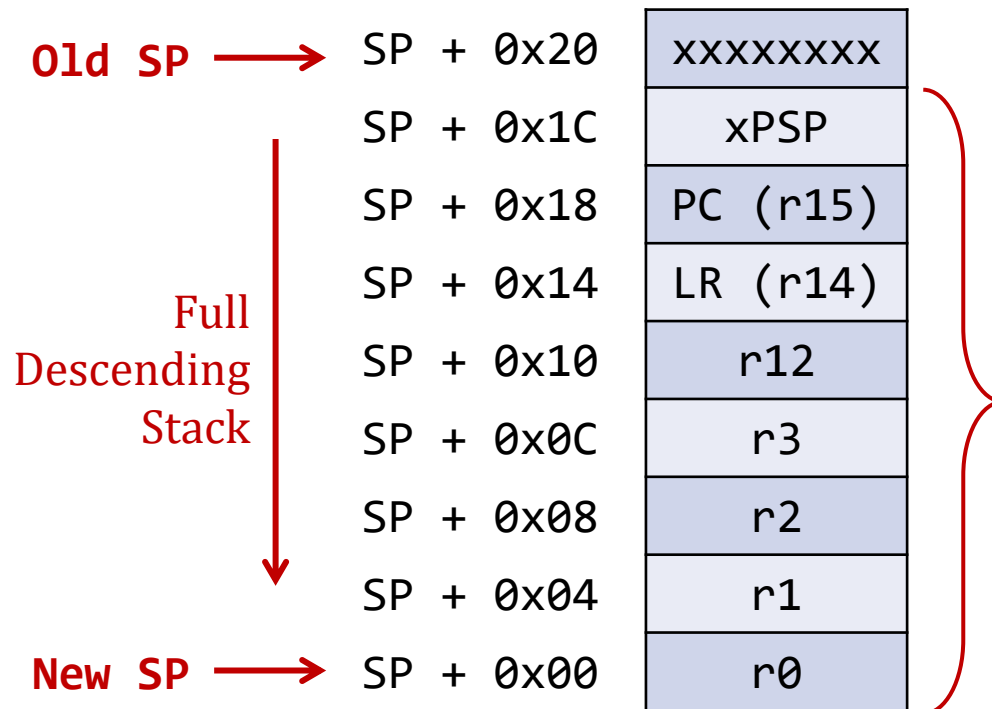


# Interrupt Handling Process



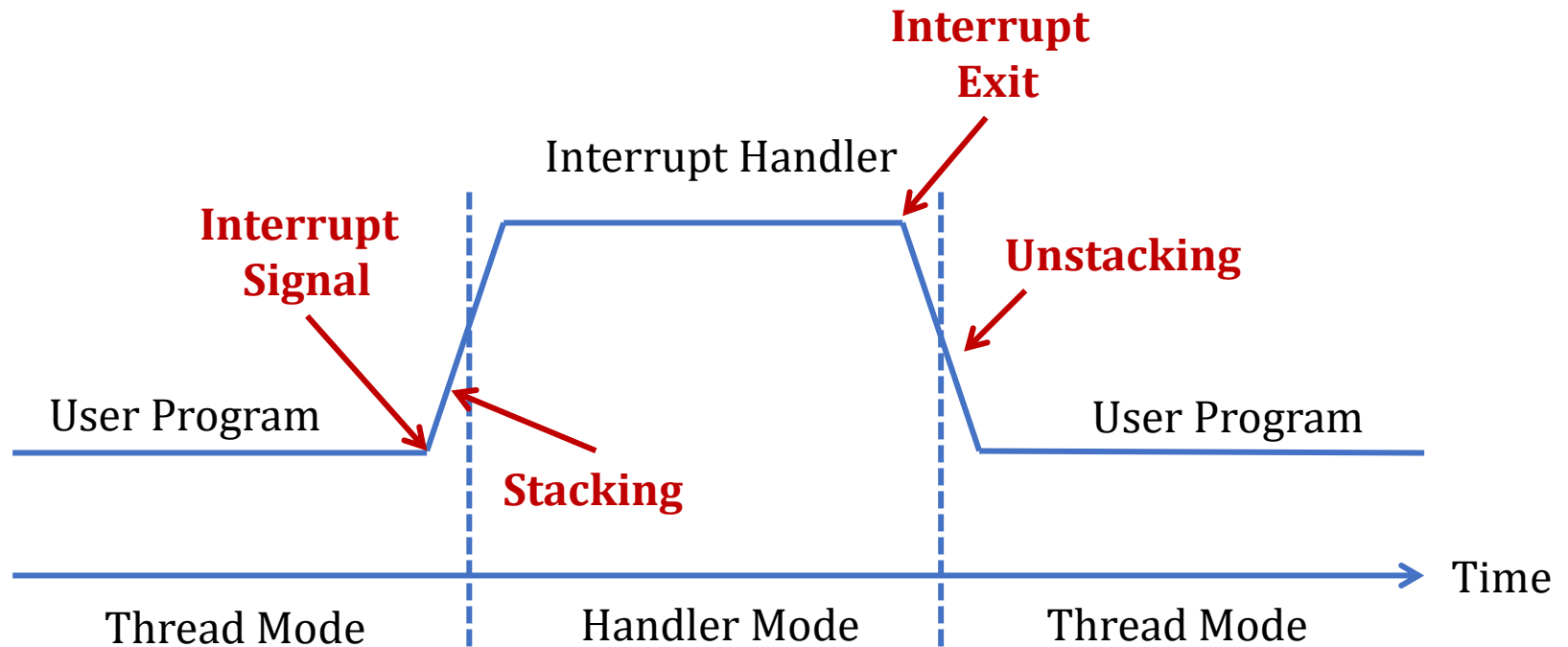
# Stacking & Unstacking

- Current mode (either Thread mode or Handler mode)'s stack is used for stacking/unstacking.

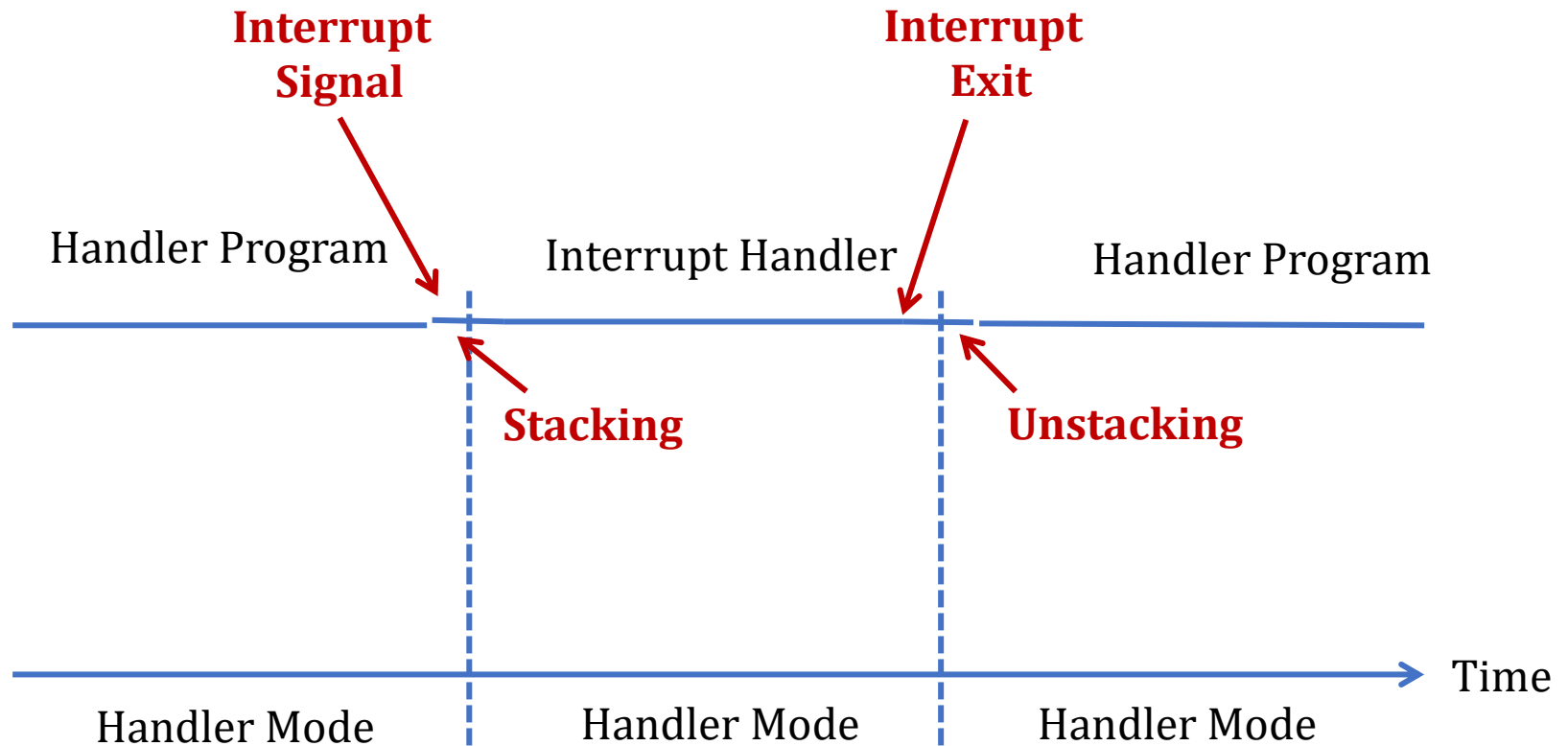


- Stacking:** The processor automatically pushes these eight registers into the stack before an interrupt handler starts
- Unstacking:** The processor automatically pops these eight registers out of the stack when an interrupt handler exits.

# Stacking & Unstacking



# Stacking & Unstacking

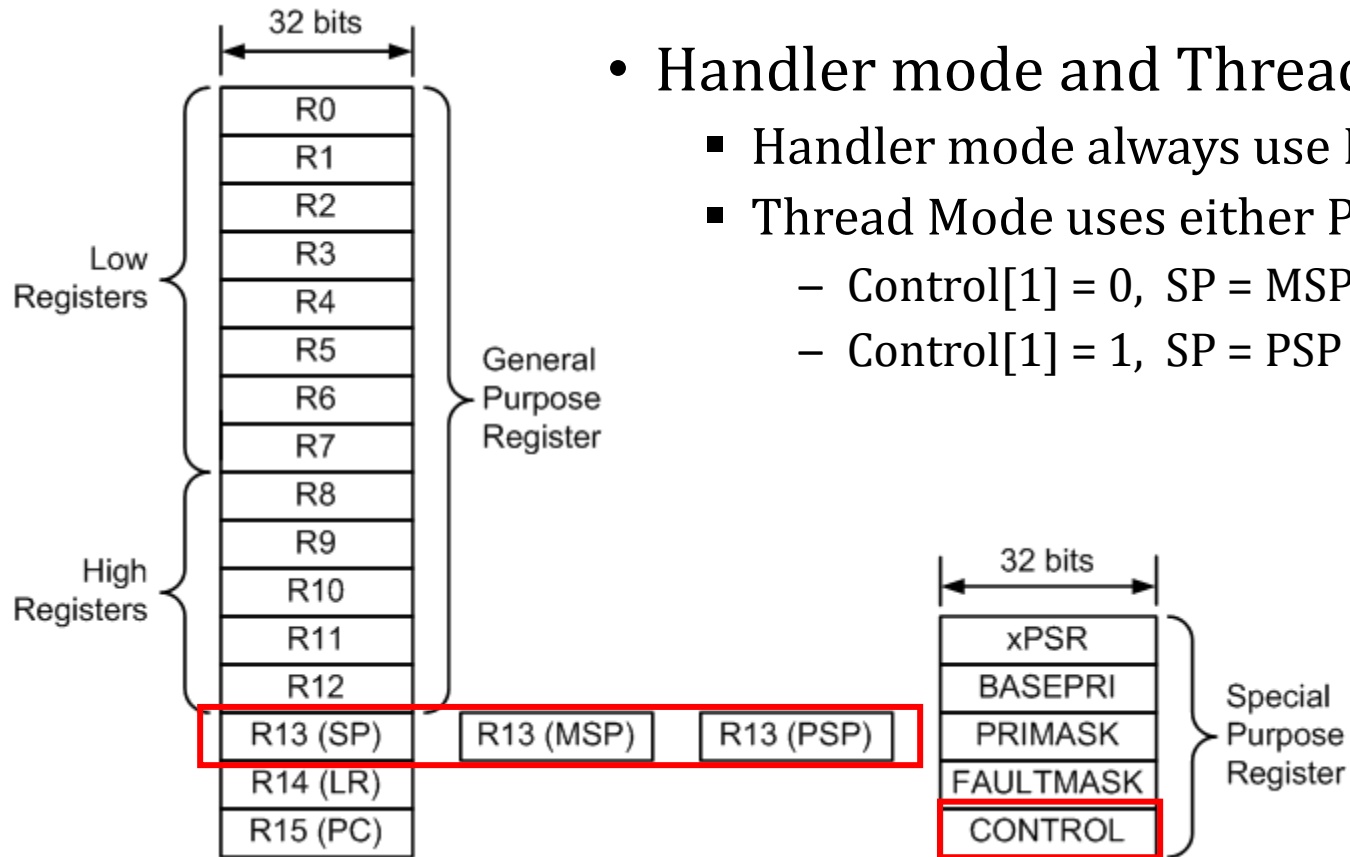


# Exception Exits

- When the **EXC\_RETURN** is loaded into the PC at the end of the exception handler execution, the processor performs an exception return sequence
- EXC\_RETURN is generated and set to **LR** by processors when an exception arises.
- There are three ways to trigger the interrupt return sequence:

Return Instruction	Description
<code>BX &lt;reg&gt;</code>	If the EXC_RETURN value is still in LR, we can use the <i>BX LR</i> instruction to perform the interrupt return.
<code>POP {PC}, or POP {..., PC}</code>	Very often the value of LR is pushed to the stack after entering the exception handler. We can use the POP instruction, either a single POP or multiple POPs, to put the EXC_RETURN value to the program counter. This will cause the processor to perform the interrupt return.
<code>LDR, or LDM</code>	It is possible to produce an interrupt return using the LDR instruction with PC as the destination register.

# Registers



- Handler mode and Thread mode
  - Handler mode always use MSP
  - Thread Mode uses either PSP or MSP
    - Control[1] = 0, SP = MSP (default)
    - Control[1] = 1, SP = PSP

**MSP:** Main Stack Pointer

**PSP:** Process Stack Pointer

## Register values in an interrupt service routine

LR = 0xFFFFFFFF9

SP = MSP

ISR always in handler mode.

The screenshot shows a debugger interface with the following components:

- Registers Window:** Displays the state of various registers. The 'Core' registers R0-R12 and xPSR are listed. R4, R5, R13 (SP), R14 (LR), and R15 (PC) are highlighted in blue. R13 (SP) has a value of 0x200005C8, and R14 (LR) has a value of 0xFFFFFFFF9. The 'Banked' registers show MSP at 0x200005C8. The 'System' registers show BASEPRI, PRIMASK, FAULTMASK, and CONTROL. The 'Internal' registers show Mode (Handler), Privilege (Privileged), Stack (MSP), States (2740), and Sec (0.00034250).
- Disassembly Window:** Shows the assembly code for the SVC\_Handler procedure. The code is as follows:

```
36: SVC_Handler PROC
37:         EXPORT SVC_Handler
0x0800017A BD30      POP      {r4-r5,pc}
38:         CPSID    I
0x0800017C B672      PUSH     {r4-r8,lr}
39:         PUSH     {r4-r8,lr}
0x0800017E E92D41F0  PUSH     {r4-r8,lr}
40:         LDR      r7, [sp, #0x14]
0x08000182 9F05      LDR      r7, [sp, #0x14]
41:         TST      r7, #0x04
0x08000184 F0170F04  TST      r7, #0x04
42:         ITE      EQ
0x08000188 BF0C      ITE      EQ
```
- Logic Analyzer Window:** Shows the logic analyzer data for the SVC\_Handler procedure. The code is as follows:

```
33:         POP      {r4-r5,pc}
34:         ENDP
35:
36: SVC_Handler PROC
37:         EXPORT SVC_Handler
38:         CPSID    I
39:         PUSH     {r4-r8,lr}
40:         LDR      r7, [sp, #0x14]
41:         TST      r7, #0x04
42:         ITE      EQ
43:         MRSEQ    r4, msp
44:         MRSNE    r4, psp
45:         LDR      r4 = 0x080001A3
46:         LDR      r6, [r4, #0x04]
47:         MOV      r0, r6
48:         BLX      r5
49:         POP      {r4-r8,lr}
50:         CPSIE    I
51:         BX       lr
52:         ENDP
53:
```

# Which stack to use when exiting an interrupt?

- EXC\_RETURN indicates processor mode and stack type to be activated when exiting an interrupt

No FP extension:

<b>EXC_RETURN</b>	<b>Return Mode</b>	<b>Return Stack</b>
<b>0xFFFFFFFF1</b>	Handler	SP = MSP
<b>0xFFFFFFFF9</b>	Thread	SP = MSP
<b>0xFFFFFFF9D</b>	Thread	SP = PSP

With FP extension:

<b>EXC_RETURN</b>	<b>Return Mode</b>	<b>Return Stack</b>
<b>0xFFFFFFE1</b>	Handler	SP = MSP
<b>0xFFFFFFE9</b>	Thread	SP = MSP
<b>0xFFFFFFFED</b>	Thread	SP = PSP



# Interrupt: Stacking & Unstacking

```

main PROC
...
MOV r3, #0
...
ENDP

SysTick_Handler PROC
EXPORT SysTick_Handler
ADD r3, #1
ADD r4, #1
BX lr
ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	MSP
R14(LR)	0x08001000
R15(PC)	0x08000044
xPSR	0x21000000
MSP	0x20000200
PSP	0x00000000

xxxxxxxx	0x20000200
	0x200001FC
	0x200001F8
	0x200001F4
	0x200001F0
	0x200001EC
	0x200001E8
	0x200001E4
	0x200001E0
	0x200001DC
	0x200001D8
	0x200001D4
	0x200001D0
	0x200001CF

# Interrupt: Stacking & Unstacking

Suppose SysTick interrupt occurs when PC = 0x08000044

```
__main PROC
...
MOV r3, #0
...
ENDP

SysTick_Handler PROC
EXPORT SysTick_Handler
ADD r3, #1
ADD r4, #1
BX lr
ENDP
```

addr = 0x08000044

addr = 0x0800001C

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	MSP
R14(LR)	0x08001000
R15(PC)	0x08000044

xPSR	0x21000000
MSP	0x20000200
PSP	0x00000000

xxxxxxxx	0x20000200
	0x200001FC
	0x200001F8
	0x200001F4
	0x200001F0
	0x200001EC
	0x200001E8
	0x200001E4
	0x200001E0
	0x200001DC
	0x200001D8
	0x200001D4
	0x200001D0
	0x200001CF

Memory

# Interrupt: Stacking & Unstacking

## STACKING

```

__main PROC
...
    MOV r3, #0
...
ENDP

SysTick_Handler PROC
    EXPORT SysTick_Handler
    ADD r3, #1
    ADD r4, #1
    BX lr
ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

R0	0		xxxxxxx	0x20000200
R1	1	xPSR	0x21000000	0x200001FC
R2	2	PC	0x08000044	0x200001F8
R3	3	LR	0x08001000	0x200001F4
R4	4	R12	12	0x200001F0
R12	12	R3	3	0x200001EC
R13(SP)	MSP	R2	2	0x200001E8
R14(LR)	0xFFFFFFFF9	R1	1	0x200001E4
R15(PC)	0x0800001C	R0	0	0x200001E0
				0x200001DC
xPSR	0x21000000			0x200001D8
MSP	0x200001E0			0x200001D4
PSP	0x00000000			0x200001D0
				0x200001CF

Memory

# Interrupt: Stacking & Unstacking

## STACKING

```

__main PROC
...
    MOV r3, #0
...
ENDP

SysTick_Handler PROC
    EXPORT SysTick_Handler
    ADD r3, #1
    ADD r4, #1
    BX lr
ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

**LR = 0xFFFFFFFF9 to indicate MSP is used.**

R0	0		xxxxxxx	0x20000200
R1	1	xPSR	0x21000000	0x200001FC
R2	2	PC	0x08000044	0x200001F8
R3	3	LR	0x08001000	0x200001F4
R4	4	R12	12	0x200001F0
R12	12	R3	3	0x200001EC
R13(SP)	MSP	R2	2	0x200001E8
R14(LR)	0xFFFFFFFF9	R1	1	0x200001E4
R15(PC)	0x0800001C	R0	0	0x200001E0
				0x200001DC
				0x200001D8
				0x200001D4
				0x200001D0
				0x200001CF

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

Memory

# Interrupt: Stacking & Unstacking

```

__main PROC
...
MOV r3, #0
...
ENDP

SysTick_Handler PROC
EXPORT SysTick_Handler
ADD r3, #1
ADD r4, #1
BX lr
ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

R0	0	xPSR	xxxxxxx	0x20000200
R1	1		0x21000000	0x200001FC
R2	2	PC	0x08000044	0x200001F8
R3	4	LR	0x08001000	0x200001F4
R4	4	R12	12	0x200001F0
R12	12	R3	3	0x200001EC
R13(SP)	MSP	R2	2	0x200001E8
R14(LR)	0xFFFFFFFF9	R1	1	0x200001E4
R15(PC)	0x0800001C	R0	0	0x200001E0
				0x200001DC
xPSR	0x21000000			0x200001D8
MSP	0x200001E0			0x200001D4
PSP	0x00000000			0x200001D0
				0x200001CF

Memory

# Interrupt: Stacking & Unstacking

```

__main PROC
...
MOV r3, #0
...
ENDP

SysTick_Handler PROC
EXPORT SysTick_Handler
ADD r3, #1
ADD r4, #1
BX lr
ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

R0	0
R1	1
R2	2
R3	4
R4	5
R12	12
R13(SP)	MSP
R14(LR)	0xFFFFFFFF9
R15(PC)	0x08000020

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

xPSR	xxxxxxxx	0x20000200
PC	0x21000000	0x200001FC
LR	0x08000044	0x200001F8
R12	0x08001000	0x200001F4
R3	12	0x200001F0
R2	3	0x200001EC
R1	2	0x200001E8
R0	1	0x200001E4
	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

Memory

# Interrupt: Stacking & Unstacking

```

__main PROC
...
MOV r3, #0
...
ENDP

SysTick_Handler PROC
EXPORT SysTick_Handler
ADD r3, #1
ADD r4, #1
BX lr
ENDP

```

addr = 0x08000044

addr = 0x0800001C

**LR = 0xFFFFFFFF9 to indicate MSP is used.**

R0	0	xPSR	xxxxxxx	0x20000200
R1	1	PC	0x21000000	0x200001FC
R2	2	LR	0x08000044	0x200001F8
R3	4	R12	0x08001000	0x200001F4
R4	5	R3	12	0x200001F0
R12	12	R2	3	0x200001EC
R13(SP)	MSP	R1	2	0x200001E8
R14(LR)	0xFFFFFFFF9	R0	1	0x200001E4
R15(PC)	0x08000024		0	0x200001E0
				0x200001DC
				0x200001D8
				0x200001D4
				0x200001D0
				0x200001CF

Memory

# Interrupt: Stacking & Unstacking

## UNSTACKING

```

__main PROC
...
    MOV r3, #0
...
ENDP

SysTick_Handler PROC
    EXPORT SysTick_Handler
    ADD r3, #1
    ADD r4, #1
    BX lr
    ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

**LR = 0xFFFFFFFF9 to indicate MSP is used.**

R0	0	xPSR	xxxxxxx	0x20000200
R1	1		0x21000000	0x200001FC
R2	2	PC	0x08000044	0x200001F8
R3	4	LR	0x08001000	0x200001F4
R4	5	R12	12	0x200001F0
R12	12	R3	3	0x200001EC
R13(SP)	MSP	R2	2	0x200001E8
R14(LR)	0xFFFFFFFF9	R1	1	0x200001E4
R15(PC)	0x08000024	R0	0	0x200001E0
				0x200001DC
				0x200001D8
xPSR	0x21000000			0x200001D4
MSP	0x200001E0			0x200001D0
PSP	0x00000000			0x200001CF

Memory



# Interrupt: Stacking & Unstacking

## UNSTACKING

```
__main PROC
...
MOV r3, #0
...
ENDP

SysTick_Handler PROC
EXPORT SysTick_Handler
ADD r3, #1
ADD r4, #1
BX lr
ENDP
```

addr = 0x08000044

addr = 0x0800001C

**Note the new value of R3 is lost!!!**

R0	0
R1	1
R2	2
R3	3
R4	5
R12	12
R13(SP)	MSP
R14(LR)	0x08001000
R15(PC)	0x08000044
xPSR	0x21000000
MSP	0x20000200
PSP	0x00000000

xxxxxxxx	0x20000200
	0x200001FC
	0x200001F8
	0x200001F4
	0x200001F0
	0x200001EC
	0x200001E8
	0x200001E4
	0x200001E0
	0x200001DC
	0x200001D8
	0x200001D4
	0x200001D0
	0x200001CF

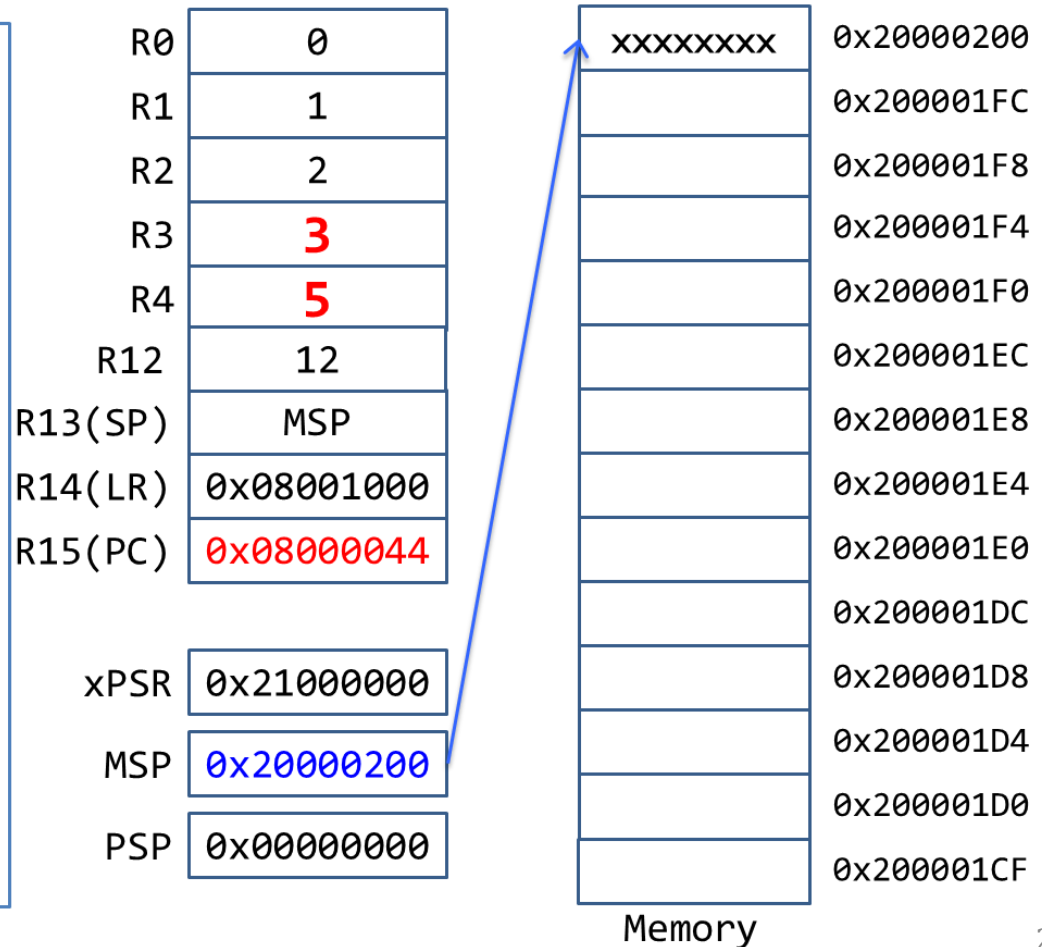
Memory

# Interrupt: Stacking & Unstacking

```
__main PROC
...
MOV r3, #0
...
ENDP

SysTick_Handler PROC
EXPORT SysTick_Handler
ADD r3, #1
ADD r4, #1
BX lr
ENDP
```

**The Main program resumes!!!**



# Interrupt: Stacking & Unstacking

```

__main PROC
...
MOV r3, #0
...
ENDP

SysTick_Handler PROC
EXPORT SysTick_Handler
ADD r4, #1 0x0800001C
BL sine 0x08000020
BX lr 0x08000024
ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	MSP
R14(LR)	0x08001000
R15(PC)	0x08000044
xPSR	0x21000000
MSP	0x20000200
PSP	0x00000000

xxxxxxxx	0x20000200
	0x200001FC
	0x200001F8
	0x200001F4
	0x200001F0
	0x200001EC
	0x200001E8
	0x200001E4
	0x200001E0
	0x200001DC
	0x200001D8
	0x200001D4
	0x200001D0
	0x200001CF

Memory

# Interrupt: Stacking & Unstacking

## STACKING

```

__main PROC
...
MOV r3, #0
...
ENDP

SysTick_Handler PROC
EXPORT SysTick_Handler
ADD r4, #1 0x0800001C
BL sine 0x08000020
BX lr 0x08000024
ENDP
    
```

**LR = 0xFFFFFFFF9 to indicate MSP is used.**

R0	0		xxxxxxx	0x20000200
R1	1	xPSR	0x21000000	0x200001FC
R2	2	PC	0x08000044	0x200001F8
R3	3	SP	0x20000200	0x200001F4
R4	4	LR	0x08001000	0x200001F0
R12	12	R3	3	0x200001EC
R13(SP)	MSP	R2	2	0x200001E8
R14(LR)	0xFFFFFFFF9	R1	1	0x200001E4
R15(PC)	0x0800001C	R0	0	0x200001E0
				0x200001DC
				0x200001D8
				0x200001D4
				0x200001D0
				0x200001CF

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

Memory

# Interrupt: Stacking & Unstacking

```

__main PROC
...
MOV r3, #0
...
ENDP

SysTick_Handler PROC
EXPORT SysTick_Handler
ADD r4, #1 0x0800001C
BL sine 0x08000020
BX lr 0x08000024
ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

**LR = 0xFFFFFFFF9 to indicate MSP is used.**

R0	0	xPSR	xxxxxxx	0x20000200
R1	1		0x21000000	0x200001FC
R2	2	PC	0x08000044	0x200001F8
R3	3	SP	0x20000200	0x200001F4
R4	5	LR	0x08001000	0x200001F0
R12	12	R3	3	0x200001EC
R13(SP)	MSP	R2	2	0x200001E8
R14(LR)	0xFFFFFFFF9	R1	1	0x200001E4
R15(PC)	0x08000020	R0	0	0x200001E0
				0x200001DC
xPSR	0x21000000			0x200001D8
MSP	0x200001E0			0x200001D4
PSP	0x00000000			0x200001D0
				0x200001CF

Memory

# Interrupt: Stacking & Unstacking

```

__main PROC
...
    MOV r3, #0
...
ENDP

SysTick_Handler PROC
    EXPORT SysTick_Handler
    ADD r4, #1
    BL sine
    BX lr
ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

BL sine

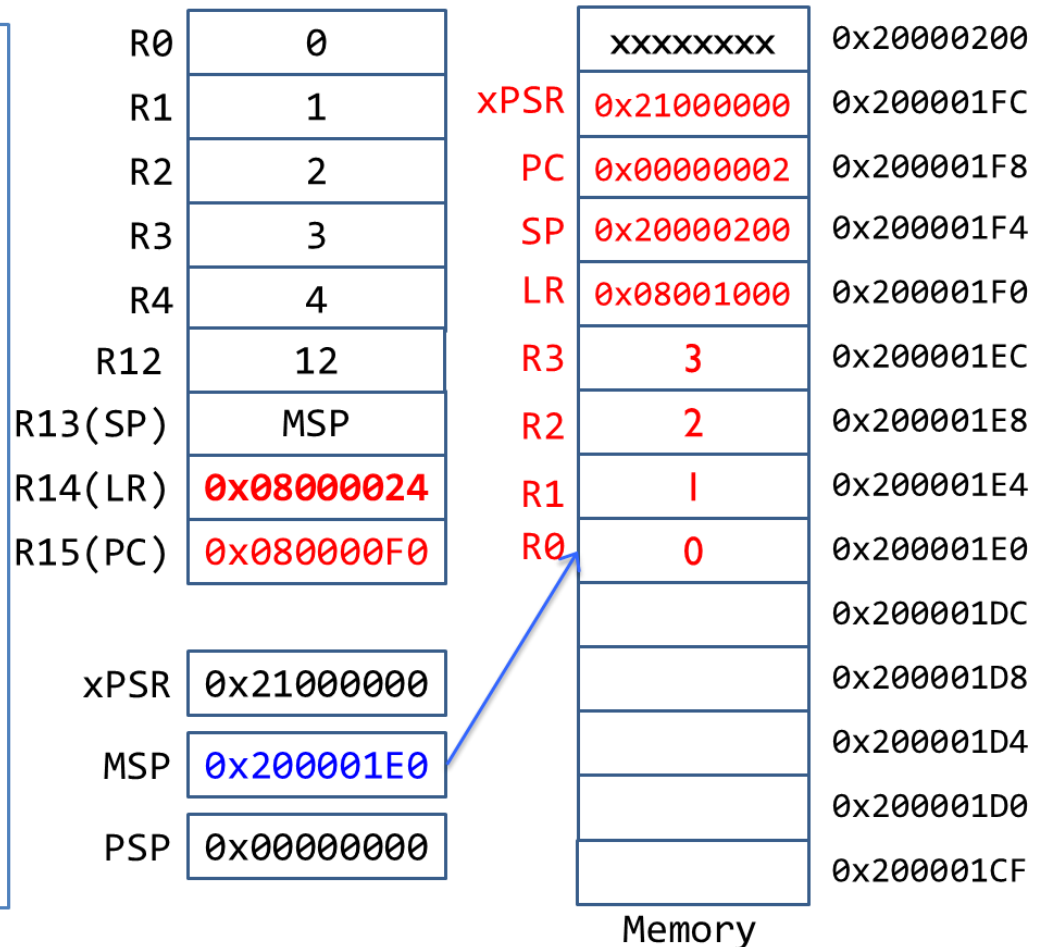
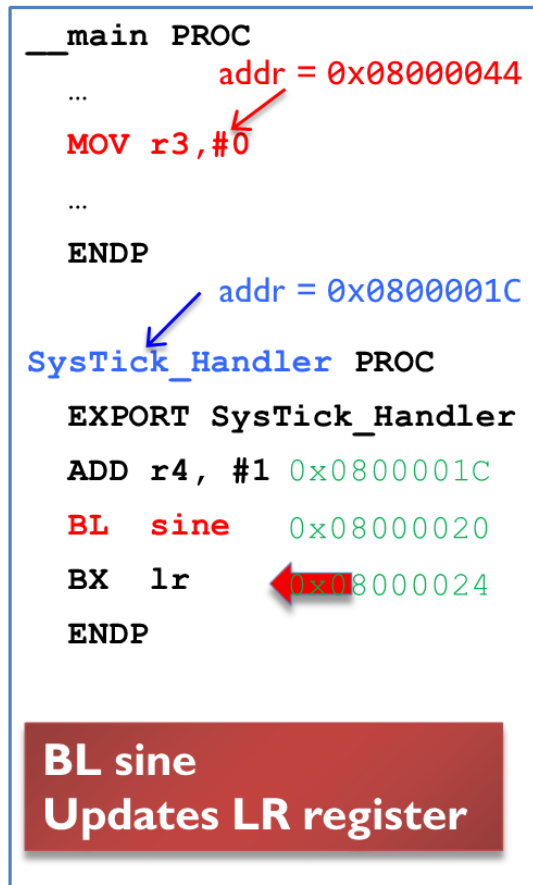
Updates LR register

R0	0	xPSR	xxxxxxx	0x20000200
R1	1		0x21000000	0x200001FC
R2	2	PC	0x08000044	0x200001F8
R3	3	SP	0x20000200	0x200001F4
R4	4	LR	0x08001000	0x200001F0
R12	12	R3	3	0x200001EC
R13(SP)	MSP	R2	2	0x200001E8
R14(LR)	0x08000024	R1	1	0x200001E4
R15(PC)	0x080000F0	R0	0	0x200001E0
				0x200001DC
				0x200001D8
				0x200001D4
				0x200001D0
				0x200001CF

Memory

Assume sine() is located at 0x08000024.

# Interrupt: Stacking & Unstacking



# Interrupt: Stacking & Unstacking

**UNSTACKING  
won't occur!**

```

__main PROC
...
    MOV r3, #0
...
ENDP

SysTick_Handler PROC
    EXPORT SysTick_Handler
    ADD r4, #1
    BL sine
    BX lr
ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

0x08000024

**BL sine  
Updates LR register**

R0	0		xxxxxxx	0x20000200
R1	1	xPSR	0x21000000	0x200001FC
R2	2	PC	0x00000002	0x200001F8
R3	3	SP	0x20000200	0x200001F4
R4	4	LR	0x08001000	0x200001F0
R12	12	R3	3	0x200001EC
R13(SP)	MSP	R2	2	0x200001E8
R14(LR)	0x08000024	R1	1	0x200001E4
R15(PC)	0x080000F0	R0	0	0x200001E0
				0x200001DC
				0x200001D8
				0x200001D4
				0x200001D0
				0x200001CF

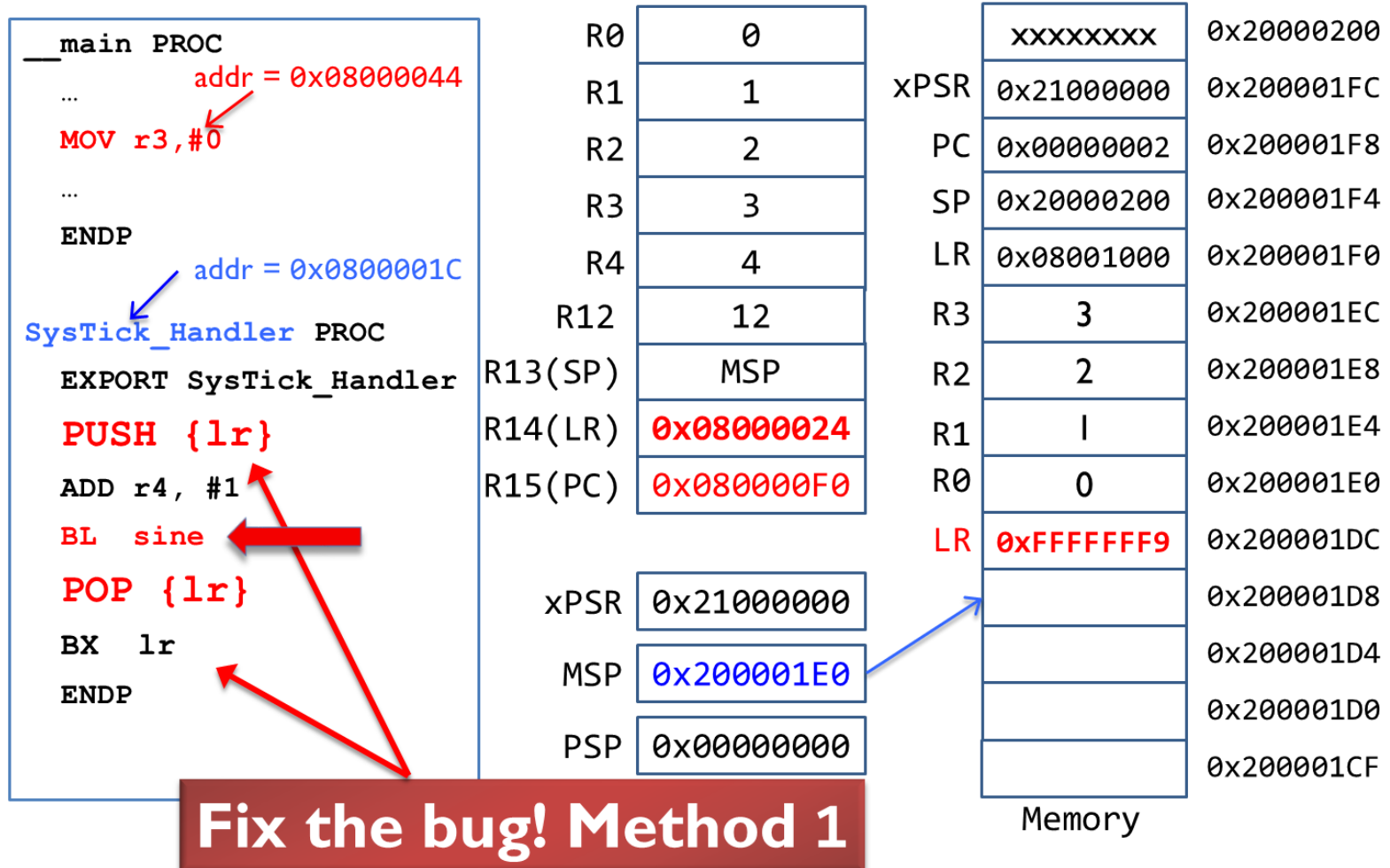
  

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

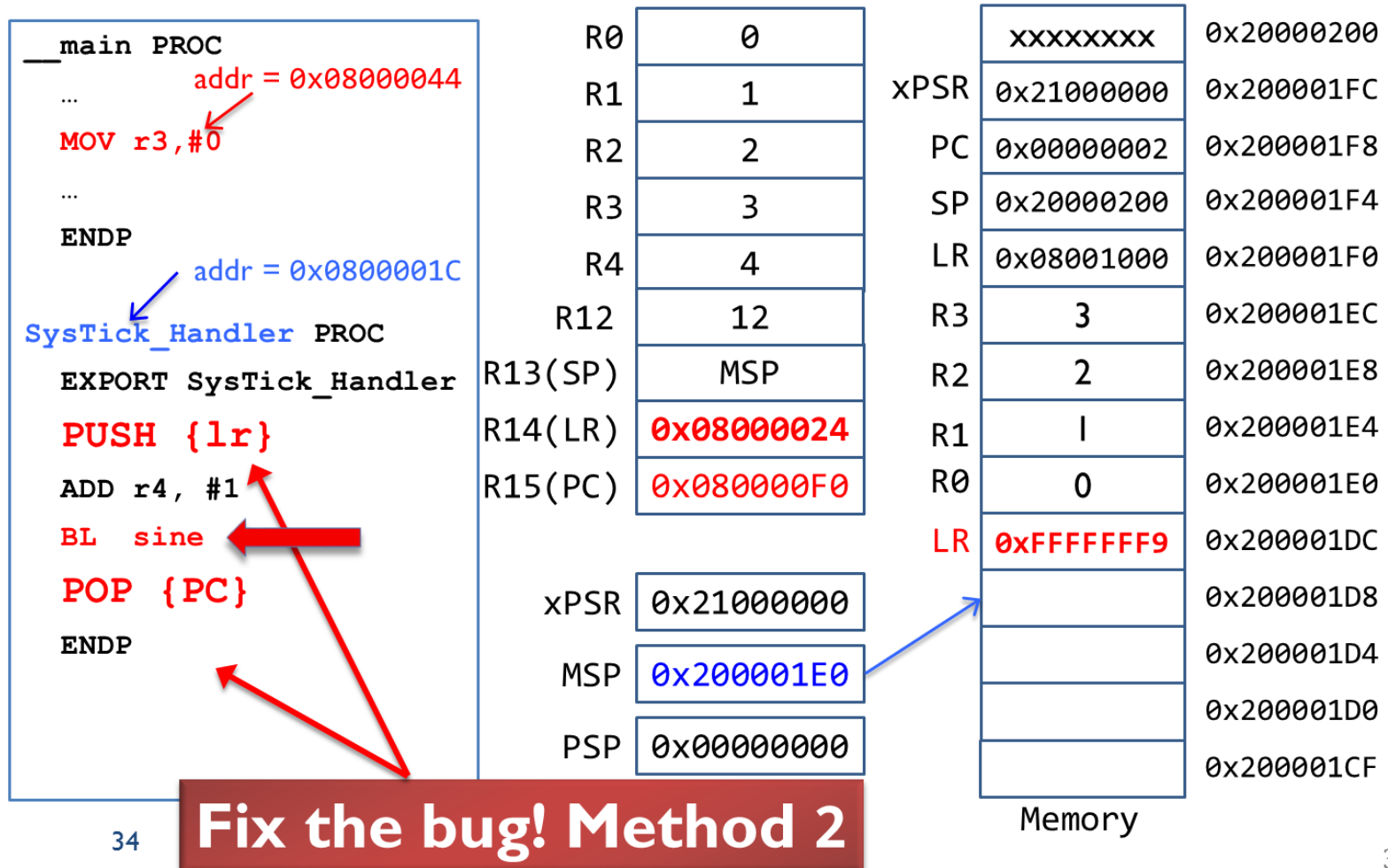
Memory



# Interrupt: Stacking & Unstacking



# Interrupt: Stacking & Unstacking



# Interrupt: Stacking & Unstacking

```

__main PROC
...
    MOV r3, #0
...
ENDP

SysTick_Handler PROC
    EXPORT SysTick_Handler
    ADD r4, #1
    BL sine
    LDR lr, =0xFFFFFFFF
    BX lr
ENDP
    
```

addr = 0x08000044

addr = 0x0800001C

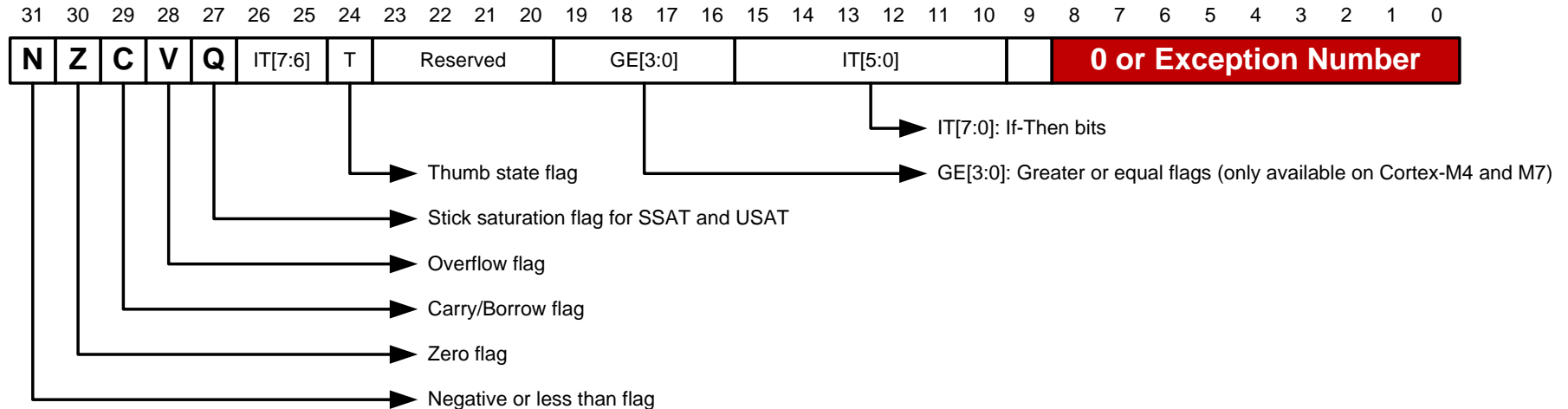
R0	0		xxxxxxx	0x20000200
R1	1	xPSR	0x21000000	0x200001FC
R2	2	PC	0x00000002	0x200001F8
R3	3	SP	0x20000200	0x200001F4
R4	4	LR	0x08001000	0x200001F0
R12	12	R3	3	0x200001EC
R13(SP)	MSP	R2	2	0x200001E8
R14(LR)	0x08000024	R1	1	0x200001E4
R15(PC)	0x080000F0	R0	0	0x200001E0
				0x200001DC
				0x200001D8
				0x200001D4
				0x200001D0
				0x200001CF

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

**Fix the bug! Method 3 (not recommended)**

# Interrupt Number in PSR



# Enable an Interrupt/Exception

- Enable a system exception
  - Some are always enabled (cannot be disabled)
  - No centralized registers for enabling/disabling
  - Each are control by its corresponding components, such as SysTick module
- Enable a peripheral interrupt
  - Centralized register arrays for enabling/disabling
  - NVIC's **ISER** 0~15 registers for enabling
    - Interrupt Set Enable Register
  - NVIC's **ICER** 0~15 registers for disabling
    - Interrupt Clear Enable Register

# Enabling Peripheral Interrupts

## Interrupt Set Enable Register 0 (ISER0)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Enable Bit	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Interrupt Number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	I2C1_EV	TIM4	TIM3	TIM2	TIM11	TIM10	TIM9	LCD	EXTI9_5	COMP	DAC	USB_LP	USB_HP	ADC1	DMA1_CH7	DMA1_CH6	DMA1_CH5	DMA1_CH4	DMA1_CH3	DMA1_CH2	DMA1_CH1	EXTI4	EXTI3	EXTI2	EXTI1	EXTI0	RCC	FLASH	RTC_WKUP	TAMPER_STAMP	PVD	WWDG

## Interrupt Set Enable Register 1 (ISER1)

*Address of ISER1 = Address of ISER0 + 4*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0													
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0													
																				44	43	42	41	40	39	38	37	36	35	34	33	32												
																				TIM7	TIM6	USB_FS_WKUP	RTC_Alarm	EXTI5_10	USART3	USART2	USART1	SPI2	SPI1	I2C2_ER	I2C2_EV	I2C1_ER												

TIM7\_IRQn = 44

**TIM7\_IRQn = 44**

**NVIC->ISER[1] = 1 << 12; // Enable Timer 7 interrupt**

# Disabling Peripheral Interrupts

## Interrupt Clear Enable Register 0 (ICER0)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Clear Enable Bit	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Interrupt Number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	I2C1_EV	TIM4	TIM3	TIM2	TIM11	TIM10	TIM9	LCD	EXTI9_5	COMP	DAC	USB_LP	USB_HP	ADC1	DMA1_CH7	DMA1_CH6	DMA1_CH5	DMA1_CH4	DMA1_CH3	DMA1_CH2	DMA1_CH1	EXTI4	EXTI3	EXTI2	EXTI1	EXTI0	RCC	FLASH	RTC_WKUP	TAMPER_STAMP	PVD	WWDG

## Interrupt Clear Enable Register 1 (ICER1) *Address of ICER1 = Address of ICER0 + 4*

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Clear Enable Bit	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Interrupt Number																				44	43	42	41	40	39	38	37	36	35	34	33	32
																				TIM7	TIM6	USB_FS_WKUP	RTC_Alarm	EXTI15_10	USART3	USART2	USART1	SPI2	SPI1	I2C2_ER	I2C2_EV	I2C1_ER

**TIM7\_IRQn = 44**

**NVIC->ICER[1] = 1 << 12;      // Disable Timer 7 interrupt**

# Priority Management



# Interrupt Priority

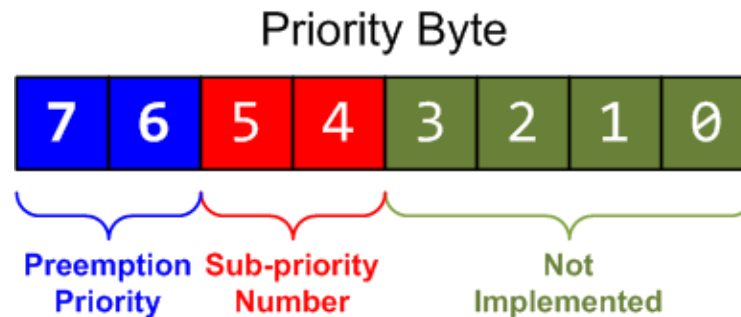
- Inverse Relationship:
  - Lower priority value means higher urgency.
    - Priority of Interrupt A = 5,
    - Priority of Interrupt B = 2,
    - B has a higher priority/urgency than A.
- Fixed priority for Reset, HardFault, and NMI.

Exception	IRQn	Priority
Reset	N/A	-3 (the highest)
Non-maskable Interrupt (NMI)	14	-2 (2 <sup>nd</sup> highest)
Hard Fault	13	-1

- Adjustable for all the other interrupts

# Interrupt Priority

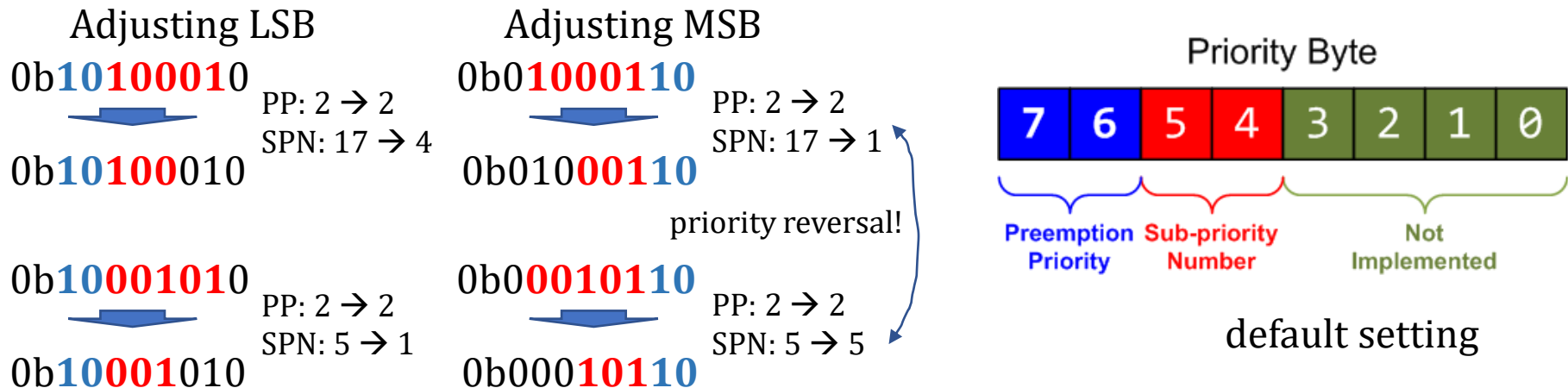
- Interrupt priority is configured by **Interrupt Priority Register (IPR)** 0~123
  - 124 IPRs \* 4 priority configuration per IPR = 496  
(= Total # of interrupts supported on ARMv7-M)
- Each priority consists of two fields, including **preempt priority number** and **sub-priority number**.
  - The preempt priority number defines the priority for preemption.
  - The sub-priority number determines the order when multiple interrupts are pending with the same preempt priority number.



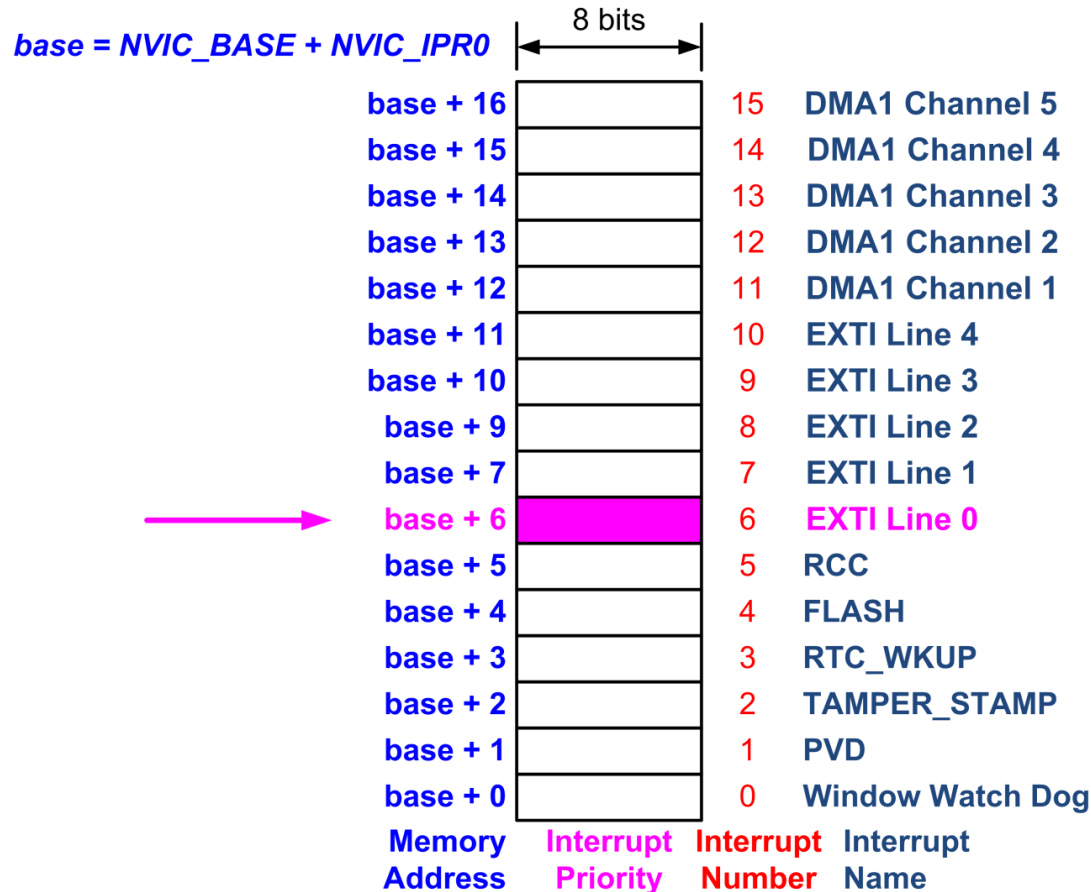
default setting

# Interrupt Priority

- The lengths of the Preemption Priority-field and the Sub-priority Number-field are configurable.
  - The position of LSB is adjusted.
- Why adjusting LSB?
  - Easier porting!
  - Programs implemented on many-bit priority-level device can run on small-bit priority level device without inversion of priority.
  - example: 2/5 bits priority fields → 2/3 bits priority fields

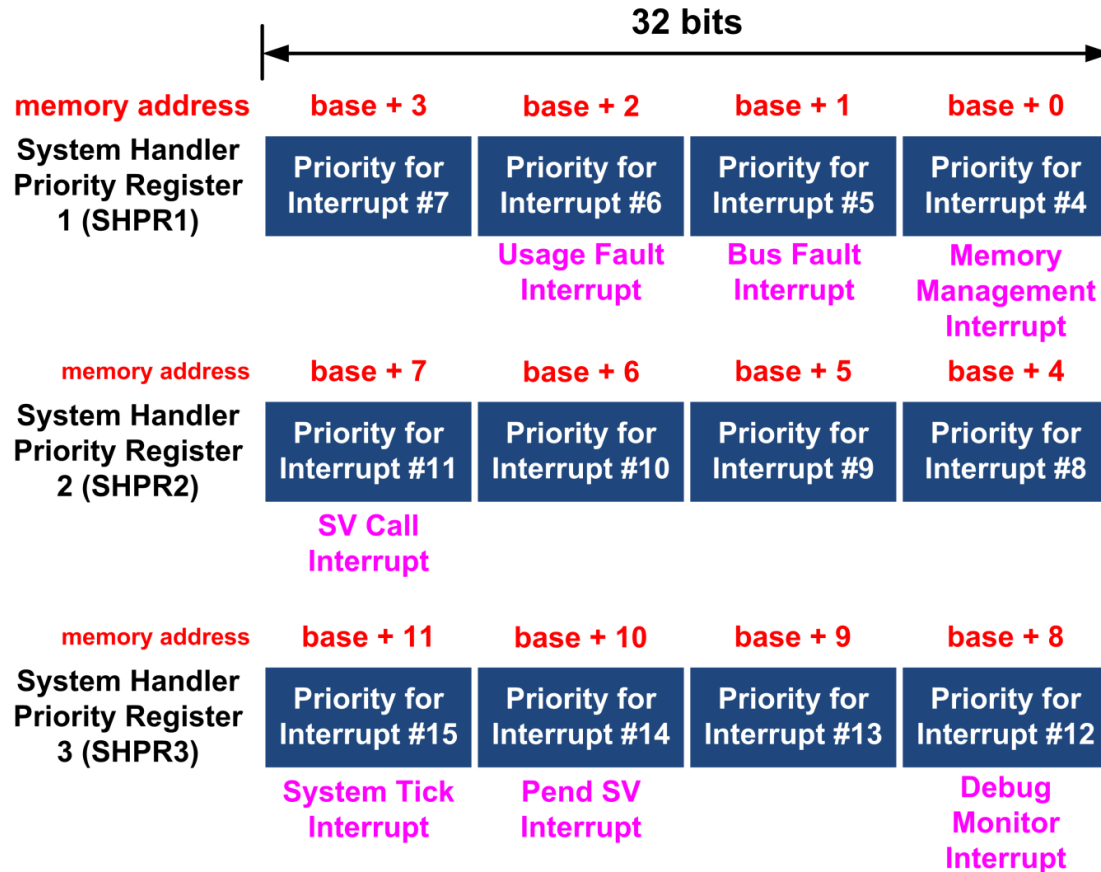


# Priority of Peripheral Interrupts



```
// Set the priority for EXTI 0 (Interrupt number 6)
NVIC->IP[6] = 0xF0;
```

# Priority of System Interrupts



```
// Set the priority of a system interrupt IRQn
SCB->SHP[(IRQn) & 0xF] - 4] = (priority << 4) & 0xFF;
```

# Exception-masking registers

- **PRIMASK**: Used to disable all exceptions except Non-maskable interrupt (NMI) and hard fault.

- Write 1 to PRIMASK to disable all interrupts except NMI and hard fault exception

```
MOV R0, #1  
MSR PRIMASK, R0
```

- Write 0 to PRIMASK to enable all interrupts

```
MOV R0, #0  
MSR PRIMASK, R0
```

- **FAULTMASK**: Like PRIMASK but change the current priority level to -1, so that even hard fault handler is blocked

- **BASEPRI**: Disable interrupts only with priority lower than a certain level

- Example, disable all exceptions with priority value higher than 0x60

```
MOV R0, #0x60  
MSR BASEPRI, R0
```