Chapter 12
Interrupt

Dr. Yifeng Zhu
Electrical and Computer Engineering
University of Maine

Spring 2015

Interrupts

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

Merriam-Webster:

"to break the uniformity or continuity of"

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.

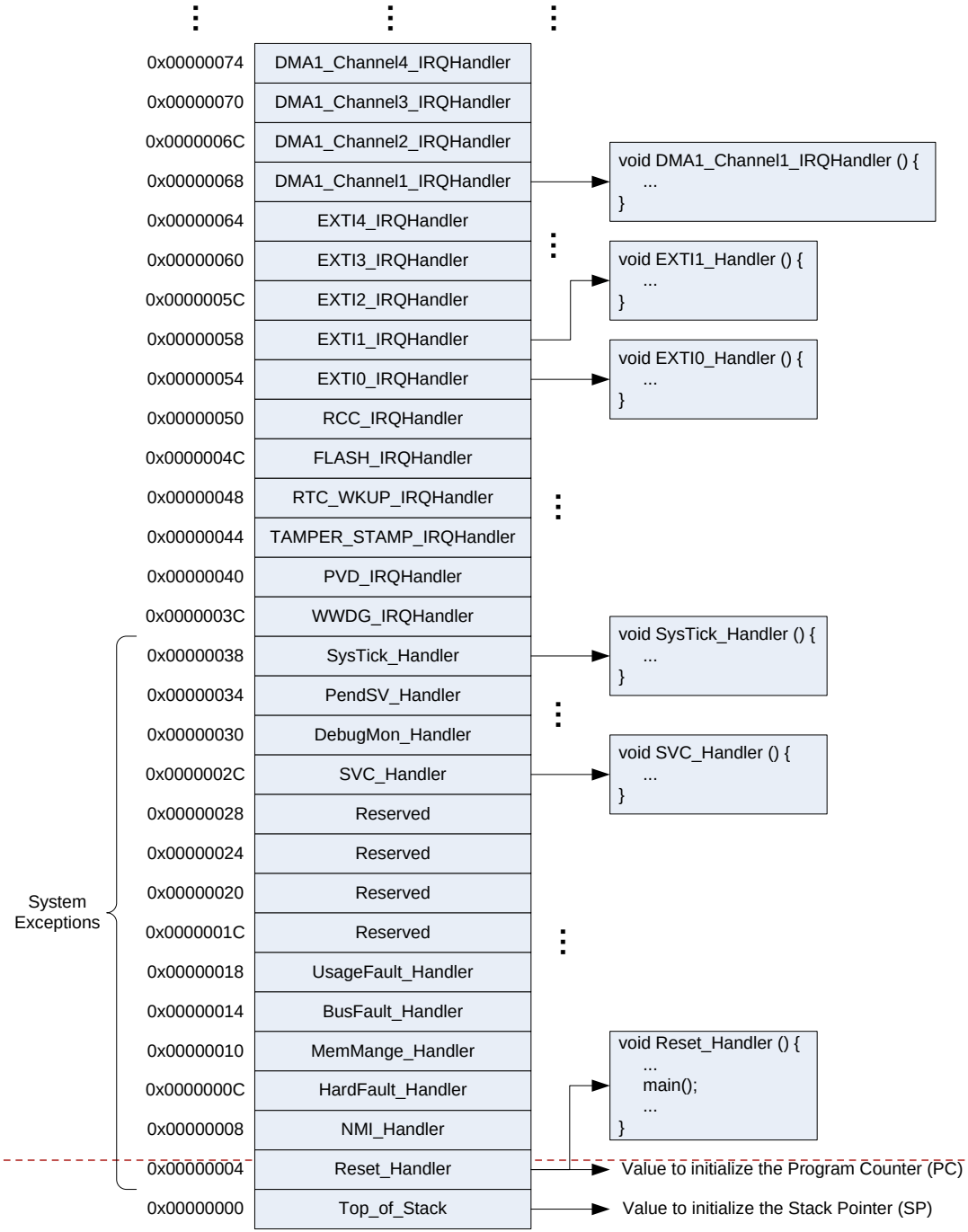
Image from http://vecto.rs

Interrupt Service Routine (ISR) Vector Table

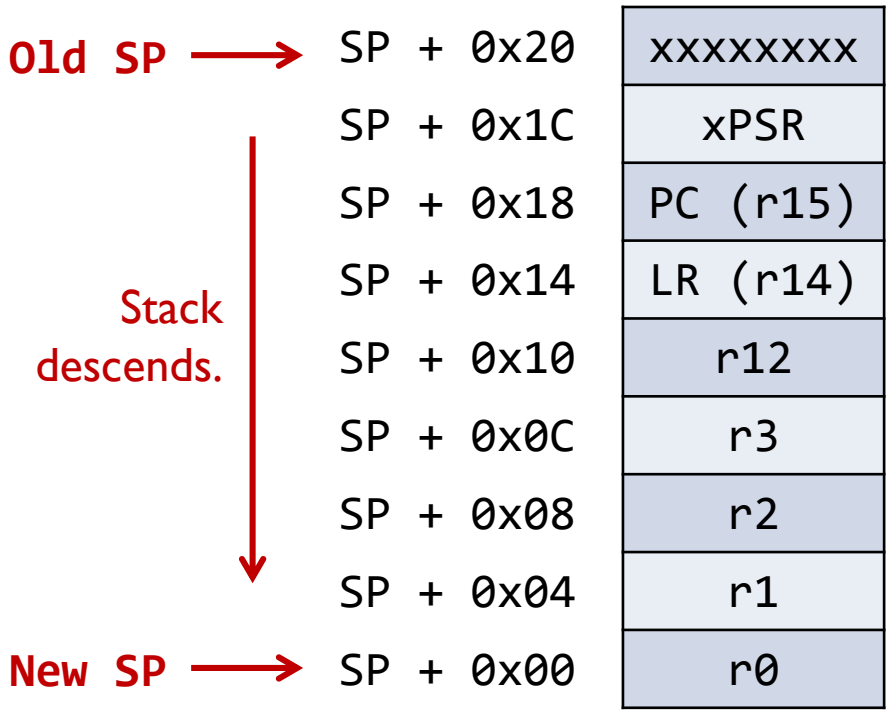
- Start address for the exception handler for each exception type is fixed and pre-defined
- Processor loads PC with this fixed, pre-defined address
- Exception Vector Table starts at memory address 0
- Program Counter pc = 0x00000004 initially

Address	Priority	Type of priority	Acronym	Description
0x0000_0000	-	-	-	Stack Pointer
0x0000_0004	-3	fixed	Reset	Reset Vector
0x0000_0008	-2	fixed	NMI_Handler	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.
0x0000_000C	-1	fixed	HardFault_Handler	All class of fault
0x0000_0010	0	settable	MemManage_Handler	Memory management
0x0000_0014	1	settable	BusFault_Handler	Pre-fetch fault, memory access fault
0x0000_0018	2	settable	UsageFault_Handler	Undefined instruction or illegal state
0x0000_001C-0x0000_002B	-	-	-	Reserved
0x0000_002C	3	settable	SVC_Handler	System service call via SWI instruction
0x0000_0030	4	settable	DebugMon_Handler	Debug Monitor
0x0000_0034	-	-	-	Reserved
0x0000_0038	5	settable	PendSV_Handler	Pendable request for system service
0x0000_003C	6	settable	SysTick_Handler	System tick timer
...				

ISR Vector Table



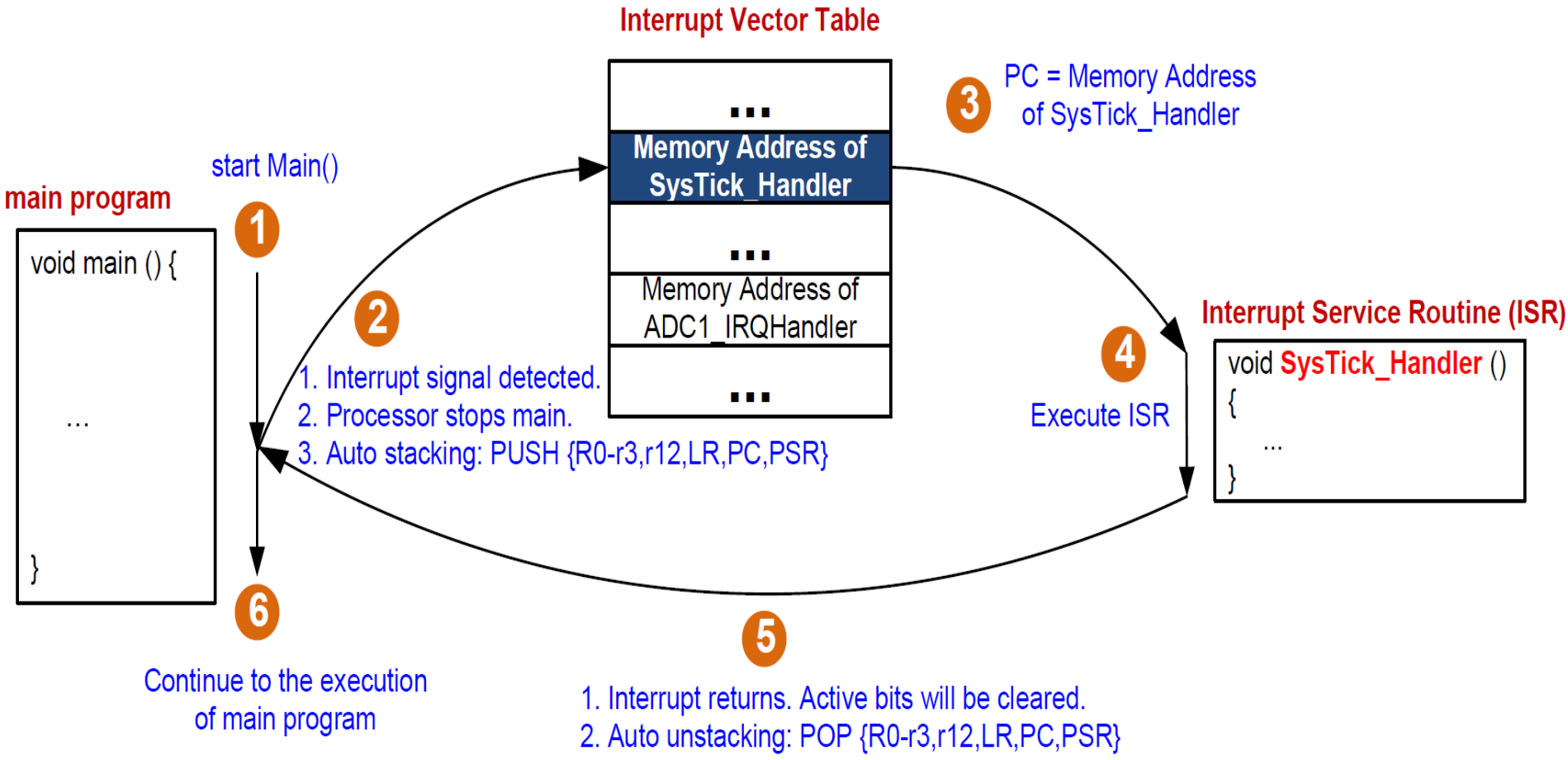
Stacking & Unstacking



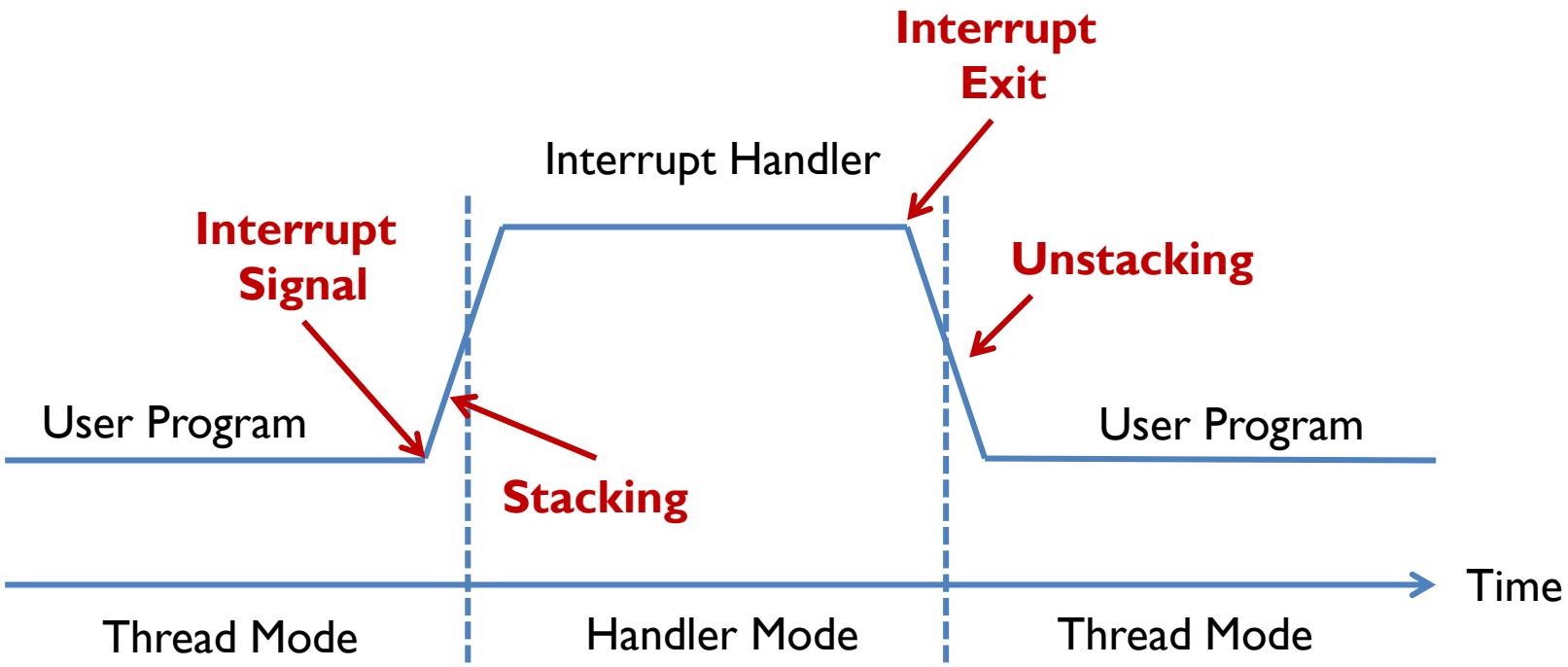
- The processor automatically pushes these eight registers into the main stack before an interrupt handler starts
- The processor automatically pops these eight register out of the main stack when an interrupt hander exits.

- Two SPs: Main SP (MSP) and Process SP (PSP)
- Determined by operating mode, and CONTROL[0]
  - Thread mode ⟶ SP = PSP
  - Handler mode ⟶ SP = MSP if CONTROL[0] = 0; Otherwise SP = PSP

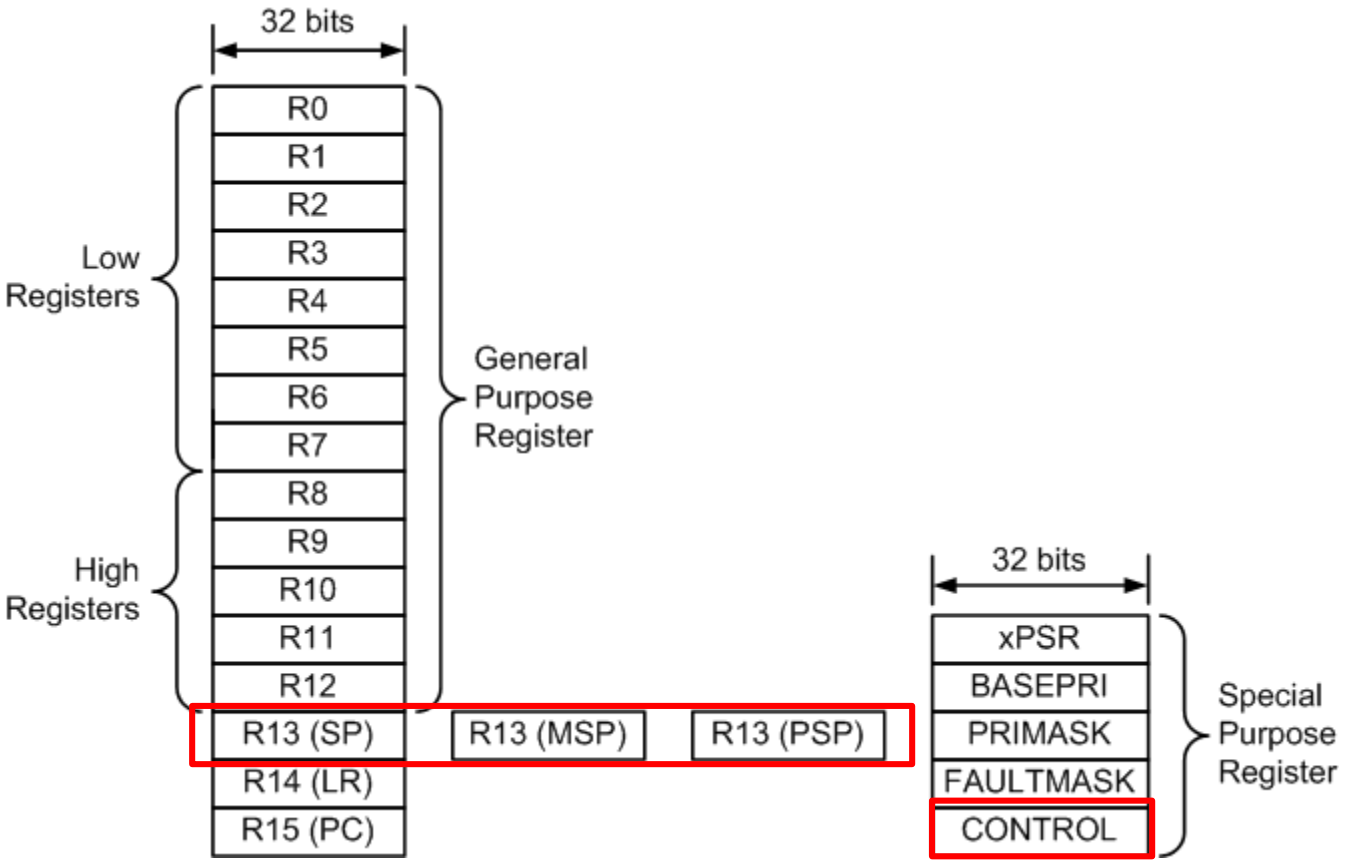
Interrupt


Interrupt Vector Table
...
Memory Address of SysTick_Handler
...
Memory Address of ADC1_IRQHandler
...
3
PC = Memory Address of SysTick_Handler
start Main()
main program
1
void main () {
...
}
2
1. Interrupt signal detected.
2. Processor stops main.
3. Auto stacking: PUSH {R0-r3,r12,LR,PC,PSR}
4
Execute ISR
Interrupt Service Routine (ISR)
void SysTick_Handler ()
{
...
}
5
1. Interrupt returns. Active bits will be cleared.
2. Auto unstacking: POP {R0-r3,r12,LR,PC,PSR}
6
Continue to the execution of main program

Stacking & Unstacking



Registers



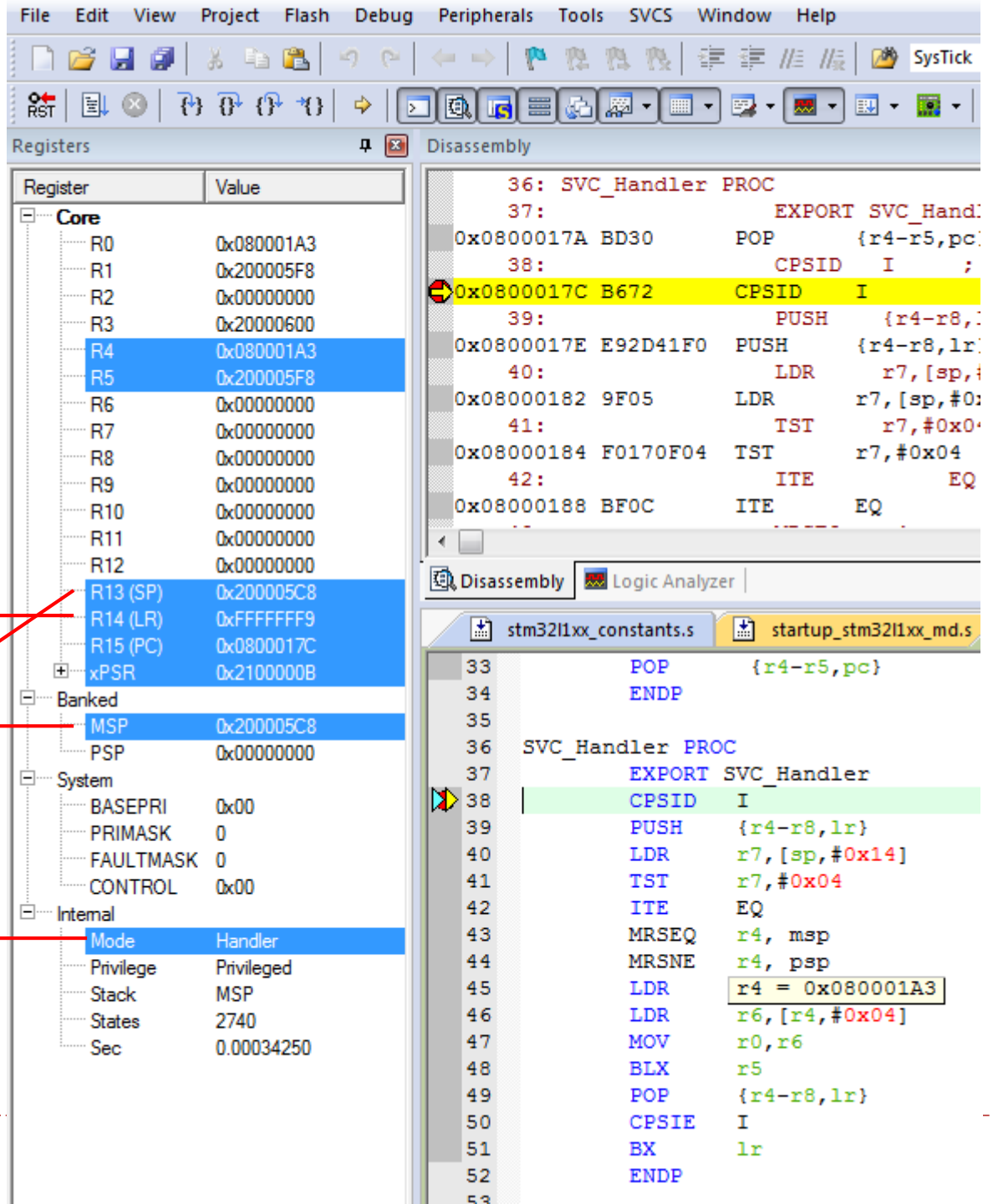
MSP: Main Stack Pointer

PSP: Process Stack Pointer

Processor Mode: Handler Mode *vs* Thread Mode

- Handler mode and Thread mode
  - Handler mode always use MSP (Main Stack Pointer)
  - Thread Mode uses either PSP (Process Stack Pointer) or MSP
    - Control[1] = 0 → SP = MSP (default)
    - Control[1] = 1 → SP = PSP
- When the processor is reset, the default is the thread mode.
- The processor enters the handler mode when an exception occurs.

Register values in a interrupt service routine



LR = 0xFFFFFFF9

SP = MSP

ISR always in hander mode.

Which stack to use when exiting an interrupt?

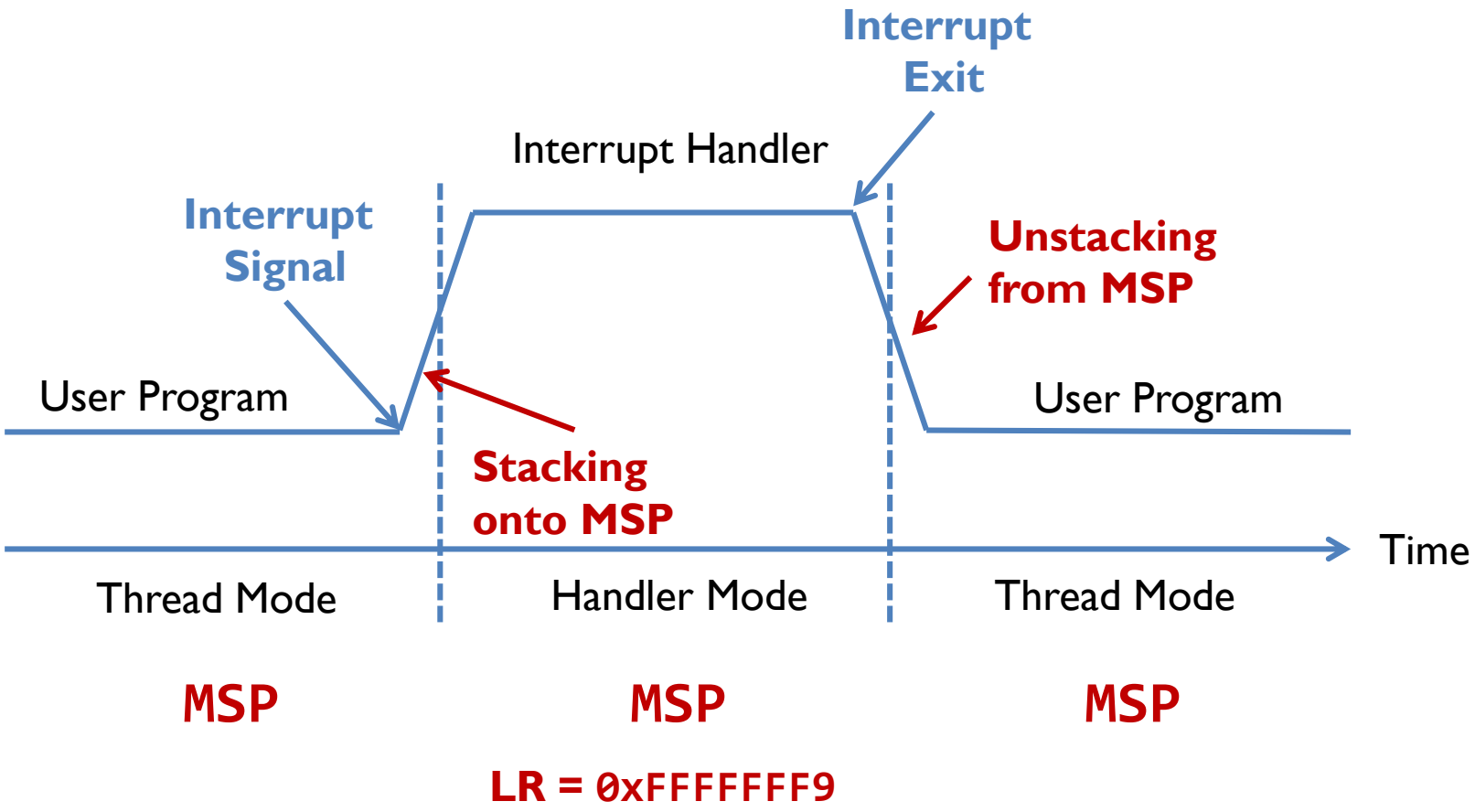
Link Register (LR) now has two usages:

- LR = address of the instruction immediately after BL
- LR indicates whether MSP or PSP is used to restore register when exiting an interrupt
  - LR value generated by processor
  - The processor set LR to **`0xFFFFFFFF`** on reset.
  - If return to handler mode, the MSP stack is always used

Link Register (LR)	Return Mode	Return Stack
`0xFFFFFFE1`	Handler	SP = MSP
`0xFFFFFFF9`	Thread	SP = MSP
`0xFFFFFFFD`	Thread	SP = PSP

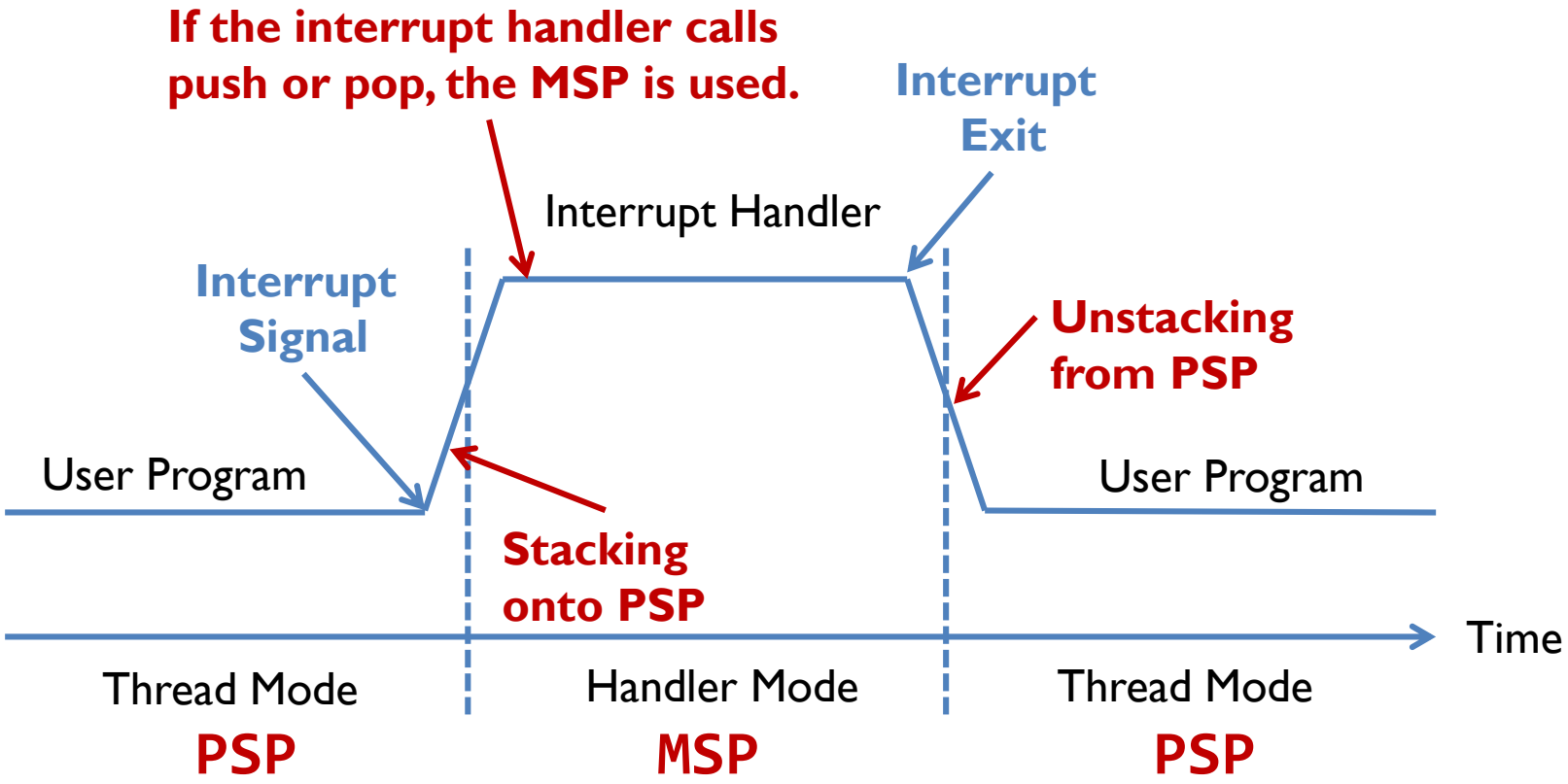
Stacking & Unstacking

Control[1] = 0 ⟹ User program uses MSP.



Stacking & Unstacking

Control[1] = 1 ⟹ User program uses PSP.



LR = 0xFFFFFFFD

Which is being used?

When exiting an interrupt:

- If LR = 0xFFFFFFF9, then SP = MSP
- If LR = 0xFFFFFFFD, then SP = PSP

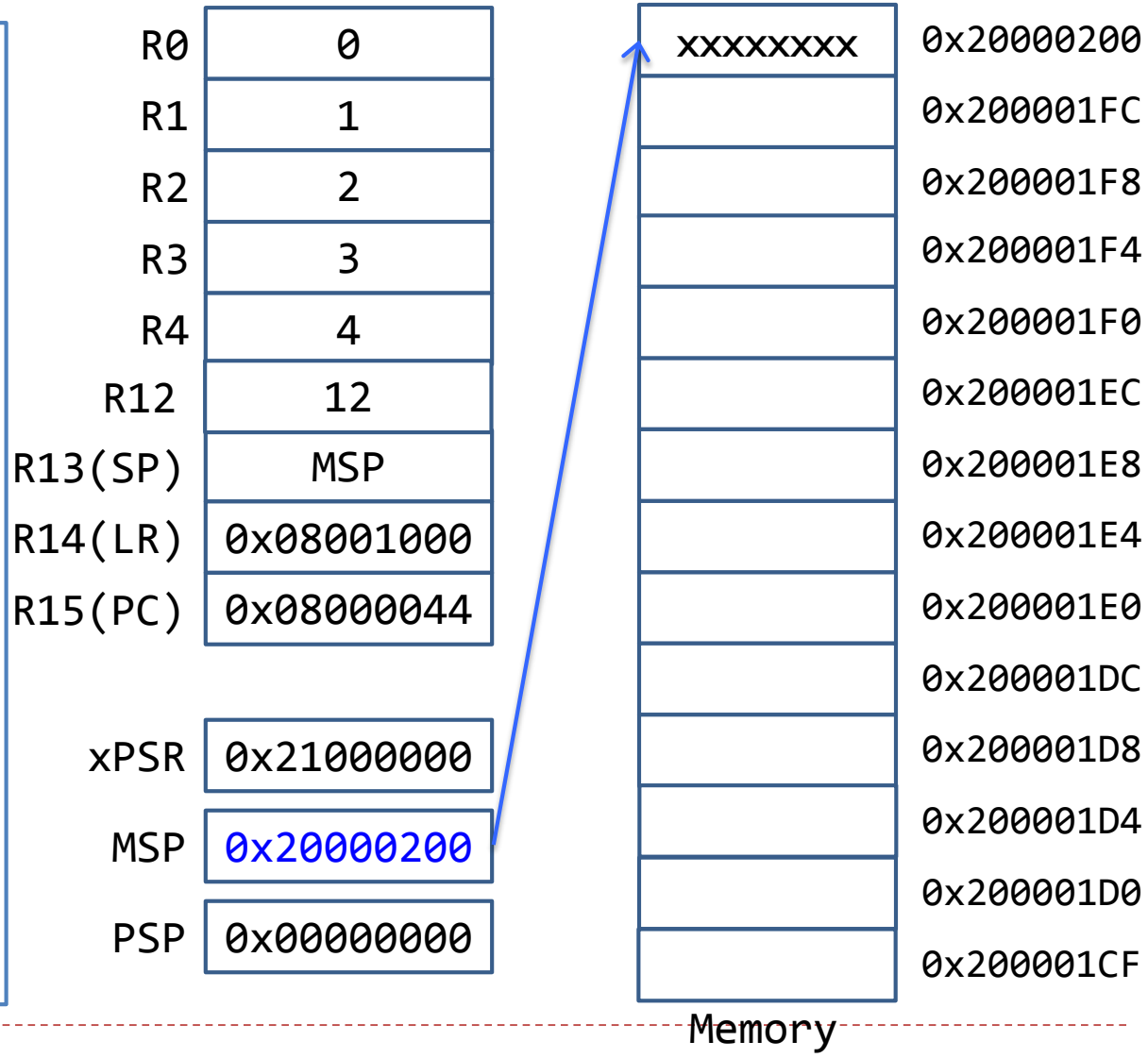
9 = 1001

D = 1101

```
TST    r7, #0x04
MRSEQ  r4, msp
MRSNE  r4, psp
```

Interrupt: Stacking & Unstacking

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```



Interrupt:

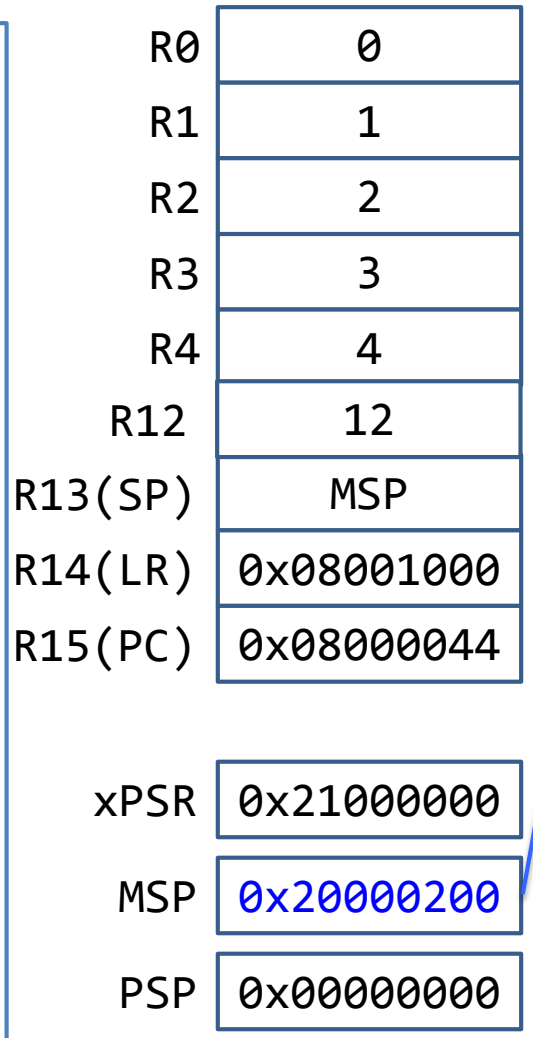
Suppose SysTick interrupt occurs when PC = 0x08000044

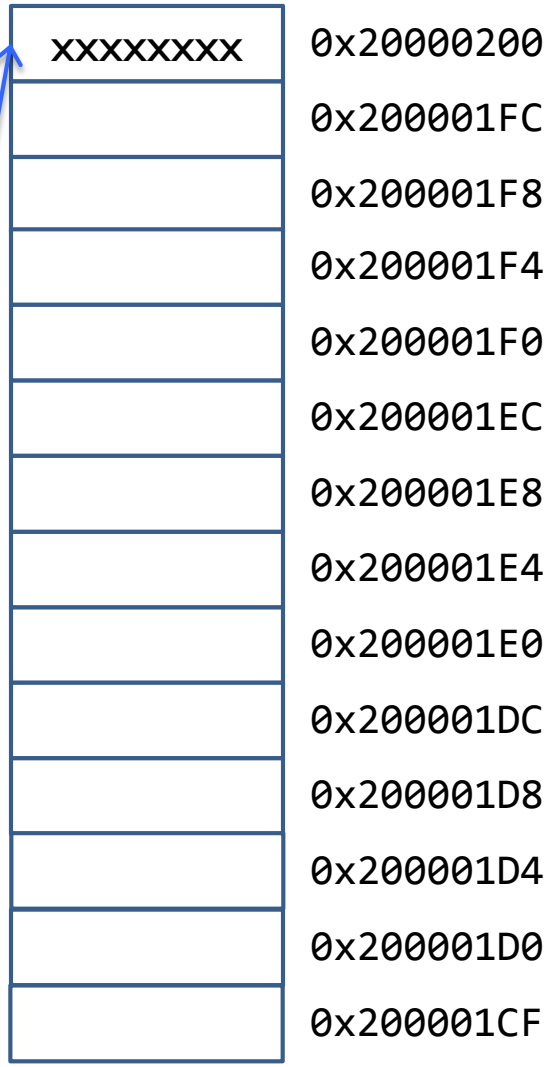
```
__main PROC
   …
   MOV r3,#0
   …
   ENDP

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

addr = 0x08000044

addr = 0x0800001C





Interrupt: Stacking & Uns

STACKING

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	MSP
R14(LR)	0xFFFFFFF9
R15(PC)	0x0800001C

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

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

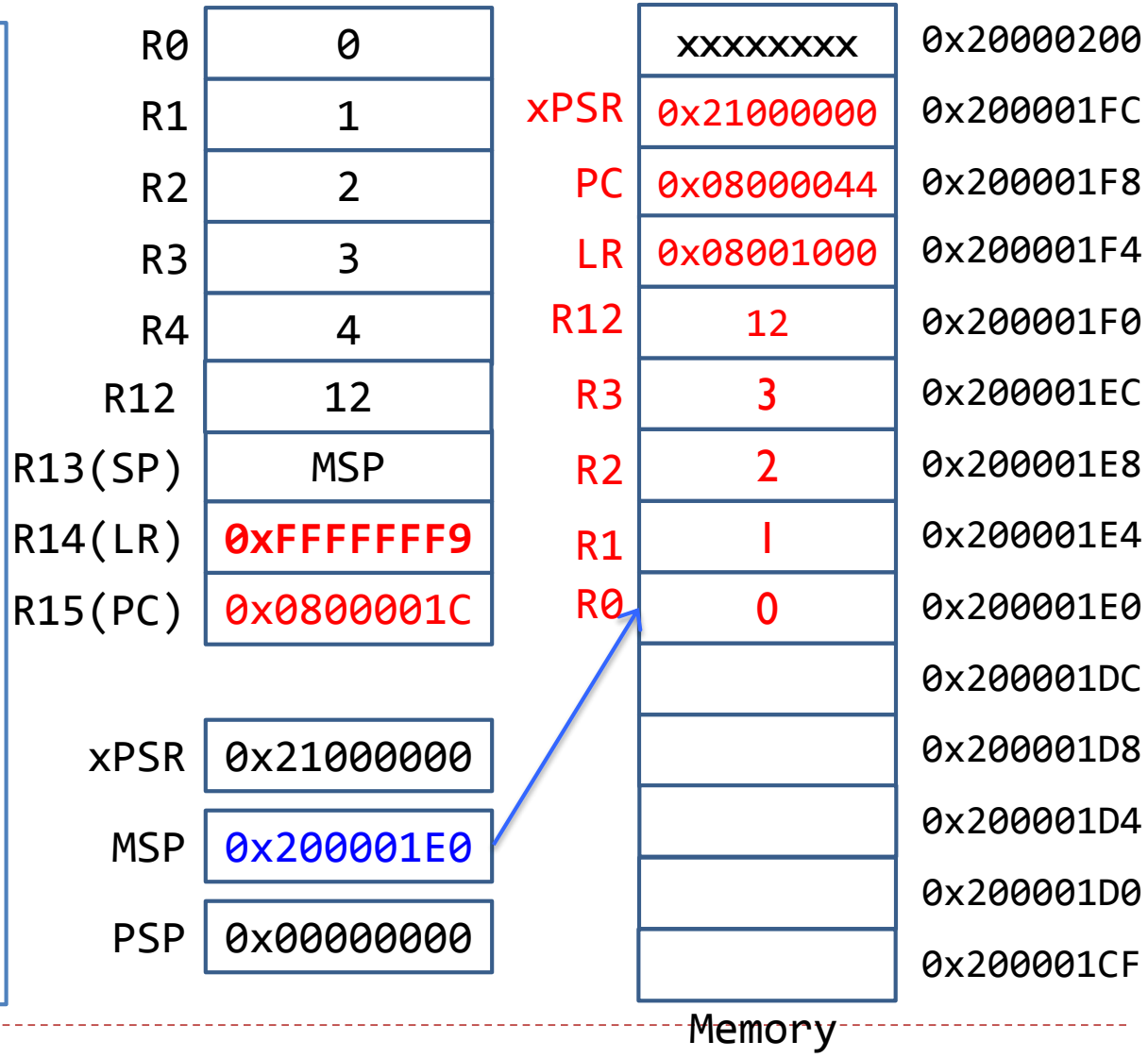
Memory

Interrupt: Stacking & Uns

STACKING

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```

LR = 0xFFFFFFF9 to indicate MSP is used.



Interrupt: Stacking & Unstacking

```
__main PROC
                addr = 0x08000044
  …
  MOV r3,#0
  …
  ENDP
                addr = 0x0800001C
SysTick_Handler PROC
  EXPORT SysTick_Handler
  ADD r3, #1
  ADD r4, #1
  BX  lr
  ENDP
```

R0	0
R1	1
R2	2
R3	**4**
R4	4
R12	12
R13(SP)	MSP
R14(LR)	0xFFFFFFF9
R15(PC)	0x0800001C

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

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

Memory

Interrupt: Stacking & Unstacking

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```

R0	0
R1	1
R2	2
R3	4
R4	5
R12	12
R13(SP)	MSP
R14(LR)	0xFFFFFFF9
R15(PC)	0x08000020

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

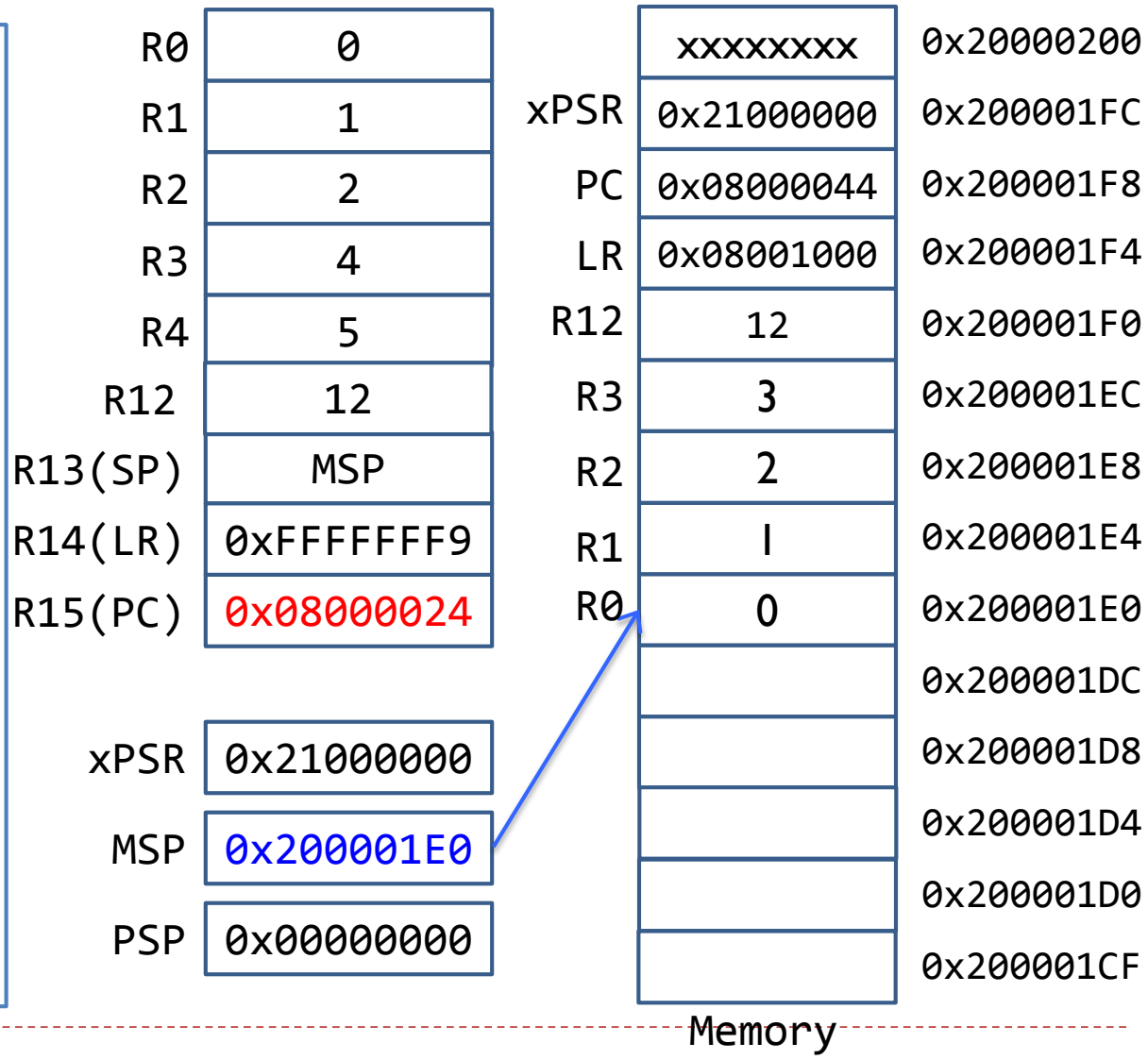
	xxxxxxxx	0x20000200
xPSR	0x21000000	0x200001FC
PC	0x08000044	0x200001F8
LR	0x08001000	0x200001F4
R12	12	0x200001F0
R3	3	0x200001EC
R2	2	0x200001E8
R1	1	0x200001E4
R0	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

Memory

Interrupt: Stacking & Unstacking

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```

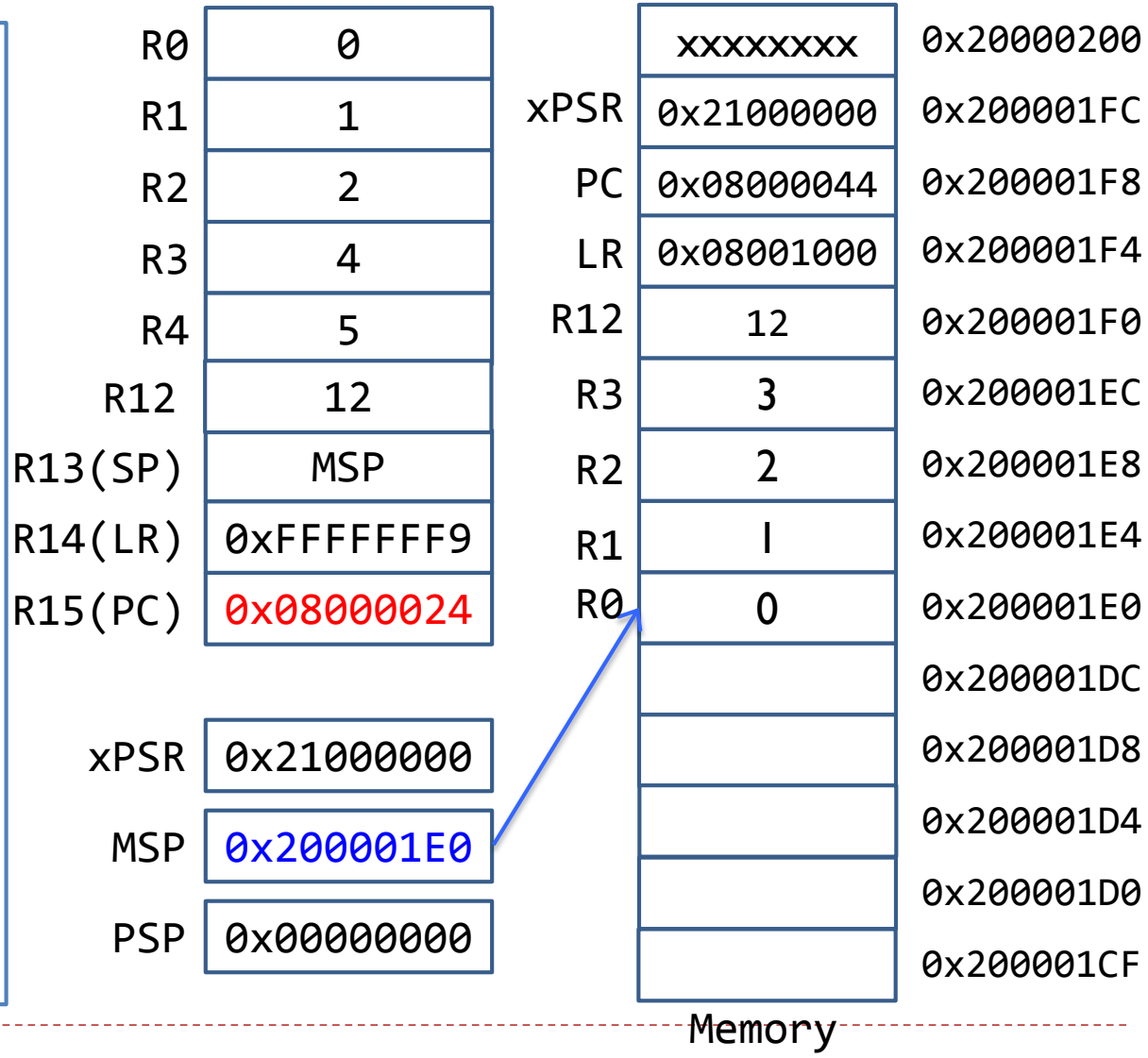
LR = 0xFFFFFFF9 to indicate MSP is used.



Interrupt: Stacking & Uns UNSTACKING

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```

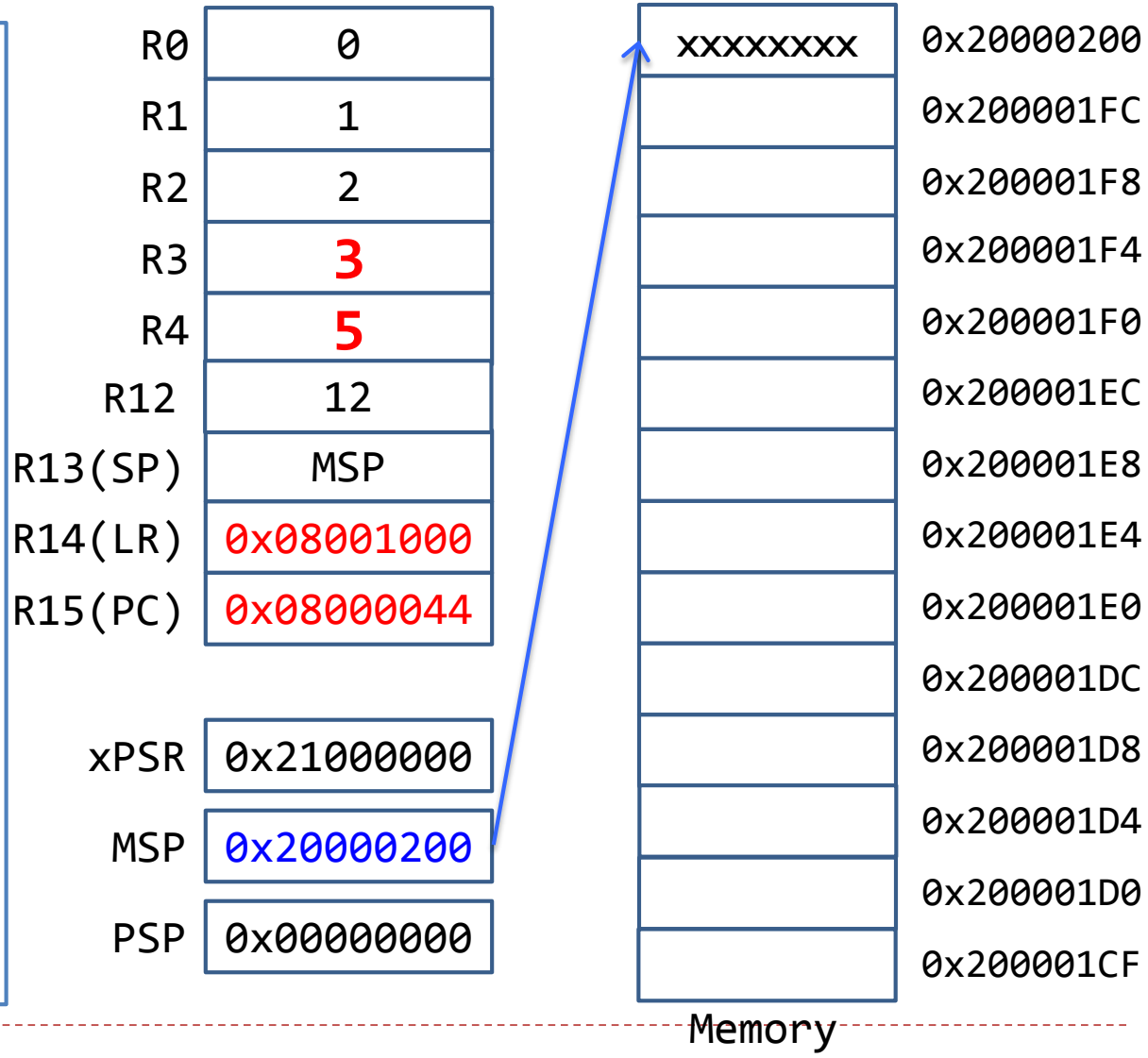
LR = 0xFFFFFFF9 to indicate MSP is used.



Interrupt: Stacking & Uns UNSTACKING

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```

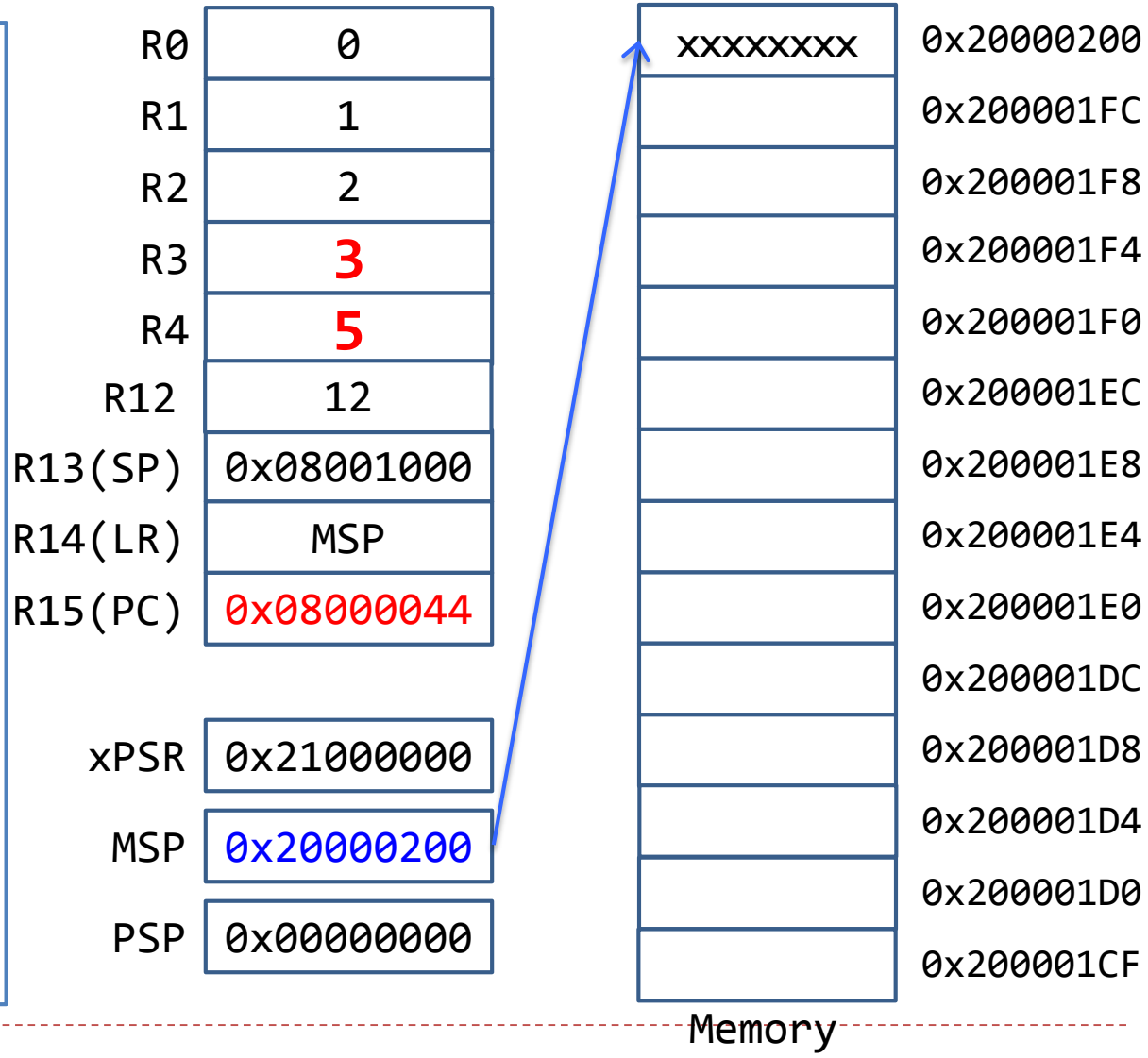
Note the new value of R3 is lost!!!



Interrupt: Stacking & Unstacking

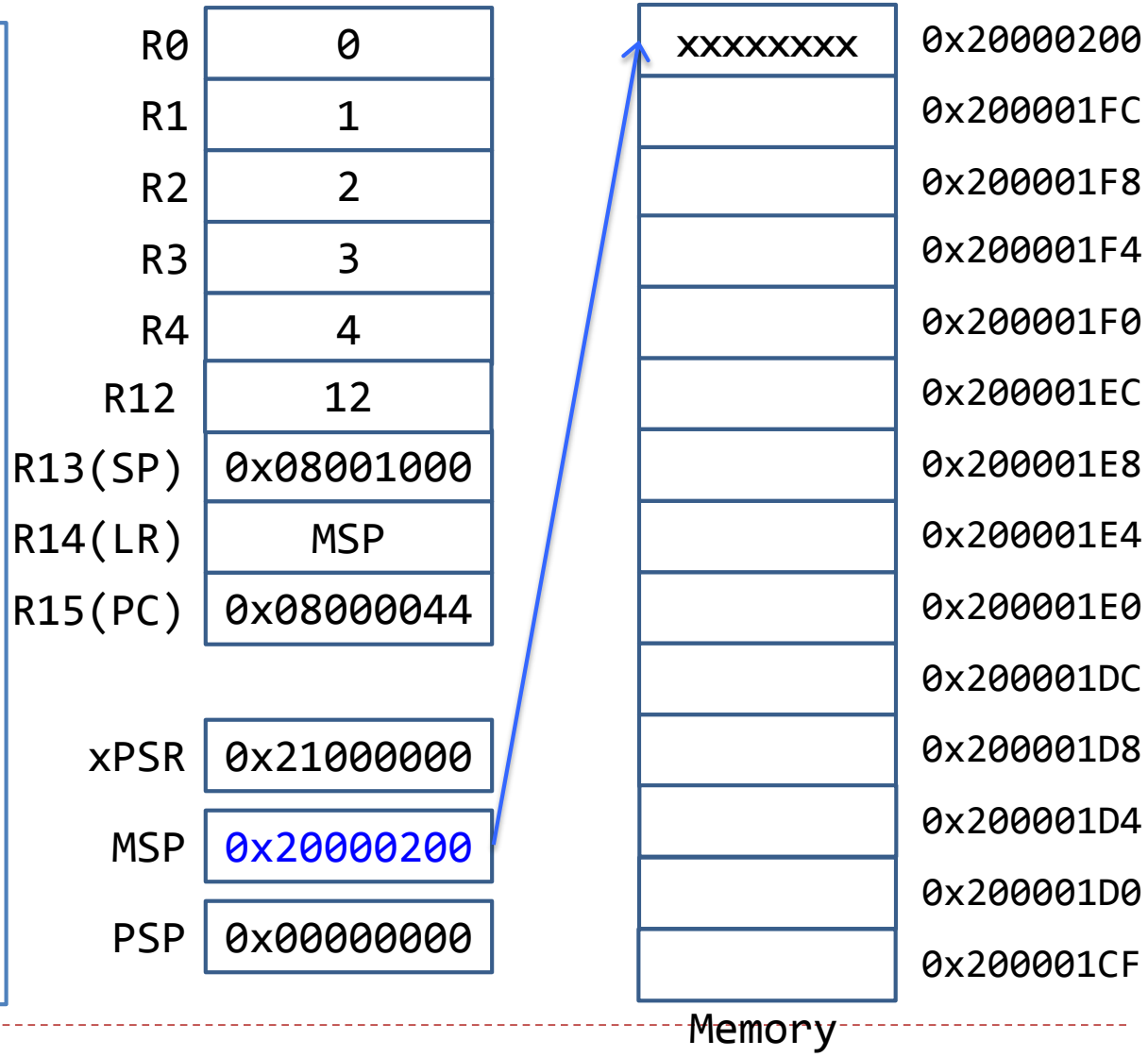
```
__main PROC
                addr = 0x08000044
  …
  MOV r3,#0
  …
  ENDP
                addr = 0x0800001C
SysTick_Handler PROC
  EXPORT SysTick_Handler
  ADD r3, #1
  ADD r4, #1
  BX  lr
  ENDP
```

The Main program resumes!!!



Interrupt: Stacking & Unstacking

```
__main PROC
                addr = 0x08000044
  …
  MOV r3,#0
  …
  ENDP
                addr = 0x0800001C
SysTick_Handler PROC
  EXPORT SysTick_Handler
  ADD r4, #1
  BL  sine
  BX  lr
  ENDP
```



Interrupt: Stacking & Uns

STACKING

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r4, #1
   BL  sine
   BX  lr
   ENDP
```

LR = 0xFFFFFFF9 to indicate MSP is used.

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	0xFFFFFFF9
R14(LR)	MSP
R15(PC)	0x0800001C

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

	xxxxxxxx	0x20000200
xPSR	0x21000000	0x200001FC
PC	0x00000002	0x200001F8
SP	0x20000200	0x200001F4
LR	0x08001000	0x200001F0
R3	3	0x200001EC
R2	2	0x200001E8
R1	1	0x200001E4
R0	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

Memory

Interrupt: Stacking & Uns

STACKING

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r4, #1
   BL  sine
   BX  lr
   ENDP
```

BL sine
Updates LR register

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	0x080000F4
R14(LR)	MSP
R15(PC)	0x080000F0

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

	xxxxxxxx	0x20000200
xPSR	0x21000000	0x200001FC
PC	0x00000002	0x200001F8
SP	0x20000200	0x200001F4
LR	0x08001000	0x200001F0
R3	3	0x200001EC
R2	2	0x200001E8
R1	1	0x200001E4
R0	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

Memory

Interrupt: Stacking & Uns

STACKING

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r4, #1
   BL  sine
   BX  lr
   ENDP
```

BL sine
Updates LR register

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	0x080000F4
R14(LR)	MSP
R15(PC)	0x080000F0

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

	xxxxxxxx	0x20000200
xPSR	0x21000000	0x200001FC
PC	0x00000002	0x200001F8
SP	0x20000200	0x200001F4
LR	0x08001000	0x200001F0
R3	3	0x200001EC
R2	2	0x200001E8
R1	1	0x200001E4
R0	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

Memory

Interrupt: Stacking & Uns

UNSTACKING won't occurs!

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r4, #1
   BL  sine
   BX  lr
   ENDP
```

BL sine
Updates LR register

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	0x080000F4
R14(LR)	MSP
R15(PC)	0x080000F0

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

	xxxxxxxx	0x20000200
xPSR	0x21000000	0x200001FC
PC	0x00000002	0x200001F8
SP	0x20000200	0x200001F4
LR	0x08001000	0x200001F0
R3	3	0x200001EC
R2	2	0x200001E8
R1	1	0x200001E4
R0	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

Memory

Interrupt: Stacking & Unstacking

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

SysTick_Handler PROC
  EXPORT SysTick_Handler
  PUSH {lr}
  ADD r4, #1
  BL  sine
  POP {lr}
  BX  lr
  ENDP
```

addr = 0x08000044

addr = 0x0800001C

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	0x080000F4
R14(LR)	MSP
R15(PC)	0x080000F0

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

	xxxxxxxx	0x20000200
xPSR	0x21000000	0x200001FC
PC	0x00000002	0x200001F8
SP	0x20000200	0x200001F4
LR	0x08001000	0x200001F0
R3	3	0x200001EC
R2	2	0x200001E8
R1	1	0x200001E4
R0	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

Memory

Fix the bug! Method 1

Interrupt: Stacking & Unstacking

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
  PUSH {lr}
   ADD r4, #1
   BL  sine
  POP {PC}
   ENDP
```

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	0x080000F4
R14(LR)	MSP
R15(PC)	0x080000F0

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

	xxxxxxxx	0x20000200
xPSR	0x21000000	0x200001FC
PC	0x00000002	0x200001F8
SP	0x20000200	0x200001F4
LR	0x08001000	0x200001F0
R3	3	0x200001EC
R2	2	0x200001E8
R1	1	0x200001E4
R0	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

Memory

Fix the bug! Method 2

Interrupt: Stacking & Unstacking

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r4, #1
   BL  sine
  LDR lr,=0xFFFFFFF9
   BX  lr
   ENDP
```

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	0x080000F4
R14(LR)	MSP
R15(PC)	0x080000F0

xPSR	0x21000000
MSP	0x200001E0
PSP	0x00000000

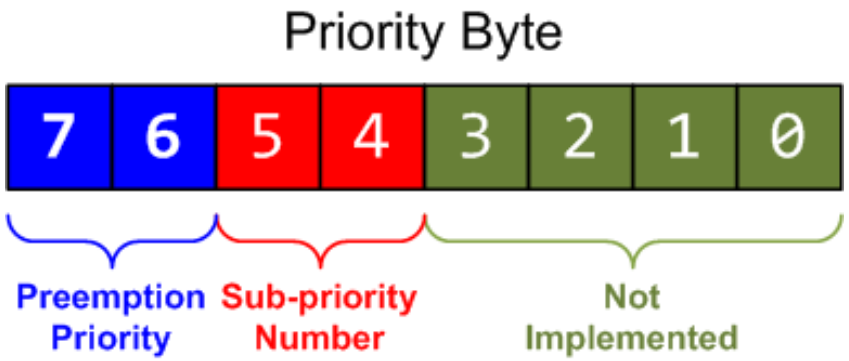
	xxxxxxxx	0x20000200
xPSR	0x21000000	0x200001FC
PC	0x00000002	0x200001F8
SP	0x20000200	0x200001F4
LR	0x08001000	0x200001F0
R3	3	0x200001EC
R2	2	0x200001E8
R1	1	0x200001E4
R0	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

Memory

Fix the bug! Method 3

Interrupt Priority Levels

- Priority determines the order to be serviced
- A lower value ⟶ a higher priority or a higher urgency.
- Each interrupt has an Interrupt Priority Register (IP)
- Each IP 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.

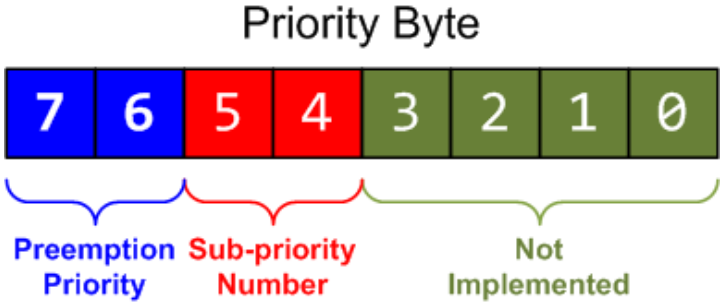


Interrupt Priority Levels

- STM32L microcontroller only supports 16 interrupt levels, ranging from 0 to 15. While Cortex-M3 uses eight bits to store the priority number, STM32L only uses four bits.

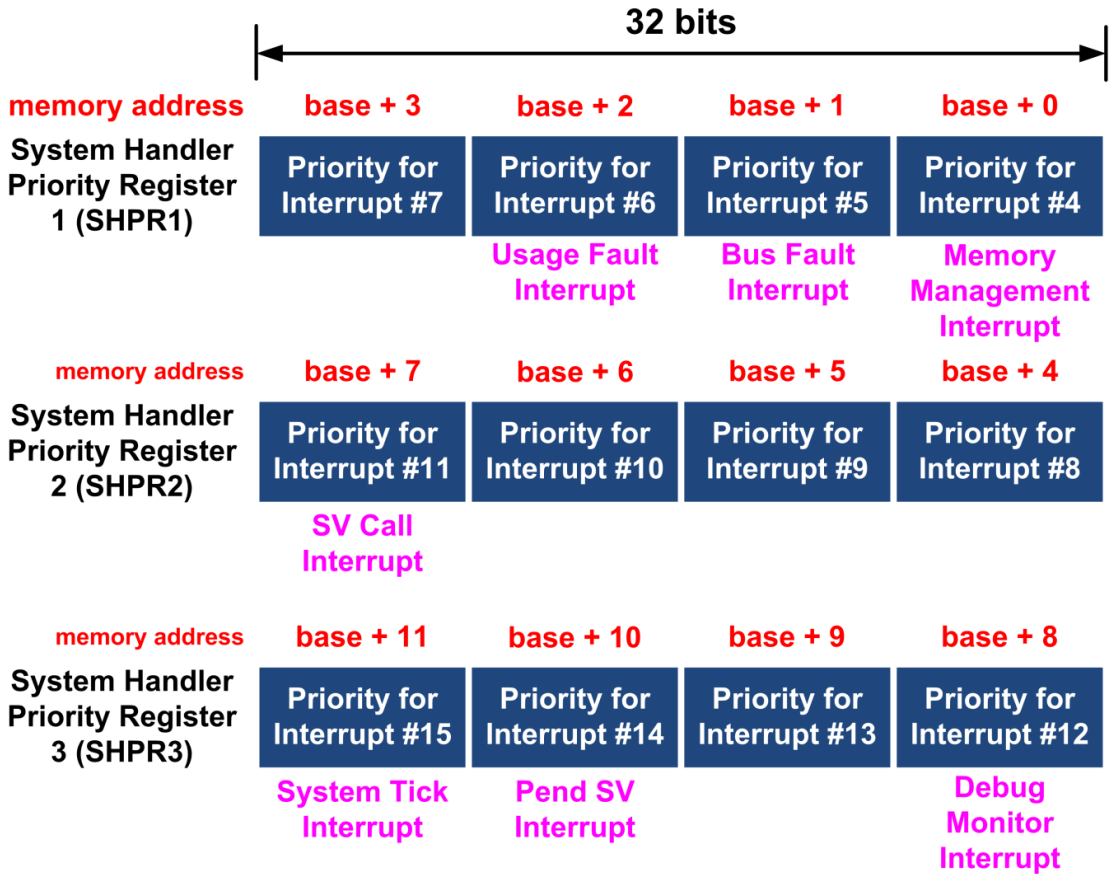
```
NVIC->IP[ IRQn ] = (priority << 4) & 0xff;
```

- For STM32L processors, by default, two bits are used for the preempt priority number, and two bits are used for the sub-priority number.



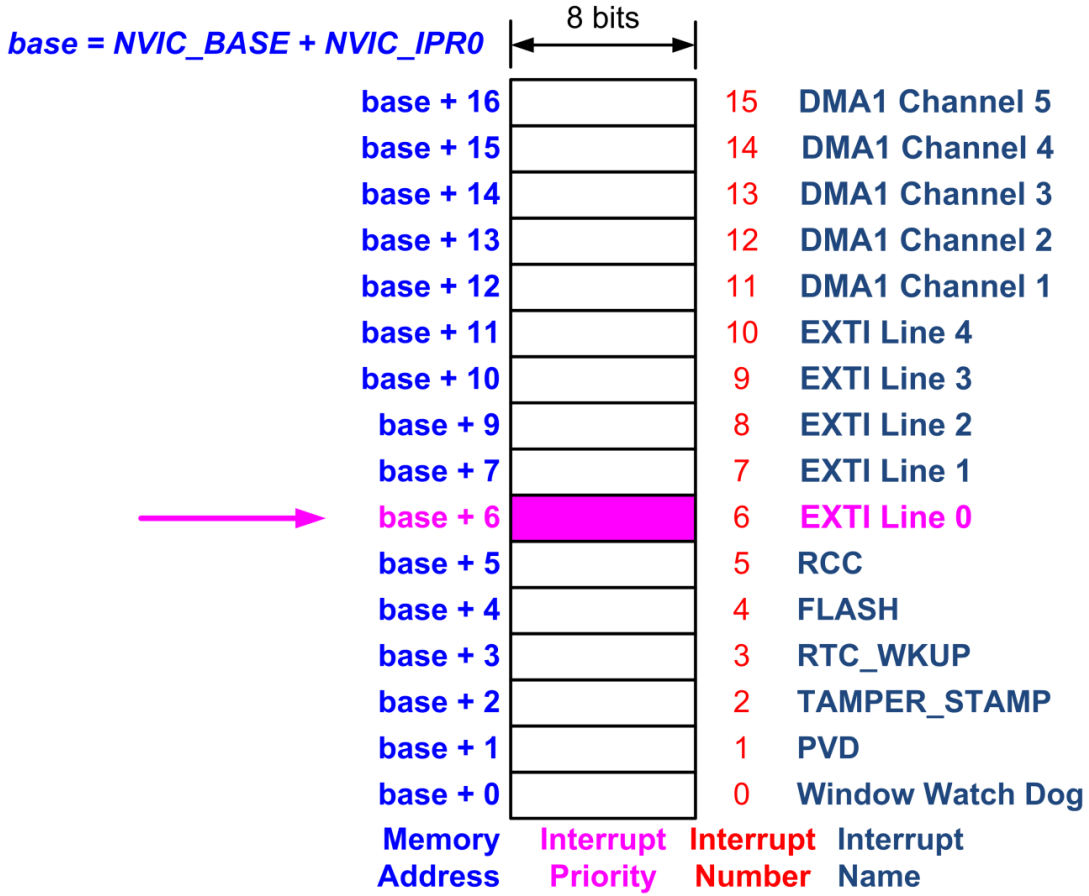
Priority level 0 is the highest priority level, and priority level 15 is the lowest.

Priority of System Interrupts




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

Priority of Peripheral Interrupts




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

Exception-masking registers (PRIMASK, FAULTMASK and BASEPRI)

- PRIMASK: Used to disable all exceptions except Non-maskable interrupt (NMI) and hard fault.
  - Write 1 to PRIMASK to disable all interrupts except NMI

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

  - Write 0 to PRIMASK to enable all interrupts

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

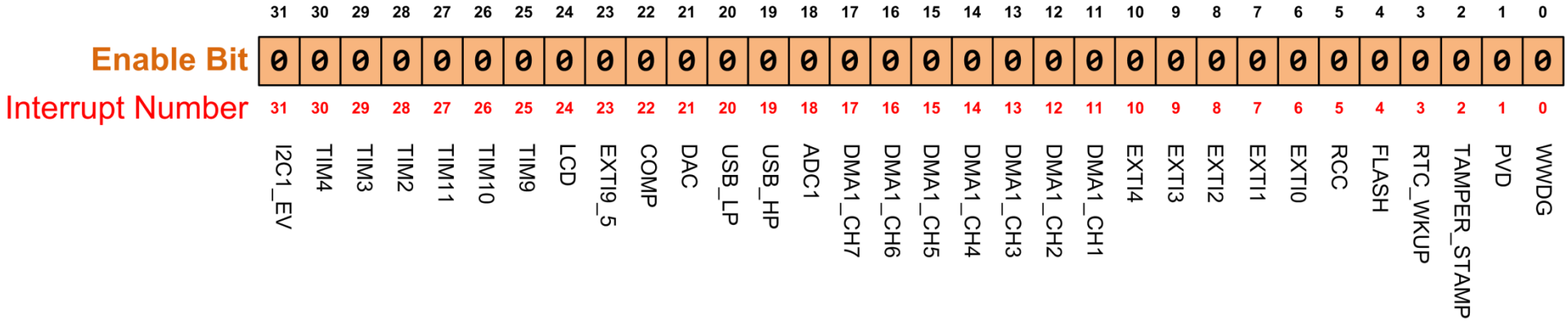
- FAULTMASK: Like PRIMASK but change the current priority level to -1, so that even hard fault handler is blocked

- BASEPRI: Disable interrupts only with priority lower than a certain level
  - Example, disable all exceptions with priority level higher than 0x60

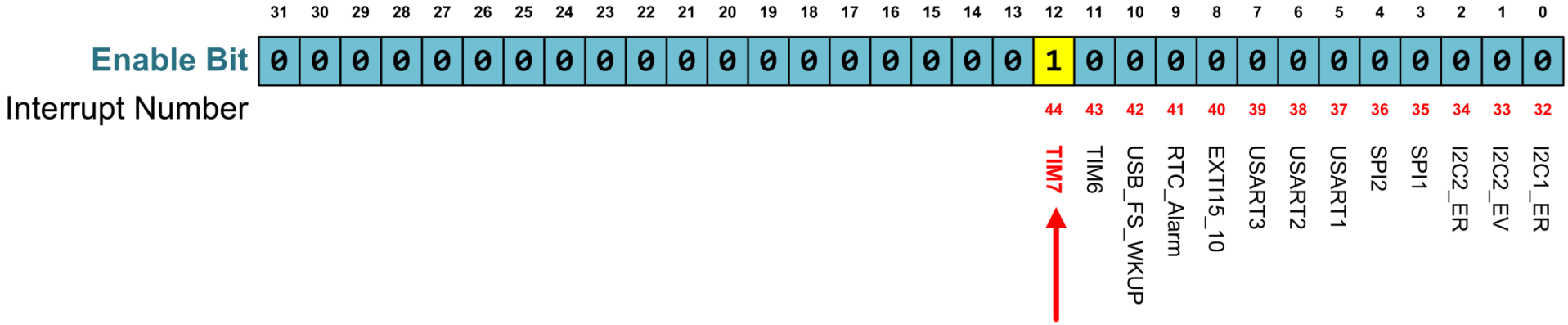
```
MOV R0, #0x60
MSR BASEPRI, R0
```

Interrupt Enable

Interrupt Set Enable Register 0 (ISER0)



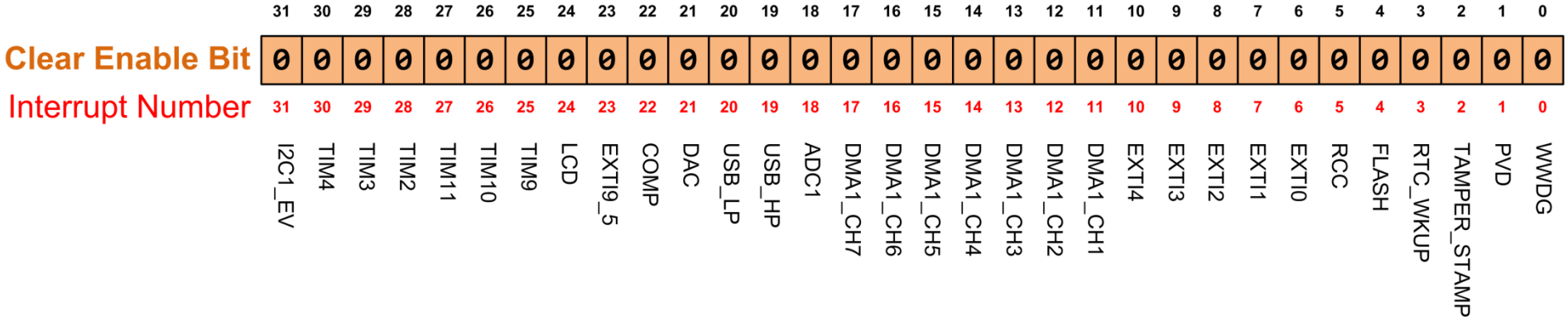
Interrupt Set Enable Register 1 (ISER1) *Address of ISER1 = Address of ISER0 + 4*



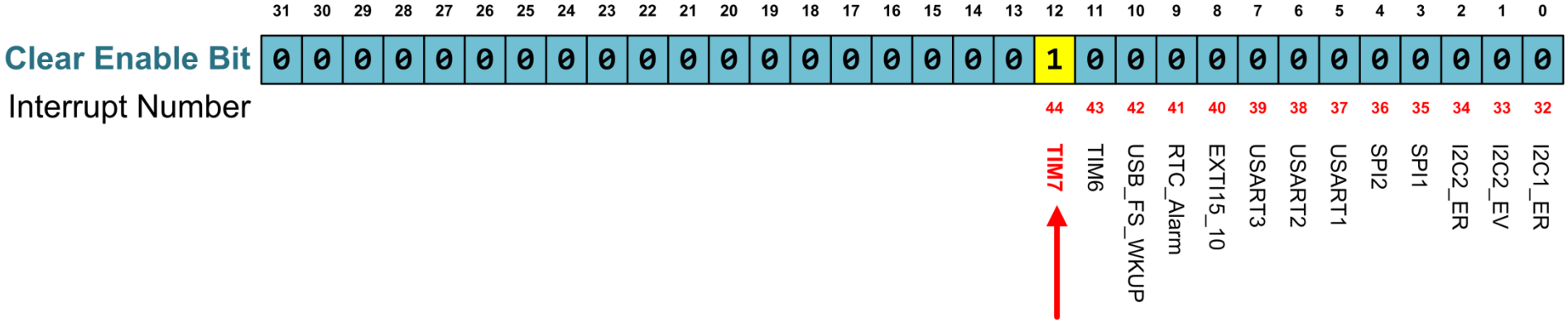

```
NVIC->ISER[1] = 1 << 12;      // Enable Timer 7 interrupt
```

Interrupt Disable

Interrupt Clear Enable Register 0 (ICER0)



Interrupt Clear Enable Register 1 (ICER1) *Address of ICER1 = Address of ISER0 + 4*




```
NVIC->ICER[1] = 1 << 12;      // Diable Timer 7 interrupt
```