Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C

Chapter 11
Interrupt

Z. Gu

Fall 2025

Acknowledgement: Lecture slides based on Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, University of Maine https://web.eece.maine.edu/~zhu/book/

Polling *vs* Interrupt

- Interrupt-driven operations
  - Allows CPU to perform other tasks until external/internal devices require service
  - CPU stops the current code and starts to execute an ISR

Polling	Interrupt
Software periodically checks	CPU takes action only if an event occurs
Waste lot of CPU cycles	Does not waste CPU cycles
Triggered by software	Triggered by hardware or software
Occurs periodically	Can occur any time

Interrupts

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

Suppose you are waiting for an important phone call.

Polling:
You pick up the phone every three seconds to check whether you are getting a call.

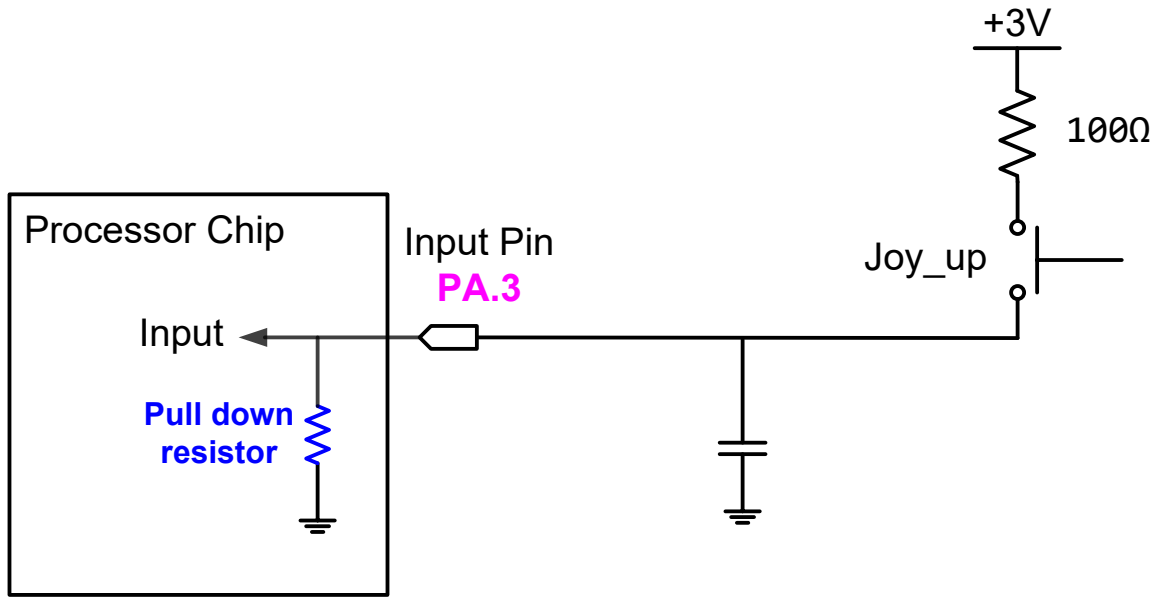
Interrupt:
Do whatever you should do and pick up the phone when it rings.

```
// Polling
while (1) {
    read_button_input;
    if (pushed)
        exit;
}
turn_on_LED;
```

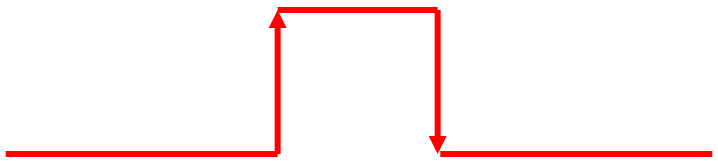
```
// Interrupt
interrupt_handler(){
 turn_on_LED;
 exit;
}
```

Example: Push a button to turn on a LED

- Check whether a button has been pressed?
- **Polling**
  - Repeatedly read IDR and check whether bit 3 is set (i.e., busy wait)
  - OK if CPU has nothing else to do
- **Interrupt**
  - When hardware detects a rising or fall edge, hardware generates a service request
  - CPU responses to the service request and starts to execute the corresponding service subroutine

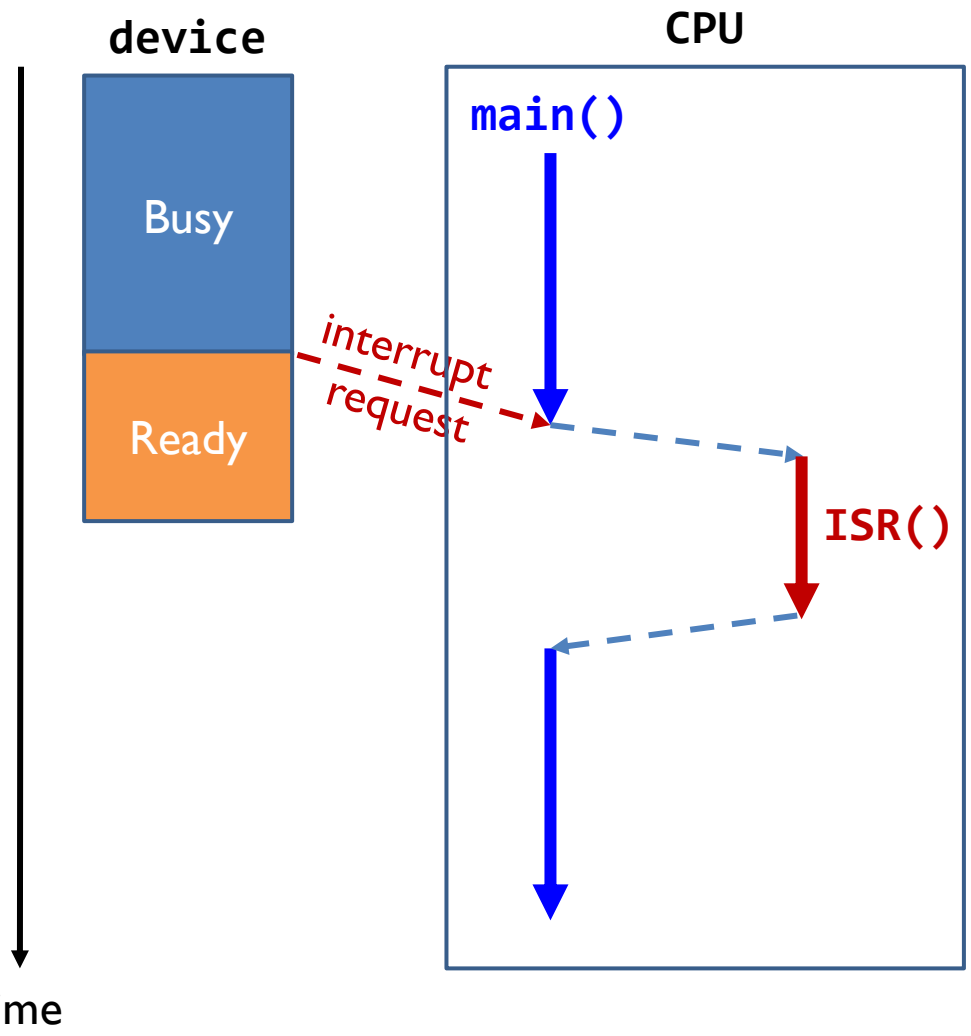


Voltage on PA.3

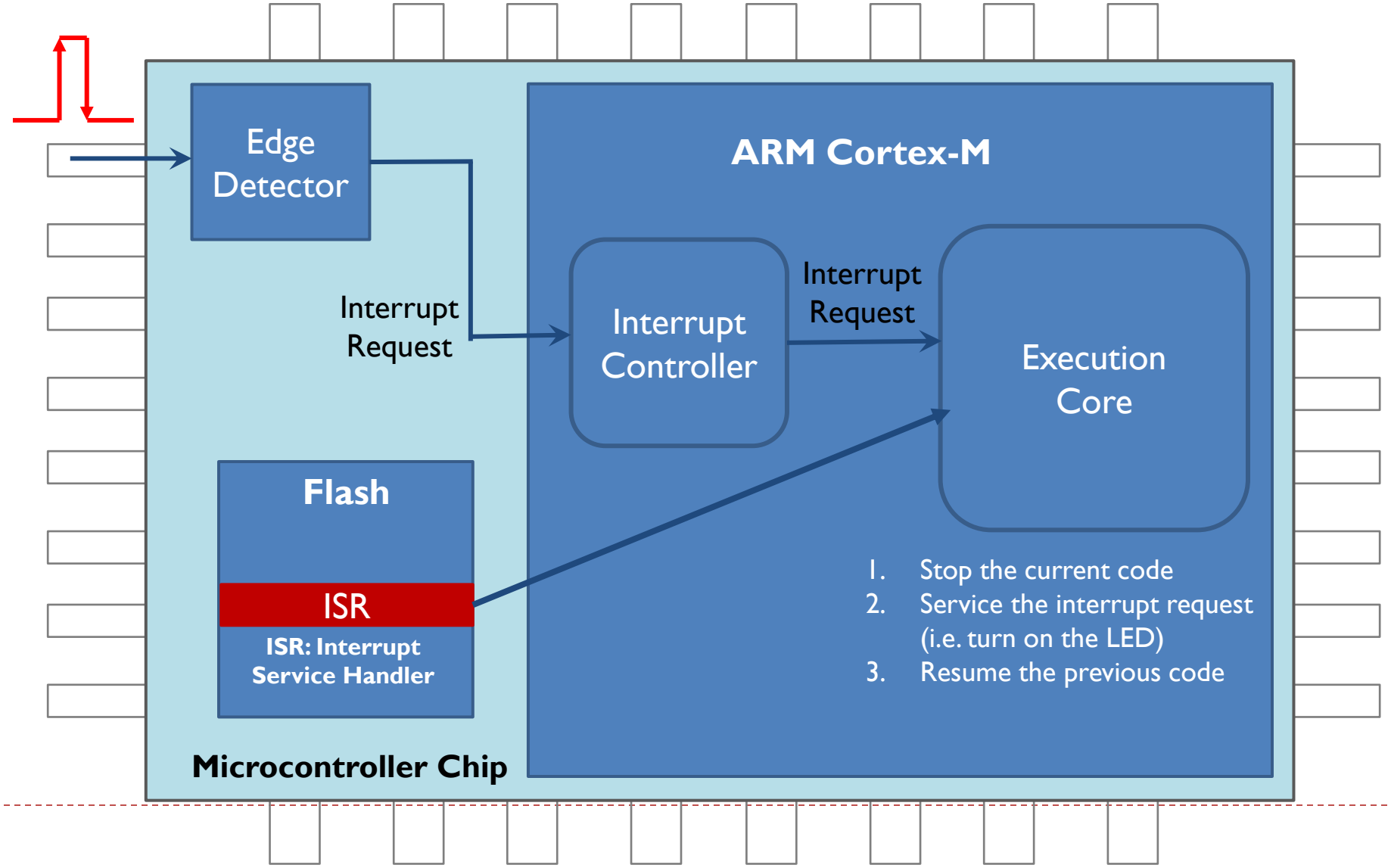


Polling *vs* Interrupt

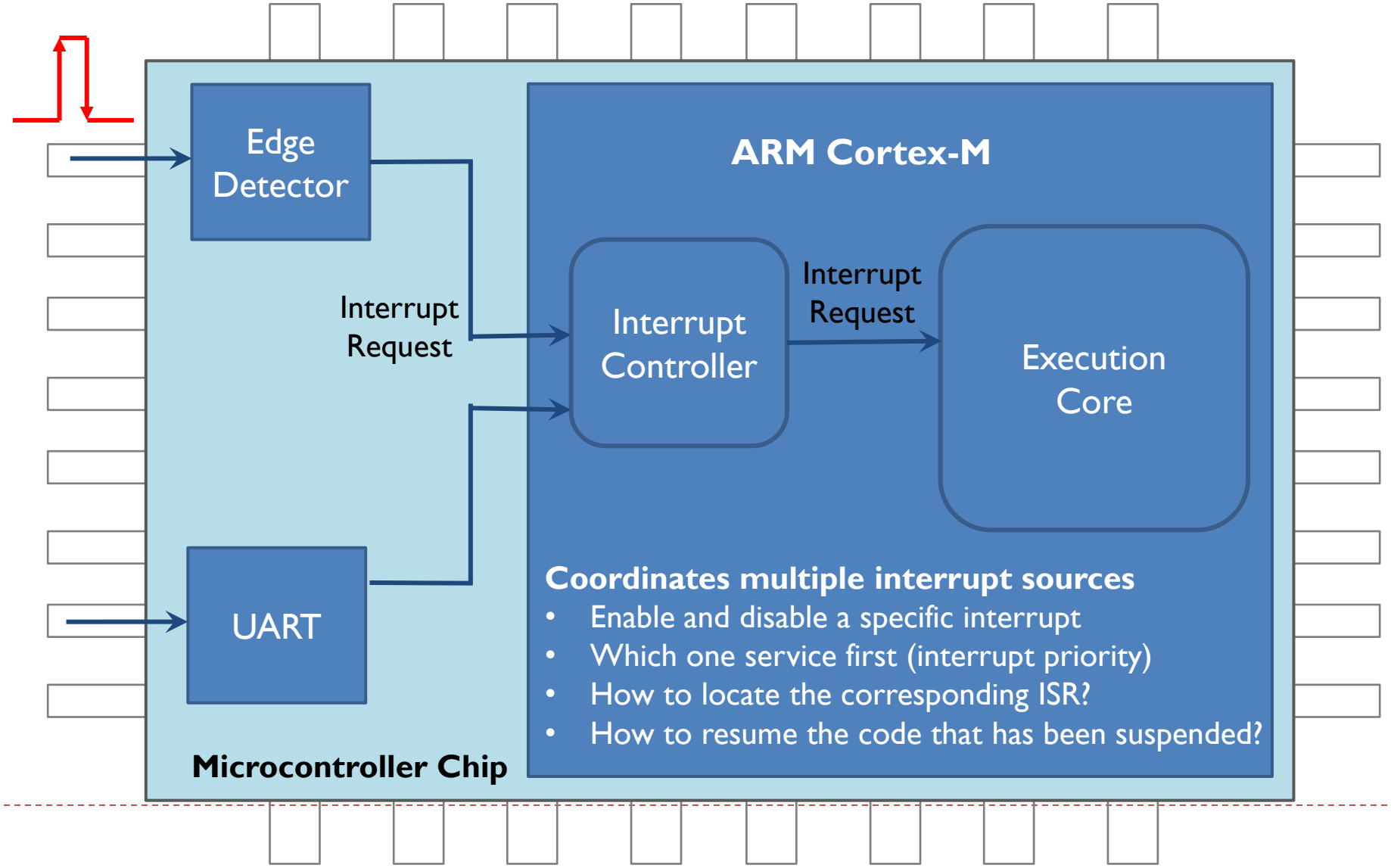
- Interrupt-driven operations
  - Allows CPU to perform other tasks until external/internal devices require service
  - CPU automatically stops the current code and starts to execute an ISR



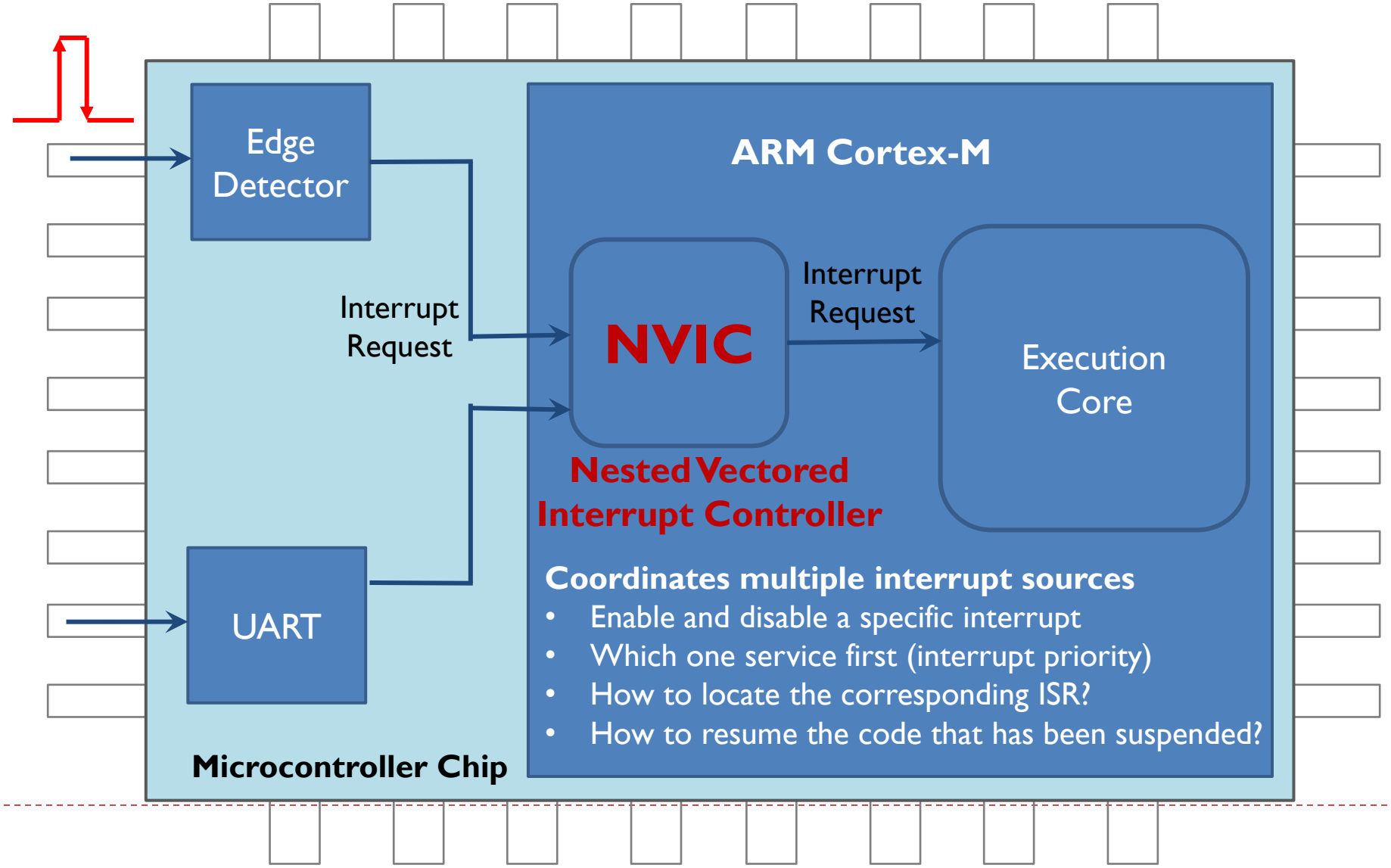
How to support interrupt?



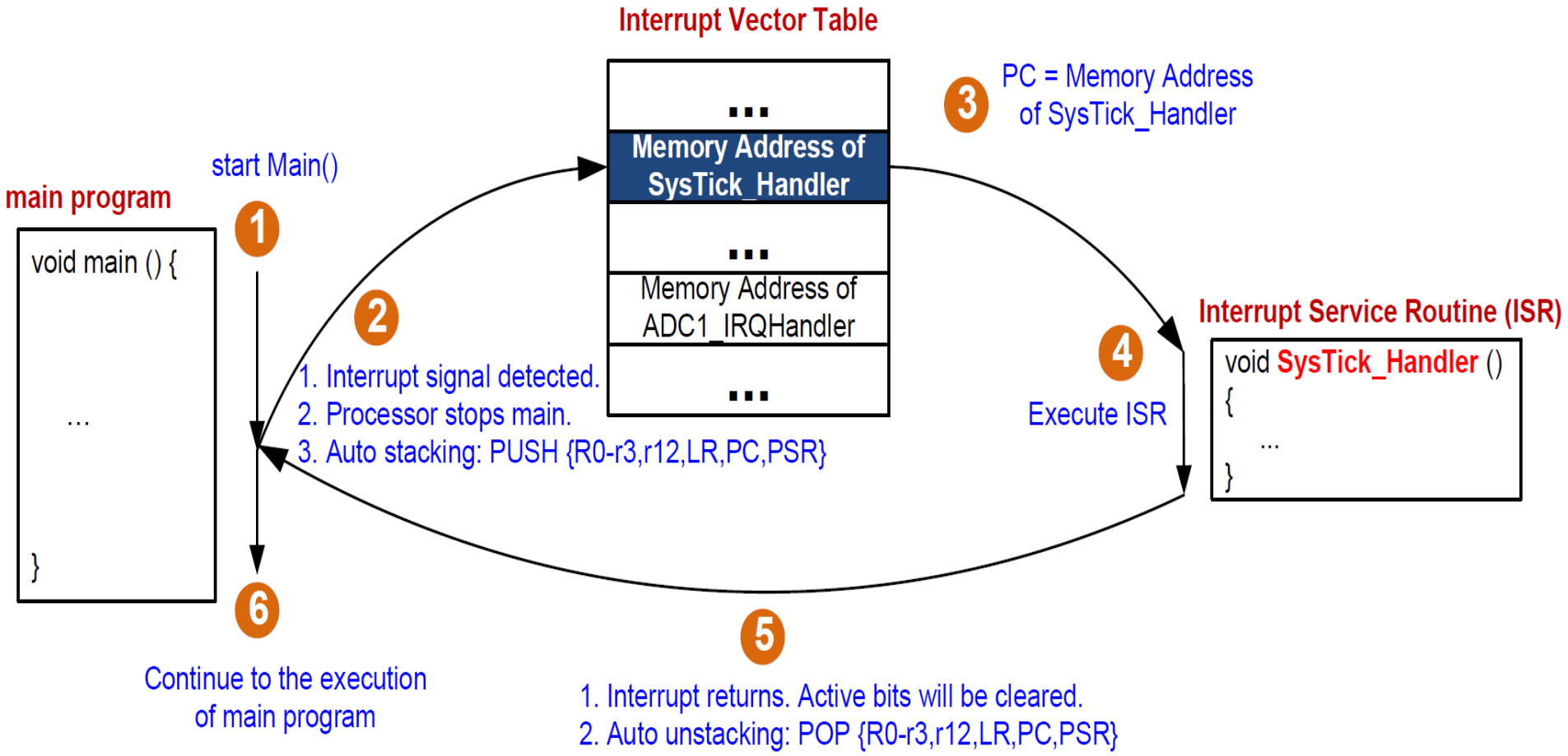
How to support interrupt?



How to support interrupt?



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}
Interrupt Service Routine (ISR)
4
Execute 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

Interrupt Service Routine Vector Table

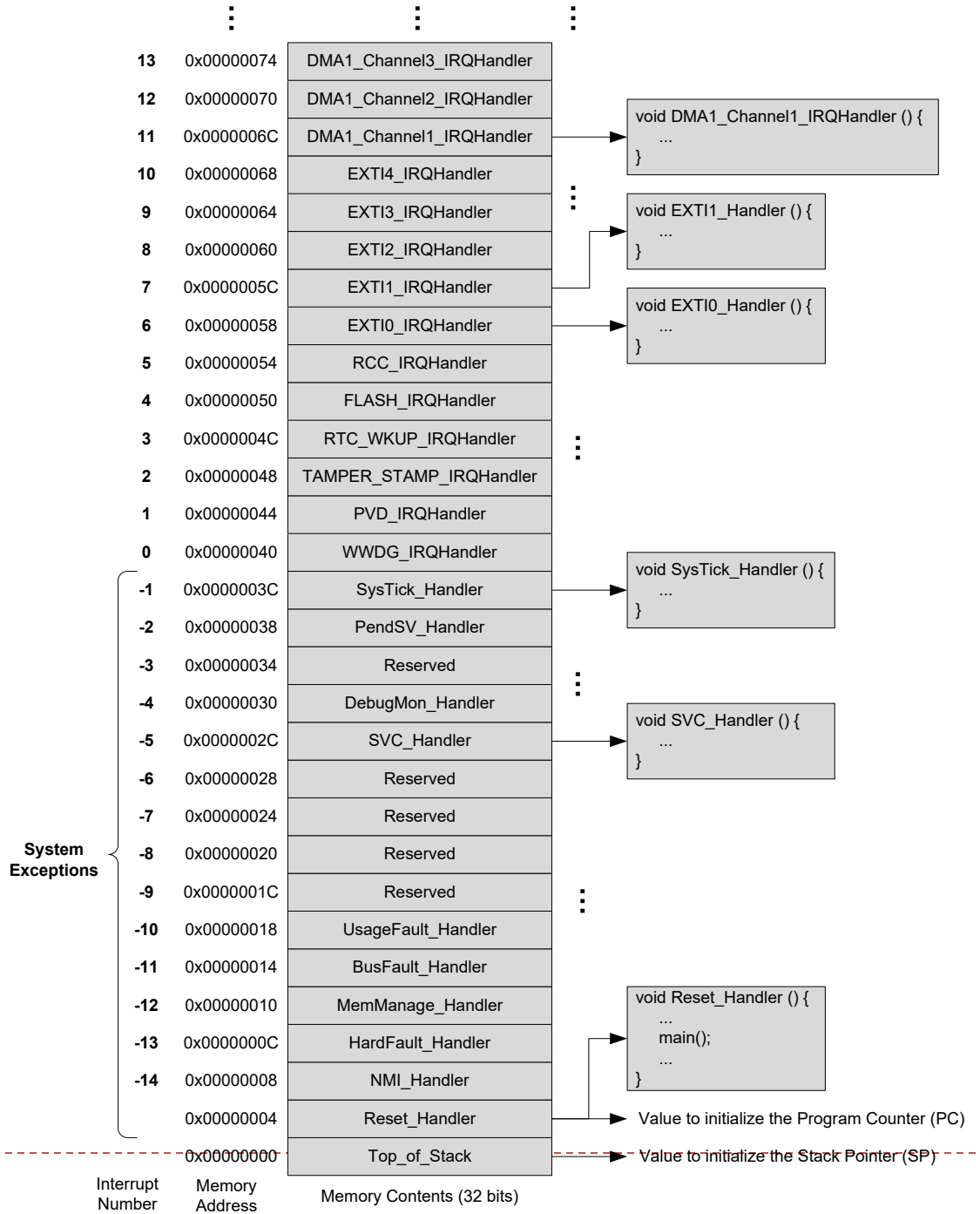
- Start address for the exception hander for each exception type is fixed and pre-defined
- Processor loads PC with this fixed, pre-defined address
- Exception Vector Table typically starts at memory address 0x00000000 (or relocated depending on Vector Table Offset Register (VTOR))
- Program Counter 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
...				

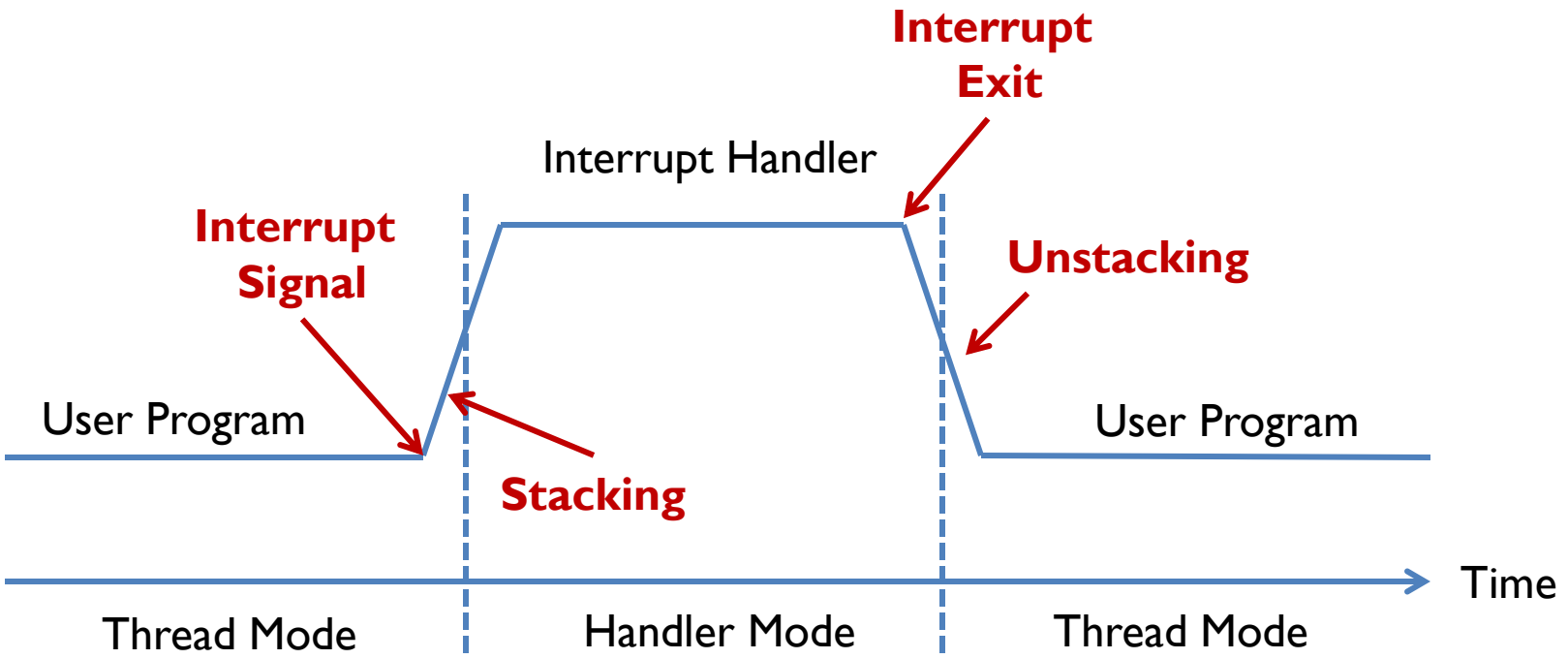
ISR Vector Table

For interrupt number **n**: (interrupt shown in the xPSR)

Common Microcontroller Software Interface Standard (CMSIS) Interrupt Number = **16 + n**



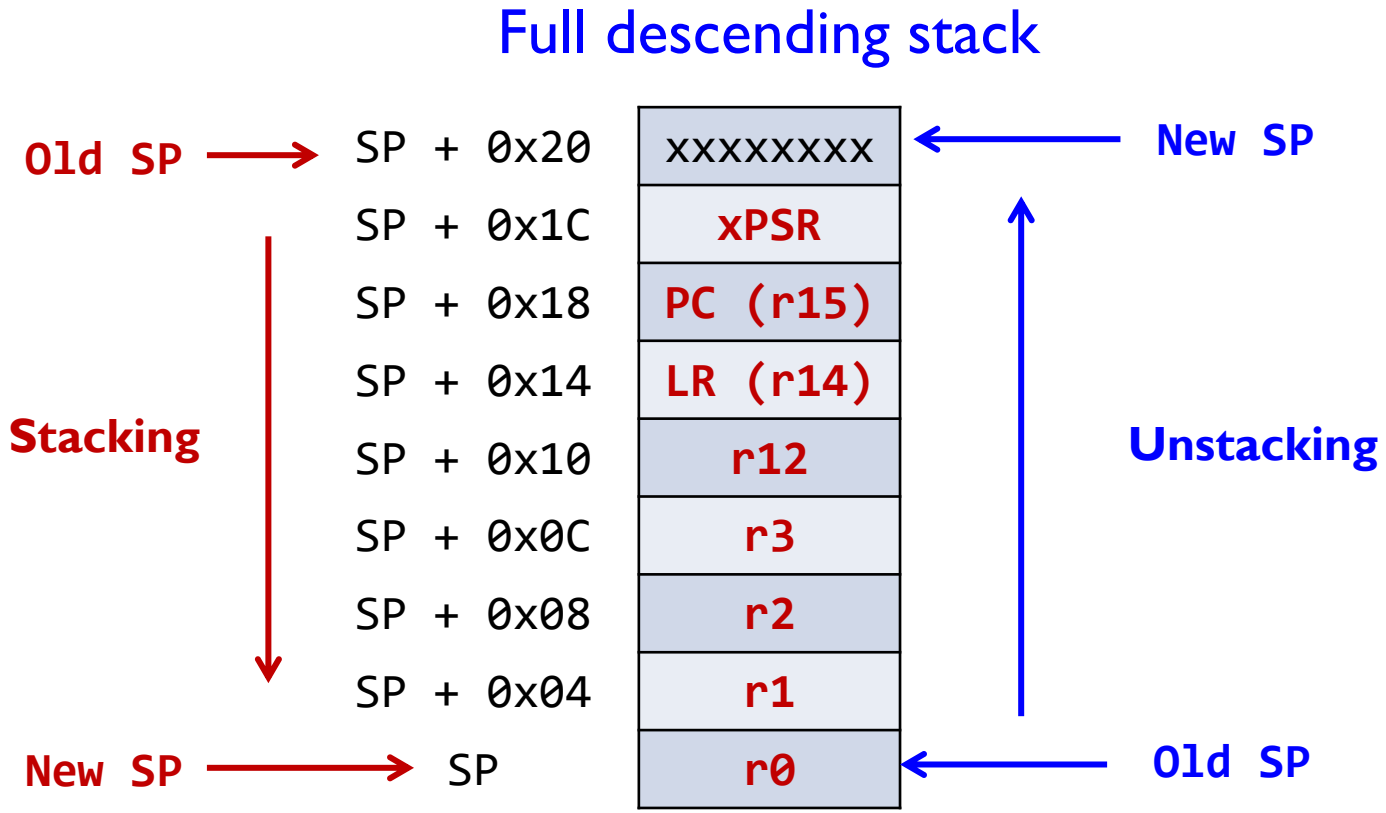
Automatic Stacking & Unstacking



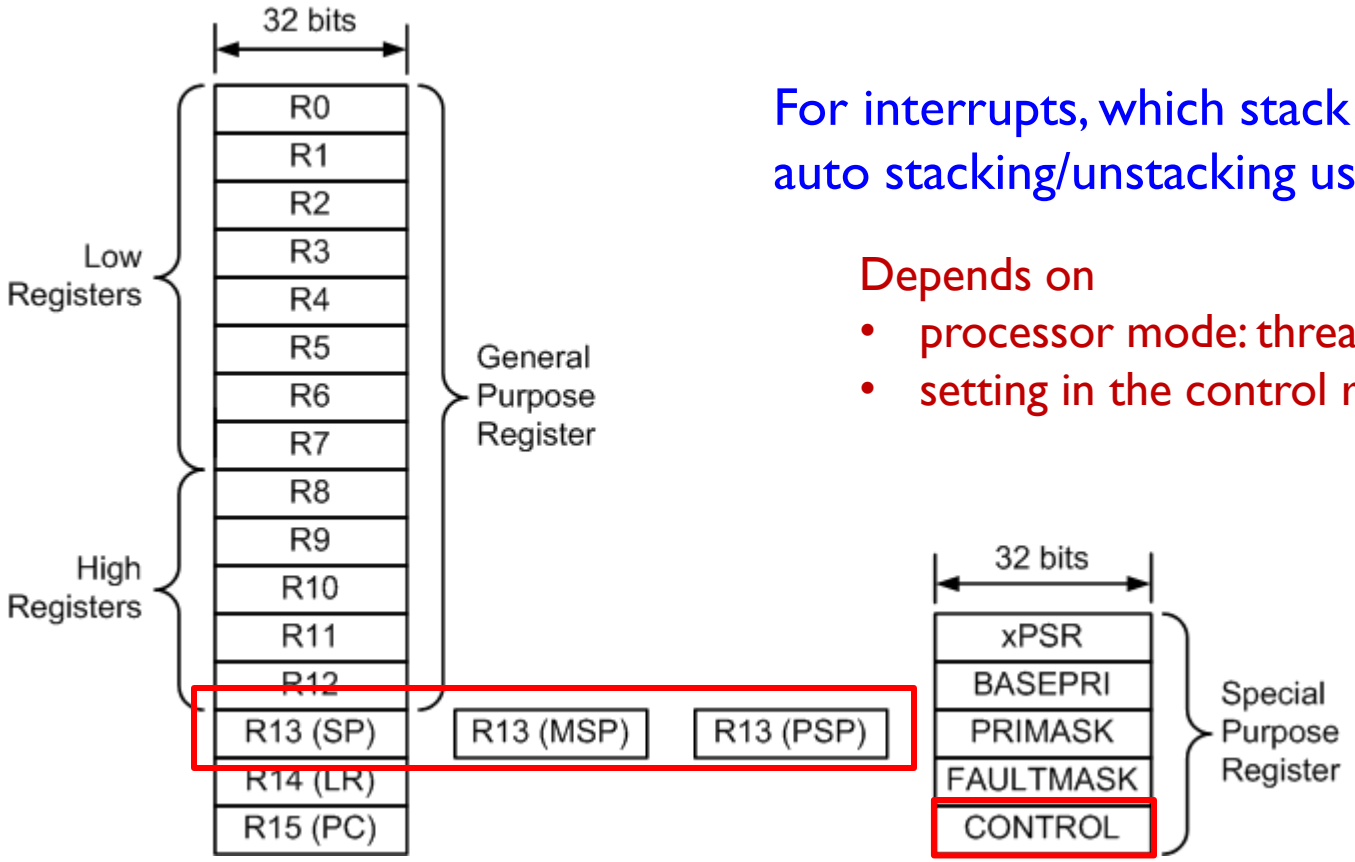
Stacking: hardware automatically pushes eight register into the stack
(xPSR,PC,LR,r12,r3,r2,r1,r0)
(additional registers if Floating Point unit is active)

Unstacking: hardware automatically pops these eight register off the stack

Automatic Stacking & Unstacking



Stack Pointer: MSP *vs* PSP



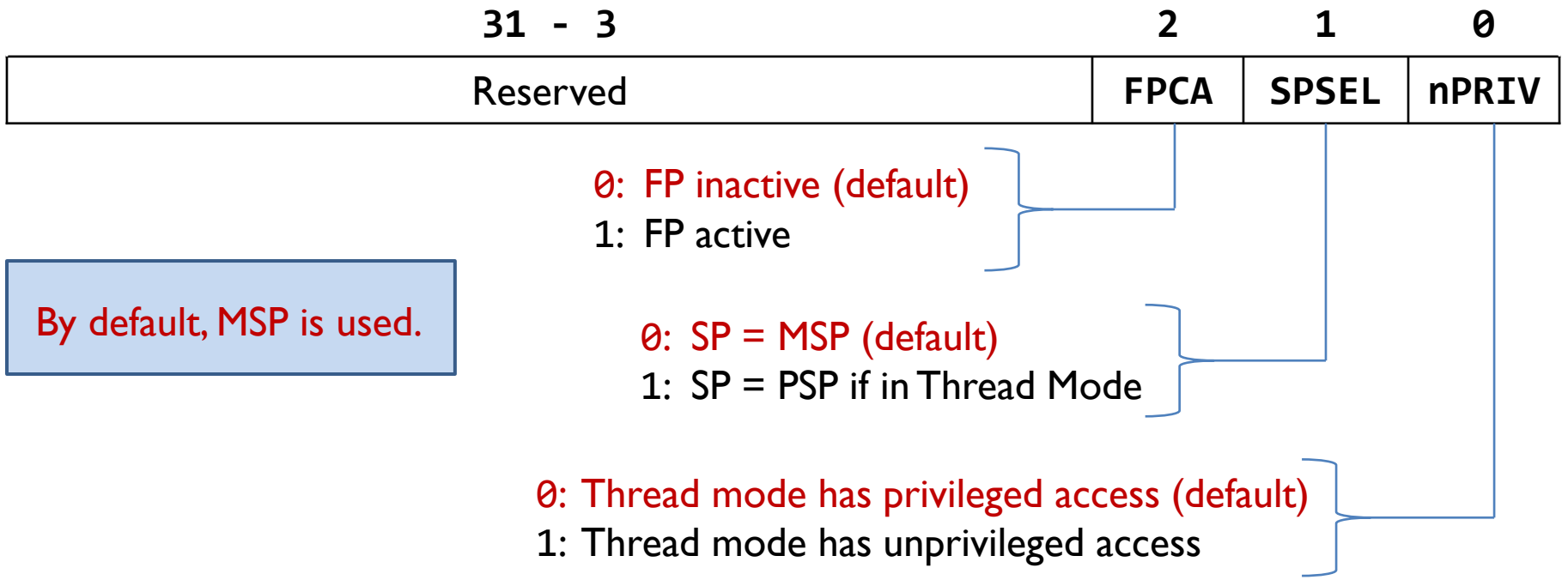
For interrupts, which stack does auto stacking/unstacking use?

Depends on

- processor mode: thread vs handler
- setting in the control register

- **MSP**: Main Stack Pointer (selected at reset)
- **PSP**: Process Stack Pointer
- R13 (SP) refers to whichever SP is active, either MSP or PSP

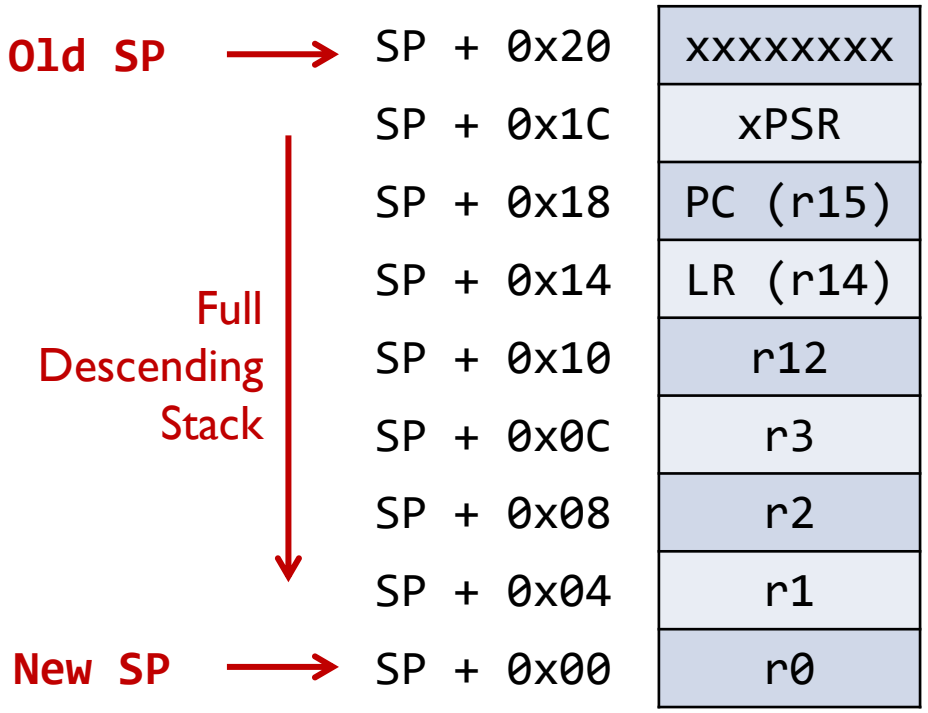
Control Register



	SP	Privileged
Handler Mode	**SP = MSP** and **SPSEL** = 0	Privileged
Thread Mode	Depending on **SPSEL**	Depending on **nPRIV**

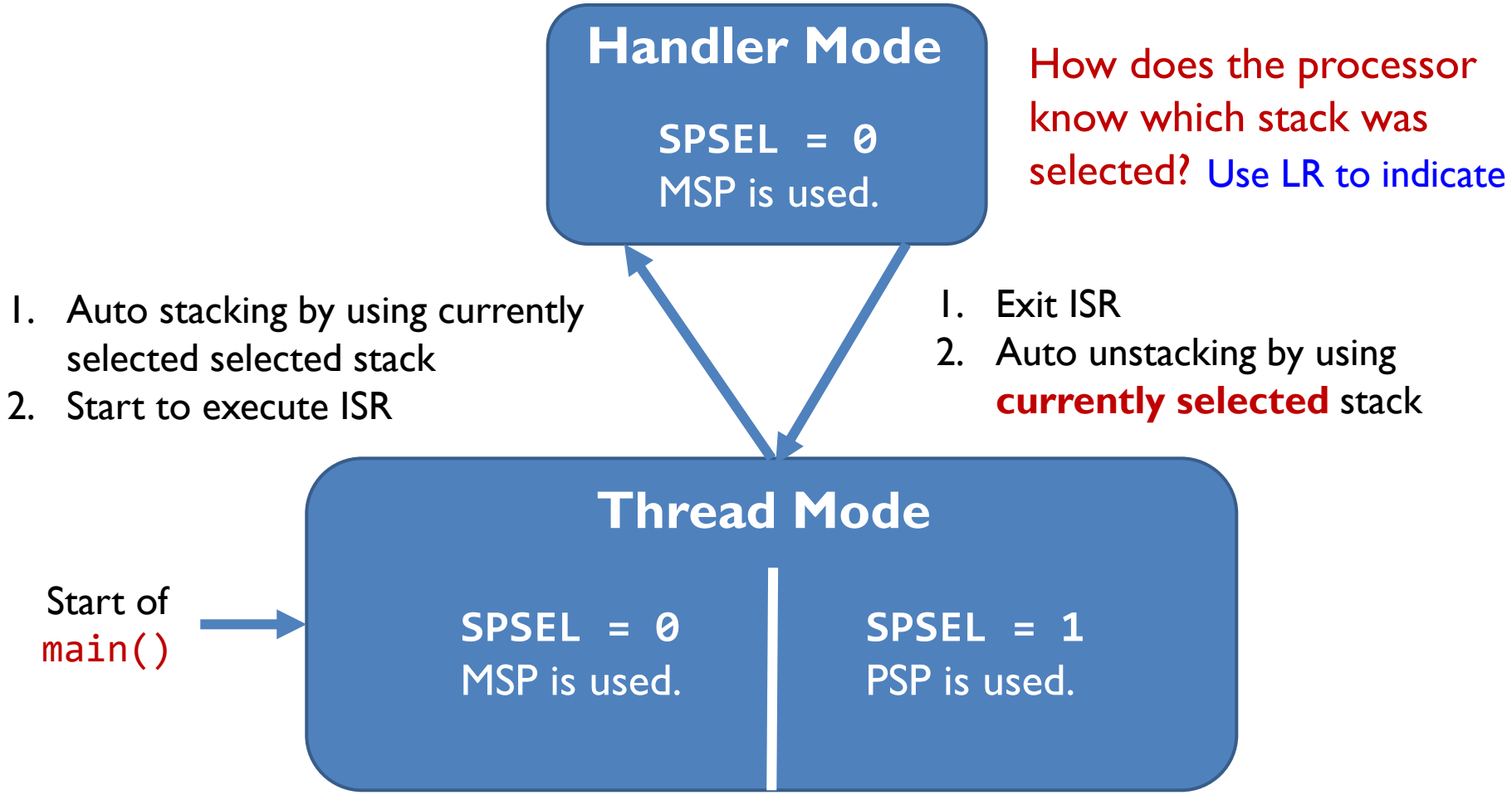
For simple applications, MSP is used.

Automatic Stacking & Unstacking

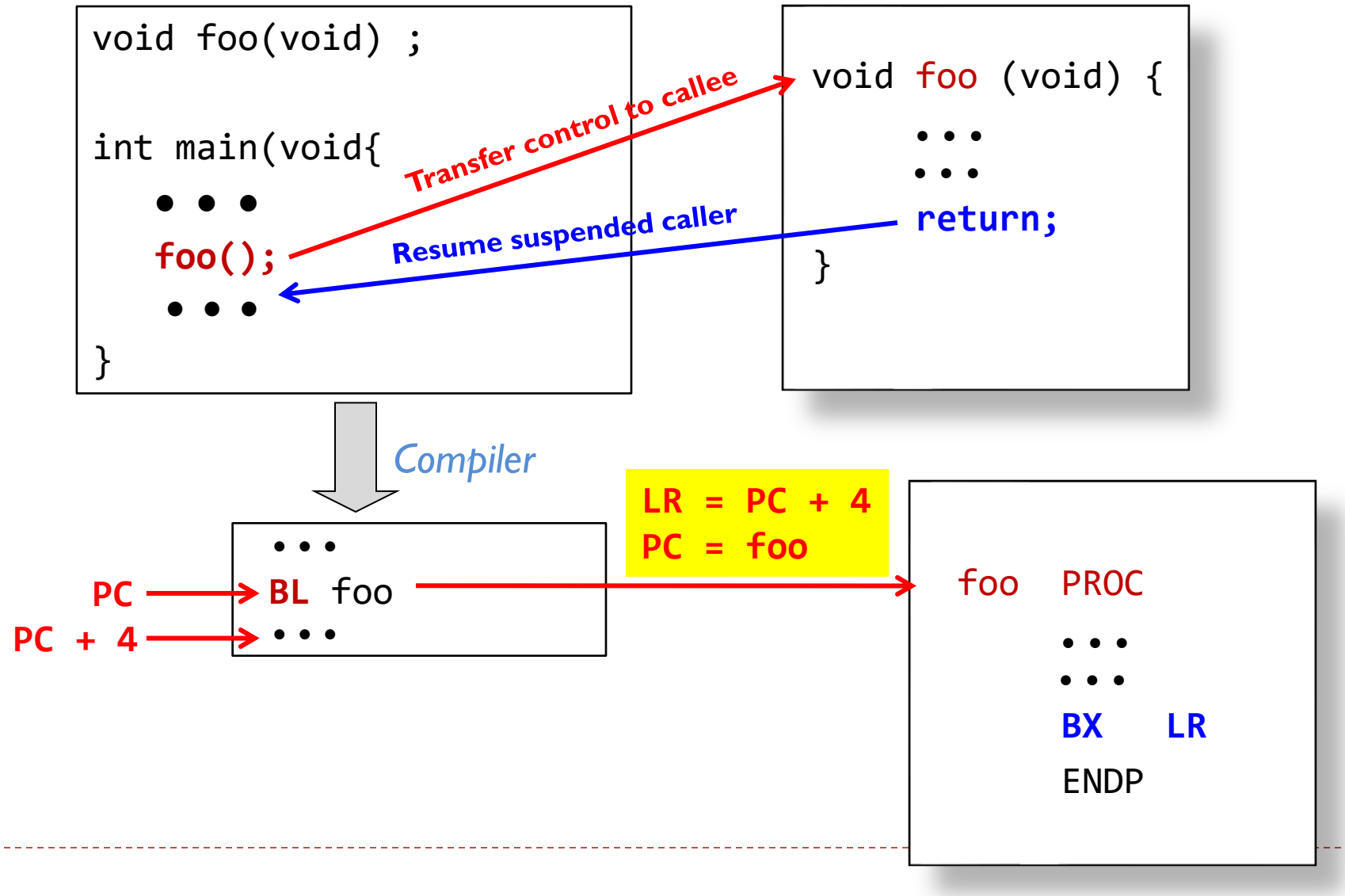


- **Stacking**: The processor automatically pushes these eight registers into the **currently selected stack** before ISR starts
- **Unstacking**: The processor automatically pops these eight register out of the **currently selected stack** when an interrupt hander exits.

MSP *vs* PSP


Handler Mode
SPSEL = 0
MSP is used.
How does the processor know which stack was selected? Use LR to indicate
1. Auto stacking by using currently selected selected stack
2. Start to execute ISR
1. Exit ISR
2. Auto unstacking by using currently selected stack
Thread Mode
Start of
main()
SPSEL = 0
MSP is used.
SPSEL = 1
PSP is used.

Recall: Link Register for calling functions



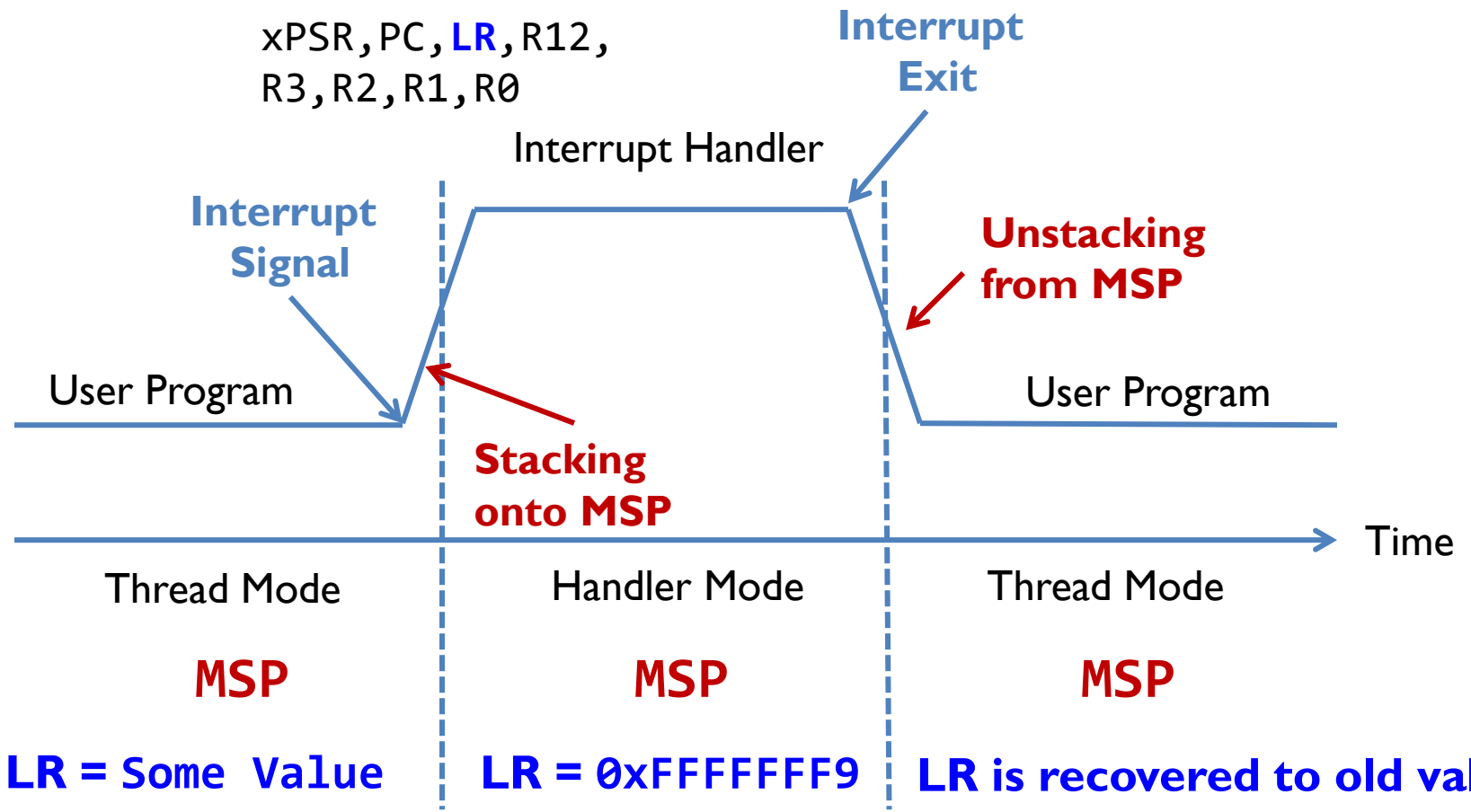
Which stack to use when an interrupt returns?

- When an interrupt (ISR) occurs on an ARM Cortex-M core:
  - The CPU automatically saves (stacks) part of the current context — registers R0-R3, R12, LR, PC, and xPSR — to the stack.
  - It then loads the ISR's address into PC, and sets LR to a special EXC_RETURN value — not a normal function return address.
- Link Register (LR) now has two usages:
  - For subroutine calls: LR holds the return address (the instruction after BL).
  - For interrupts: LR holds EXC_RETURN value indicating how to restore context when exiting the interrupt. The CPU recognizes this special pattern (bits [31:28] = 0xF) and performs an exception return sequence rather than a regular branch. It returns to the original PC before the interrupt occurred.
    - Thread mode: Normal program execution; Handler mode: ISR execution.
    - If an interrupt occurs while already in Handler mode (nested interrupt), LR is set to a different EXC_RETURN value (0xFFFFFFF1) to indicate that the CPU should return to Handler mode when that ISR completes.

EXC_RETURN value	Meaning
0xFFFFFFF1	Return to Handler mode, using MSP (for nested interrupts)
0xFFFFFFF9	Return to Thread mode, using MSP
0xFFFFFFFD	Return to Thread mode, using PSP

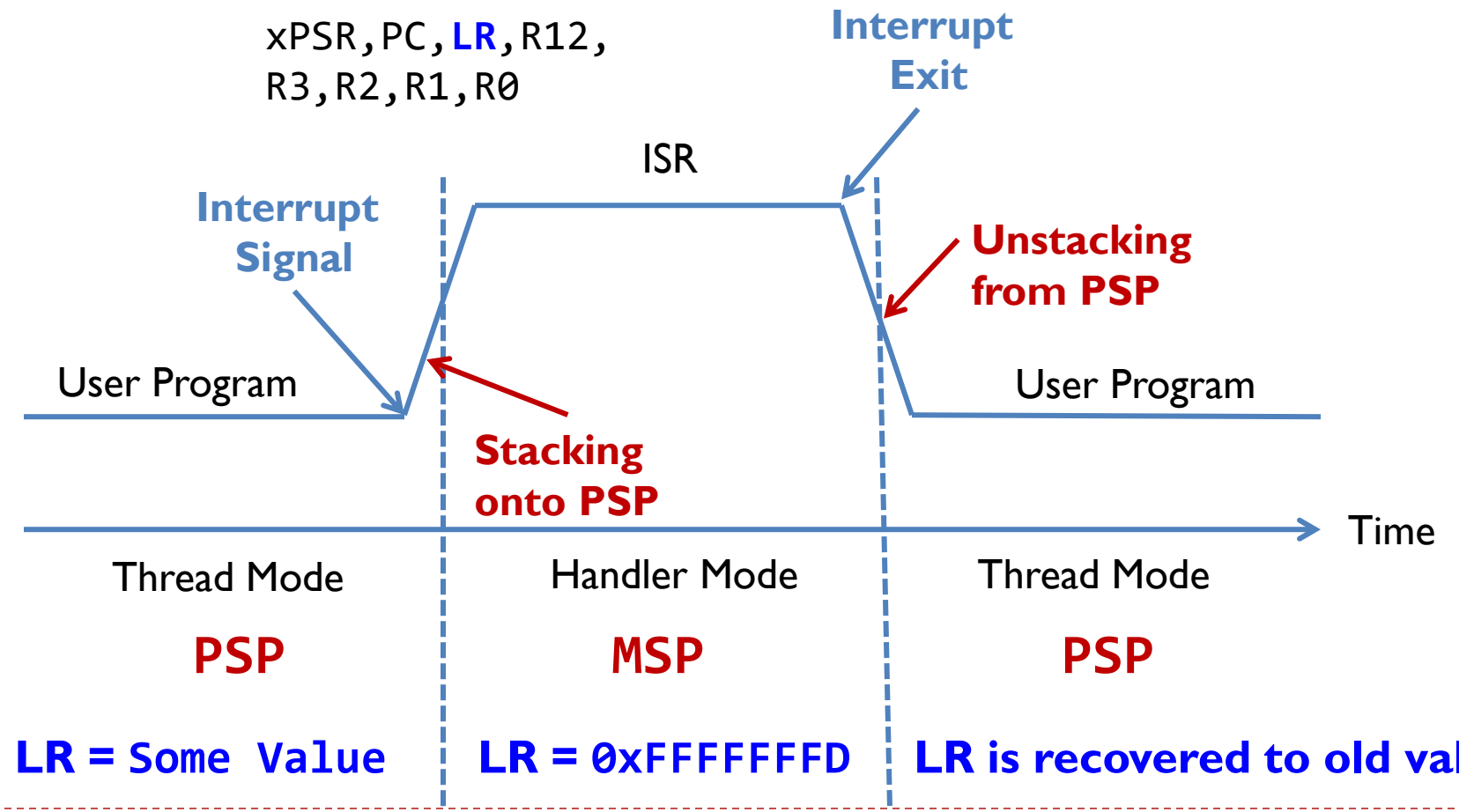
Stacking & Unstacking

Assume SPSEL = 0 and no FP is used ⟹ User program uses MSP.



Stacking & Unstacking

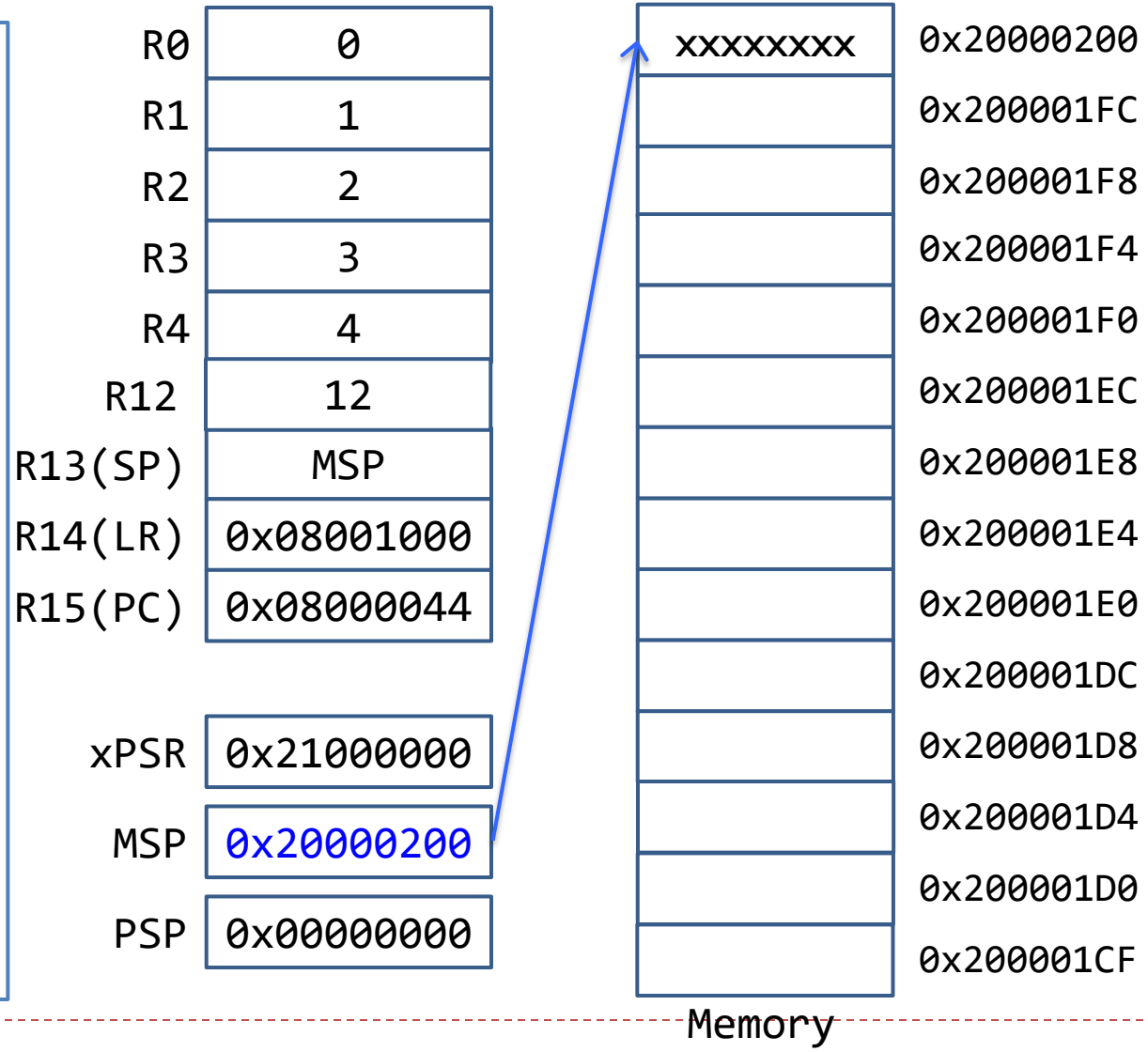
Assume SPSEL = 1 and no FP is used ⟹ User program uses PSP.



An example to illustrate stacking and unstacking (assuming MSP is used by the main program)

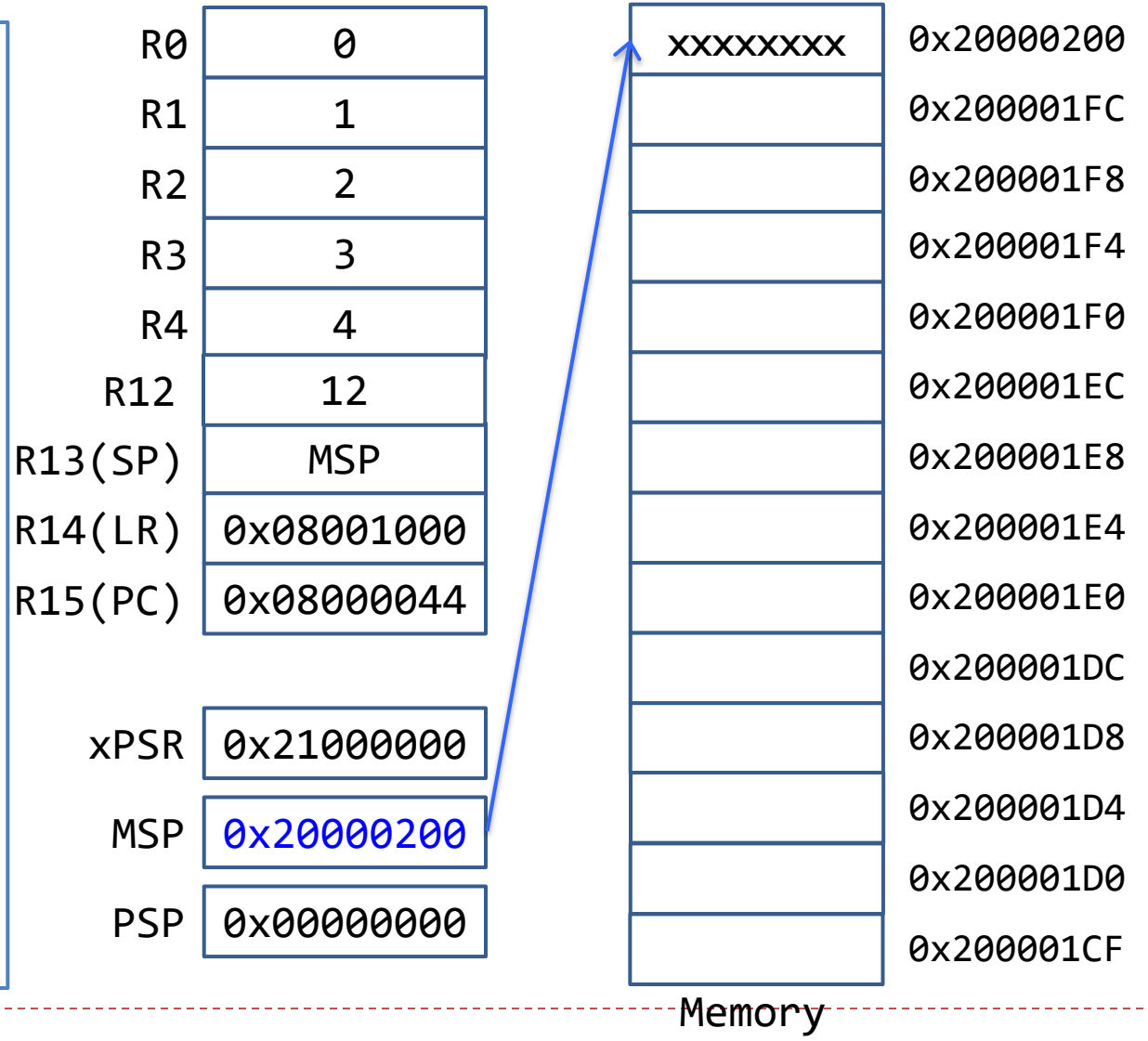
Interrupt: Stacking & Unstacking

```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r4, #1   0x0800001C
   BL  sine     0x08000020
   BX  lr       0x08000024
   ENDP
```



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
```



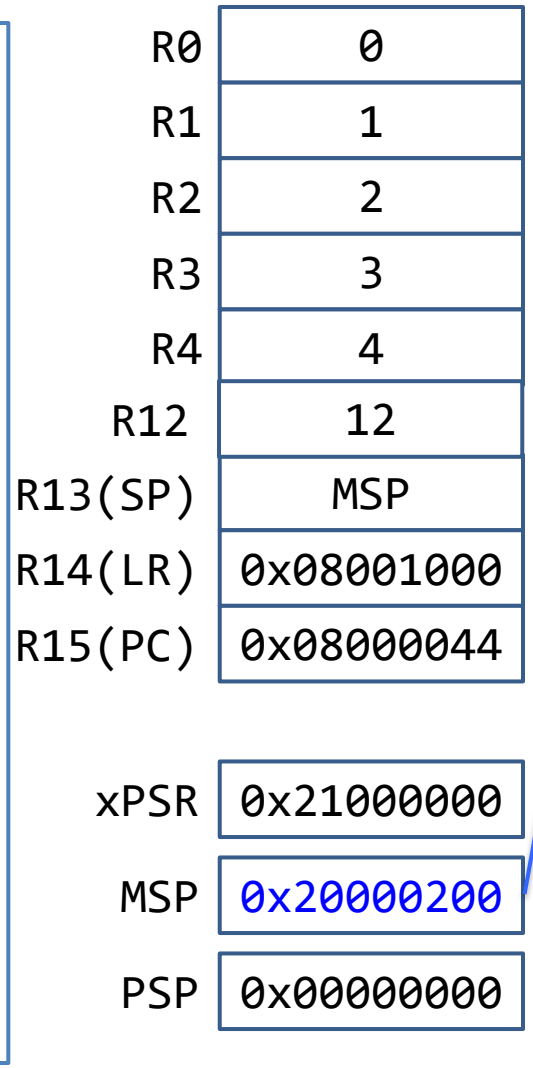
Interrupt:

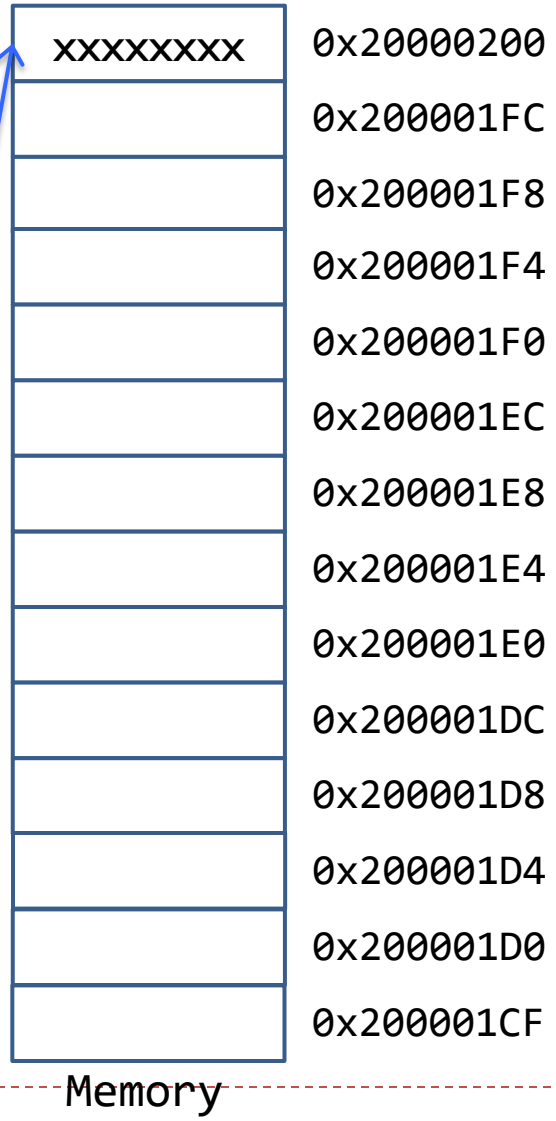
Suppose SysTick interrupt occurs when PC = 0x08000044

```
__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
   …
   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)	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

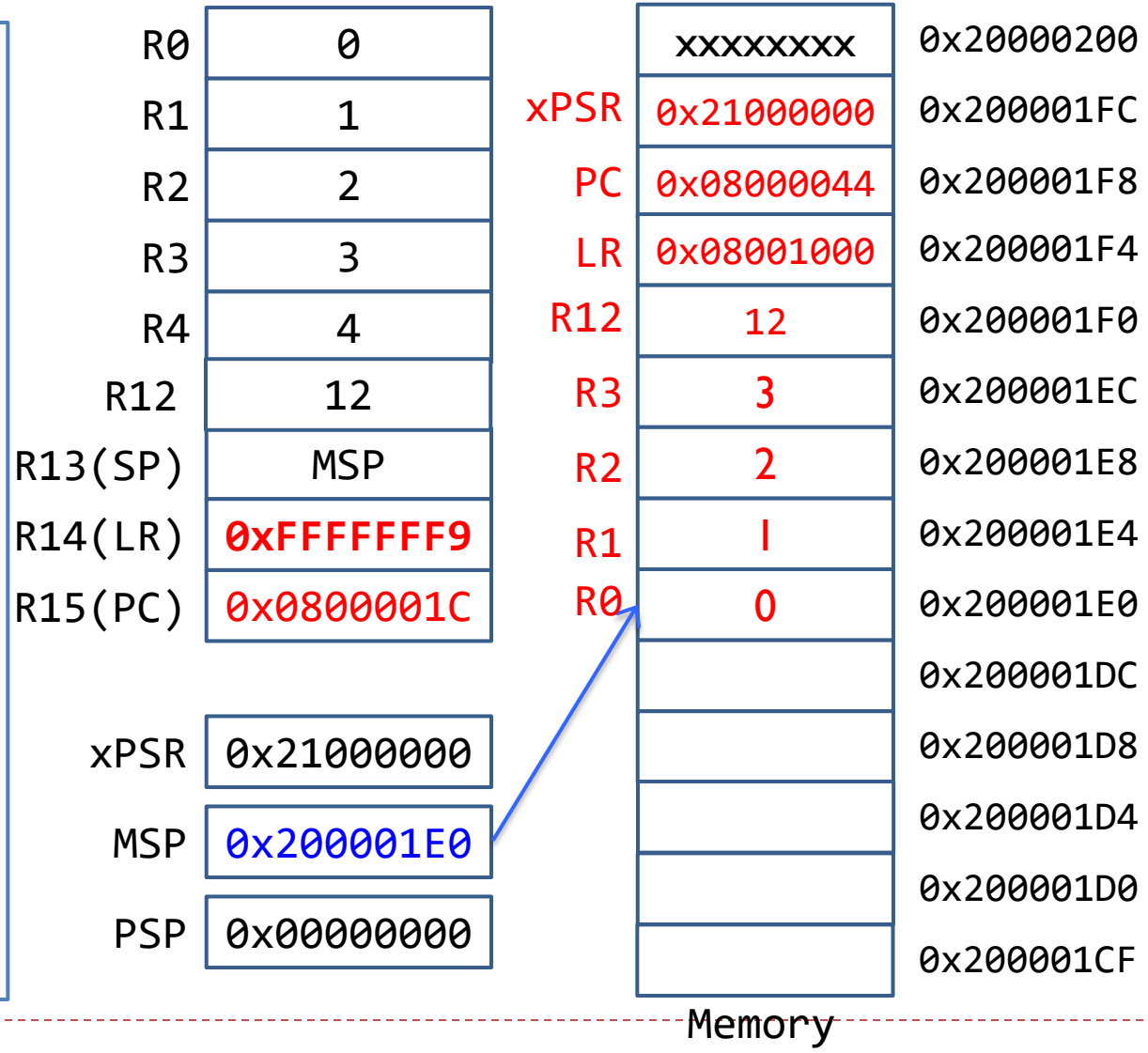
Memory

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.

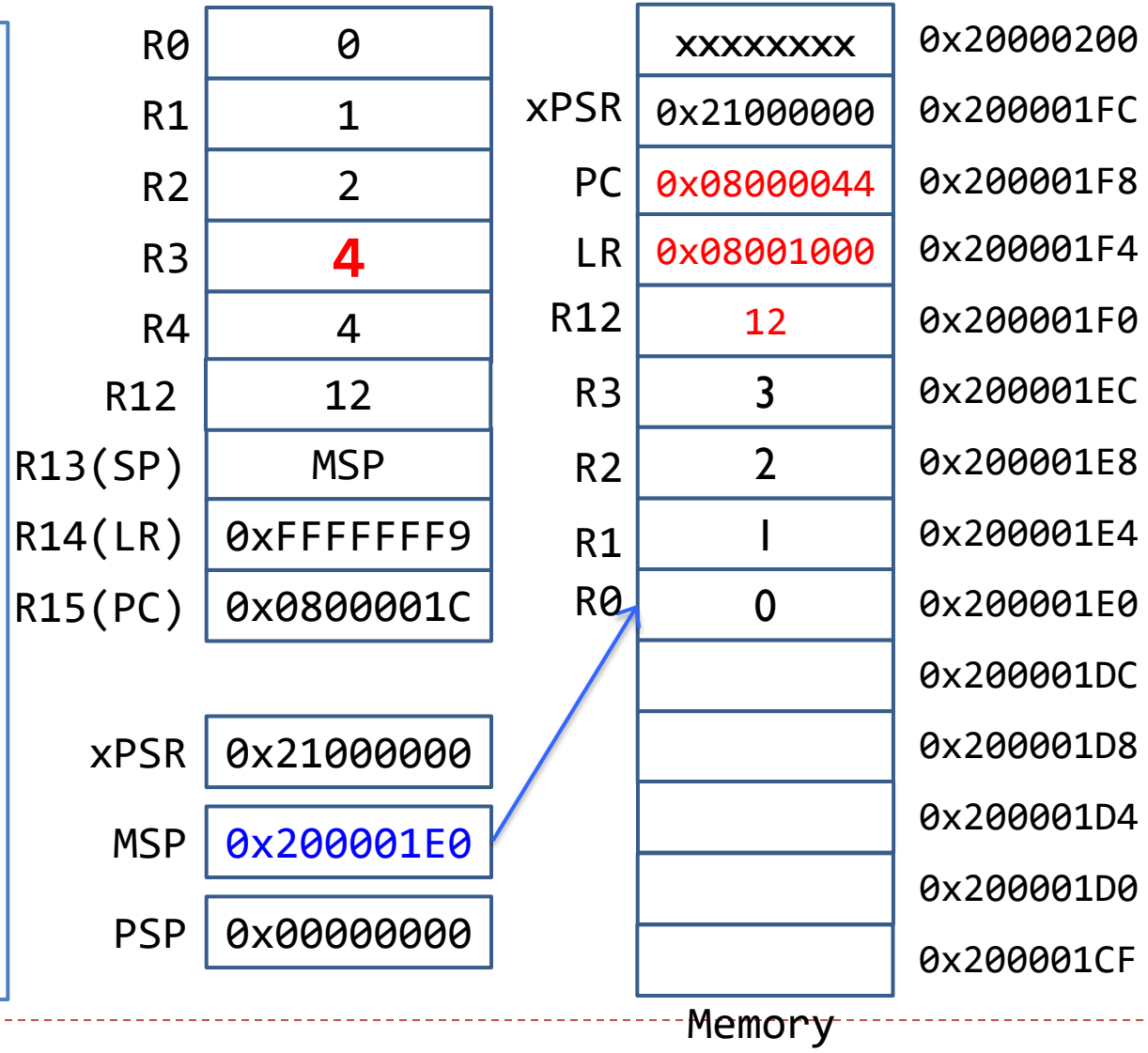


Interrupt: Stacking & Unstacking

```
__main PROC
   …
   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 & 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

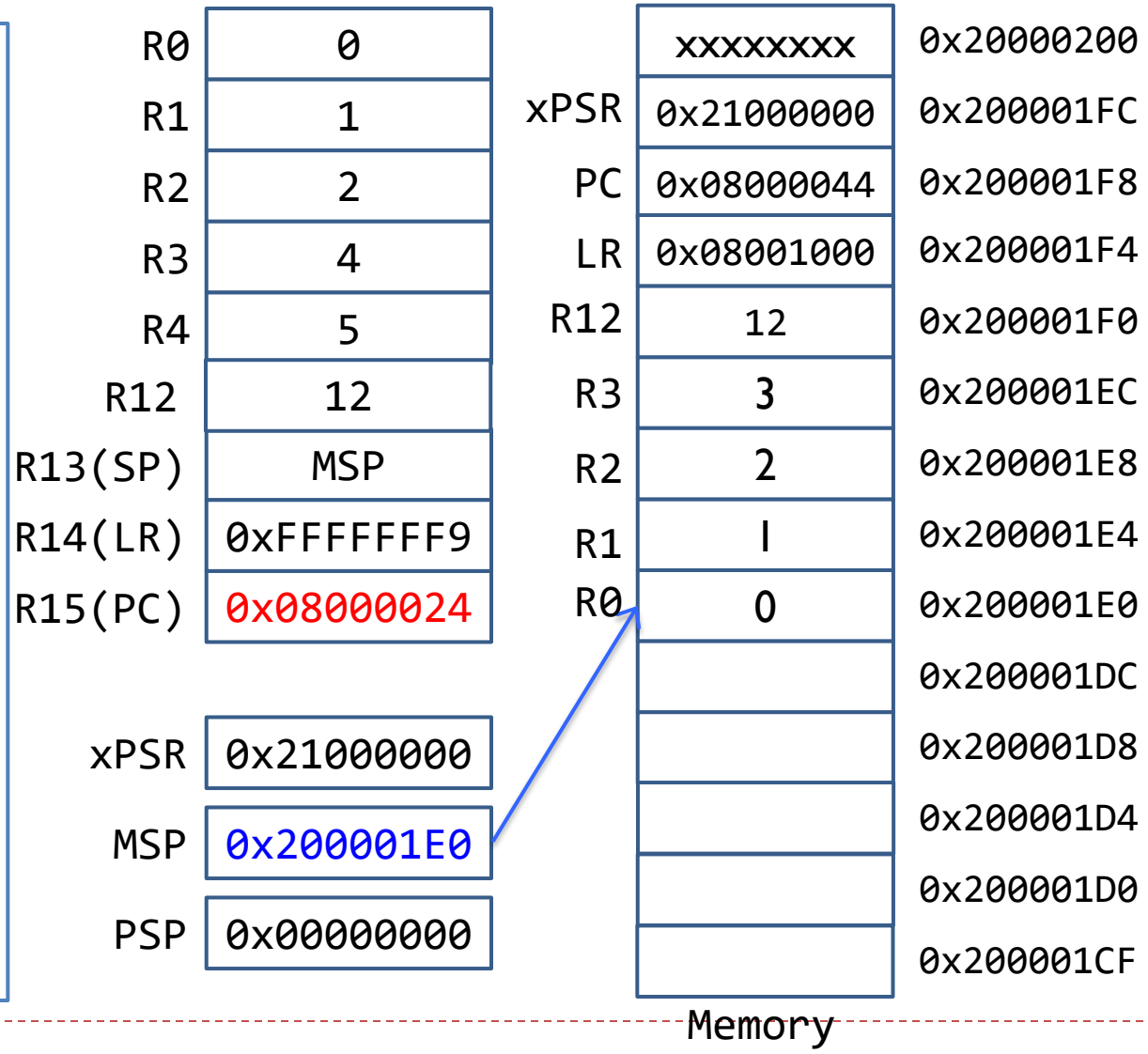
	xxxxxxxx	0x20000200
xPSR	0x21000000	0x200001FC
PC	0x08000044	0x200001F8
LR	0x08001000	0x200001F4
R12	12	0x200001F0
R3	3	0x200001EC
R2	2	0x200001E8
R1	1	0x200001E4
R0	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

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
```

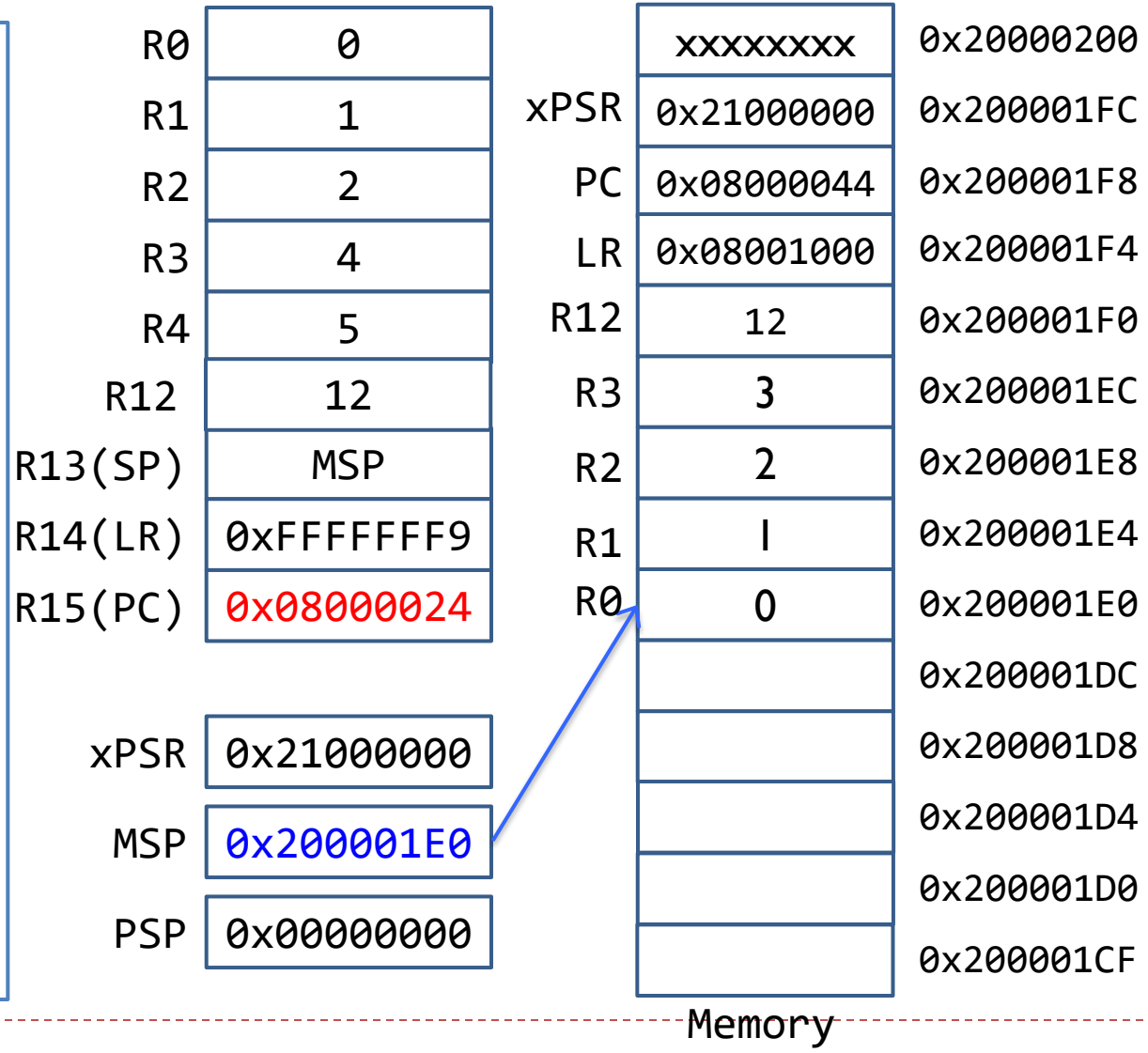
LR = 0xFFFFFFF9 to indicate MSP is used.



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
```

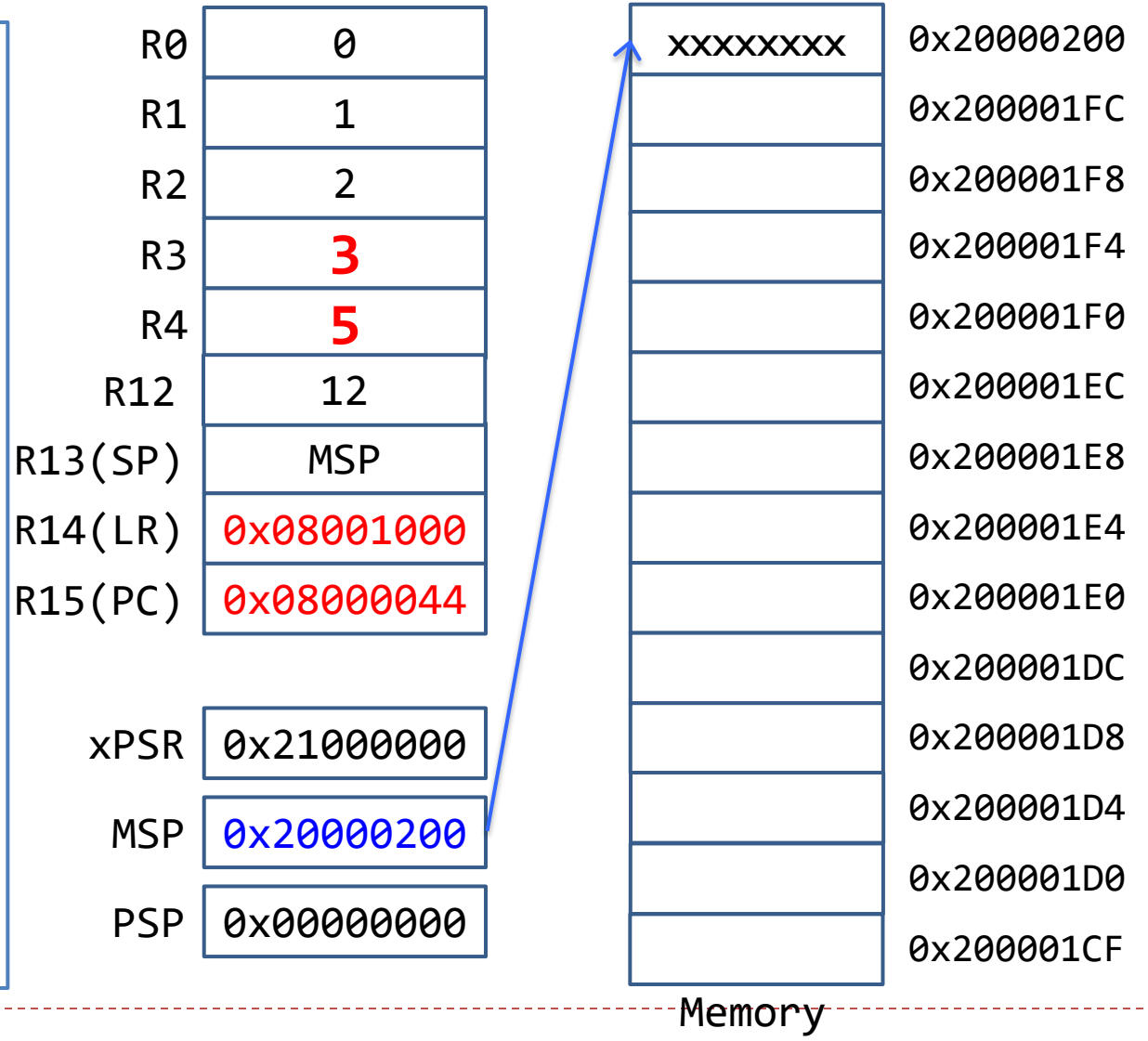
LR = 0xFFFFFFF9 to indicate MSP is used.



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
```

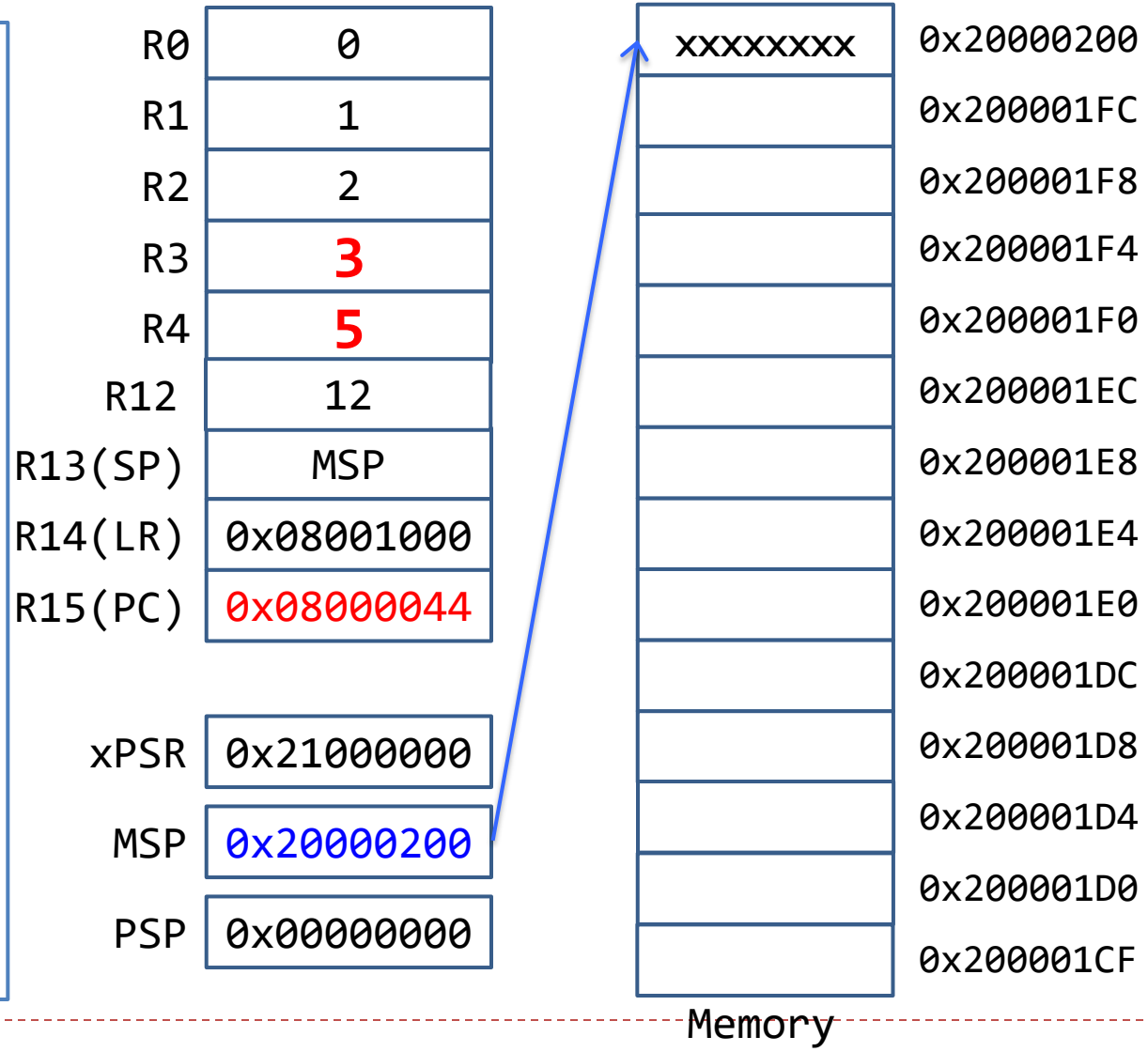
Note the new value of R3 is lost!!!



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!!!



Explanations

- The value of R3 updated in ISR is lost because during the interrupt unstacking (return from interrupt), the processor restores R3 (and other registers) from the stack — overwriting whatever changes were made to R3 inside ISR.
- 1. Before the interrupt (in main)
  - The main program is executing at R3 = 0 (set by MOV r3, #0)
- 2. Interrupt occurs (SysTick_Handler)
  - The processor automatically pushes (stacks) certain registers onto the stack: xPSR,PC,LR,R12,R3,R2,R1,R0
- 3. Inside ISR
  - R3 = 3+1 = 4, R4 = 4+1 = 5 — but these are local to the ISR
- 4. Returning from interrupt (unstacking)
  - When BX lr executes with LR = 0xFFFFFFF9, the CPU knows: "Return to Thread mode using MSP." It then automatically pops (unstacks) the previously saved registers. This restores the pre-interrupt state:R3 ← (the saved value from the stack, which was 3 before the interrupt); R4 remains 5 (not part of automatic stacking/unstacking); PC ← 0x08000044 (returns to main); xPSR restored.

Step	Action	Effect on R3
Before interrupt	R3 = 3 in main	—
Interrupt entry	R3 = 3 stacked	Saved on MSP
ISR executes	R3 = 4 (incremented)	Temporary
ISR returns	R3 restored from stack	R3 = 3 again

Review

Nested Subroutines: Solution #1

foo saves and restores its LR for returning to its caller, before calling bar.

Caller Program	Subroutine foo	Subroutine bar

MOV r4, #100
...
BL foo
...
ADD r4, r4, #1
	foo PROC
	PUSH {r4, LR}
	...
	MOV r4, #10
	...
	BL bar
	...
	POP {r4, LR}
	BX LR
	ENDP
		bar PROC
		...
		BX LR
		ENDP

Review

Nested Subroutines: Solution #2

POP {r4, PC} is equivalent to POP {r4, LR} followed by BX LR.

Caller Program	Subroutine foo	Subroutine bar

MOV r4, #100
...
BL foo
...
ADD r4, r4, #1
	foo PROC
	PUSH {r4, LR}
	...
	MOV r4, #10
	...
	BL bar
	...
	POP {r4, PC}
	~~BX LR~~
	ENDP
		bar PROC
		...
		BX LR
		ENDP

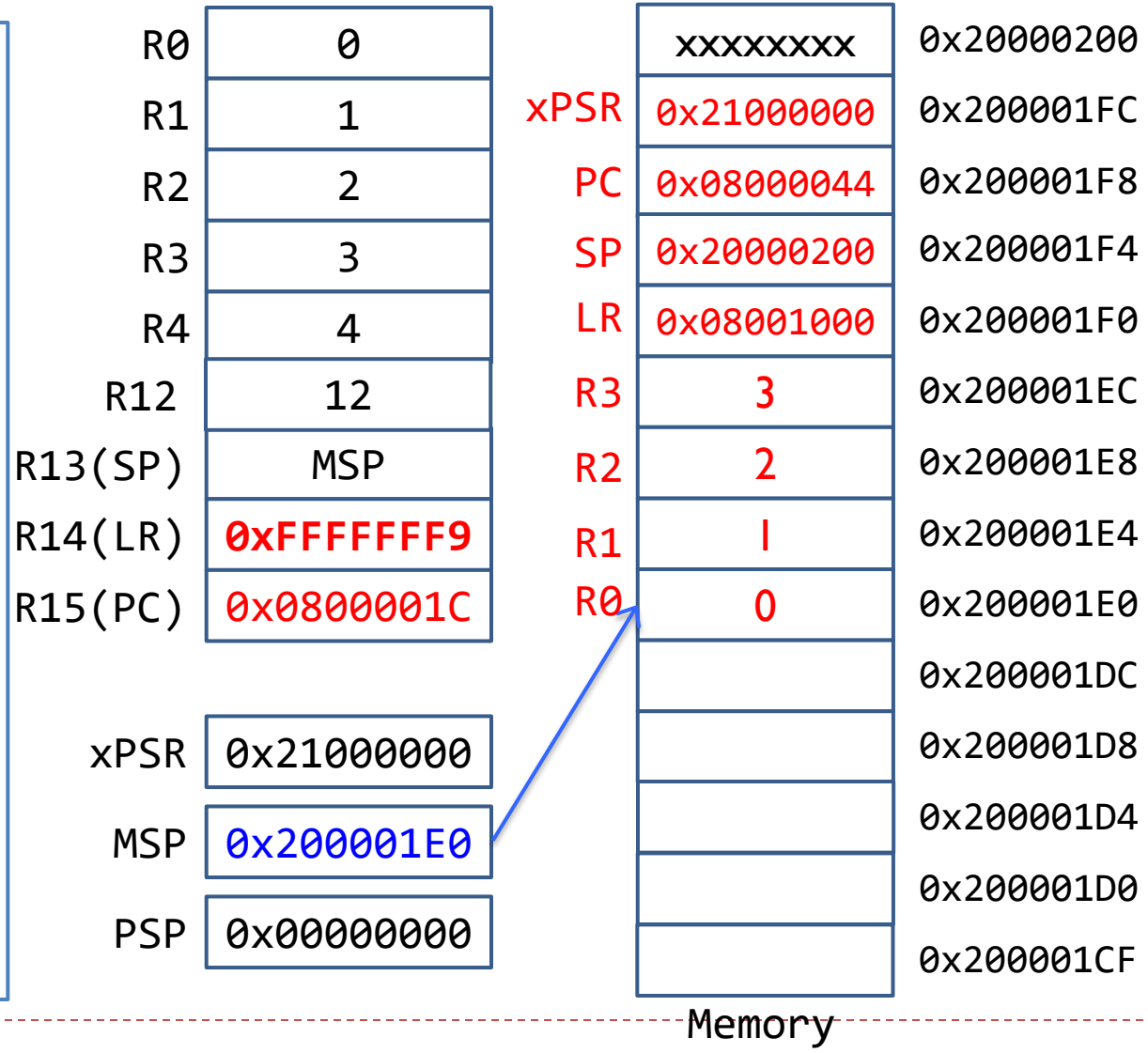
An example where an ISR calls a subroutine, similar to the nested subroutines example

Interrupt: Stacking & Uns

STACKING

```
__main PROC
                addr = 0x08000044
   …
   MOV r3,#0
   …
   ENDP
                addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r4, #1 0x0800001C
   BL  sine   0x08000020
   BX  lr     0x08000024
   ENDP
```

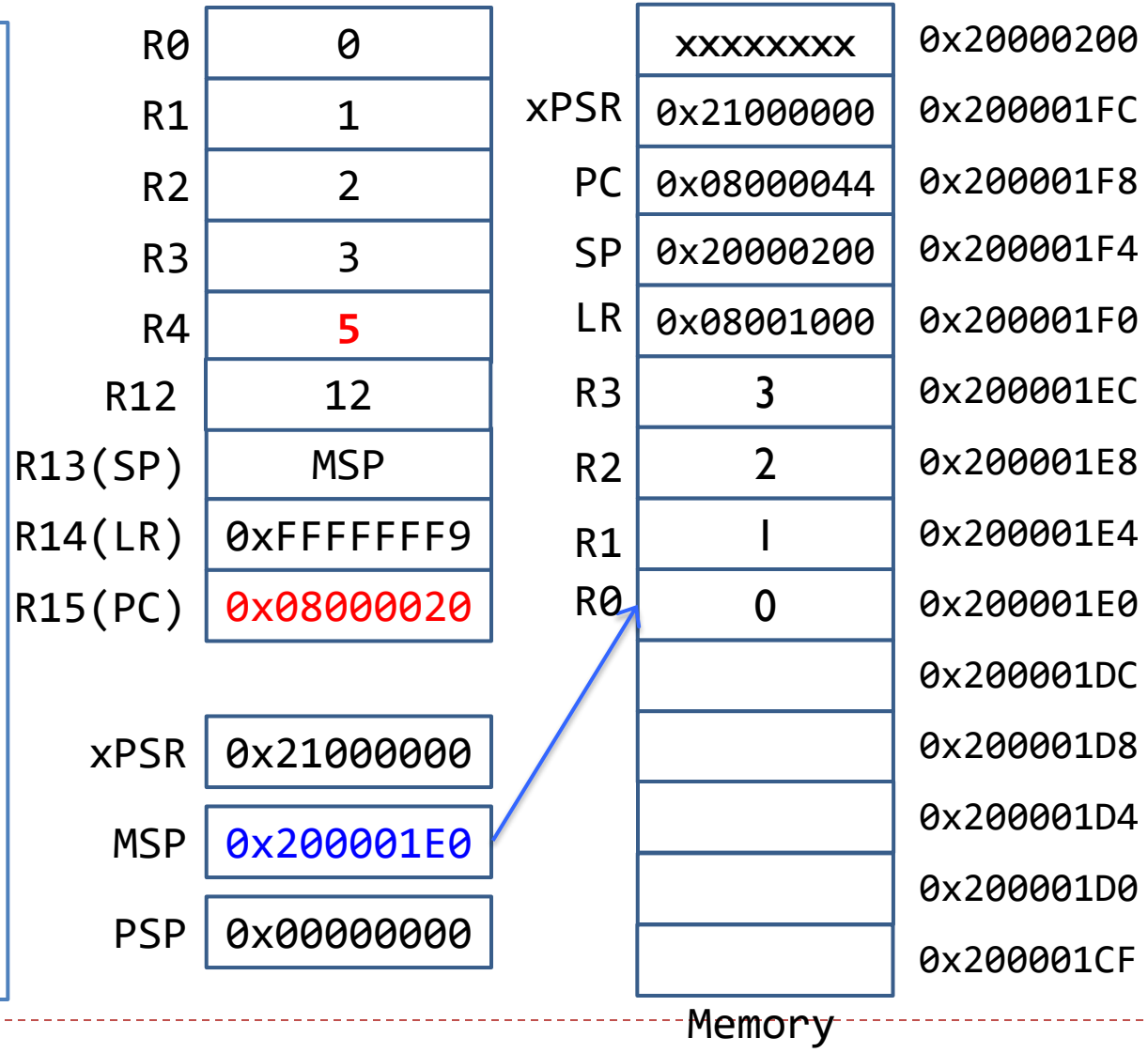
LR = 0xFFFFFFF9 to indicate MSP is used.



Interrupt: Stacking & Unstacking

```
__main PROC
                addr = 0x08000044
   …
   MOV r3,#0
   …
   ENDP
                addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r4, #1   0x0800001C
   BL  sine     0x08000020
   BX  lr       0x08000024
   ENDP
```

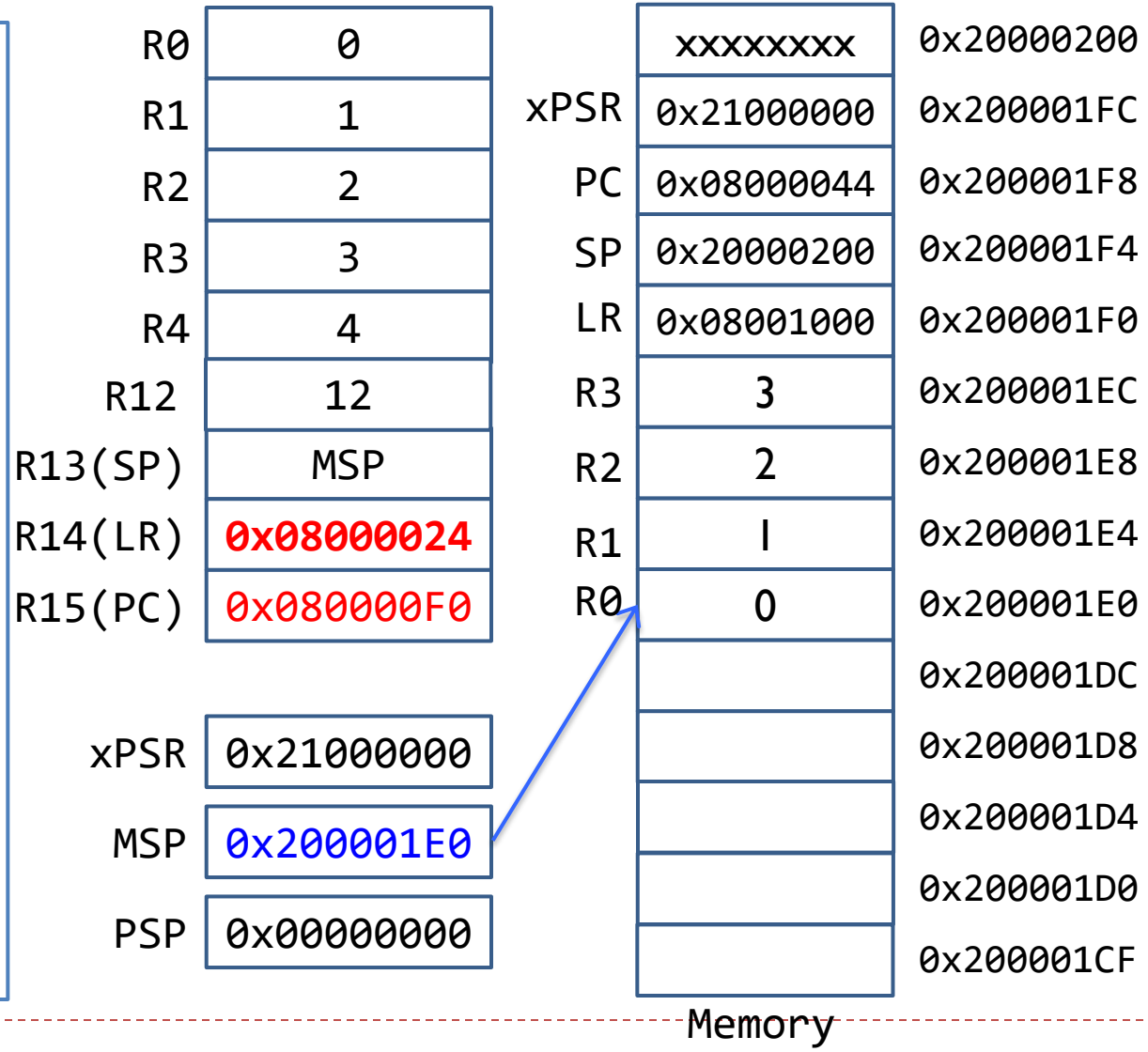
LR = 0xFFFFFFF9 to indicate MSP is used.



Interrupt: Stacking & Unstacking

```
__main PROC
                addr = 0x08000044
   …
   MOV r3,#0
   …
   ENDP
                addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r4, #1  0x0800001C
   BL  sine    0x08000020
   BX  lr      0x08000024
   ENDP
```

BL sine
Updates LR

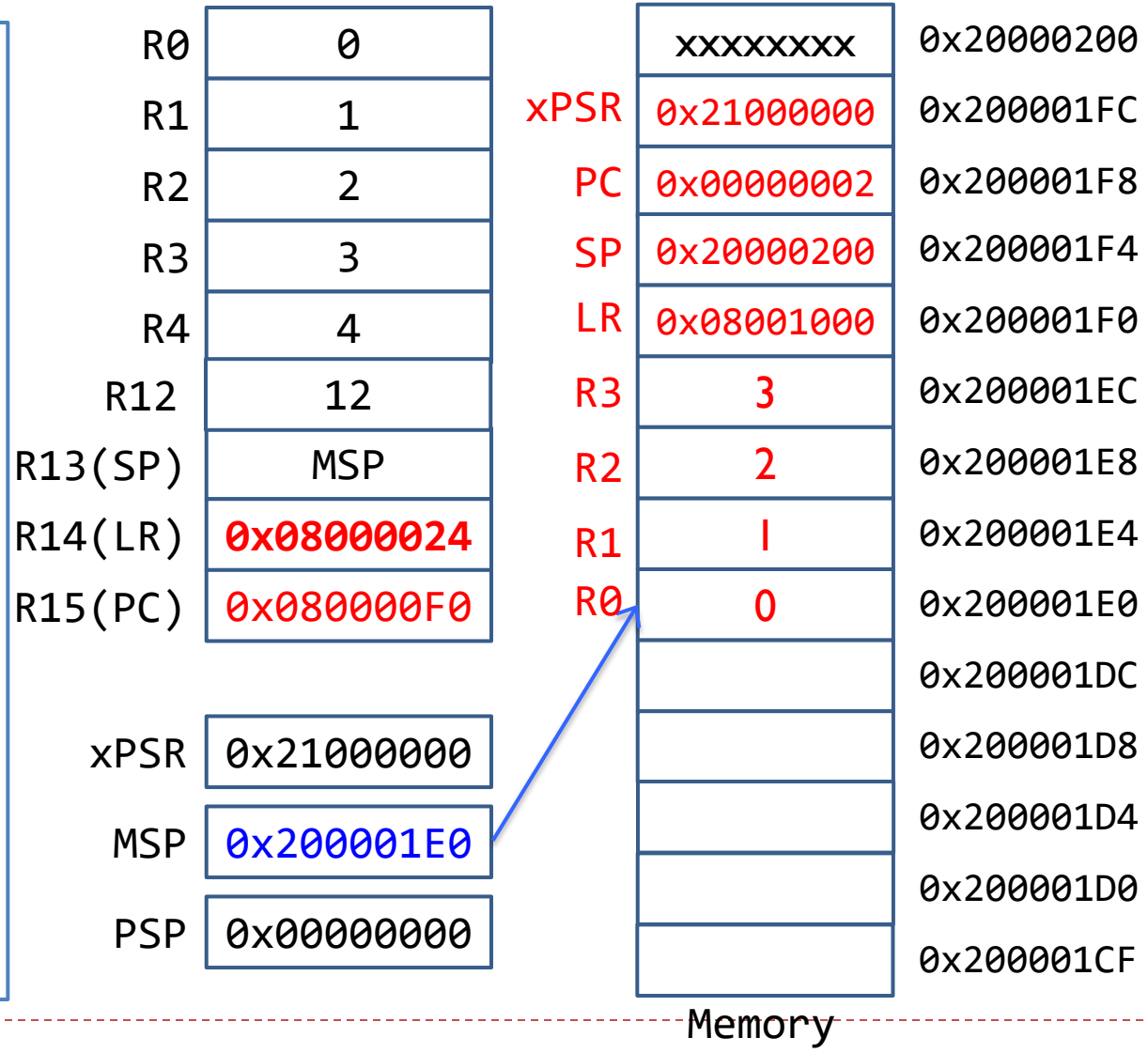


Assume sine() is located at 0x08000024.

Interrupt: Stacking & Unstacking

```
__main PROC
                addr = 0x08000044
  …
  MOV r3,#0
  …
  ENDP
                addr = 0x0800001C
SysTick_Handler PROC
  EXPORT SysTick_Handler
  ADD r4, #1  0x0800001C
  BL  sine    0x08000020
  BX  lr      0x08000024
  ENDP
```

BL sine
Updates LR

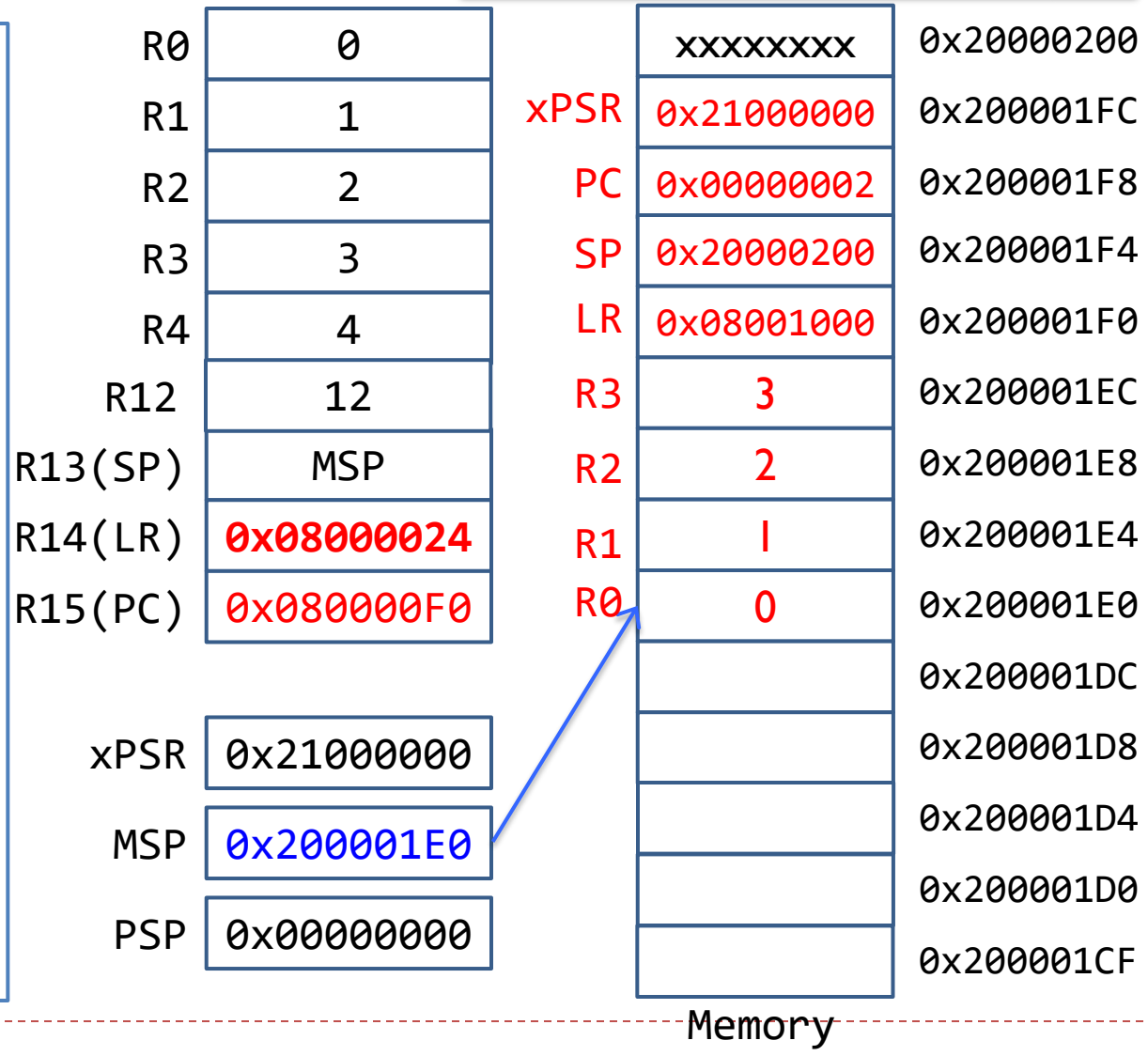


Interrupt: Stacking & Uns

UNSTACKING won't occur!

```
__main PROC
                addr = 0x08000044
   …
   MOV r3,#0
   …
   ENDP
                addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r4, #1  0x0800001C
   BL  sine    0x08000020
   BX  lr      0x08000024
   ENDP
```

BL sine
Updates LR



Fixing the Bug

- LR has two different usages for function calls and for interrupts.
- After calling function sine(), LR points to the return address 0x08000024; the previous value of 0xFFFFFFF9 is overwritten and lost.
- Fix the bug:
  - Method 1: PUSH{lr}/POP{lr} in the function to save and restore the original LR value of 0xFFFFFFF9
  - Method 2: PUSH{lr}/POP{PC}
    - POP {PC} is equivalent to POP {lr} followed by BX lr

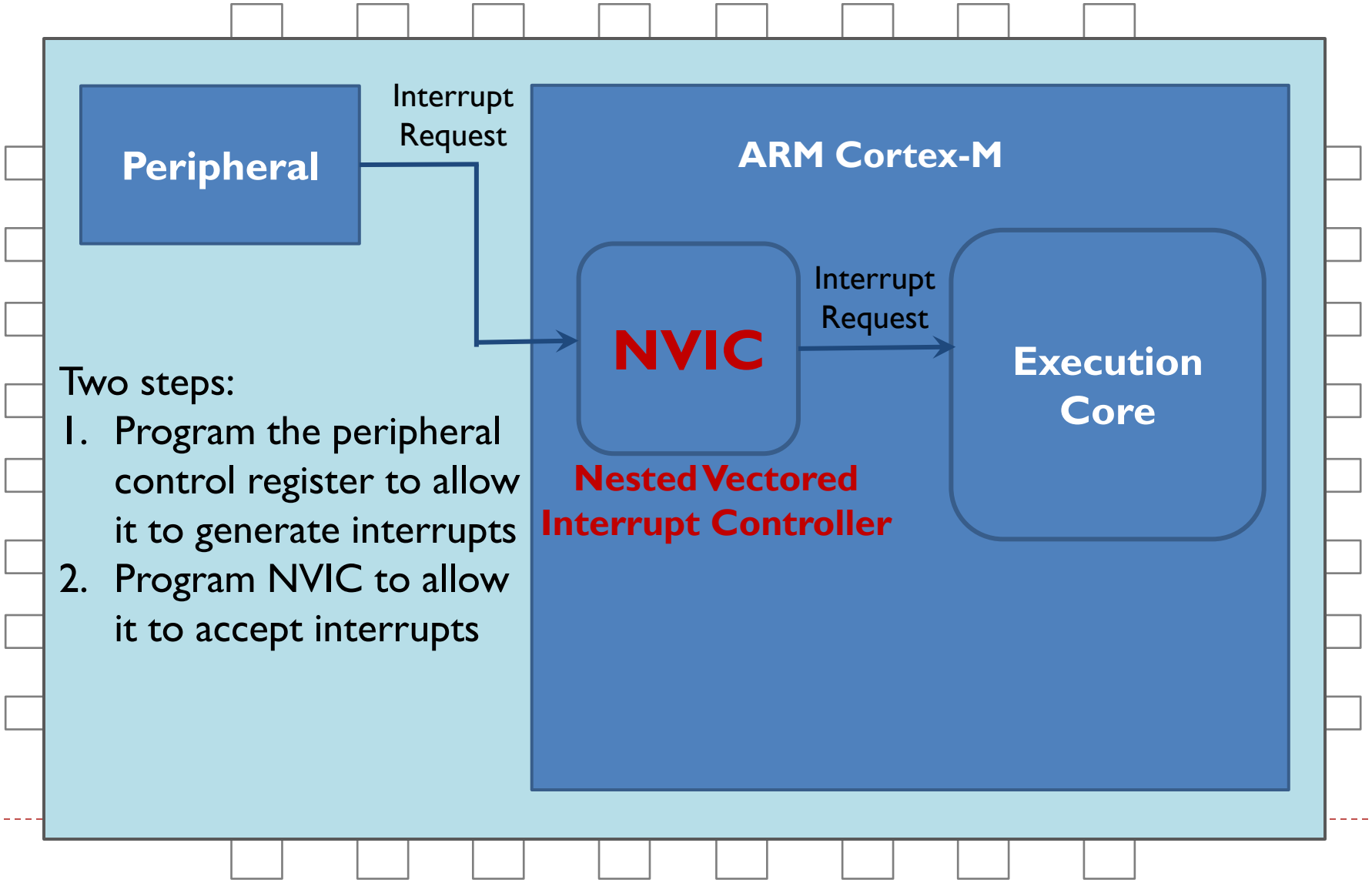
```
__main PROC
   …
   MOV r3,#0
   …
   ENDP

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

Method 1

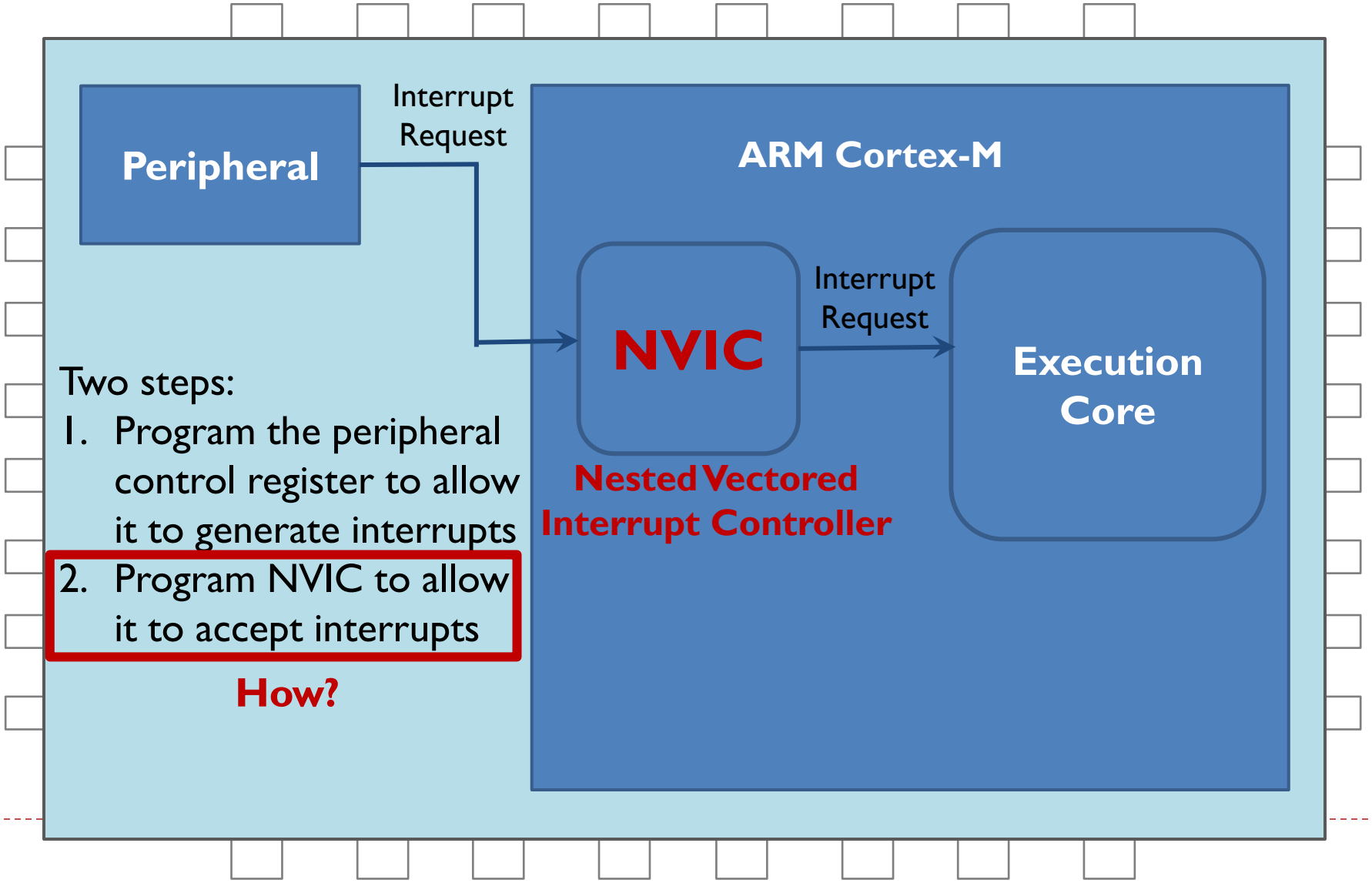
```
__main PROC
   …
   MOV r3,#0
   …
   ENDP

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

Method 2

Enable an Interrupt

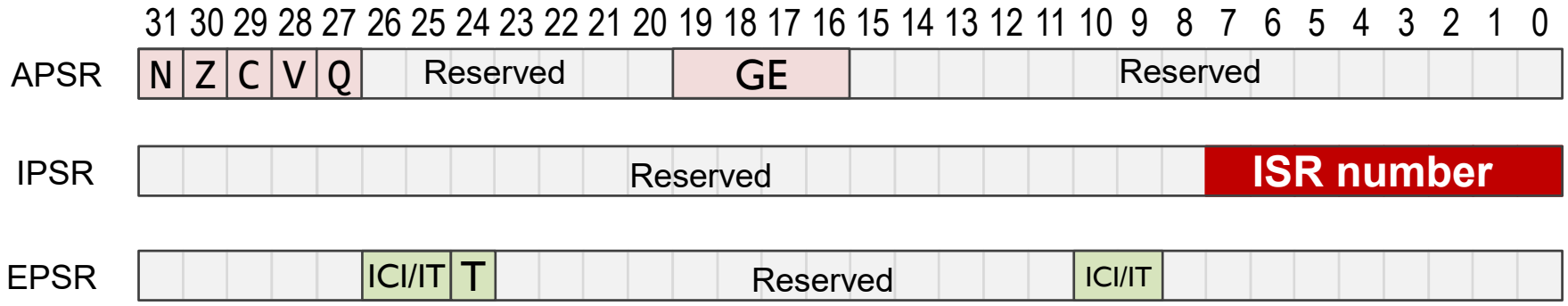


Enable an Interrupt



Interrupt Number in PSR

- Application PSR (**APSR**), Interrupt PSR (**IPSR**), Execution PSR (**EPSR**)

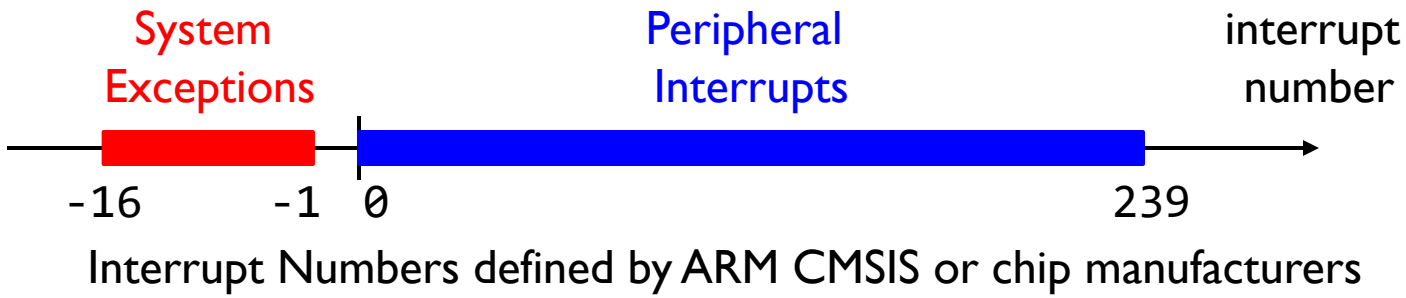


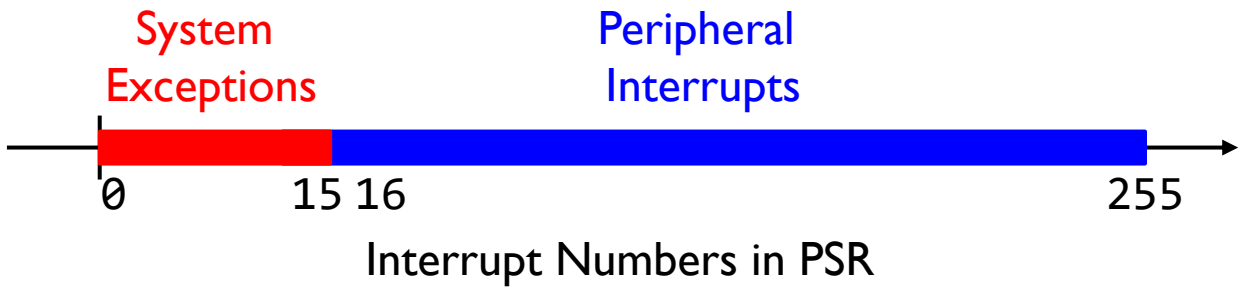
Combine them together into one register (PSR = APSR | IPSR | EPSR ("|" stands for bitwise OR))

8-bit interrupt number in **PSR**
Range: **`0 - 255`**

Interrupt Number in CMSIS *vs* in PSR

- Cortex-M supports up to 256 interrupts.
  - Interrupt numbers -16 to -1 denote system exceptions, as defined by ARM CMSIS (Cortex Microcontroller Software Interface Standard);
  - Interrupt numbers 0-239 denote peripheral interrupts, as defined by chip manufacturers
- Interrupt Number in PSR = 16 + Interrupt Number for CMSIS



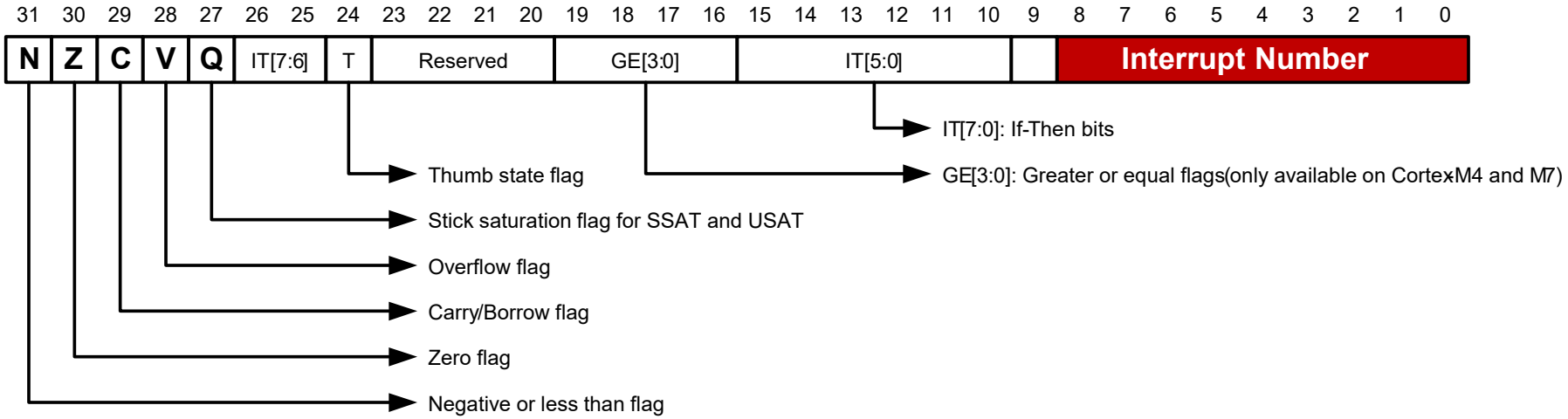


Interrupt Number

Interrupt number for CMSIS functions

```
NVIC_DisableIRQ (IRQn);             // Disable interrupt
NVIC_EnableIRQ (IRQn);              // Enable interrupt
NVIC_ClearingPending (IRQn);        // clear pending status
NVIC_SetPriority (IRQn, priority); // set priority level
```

Interrupt number is stored in the last Byte of Program Status Register (PSR)



Interrupt Numbers in ARM CMSIS library

```
/******  Cortex-M4 System Exceptions ********************************************************/
  NonMaskableInt_IRQn     = -14,    /* 2 Cortex-M4 Non Maskable Interrupt                */
  HardFault_IRQn          = -13,    /* 3 Cortex-M4 Hard Fault Interrupt                  */
  MemoryManagement_IRQn   = -12,    /* 4 Cortex-M4 Memory Management Interrupt           */
  BusFault_IRQn           = -11,    /* 5 Cortex-M4 Bus Fault Interrupt                   */
  UsageFault_IRQn         = -10,    /* 6 Cortex-M4 Usage Fault Interrupt                 */
  SVCall_IRQn             = -5,     /* 11 Cortex-M4 SV Call Interrupt                    */
  DebugMonitor_IRQn       = -4,     /* 12 Cortex-M4 Debug Monitor Interrupt              */
  PendSV_IRQn             = -2,     /* 14 Cortex-M4 Pend SV Interrupt                    */
  SysTick_IRQn            = -1,     /* 15 Cortex-M4 System Tick Interrupt                */
/******  Peripheral Interrupt Numbers *******************************************************/
  WWDG_IRQn               = 0,      /* Window WatchDog Interrupt                         */
  PVD_PVM_IRQn            = 1,      /* PVD/PVM1,2,3,4 through EXTI Line detection Interrupts */
  TAMP_STAMP_IRQn         = 2,      /* Tamper and TimeStamp interrupts through the EXTI line */
  RTC_WKUP_IRQn           = 3,      /* RTC Wakeup interrupt through the EXTI line        */
  FLASH_IRQn              = 4,      /* FLASH global Interrupt                            */
  RCC_IRQn                = 5,      /* RCC global Interrupt                              */
  EXTI0_IRQn              = 6,      /* EXTI Line0 Interrupt                              */
  ...
```

System Exceptions Defined by ARM

Peripheral Interrupts Defined by chip vendor

stm32l476xx.h

NVIC Registers

- ISER (Interrupt Set-Enable Register)
  - Used to enable interrupts or to determine which interrupts are currently enabled
- ICER (Interrupt Clear-Enable Register)
  - Used to disable interrupts or to determine which interrupts are currently disabled
- ISPR (Interrupt Set-Pending Register)
  - Used to force interrupts into the pending state, or to determine which interrupts are currently pending
- ICPR (Interrupt Clear-Pending Register)
  - Used to clear pending interrupts, or to determine which interrupts are currently pending
- Interrupt Priority Registers
  - Used to set interrupt priority (importance)

Enable/Disable Interrupts

- Enable a system interrupt
  - Some are always enabled (cannot be disabled)
  - No centralized registers for enabling/disabling
  - Each are control by its corresponding components, such as SysTick module

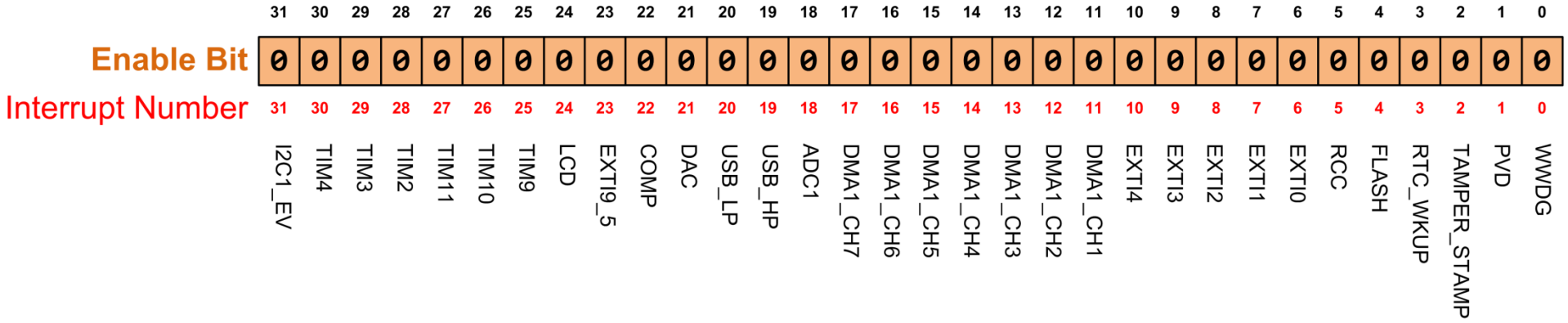
- Enable a peripheral interrupt
  - Centralized register arrays for enabling/disabling
  - **ISER** registers for enabling
  - **ICER** registers for disabling
  - They are separate write-only registers that control the same enable flip-flops inside the NVIC:
  - Writing 1 to a bit in ISER[x] → sets the enable bit (enables interrupt)
  - Writing 1 to a bit in ICER[x] → clears the enable bit (disables interrupt)
  - Writing 0 to a bit in ISER[x] or ICER[x] has no effect, so we never do it.
  - Separating enable bits and disable bits in two separate sets of registers, ICER and ISER, provides great convenience and flexibility for programmers.

Enable/Disable Peripheral Interrupts

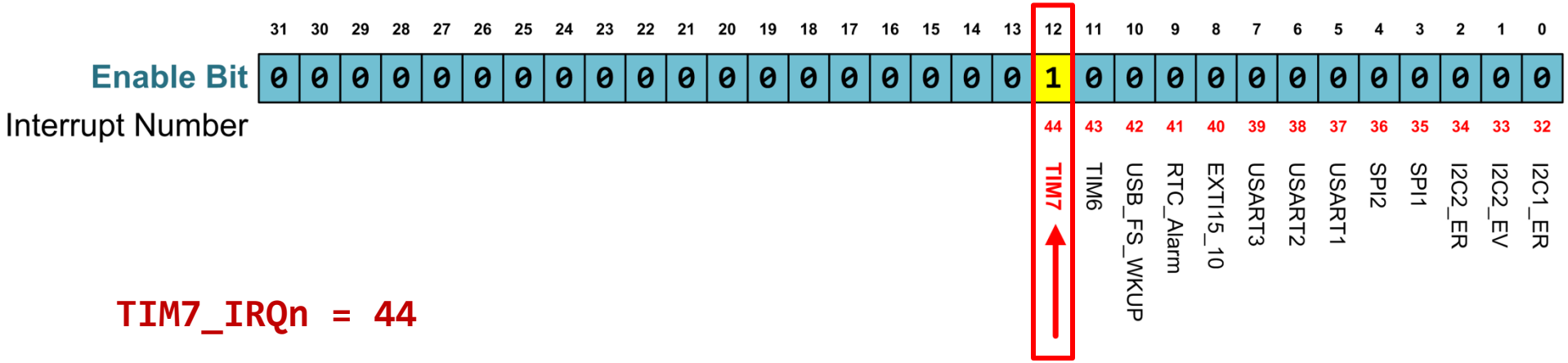
- For all peripheral interrupts: `IRQn ≥ 0`
- Method 1 (Interrupt number for CMSIS):
- These functions are defined in the ARM Cortex core header file
  - `NVIC_DisableIRQ (IRQn);`
  - `NVIC_EnableIRQ (IRQn);`
- Method 2 (Interrupt number for PSR):
- We enable (disable) a peripheral interrupt by setting the corresponding bit of the ISER (ICER) register. To enable a given IRQn, we divide it by 32 to find out in which ISER register the target enable bit is located, since each ISER register has 32 bits and can enable 32 interrupts. The bit offset within the target ISER register is determined by the result of IRQn mod 32.
  - Enable:
    - `NVIC->ISER[IRQn / 32] |= 1 << (IRQn % 32);`
    - `Better solution (IRQn / 32 = IRQn >> 5, IRQn % 32 = IRQn & 0x1F):`
    - `NVIC->ISER[IRQn >> 5] |= 1 << (IRQn & 0x1F);`
  - Disable:
    - `NVIC->ICER[IRQn >> 5] |= 1 << (IRQn & 0x1F);`

Enabling Peripheral Interrupts

Interrupt Set Enable Register 0 (ISER0)



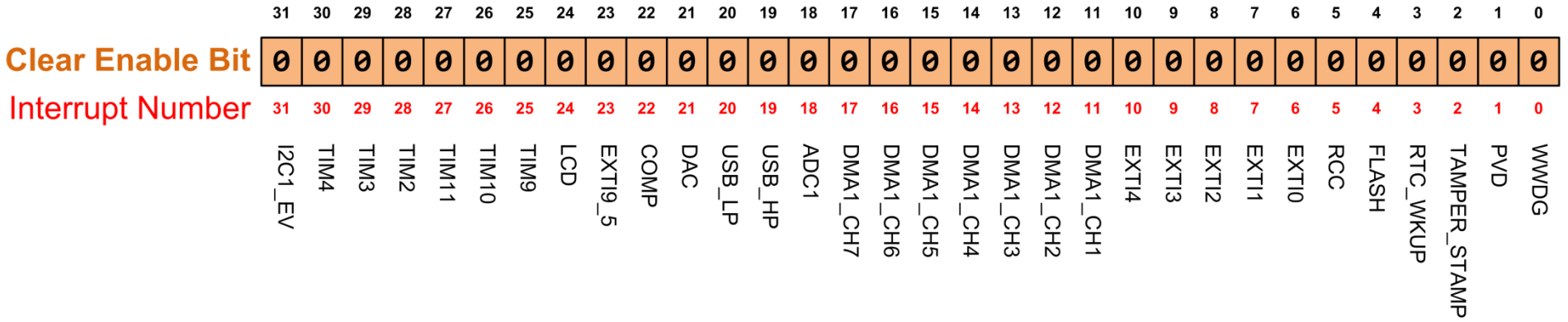
Interrupt Set Enable Register 1 (ISER1) *Address of ISER1 = Address of ISER0 + 4*



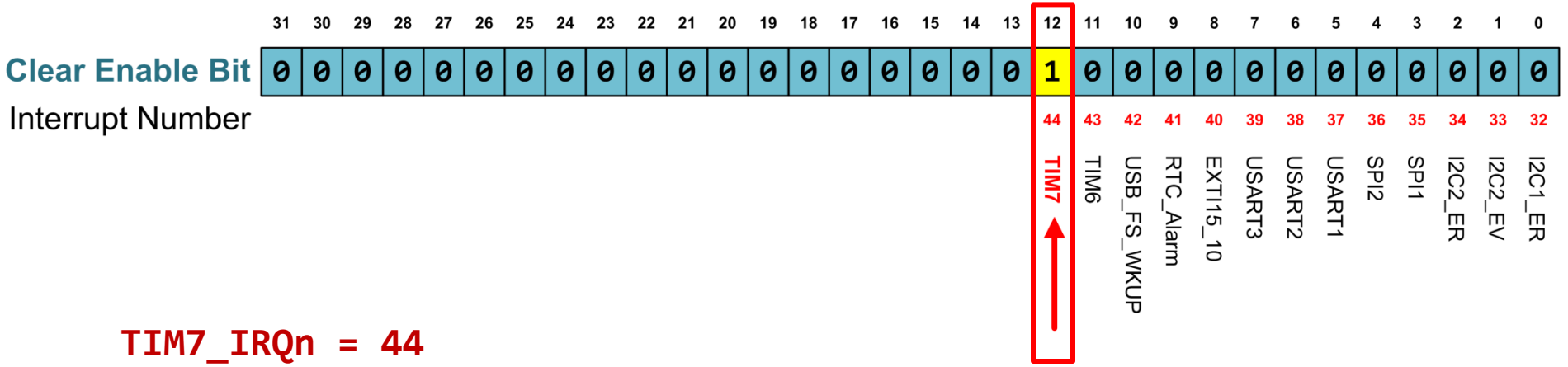

```
TIM7_IRQn = 44

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

Disabling Peripheral Interrupts

Interrupt Clear Enable Register 0 (ICER0)



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




```
TIM7_IRQn = 44

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

Explanations

- To enable interrupt "Timer 7" with interrupt number 44:
  - ISER0 controls interrupts 0 to 31; ISER1 controls interrupts 32 to 63.
  - To enable interrupt 44, we set bit 12 (44-32) of ISER1 to 1 by executing NVIC->ISER[1] |= 1 << 12.
- To disable interrupt "Timer 7" with interrupt number 44:
  - To disable interrupt 44, we set bit 12 (44-32) of ICER1 to 1 by executing NVIC->ICER[1] |= 1 << 12.

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.

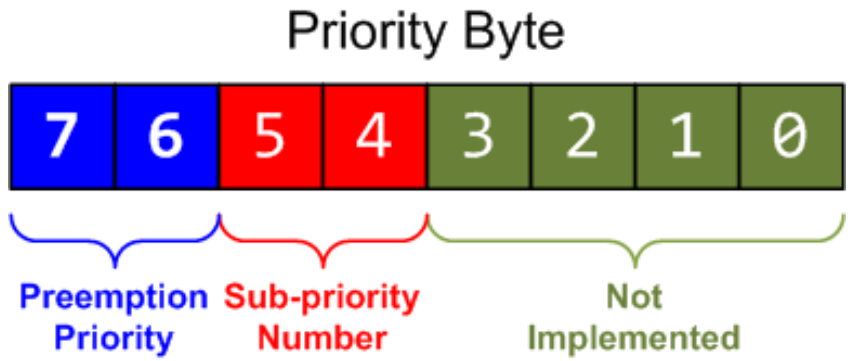
- Fixed priority for Reset, HardFault, and NMI.

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

- Adjustable for all the other interrupts

Interrupt Priority

- Interrupt priority is configured by Interrupt Priority Register (IP)
- 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.

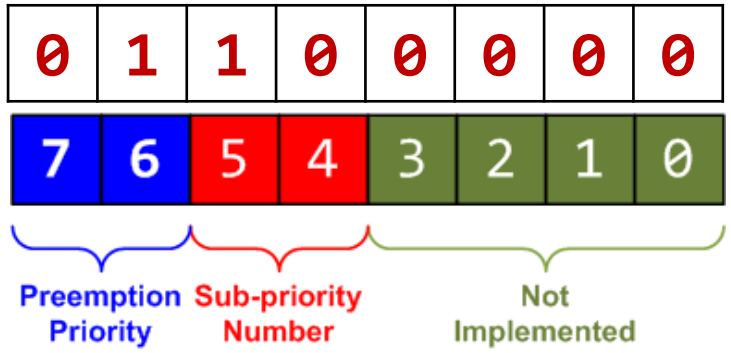


default setting

Interrupt Priority Levels

Configure interrupt priority for IRQ number 7 to be level 6 (01100000 in binary, or 96 in decimal):

```
NVIC_SetPriority(7, 6);
```



IP = 0x60 = 96

core_cm4.h or core_cm3.h

```
typedef struct {
  ...
  // Interrupt Priority Register
  volatile uint8_t IP[240];
  ...
} NVIC_Type;
```

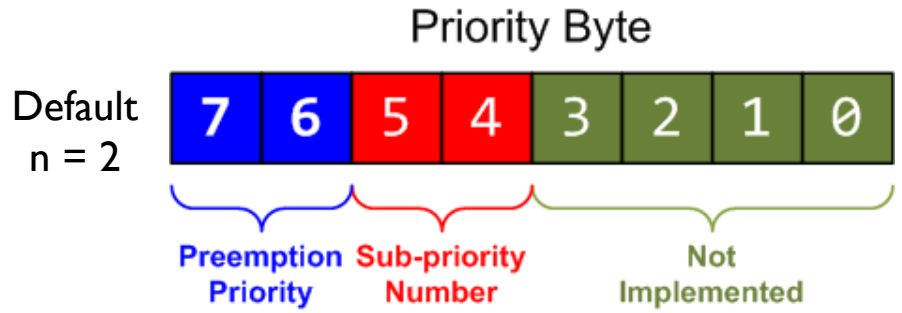
It is equivalent to (Priority value is shifted left by 4 bits, and the result is stored in the corresponding interrupt priority byte):

```
NVIC->IP[7] = (6 << 4) & 0xff;
```

Preemption and Sub-priority Configuration

- `NVIC_SetPriorityGrouping(n)`
  - `Perform unlock, and update AIRCR register`

n	# of bits in preemption priority	# of bits in sub-priority
0	0	4
1	1	3
2 (default)	2	2
3	3	1
4	4	0



Masking Priority

- We have discussed enabling/disabling individual interrupts by setting NVIC registers. ARM also provides mechanisms to enable/disable a group of interrupts.
- 3 Interrupt Mask Registers:

Register Name	Description
PRIMASK	A 1-bit register. When this is set, it allows Reset, NMI and Hard Fault; all other interrupts and exceptions are disabled (masked); default is 0 (no masking)
FAULTMASK	A 1-bit register. When this is set, it allows only Reset and NMI; all other interrupts and exceptions (including Hard Fault) are disabled; default is 0 (no masking)
BASEPRI	A register of up to 9 bits. It defines the masking priority level. When this is set, it disables all interrupts of the same or lower importance (same or larger priority values)

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 larger than 0x60 (MSR moves the value in R0 into the BASEPRI special register.)

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

References

- Lecture 9: Interrupts
  - https://www.youtube.com/watch?v=uFBNf7F3I60&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=9