L6 Interrupts

Acknowledgement: some slides taken from Yifeng Zhu's courseware

Polling *vs* Interrupt

Suppose you are waiting for an important phone call.

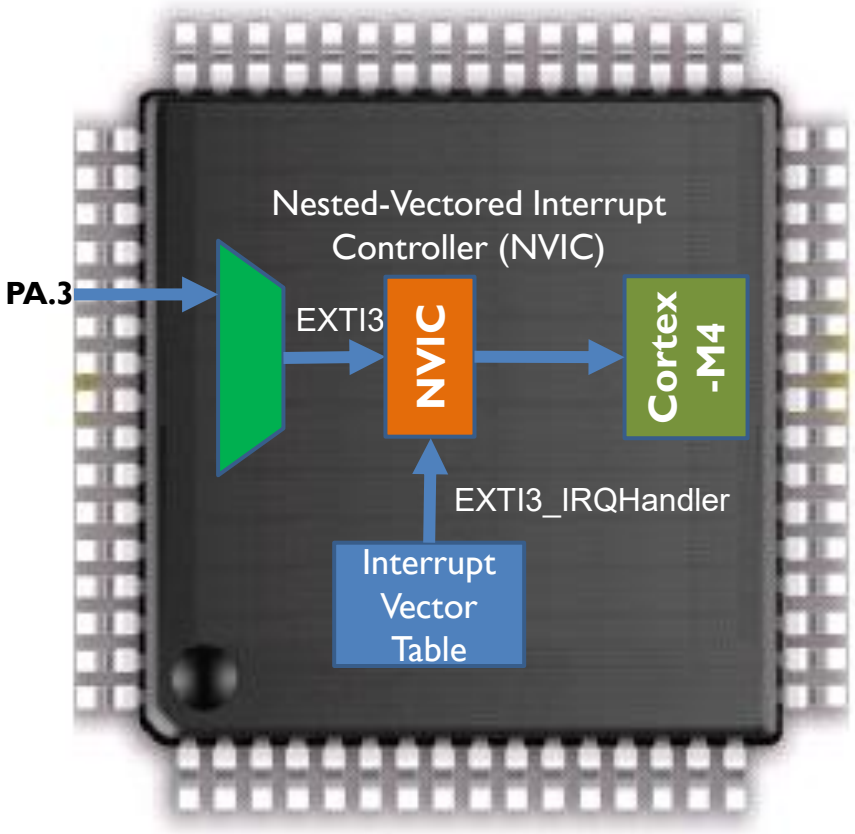
Polling:
You pick up the phone every three seconds to check whether you are getting a call.

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

```
// Polling method
while (1) {

read_button_input;
    if (pushed)
        exit;
}

turn_on_LED;
```

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

Interrupt Vector Table



Address of ISR 1

ISR Vector Table

Interrupt Number (8 bits)	Memory Address of ISR (32 bits)
1	ISR for interrupt 1
2	ISR for interrupt 2
3	ISR for interrupt 3
4	ISR for interrupt 4
5	ISR for interrupt 5
…	…

When interrupt x is triggered, jump to the ISR for interrupt x. ($1 \le x \le 255$)

Interrupts and Exceptions

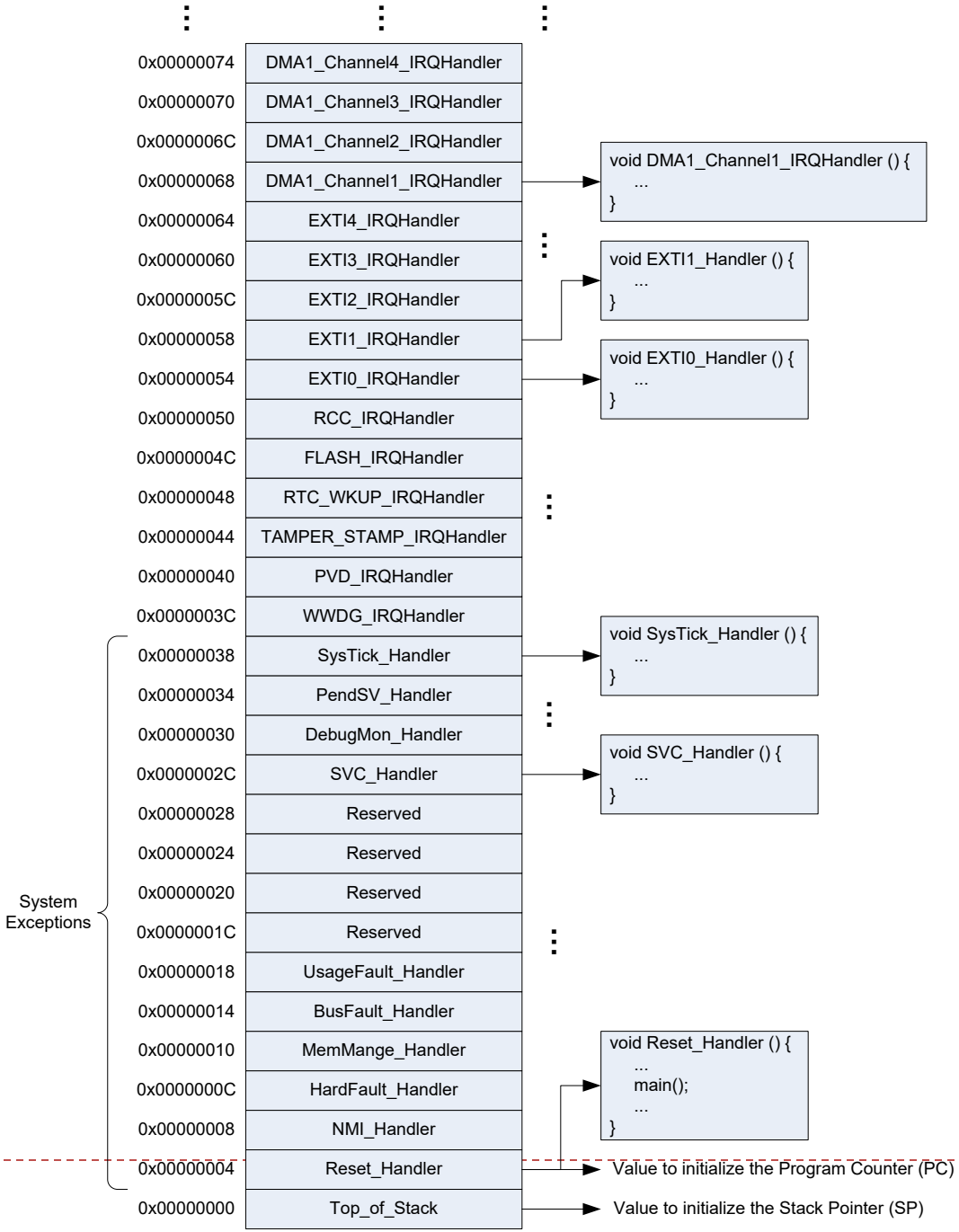
- Asynchronous (peripheral interrupts)
  - Triggered by external I/O devices
    - Mouse clicks, keyboard presses, incoming data packets on the network card…
  - Each interrupt triggers an Interrupt Handler, or Interrupt Service Routine (ISR)
- Synchronous (system exceptions)
  - Triggered by instruction execution
  - *Processor-detected* exceptions:
    - *Faults* — correctable; offending instruction is *retried*
    - *Traps* — often for debugging; instruction is *not* retried
    - *Aborts* — major error (hardware failure)
  - *Programmed* exceptions:
    - System calls to the OS kernel
  - Each exception triggers an Exception Handler
- In practice, the terms exception and interrupt, Exception Handler , Interrupt Handler and Interrupt Service Routine (ISR) are often used interchangeably.

ISR Vector Table

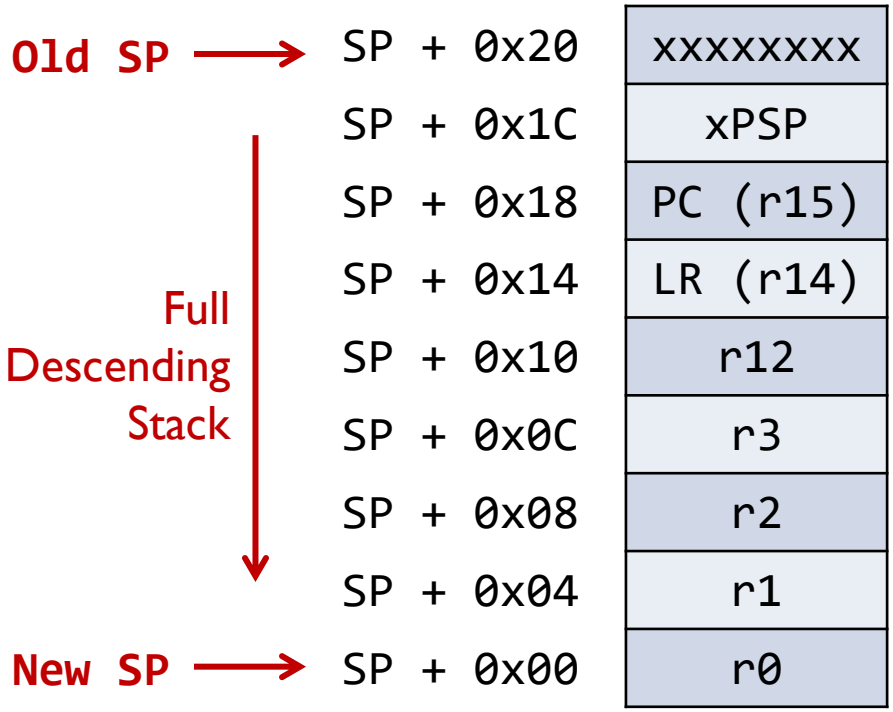
- Start address for each ISR for each exception/interrupt type is defined in the ISR Vector Table.
- Upon each exception/interrupt, processor jumps to the corresponding ISR by loading PC with its instruction address stored in the table.

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



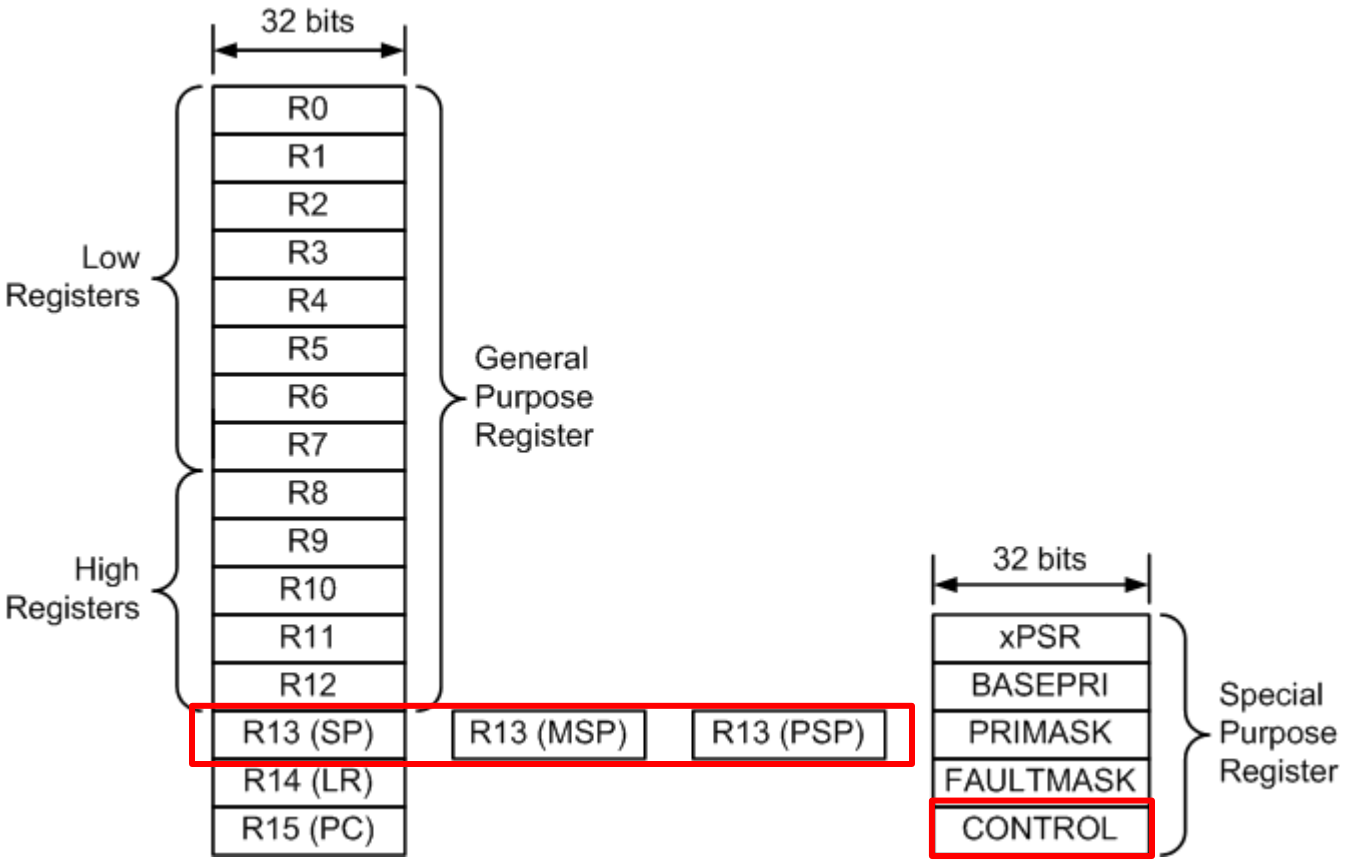
Stacking & Unstacking



- **Stacking:** The processor automatically pushes these 8 registers into the stack before an ISR starts
- **Unstacking:** The processor automatically pops these eight register out of the stack and restores the 8 registers when an ISR exits.

- Two types of stack pointers
  - Main SP (MSP): only one main stack system-wide
  - Process SP (PSP): one process stack per thread

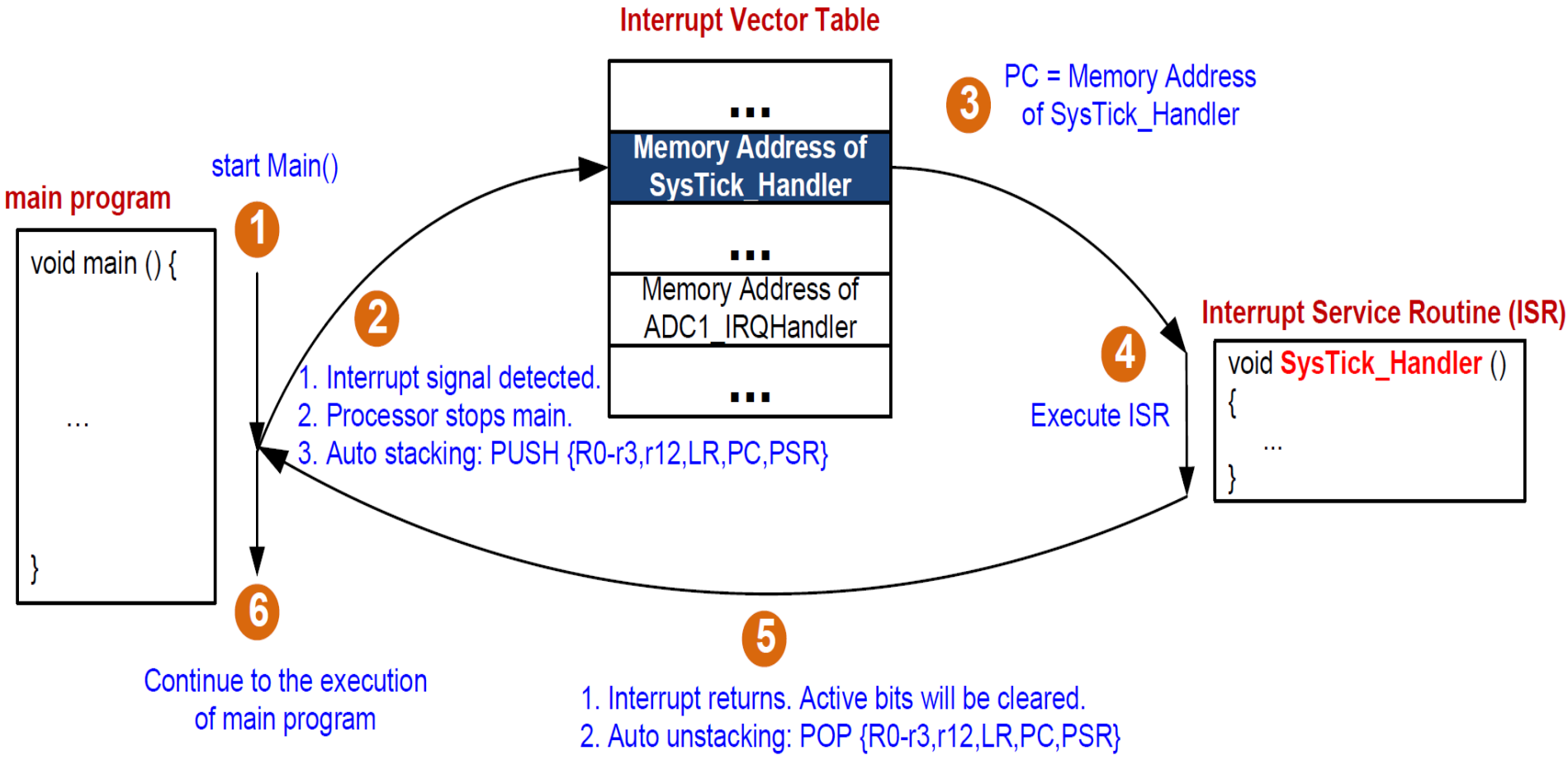
Registers



MSP: Main Stack Pointer

PSP: Process Stack Pointer

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

Processor Modes and Privilege Levels

- Processor Modes:
  - Thread Mode: used to execute application or OS software
  - Handler Mode: used to ISRs triggered by exceptions/interrupts
- Privilege Levels:
  - User Level: cannot use MSR/MRS instructions; cannot access the system timer, NVIC, or system control block; restricted access to memory or peripherals
    - Application software executes at User Level
  - Privileged Level: can use all instructions and has access to all resources.
    - ISR and OS execute at Privileged Level
- Controlled by a special 2-bit register CONTROL

CONTROL Register

- CONTROL[0] controls the access level (Privileged or User)
  - CONTROL[0]=0 → Privileged Level; CONTROL[0]=1 → User Level
  - Writable only when executing at Privileged Level.
- CONTROL[1] controls the selection of MSP or PSP for stacking/unstacking
  - CONTROL[1]=0 → use MSP for stacking/unstacking; CONTROL[0]=1 → use PSP for stacking/unstacking
  - CONTROL[1] is always 0 in Handler Mode; can be either 0 or 1 in Thread Mode.
  - Writable only in Thread Mode and at Privileged Level.

Operation Modes and Privilege Levels

	Privileged Level	*User Level*
Handler Mode (executing Interrupt Handler)	*CONTROL[0]=0* *CONTROL[1]=0*	*N/A*
Thread Mode (executing app software or OS)	*CONTROL[0]=0* *CONTROL[1]=0 or 1*	*CONTROL[0]=1* *CONTROL[1]=0 or 1*

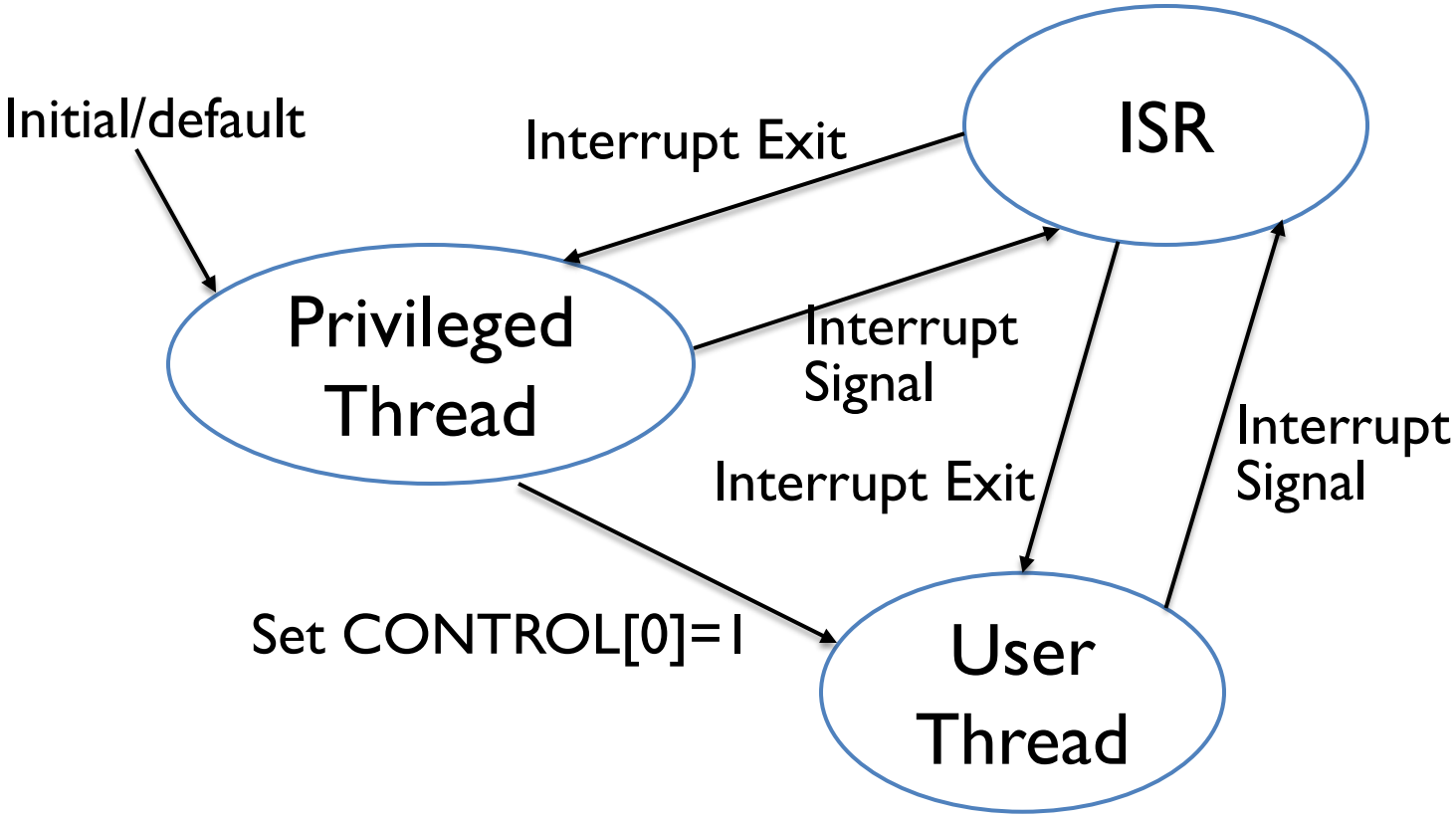
MSP/PSP Selection for Stacking/Unstacking

	Use MSP	*Use PSP*
Handler or Thread Mode	*CONTROL[1]=0*	*CONTROL[1]=1*

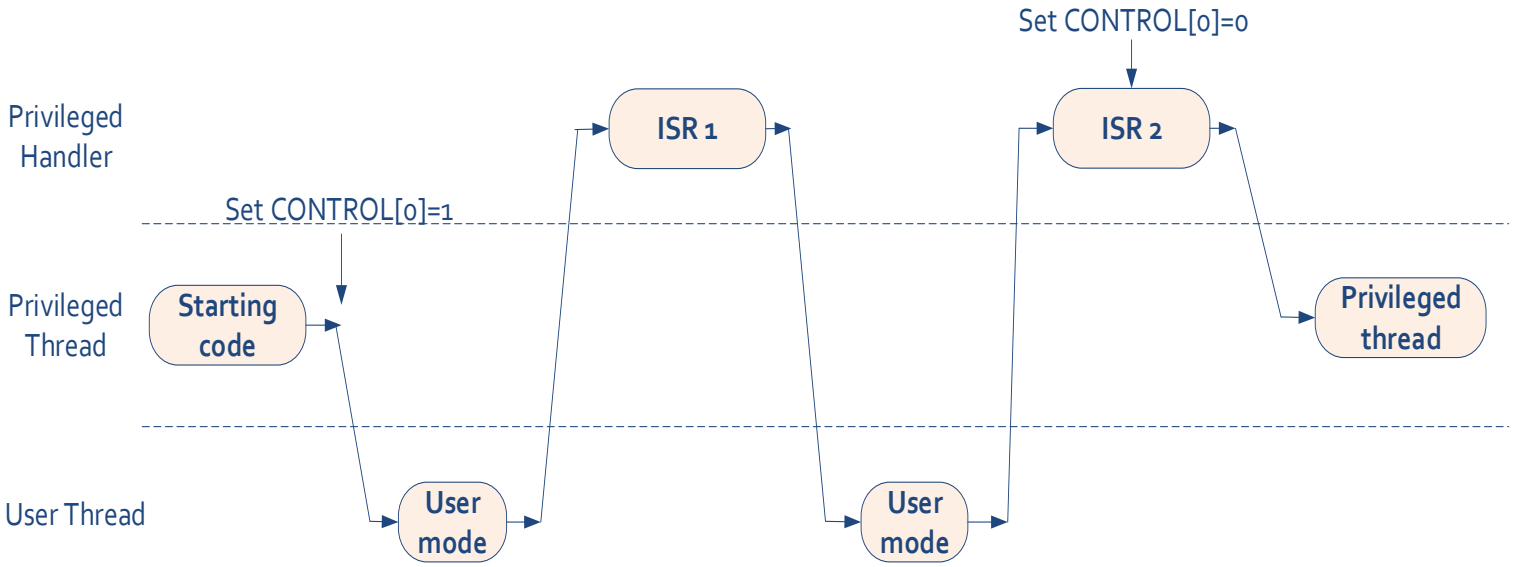
MSP/PSP Selection for Stacking/Unstacking

- Determined by operating mode, and bit 1 of the CONTROL register
  - Handler mode ⟶ `SP = MSP, CONTROL[1] = 0 always`
  - Thread mode ⟶ `SP = MSP, if CONTROL[1] = 0`
    `SP = PSP, if CONTROL[1] = 1`
- In Handler mode, MSP is always used.
- In Thread mode, programmer can choose MSP or PSP by setting CONTROL[1], but ARM recommends that application software (User Level) uses the PSP; the OS (Privileged Level) uses the MSP.

Switching of Operation Modes



Switching of Operation Modes: Example



- Upon system reset, the CPU starts in Thread mode with Privileged Level ("Privileged Thread").
- Switching of operation modes can be done by setting the CONTROL register or be triggered by exceptions.
  - An application in Thread Mode, Privileged Level ("Privileged Thread") can switch into Thread Model, User level ("User Thread") by setting CONTROL[0]=1.
  - When an interrupt occurs, the ISR executes in Privileged access level ("Privileged Handler"); upon exiting the ISR, the application thread continues execution in User Level ("User Thread") by default, or Privileged Level ("Privileged Thread") by setting CONTROL[0]=0 in ISR.
  - CONTROL[0] can only be set in Privileged Level, hence an application thread CANNOT directly switch from "User Thread" to "Privileged Thread" by setting CONTROL[0]=0

Accessing CONTROL Register

- To access the CONTROL register, the MRS and MSR instructions are used:
  - MRS <reg>, <special_reg>; Read special register, e.g.
    - MRS R0, CONTROL ;Read CONTROL register into R0
  - MSR <special_reg>, <reg>; write to special register, e.g.,
    - MOV R0, #0x3
    - MSR CONTROL, R0 ;Write CONTROL[0]=1; CONTROL[1]=1.

      (Cannot directly write #0x3 to special register with MSR)

- Think of MRS as "Move Register SpecialRegister"; MSR as "Move SpecialRegister Register"

Which stack to use when exiting an interrupt?

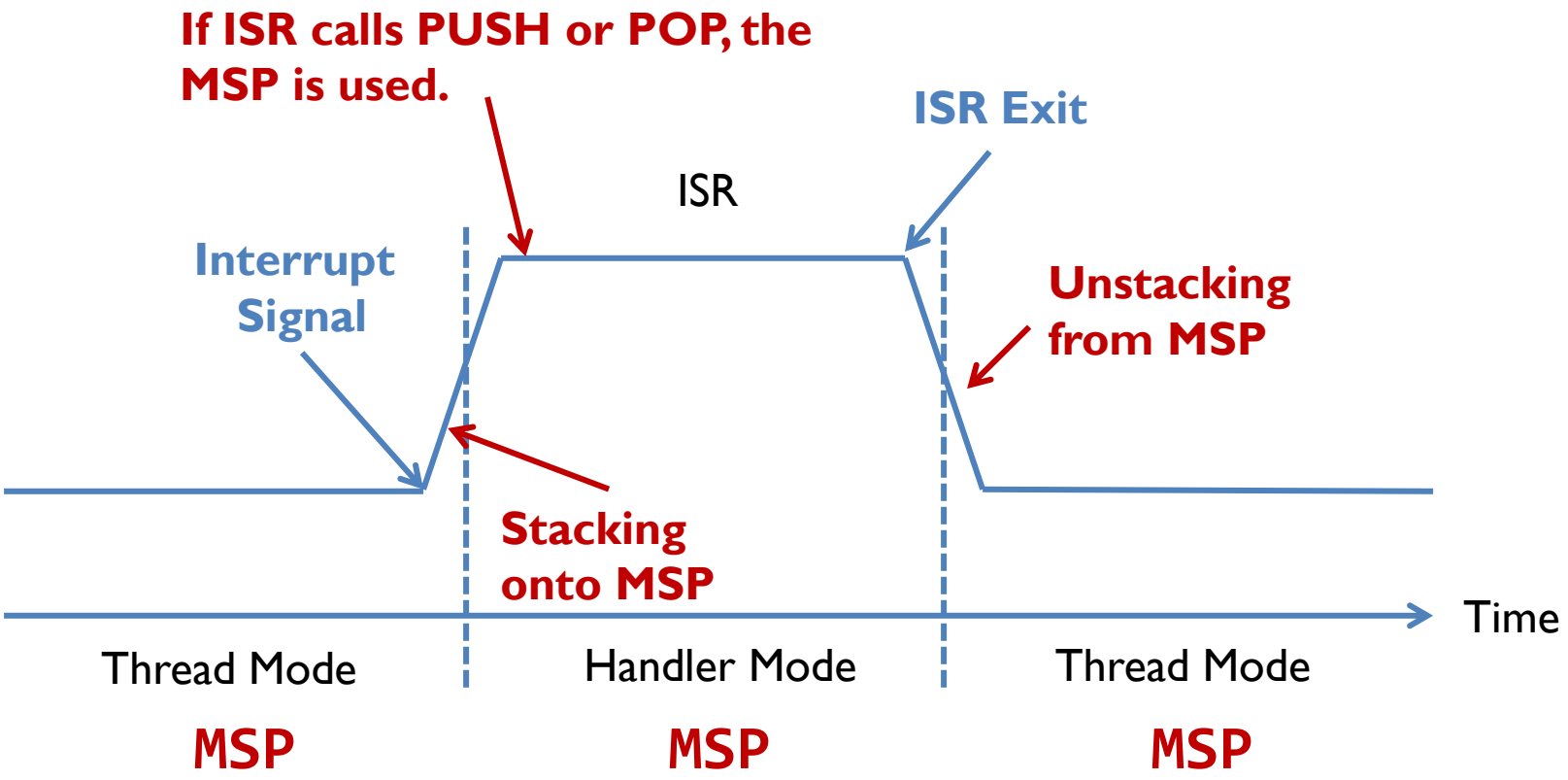
Link Register (LR) has two different usages:

- For function calls: LR = address of the instruction immediately after BL, i.e., instruction address to return to after the function call finishes
- For interrupts: LR indicates whether MSP or PSP is used for unstacking when exiting an interrupt (return address is stored in the PC, which is pushed on the stack)
  - LR is initialized to **`0xFFFFFFFF`** upon system reset (reboot)
  - Upon entering ISR, LR is set to one of 3 EXC_RETURN values
  - All 3 EXC_RETURN values have bits[31:5] set to 1. When this value is loaded into the PC it indicates to the processor that the ISR is complete, and the processor initiates the appropriate return sequence.
  - Bits[3:0] of EXC_RETURN indicate the required return stack and processor mode

EXC_RETURN	Bits[3:2]	Return to	Unstack from
`0xFFFFFFF1`	**`00`**	Handler Mode	MSP
`0xFFFFFFF9`	**`01`**	Thread Mode	MSP
`0xFFFFFFFD`	**`11`**	Thread Mode	PSP

Stacking & Unstacking

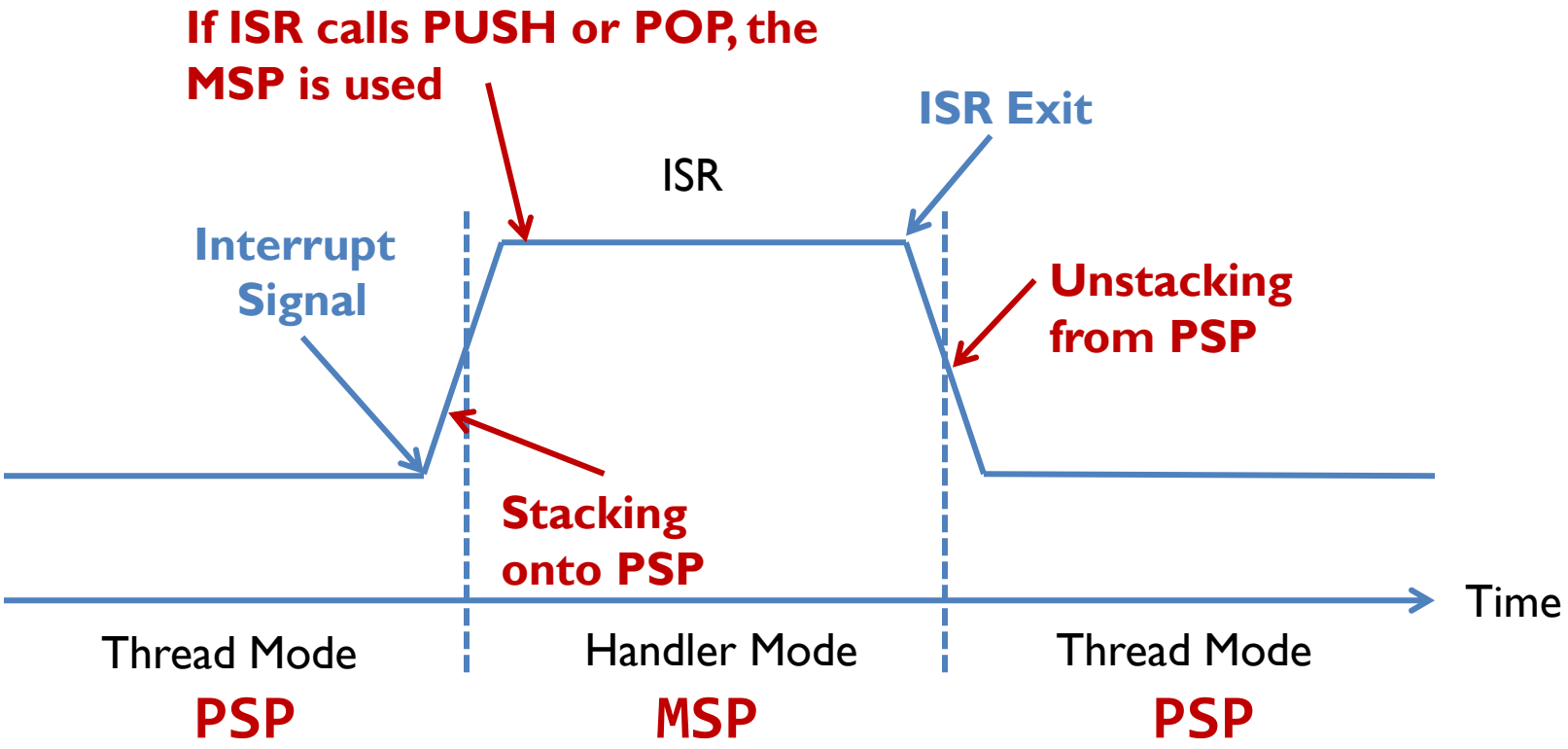
CONTROL[1] = 0 ⟹ User program uses MSP;
LR is set to 0xFFFFFFF9 upon interrupt signal ⟹ unstacking from MSP upon interrupt exit.



EXC_RETURN = 0xFFFFFFF9

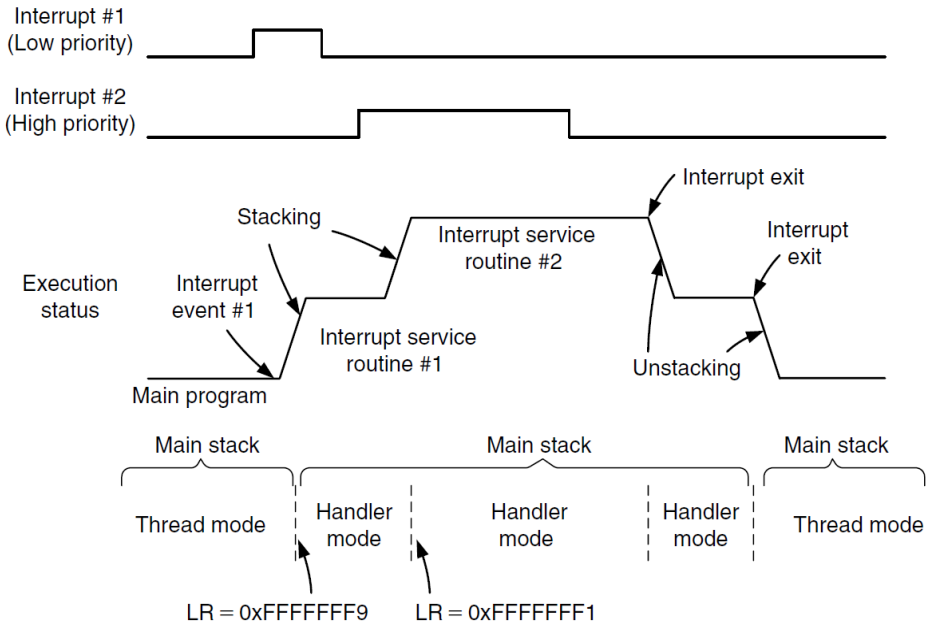
Stacking & Unstacking

CONTROL[1] = 1 ⟹ User program uses PSP;
LR is set to 0xFFFFFFFD upon interrupt signal ⟹ unstacking from PSP upon interrupt exit.

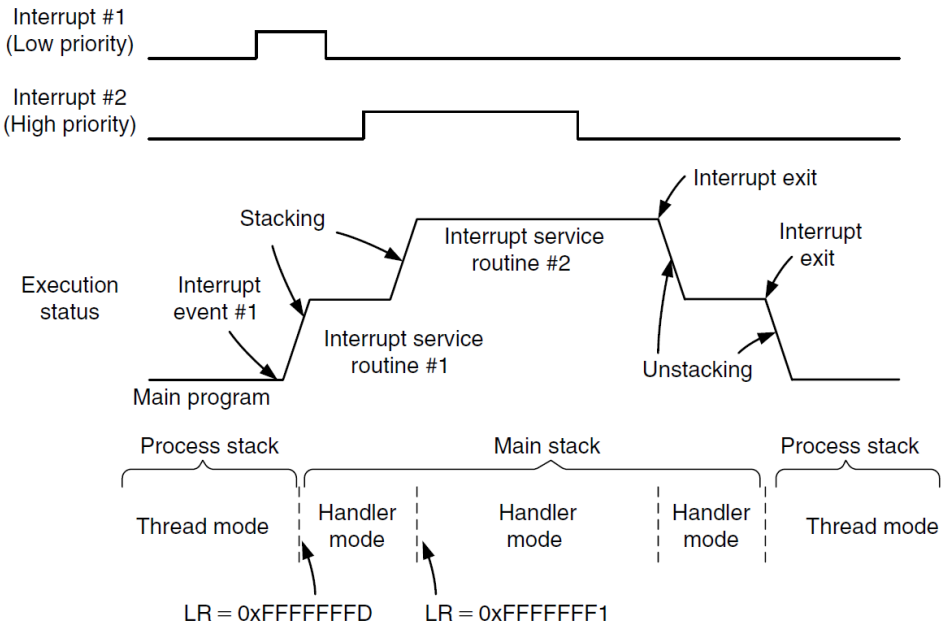


EXC_RETURN = 0xFFFFFFFD

Nested Interrupts



- The user program uses MSP
- LR is set to 0xFFFFFFF9 when ISR #1 is entered (return to Thread mode upon exit)
- LR is set to 0xFFFFFFF1 when nested ISR #2 is entered (return to Handler mode upon exit)

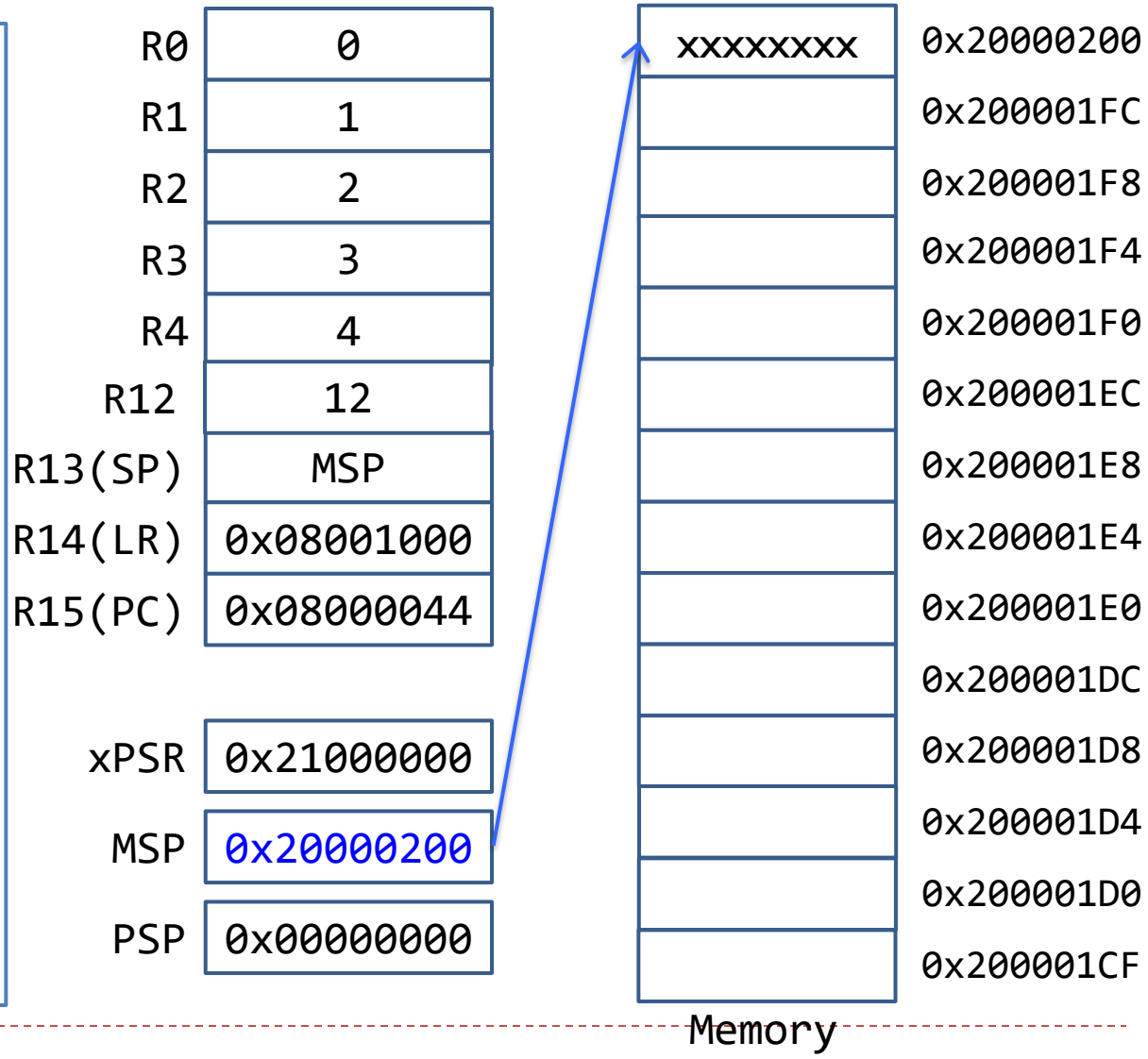


- The user program uses PSP; LR is set to 0xFFFFFFFD when ISR #1 is entered; LR is set to 0xFFFFFFF1 when nested ISR #2 is entered (return to handler mode upon exit).

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

Interrupt: Stacking & Unstacking

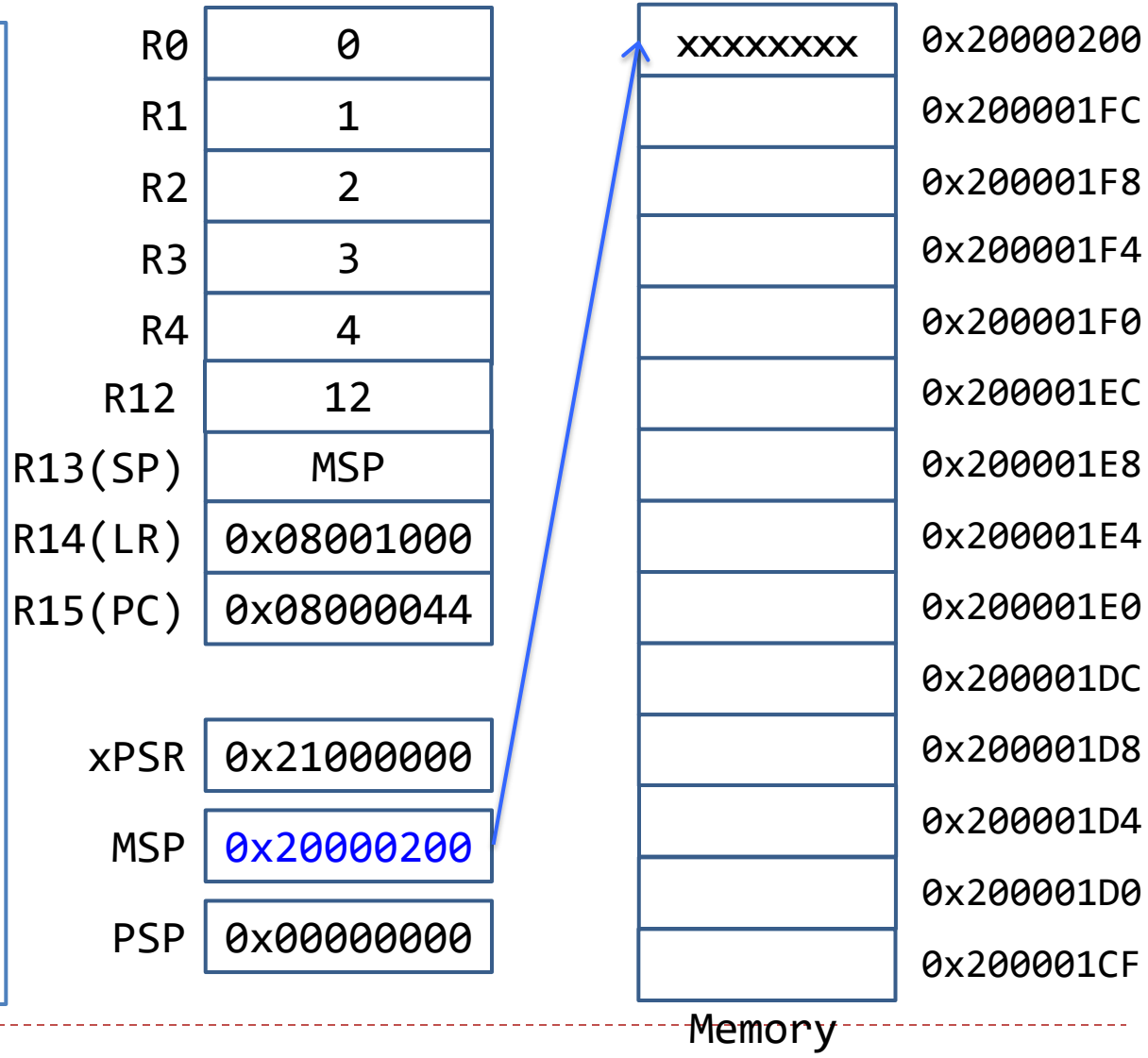
```
__main PROC
   …          addr = 0x08000044
   MOV r3,#0
   …
   ENDP
              addr = 0x0800001C
SysTick_Handler PROC
   EXPORT SysTick_Handler
   ADD r3, #1
   ADD r4, #1
   BX  lr
   ENDP
```



Interrupt:

Suppose SysTick interrupt occurs when PC = 0x08000044

```
__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: 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

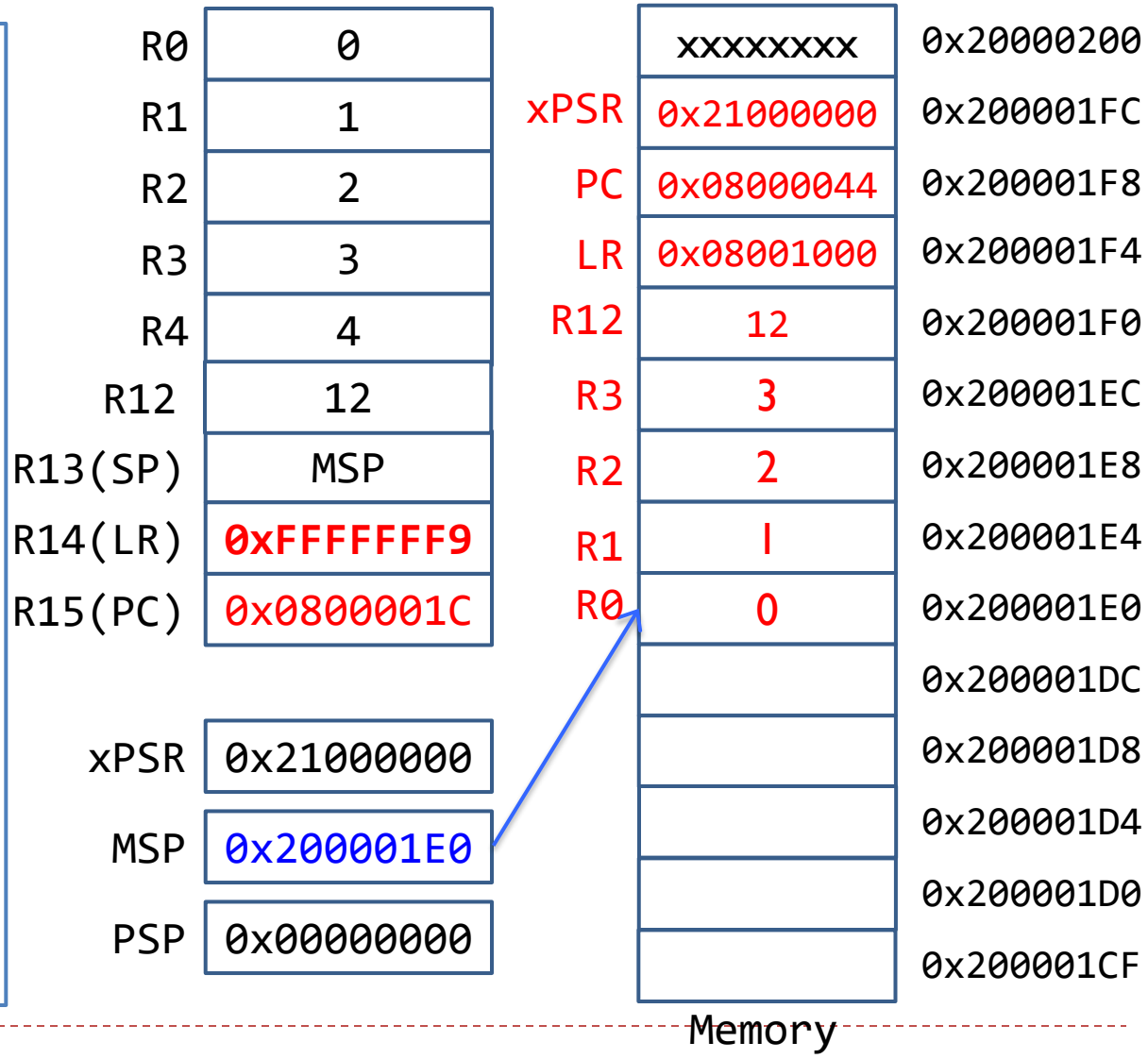
Memory

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

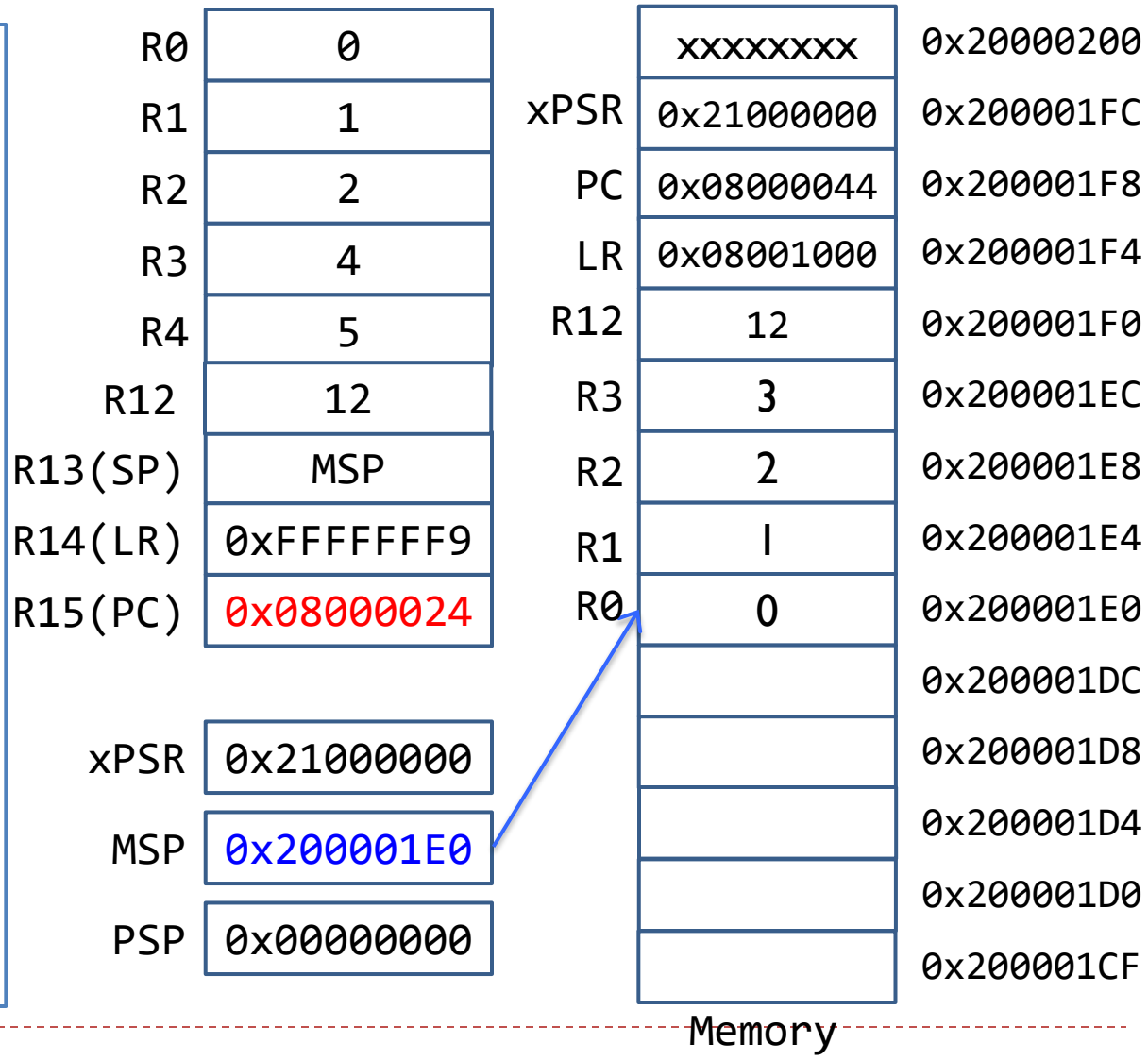
	xxxxxxxx	0x20000200
xPSR	0x21000000	0x200001FC
PC	0x08000044	0x200001F8
LR	0x08001000	0x200001F4
R12	12	0x200001F0
R3	3	0x200001EC
R2	2	0x200001E8
R1	1	0x200001E4
R0	0	0x200001E0
		0x200001DC
		0x200001D8
		0x200001D4
		0x200001D0
		0x200001CF

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
```

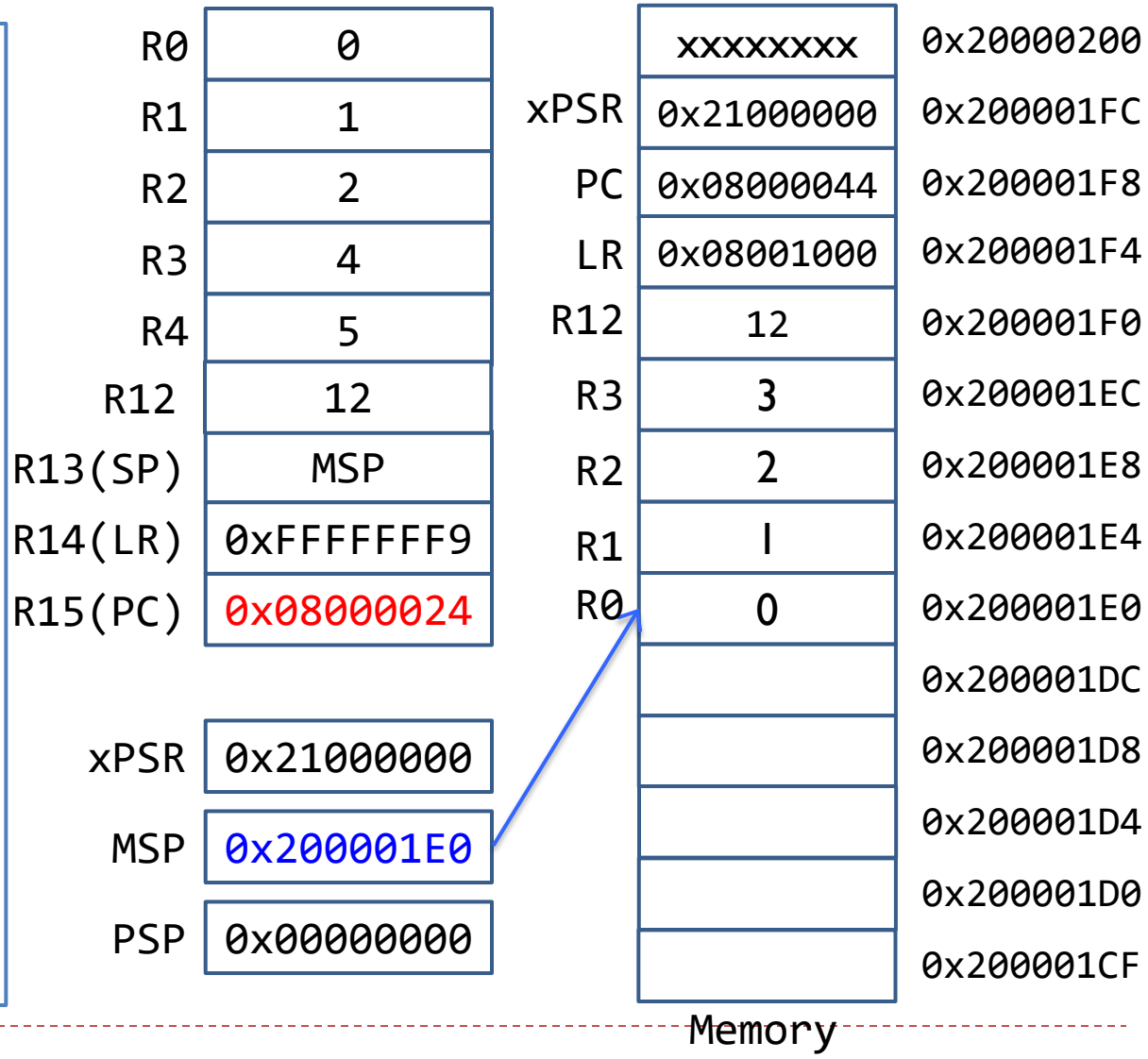
LR = 0xFFFFFFF9 to indicate MSP is used.



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
```

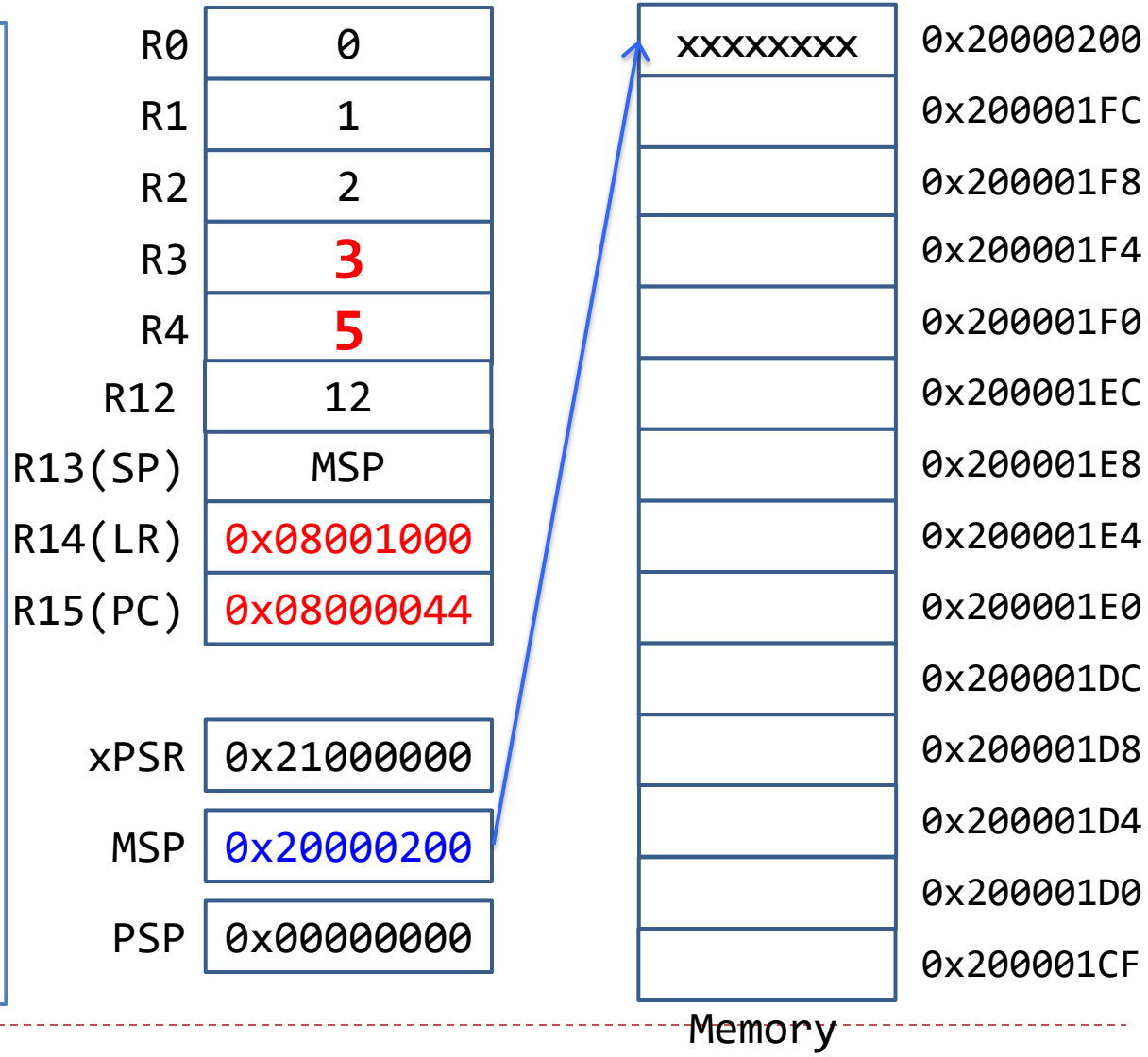
LR = 0xFFFFFFF9 to indicate MSP is used.



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
```

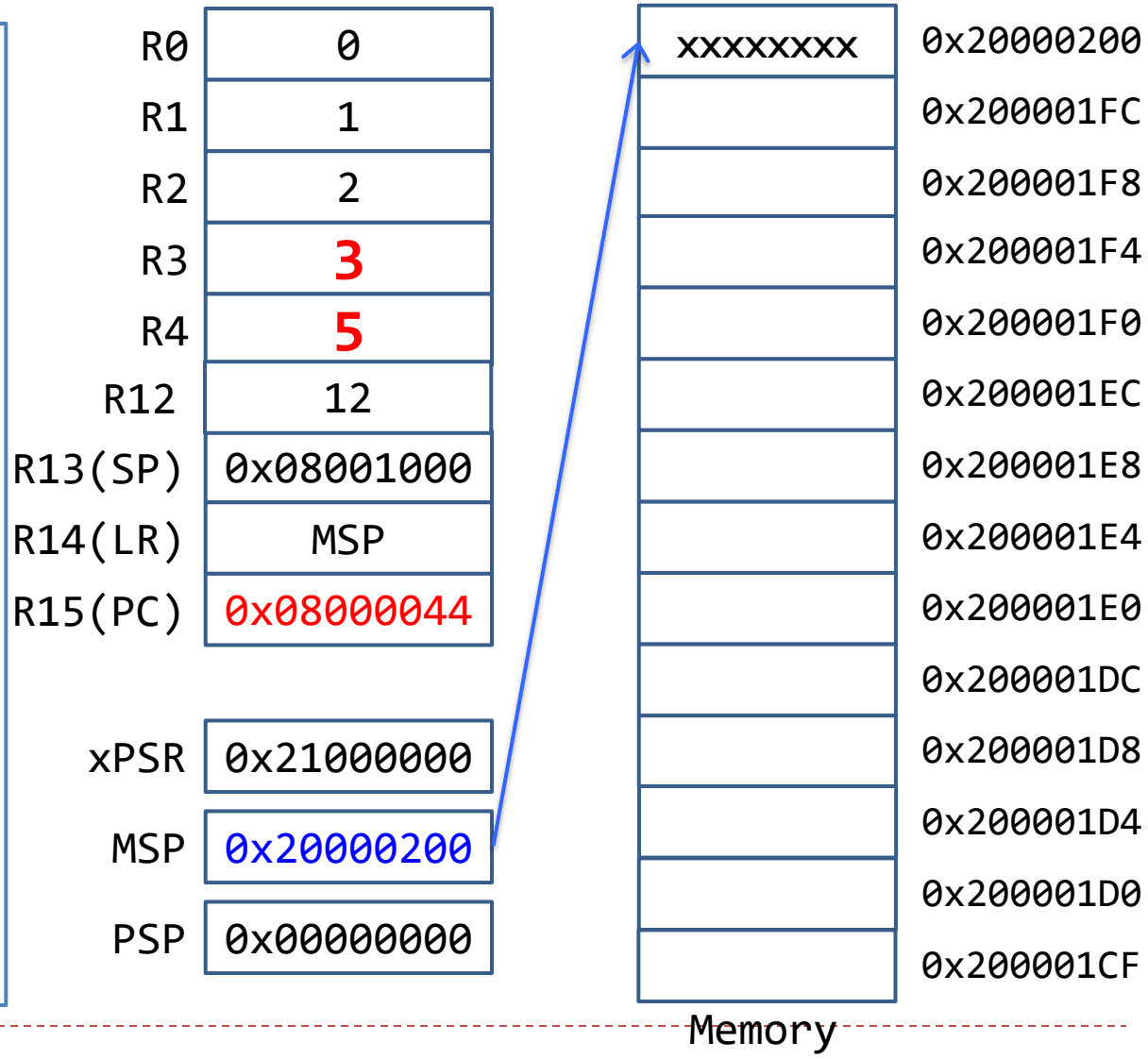
R3 is restored to its old value of 3 before calling SysTick_Handler; the new computed value of r3=4 is lost after returning from SysTick_Handler!



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 execution

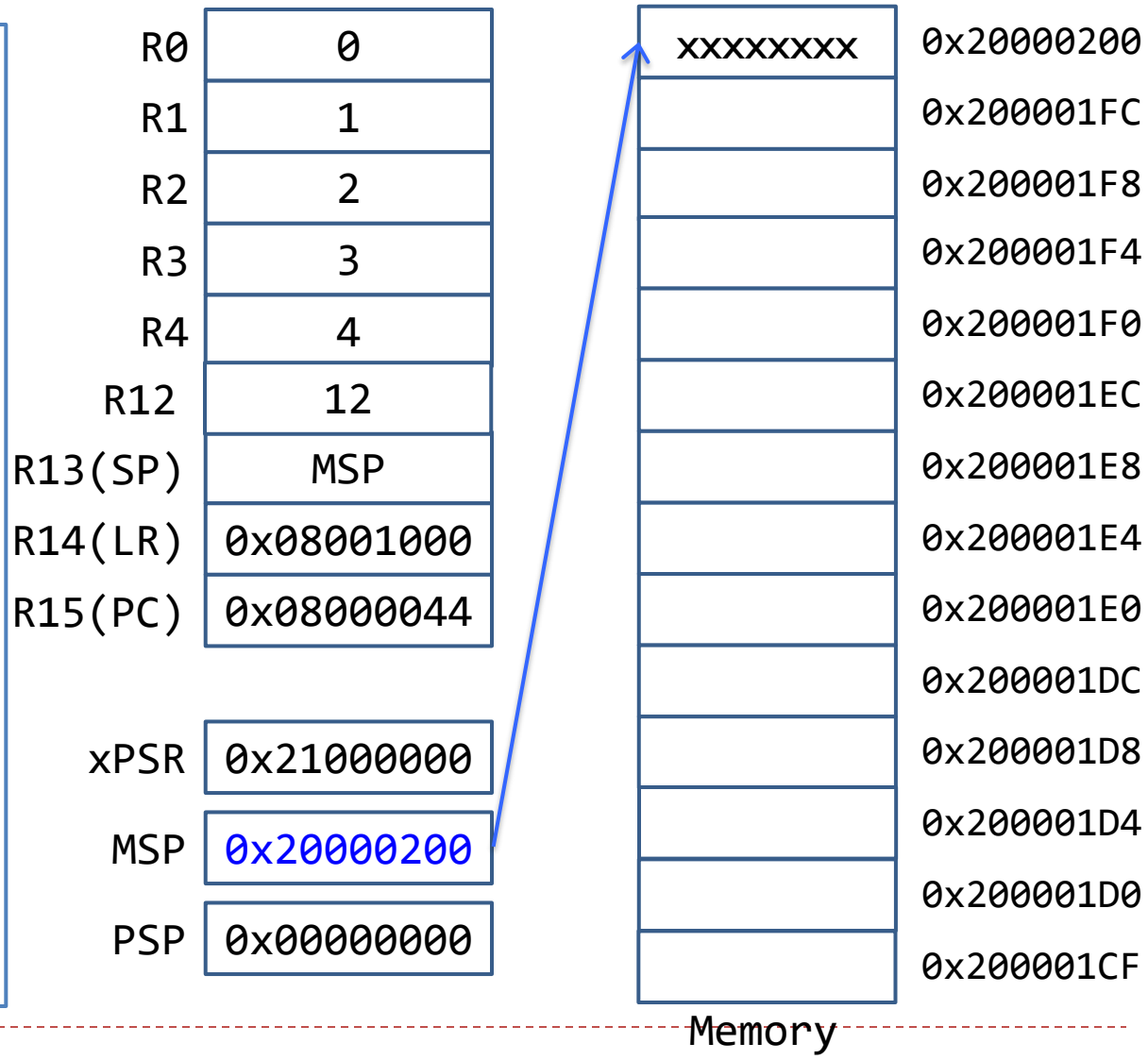


Another example with a function call in the interrupt handler

Buggy Program

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
```

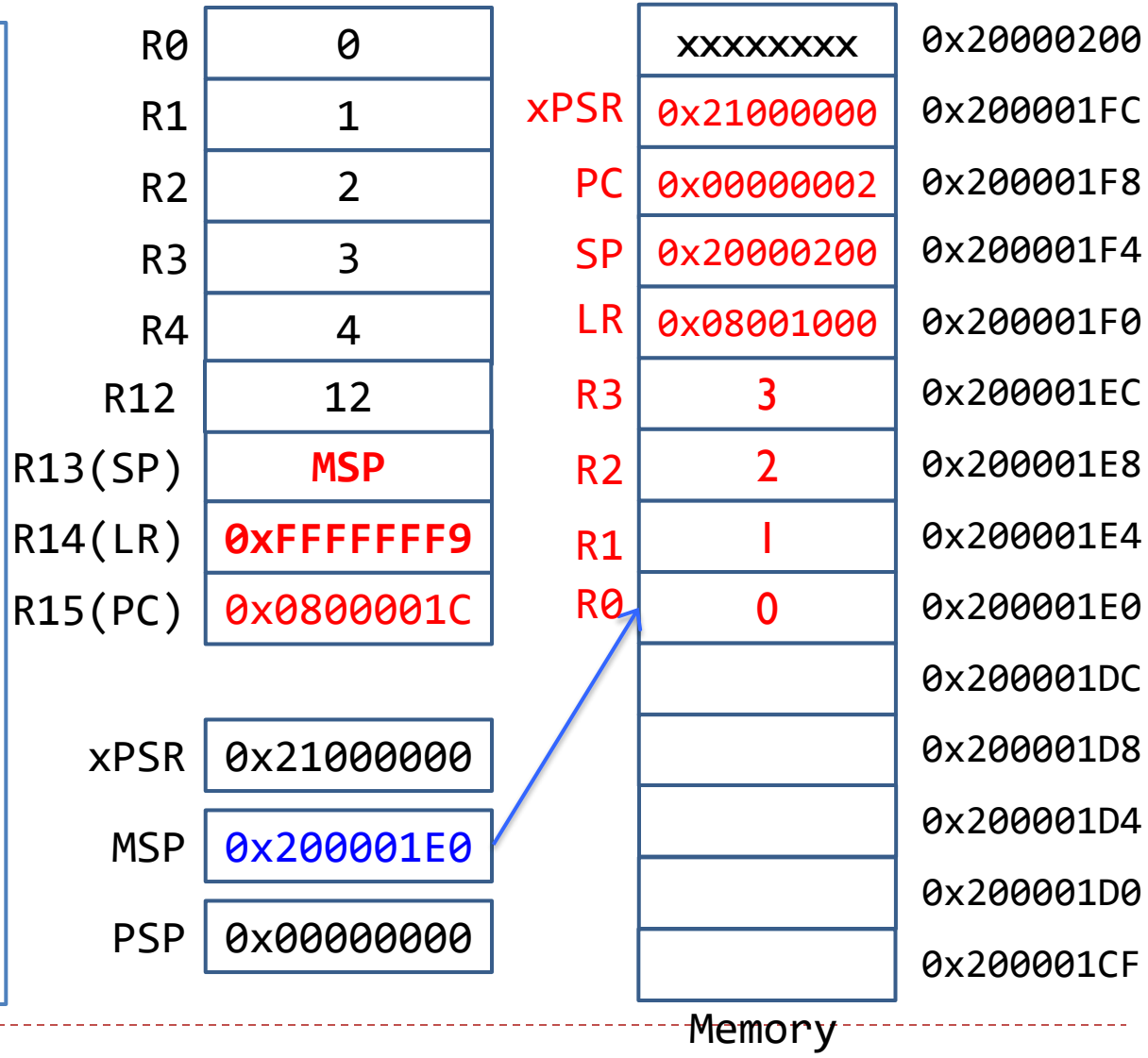


Interrupt: Stacking & Uns

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.

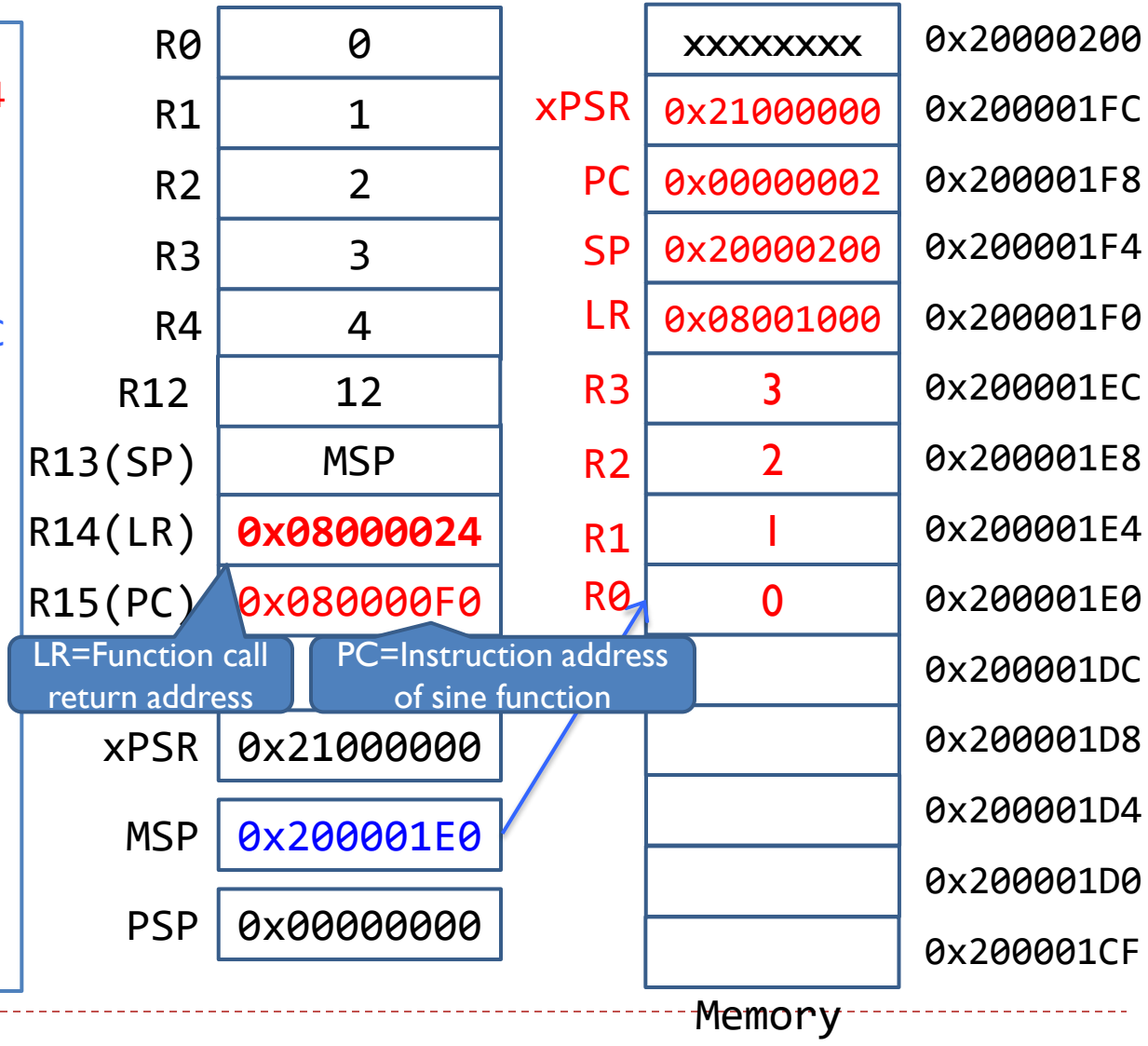


Interrupt: Stacking & Uns

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 to contain address of instruction right after function call BL sine



Buggy Program

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)	MSP
R14(LR)	0x08000024
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

Buggy Program

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
   BL  sine
   BX  lr
   ENDP
```

BL sine Updates LR register

R0	0
R1	1
R2	2
R3	3
R4	4
R12	12
R13(SP)	MSP
R14(LR)	0x08000024
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

What is 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:
  - Option 1: LDR lr,=0xFFFFFFF9 just before function return
  - Option 2: PUSH{lr}/POP{lr} in the function to save and restore the original LR value of 0xFFFFFFF9
  - Option 3: PUSH{lr}/POP{PC} is equivalent to Option 2,
    - POP {PC} is equivalent to POP {lr} followed by BX lr

Fix the bug: Wrong solution

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

SysTick_Handler PROC
  EXPORT SysTick_Handler
  ADD r4, #1
  BL  sine
  LDR lr,=0xFFFFFFF9
  BX  lr
  ENDP
```

- This fix works for the specific example, but this SysTick_Handler definition is only correct if LR had the value of 0xFFFFFFF9 before calling the sine() function.
- Not always true (refer to Slide 16, with 3 possible value of EXC_RETURN for LR)

Fix the bug: 2 options (similar to nested function calls)

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

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

Option 1

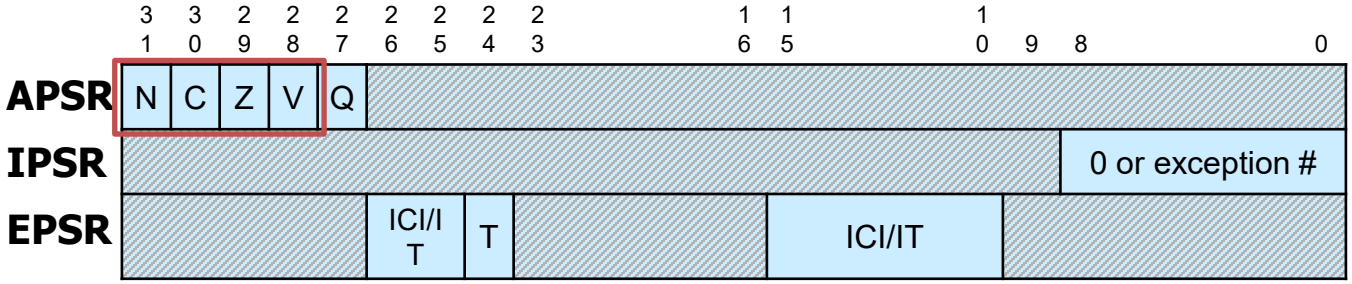
```
__main PROC
  …
  MOV r3,#0
  …
  ENDP

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

Option 2

Review

Status Registers

- A 32-bit PSR (Program Status Register) stores a collection of 1-bit status flags and other information, divided into three bit fields:
  - APSR (Application Program Status Register), IPSR (Interrupt Program Status Register), and EPSR (Execution Program Status Register).
  - PSR = APSR | IPSR | EPSR ("|" stands for bitwise OR)



  - CPSR (Current Program Status Register) holds PSR of the current instruction being executed
  - (Exception number == interrupt number in this context)

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 ARM CMSIS or chip manufacturers
- Interrupt Number in Program Status Register (PSR) = 16 + Interrupt Number by ARM CMSIS or chip manufacturers

Interrupt Numbers defined by ARM CMSIS or chip manufacturers

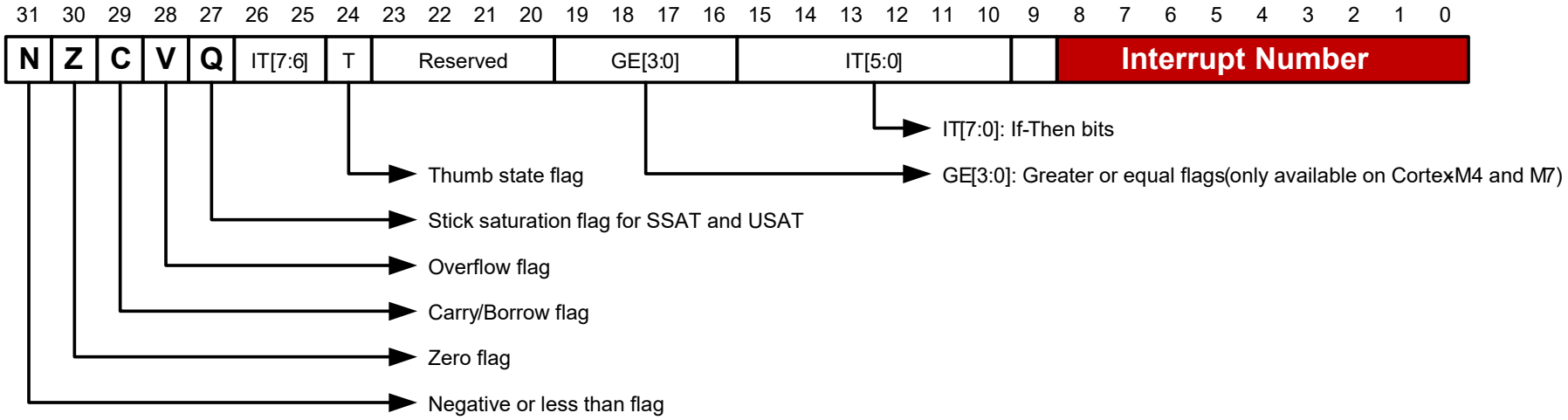
Interrupt Numbers in PSR

Interrupt Number

Interrupt number for CMSIS functions (NVIC: Nested Vectored Interrupt Controller)

```
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)



CMSIS Interrupt Number

```
/******  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

Peripheral Interrupts

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)

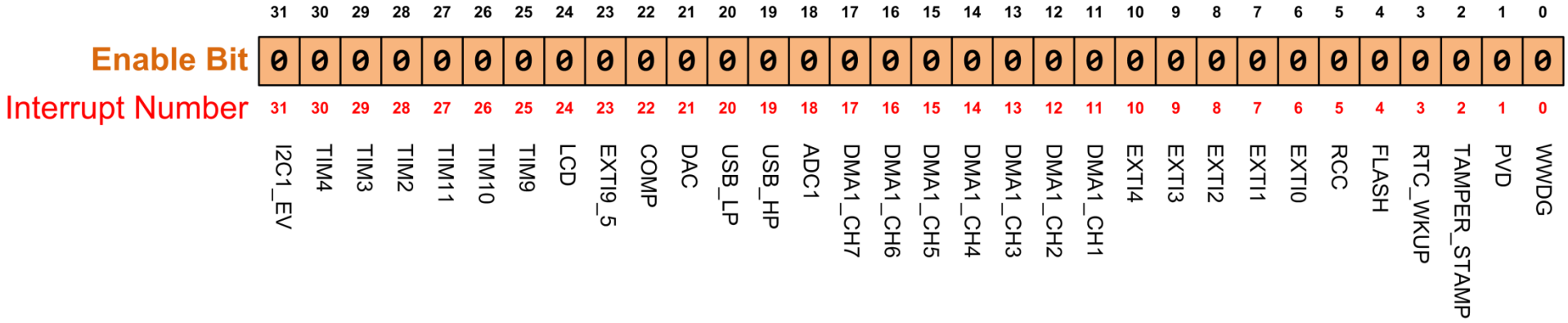
Enable/Disable Exception/Interrupt

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

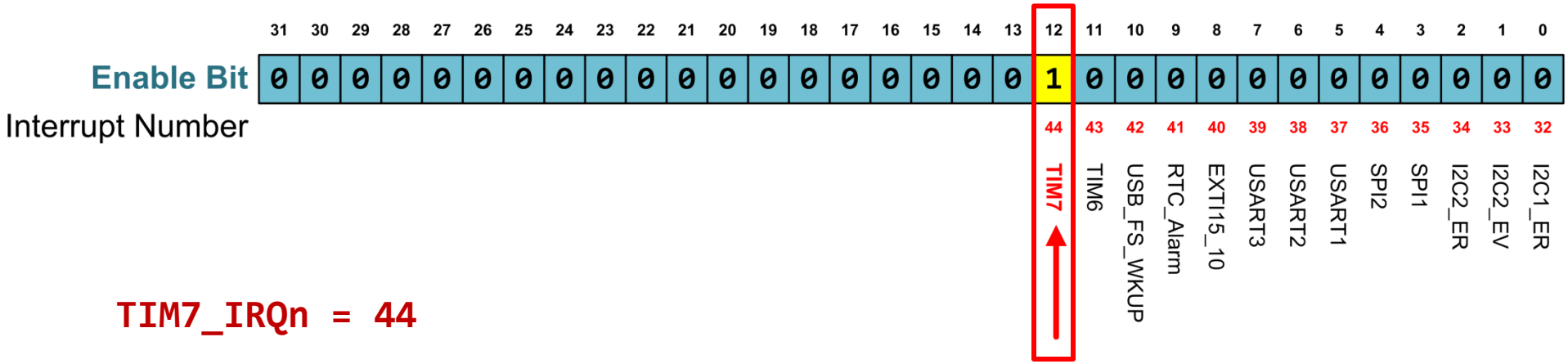
- Enable a peripheral interrupt
  - **`ISER`** for enabling
  - **`ICER`** for disabling

Enabling Peripheral Interrupts

Interrupt Set Enable Register 0 (ISER0)



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



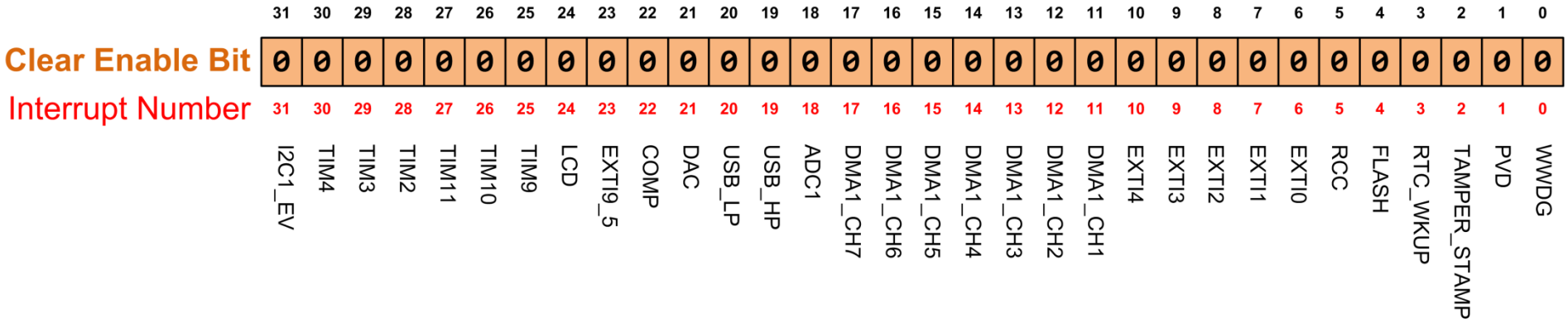

```
TIM7_IRQn = 44

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

Explanations of Previous Slide

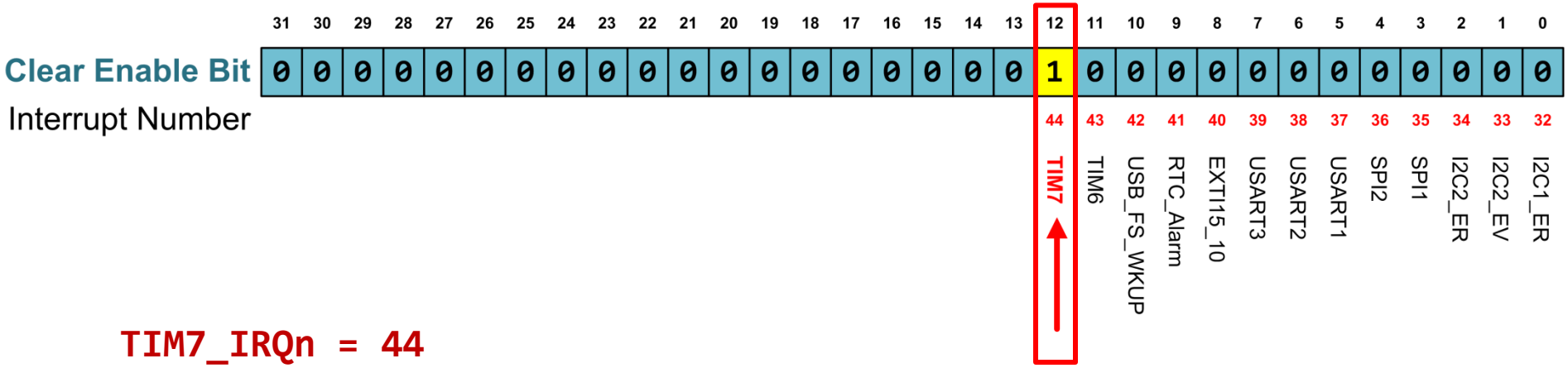
- We can enable a peripheral interrupt by writing 1 to the corresponding bit of the ISER register.
- To enable interrupt "Timer 7" with interrupt number 44:
  - ISER0 controls interrupts 0 to 31; ISER1 controls interrupts 32 to 63.
  - To enable interrupt 44, we set bit 12 (44-32) of ISER1 to 1 (other bits to 0) by executing ISER[1] = 1 << 12.
    - Setting a bit to 1 in ISER automatically clears the corresponding bit in ICER (sets it to 0)
    - Clearing a bit in ISER by writing to it has no effect

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 of Previous Slide

- We can disable a peripheral interrupt by writing 1 to the corresponding bit of the ICER register.
- To disable interrupt "Timer 7" with interrupt number 44:
  - ISER0 controls interrupts 0 to 31; ISER1 controls interrupts 32 to 63.
  - To disable interrupt 44, we set bit 12 (44-32) of ICER1 to 1 (other bits to 0) by executing ICER[1] = 1 << 12.
    - Setting a bit to 1 in ICER automatically clears the corresponding bit in ISER (sets it to 0)
    - Clearing a bit in ICER by writing to it has no effect
- Separating enable bits and disable bits in two separate sets of registers, ICER and ISER, provides great convenience and flexibility for programmers

Disable/Enable Peripheral Interrupts

- For all peripheral interrupts: `IRQn ≥ 0`
- Method 1: use functions
  - `NVIC_EnableIRQ (IRQn);  // Enable interrupt`
  - `NVIC_DisableIRQ (IRQn); // Disable interrupt`
- Method 2: directly set the bit in ISER/ICER
  - Enable:
    - `NVIC->ISER[ IRQn / 32] = 1 << (IRQn % 32);`
    - Divide IRQn by 32 to find out in which ISER register the target enable bit is located; The bit offset within the target ISER register is determined by the result of IRQn mod 32.
    - More efficient solution:
    - `NVIC->ISER[ IRQn >> 5] = 1 << (IRQn & 0x1F);`
    - IRQn>>5 == IRQn/32; IRQn & 0x1F == IRQn%32
  - Disable:
    - `NVIC->ICER[ IRQn >> 5] = 1 << (IRQn & 0x1F);`

Interrupt Priority

- Inverse Relationship:
  - Lower priority value means higher importance.
    - Priority of Interrupt A = 5,
    - Priority of Interrupt B = 2,
    - B has a higher importance than A.

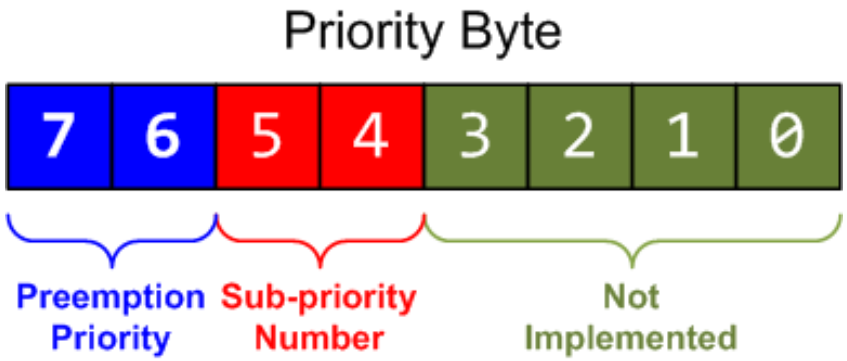
- Priorities are pre-defined for Reset, HardFault, and NMI.

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

- Other interrupts have priority ≥0, adjustable by software.

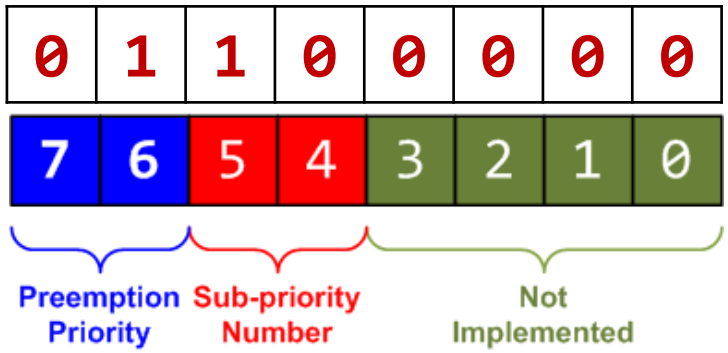
Interrupt Priority

- 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.



Interrupt Priority

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



core_cm4.h or core_cm3.h

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

```
IP = 0x60
```

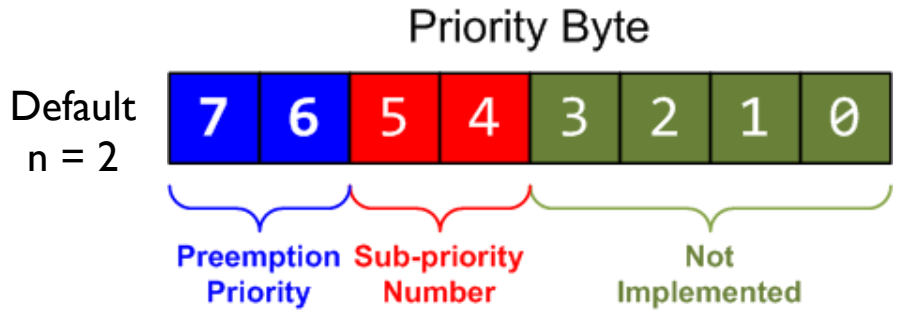
The priority is stored in the most significant 4 bits of the interrupt priority byte. NVIC_SetPriority (7, 6) sets the priority byte of interrupt 7 to 0x60=01100000 in binary. It is equivalent to:

```
NVIC->IP[7] = 6 << 4;
```

Preemption and Sub-priority Configuration

- `NVIC_SetPriorityGrouping(n)`
  - Can configure the number of bits in the preemption priority field and the sub-priority field

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)

Masking Priority

- The MRS and MSR instructions are used to access the PRIMASK, FAULTMASK, and BASEPRI registers. Examples:

```
MRS     R0, PRIMASK          ; Read PRIMASK register into R0

MOV     R0, #1
MSR     PRIMASK, R0          ;Write R0=1 into PRIMASK register. (Cannot directly
                             ; write #1 to special register with MSR)

MOV     R0, #0x60
MSR     BASEPRI, R0          ;Write R0= 0x60 into BASEPRI register. (Cannot directly
                             ; write #0x60 to special register with MSR)
```

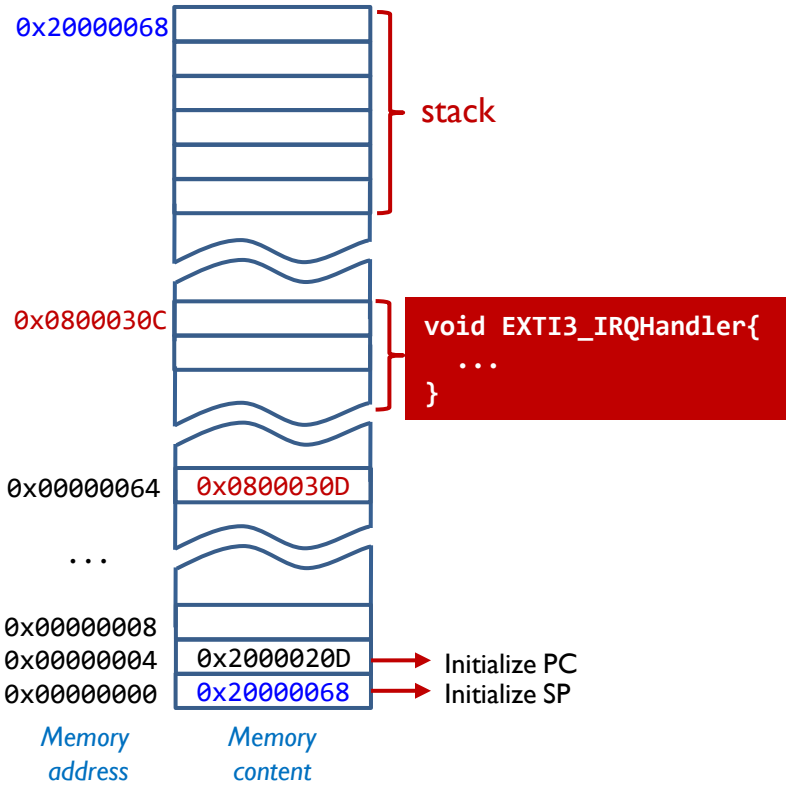
- Convenience functions can also be used, which calls MRS/MSR within them:

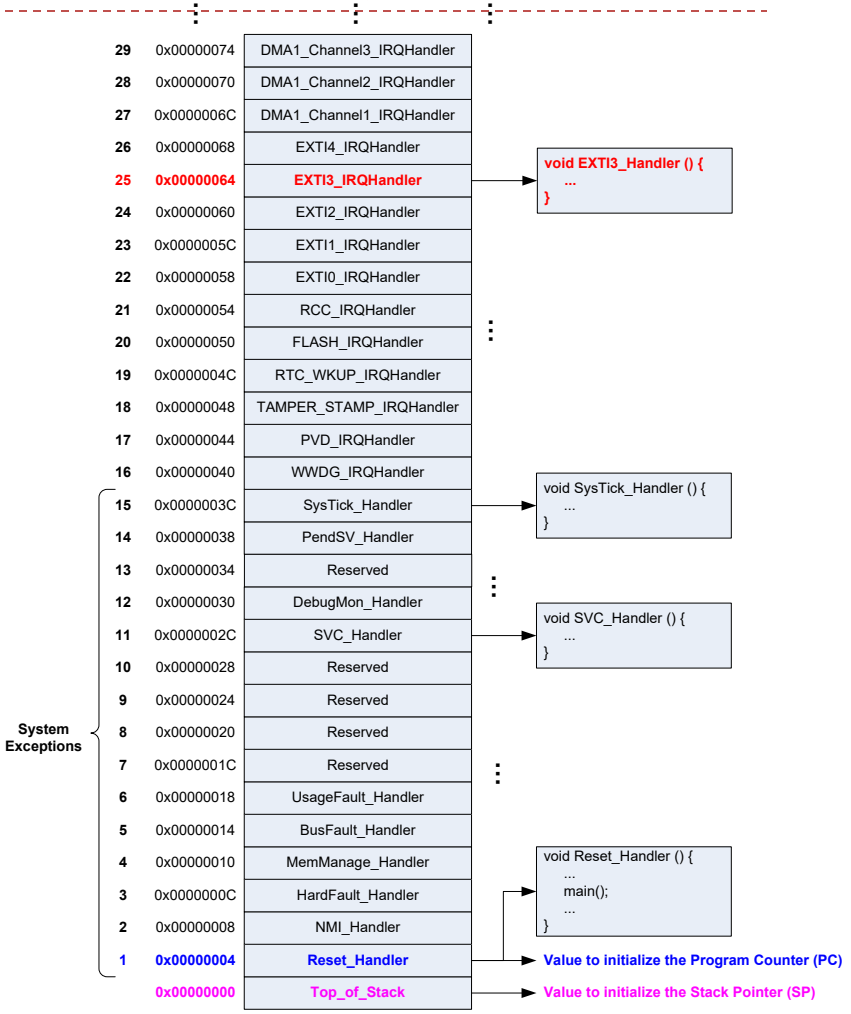
__get_BASEPRI, __set_BASEPRI, __get_PRIMASK, __set_PRIMASK,
__get_FAULTMASK, __set_FAULTMASK

- These mask registers can only be set at the Privileged Level.

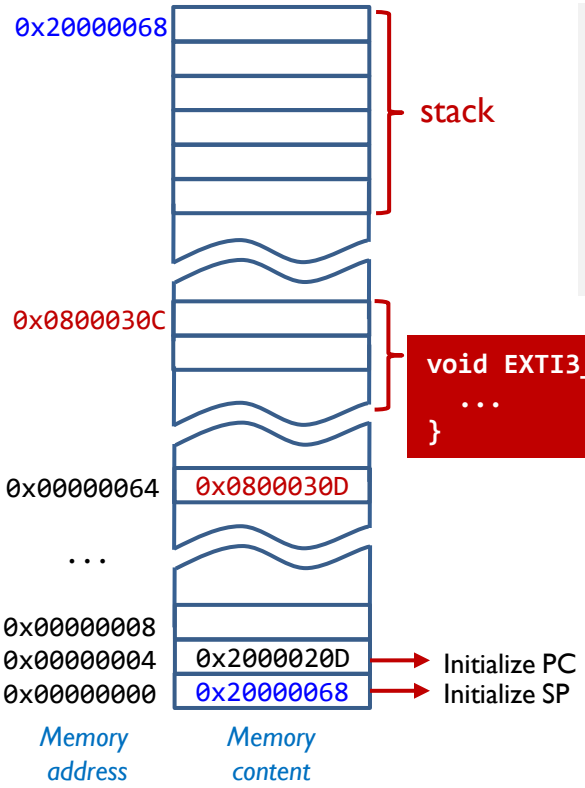
An Example of either a single interrupt or 2 nested interrupts

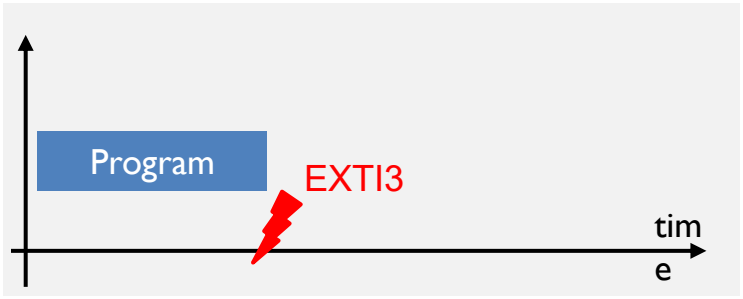
Interrupt Service Routine (ISR)

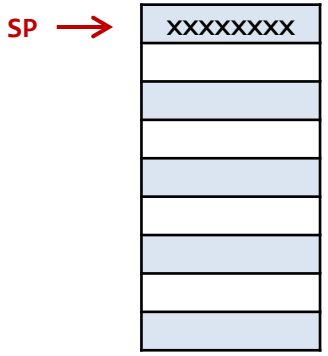




Single Interrupt

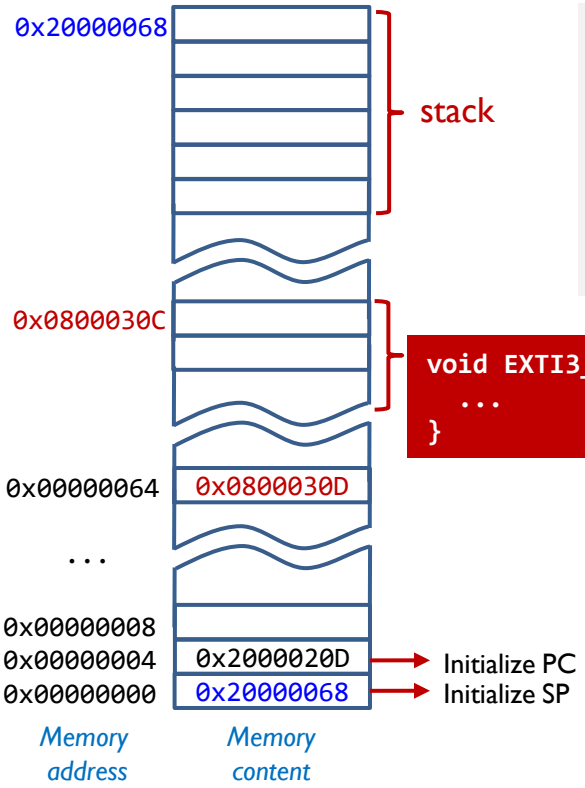


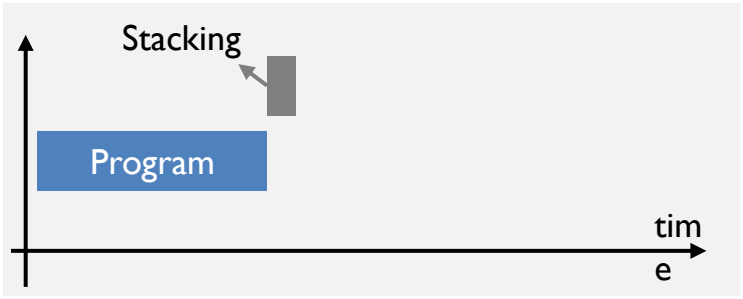


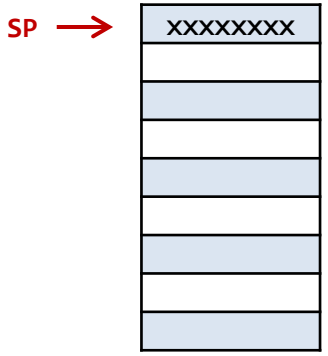


	EXTI3				
Interrupt Number	12	11	10	9	8
Enable Register	0	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	2	8

Single Interrupt

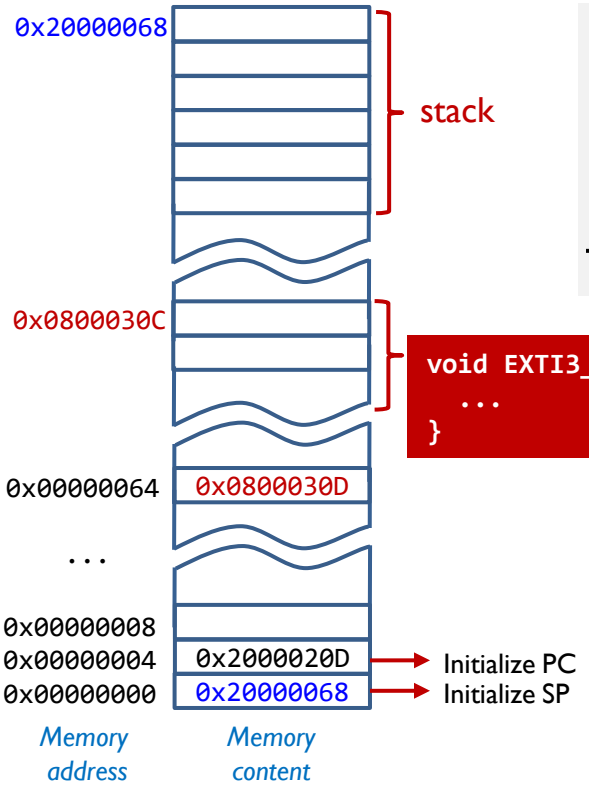


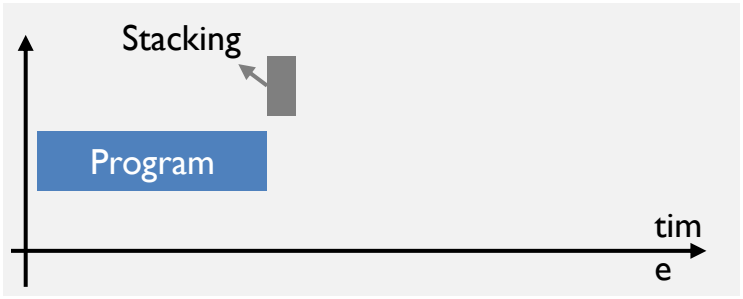


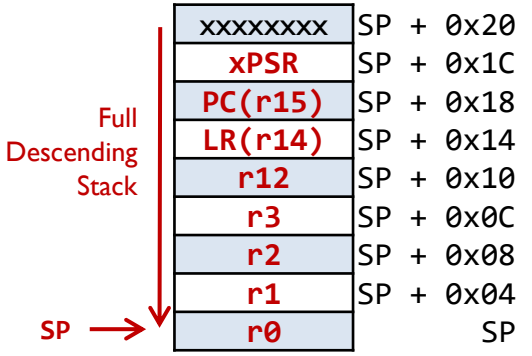


	EXTI3				
Interrupt Number	12	11	10	9	8
Enable Register	0	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	1	0
Priority Register	3	4	7	2	8

Single Interrupt: after stacking

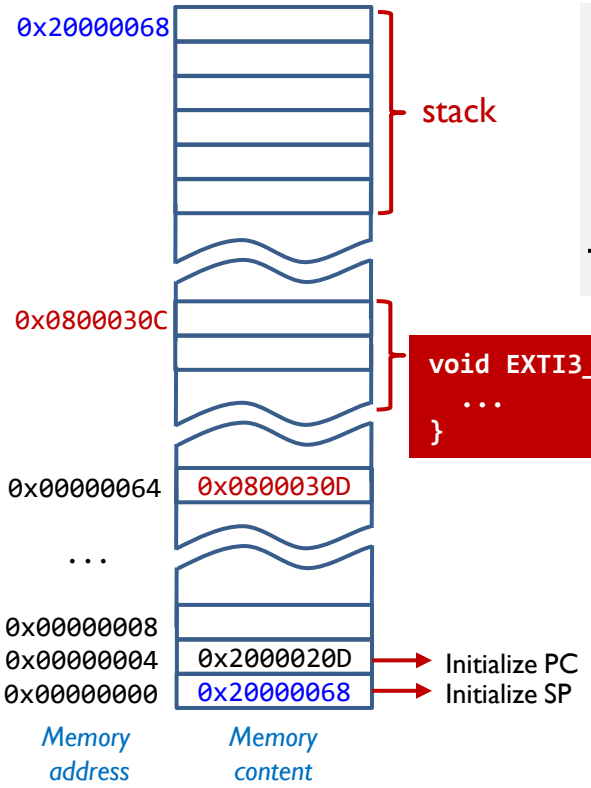


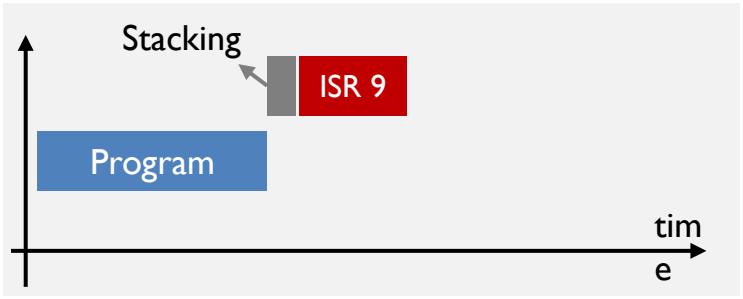


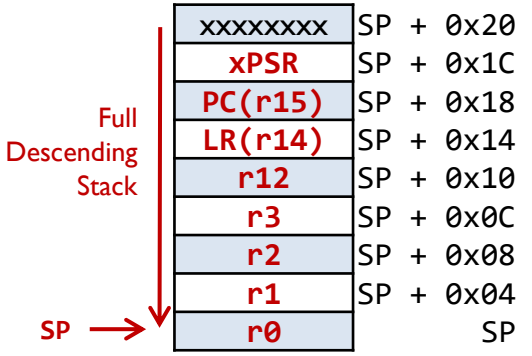


	EXTI3				
Interrupt Number	12	11	10	9	8
Enable Register	0	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	1	0
Priority Register	3	4	7	2	8

Single Interrupt



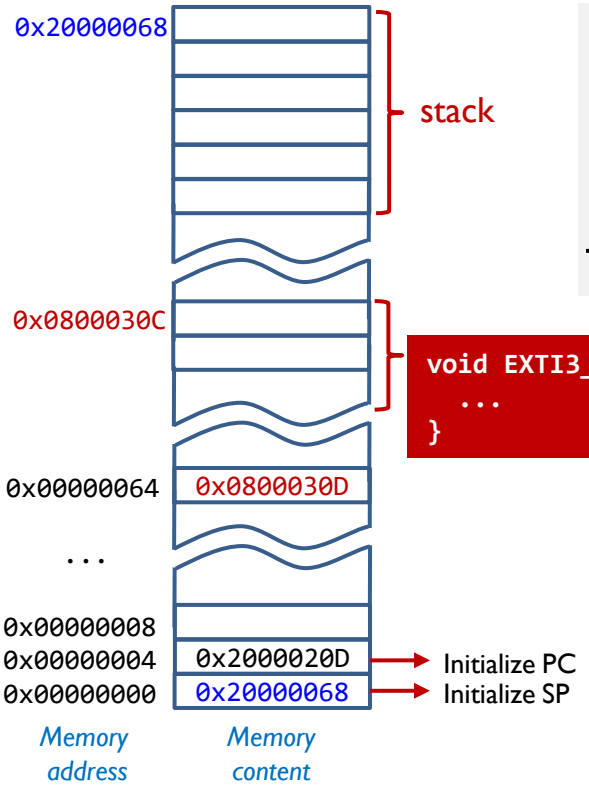


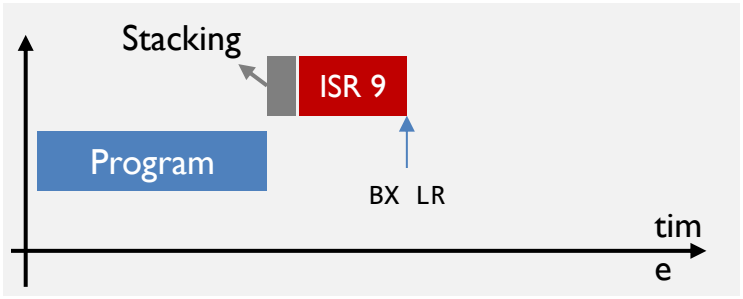


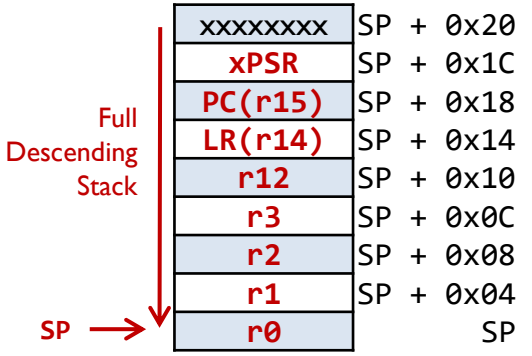
EXTI3

Interrupt Number	12	11	10	9	8
Enable Register	0	0	0	1	0
Active Register	0	0	0	1	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	2	8

Single Interrupt

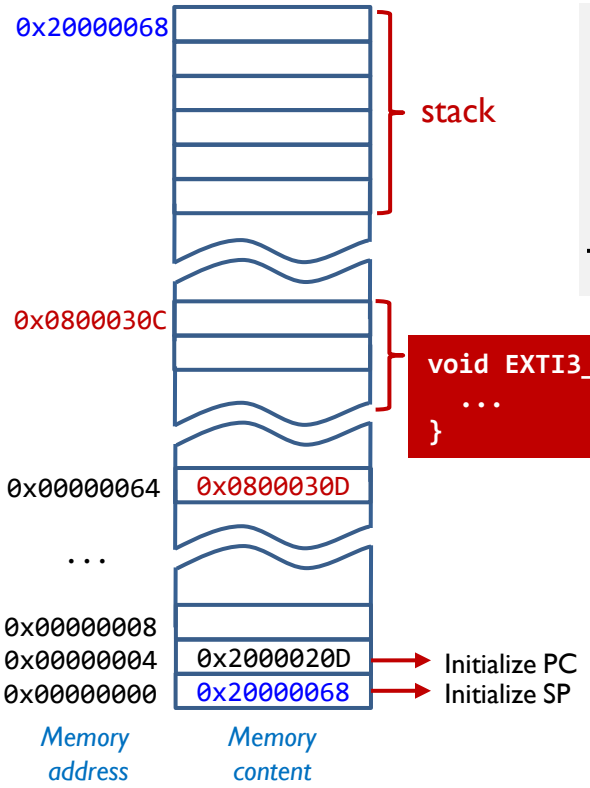


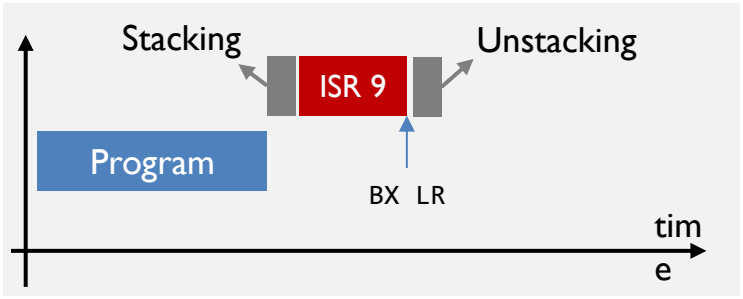


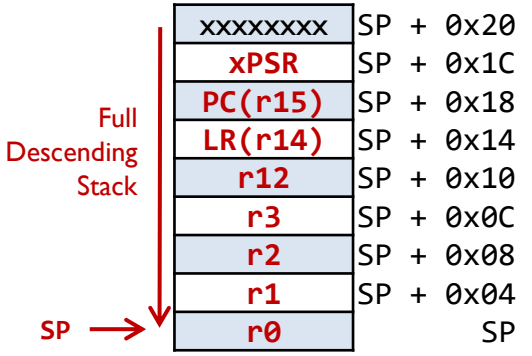


	EXTI3				
Interrupt Number	12	11	10	9	8
Enable Register	0	0	0	1	0
Active Register	0	0	0	1	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	2	8

Single Interrupt

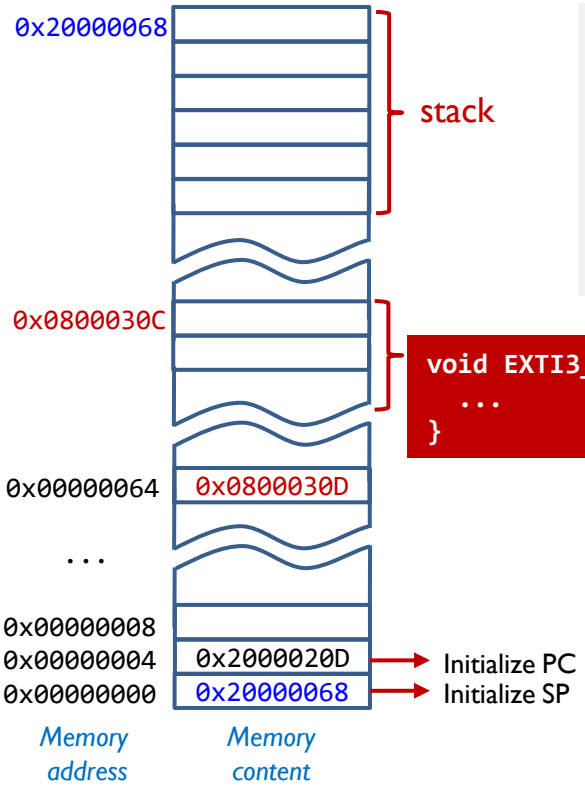


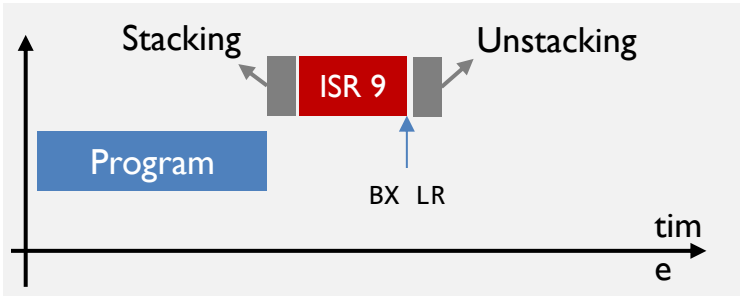


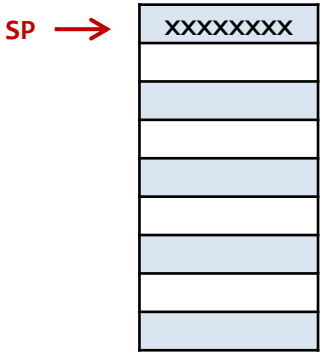


	EXTI3				
Interrupt Number	12	11	10	9	8
Enable Register	0	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	2	8

Single Interrupt: after unstacking

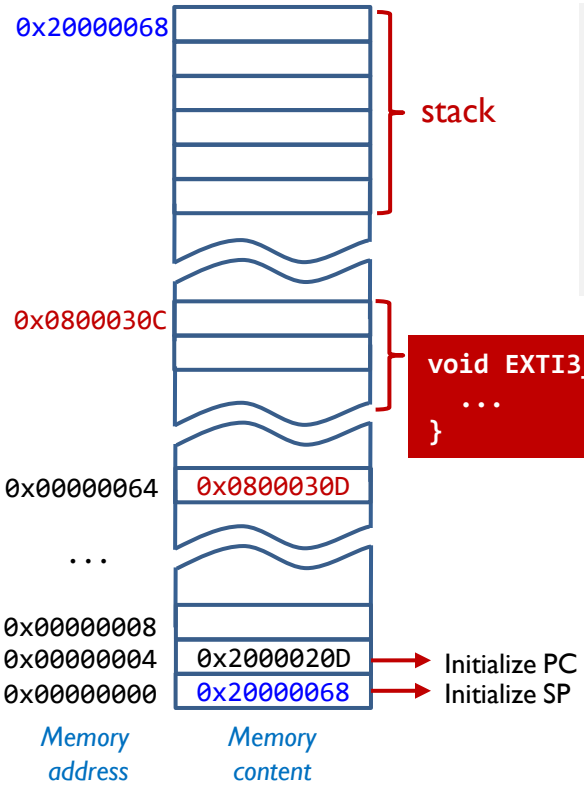


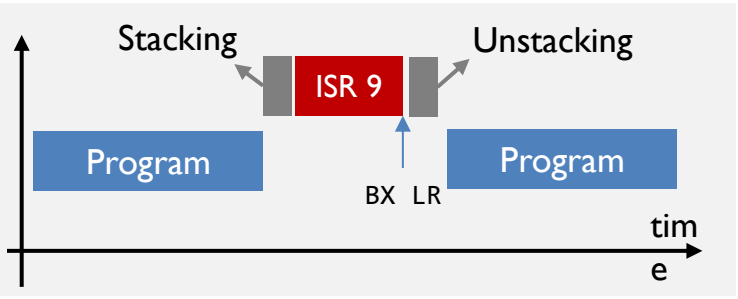


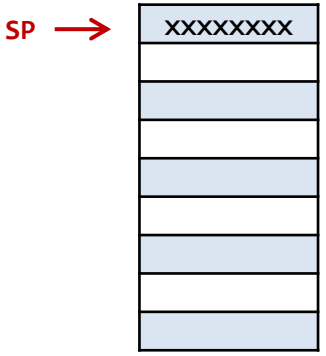


	EXTI3				
Interrupt Number	12	11	10	9	8
Enable Register	0	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	2	8

Single Interrupt





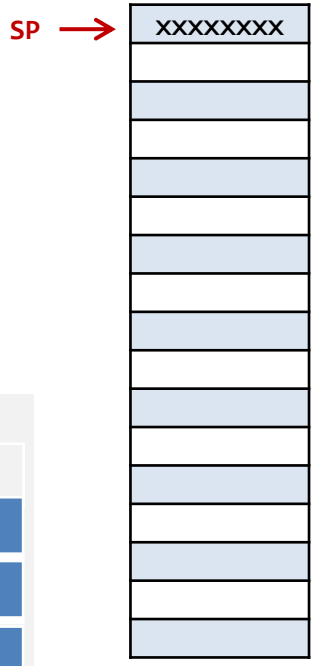


	EXTI3				
Interrupt Number	12	11	10	9	8
Enable Register	0	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	2	8

Nested Interrupts: Example of Preemption

External Interrupt 3 (EXTI3) arrives. It has priority=5.

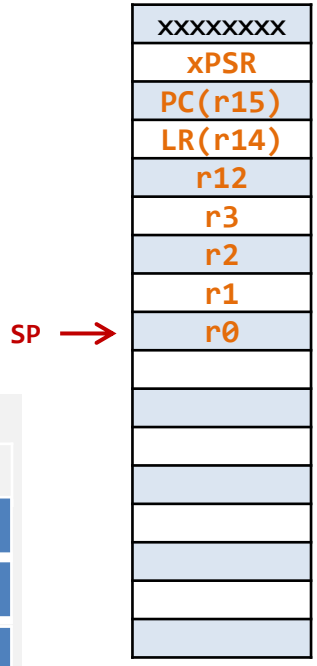
	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3



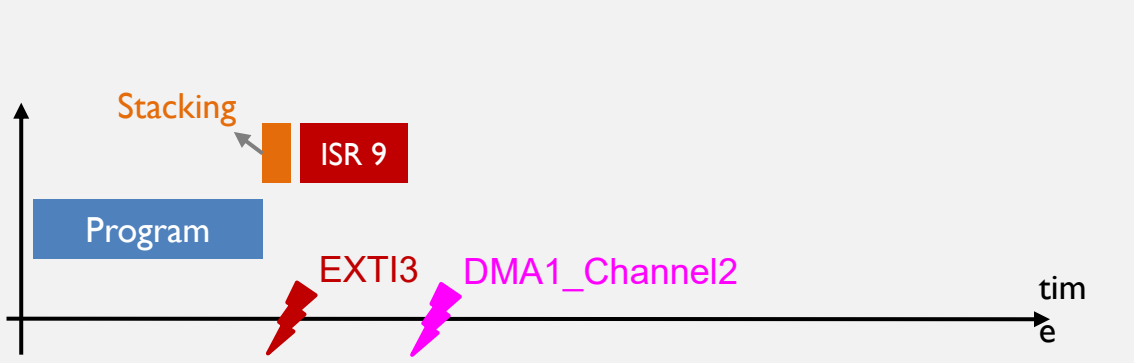
Nested Interrupts: Example of Preemption

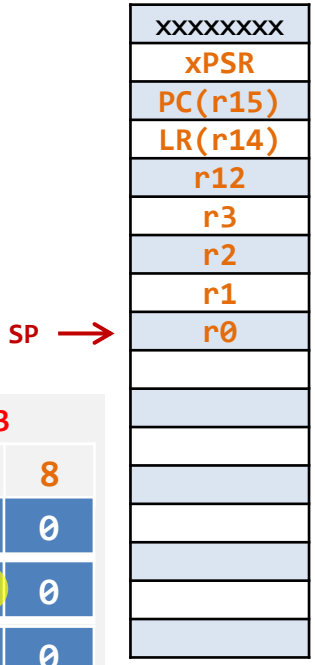
ISR 9 that serves EXTI3 starts execution after stacking operation.

	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	1	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3



Nested Interrupts: Example of Preemption





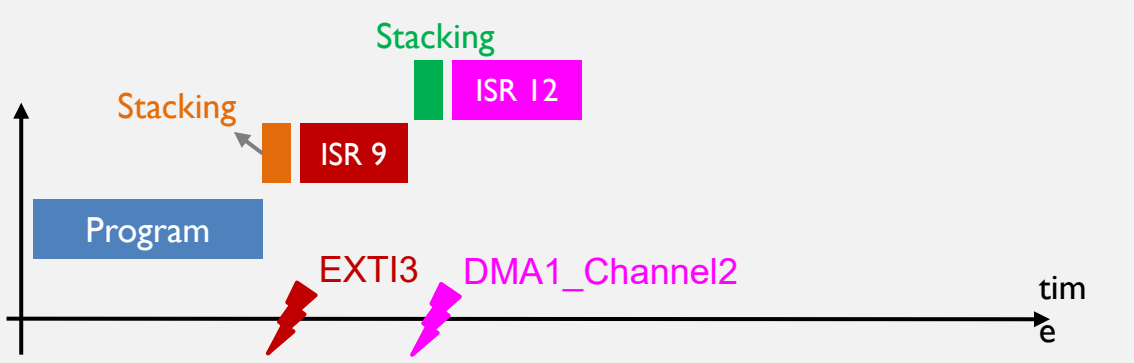
Suppose another interrupt request (DMA1_Channel2) arrives, before ISR 9 completes; this new interrupt has higher importance (priority=3) than the current interrupt being served (EXTI3 priority=5). Hence the CPU will respond to the new interrupt by preempting the currently executing ISR 9 that serves EXTI3.

EXT3 → ISR 9
DMA1_Channel2 → ISR 12

	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	1	0
Pending Register	1	0	0	0	0
Priority Register	3	4	7	5	3

Lower priority value means higher importance.

Nested Interrupts: Example of Preemption



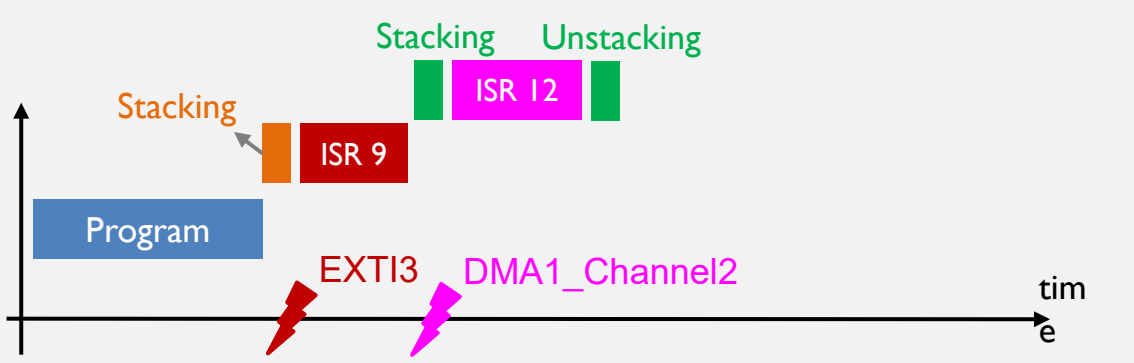
The stacking operation for ISR 12 that serves the new interrupt DMA1_Channel2 pushes another set of 8 registers onto the MSP. Note that these two sets of registers have different values. The first set holds register values, for the user program; the second set holds register values for ISR 9. Afterwards, ISR 12 starts execution.

	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	1	0	0	1	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3

SP →

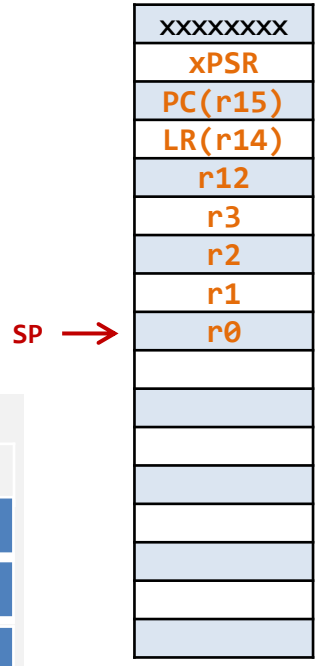
xxxxxxxx
xPSR
PC(r15)
LR(r14)
r12
r3
r2
r1
r0
xPSR
PC(r15)
LR(r14)
r12
r3
r2
r1
r0

Nested Interrupts: Example of Preemption

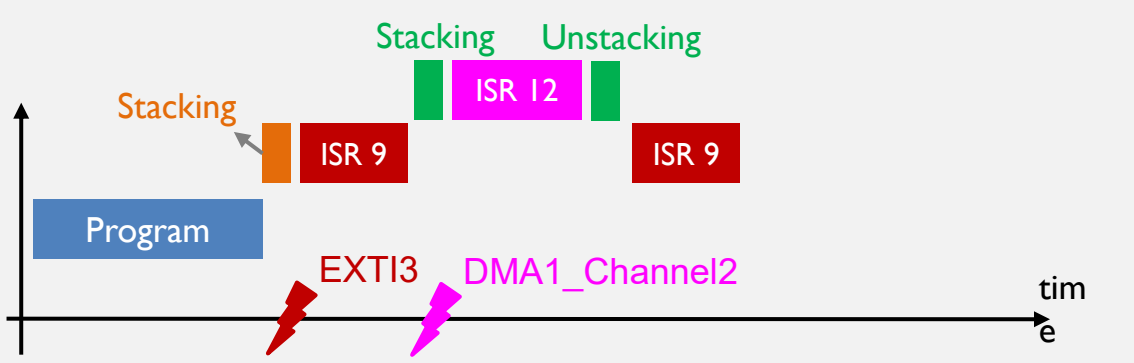


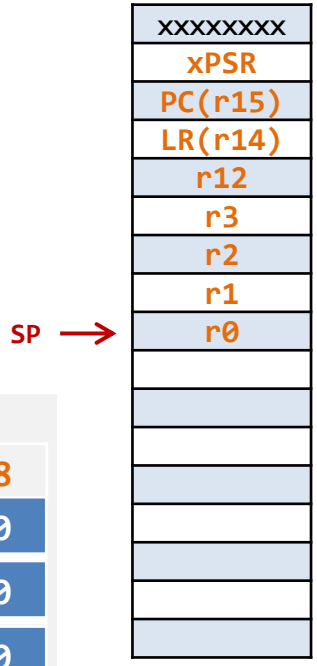
After ISR12 completes, unstacking operation is performed, and ISR 9 resumes execution.

	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	1	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3



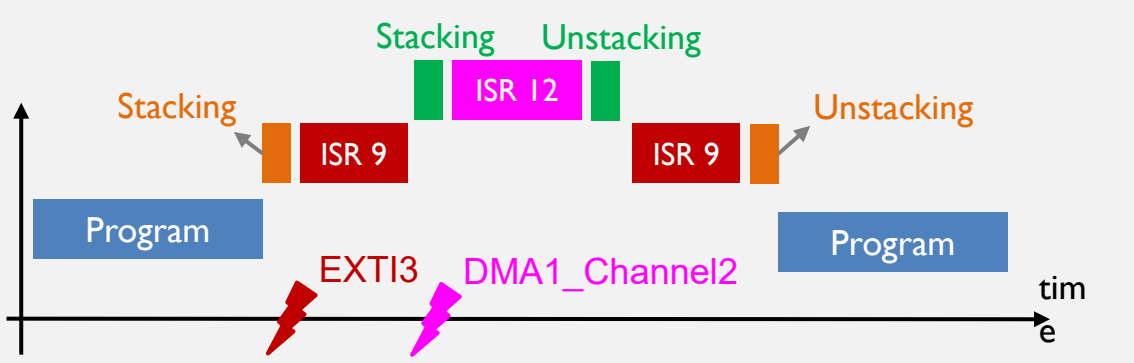
Nested Interrupts: Example of Preemption

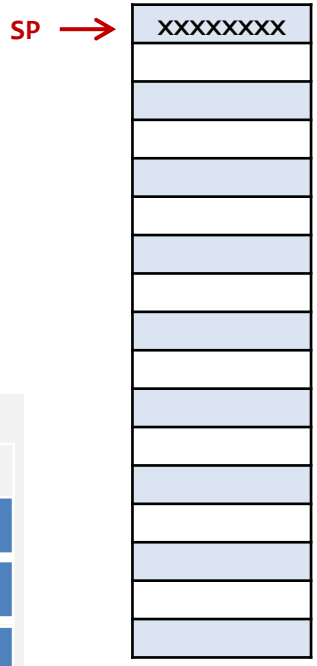




	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	1	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3

Nested Interrupts: Example of Preemption





After ISR9 completes, unstacking operation is performed, and the user program resumes execution.

	DMA1_Channel2			EXTI3	
Interrupt Number	12	11	10	9	8
Enable Register	1	0	0	1	0
Active Register	0	0	0	0	0
Pending Register	0	0	0	0	0
Priority Register	3	4	7	5	3

Nested Interrupts: Tail Chaining

- In the previous example, we have shown that, an interrupt with higher importance can preempt another interrupt with lower importance.
- Suppose interrupt EXTI4 has lower importance than EXTI3. If EXTI4 arrives before the ISR 9 that serves EXTI3 completes, then it will wait in pending state until ISR 9 completes execution.
  - EXTI3 → ISR 9; EXTI4 → ISR 10
- The stacking/unstacking operations when ISR 9 finishes and ISR 10 starts immediately are unnecessary. Tail chaining is used to reduce latency of context switching between two ISRs, by omitting stacking/unstacking operations.
  - Typically the stacking/unstacking operation takes12 cycles each, while tail chaining takes only 6 cycles.