

EE3220 System-on-Chip Design

Lecture Note 7

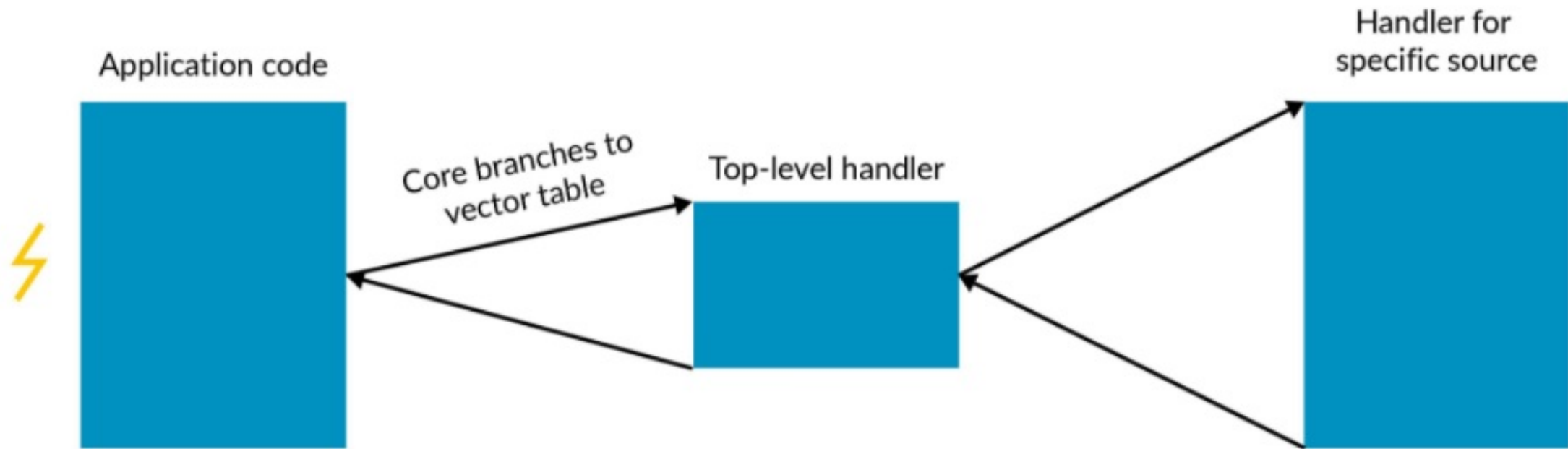
Exceptions and Interrupts

Outline

- Entering an exception handler
 - Push context to the current stack
 - Examine current vector table
- Exiting an exception handler
 - Pop from the stack
- Processor core interrupts
 - NVIC registers and states
- Timing analysis
 - Interrupt response latency
- Program design with interrupts
- Data sharing safely

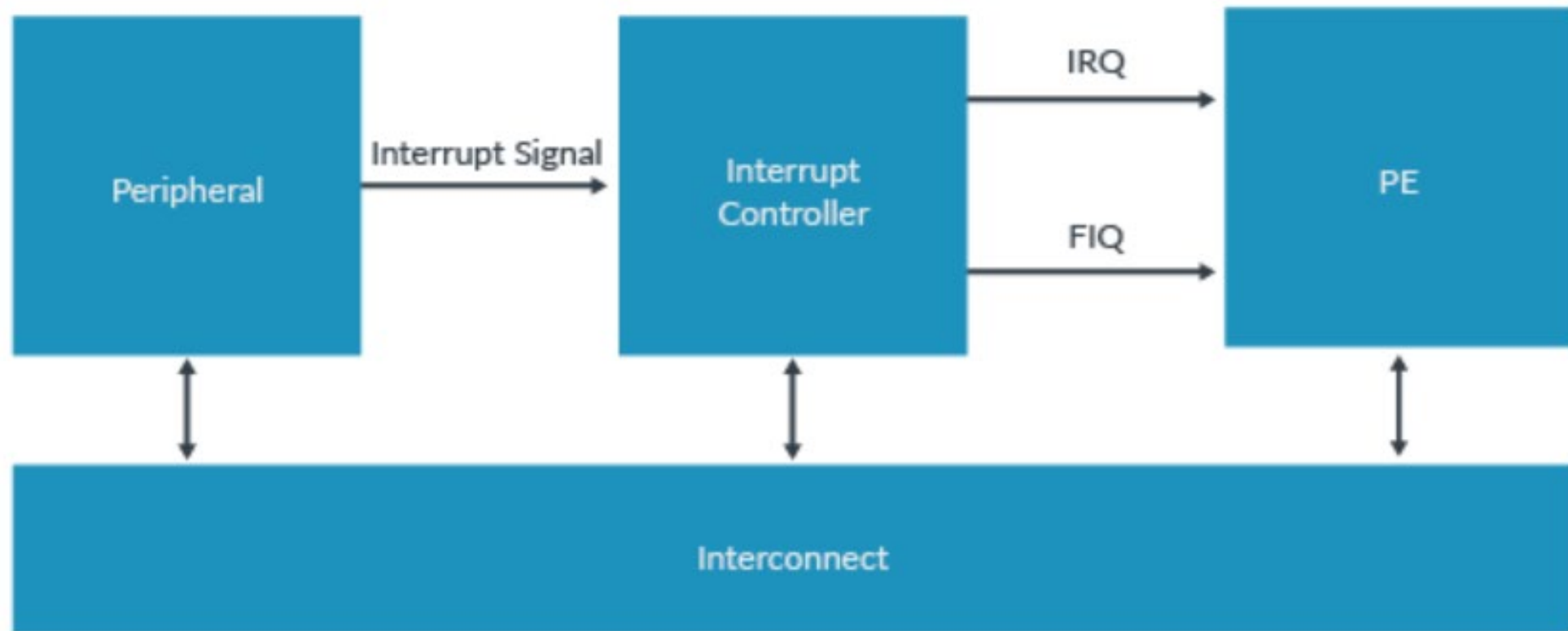
Handling Exceptions

- When an exception occurs, the current program flow is interrupted.
- The state that the processor is in when the exception is recognized is known as the state the exception is taken from.



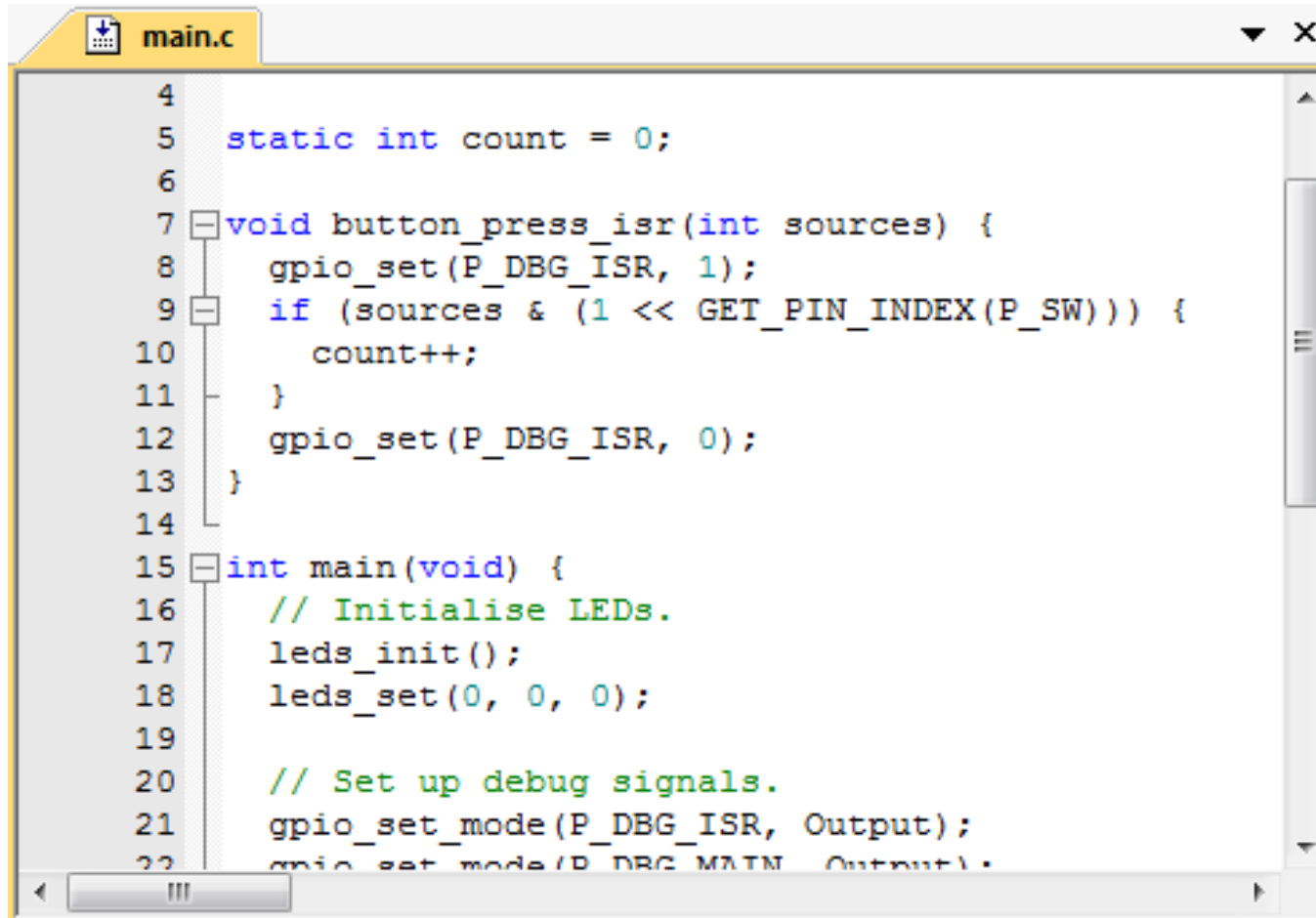
Interrupt Signals to the Interrupt Controller

- Interrupts are signaled from a peripheral to the interrupt controller using a dedicated hardware signal, as shown below.



Example Exception Handler

- We will examine processor's response to exception in detail

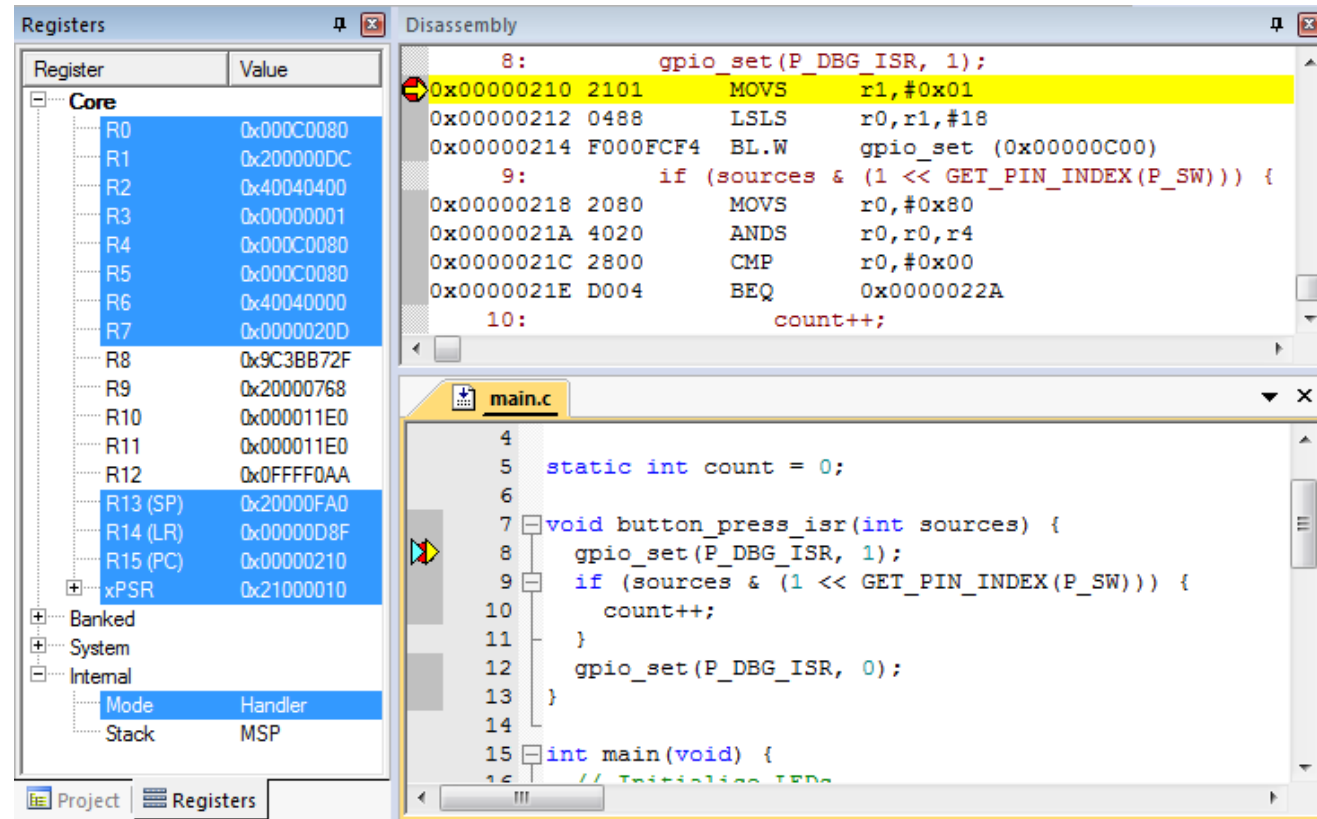


```
main.c
4
5 static int count = 0;
6
7 void button_press_isr(int sources) {
8     gpio_set(P_DBG_ISR, 1);
9     if (sources & (1 << GET_PIN_INDEX(P_SW))) {
10         count++;
11     }
12     gpio_set(P_DBG_ISR, 0);
13 }
14
15 int main(void) {
16     // Initialise LEDs.
17     leds_init();
18     leds_set(0, 0, 0);
19
20     // Set up debug signals.
21     gpio_set_mode(P_DBG_ISR, Output);
22     gpio_set_mode(P_DBG_MAIN, Output);
```

Here is an example of an exception handler for an interrupt request.

Use Debugger for Detailed Processor View

- Can see registers, stack, source code, disassembly (object code)
- Note: Compiler may generate code for function entry
- Place breakpoint on Handler function declaration line in source code, not at first line of function code



Normal Flow of Execution vs. Processor Exceptions

- Program counter (PC) increases sequentially through the address space, with branches to nearby labels or branch and links to subroutines.
- Processor exceptions occur when this normal flow of execution is diverted, to enable the processor to handle events generated by internal or external sources.
- Examples of such events are:
 - externally generated interrupts
 - an attempt by the processor to execute an undefined instruction
 - accessing privileged operating system functions
- When an exception is taken, execution branches to an address in the *vector table*.

Entering an Exception Handler

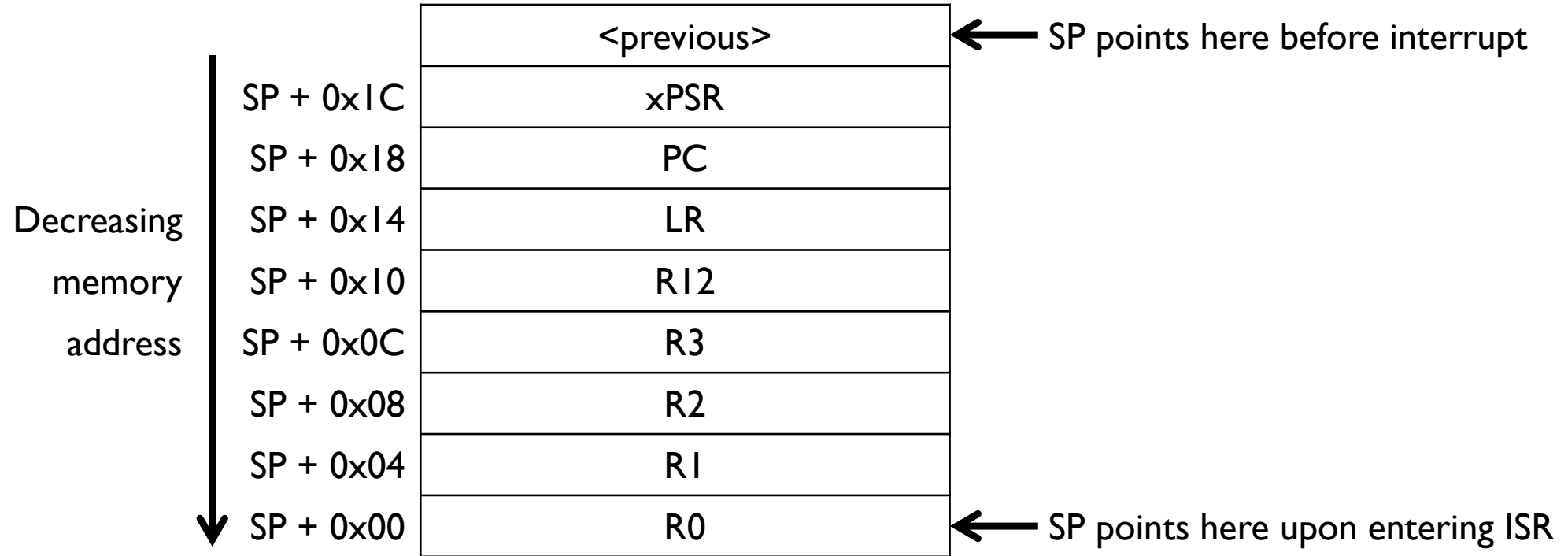
CPU's Hardwired Exception Processing

1. Finish current instruction (except for lengthy instructions)
2. Push context (8 32-bit words) onto current stack (MSP or PSP)
 - xPSR, Return address, LR (R14), R12, R3, R2, R1, R0
3. Switch to handler/privileged mode, use MSP
4. Load PC with address of exception handler
5. Load LR with EXC_RETURN code
6. Load IPSR with exception number
7. Start executing code of exception handler
8. Usually 16 cycles from exception request to execution of first instruction in handler

I. Finish Current Instruction

- Most instructions are short and finish quickly
- Some instructions may take many cycles to execute
 - Load Multiple (LDM), Store Multiple (STM), Push, Pop, MULS (32 cycles for some CPU core implementations)
- This will delay interrupt response significantly
- If one of these is executing when the interrupt is requested, the processor:
 - *abandons* the instruction
 - responds to the interrupt
 - executes the ISR
 - returns from interrupt
 - *restarts* the abandoned instruction

2. Push Context onto Current Stack



- Two SPs: Main (MSP), process (PSP)
- Which is active depends on operating mode, CONTROL register bit I
- Stack grows toward smaller addresses

Context Saved on Stack

Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x0000008C
R4	0x40040000
R5	0x00000ECC
R6	0x2EE00000
R7	0x00000EAB
R8	0x9C7FB76F
R9	0x20000768
R10	0x00000ECC
R11	0x00000ECC
R12	0x0FFF0AA
R13 (SP)	0x2000FD8
R14 (LR)	
R15 (PC)	
xPSR	
Banked	
MSP	0x2000FD8
PSP	0xBEF5DFF0
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	
Stack	

SP value is reduced since registers have been pushed onto stack

Memory 1					
Address: sp					
0x2000FD8:	00000000	00000000	00000000	0000008C	0FFF0AA
0x2000FEC:	00000A0B	00000A70	21000000	00000002	00000A0B
0x2001000:	622361E2	B5103D30	6800482F	04892101	492D4308
0x2001014:	20186008	FFD4F7FF	49292019	200562C8	F8EEF000
0x2001028:	30204824	49257940	20056288	F929F000	68004823

Saved R0

Saved R1

Saved R2

Saved R3

Saved R12

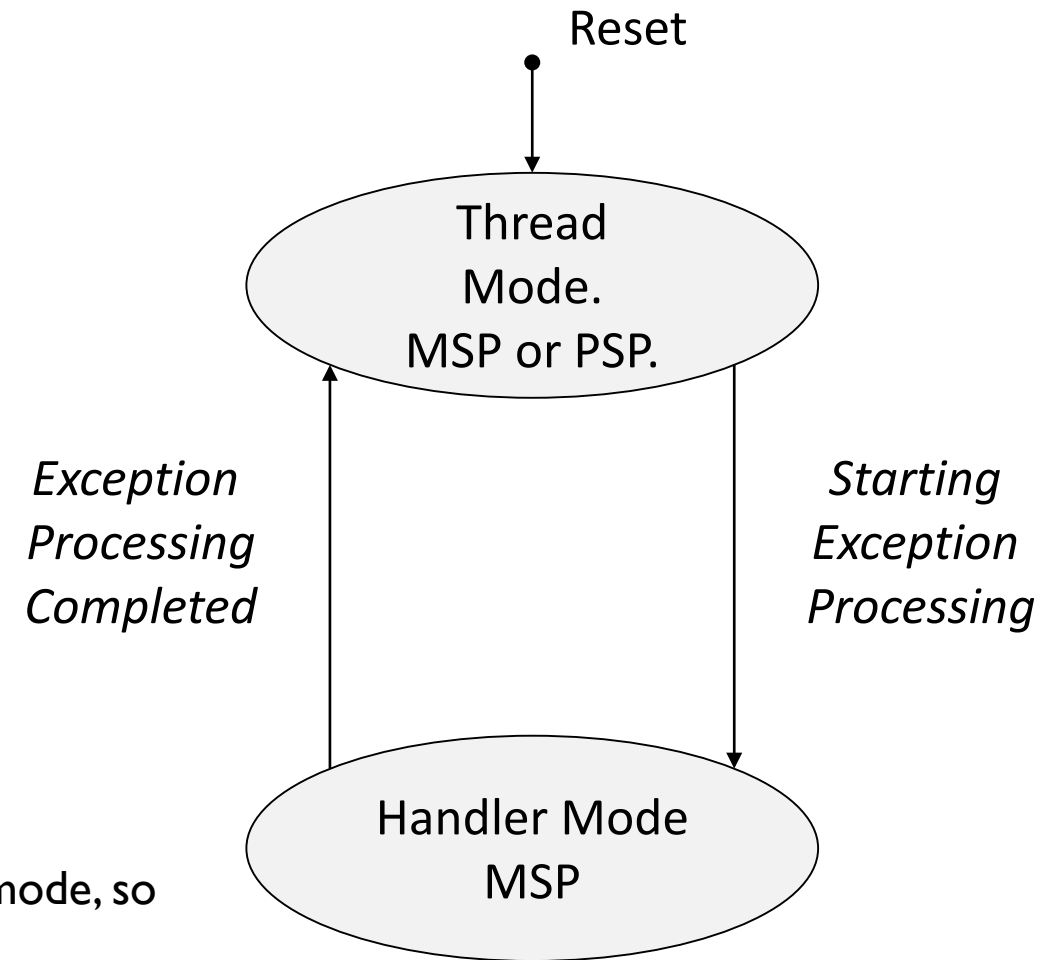
Saved LR

Saved PC

Saved xPSR

3. Switch to Handler/Privileged Mode

- Handler mode always uses Main SP



The exception handler executes in Handler/Privileged mode, so it uses the main stack pointer (MSP).

ARM Processor Modes

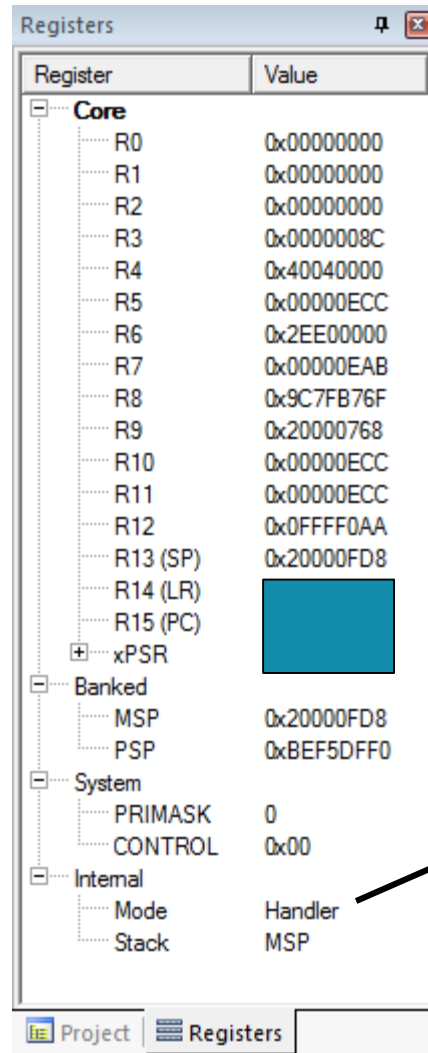
- The Cortex-M processor supports:
 - two operation modes, Thread mode and Handler mode
 - two execution modes, Privileged mode and User / Unprivileged mode.
- Thread mode is entered on reset and is the normal mode that programs run in.
- Thread mode can be privileged or unprivileged software execution. When in thread mode, code can be executed in either Privileged or User mode.
- Handler mode is entered as a result of an exception. All code is executed as Privileged. The core automatically switches to Privileged mode when exceptions occur.
- Privileged mode has full access rights.
- User mode has limited access rights. The limitations include:
 - restrictions on instruction use such as which fields can be used in MSR instructions
 - restrictions on the use of certain coprocessor registers
 - restrictions on access to memory and peripherals based on system design
 - restrictions on access to memory and peripherals imposed by the MPU configuration.

ARM Processor Modes

- More info: [*ARM Architecture Reference Manual*](#)

Processor mode	Architectures	Mode number
User	All	0b10000
FIQ	All	0b10001
IRQ	All	0b10010
Supervisor	All	0b10011
Monitor	Security Extensions only	0b10110
Abort	All	0b10111
Hyp	Virtualization Extensions only	0b11010
Undefined	All	0b11011
System	ARMv4 and later	0b11111

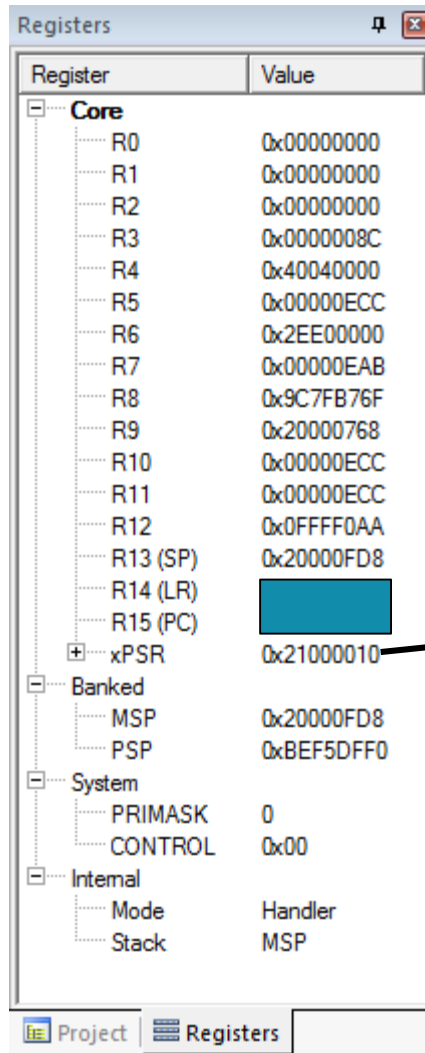
Handler and Privileged Mode



Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x0000008C
R4	0x40040000
R5	0x00000ECC
R6	0x2EE00000
R7	0x00000EAB
R8	0x9C7FB76F
R9	0x20000768
R10	0x00000ECC
R11	0x00000ECC
R12	0x0FFF0AA
R13 (SP)	0x2000FD8
R14 (LR)	
R15 (PC)	
xPSR	
Banked	
MSP	0x2000FD8
PSP	0xBEF5DFF0
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Stack	MSP

Mode changed to Handler. Was already using MSP

Update IPSR with Exception Number



Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x0000008C
R4	0x40040000
R5	0x0000ECC
R6	0x2EE00000
R7	0x0000EAB
R8	0x9C7FB76F
R9	0x20000768
R10	0x0000ECC
R11	0x0000ECC
R12	0x0FFF0AA
R13 (SP)	0x2000FD8
R14 (LR)	
R15 (PC)	
xPSR	0x21000010
Banked	
MSP	0x2000FD8
PSP	0xBEF5DFF0
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Stack	MSP

Exception number 0x10
(interrupt number + 0x10)

Now IPSR has been updated with the number of the exception.

4. Load PC With Address Of Exception Handler

Memory Address	Value
0x0000_0000	Initial Stack Pointer
0x0000_0004	Reset
0x0000_0008	NMI_IRQHandler
...	
	IRQ0_Handler
	IRQI_Handler
...	
Reset:	
...	
NMI_IRQHandler:	
...	
IRQ0_Handler:	
...	
IRQI_Handler:	

- The program counter is selected from the vector table depending on exception

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

Can Examine Vector Table With Debugger

Exception number	IRQ number	Vector	Offset
		Initial SP	0x00
1		Reset	0x04
2	-14	NMI	0x08
3	-13	HardFault	0x0C
4		Reserved	0x10
5			
6			
7			
8			
9			
10			
11	-5	SVCall	0x2C
12		Reserved	
13			
14	-2	PendSV	0x38
15	-1	SysTick	0x3C
16	0	IRQ0	0x40
17	1	IRQ1	0x44
18	2	IRQ2	0x48
.		.	
16+n	n	IRQn	0x40+4n

- Why is the vector odd?
- LSB of address indicates that handler uses Thumb code
- The first 15 interrupts are called “system exceptions”

Upon Entry to Handler

Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x0000008C
R4	0x40040000
R5	0x00000ECC
R6	0x2EE00000
R7	0x00000EAB
R8	0x9C7FB76F
R9	0x20000768
R10	0x00000ECC
R11	0x00000ECC
R12	0x0FFFF0AA
R13 (SP)	0x20000FD8
R14 (LR)	0xFFFFFFFF9
R15 (PC)	0x00000ACC
xPSR	0x21000010
Banked	
System	
Internal	
Mode	Handler
Stack	MSP

```
42: void switch_isr(void) {  
0x00000ACC B510      PUSH      {r4,lr}
```

PC has been loaded with
start address of handler

We can now see that the PC has now been loaded with the handler address.

5. Load LR With EXC_RETURN Code

EXC_RETURN	Return Mode	Return Stack	Description
0xFFFF_FFF1	0 (Handler)	0 (MSP)	Return to exception handler
0xFFFF_FFF9	1 (Thread)	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	1 (Thread)	1 (PSP)	Return to thread with PSP

- EXC_RETURN value generated by CPU to provide information on how to return
 - Which SP to restore registers from? MSP (0) or PSP (1)
 - Previous value of SPSEL (Stack Pointer Select)
 - Which mode to return to? Handler (0) or Thread (1)
 - Another exception handler may have been running when this exception was requested
- The EXC_RETURN is a 32-bit value, the upper 28 bits are all set to 1, with bit 2 and bit 3 used to provide information for exception return mechanism

Updated LR With EXC_RETURN Code


The screenshot shows a 'Registers' window with a table of registers. A blue arrow points to the 'R14 (LR)' row, which has the value '0xFFFFFFFF9'. The 'R13 (SP)' row has the value '0x2000FD8'. The 'R15 (PC)' row has the value '0x0000ACC'. The 'xPSR' register has the value '0x2100010'. The 'Internal' section shows 'Mode' as 'Handler' and 'Stack' as 'MSP'.

Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x0000008C
R4	0x40040000
R5	0x0000ECC
R6	0x2EE00000
R7	0x0000EAB
R8	0x9C7FB76F
R9	0x20000768
R10	0x0000ECC
R11	0x0000ECC
R12	0x0FFF0AA
R13 (SP)	0x2000FD8
R14 (LR)	0xFFFFFFFF9
R15 (PC)	0x0000ACC
xPSR	0x2100010
Banked	
System	
Internal	
Mode	Handler
Stack	MSP

Now the LR is updated with the exception return code.

6. Start Executing Exception Handler

- Exception handler starts running, unless preempted by a higher-priority exception
- Exception handler may save additional registers on stack
 - E.g. if handler may call a subroutine, LR and R4 must be saved



```
42: void switch_isr(void) {  
0x00000ACC B510      PUSH      {r4,lr}
```

And finally our handler code starts running. The first thing this code does is to save some additional registers.

After Handler Has Saved More Context

Registers	
Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x0000008C
R4	0x40040000
R5	0x00000ECC
R6	0x2EE00000
R7	0x00000EAB
R8	0x9C7FB76F
R9	0x20000768
R10	0x00000ECC
R11	0x00000ECC
R12	0x0FFFF0AA
R13 (SP)	0x20000FD0
R14 (LR)	0xFFFFFFFF9
R15 (PC)	0x00000ACE
+ xPSR	
xPSR	0x21000010
Banked	
MSP	0x20000FD0
PSP	0xBEF5DFF0
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Stack	MSP

```
42: void switch_isr(void) {  
0x00000ACC B510 PUSH {r4,lr}
```

Memory 2	
Address: sp	
0x20000FD0:	40040000 FFFFFFF9 00000000 00000000 00000000
0x20000FE4:	0000008C 0FFFF0AA 00000A0B 00000A70 21000000
0x20000FF8:	00000002 00000A0B 000001F1 000001F1 000001F1
0x2000100C:	000001F1 000001F1 000001F1 000001F1 000001F1
0x20001020:	000001F1 000001F1 000001F1 000001F1 000001F1

Labels for Saved Registers:

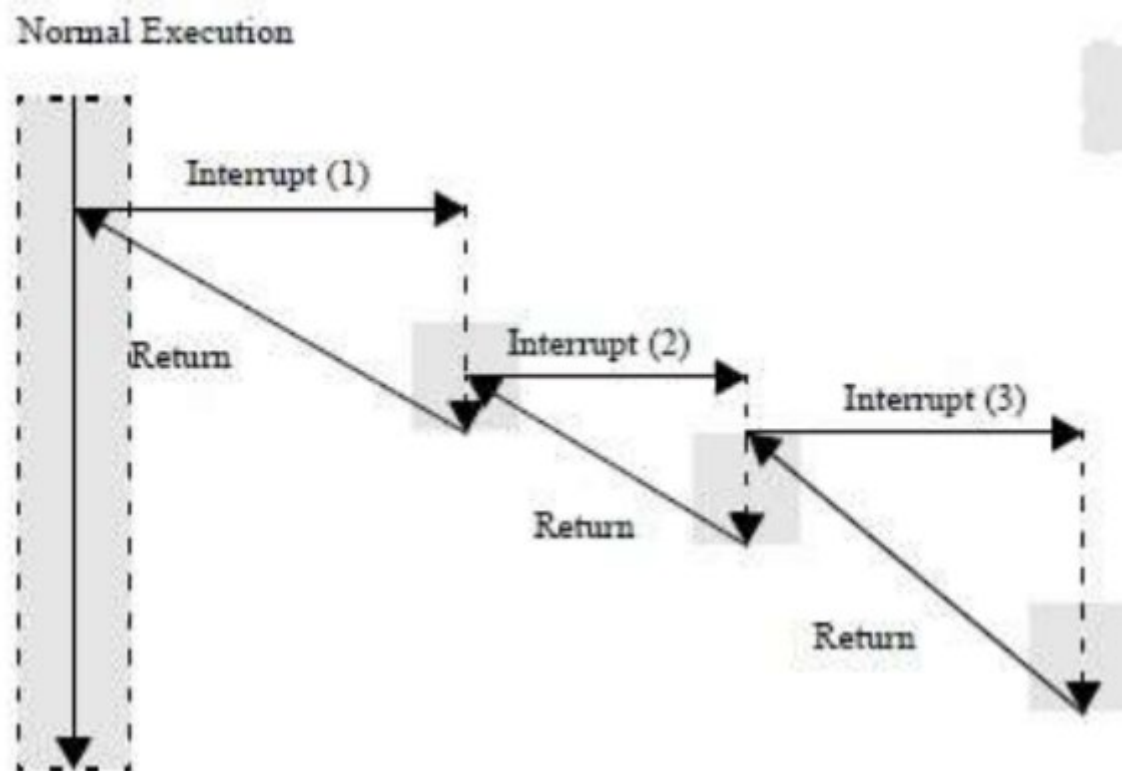
- Saved R4
- Saved LR
- Saved R0
- Saved R1
- Saved R2
- Saved R3
- Saved R12
- Saved LR
- Saved PC
- Saved xPSR

SP value reduced since registers have been pushed onto stack

Exiting an Exception Handler

Exiting an Exception Handler

1. Execute instruction triggering exception return processing
2. Select return stack, restore context from that stack
3. Resume execution of code at restored address



I. Execute Instruction for Exception Return

- No “return from interrupt” instruction
- Use regular instruction instead
 - BX LR - Branch to address in LR by loading PC with LR contents
 - POP ..., PC - Pop address from stack into PC
- ... with a special value EXC_RETURN loaded into the PC to trigger exception handling processing
 - BX LR used if EXC_RETURN is still in LR
 - If EXC_RETURN has been saved on stack, then use POP
- Also called: unstacking step

```
51: }  
⇒ 0x00000B08 BD10      POP      {r4,pc}
```

What Will Be Popped from Stack?

- R4: 0x4040_0000
- PC: 0xFFFF_FFF9

```
51: }  
→ 0x00000B08 BD10 POP {r4,pc}
```

Registers	
Register	Value
Core	
R0	0x0000008C
R1	0x40040000
R2	0xE000E280
R3	0x0000008C
R4	0x40040000
R5	0x00000ECC
R6	0x2EE00000
R7	0x00000EAB
R8	0x9C7FB76F
R9	0x20000768
R10	0x00000ECC
R11	0x00000ECC
R12	0x0FFF0AA
R13 (SP)	0x2000FD0
R14 (LR)	0x00000AE1
R15 (PC)	0x00000B08
xPSR	0x01000010
Banked	
MSP	0x2000FD0
PSP	0xBEF5DFF0
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Stack	MSP

Memory 2	
Address:	sp
0x2000FD0:	40040000 FFFFFFFF9 00000000 00000000 00000000
0x2000FE4:	0000008C 0FFF0AA 0000A0B 0000A70 21000000
0x2000FF8:	00000002 0000A0B 00001F1 00001F1 00001F1
0x200100C:	00001F1 00001F1 00001F1 00001F1 00001F1
0x2001020:	00001F1 00001F1 00001F1 00001F1 00001F1

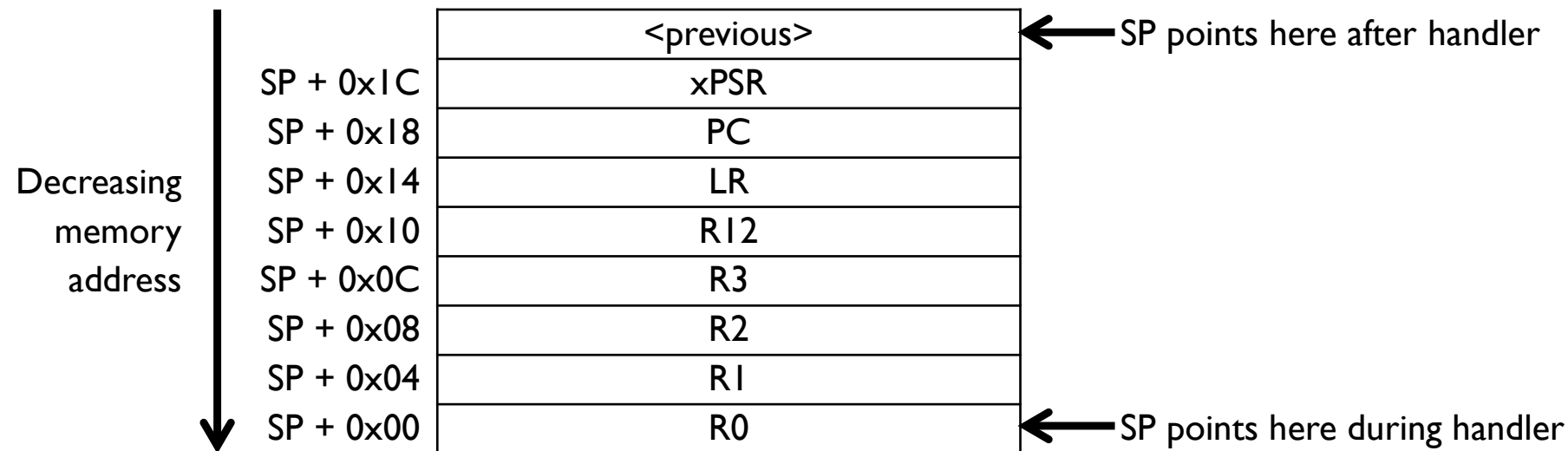
Watch 1 | Call Stack + Locals | Memory 1 | Memory 2

2. Select Stack, Restore Context

- Check EXC_RETURN (bit 2) to determine from which SP to pop the context

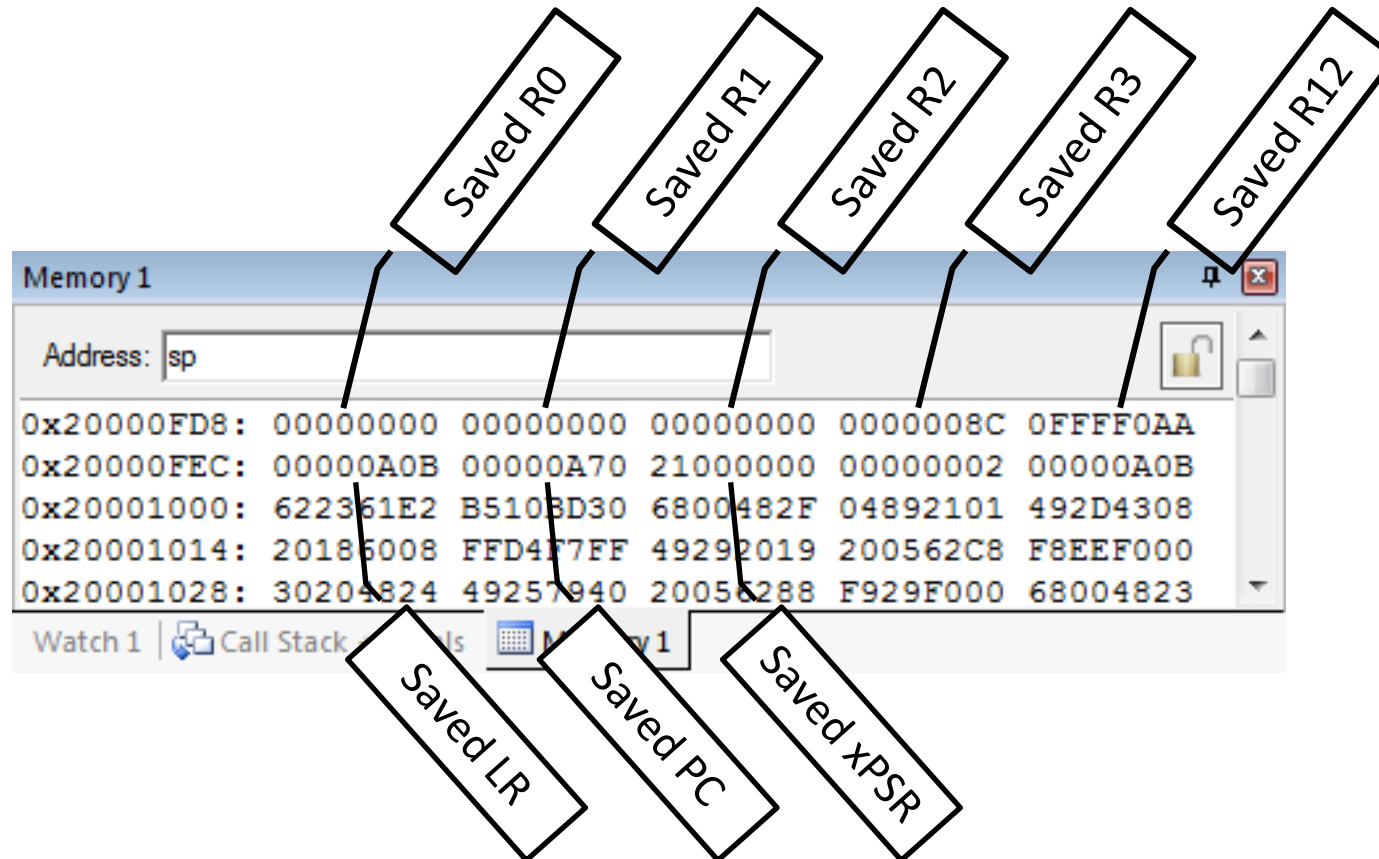
EXC_RETURN	Return Stack	Description
0xFFFF_FFF1	0 (MSP)	Return to exception handler with MSP
0xFFFF_FFF9	0 (MSP)	Return to thread with MSP (9 = 1001)
0xFFFF_FFFD	1 (PSP)	Return to thread with PSP (D = 1101)

- Pop the registers from that stack



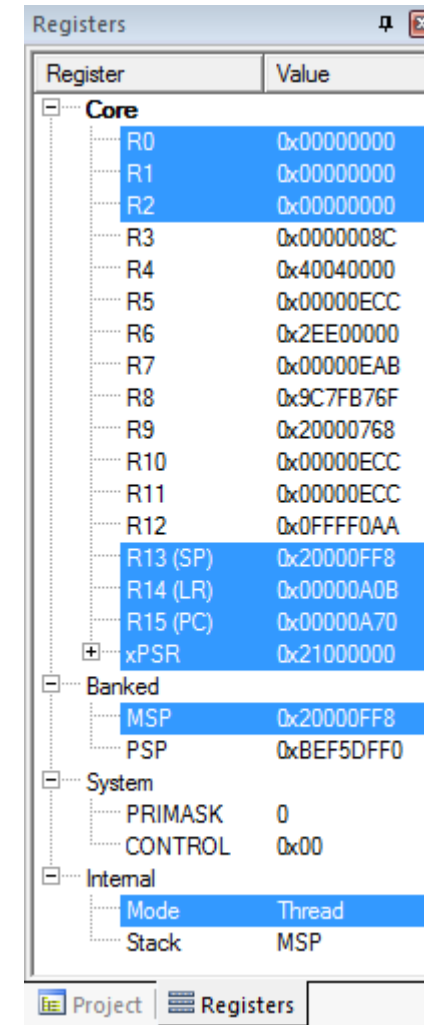
Example

- PC=0xFFFF_FFF9, so return to thread mode with main stack pointer
- Pop exception stack frame from stack back into registers



Resume Executing Previous Main Thread Code

- Exception handling registers have been restored: R0, R1, R2, R3, R12, LR, PC, xPSR
- SP is back to previous value
- Back in thread mode
- Next instruction to execute is at 0x0000_0A70



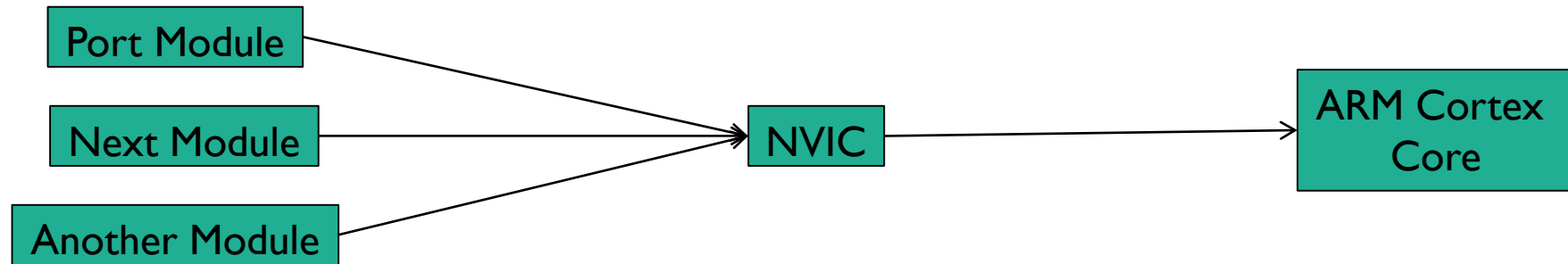
Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x0000008C
R4	0x40040000
R5	0x0000ECC
R6	0x2EE00000
R7	0x0000EAB
R8	0x9C7FB76F
R9	0x20000768
R10	0x0000ECC
R11	0x0000ECC
R12	0x0FFFF0AA
R13 (SP)	0x2000FF8
R14 (LR)	0x0000A0B
R15 (PC)	0x0000A70
xPSR	0x21000000
Banked	
MSP	0x2000FF8
PSP	0xBEF5DFF0
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Thread
Stack	MSP

Processor Core Interrupts

Microcontroller Interrupts

- Types of interrupts
 - Hardware interrupts
 - *Asynchronous*: not related to what code the processor is currently executing
 - Examples: interrupt is asserted, character is received on serial port, or ADC converter finishes conversion
 - Exceptions, Faults, software interrupts
 - *Synchronous*: are the result of specific instructions executing
 - Examples: undefined instructions, overflow occurs for a given instruction
 - We can enable and disable (*mask*) most interrupts as needed (*maskable*), others are *non-maskable*
- Interrupt service routine (ISR)
 - Subroutine which processor is *forced to execute* to respond to a *specific event*
 - After ISR completes, MCU goes back to previously executing code

Nested Vectored Interrupt Controller



- NVIC manages and prioritizes external interrupts
- Interrupts are types of exceptions
 - Exceptions 16 through 16+N (why starting from 16?)
- Modes
 - Thread Mode: entered on Reset
 - Handler Mode: entered on executing an exception
- Privilege level
- Stack pointers
 - Main Stack Pointer, MSP
 - Process Stack Pointer, PSP
- Exception states: Inactive, Pending, Active

NVIC Key Functions

- Enable and disable interrupts
- Set the preemption priority and sub-priority of a specific interrupt
- Set and clear the handling bit of a specific interrupt
- Handle up to 256 interrupts:
 - 16 System exceptions
 - 240 peripheral interrupts

NVIC Registers and State

- Enable - Allows interrupt to be recognized
 - Accessed through two registers (set bits for interrupts)
 - Set enable with NVIC_ISER, clear enable with NVIC_ICER
 - ISER = Interrupt Set Enable Register
 - ICER = Interrupt Clear Enable Register
 - CMSIS Interface: NVIC_EnableIRQ(IRQnum), NVIC_DisableIRQ(IRQnum)
- Pending - Interrupt has been requested but is not yet serviced
 - CMSIS: NVIC_SetPendingIRQ(IRQnum), NVIC_ClearPendingIRQ(IRQnum)

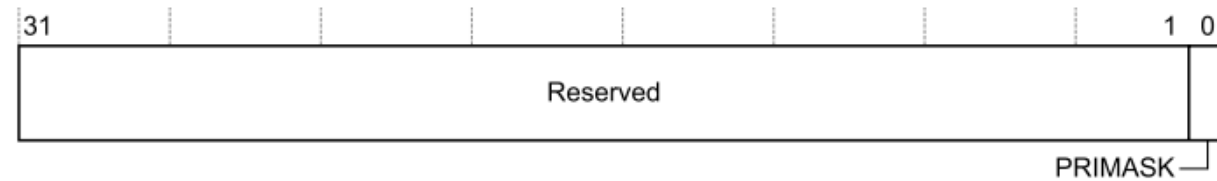
- Other registers
 - ISPR, ICPR, IABR, STIR

NVIC				Nested Vectored Interrupt Controller
▶	NVIC_ISER	RW	0xE000E100	Interrupt Set Enable Register
▶	NVIC_ICER	RW	0xE000E180	Interrupt Clear Enable Register
▶	NVIC_ISPR	RW	0xE000E200	Interrupt Set Pending Register
▶	NVIC_ICPR	RW	0xE000E280	Interrupt Clear Pending Register

Core Exception Mask Register

Priority Mask Register

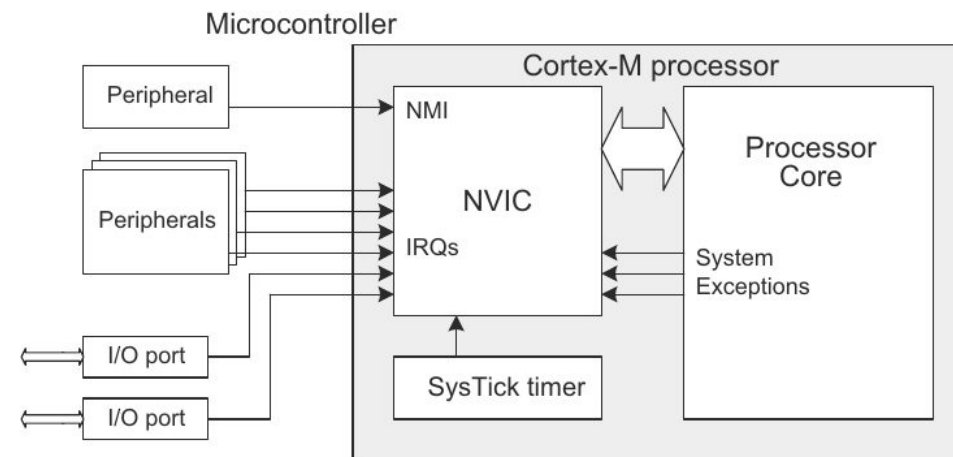
The PRIMASK register prevents activation of all exceptions with configurable priority. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:



- Similar to “Global interrupt disable” bit in other MCUs
- PRIMASK - Exception mask register (CPU core)
 - Bit 0: PM Flag
 - Set to 1 to prevent activation of all exceptions with configurable priority
 - Clear to 0 to allow activation of all exception
 - Access using CPS, MSR and MRS instructions
 - Use to prevent data race conditions with code needing atomicity
- CMSIS-CORE API
 - void __enable_irq() - clears PM flag
 - void __disable_irq() - sets PM flag
 - uint32_t __get_PRIMASK() - returns value of PRIMASK
 - void __set_PRIMASK(uint32_t x) - sets PRIMASK to x

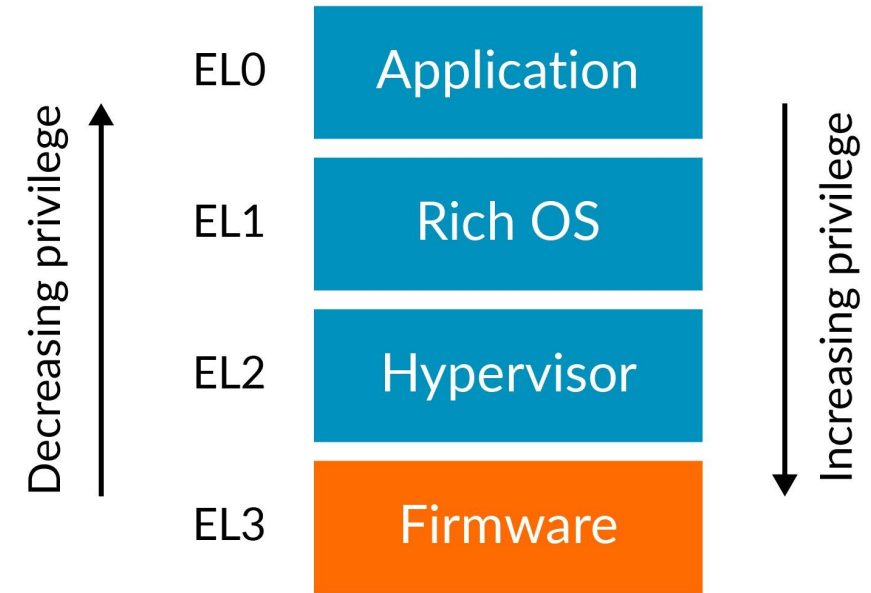
Prioritization

- Exceptions are prioritized to order the response simultaneous requests (smaller number = higher priority)
- Priorities of some exceptions are fixed
 - Reset: -3, highest priority
 - NMI: -2
 - Hard Fault: -1
- Priorities of other (peripheral) exceptions are adjustable
 - Value is stored in the interrupt priority register (IPR0-7)
 - 0x00
 - 0x40
 - 0x80
 - 0xC0



Special Cases of Prioritization

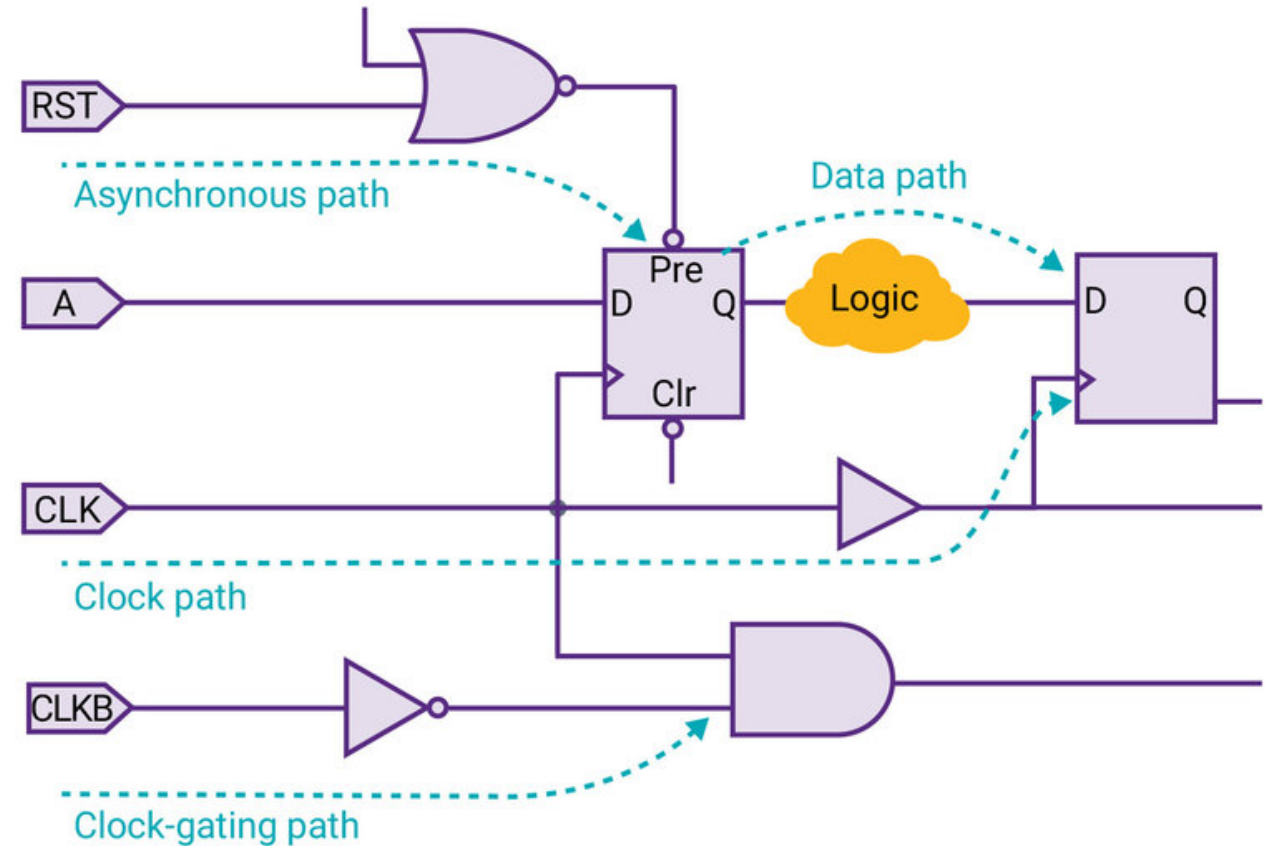
- Simultaneous exception requests?
 - Lowest exception type number is serviced first
- New exception requested while a handler is executing?
 - New priority higher than current priority?
 - New exception handler preempts current exception handler
 - New priority lower than or equal to current priority?
 - New exception held in pending state
 - Current handler continues and completes execution
 - Previous priority level restored
 - New exception handled if priority level allows
- Further reading:
 - <https://developer.arm.com/architectures/learn-the-architecture/exception-model/single-page>



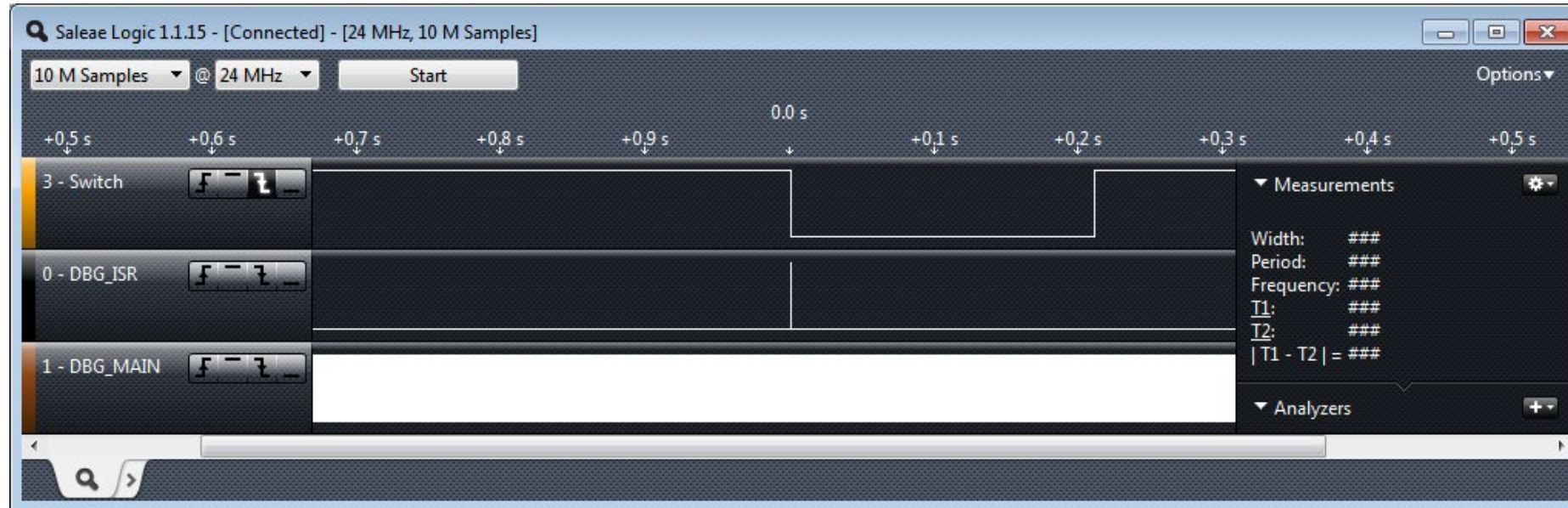
Timing Analysis

SYNOPSYS®

Types of paths considered for timing analysis



Big Picture Timing Behavior



- Switch was pressed for about 0.21 s
- ISR runs in response to switch signal's falling edge
- Main seems to be running continuously (signal toggles between 1 and 0)
 - Does it really? You can check it later on.

Interrupt Response Latency

- Latency = time delay
- Why do we care?
 - This is overhead which wastes time, and increases as the interrupt rate rises
 - This delays our response to external events, which may or may not be acceptable for the application, such as sampling an analog waveform
- How long does it take?
 - Finish executing the current instruction or abandon it
 - Push various registers on to the stack, fetch vector
 - $C_{\text{IntResponseOvhd}}$: Overhead for responding to each interrupt
 - If we have external memory with wait states, this takes longer

Maximum Interrupt Rate

- We can only handle so many interrupts per second
 - $F_{\text{Max_Int}}$: maximum interrupt frequency
 - F_{CPU} : CPU clock frequency
 - C_{ISR} : Number of cycles ISR takes to execute
 - C_{Overhead} : Number of cycles of overhead for saving state, vectoring, restoring state, etc.
 - $F_{\text{Max_Int}} = F_{\text{CPU}} / (C_{\text{ISR}} + C_{\text{Overhead}})$
 - Note that model applies only when there is one interrupt in the system
- When processor is responding to interrupts, it isn't executing our other code
 - U_{Int} : Utilization (fraction of processor time) consumed by interrupt processing
 - $U_{\text{Int}} = 100\% * F_{\text{Int}} * (C_{\text{ISR}} + C_{\text{Overhead}}) / F_{\text{CPU}}$
 - CPU looks like it's running the other code with CPU clock speed of $(1 - U_{\text{Int}}) * F_{\text{CPU}}$

Program Design with Interrupts

Program Design with Interrupts

- How much work to do in ISR?
- Should ISRs re-enable interrupts?
- How to communicate between ISR and other threads?
 - Data buffering
 - Data integrity and race conditions

How Much Work Is Done in ISR?

- Trade-off: Faster response for ISR code will delay completion of other code
- In system with multiple ISRs with short deadlines, perform critical work in ISR and buffer partial results for later processing

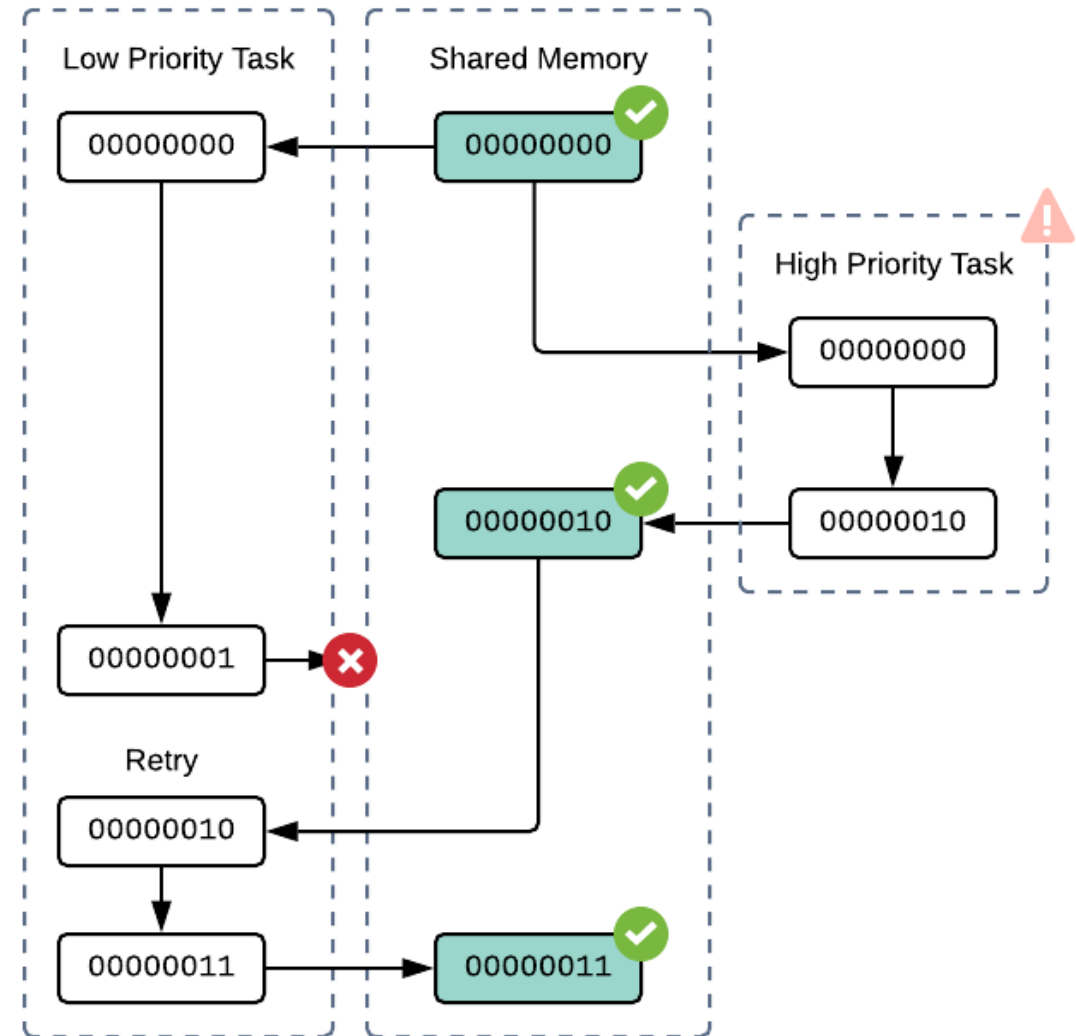
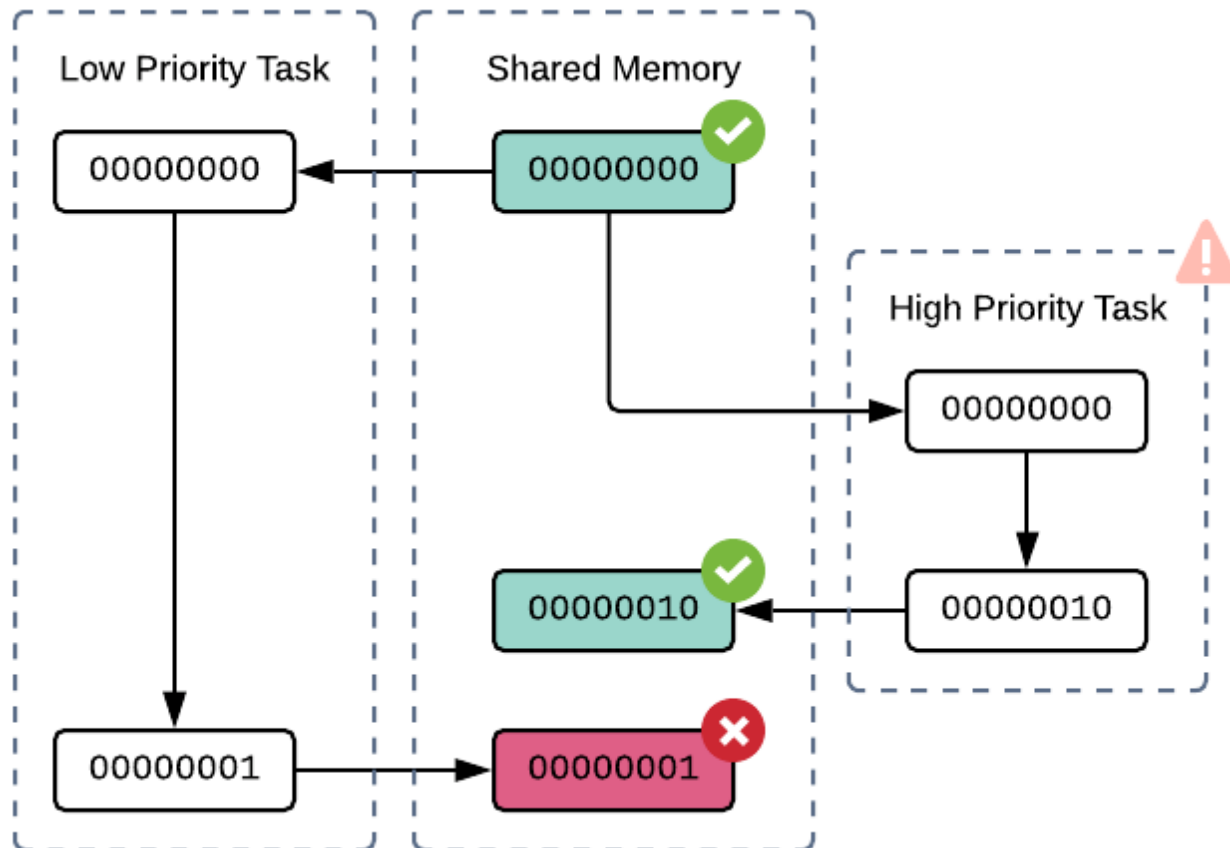
Sharing Data Safely between ISRs and other Threads

Overview

- Volatile data – can be updated outside of the program's immediate control
- Non-atomic shared data – can be interrupted partway (part of the way) through read or write, is vulnerable to race conditions

The Problem

- Data will be corrupted



Volatile Data

- Compilers assume that variables in memory do not change spontaneously, and optimize based on that belief
 - *Don't reload a variable from memory if current function hasn't changed it*
 - Read variable from memory into register (faster access)
 - Write back to memory at end of the procedure, or before a procedure call, or when compiler runs out of free registers
- This optimization can fail
 - Example: reading from input port, polling for key press
 - while (SW_0) ; will read from SW_0 once and reuse that value
 - Will generate an infinite loop triggered by SW_0 being true
- Variables for which it fails
 - Memory-mapped peripheral register – register changes on its own
 - Global variables modified by an ISR – ISR changes the variable
 - Global variables in a multithreaded application – another thread or ISR changes the variable

The Volatile (Always Changing) Directive

- Need to tell compiler which variables may change outside of its control
 - Use volatile keyword to force compiler to reload these vars from memory for each use
`volatile unsigned int num_ints;`
 - Pointer to a volatile int
`volatile int * var; // or`
`int volatile * var;`
- Now each C source read of a variable (e.g. status register) will result in an assembly language LDR instruction
- Good explanation in Nigel Jones' "Volatile," *Embedded Systems Programming* July 2001

Non-Atomic Shared Data

- Want to keep track of current time and date
- Use 1 Hz interrupt from timer
- System
 - current_time structure tracks time and days since some reference event
 - current_time's fields are updated by periodic 1 Hz timer ISR

```
void GetDateTime(DateTimeType * DT){  
    DT->day = current_time.day;  
    DT->hour = current_time.hour;  
    DT->minute = current_time.minute;  
    DT->second = current_time.second;  
}
```

```
void DateTimeISR(void){  
    current_time.second++;  
    if (current_time.second > 59){  
        current_time.second = 0;  
        current_time.minute++;  
        if (current_time.minute > 59) {  
            current_time.minute = 0;  
            current_time.hour++;  
            if (current_time.hour > 23) {  
                current_time.hour = 0;  
                current_time.day++;  
                ... etc.  
            }  
        }  
    }  
}
```

Example: Checking the Time

- Problem
 - An interrupt at the wrong time will lead to half-updated data in DT
- Failure Case
 - `current_time` is {10, 23, 59, 59} (10th day, 23:59:59)
 - Task code calls `GetDateTime()`, which starts copying the `current_time` fields to DT: day = 10, hour = 23
 - A timer interrupt occurs, which updates `current_time` to {11, 0, 0, 0}
 - `GetDateTime()` resumes executing, copying the remaining `current_time` fields to DT: minute = 0, second = 0
 - DT now has a time stamp of {10, 23, 0, 0}.
 - *The system thinks time just jumped backwards one hour!*
- Fundamental problem – “race condition”
 - Preemption enables ISR to interrupt other code and possibly overwrite data
 - Must ensure *atomic (indivisible)* access to the object
 - Native atomic object size depends on processor’s instruction set and word size.
 - Is 32 bits for ARM

Examining the Problem More Closely

- Must protect any data object which both:
 - Requires multiple instructions to read or write (non-atomic access), and
 - Is potentially written by an ISR
- How many tasks/ISRs can write to the data object?
 - One? Then we have one-way communication
 - Must ensure *the data isn't overwritten* partway through being read
 - Writer and reader don't interrupt each other
 - More than one?
 - Must ensure *the data isn't overwritten* partway through being read
 - Writer and reader don't interrupt each other
 - Must ensure *the data isn't overwritten* partway through being written
 - Writers don't interrupt each other

Definitions

- Race condition: Anomalous behavior due to unexpected critical dependence on the relative timing of events. Result of example code depends on the relative timing of the read and write operations.
- Critical section: A section of code which creates a possible race condition. The code section can only be executed by one process at a time. Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use.
 - This is an OS course material. You can go deeper into this topic later on.

Solution: Briefly Disable Preemption

- Prevent preemption within critical section
- If an ISR can *write* to the shared data object, need to *disable interrupts*
 - save current interrupt masking state in m
 - disable interrupts
- Restore *previous state* afterwards (interrupts may have already been disabled for another reason)
- Use CMSIS-CORE to save, control and restore interrupt masking state
- Avoid if possible
 - Disabling interrupts delays response to all other processing requests
 - Make this time as short as possible (e.g. a few instructions)

```
void GetDateTime(DateTimeType * DT) {  
    uint32_t m;  
  
    m = __get_PRIMASK();  
    __disable_irq();  
  
    DT->day = current_time.day;  
    DT->hour = current_time.hour;  
    DT->minute = current_time.minute;  
    DT->second = current_time.second;  
    __set_PRIMASK(m);  
}
```


Summary for Sharing Data

- In thread/ISR diagram, identify shared data
- Determine which shared data is too large to be handled atomically by default
 - This needs to be protected from preemption (e.g. disable interrupt(s), use an RTOS synchronization mechanism)
- Declare (and initialize) shared variables as volatile in main file (or globals.c)
 - `volatile int my_shared_var=0;`
- Update extern.h to make these variables available to functions in other files
 - `volatile int my_shared_var;`
 - `#include "extern.h"` in every file which uses these shared variables
- When using long (non-atomic) shared data, save, disable and restore interrupt masking status
 - CMSIS-CORE interface: `__disable_irq()`, `__get_PRIMASK()`, `__set_PRIMASK()`

Summary

- Entering an exception handler
 - Push context to the current stack
 - Examine current vector table
- Exiting an exception handler
 - Pop from the stack
- Processor core interrupts
 - NVIC registers and states
- Timing analysis
 - Interrupt response latency
- Program design with interrupts
- Data sharing safely

<https://developer.arm.com/architectures/system-architectures>

