

EE3220 System-on-Chip Design

Lecture Note 8

Exceptions and Interrupts

Review and Preview

- Lecture 1: Course Overview
- Lecture 2: Intro to Modern Microprocessor (CPU Architecture)
- Lecture 3: Embedded System Software (Software Architecture)
- Lecture 4: ARM Cortex-M4 (ARM Hardware)
- Lecture 5: From C to ARM Assembly (C Language, ARM Firmware)
- Lecture 6: Team Based Learning (ARM Firmware)
- Lecture 7: AAPCS (ARM Firmware)
- Lecture 8: Exceptions & Interrupts (ARM Firmware)
- Lecture 9: HDL / FPGA (Verilog HDL, Hardware Design)

Outline

- Background on Exceptions & Interrupts
- Entering Interrupt Handler
- Exiting Interrupt Handler
- ARM Core Interrupt Design and APIs
- Interrupt Timing Analysis
- Interrupt Service Routine (ISR) Design Trade-offs
- Robust Data Sharing among ISRs and Other Threads

Background on Interrupts & Exceptions

Why Exceptions & Interrupts critical in Firmware Design?

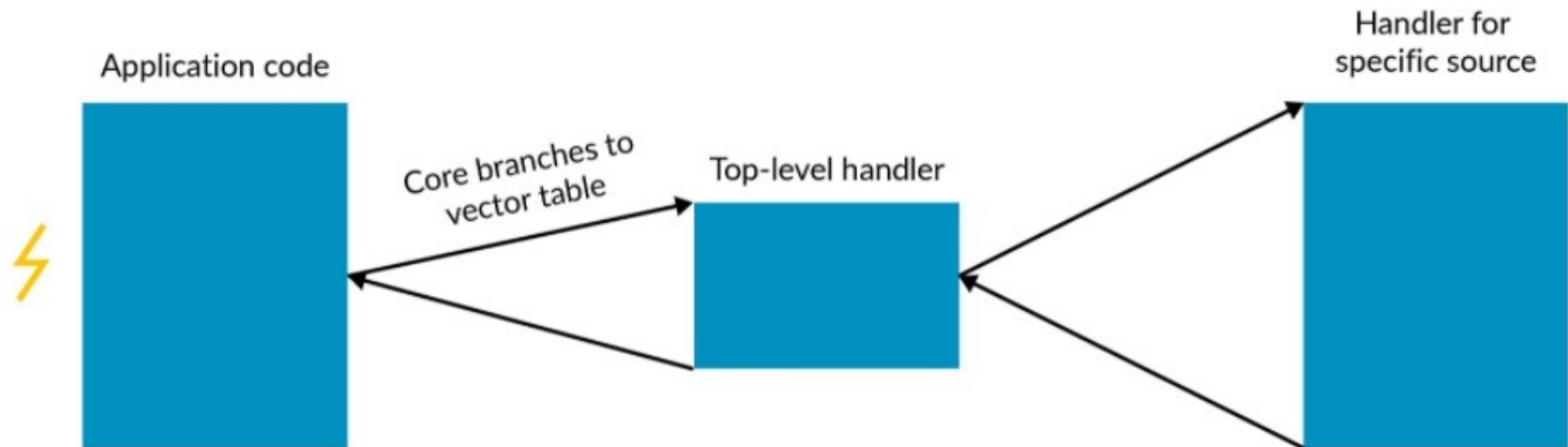
- Exception and interrupt handling enables **RTOS**, representing a **KEY DIFFERENCE** between:
 - **EMBEDDED SOC FIRMWARE** and
 - **GENERAL-PURPOSE SOFTWARE.**
- Other key differences in embedded firmware design:
 - Hardware Dependency
 - Resource Constraints
 - Real-Time Requirements
 - Debugging Challenges

Definitions (Exceptions & Interrupts)

- Exceptions versus Interrupts
 - Exceptions and Interrupts share **SIMILAR S/W BEHAVIORS**.
 - Both disrupt Normal Program Flow.
 - However, they are used to serve **DIFFERENT PURPOSES**.
- **EXCEPTIONS**
 - Exceptions are usually caused by **PROGRAM ERRORS**.
 - Examples: Div-by-zero, Invalid Memory Address.
- **INTERRUPTS**
 - Interrupts are usually caused by **EXTERNAL EVENTS**.
 - Examples: Keyboard Press, Timer Interrupt, Network Packet Arrival.

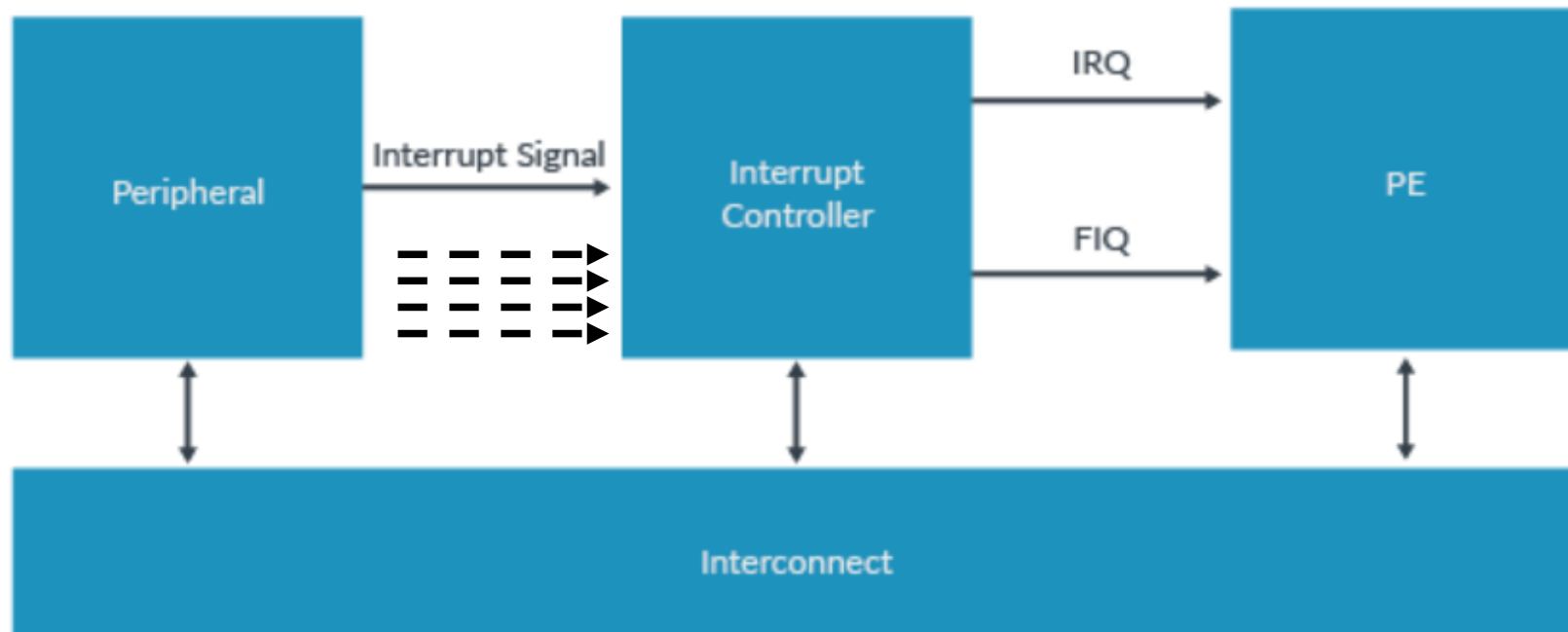
Handling Exceptions

- 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.



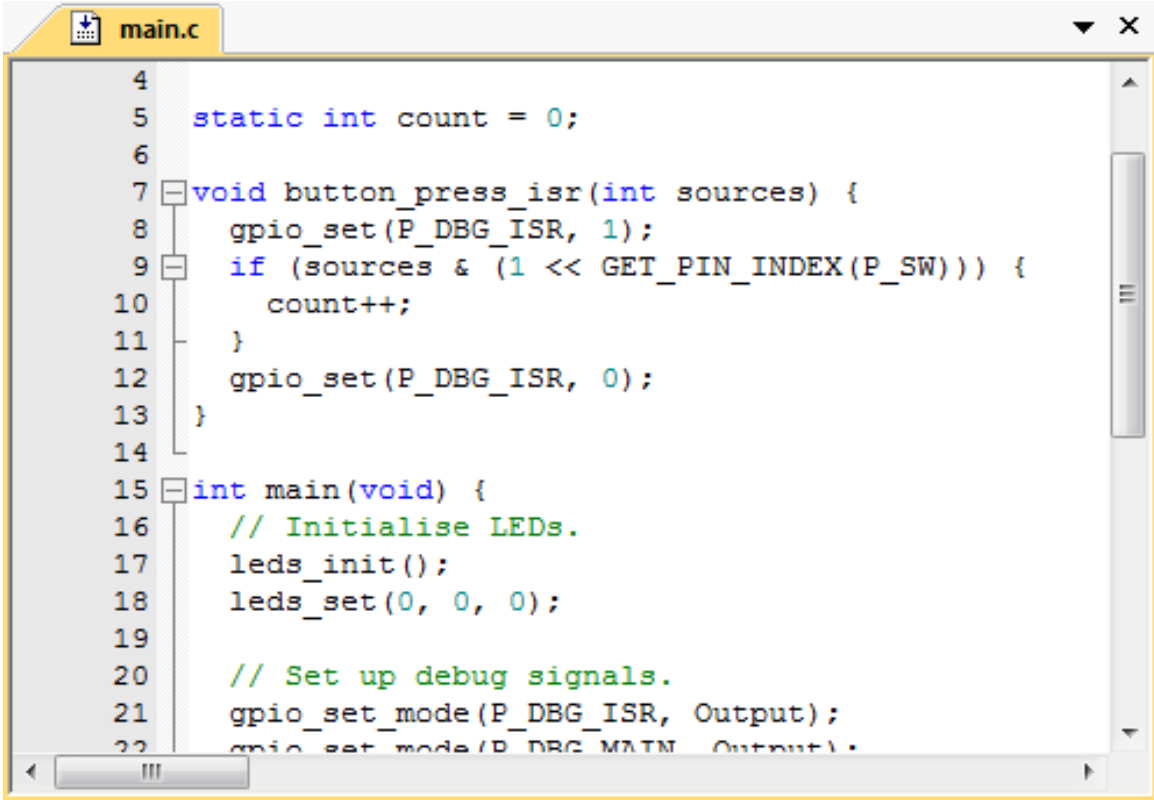
Interrupt Signals to the Interrupt Controller

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



Example Exception/Interrupt Handler

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

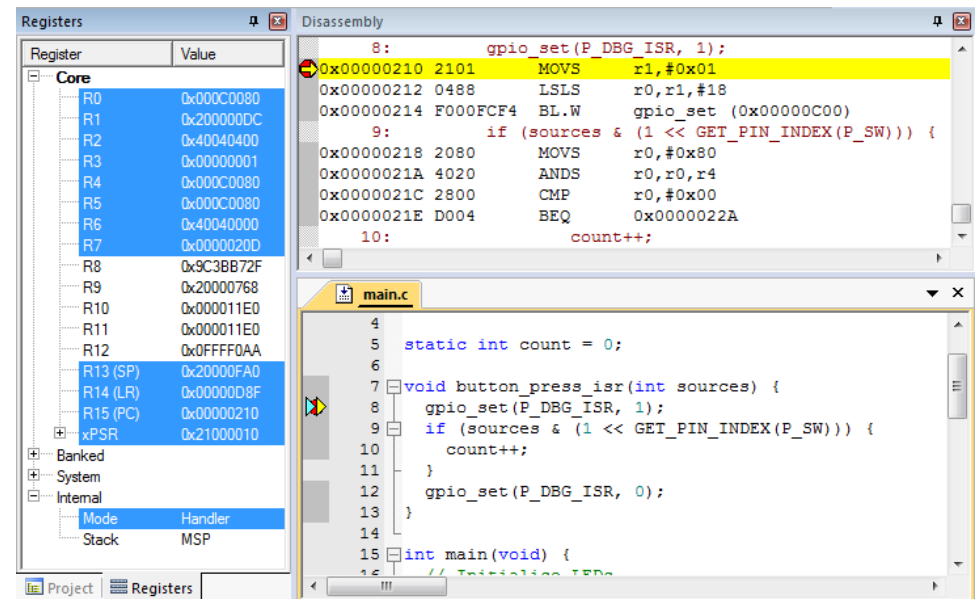


```
main.c
4
5 static int count = 0;
6
7 void button_press_isr(int sources) {
8     gpio_set(P_DBG_ISR, 1);
9     if (sources & (1 << GET_PIN_INDEX(P_SW))) {
10         count++;
11     }
12     gpio_set(P_DBG_ISR, 0);
13 }
14
15 int main(void) {
16     // Initialise LEDs.
17     leds_init();
18     leds_set(0, 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/interrupt handler for an interrupt request.

Use Debugger for Detailed Processor View

- Ability to examine registers, stack, source code, disassembled code (object code)
- Compiler may generate additional code for function entry.
- Place your breakpoint at the handler's **FUNCTION DECLARATION LINE** in source code, not at the first line of function code



Normal Flow of Execution vs. Processor Exceptions

- **NORMAL FLOW:** Program counter (PC) increases sequentially in the address space, with branches to nearby labels or branch and links to subroutines.
- **PROCESSOR EXCEPTIONS:** Processor exceptions occur when this normal flow of execution is diverted, enabling 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
- In case of exception, execution branches to an address specified in the **VECTOR TABLE**.

Table of Addresses
representing the Entry
Point of Each Interrupt.
(In Code Segment)

Entering ARM Cortex-M4 Exception/Interrupt Handler (Hardware + Software)

CPU's Hardwired Exception Processing

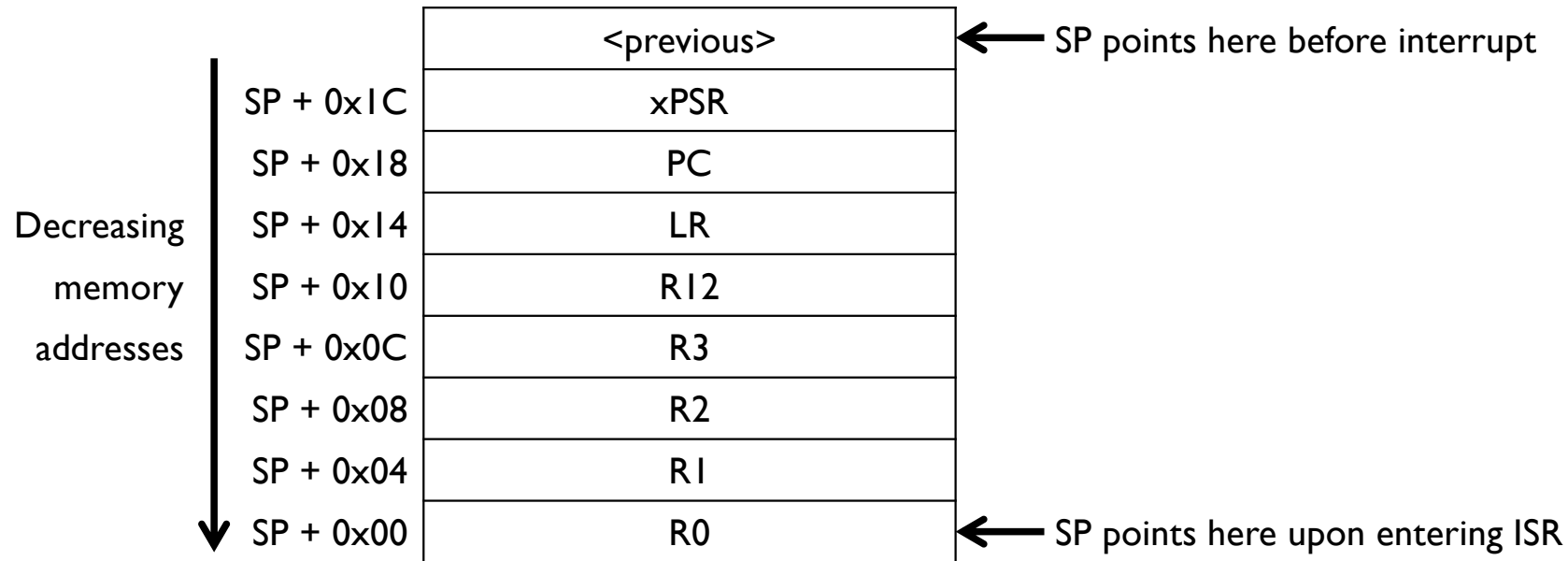
1. Finish current instruction (except for lengthy instructions)
2. Push context (**8 x 32-BIT WORDS**) onto the **CURRENT STACK** (MSP or PSP)
 - xPSR, RETURN PC, LR (R14), R12, R3, R2, R1, R0
3. Switch to **HANDLER / PRIVILEGED MODE**, use **MSP** (main stack pointer)
4. Load IPSR with the exception number
5. Load PC with the address of exception handler (from the vector table)
6. Load LR with an appropriate EXC_RETURN code
7. Start executing the exception handler code

*Note: **16 CLOCK CYCLES** are usually needed between an **EXCEPTION REQUEST** and the **FIRST INSTRUCTION EXECUTION** of the interrupt handler.*

Step 1 of 7: Finish Current Instruction (H/W)

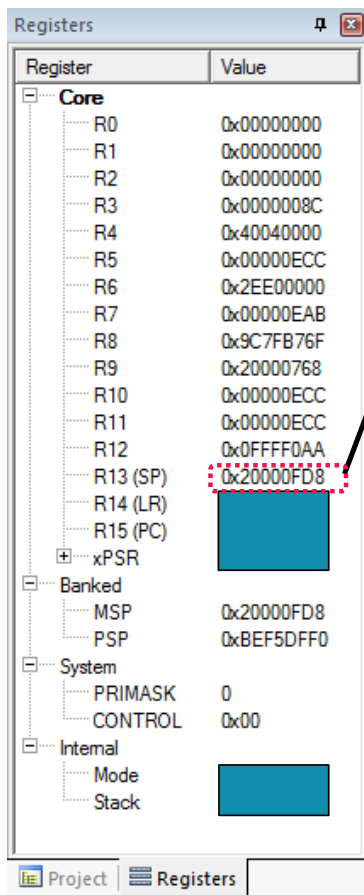
- Most instructions are short and finish quickly
- However, some instructions may take many cycles to execute, which may delay interrupt response significantly
 - Load Multiple (**LDM**), Store Multiple (**STM**), **PUSH**, **POP**, **MULS**
 - **MULS** may take 32 cycles for some CPU core implementations
- If one of these is executing when the interrupt is requested, the processor:
 1. **ABANDONS** the instruction
 2. Responds to the interrupt
 3. Executes the ISR (interrupt service routine)
 4. Returns from the interrupt
 5. **RESTARTS** the abandoned instruction

Step 2 of 7: Push Context onto Current Stack (H/W)



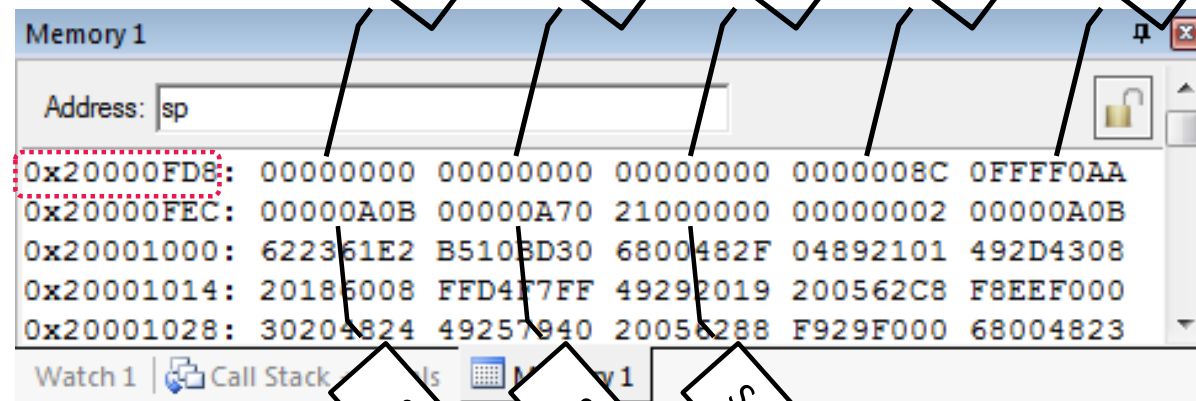
- Two Stack Pointers: Main (MSP), process (PSP)
- Which stack is active depends on the operating mode, CONTROL register bit 1
- Stack grows toward smaller addresses
- **UNLIKE C FUNCTION CALLS**, this is done in **HARDWARE**.

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	0x0FFFF0AA
R13 (SP)	0x20000FD8
R14 (LR)	
R15 (PC)	
xPSR	
Banked	
MSP	0x20000FD8
PSP	0xBEF5DFF0
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	
Stack	

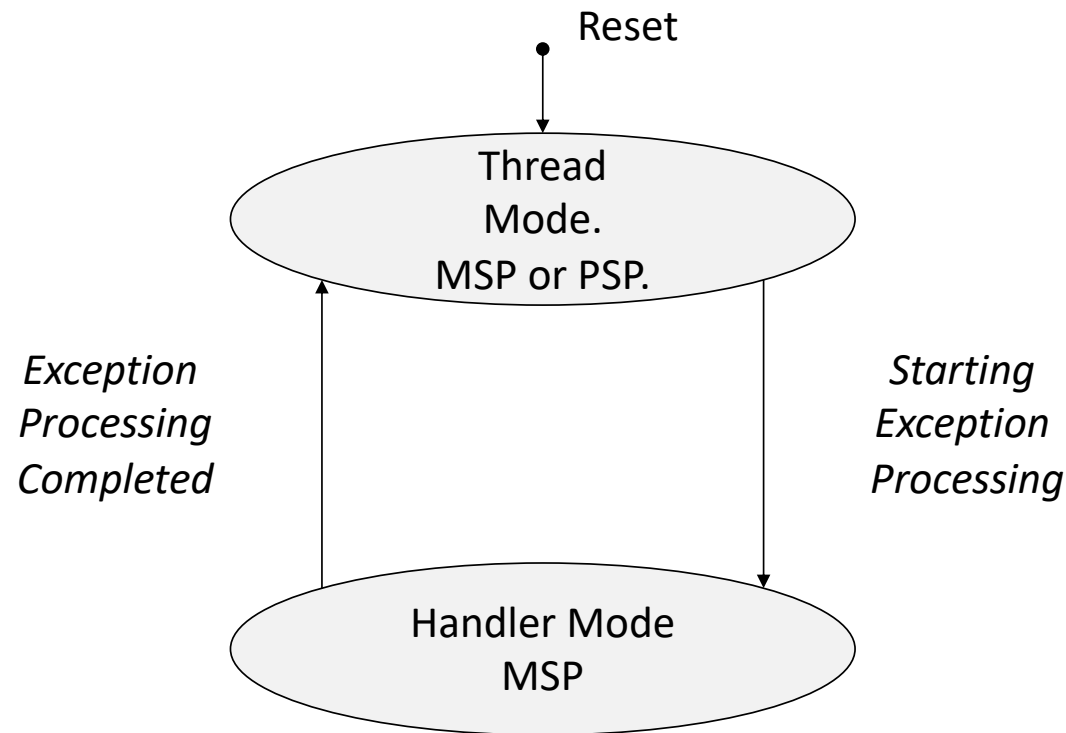
SP value is reduced since registers have been pushed onto stack



Address	Value
0x20000FD8:	00000000 00000000 00000000 0000008C 0FFFF0AA
0x20000FEC:	00000A0B 00000A70 21000000 00000002 00000A0B
0x20001000:	622361E2 B5108D30 6800482F 04892101 492D4308
0x20001014:	20186008 FFD417FF 49292019 200562C8 F8EEF000
0x20001028:	30204824 49257940 20056288 F929F000 68004823

Step 3 of 7: Switch to Handler/Privileged Mode (H/W)

- **THREAD MODE** may use MSP or PSP but **HANDLER MODE** always uses MSP.
- The exception handler executes in **HANDLER + PRIVILEGED** mode using MSP.



ARM Processor Modes (Cortex-M)

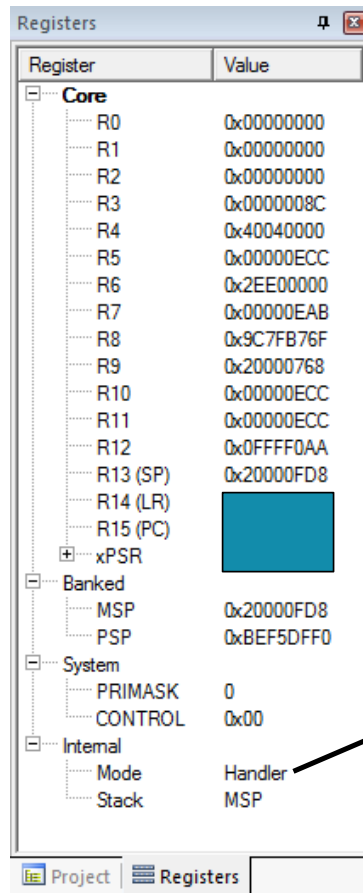
- The Cortex-M processor supports:
 - Two **OPERATION** Modes, **THREAD** Mode and **HANDLER** Mode
 - Two **EXECUTION** Modes, **USER** Mode and **PRIVILEGED** Mode.
- Normal programs run in Thread Mode, and Thread Mode is entered on Reset.
 - Thread Mode: Code can be executed in either User Mode or Privileged Mode.
 - Handler Mode: Code is executed in Privileged Mode.
 - 3 Scenarios: **THREAD+USER**, **THREAD+PRIVILEGED**, **HANDLER+PRIVILEGED**
- Privileged Mode has full access rights, but User Mode has only limited rights:
 - Restrictions on instruction use such as which fields can be used in MSR instructions
 - Restrictions on the use of certain coprocessor registers, some memory locations and peripherals (based on system design).
- Cortex-M4 special CONTROL register bit 0 specifies Privileged Mode (0) or User Mode (1) , bit 1 specifies MSP (0) or PSP (1) stack pointer.

ARM Processor Modes (Cortex-A, Cortex-R, Cortex-M)

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

Cortex-A, Cortex-R (CPSR)	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
	THREAD MODE		
Cortex-M	HANDLER MODE		

Handler Operation Mode & Privileged Execution Mode

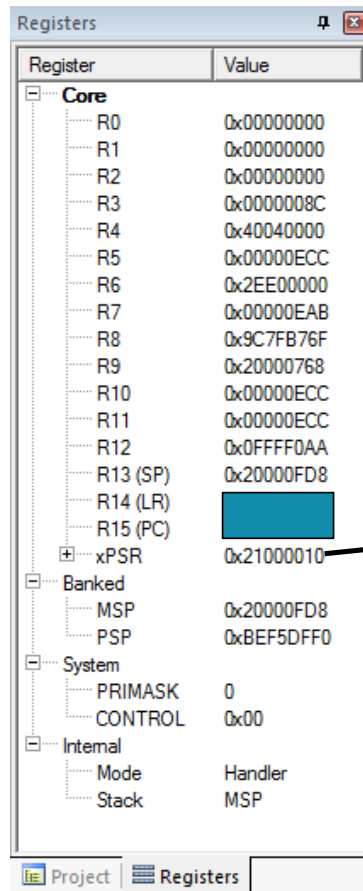


The screenshot shows a 'Registers' window with a tree view on the left and a table of register values on the right. An arrow points from the 'Mode' field under the 'Internal' category to a text box on the right.

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)	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. It was already using MSP in this example.

Step 4 of 7: Update IPSR with Exception Number (H/W)



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)	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.

Step 5 of 7: Load PC with Exception Handler Address (H/W)

Memory Address	Value
0x0000_0000	Initial Stack Pointer
0x0000_0004	Reset
0x0000_0008	NMI_IRQHandler
...	
	IRQ0_Handler
	IRQ1_Handler
...	
Reset:	
...	
NMI_IRQHandler:	
...	
IRQ0_Handler:	
...	
IRQ1_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			
9			Reserved
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

Examining Vector Table With Debugger

IRQ #	Vector	Offset	Example
	Initial SP	0x00	0x20001000
	Reset	0x04	0x08000009
-14	NMI	0x08	0x0800000D
-13	HardFault	0x0C	0x08000011
	Reserved	0x10	
-5	SVCall	0x2C	0x08000019
	Reserved		
-2	PendSV	0x38	0x0800001D
-1	SysTick	0x3C	0x08000021
0	IRQ0	0x40	...
1	IRQ1	0x44	...
2	IRQ2	0x48	...
	.		
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	0x0FFF0AA
R13 (SP)	0x20000FD8
R14 (LR)	0xFFFFFFFF
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

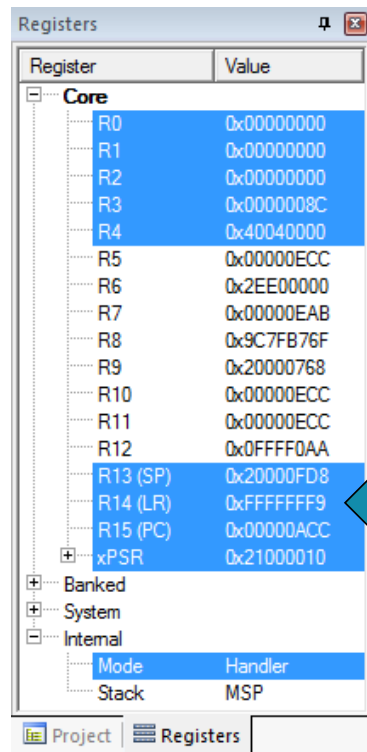
We can now see that PC has been loaded with the handler address.

Step 6 of 7: Load LR With EXC_RETURN Code (H/W)

EXC_RETURN	Return Mode (bit 3)	Return Stack (bit 2)	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

- **EXCEPTION RETURN CODE** value generated by CPU to specify 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 specify the exception return mechanism

Updated LR With EXC_RETURN Code

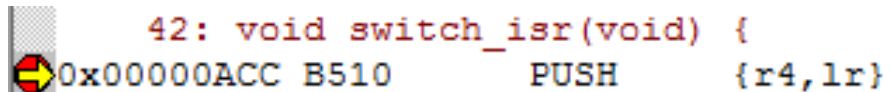


Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x0000008C
R4	0x40040000
R5	0x00000ECC
R6	0x2EE00000
R7	0x00000EAB
R8	0x9C7FB76F
R9	0x20000768
R10	0x00000ECC
R11	0x00000ECC
R12	0x0FFF0AA
R13 (SP)	0x2000FD8
R14 (LR)	0xFFFFFFFF9
R15 (PC)	0x0000ACC
xPSR	0x2100010
Banked	
System	
Internal	
Mode	Handler
Stack	MSP

Now the LR is updated with the exception return code.

Step 7 of 7: Start Exception Handler (S/W)

- Exception handler starts running, unless preempted by a higher-priority exception
- Exception handler (**SOFTWARE**) may save additional registers on stack
- Example: if handler may call a subroutine, **LR** and **R4** must be saved



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

1. Our handler code starts running.
2. The first thing this code does is to save some additional registers.

After Handler Saves 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)	0x2000FD0
R14 (LR)	0xFFFFF9
R15 (PC)	0x0000ACE
xPSR	0x21000010
Banked	
MSP	0x2000FD0
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
0x2000FD0:	40040000 FFFFFFFF 00000000 00000000 00000000
0x2000FE4:	0000008C 0FFFFFFA 00000A0B 00000A70 21000000
0x2000FF8:	00000002 00000A0B 000001F1 000001F1 000001F1
0x200100C:	000001F1 000001F1 000001F1 000001F1 000001F1
0x2001020:	000001F1 000001F1 000001F1 000001F1 000001F1

SP value reduced since registers have been pushed onto stack

Saved R4

Saved LR

Saved R0

Saved R1

Saved R2

Saved R3

Saved R12

Saved LR

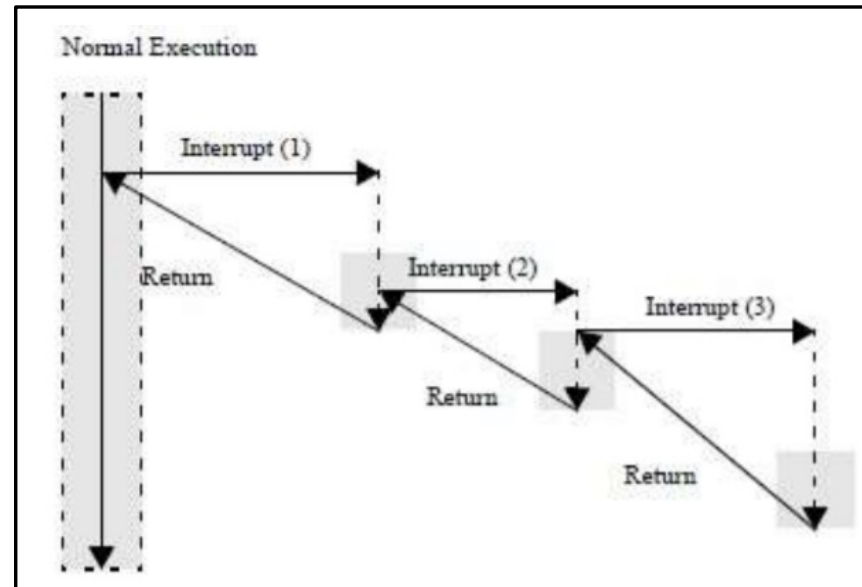
Saved PC

Saved xPSR

Exiting ARM Cortex-M4 Exception/Interrupt Handler (Software + Hardware)

Exiting an Exception Handler

1. Execute instruction triggering **EXCEPTION RETURN PROCESSING**
2. Select correct **RETURN STACK**
3. Restore context from **RETURN STACK**
4. Resume executing previous code (before exception)



Step 1 of 4: Execute Instruction for Exception Return (S/W)

- No special “return from interrupt” instruction
- Use “normal function return” in S/W
 1. “BX LR”: Branch to address in LR by loading PC with LR contents
 2. “POP ..., PC”: Pop address from stack into PC
- ... with special **EXC_RETURN** value loaded to PC to trigger exception handling processing
 1. “BX LR”: If EXC_RETURN is still in LR
 2. “POP ..., PC”: If EXC_RETURN has been saved on stack
- Also Known As: **UNSTACKING STEP**



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

Step 1 of 4: Triggering Exception Return Processing (H/W)

- 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)	0x20000FD0
R14 (LR)	0x00000AE1
R15 (PC)	0x00000B08
xPSR	0x01000010
Banked	
MSP	0x20000FD0
PSP	0xBEF5DFF0
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Stack	MSP

Memory 2	
Address:	sp
0x20000FD0:	40040000 FFFFFFFF9 00000000 00000000 00000000
0x20000FE4:	0000008C 0FFFFFFAA 00000A0B 00000A70 21000000
0x20000FF8:	00000002 00000A0B 000001F1 000001F1 000001F1
0x2000100C:	000001F1 000001F1 000001F1 000001F1 000001F1
0x20001020:	000001F1 000001F1 000001F1 000001F1 000001F1

Saved R4

Saved LR

Saved R0

Saved R1

Saved R2

Saved R3

Saved R12

Saved LR

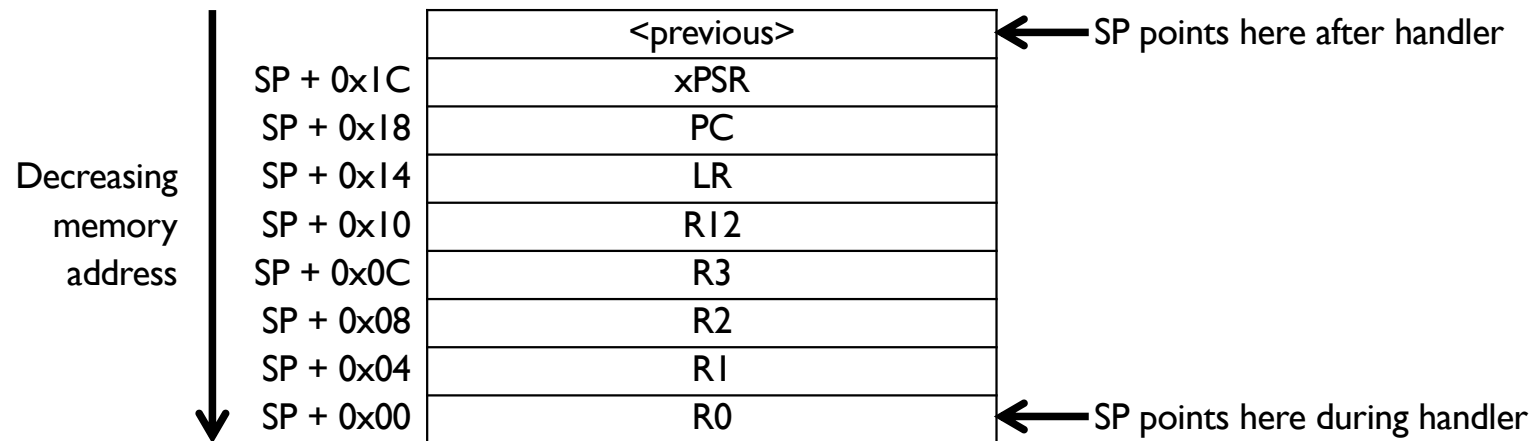
Saved PC

Saved xPSR

Step 2 of 4: Select Correct Return Stack (H/W)

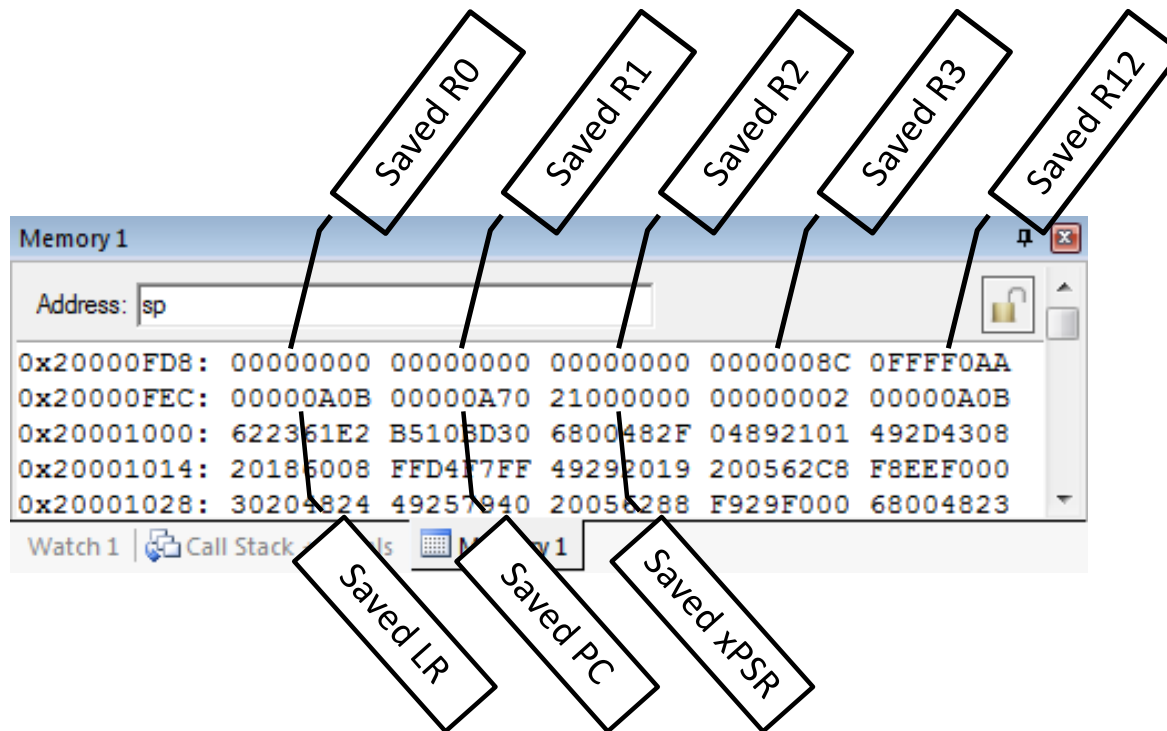
EXC_RETURN	Return Stack	Description
0xFFFF_FFF1	0 (MSP)	Return to exception handler with MSP
0xFFFF_FFF9	0 (MSP)	Return to thread with MSP (9 = 1001)
0xFFFF_FFFD	1 (PSP)	Return to thread with PSP (D = 1101)

- Check **EXC_RETURN (bit3)** to determine **HANDLER** or **THREAD** mode
- Check **EXC_RETURN (bit2)** to determine from **WHICH STACK** to pop the context
 - Pop the saved registers from the corresponding stack



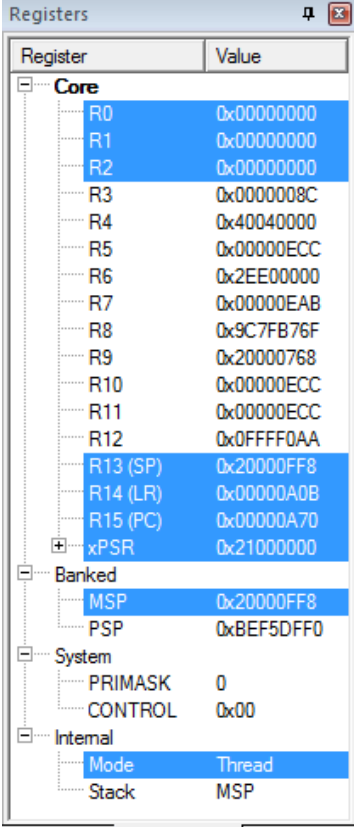
Step 3 of 4: Restoring Stack Frame (H/W)

- $PC=0xFFFF_FFF9$, so return to **THREAD MODE** with **MAIN STACK POINTER**
- Pop exception stack frame from stack back into registers



Step 4 of 4: Resume Executing Previous Code (S/W)

- Registers have been restored: R0, R1, R2, R3, R12, LR, PC, xPSR
- SP is back to previous value
- Back in the previous Thread mode
- PC=0x0000_0A70
- Next instruction to execute is at 0x0000_0A70



The screenshot shows a 'Registers' window with a tree view on the left and a table of values on the right. The tree view includes 'Core', 'Banked', 'System', and 'Internal' categories. The table lists registers and their values, with some registers highlighted in blue.

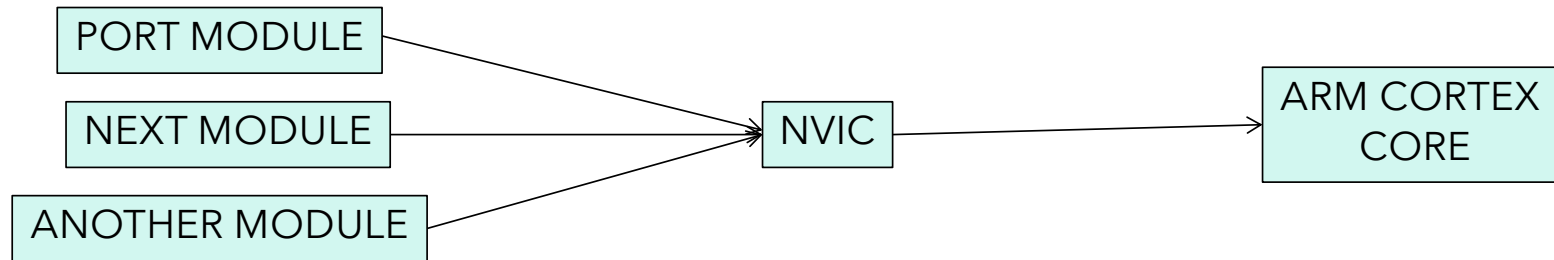
Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x0000008C
R4	0x40040000
R5	0x00000ECC
R6	0x2EE00000
R7	0x00000EAB
R8	0x9C7FB76F
R9	0x20000768
R10	0x00000ECC
R11	0x00000ECC
R12	0x0FFF0AA
R13 (SP)	0x2000FF8
R14 (LR)	0x00000A0B
R15 (PC)	0x00000A70
xPSR	0x21000000
Banked	
MSP	0x2000FF8
PSP	0xBEF5DFF0
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Thread
Stack	MSP

ARM Core Interrupt Design & APIs

Types of Interrupts

- Types of Interrupts / Exceptions
 - 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 the ISR completes, MCU goes back to the previously executing code

Nested Vectored Interrupt Controller (NVIC)



- 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 Exception
- Privilege level: User Mode / Privileged Mode
- Stack pointers: Main Stack Pointer (MSP) and Process Stack Pointer (PSP)
- Exception states: **INACTIVE**, **PENDING**, **ACTIVE**

ARM 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**

ARM NVIC Registers and CMSIS APIs

- Enable - Allows interrupt to be serviced: NVIC_ISER (set), NVIC_ICER (clear)
 - CMSIS (ARM Cortex Microcontroller Software Interface Standard) Interface:
 - `NVIC_EnableIRQ(IRQnum)`, `NVIC_DisableIRQ(IRQnum)`
- Pending - Interrupt requested but not yet serviced: NVIC_ISPR (set), NVIC_ICPR (clear)
 - CMSIS Interface:
 - `NVIC_SetPendingIRQ(IRQnum)`, `NVIC_ClearPendingIRQ(IRQnum)`
- Other related registers
 - IABR (active bit register)
 - STIR (s/w trigger interrupt)

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 and CMSIS APIs

- Global Interrupt Enable and Disable
- PRIMASK – Core Exception Mask Register

- Bit0: PM Flag

- 0: Allows activation of all exceptions
- 1: Prevents activation of all exceptions with configurable priority

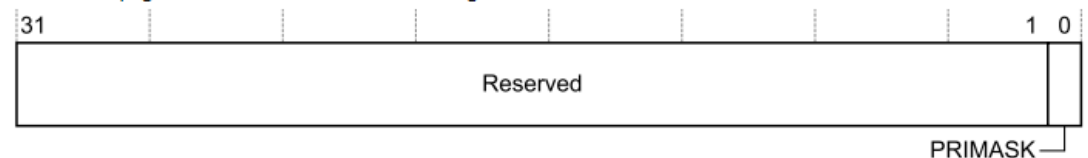
- Access using CPS (change processor state), MSR (move to SR) and MRS (move from SR)

- 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

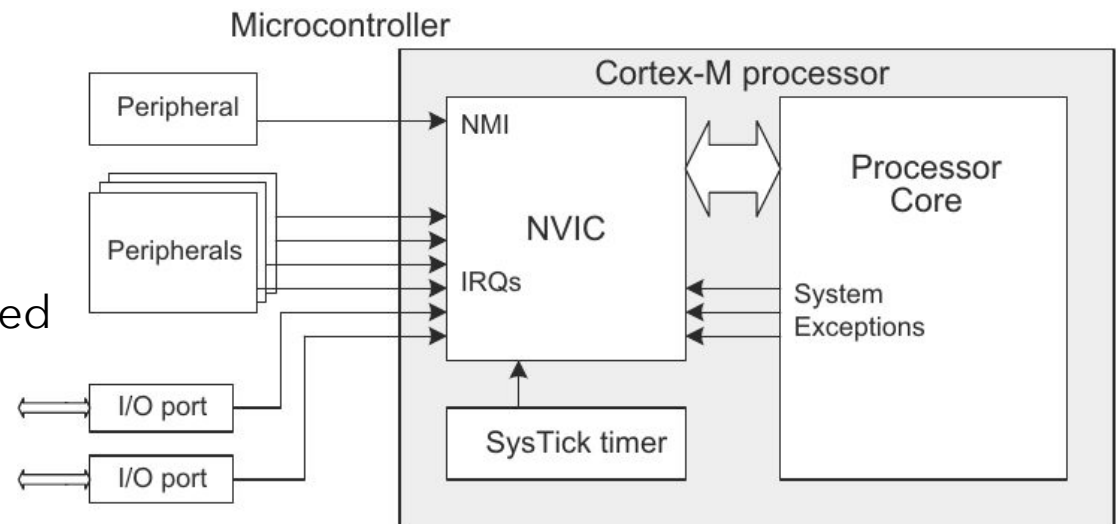
Priority Mask Register

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:



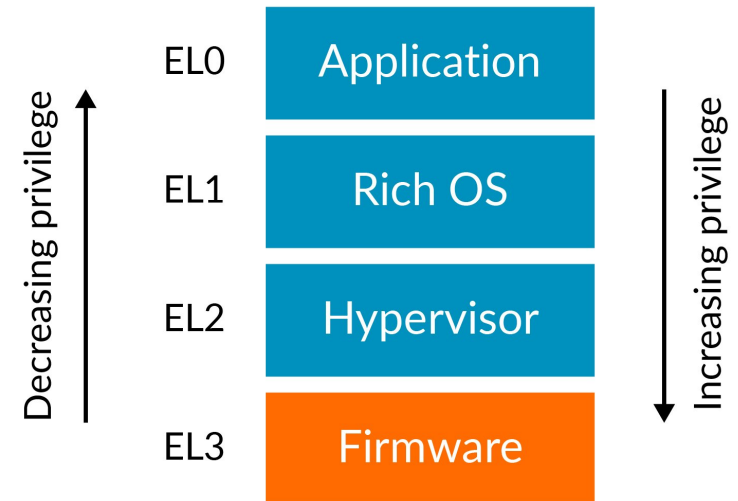
Interrupt and Exception Priorities

- Exceptions are prioritized to order the response simultaneous requests (smaller number = higher priority)
- Priorities of some exceptions are fixed
 - **RESET: -3 (HIGHEST)**
 - NMI: -2
 - HARD FAULT: -1
- Priorities of other exceptions are configured by interrupt priority registers (IPRs)
- As an example:
 - 8 IPRs: IPR0 - IPR7
 - Each 32-bit IPR configures 4 interrupts (8 bits each)
 - **0x00 (HIGHEST)**, 0x40, 0x80, 0xC0 (LOWEST)



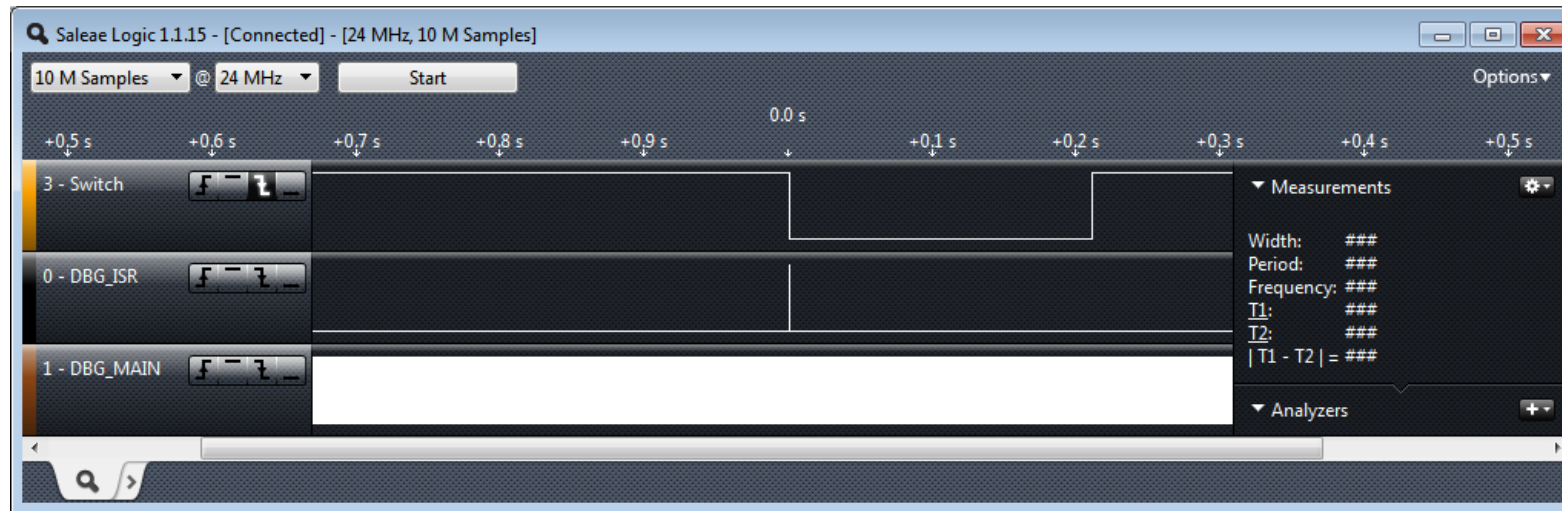
Special Cases of Prioritization

- 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 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>



Interrupt Timing Analysis

Big Picture Timing Behavior



- Switch pressed for about 0.21 sec in this example
- DBG_ISR runs in response to switch signal's falling edge
- DBG_MAIN is running continuously
 - Signal toggles between 0 and 1

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 signal
- How long does it take?
 - Finish executing the current instruction or abandon it
 - Push various registers on to the stack, fetch vector
 - C_{OVERHEAD} : extra overhead to respond to an interrupt
 - If we have external memory with wait states, this takes longer

Theoretical Maximum Interrupt Rate

- The CPU can only handle so many interrupts per second
 - F_{INT} : Actual Interrupt Rate
 - F_{MAX_INT} : Maximum Interrupt Rate
 - F_{CPU} : CPU Clock Frequency
 - C_{ISR} : No. of Cycles to Execute ISR
 - $C_{OVERHEAD}$: No. of Cycles of Overheads (saving state, vectoring, restoring state)
 - $F_{MAX_INT} = F_{CPU} / (C_{ISR} + C_{OVERHEAD})$ (assume one interrupt in the system)
- When processor is responding to interrupts, it isn't executing the other code
 - U_{INT} : Utilization consumed by Interrupt Processing
 - $U_{INT} = 100\% \times F_{INT} \times (C_{ISR} + C_{OVERHEAD}) / F_{CPU}$
 - CPU appears to run the other code at clock speed of: $(1 - U_{INT}) \times F_{CPU}$

Interrupt Service Routine (ISR) Design Trade-offs

Program Design with Interrupts

- **HOW MUCH WORK** should be done in ISR?
 - How much work is done in the non-ISR (such as, foreground)?
- Should ISRs re-enable interrupts?
 - **REENTRANCY** issues (timing, stack, priority reversal issues)
- How to **COMMUNICATE** among ISRs and other threads?
 - Data buffering
 - Data integrity
 - Race conditions

How Much Work Is Done in ISR?

- Design Trade-offs
 - Faster ISR response will delay completion of other code
 - Slower ISR response may accept ISR task timing
 - Evaluate the trade-offs based on **SYSTEM REQUIREMENTS**
- In system with multiple ISRs with short deadlines,
 - Perform critical works in ISR
 - Buffer partial results for later processing
 - Calculate the **WORST-CASE SCENARIO**

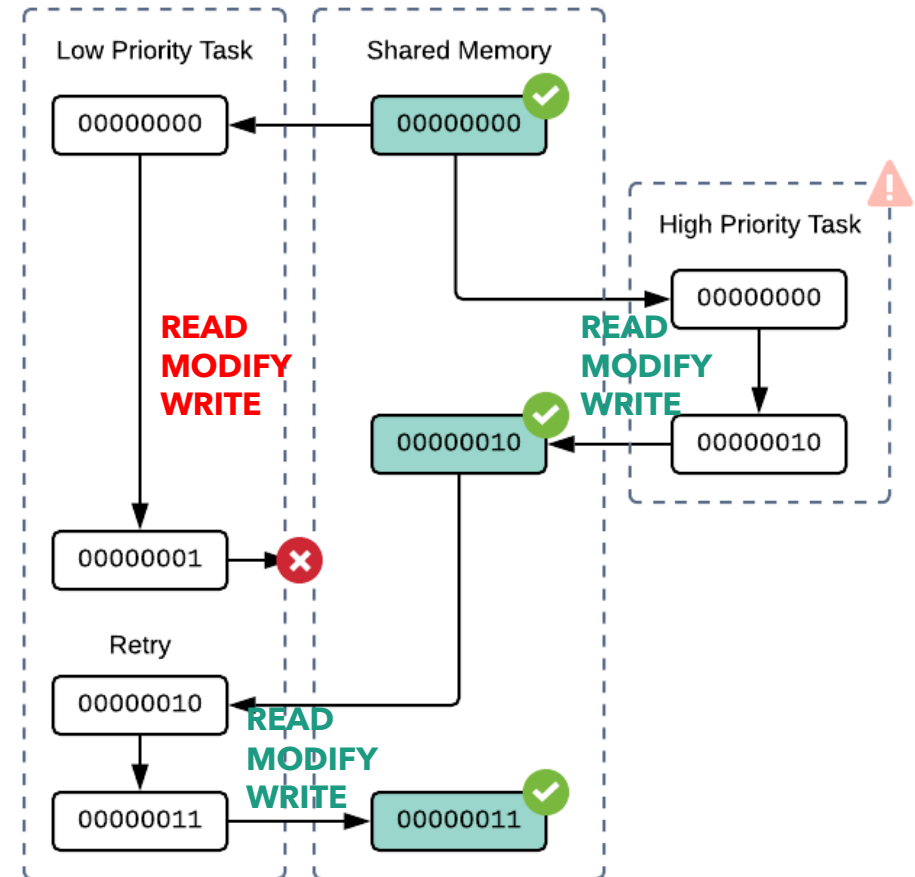
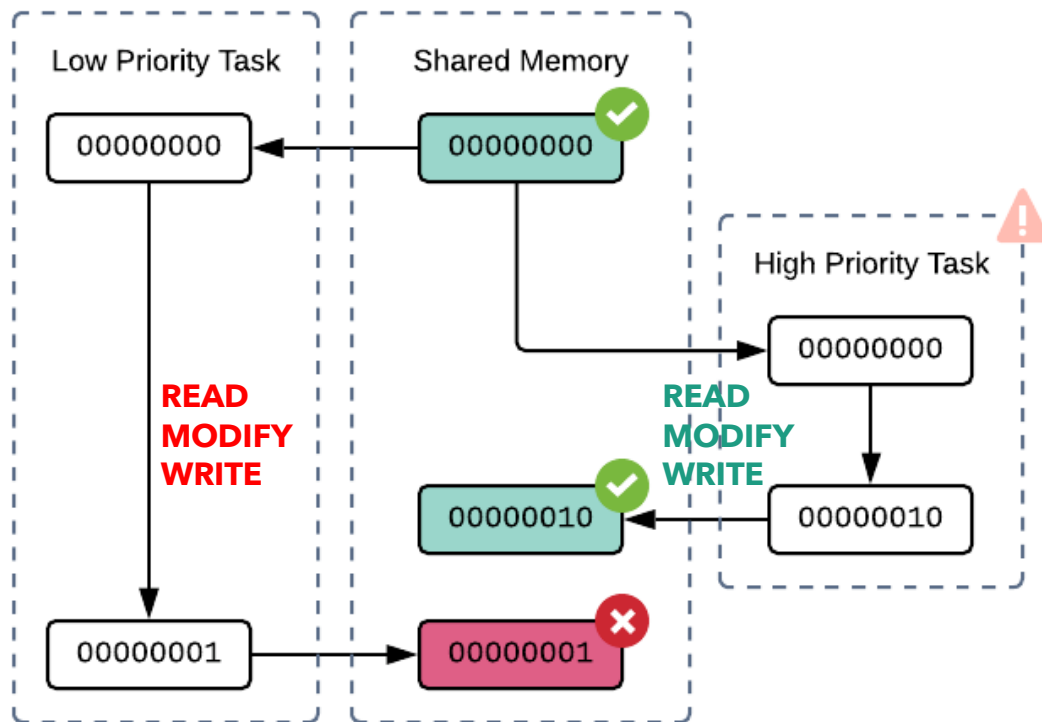
Robust Data Sharing among ISRs and Other Threads

Overview

- **VOLATILE** data – can be updated outside of the program's immediate control
- **NON-ATOMIC** shared data – can be interrupted part-way through read or write, is vulnerable to race conditions.
- As an illustration:
 - A long counter incremented from `0xFFFFFFFF_FFFFFFFF` to `0x00000000_00000000`
 - Reading the long counter in interrupt could be:
 - `0xFFFFFFFF_00000000`
 - `0x00000000_FFFFFFFF`

Data Corruption (Scenario One)

- Data can be corrupted by different threads



Volatile Data

- Compilers assume that variables in memory do not change spontaneously, and assuming:
 - *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 or before a procedure call, or when free registers run out
- This optimization can fail in some scenarios:
 - 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
- Failure Scenarios:
 1. Memory-mapped peripheral register – register changes on its own
 2. Global variables modified by an ISR – ISR changes the variable
 3. Global variables in a multithreaded application – another thread or ISR changes the variable

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 (such as: 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 (Scenario Two)

- 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 (Scenario Two)

- Problem: An **INTERRUPT AT WRONG TIME** can 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
 - DT now has a time stamp of **{10, 23, 0, 0}**
 - *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 the instruction set and word size (32 bits in 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 Writer? 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 Writers?
 - 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.
- Operating System (OS) class can cover this topic in more detail.

Solution: Briefly Disable Preemption

- **NO PREEMPTION** within **CRITICAL REGION**
- 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 (such as, 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);  
}
```

Recap on Robust Sharing of Data among Threads

- In thread/ISR diagram, **IDENTIFY** shared data
- Determine which shared data size 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 **VOLTAILE** 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, RESTORE** masking status
 - CMSIS-CORE interface: `__disable_irq()`, `__get_PRIMASK()`, `__set_PRIMASK()`

Quick Summary

- Background
- Entering Interrupt Handler
- Exiting Interrupt Handler
- ARM Core Interrupt Design and APIs
- Interrupt Timing Analysis
- Interrupt Service Routine (ISR) Design Trade-offs
- Robust Data Sharing among ISRs & Other Threads