

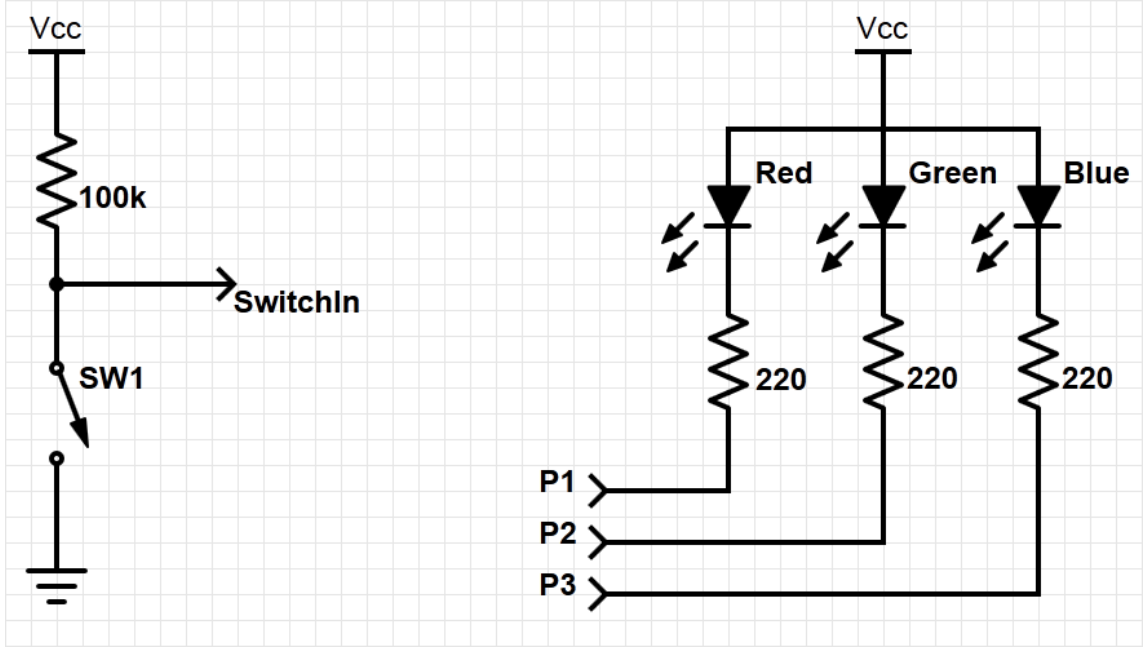
# Exceptions and Interrupts

# Overview

- Exception and Interrupt Concepts
  - Entering an Exception Handler
  - Exiting an Exception Handler
- Core Interrupts
  - Using Port Module and External Interrupts
- Timing Analysis
- Program Design with Interrupts
  - Sharing Data Safely Between ISRs and Other Threads

# **EXCEPTION AND INTERRUPT CONCEPTS**

# Example System with Interrupt

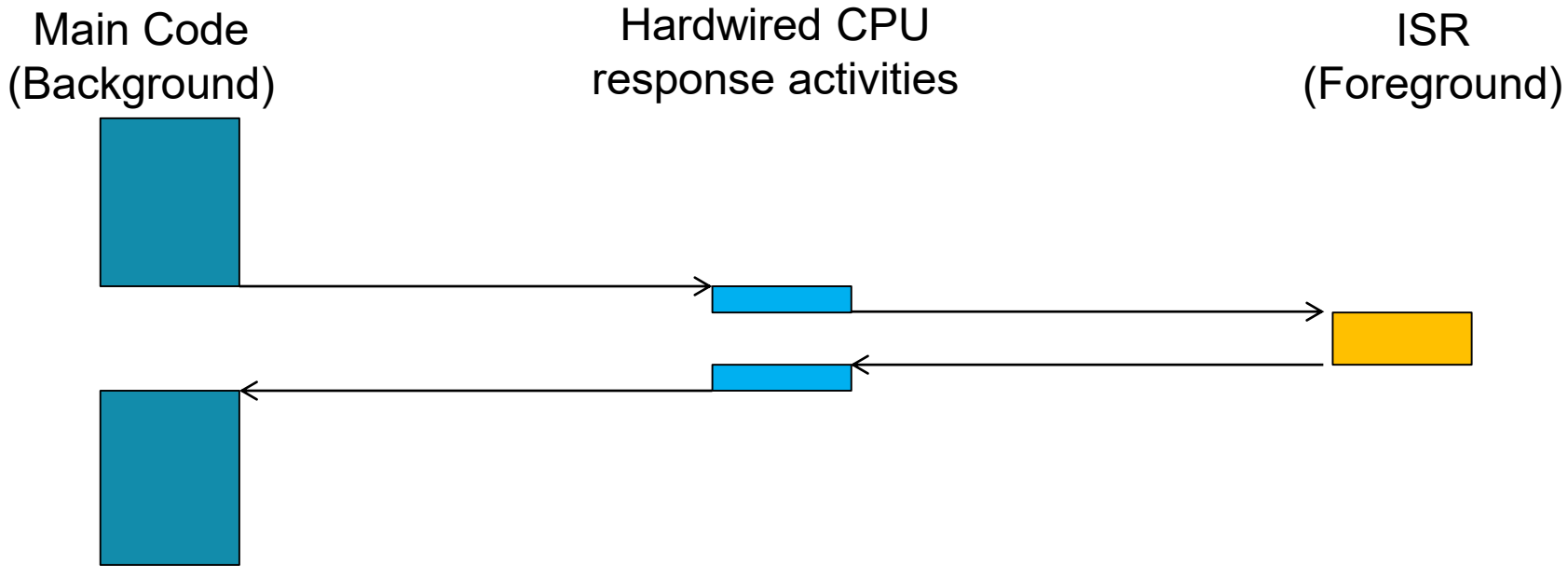


- Goal: Change color of RGB LED when switch is pressed
- Will explain details of interfacing with switch and LEDs in GPIO module later
- Need to add external switch

# How to Detect Switch is Pressed?

- Polling - use software to check it
  - Slow - need to explicitly check to see if switch is pressed
  - Wasteful of CPU time - the faster a response we need, the more often we need to check
  - Scales badly - difficult to build system with many activities which can respond quickly. Response time depends on all other processing.
- Interrupt - use special hardware in MCU to detect event, run specific code (interrupt service routine - ISR) in response
  - Efficient - code runs only when necessary
  - Fast - hardware mechanism
  - Scales well
    - ISR response time doesn't depend on most other processing.
    - Code modules can be developed independently

# Interrupt or Exception Processing Sequence

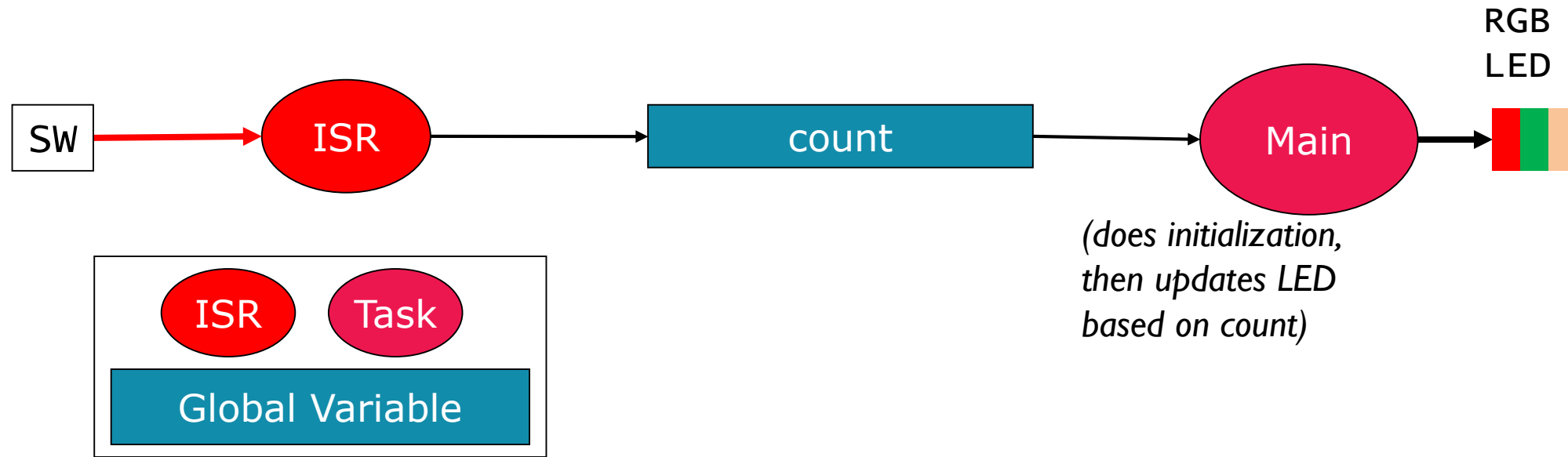


- Other code (background) is running
- Interrupt trigger occurs
- Processor does some hard-wired processing
- Processor executes ISR (foreground), including return-from-interrupt instruction at end
- Processor resumes other code

# Interrupts

- Hardware-triggered asynchronous software routine
  - Triggered by hardware signal from peripheral or external device
  - Asynchronous - can happen anywhere in the program (unless interrupt is disabled)
  - Software routine - Interrupt service routine runs in response to interrupt
- Fundamental mechanism of microcontrollers
  - Provides efficient event-based processing rather than polling
  - Provides quick response to events regardless\* of program state, complexity, location
  - Allows many multithreaded embedded systems to be responsive without an operating system (specifically task scheduler)

# Example Program Requirements & Design

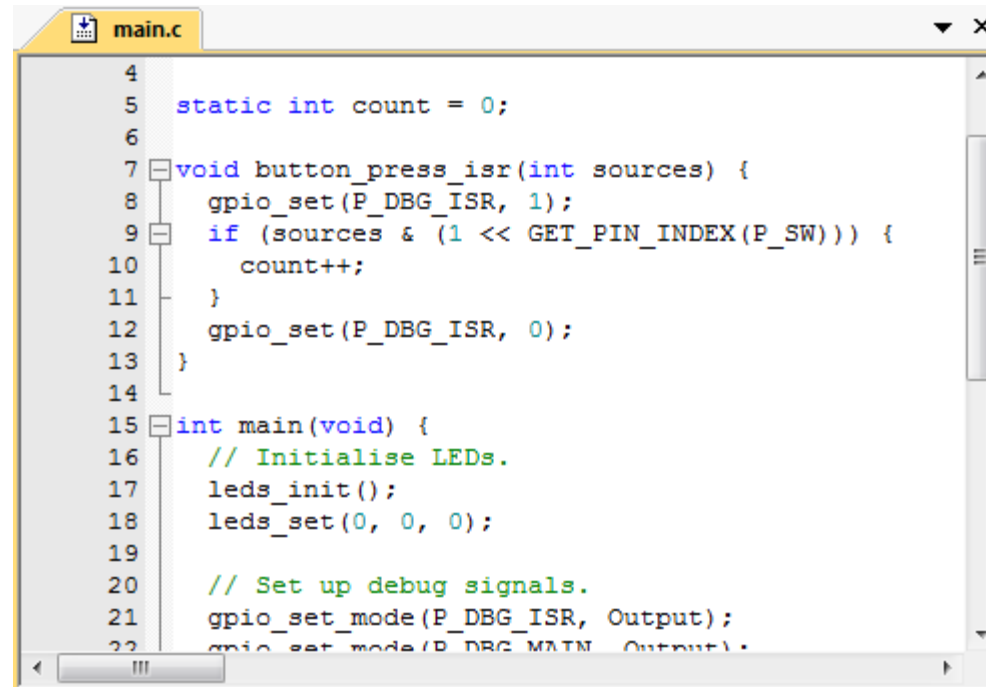


- Req1: When Switch SW is pressed, ISR will increment count variable
- Req2: Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1)
- Req3: Main code will toggle its debug line each time it executes
- Req4: ISR will raise its debug line (and lower main's debug line) whenever it is executing



# Example Exception Handler

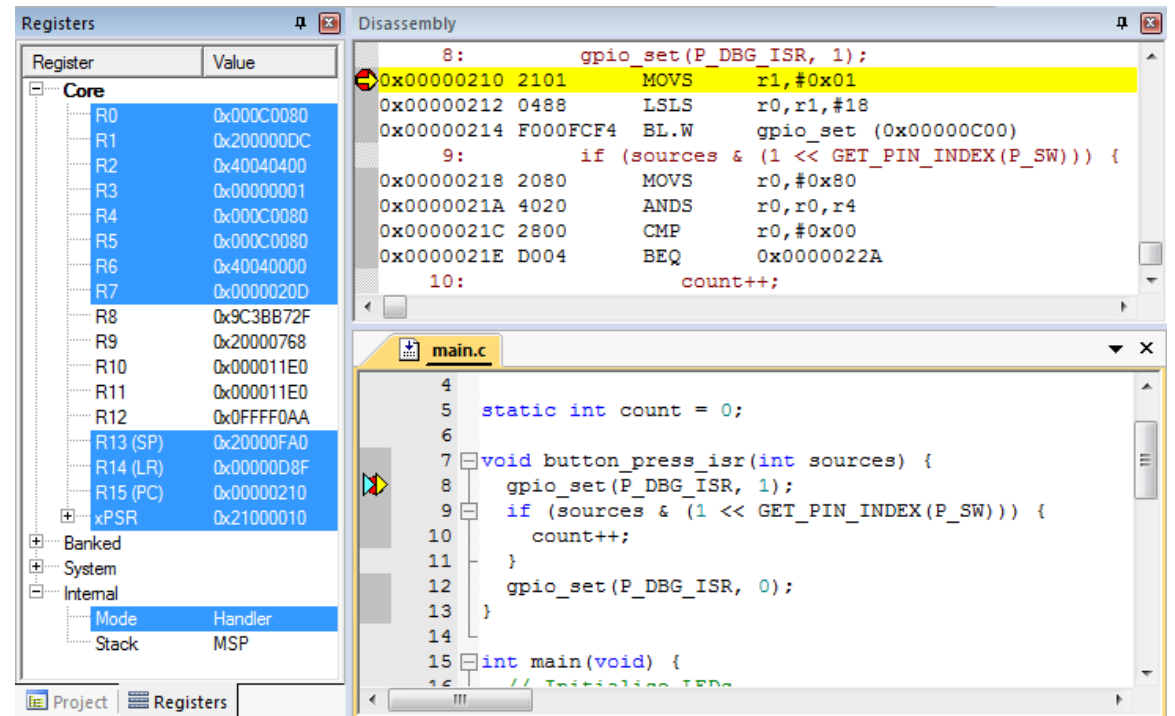
- We will examine processor's response to exception in detail



```
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);
```

# 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



# **ENTERING AN EXCEPTION HANDLER**

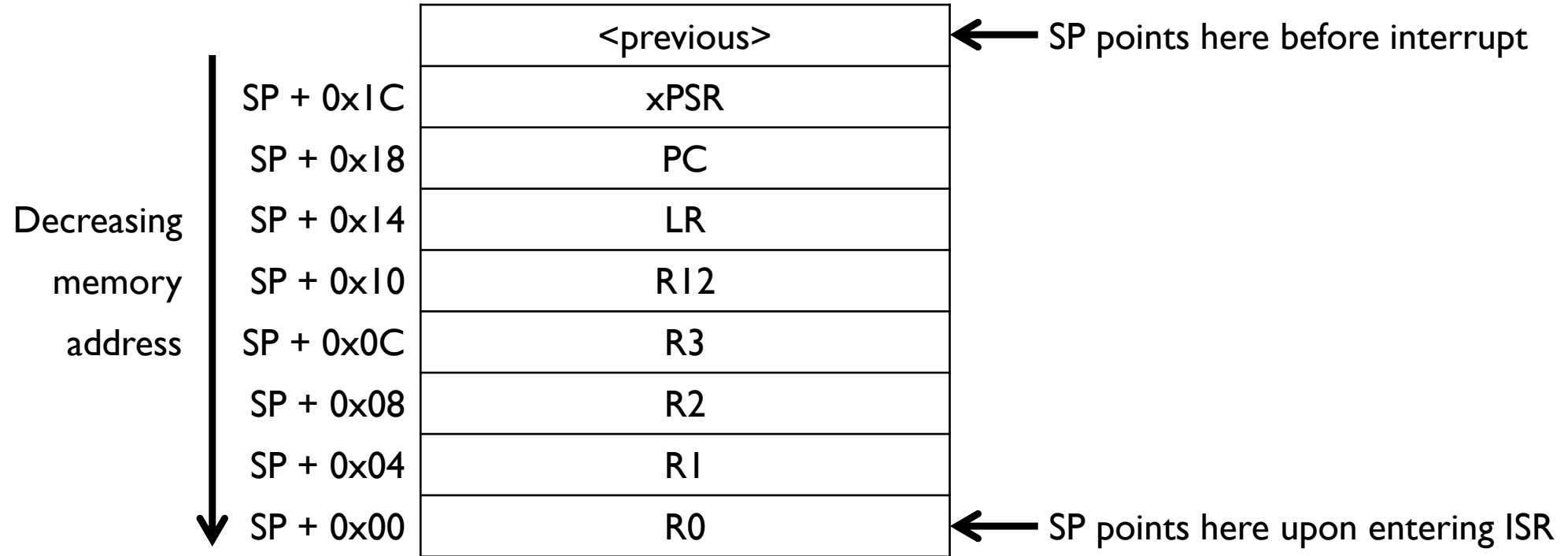
# CPU's Hardwired Exception Processing

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
<b>Core</b>	
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)	0x2000FD8
R14 (LR)	
R15 (PC)	
<b>xPSR</b>	
<b>Banked</b>	
MSP	0x2000FD8
PSP	0xBEF5DFF0
<b>System</b>	
PRIMASK	0
CONTROL	0x00
<b>Internal</b>	
Mode	
Stack	

SP value is reduced since registers have been pushed onto stack

Memory 1					
Address: sp					
0x2000FD8:	00000000	00000000	00000000	0000008C	0FFFF0AA
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

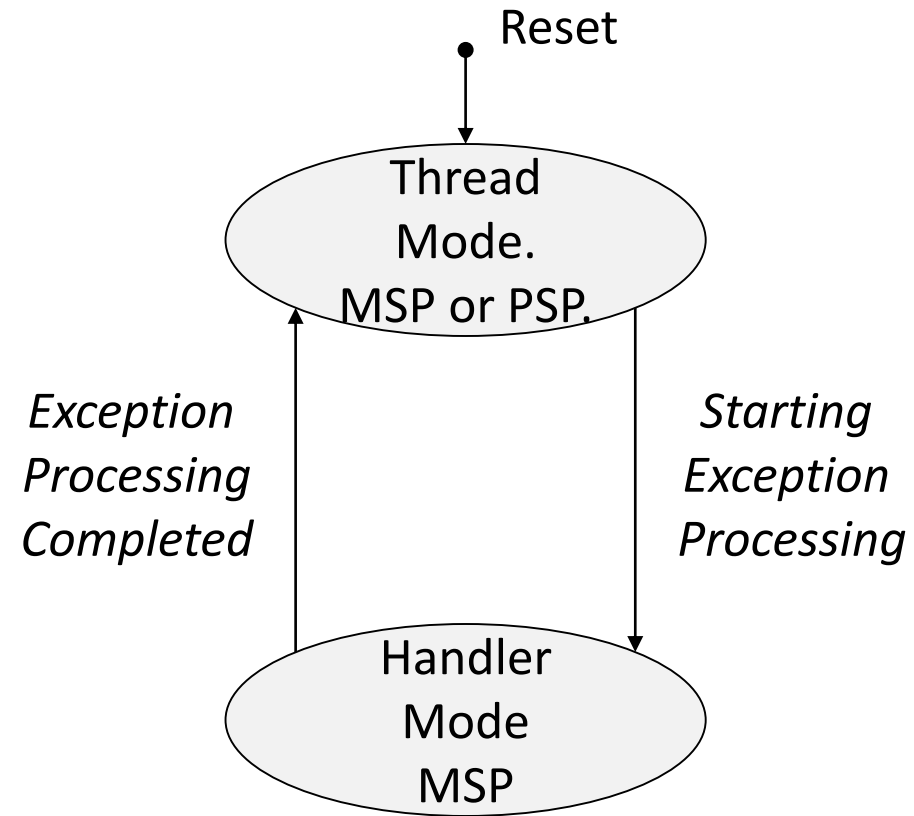
Saved LR

Saved PC

Saved xPSR

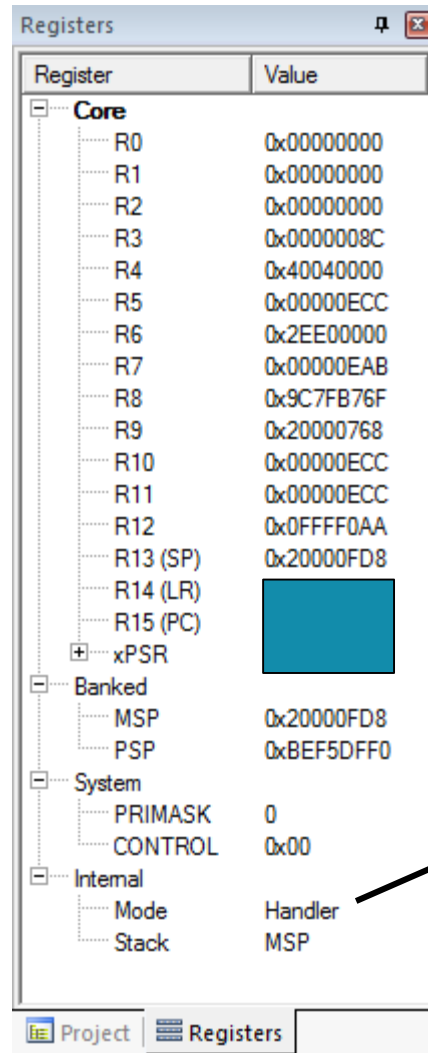
### 3. Switch to Handler/Privileged Mode

- Handler mode always uses Main SP





# Handler and Privileged Mode

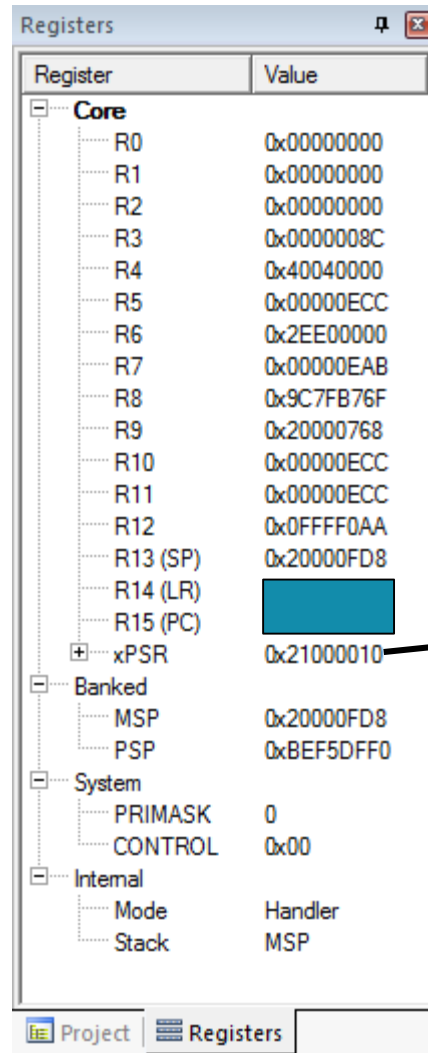


The image shows a 'Registers' window from a debugger. It contains a tree view on the left and a table of register values on the right. The tree view has categories: Core, Banked, System, and Internal. The 'Internal' category is expanded, showing 'Mode' as 'Handler' and 'Stack' as 'MSP'. A blue square is visible next to the 'R15 (PC)' register value. A black arrow points from the 'Mode' field to a text box on the right.

Register	Value
<b>Core</b>	
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)	0x2000FD8
R14 (LR)	
R15 (PC)	
<b>Banked</b>	
MSP	0x2000FD8
PSP	0xBEF5DFF0
<b>System</b>	
PRIMASK	0
CONTROL	0x00
<b>Internal</b>	
Mode	Handler
Stack	MSP

Mode changed to Handler. Was already using MSP

# Update IPSR with Exception Number



Register	Value
<b>Core</b>	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x0000008C
R4	0x40040000
R5	0x0000ECC
R6	0x2EE00000
R7	0x0000EAB
R8	0x9C7FB76F
R9	0x2000768
R10	0x0000ECC
R11	0x0000ECC
R12	0x0FFF0AA
R13 (SP)	0x2000FD8
R14 (LR)	
R15 (PC)	
xPSR	0x21000010
<b>Banked</b>	
MSP	0x2000FD8
PSP	0xBEF5DFF0
<b>System</b>	
PRIMASK	0
CONTROL	0x00
<b>Internal</b>	
Mode	Handler
Stack	MSP

Exception number 0x10  
(interrupt number + 0x10)

## 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

# 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

# Upon Entry to Handler

Register	Value
<b>Core</b>	
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)	0x2000FD8
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}
```

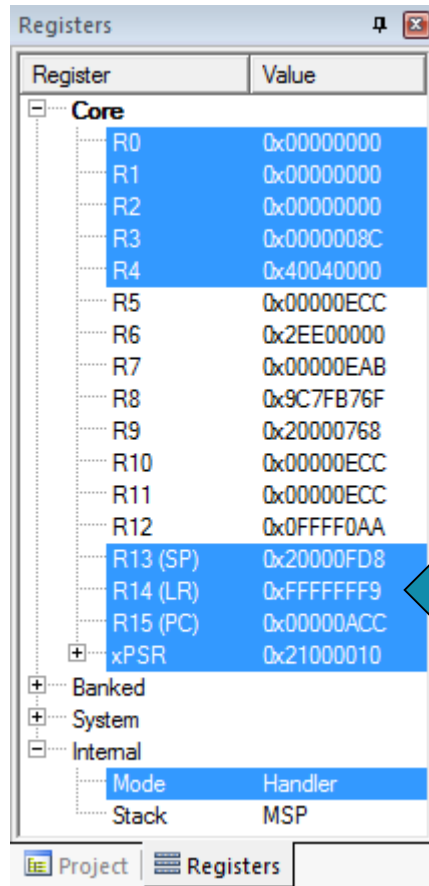
PC has been loaded with  
start address of handler

## 5. Load LR With EXC\_RETURN Code

EXC_RETURN	Return Mode	Return Stack	Description
0xFFFF_FFFI	0 (Handler)	0 (MSP)	Return to exception handler
0xFFFF_FFF9	I (Thread)	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	I (Thread)	I (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 (I)
    - Previous value of SPSEL
  - Which mode to return to? Handler (0) or Thread (I)
    - Another exception handler may have been running when this exception was requested

# Updated LR With EXC\_RETURN Code



The image shows a 'Registers' window from a debugger. It displays a list of registers for a 'Core'. Registers R0 through R15 are listed, along with xPSR. R14 (LR) is highlighted in blue, and a large blue arrow points to it from the right. Below the Core registers, there are sections for Banked, System, and Internal registers. The Internal section shows 'Mode' and 'Handler' registers, and 'Stack' and 'MSP' registers.

Register	Value
<b>Core</b>	
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)	0xFFFFF9
R15 (PC)	0x0000ACC
xPSR	0x2100010
<b>Banked</b>	
<b>System</b>	
<b>Internal</b>	
Mode	Handler
Stack	MSP

## 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}
```



# After Handler Has Saved More Context

Register	Value
<b>Core</b>	
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)	0xFFFFFFFF
R15 (PC)	0x0000ACE
xPSR	0x21000010
<b>Banked</b>	
MSP	0x2000FD0
PSP	0xBEF5DFF0
<b>System</b>	
PRIMASK	0
CONTROL	0x00
<b>Internal</b>	
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	0FFFF0AA	00000A0B	00000A70	21000000
0x2000FF8:	00000002	00000A0B	000001F1	000001F1	000001F1
0x200100C:	000001F1	000001F1	000001F1	000001F1	000001F1
0x2001020:	000001F1	000001F1	000001F1	000001F1	000001F1

Labels for Memory 2:

- 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

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

# What Will Be Popped from Stack?

- R4: 0x4040\_0000
- PC: 0xFFFF\_FFF9

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

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	FFFFFFF9	00000000	00000000	00000000
0x20000FE4:	0000008C	0FFF0AA	00000A0B	00000A70	21000000
0x20000FF8:	00000002	00000A0B	000001F1	000001F1	000001F1
0x2000100C:	000001F1	000001F1	000001F1	000001F1	000001F1
0x20001020:	000001F1	000001F1	000001F1	000001F1	000001F1

Labels pointing to memory locations:

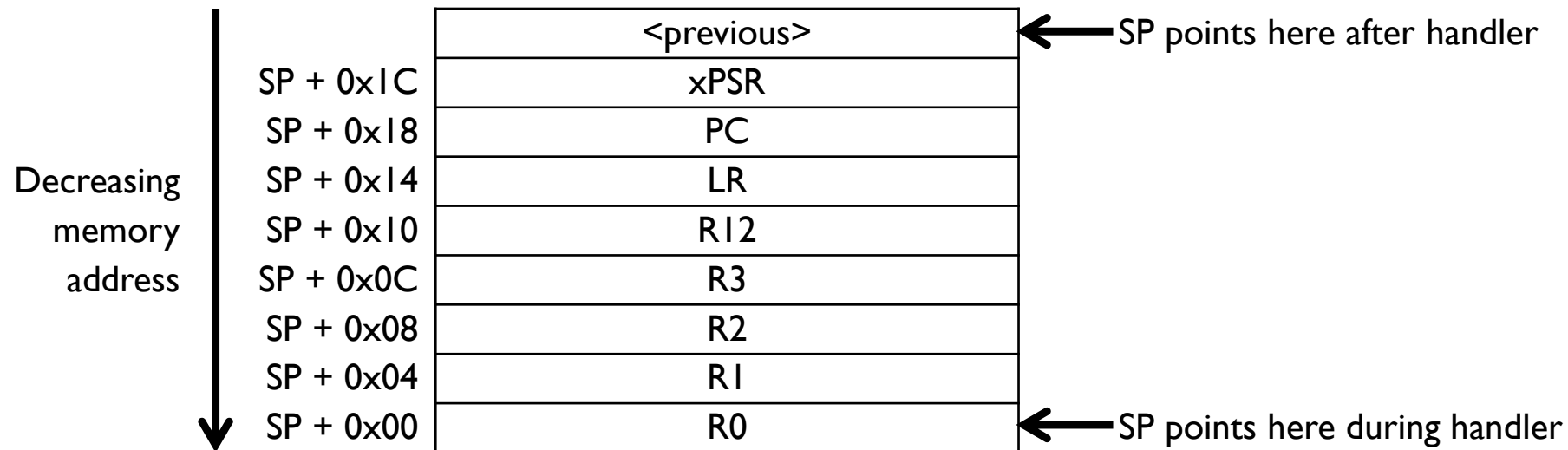
- Saved R4 (points to 0x20000FD0)
- Saved LR (points to 0x20000FE4)
- Saved R0 (points to 0x20000FF8)
- Saved R1 (points to 0x20000FF8)
- Saved R2 (points to 0x20000FF8)
- Saved R3 (points to 0x20000FF8)
- Saved R12 (points to 0x20000FF8)
- Saved LR (points to 0x20000FF8)
- Saved PC (points to 0x20000FF8)
- Saved xPSR (points to 0x20000FF8)

## 2. Select Stack, Restore Context

- Check EXC\_RETURN (bit 2) to determine from which SP to pop the context

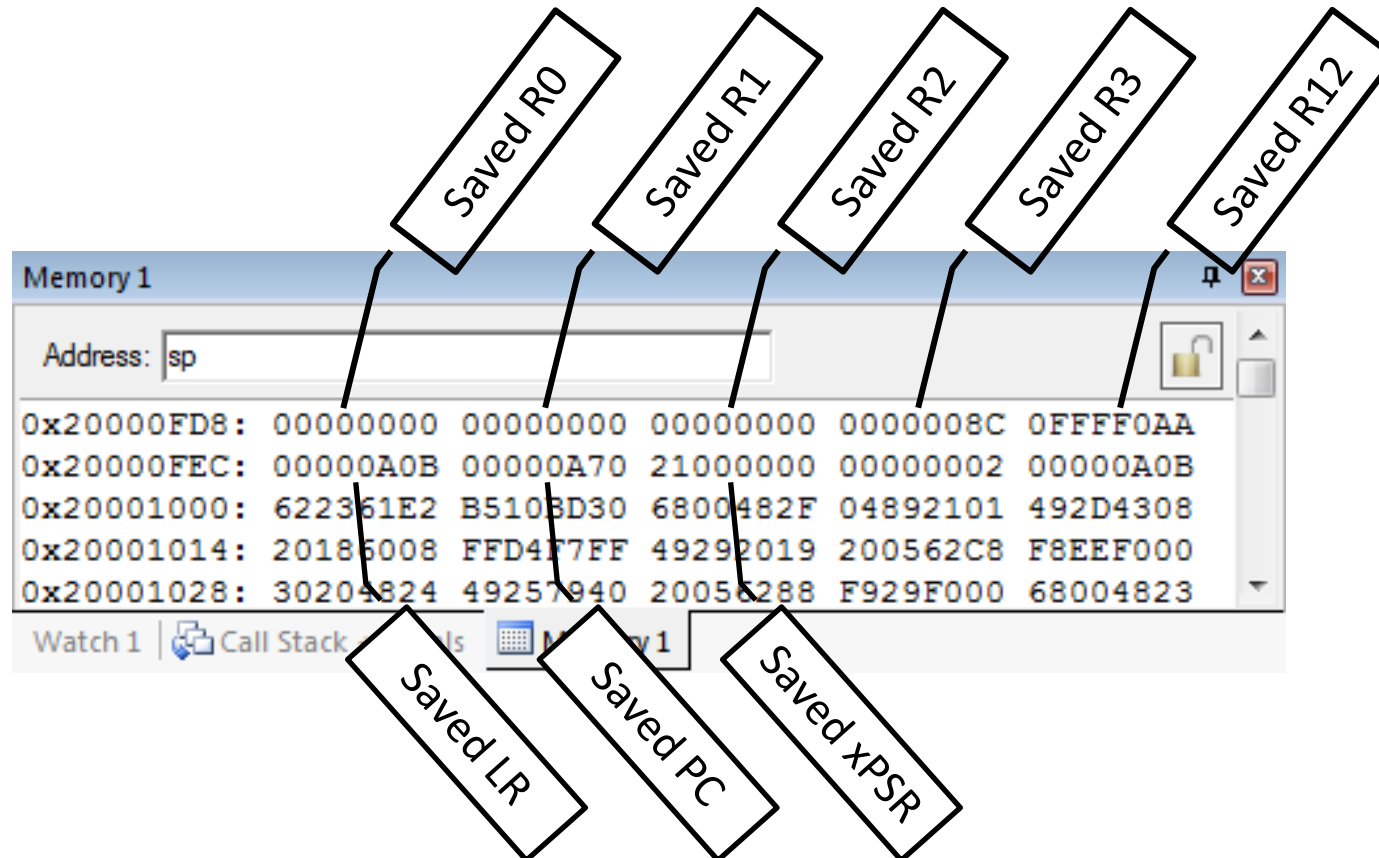
EXC_RETURN	Return Stack	Description
0xFFFF_FFF1	0 (MSP)	Return to exception handler with MSP
0xFFFF_FFF9	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	1 (PSP)	Return to thread with PSP

- Pop the registers from that stack



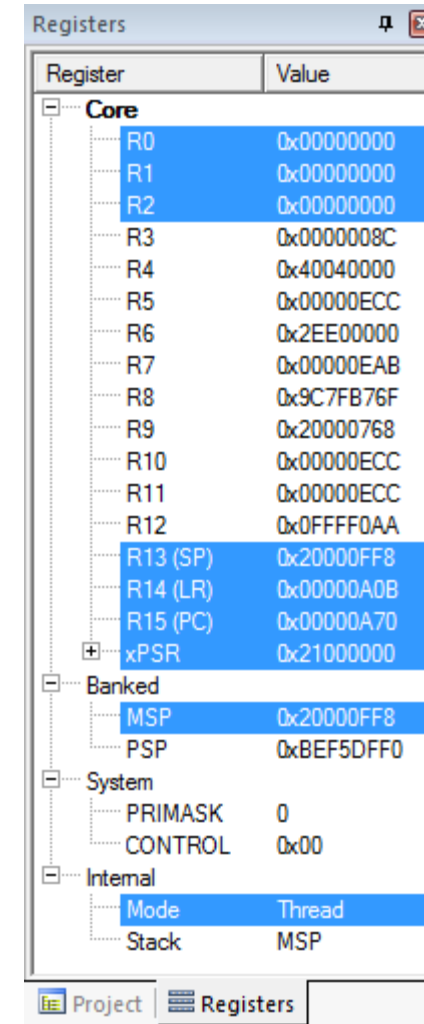
# Example

- 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
<b>Core</b>	
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
<b>Banked</b>	
MSP	0x2000FF8
PSP	0xBEF5DFF0
<b>System</b>	
PRIMASK	0
CONTROL	0x00
<b>Internal</b>	
Mode	Thread
Stack	MSP

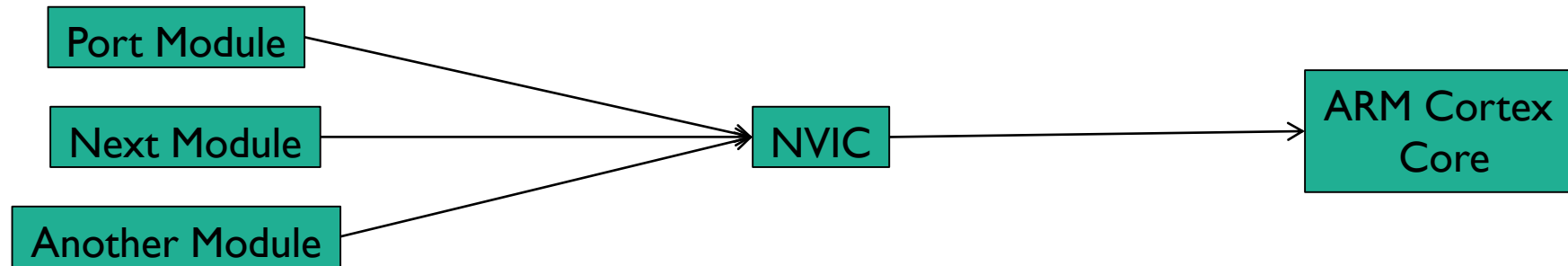


# PROCESSOR CORE INTERRUPTS

# 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
- 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, A&P

# 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
  - 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)

# Core Exception Mask Register

- 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

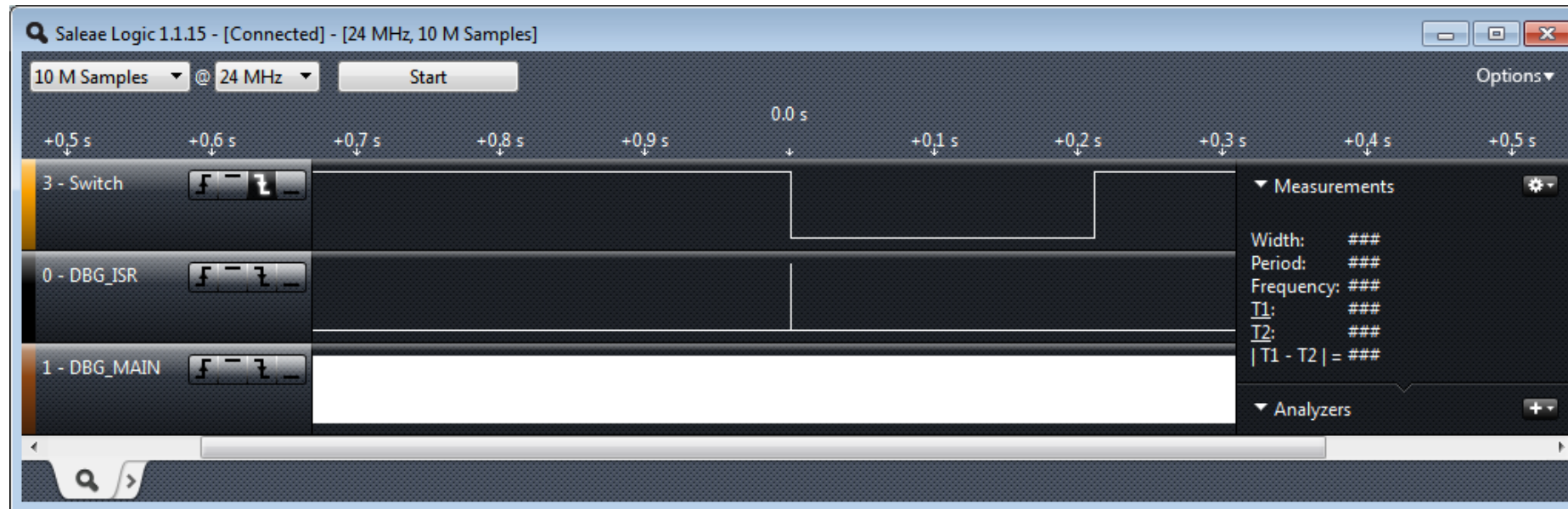
# Special Cases of Prioritization

- 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

# **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 will investigate this in the lab exercise.

# 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
  - $F_{\text{CPU}}$ : CPU clock frequency
  - $C_{\text{ISR}}$ : Number of cycles ISR takes to execute
  - $C_{\text{Overhead}}$ : 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
  - $U_{\text{Int}}$ : 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