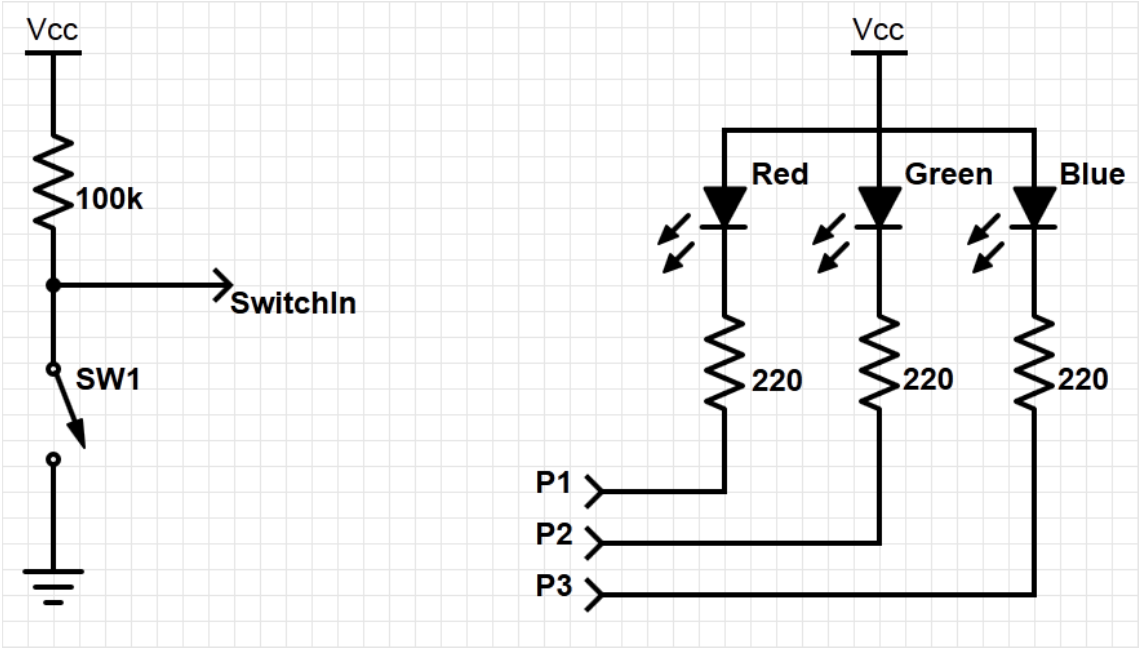
Exceptions and Interrupts

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



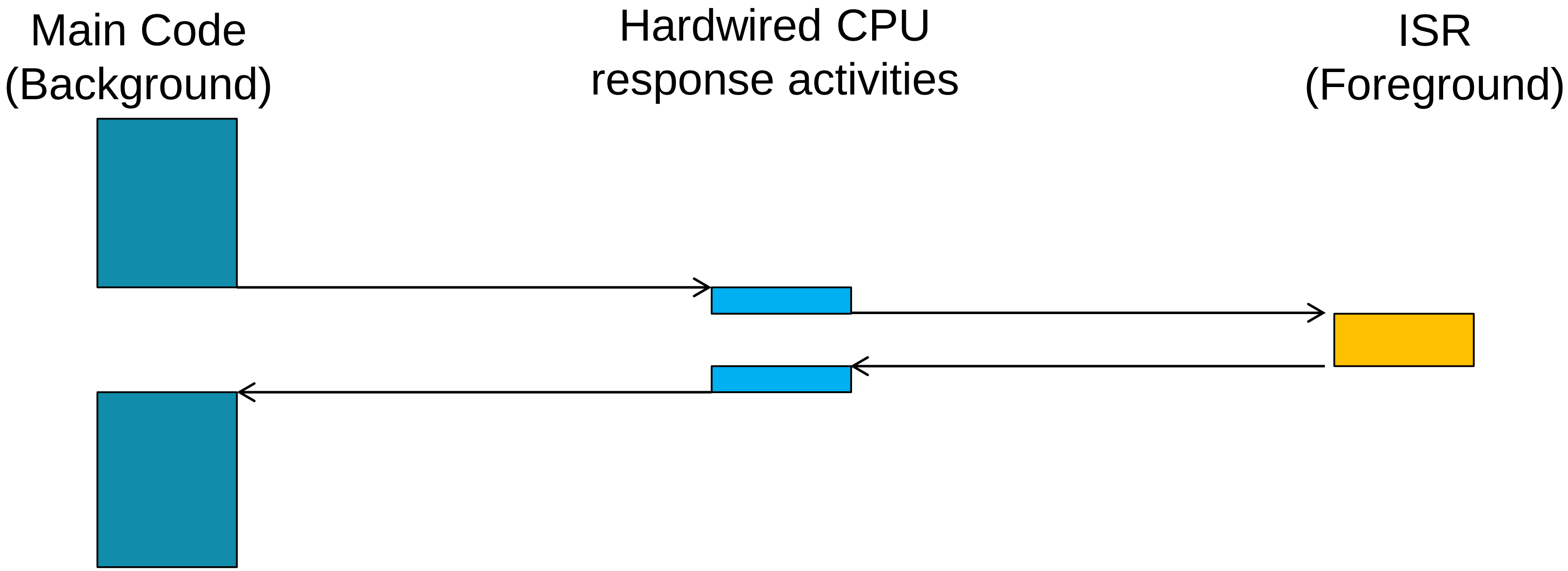
- Goal: Change color of RGB LED when switch is pressed
- Will explain details of interfacing with switch and LEDs in GPIO module later
- Need to add external switch

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



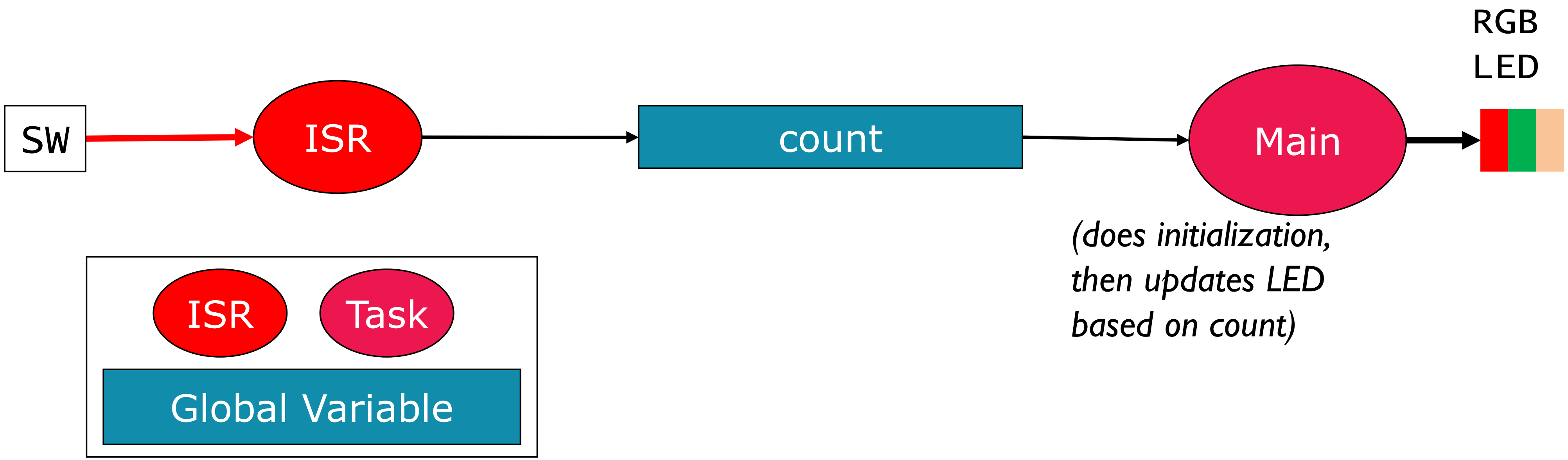
- Other code (background) is running
- Interrupt trigger occurs
- Processor does some hard-wired processing
- Processor executes ISR (foreground), including return-from-interrupt instruction at end
- Processor resumes other code

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

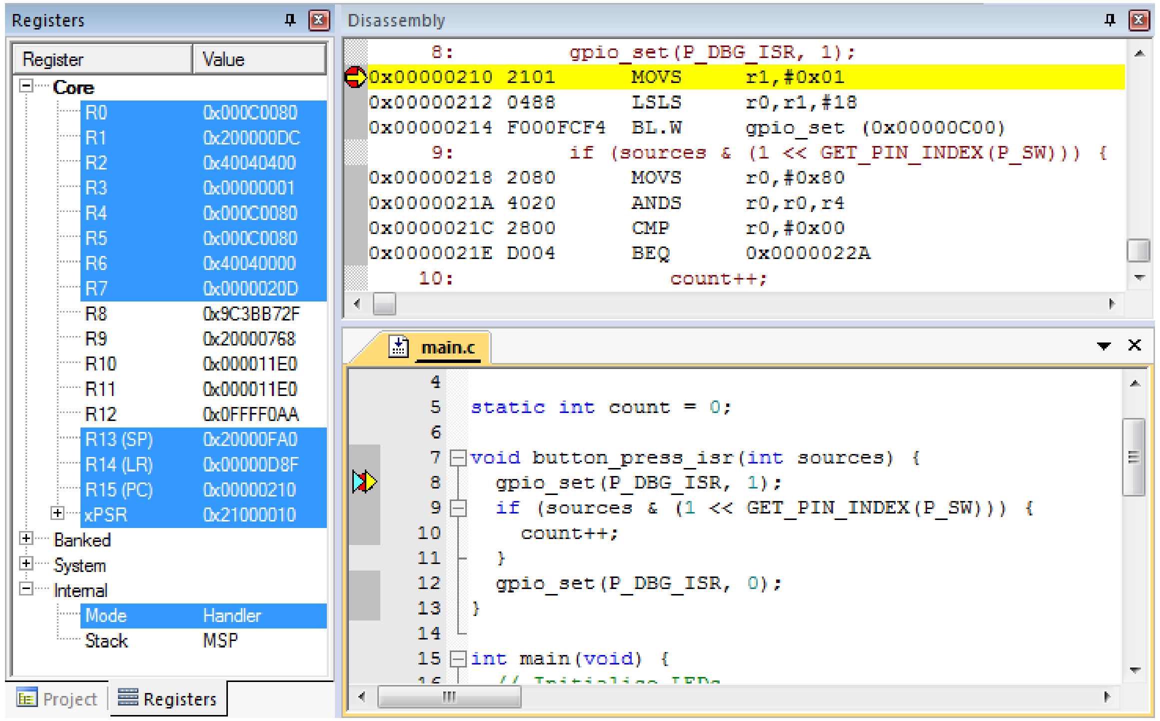
- We will examine processor's response to exception in detail

main.c

```c

static int count = 0;

void button_press_isr(int sources) {
  gpio_set(P_DBG_ISR, 1);
  if (sources & (1 << GET_PIN_INDEX(P_SW))) {
    count++;
  }
  gpio_set(P_DBG_ISR, 0);
}

int main(void) {
  // Initialise LEDs.
  leds_init();
  leds_set(0, 0, 0);

  // Set up debug signals.
  gpio_set_mode(P_DBG_ISR, Output);
  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



ARM

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

1. Finish Current Instruction

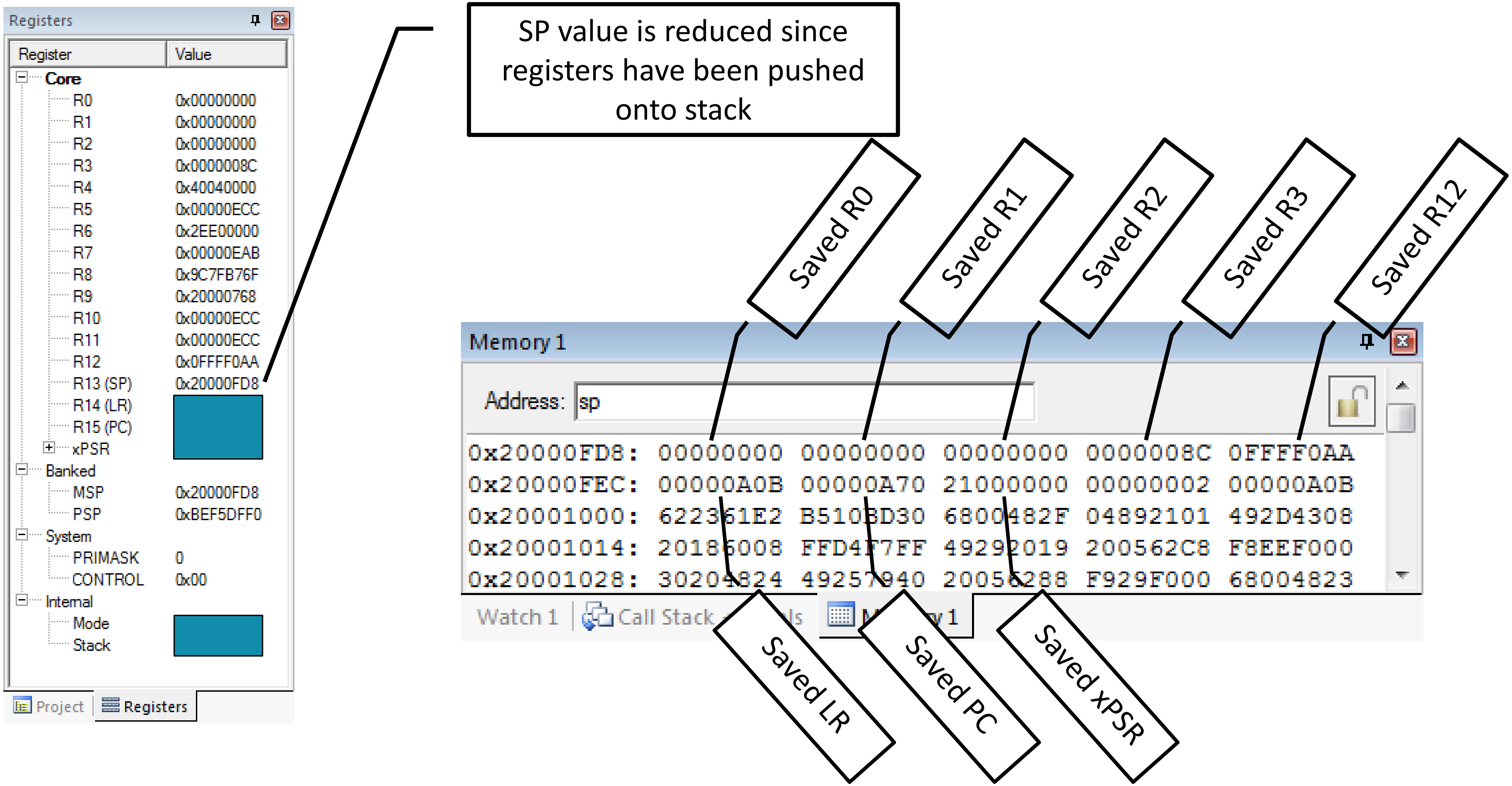
- Most instructions are short and finish quickly
- Some instructions may take many cycles to execute
  - Load Multiple (LDM), Store Multiple (STM), Push, Pop, MULS (32 cycles for some CPU core implementations)
- This will delay interrupt response significantly
- If one of these is executing when the 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

	Offset	Stack	
		<previous>	SP points here before interrupt
Decreasing memory address	SP + 0x1C	xPSR	
	SP + 0x18	PC	
	SP + 0x14	LR	
	SP + 0x10	R12	
	SP + 0x0C	R3	
	SP + 0x08	R2	
	SP + 0x04	R1	
	SP + 0x00	R0	SP points here upon entering ISR

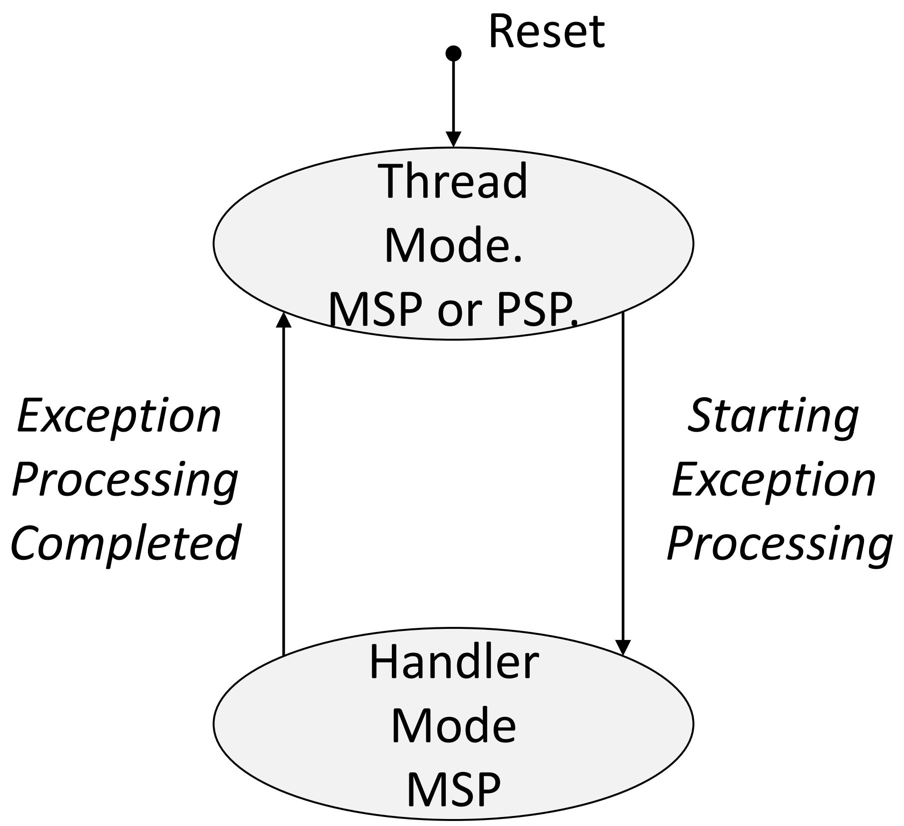
- Two SPs: Main (MSP), process (PSP)
- Which is active depends on operating mode, CONTROL register bit 1
- Stack grows toward smaller addresses

Context Saved on Stack

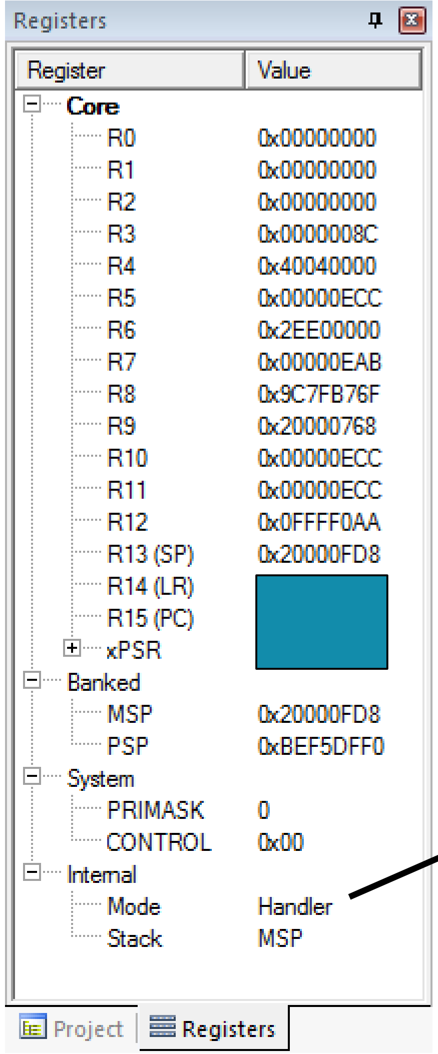


3. Switch to Handler/Privileged Mode

- Handler mode always uses Main SP

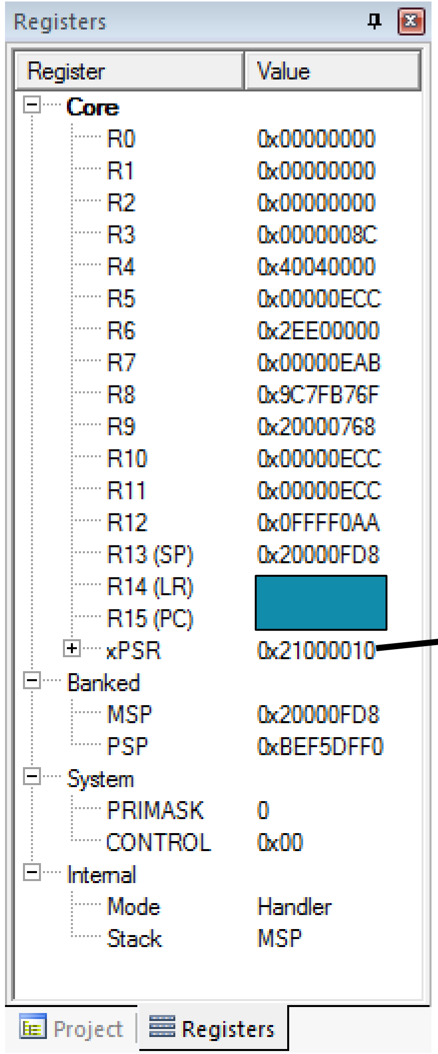


Handler and Privileged Mode



Mode changed to Handler. Was already using MSP

Update IPSR with Exception Number



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
	IRQ1_Handler
…	
Reset:	
…	
NMI_IRQHandler:	
…	
IRQ0_Handler:	
…	
IRQ1_Handler:	

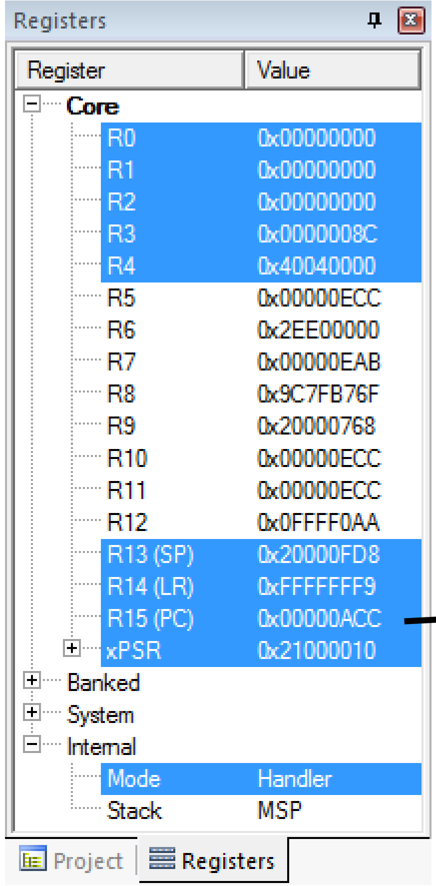
- The program counter is selected from the vector table depending on exception

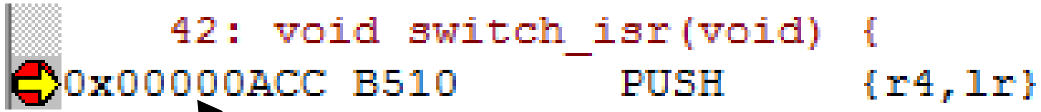
Can Examine Vector Table With Debugger

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





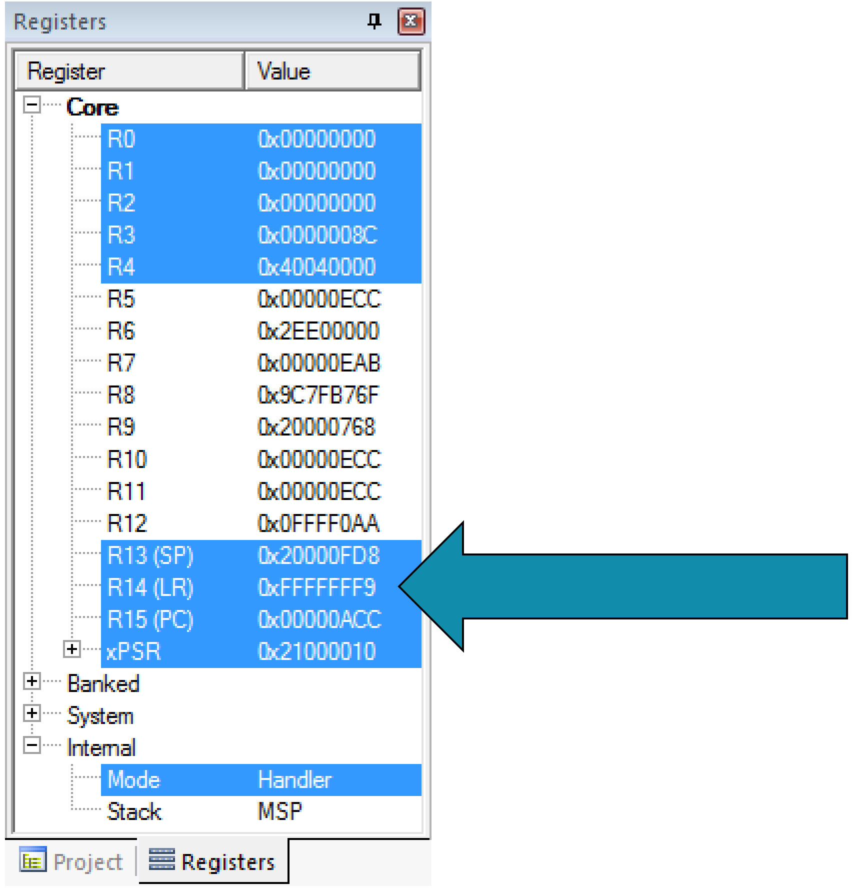
PC has been loaded with start address of handler

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
  - Which mode to return to? Handler (0) or Thread (1)
    - Another exception handler may have been running when this exception was requested

Updated LR With EXC_RETURN Code



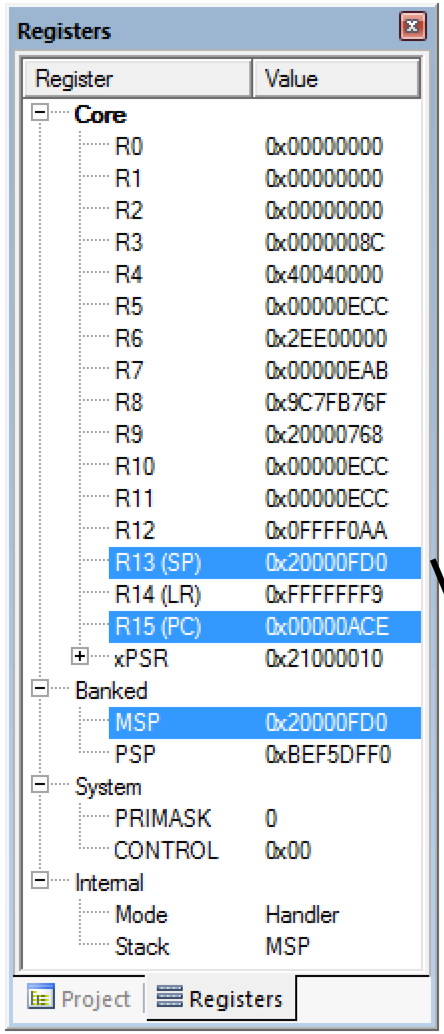
6. Start Executing Exception Handler

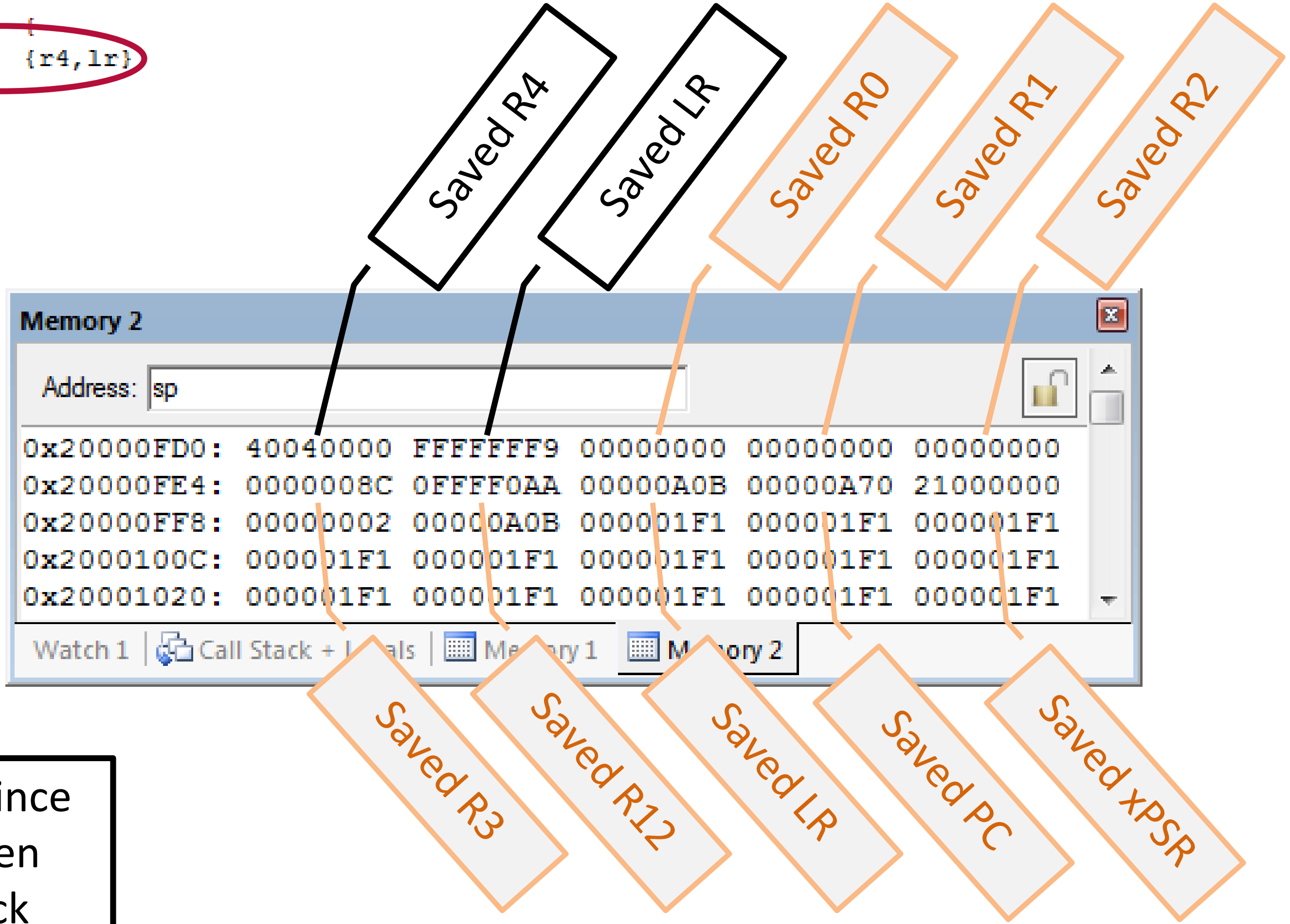
- Exception handler starts running, unless preempted by a higher-priority exception

- 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}
```

After Handler Has Saved More Context





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

1. Execute Instruction for Exception Return

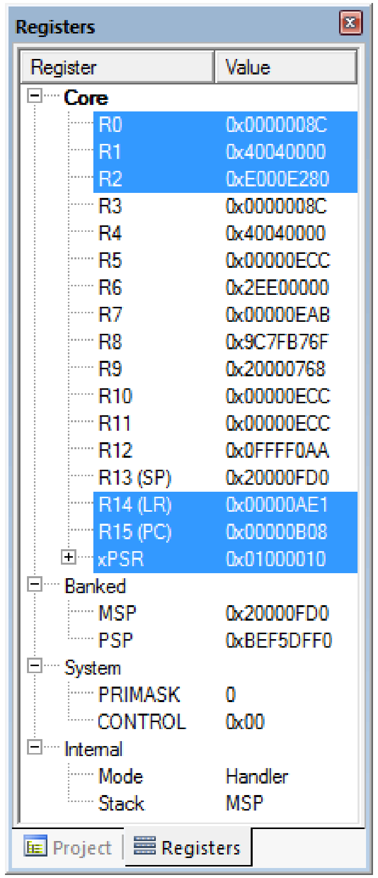
- No “return from interrupt” instruction
- Use regular instruction instead
  - BX LR - Branch to address in LR by loading PC with LR contents
  - POP …, PC - Pop address from stack into PC
- … with a special value EXC_RETURN loaded into the PC to trigger exception handling processing
  - BX LR used if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

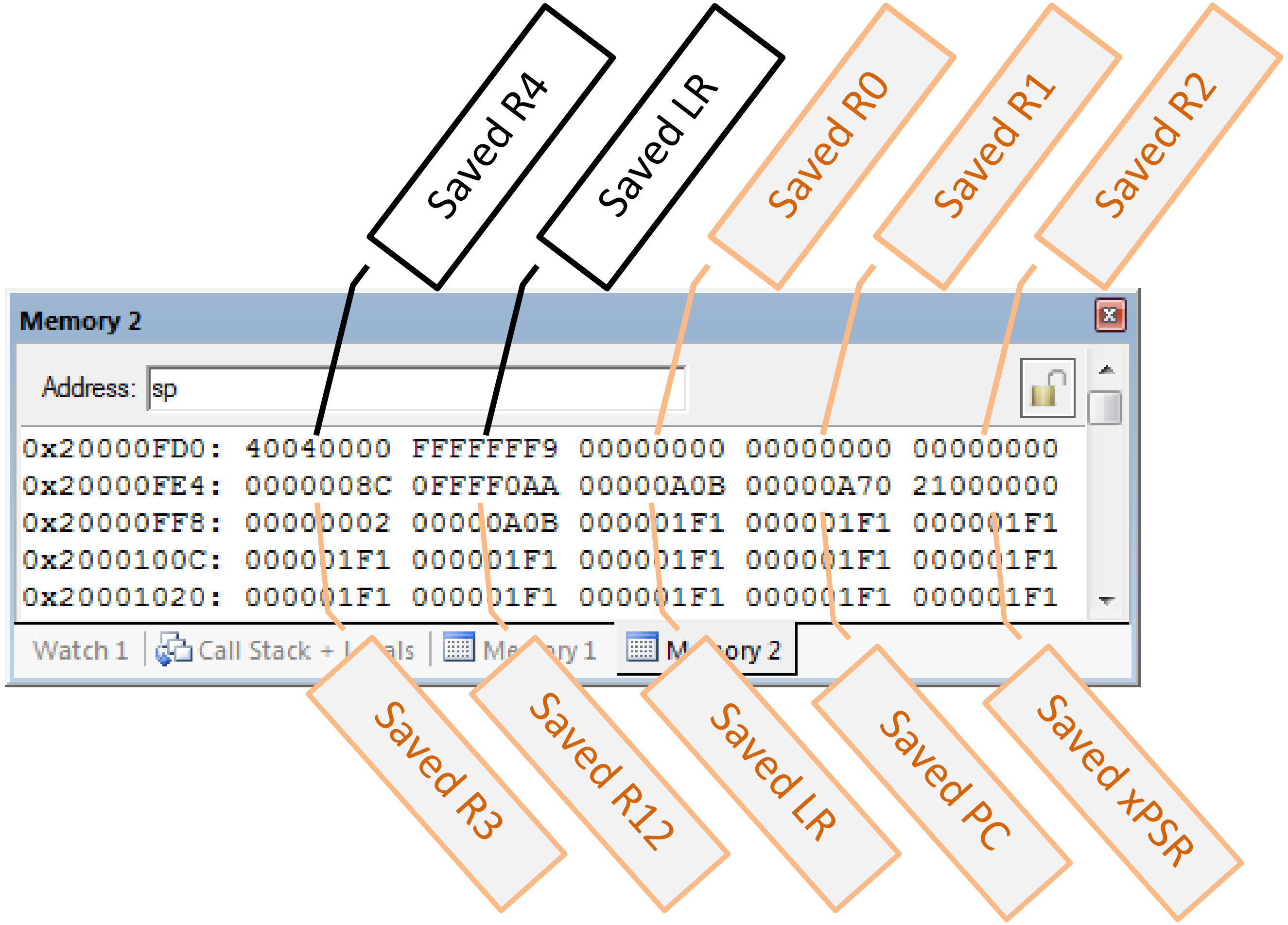
```
    51: }
0x00000B08 BD10       POP      {r4,pc}
```

What Will Be Popped from Stack?

- R4: 0x4040_0000
- PC: 0xFFFF_FFF9

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



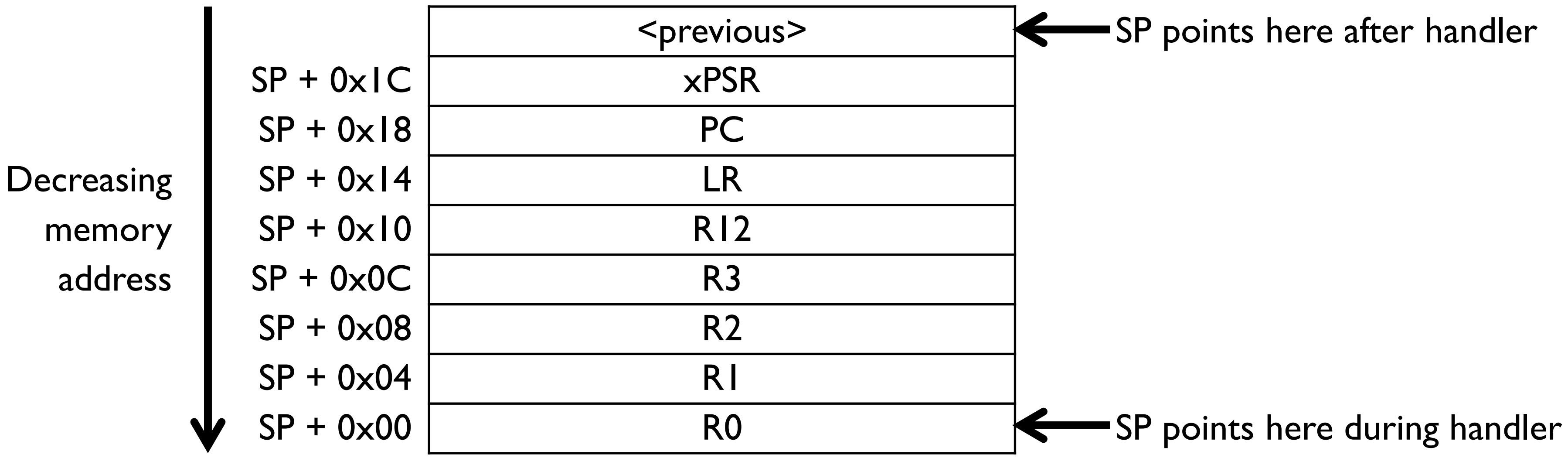


2. Select Stack, Restore Context

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

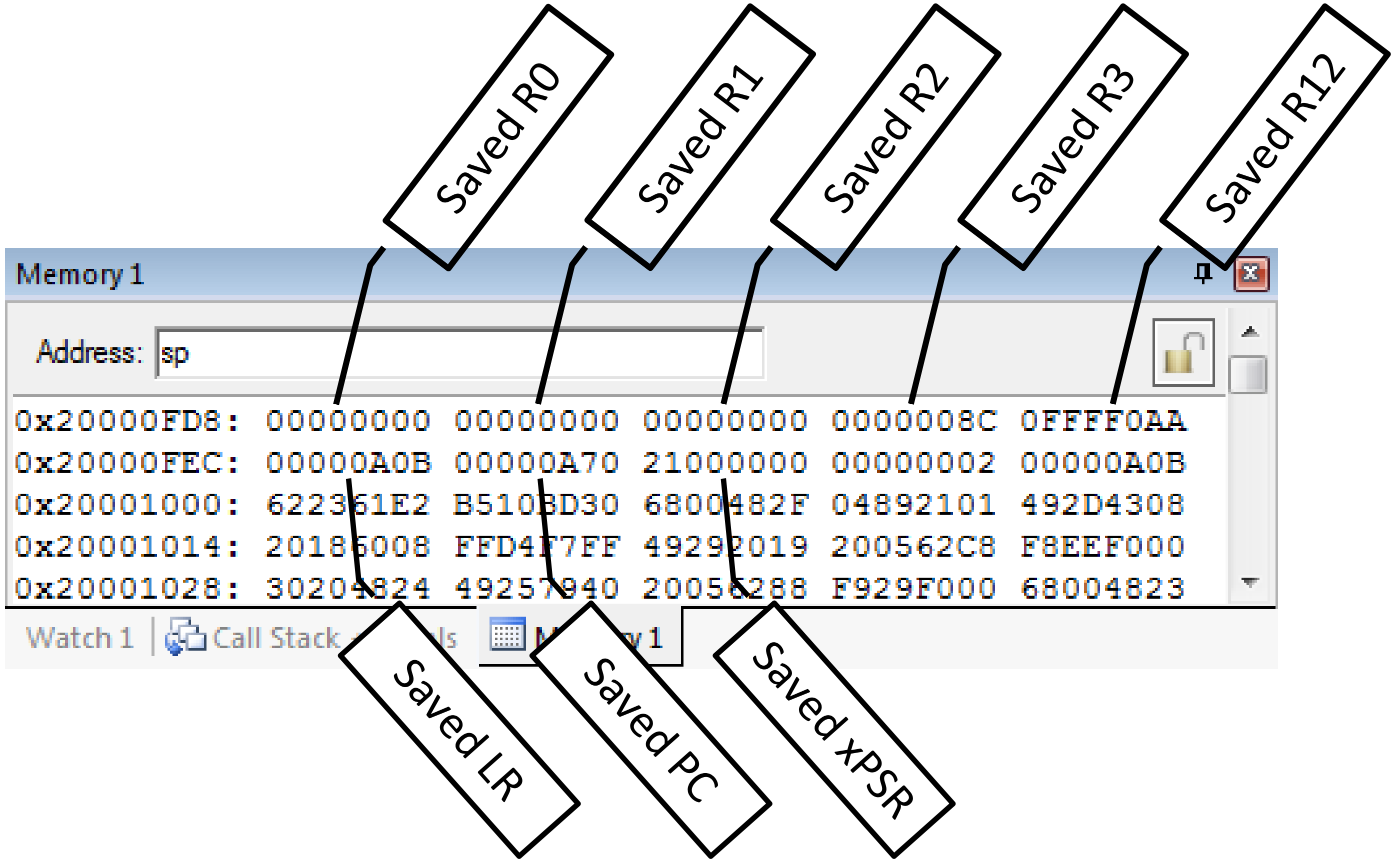
EXC_RETURN	Return Stack	Description
0xFFFF_FFF1	0 (MSP)	Return to exception handler with MSP
0xFFFF_FFF9	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	1 (PSP)	Return to thread with PSP

- Pop the registers from that stack



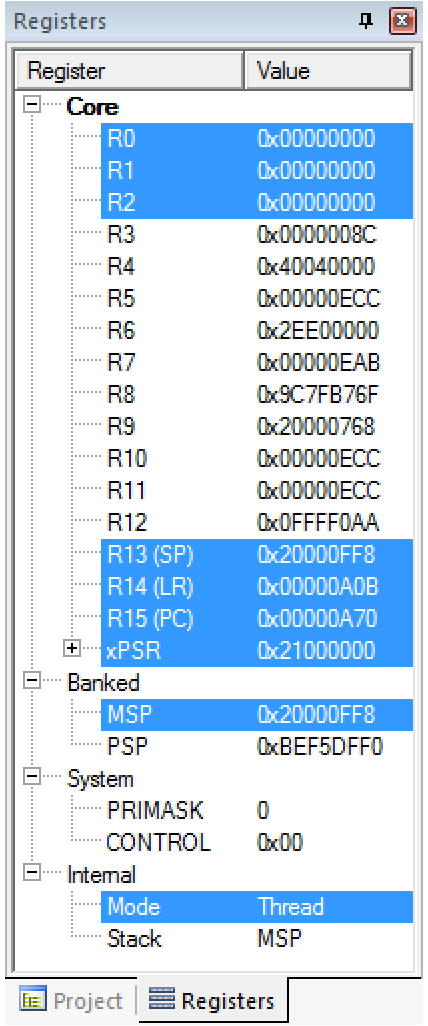
Example

- 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

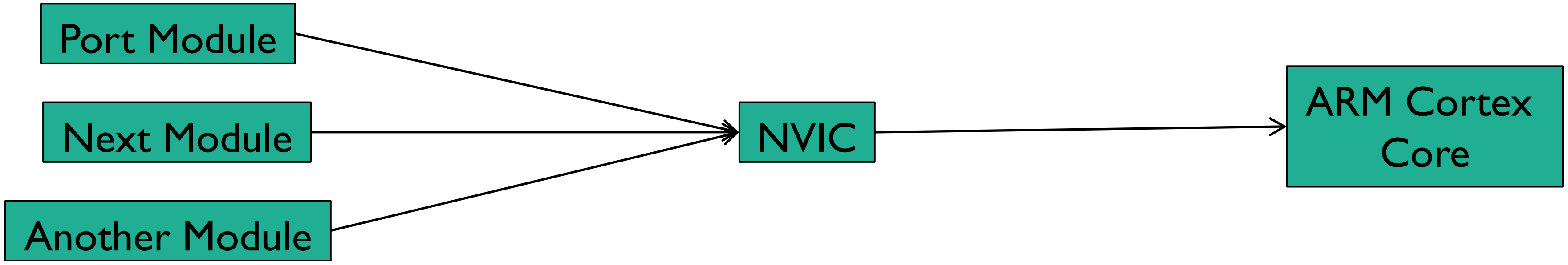


PROCESSOR CORE INTERRUPTS

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

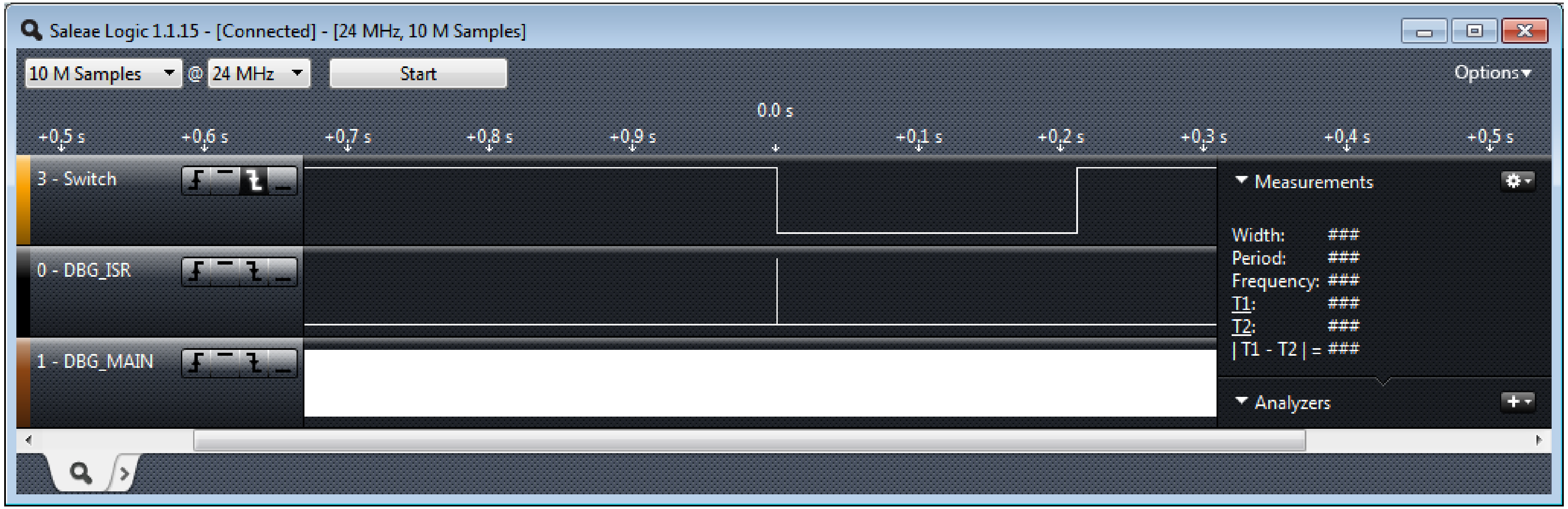
- Exceptions are prioritized to order the response simultaneous requests (smaller number = higher priority)
- Priorities of some exceptions are fixed
  - Reset: -3, highest priority
  - NMI: -2
  - Hard Fault: -1
- Priorities of other (peripheral) exceptions are adjustable
  - Value is stored in the 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
    - $C_{IntResponseOvhd}$: 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_{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.
  - $F_{Max\_Int} = F_{CPU}/(C_{ISR} + C_{Overhead})$
  - 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
  - $U_{Int} = 100\%*F_{Int}*(C_{ISR}+C_{Overhead})/F_{CPU}$
  - CPU looks like it's running the other code with CPU clock speed of $(1-U_{Int})*F_{CPU}$

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 through read or write, is vulnerable to race conditions

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 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.

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);
}
```


ARM

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()`