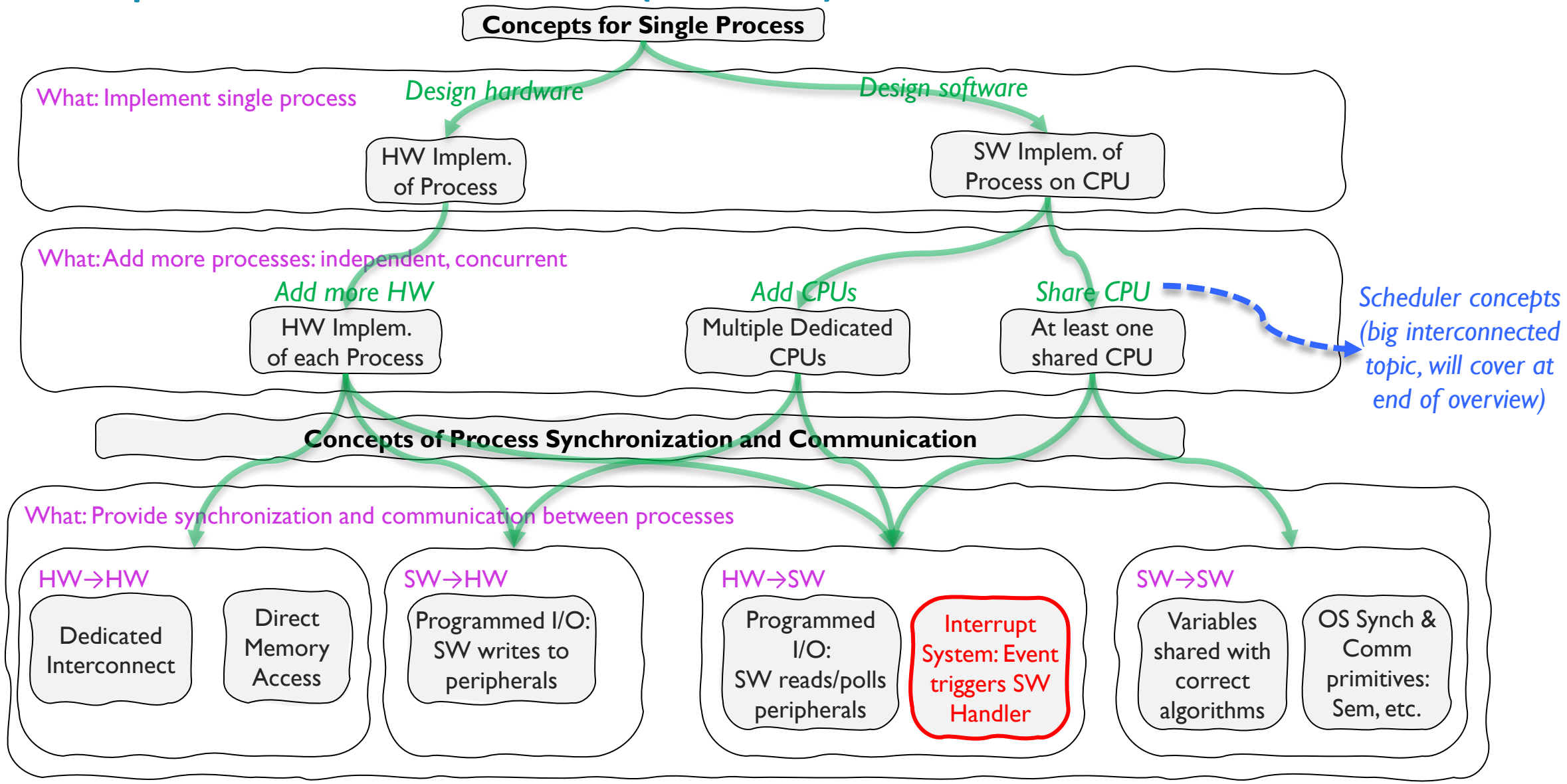
NC STATE UNIVERSITY

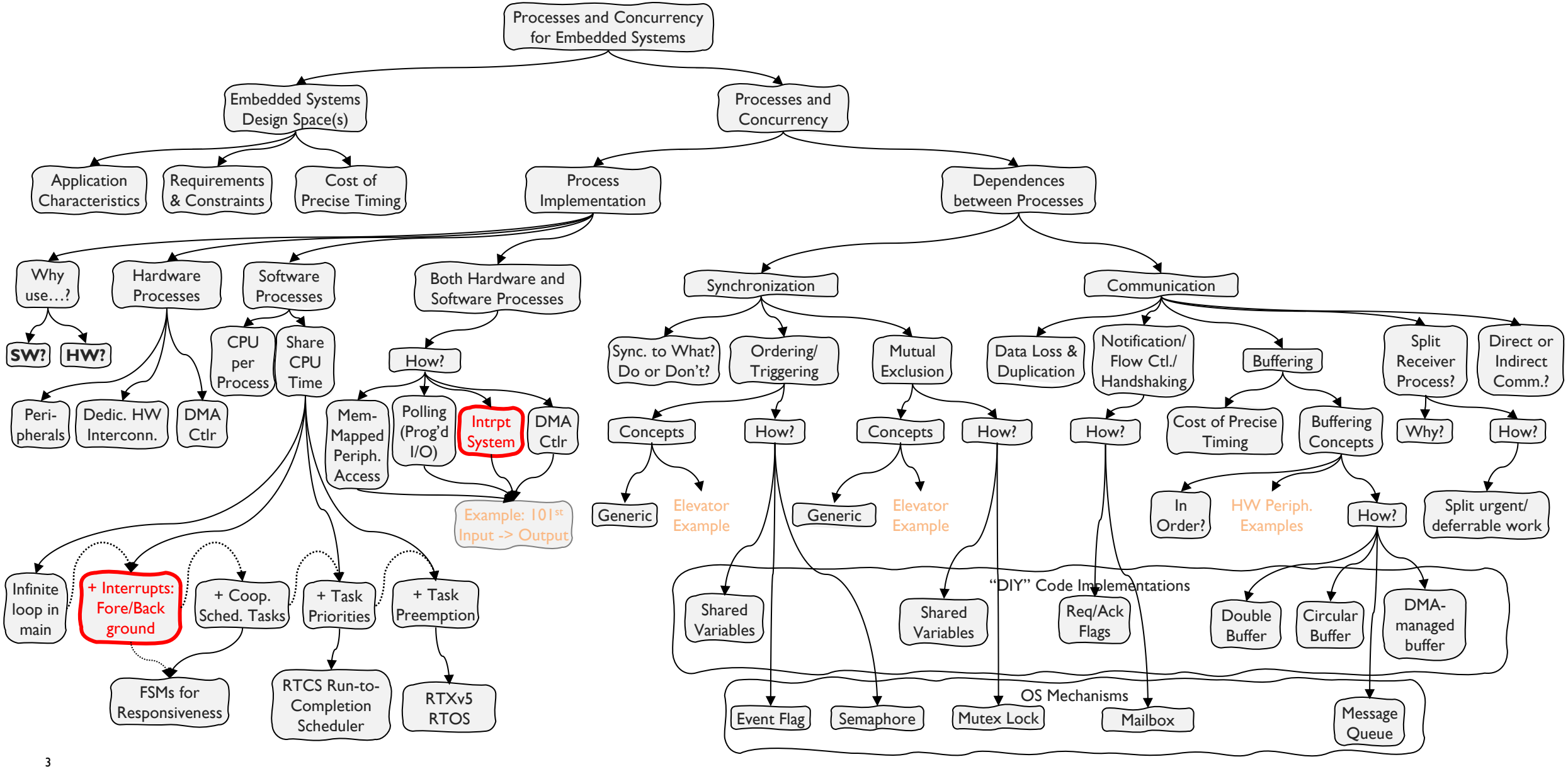
# Cortex-M0+ Interrupts and Exceptions CPU Activities (Level 3)

7/30/2025

# Interrupts: Where Are We (view 1)?

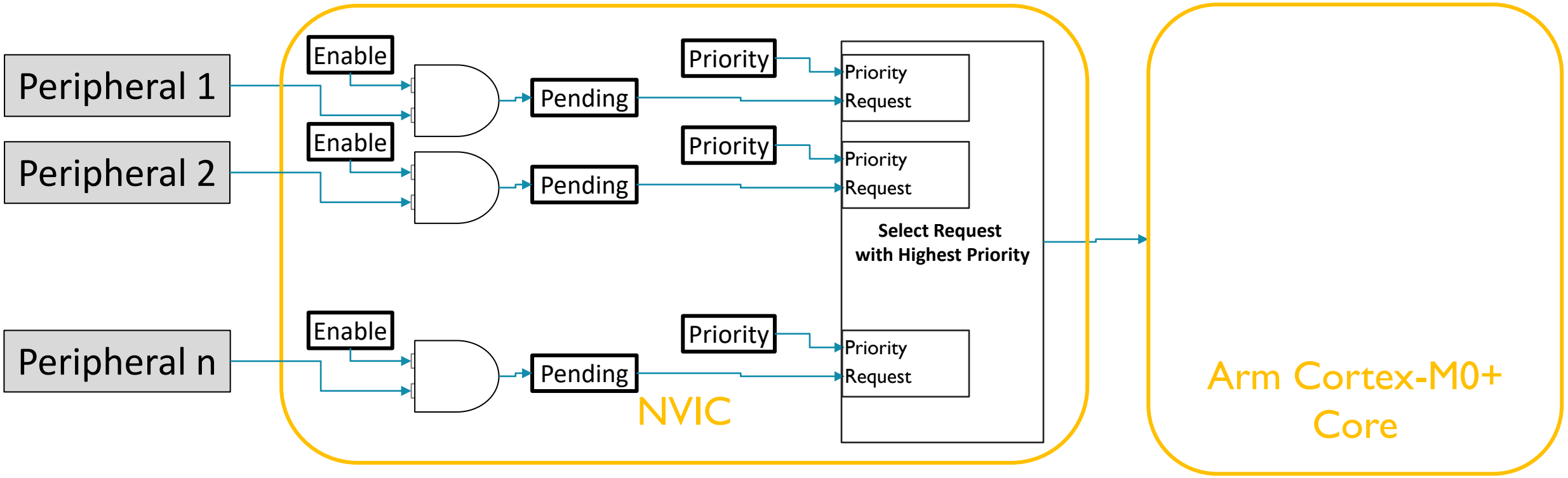


# Interrupts: Where Are We (view 2)?



# CORTEX-M0+ INTERRUPT SUPPORT: NVIC AND PM DETAILS

# Nested Vectored Interrupt Controller (NVIC)




- NVIC manages and prioritizes interrupts and exceptions for CPU core

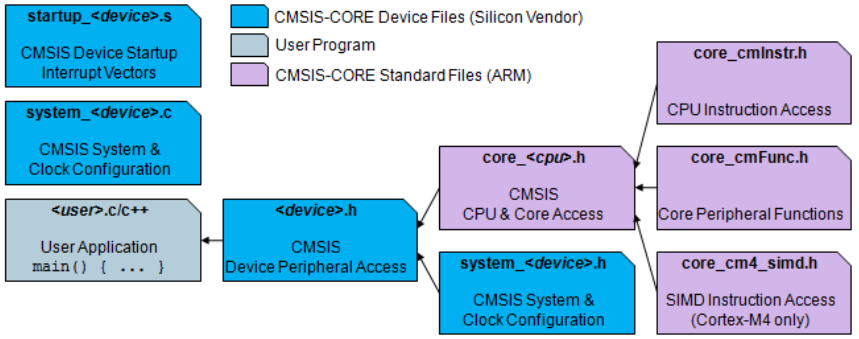
- Register fields in NVIC
  - Enable - Allows interrupt to be recognized. 1 bit
  - Pending - Interrupt requested, not yet serviced. 1 bit
  - Priority - allows program to prioritize response if both interrupts are requested simultaneously

# Details

- Many sources
  - CPU (Cortex-M0+)
    - Generates requests for vectors 1-15
  - MCU (KL25Z)
    - Generates requests for vectors 16-47
- Exception/Interrupt Priorities
  - Three are ***fixed priority***
    - Reset: -3, highest priority
    - NMI: -2
    - Hard Fault: -1
  - Others have ***programmable*** priority
    - Cortex-M0+ NVIC supports four levels
      - Two most-significant bits of byte register, other bits read as zero
      - So, 0, 1, 2, 3 translated to 0x00, 0x40, 0x80, 0xC0
    - Cortex-M3, M4, etc. support more levels

| Vector Address | Vector # | IRQ | Description |
|---|---|---|---|
| 0x0000_0004 | 1 | | CPU Reset |
| 0x0000_0008 | 2 | | NMI: Non-maskable interrupt |
| 0x0000_000C | 3 | | Hard fault error |
| 0x0000_002C | 11 | | SVCall to supervisor with SVC instruction |
| 0x0000_0038 | 14 | | PendSV: System-level service request |
| 0x0000_003C | 15 | | SysTick: System timer tick |
| 0x0000_0040, 44, 48, 4C | 16-19 | 0-3 | Direct Memory Access Controller |
| 0x0000_0058 | 22 | 6 | Power Management Controller |
| 0x0000_005C | 23 | 7 | Low Leakage Wake Up |
| 0x0000_0060, 64 | 24-25 | 8-9 | I2C Communications |
| 0x0000_0068, 6C | 26-27 | 10-11 | SPI Communications |
| 0x0000_0070, 74, 78 | 28-30 | 12-14 | UART Communications |
| 0x0000_007C | 31 | 15 | Analog to Digital Converter |
| 0x0000_0080 | 32 | 16 | Comparator |
| 0x0000_0084, 88, 8C | 33-35 | 17-19 | Timers and Pulse-Width Modulation |
| 0x0000_0090, 94 | 36-37 | 20-21 | Real-Time Clock alarm and seconds |
| 0x0000_0098 | 38 | 22 | Programmable Interval Timer |
| 0x0000_00A0 | 40 | 24 | USB On-The-Go |
| 0x0000_00A4 | 41 | 25 | Digital to Analog Converter |
| 0x0000_00A8 | 42 | 26 | Touch Sense Interface |
| 0x0000_00AC | 43 | 27 | Main Clock Generator |
| 0x0000_00B0 | 44 | 28 | Low Power Timer |
| 0x0000_00B8 | 46 | 30 | Port Control Module, Port A pin detect |
| 0x0000_00BC | 47 | 31 | Port Control Module, Port D pin detect |

# CMSIS Access Functions for NVIC and PM




NVIC_EnableIRQ(IRQnum)

NVIC_DisableIRQ(IRQnum)

NVIC_SetPendingIRQ(IRQnum)

NVIC_ClearPendingIRQ(IRQnum)

NVIC_SetPriority(IRQnum, priority)

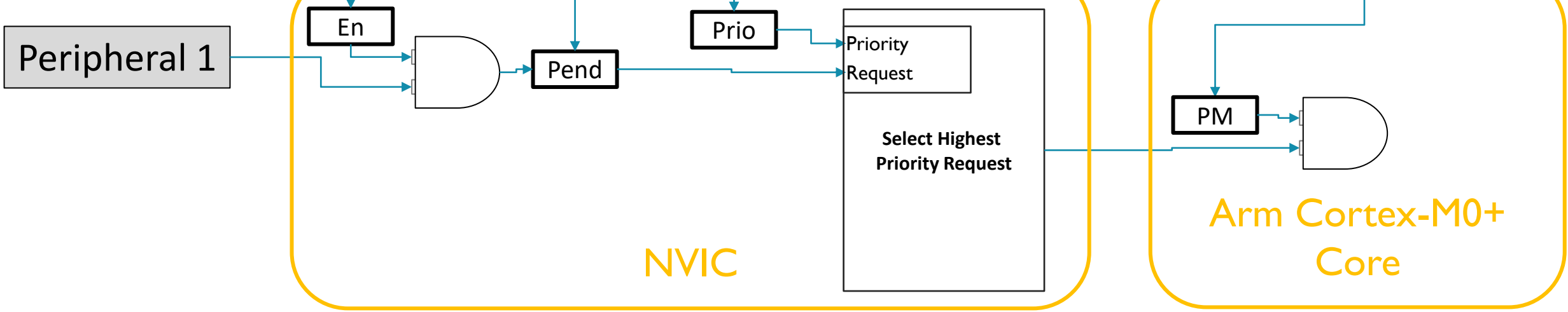
NVIC_GetPriority(IRQnum)

__get_PRIMASK()

__set_PRIMASK(x)

__enable_irq()

__disable_irq()



# Priority Masking Bit

- PM bit is NOT saved or restored by hardware exception response
- Implications? We'll see later

# 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)

# NVIC Registers and State

*Details*

| Bits | 31:30 | 29:24 | 23:22 | 21:16 | 15:14 | 13:8 | 7:6 | 5:0 |
|---|---|---|---|---|---|---|---|---|
| IPR0 | IRQ3 | reserved | IRQ2 | reserved | IRQ1 | reserved | IRQ0 | reserved |
| IPR1 | IRQ7 | reserved | IRQ6 | reserved | IRQ5 | reserved | IRQ4 | reserved |
| IPR2 | IRQ11 | reserved | IRQ10 | reserved | IRQ9 | reserved | IRQ8 | reserved |
| IPR3 | IRQ15 | reserved | IRQ14 | reserved | IRQ13 | reserved | IRQ12 | reserved |
| IPR4 | IRQ19 | reserved | IRQ18 | reserved | IRQ17 | reserved | IRQ16 | reserved |
| IPR5 | IRQ23 | reserved | IRQ22 | reserved | IRQ21 | reserved | IRQ20 | reserved |
| IPR6 | IRQ27 | reserved | IRQ26 | reserved | IRQ25 | reserved | IRQ24 | reserved |
| IPR7 | IRQ31 | reserved | IRQ30 | reserved | IRQ29 | reserved | IRQ28 | reserved |

- Priority - allows program to prioritize response if both interrupts are requested simultaneously
  - IPR0-7 registers: two bits per interrupt source, four interrupt sources per register
  - Set priority to 0 (highest priority), 1, 2 or 3 (lowest)
  - CMSIS: NVIC_SetPriority(IRQnum, priority)

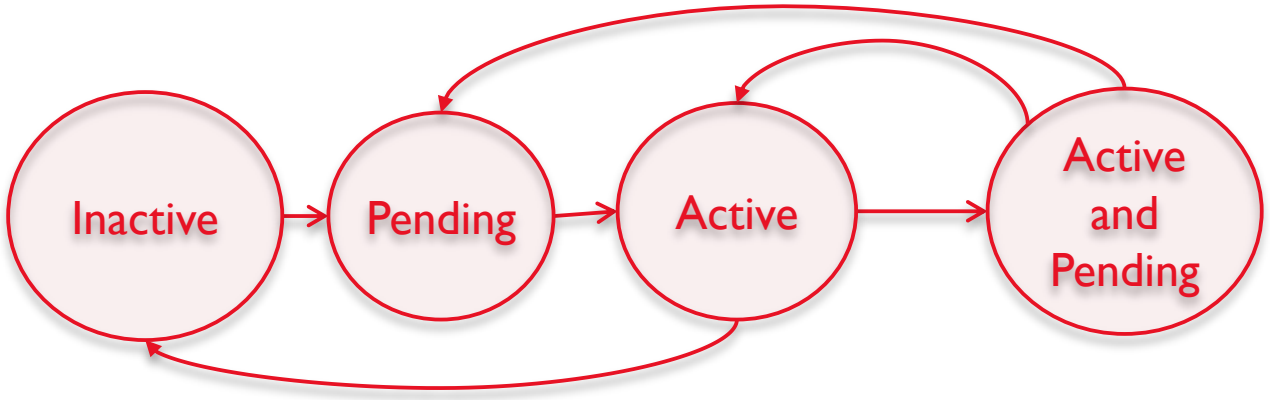
# Prioritization

- 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

# DETAILS:
# ENTERING AN EXCEPTION HANDLER

# Exception Processing States

- Inactive
- Pending
- Active
- Active and Pending



# What If …? Special Cases of Prioritization

- New exception requested while current handler is executing?
  - New priority **higher** than current priority?
    - New exception handler preempts current handler
    - Stacks registers, Executes new handler, unstacks registers
    - Resumes current handler
  - New priority **lower** than or equal to current priority?
    - New exception held in **pending state**
    - Current handler continues and completes execution
    - New exception handler executes
    - Registers unstacked
- Simultaneous exception requests with **same priority**?
  - Lowest exception type number is serviced first
- Special features improve response time, covered later
  - Late Arrival
  - Tail Chaining

# CPU's Hardwired Exception Processing

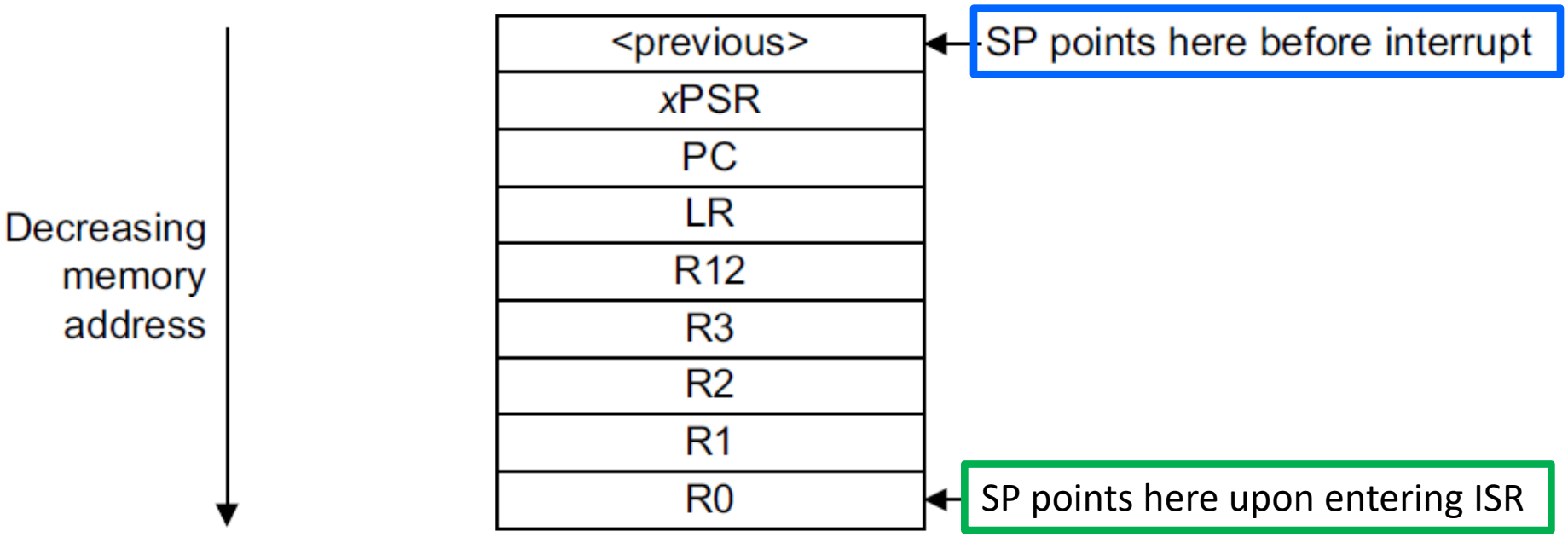
1. Finish current instruction (except for lengthy instructions)
2. Push context (8 32-bit words) onto current stack (MSP or PSP)
   - xPSR, Return address, LR (R14), R12, R3, R2, R1, R0
3. Switch to handler/privileged mode, use MSP
4. Load PC with address of exception handler
5. Load LR with EXC_RETURN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

Usually 15 cycles from exception request to execution of first instruction in handler *(assuming fast memory without wait states)*

# 1. Finish Current Instruction

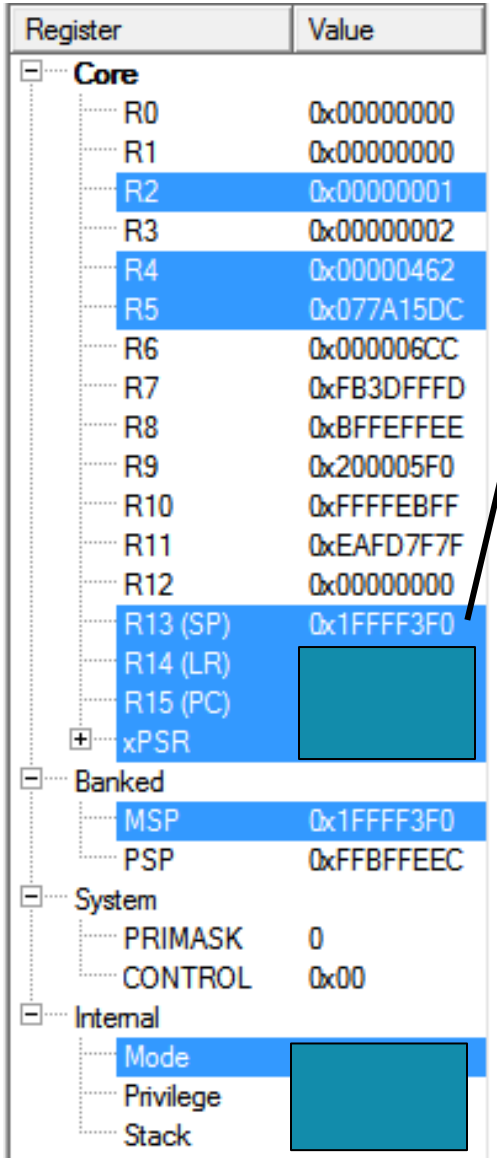
- Most instructions are short and finish quickly
- Some instructions may take many cycles to execute
  - Load Multiple (LDM), Store Multiple (STM), Push, Pop, MULS (32 cycles for some CPU core implementations)
- This will delay interrupt response significantly
- If one of these is executing when the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

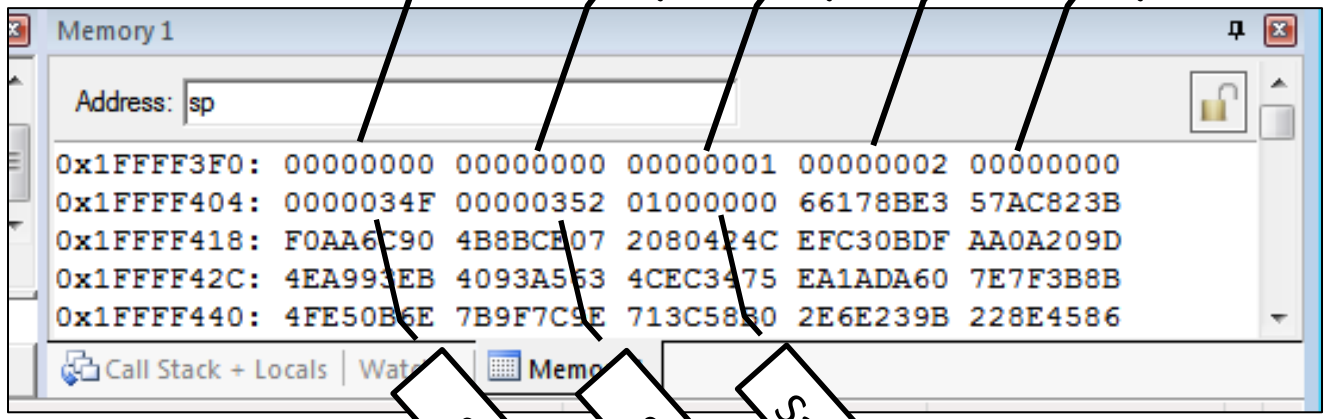



- Two SPs: Main (MSP), process (PSP)
- Which is active depends on operating mode, CONTROL register bit 1
- Stack
  - Full: SP points to a location currently holding data
  - Descending: grows toward smaller addresses

# Context Saved on Stack




SP value is reduced since registers have been pushed onto stack




Saved R0

Saved R1

Saved R2

Saved R3

Saved R12

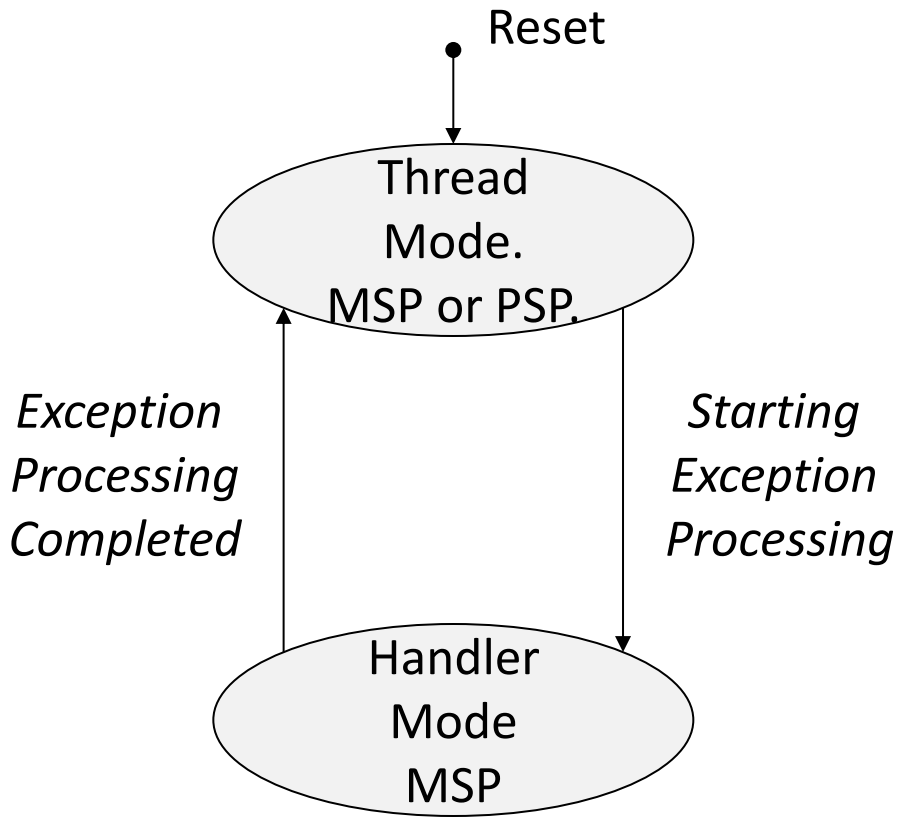
Saved LR

Saved PC

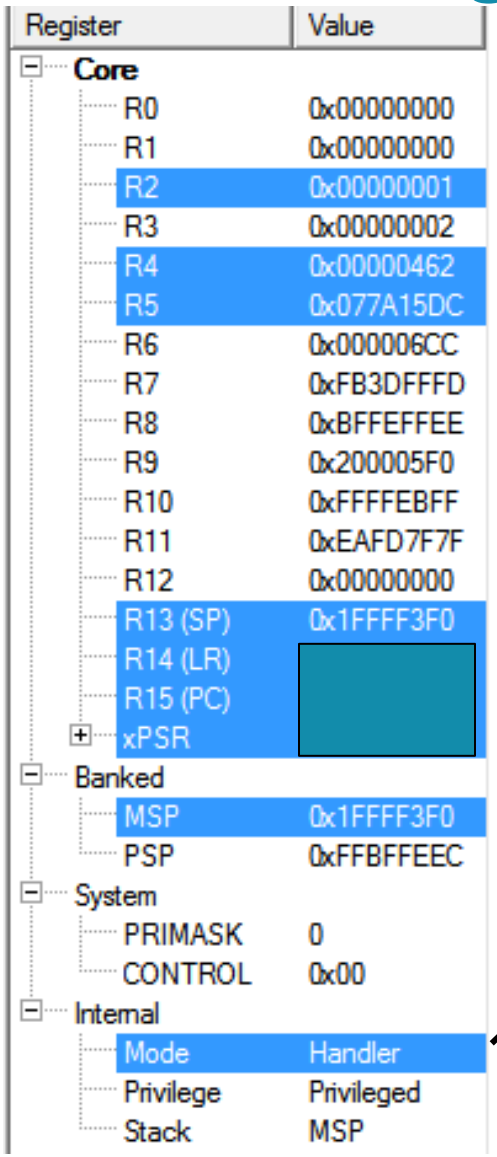
Saved xPSR

# 3. Switch to Handler/Privileged Mode

- Handler mode always uses Main SP

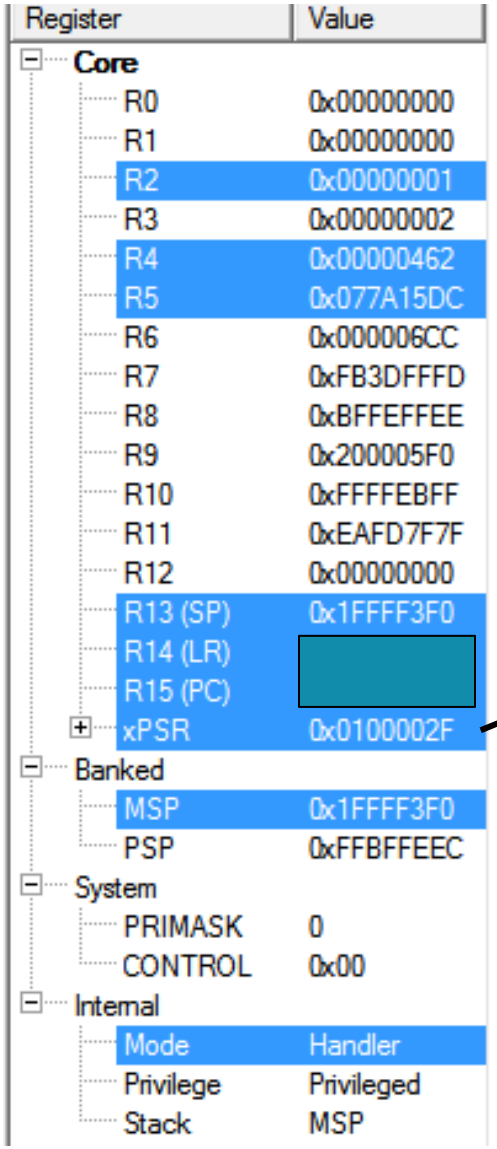


# Handler and Privileged Mode



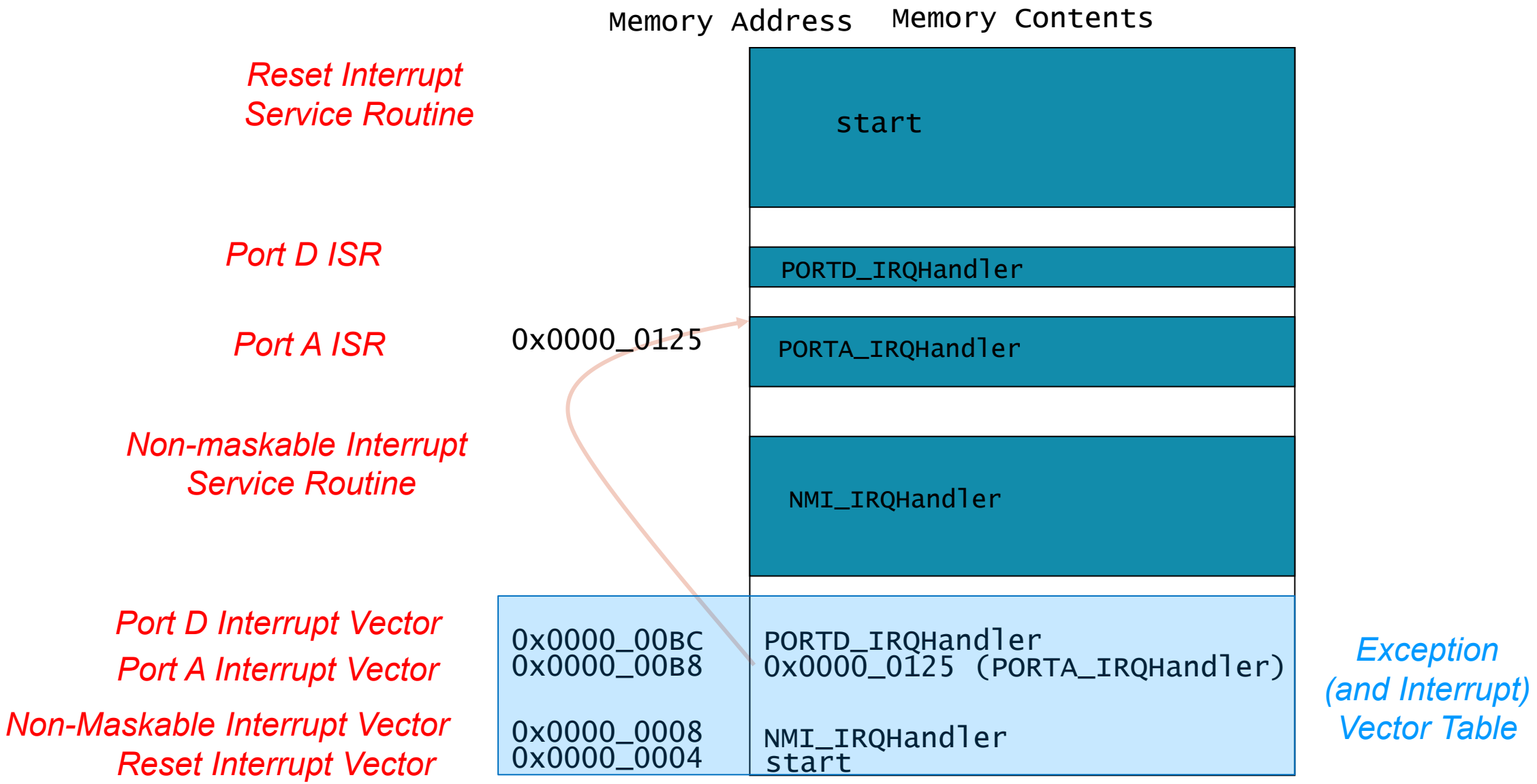

Mode changed to Handler. Was already using MSP and in Privileged mode

# Update IPSR with Exception Number




PORTD_IRQ is Exception number 0x2F (interrupt number + 0x10)

# 4. Load PC With Address Of Exception Handler



# Can Examine Vector Table With Debugger

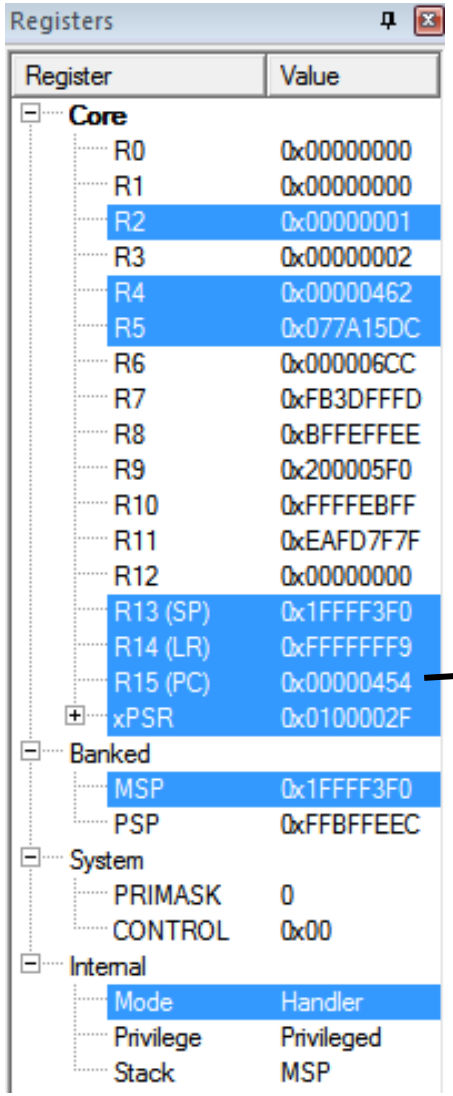
| Exception number | IRQ number | Vector | Offset |
|---|---|---|---|
| 16+n | n | IRQn | 0x40+4n |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |
| 18 | 2 | IRQ2 | 0x48 |
| 17 | 1 | IRQ1 | 0x44 |
| 16 | 0 | IRQ0 | 0x40 |
| 15 | –1 | SysTick, if implemented | 0x3C |
| 14 | –2 | PendSV | 0x38 |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

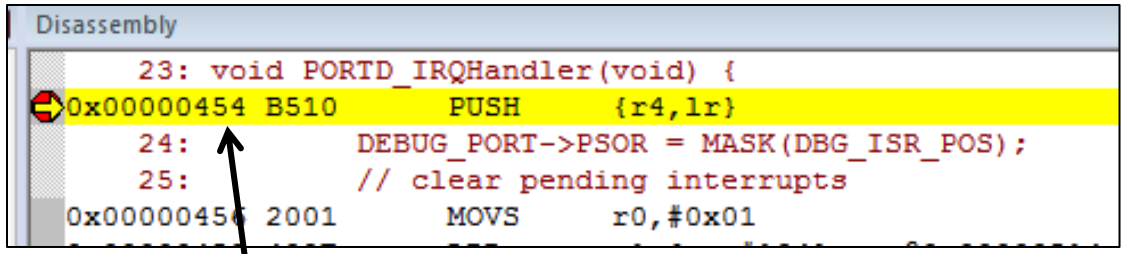
Disassembly

```
0x000000B0 00E7    DCW    0x00E7
0x000000B2 0000    DCW    0x0000
0x000000B4 00E7    DCW    0x00E7
0x000000B6 0000    DCW    0x0000
0x000000B8 00E7    DCW    0x00E7
0x000000BA 0000    DCW    0x0000
0x000000BC 0455    DCW    0x0455
0x000000BE 0000    DCW    0x0000
```

- PORTD ISR is IRQ #31 (0x1F), so vector to handler begins at 0x40+4*0x1F = 0xBC
- Why is the vector odd? 0x0000_0455
- LSB of address indicates that handler uses Thumb code

# Upon Entry to Handler



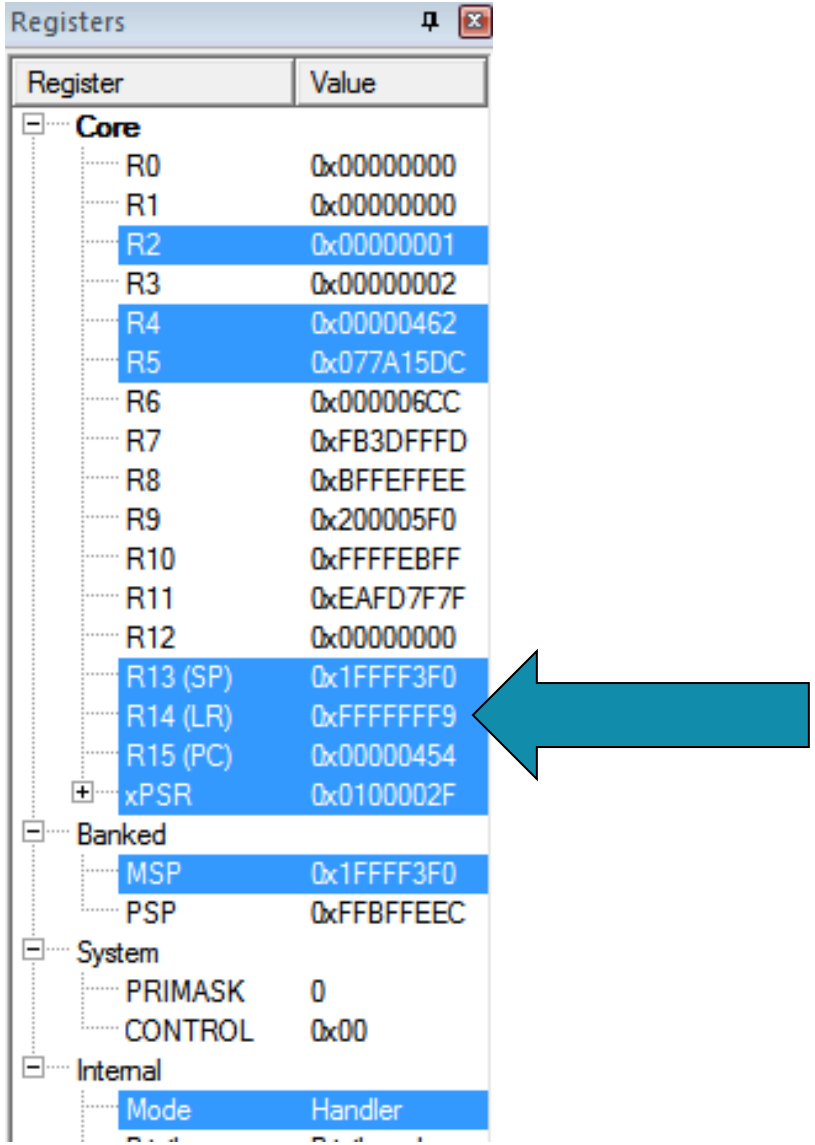

PC has been loaded with start address of handler

# 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

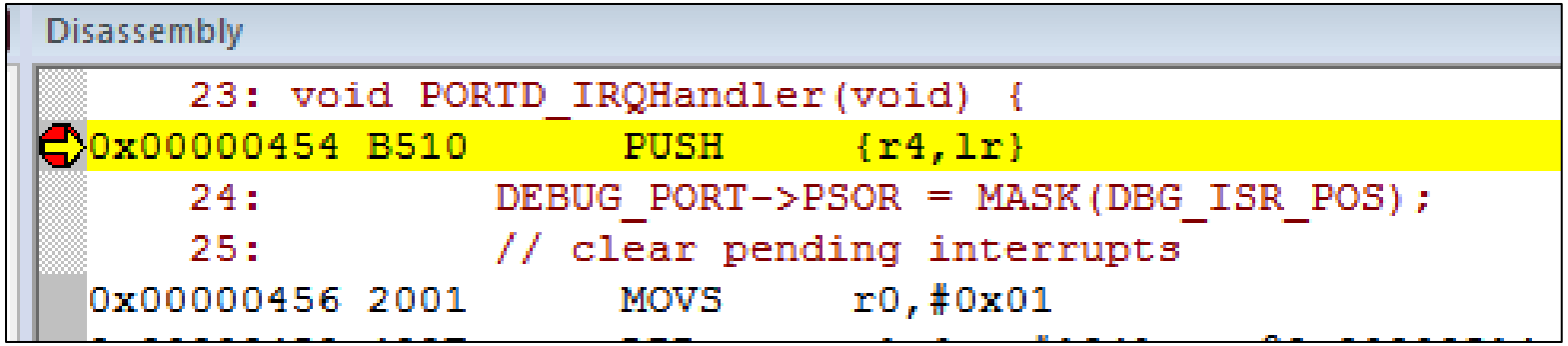
# Updated LR With EXC_RETURN Code



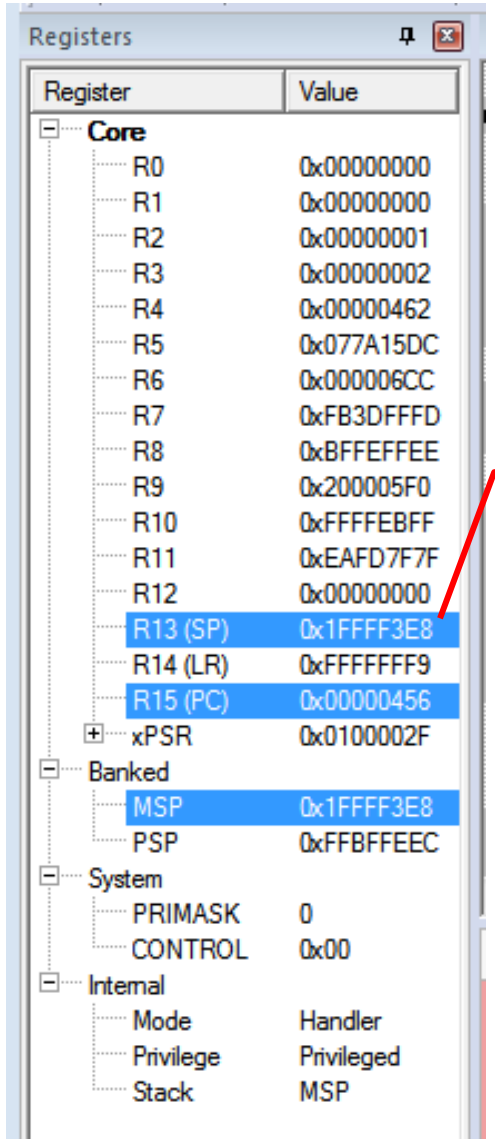
# 6. Start Executing Exception Handler

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

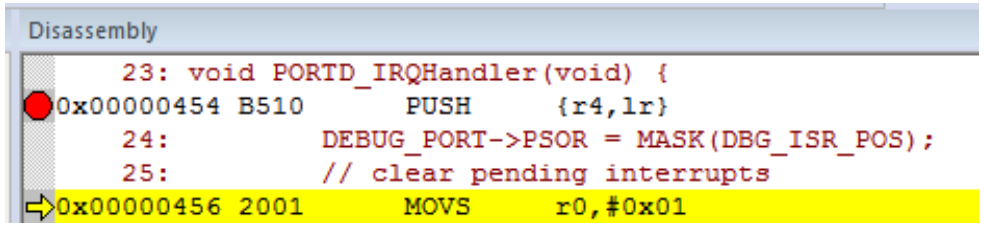
- Exception handler may save additional registers on stack
  - E.g. if handler may call a subroutine, LR and R4 must be saved



# Example: Handler Instructions May Save More Context




*SP reduced since registers were pushed onto stack*




*Registers saved by software*

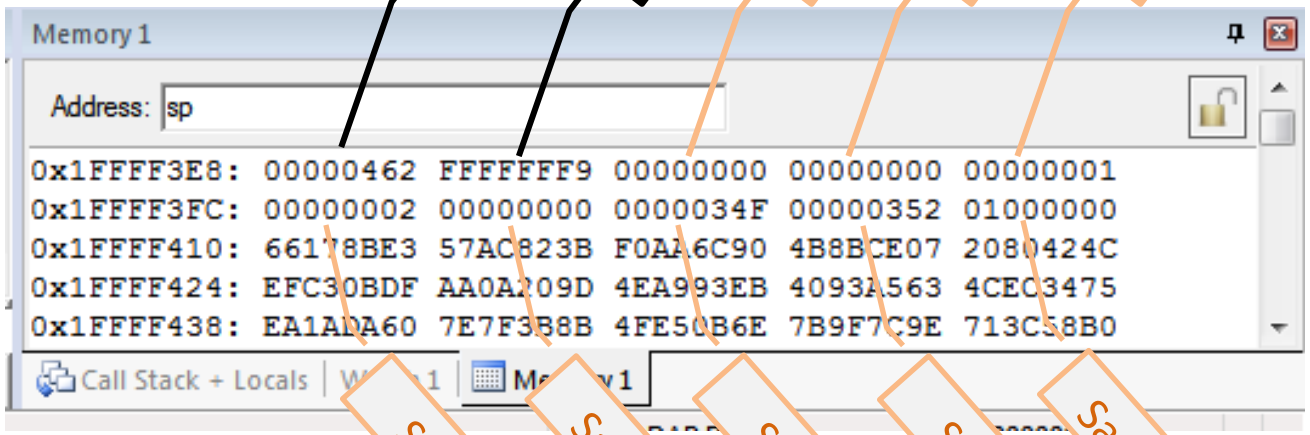
Saved R4

Saved LR

Saved R0

Saved R1

Saved R2




Saved R3

Saved R12

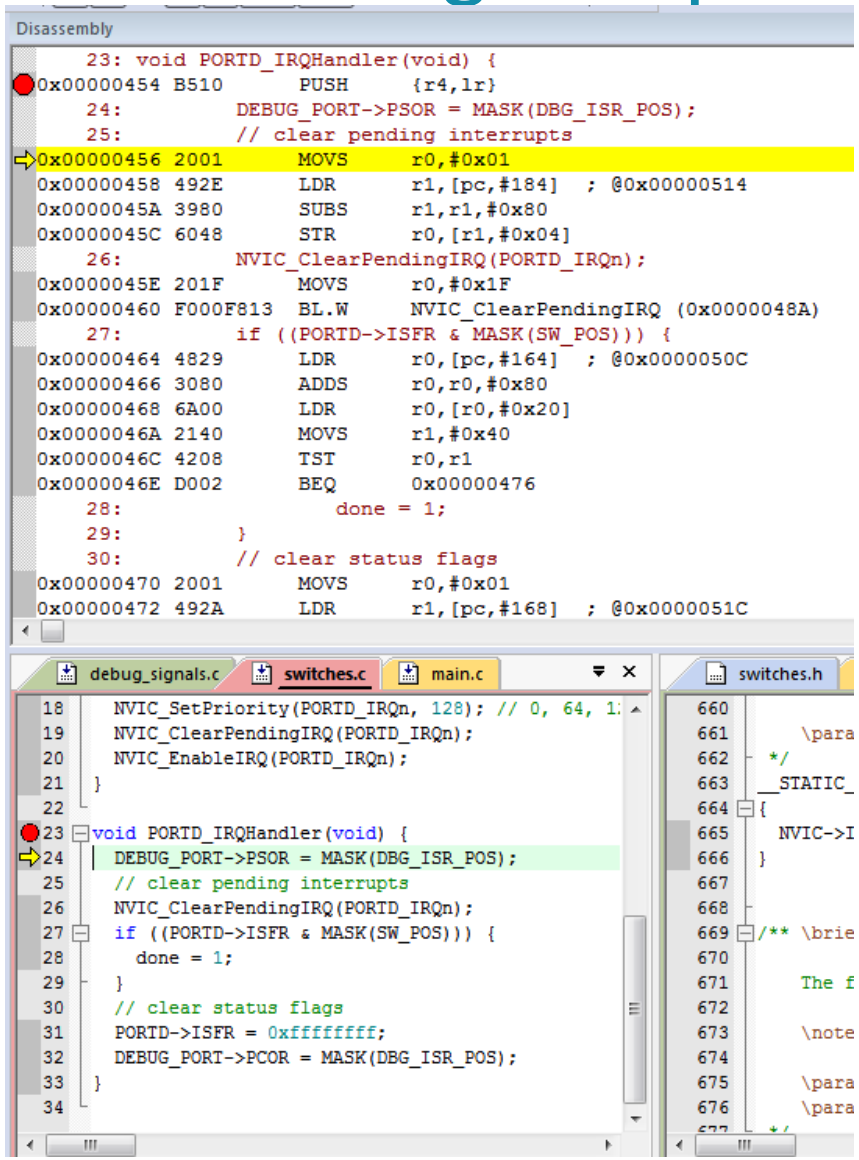
Saved LR

Saved PC

Saved xPSR

*Registers saved by hardware*

# Continue Executing Exception Handler




- Execute user code in handler

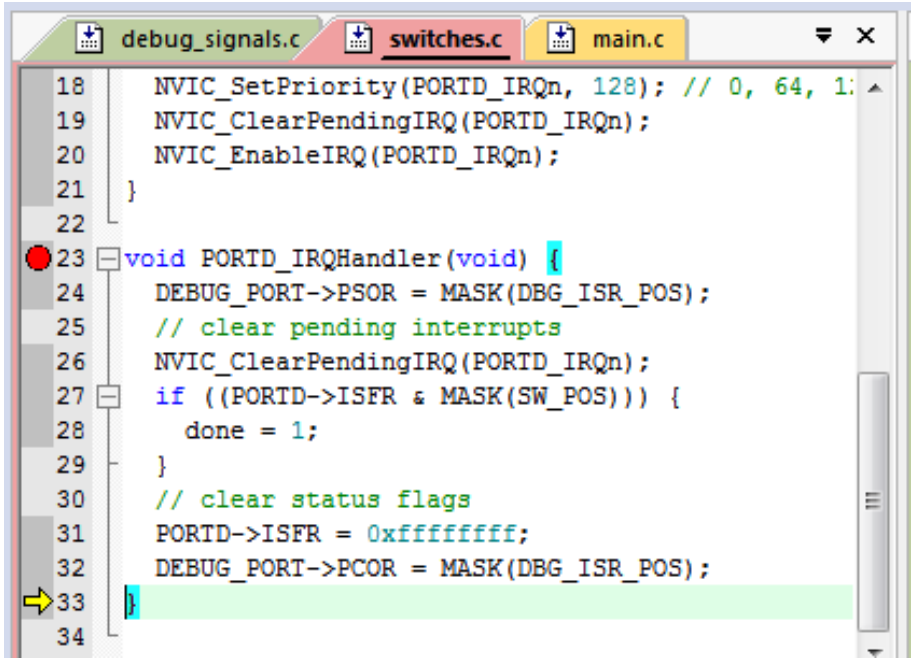
# DETAILS:
# EXITING AN EXCEPTION HANDLER

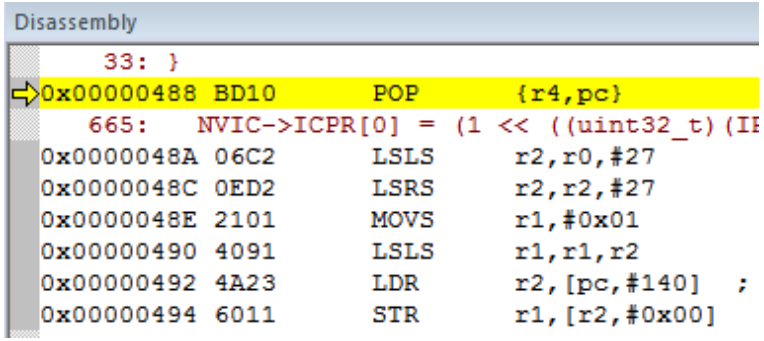
# Exiting an Exception Handler

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

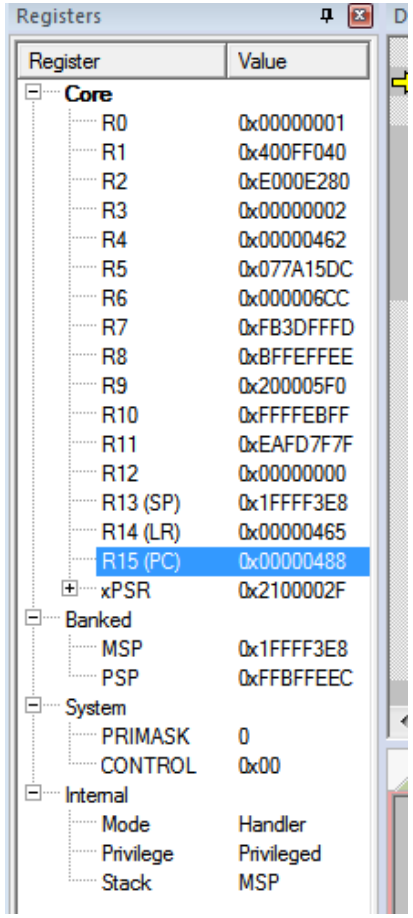
- No “return from interrupt” instruction
- Use regular instruction instead
  - BX LR - Branch to address in LR by loading PC with LR contents
  - POP …, PC - Pop address from stack into PC
- … with a special value EXC_RETURN loaded into the PC to trigger exception handling processing
  - BX LR used if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

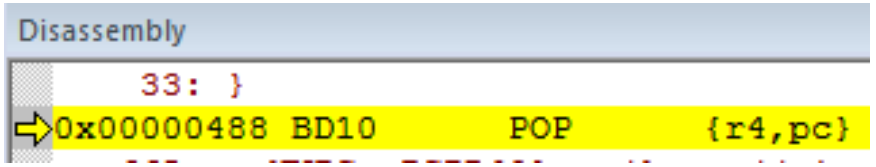


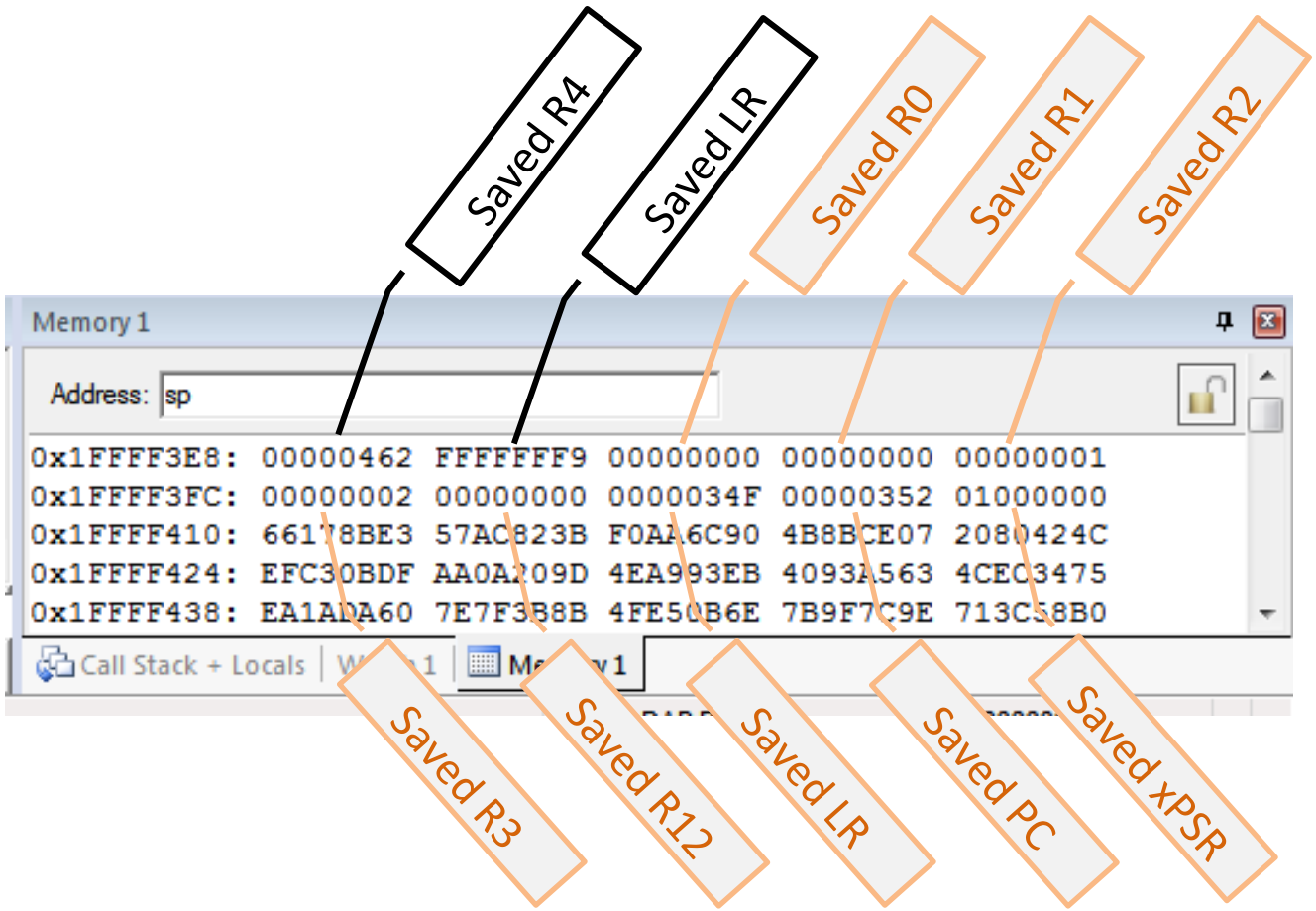



# What Will Be Popped from Stack?

- R4: 0x0000_0462
- PC: 0xFFFF_FFF9



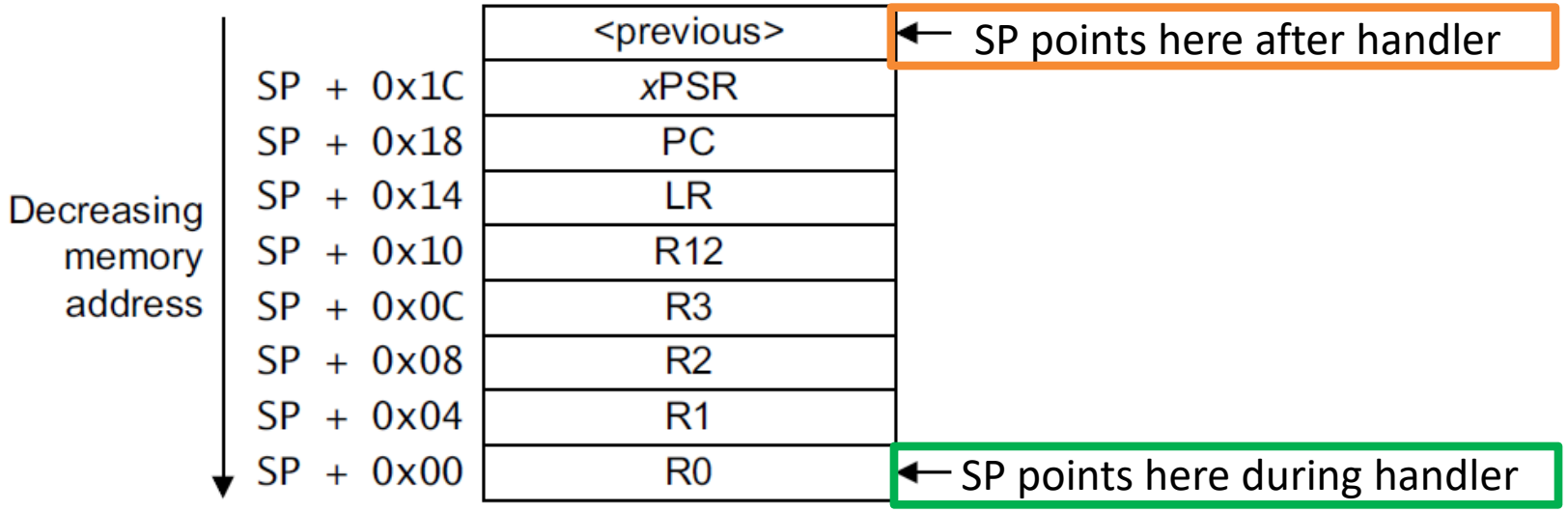



# 2. Select Stack, Restore Context

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

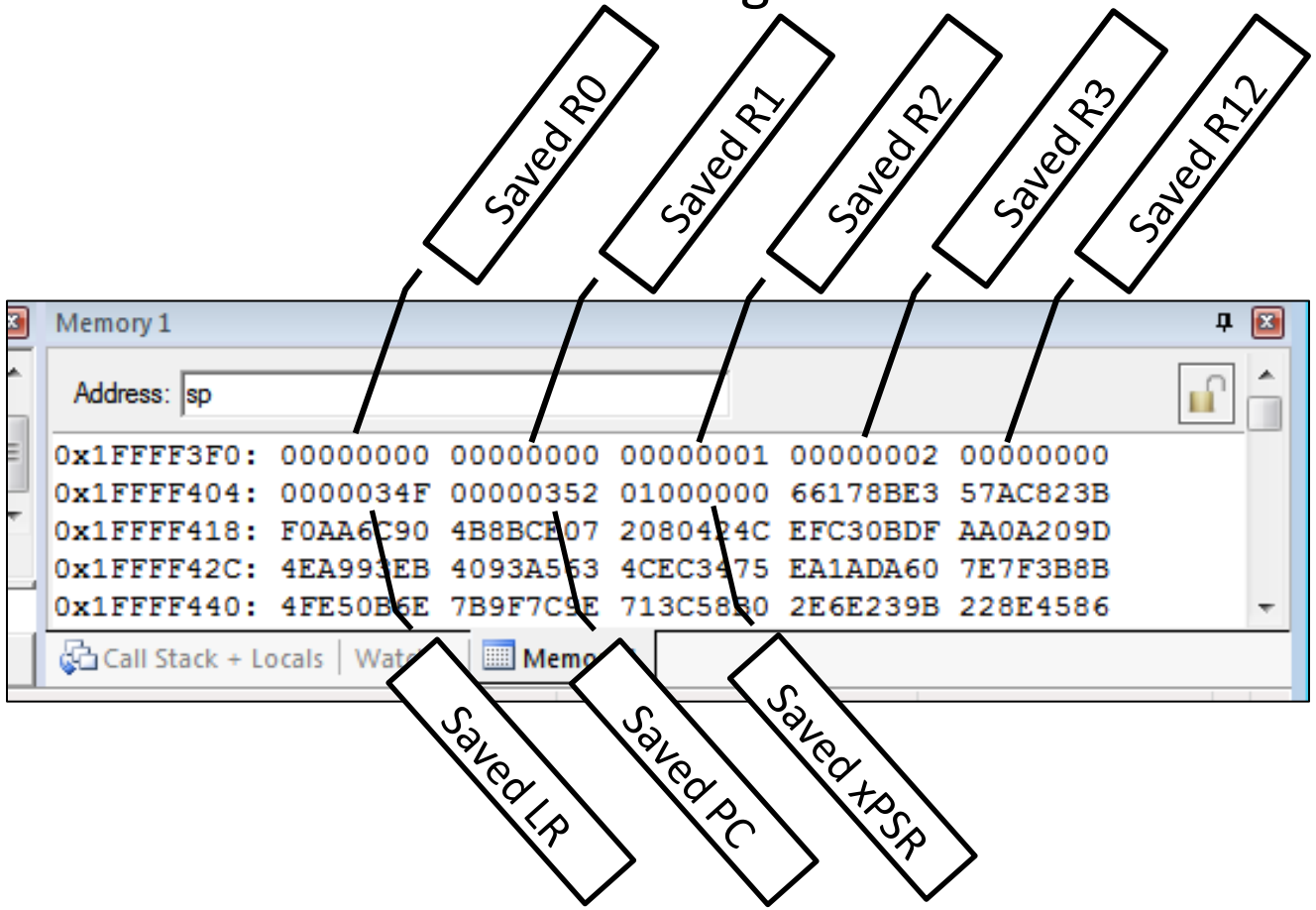
| EXC_RETURN | Return Stack | Description |
|---|---|---|
| 0xFFFF_FFF1 | 0 (MSP) | Return to exception handler with MSP |
| 0xFFFF_FFF9 | 0 (MSP) | Return to thread with MSP |
| 0xFFFF_FFFD | 1 (PSP) | Return to thread with PSP |

- Pop the registers from that stack



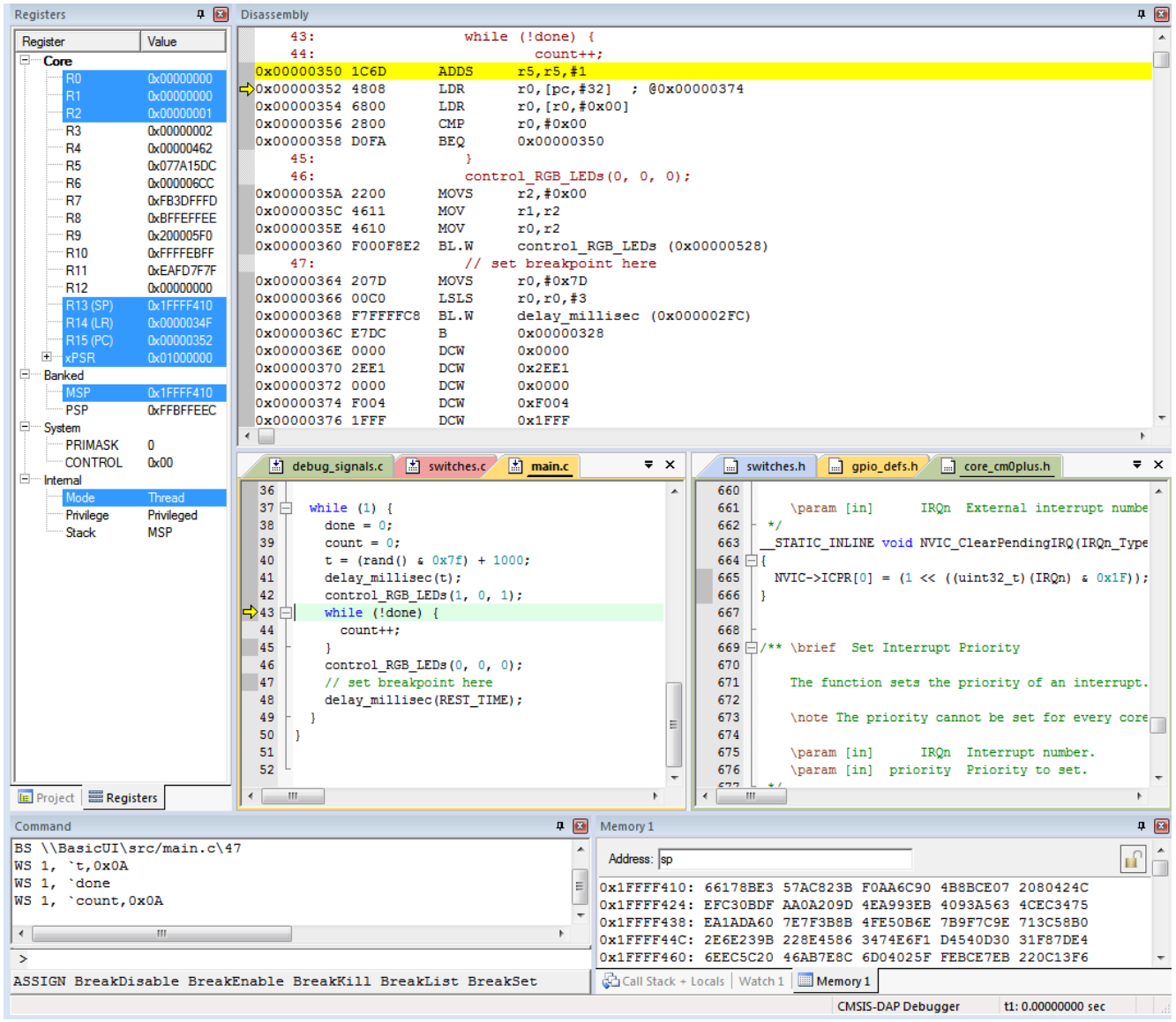
# Example

- 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_0352



# 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
  - TimerVal structure tracks time and days since some reference event
  - TimerVal's fields are updated by periodic 1 Hz timer ISR

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

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

# Example: Checking the Time

- Problem
  - An interrupt at the wrong time will lead to half-updated data in DT
- Failure Case
  - TimerVal is {10, 23, 59, 59} (10th day, 23:59:59)
  - Task code calls GetDateTime(), which starts copying the TimerVal fields to DT: day = 10, hour = 23
  - A timer interrupt occurs, which updates TimerVal to {11, 0, 0, 0}
  - GetDateTime() resumes executing, copying the remaining TimerVal 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
  - (1) requires multiple instructions to read or write (non-atomic access), and
  - (2) 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

```
void GetDateTime(DateTimeType *
DT){
 uint32_t m;

 m = __get_PRIMASK();
 __disable_irq();

 DT->day = TimerVal.day;
 DT->hour = TimerVal.hour;
 DT->minute = TimerVal.minute;
 DT->second = TimerVal.second;
 __set_PRIMASK(m);
}
```

- Avoid disabling preemption if possible
  - Disabling interrupts delays response to all other processing requests
  - Make this time as short as possible (e.g. a few instructions)

# 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*** extern 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()