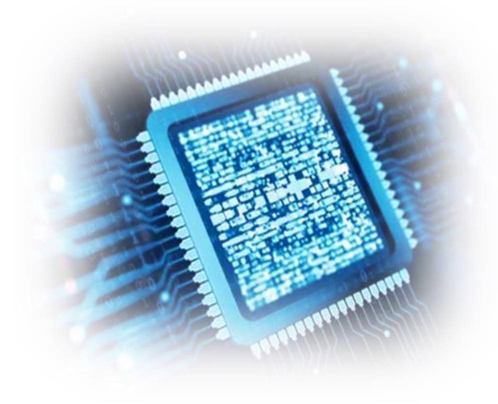


MICROPROCESSOR SYSTEMS

BLG212E

Burak Berk Üstündağ CRN: 11450

Gökhan İnce / Ayşe Yılmazer CRN: 11446

İTÜ Bilgisayar ve Bilişim Fakültesi

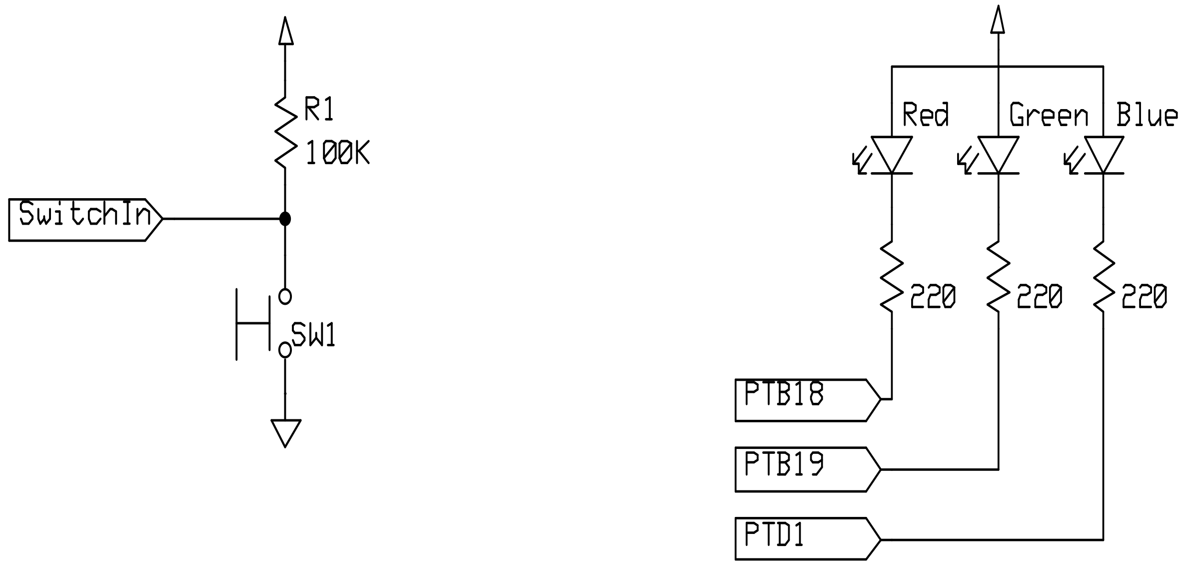
Week 11: Arm Core-M0+ Exceptions and Interrupts

Overview

- Exception and Interrupt Concepts
  - Entering an Exception Handler
  - Exiting an Exception Handler
- Cortex-M0+ Interrupts
  - Using Port Module and External Interrupts
- Program Design with Interrupts
  - Sharing Data Safely Between ISRs and Other Threads
- Sources
  - Cortex M0+ Device Generic User Guide - DUI0662
  - Cortex M0+ Technical Reference Manual - DUI0484

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 later
- Need to add external switch

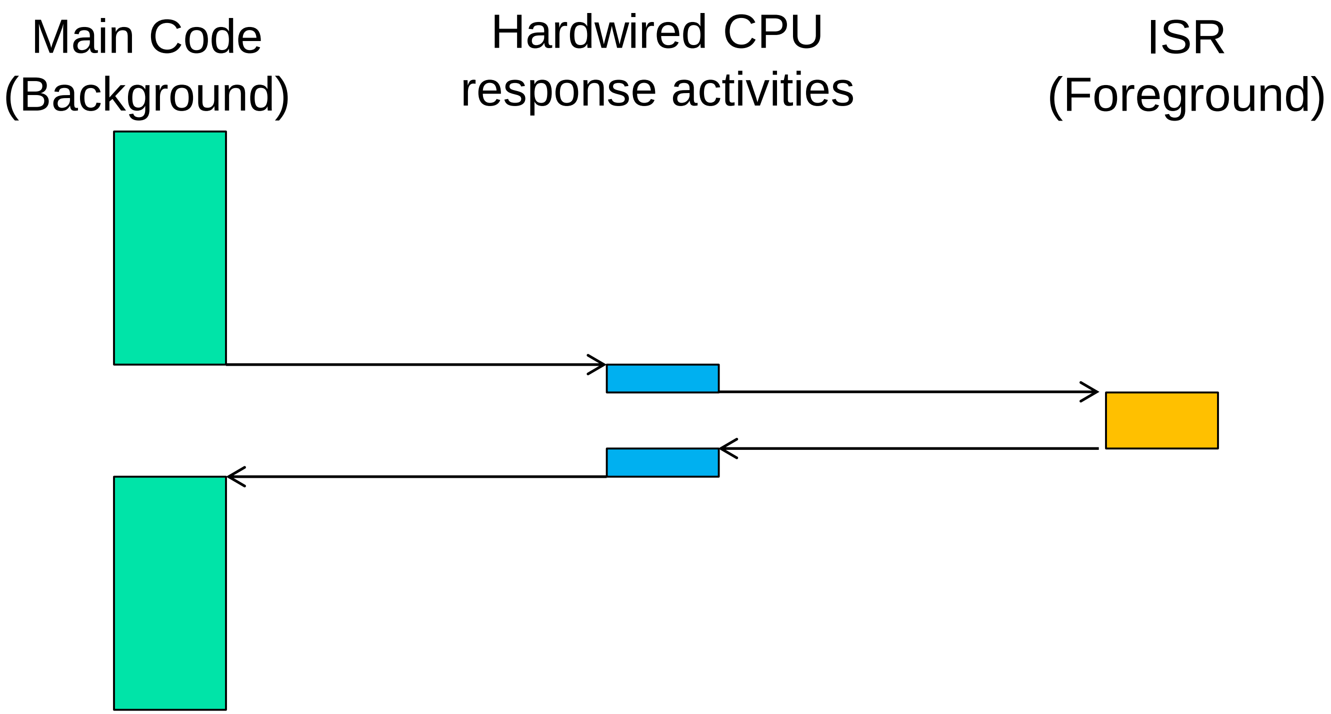
How to Detect Switch is Pressed?

- 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

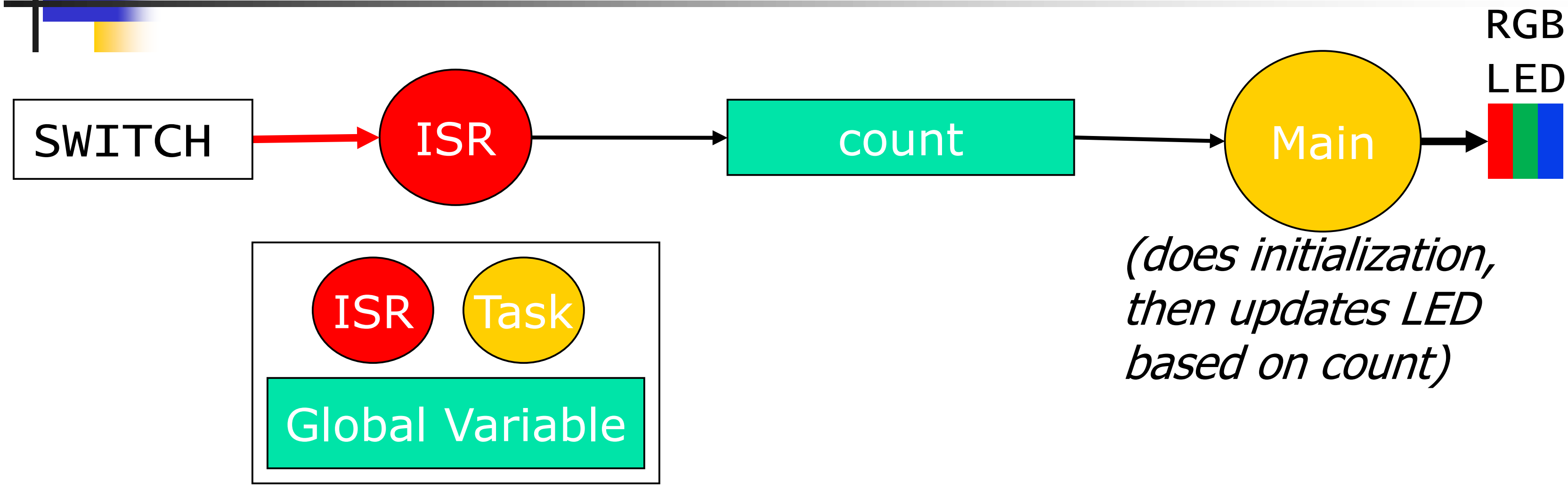


Interrupts

- 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)

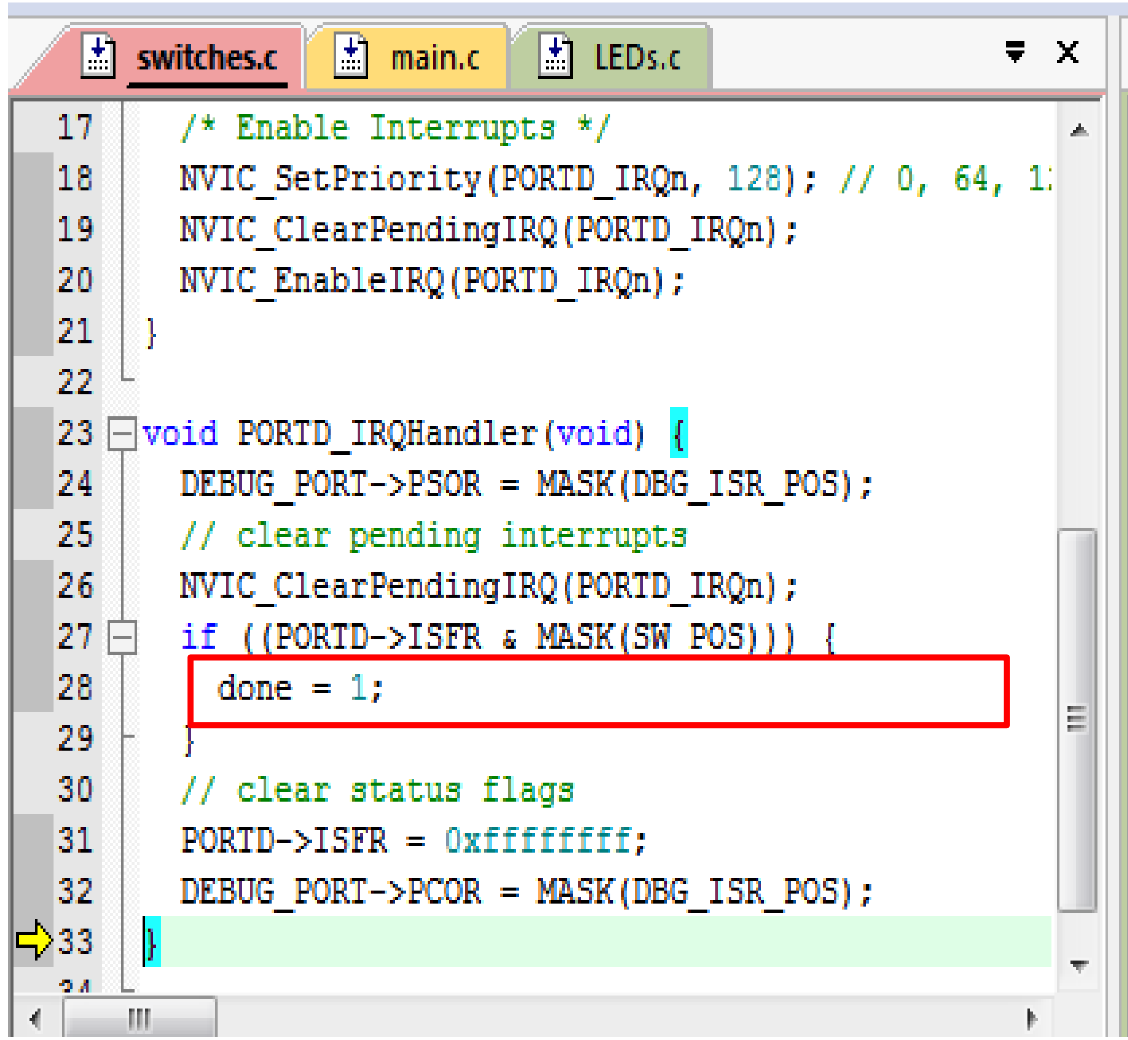
Example Program Requirements & Design



- Req1: When Switch 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

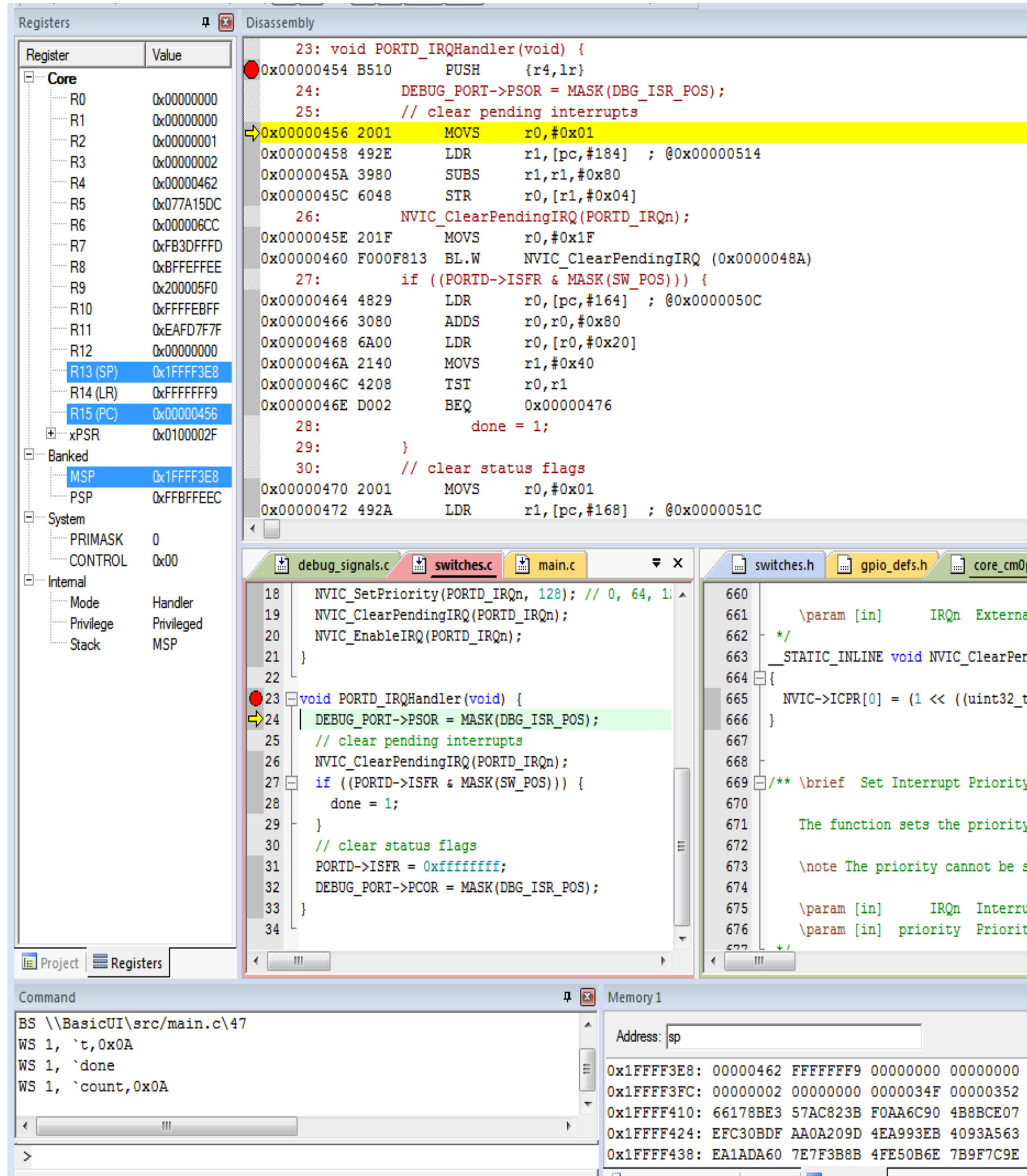
Example Exception Handler

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



Use Debugger for Detailed Processor View

- Can see registers, stack, source code, dissassembly (object code)
- Note: Compiler may generate code for function entry (see address 0x0000_0454)
- Place breakpoint on Handler function declaration line in source code (23), not at first line of function code (24)



ENTERING AN EXCEPTION HANDLER

CPU's Hardwired Exception Processing

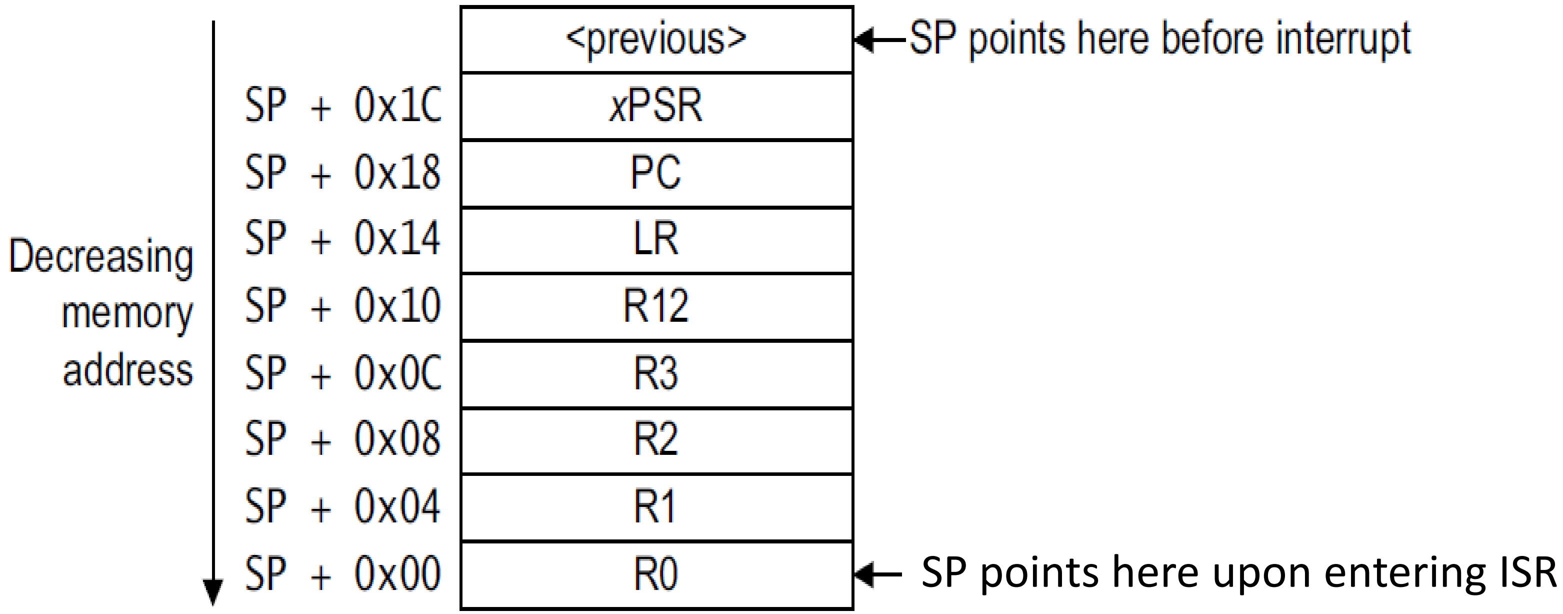
1. Finish current instruction (except for lengthy instructions)
2. Push context (8 32-bit words) onto current stack (MSP or PSP)
   - xPSR, Return address, LR (R14), R12, R3, R2, R1, R0
3. Switch to handler/privileged mode, use MSP
4. Load PC with address of exception handler
5. Load LR with EXC_RETURN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

Usually 16 cycles from exception request to execution of first instruction in handler

1. Finish Current Instruction

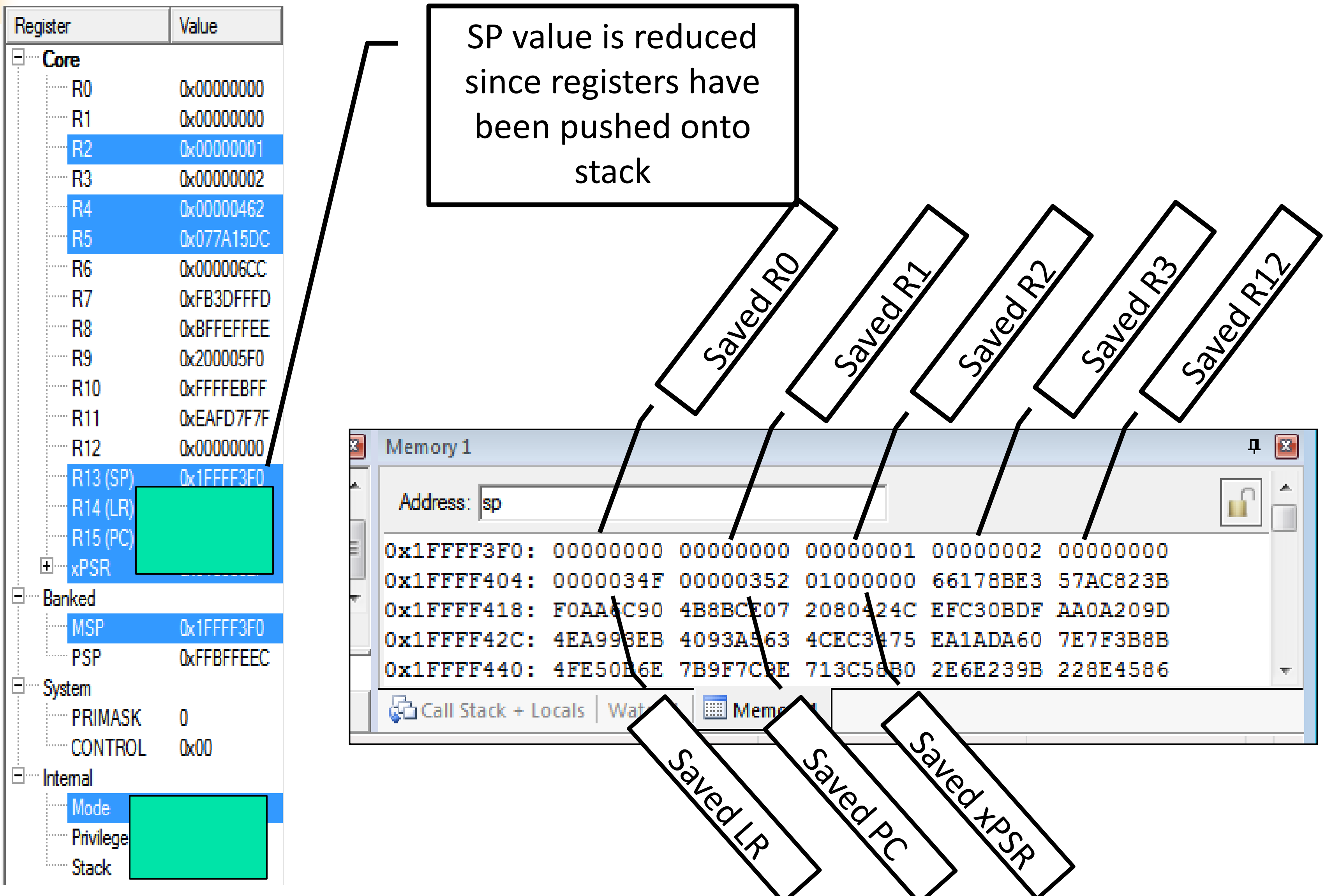
- Most instructions are short and finish quickly
- Some instructions may take many cycles to execute
  - Load Multiple (LDM), Store Multiple (STM), Push, Pop, MULS (32 cycles for some CPU core implementations)
- This will delay interrupt response significantly
- If one of these is executing when the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



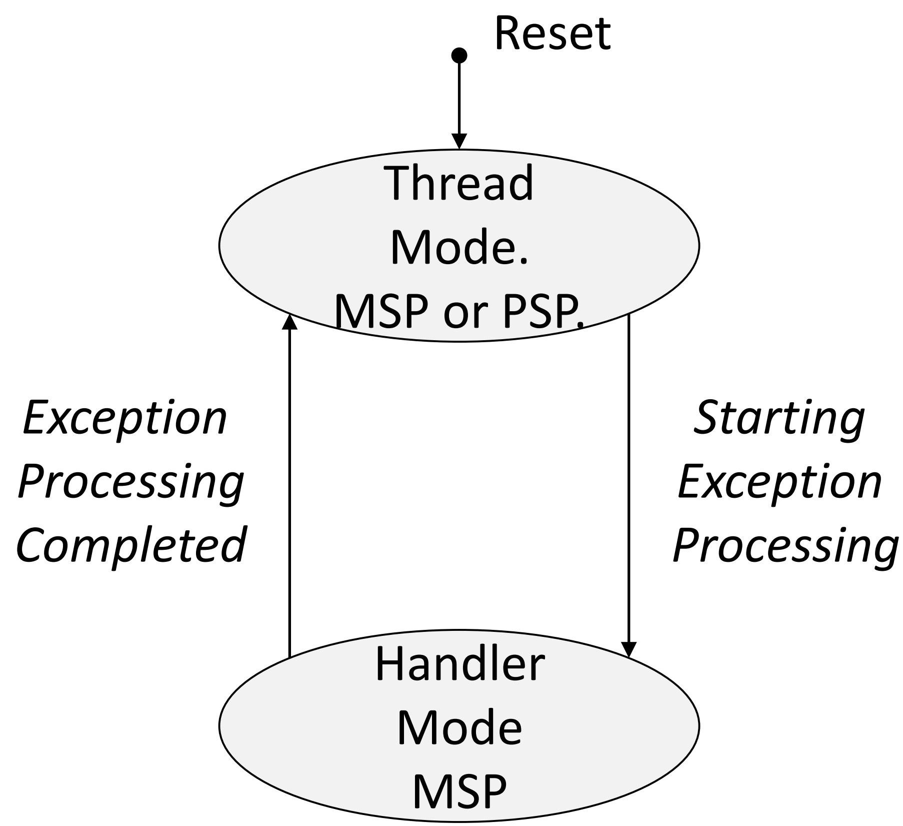
- Two SPs: Main (MSP), process (PSP)
- Which is active depends on operating mode, CONTROL register bit 1
- Stack grows toward smaller addresses

Context Saved on Stack

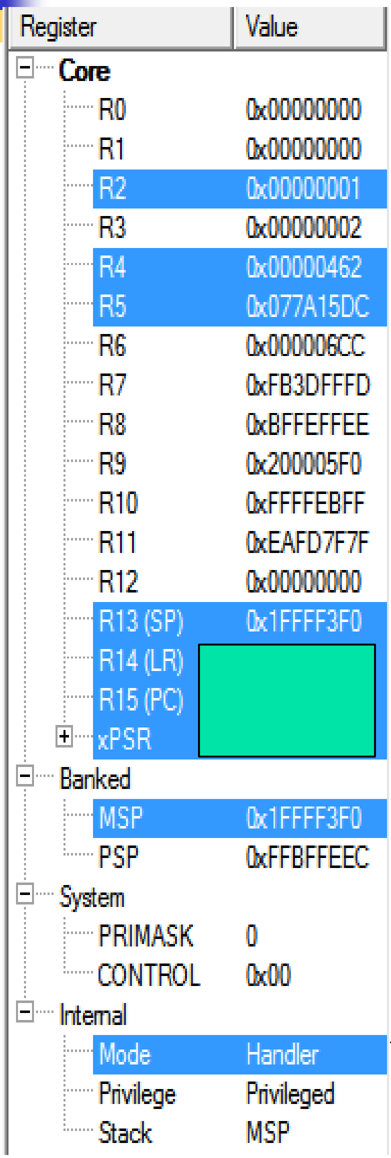


3. Switch to Handler/Privileged Mode

- Handler mode always uses Main SP



Handler and Privileged Mode



Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000001
R3	0x00000002
R4	0x00000462
R5	0x077A15DC
R6	0x000006CC
R7	0xFB3DFFFD
R8	0xBFFEFFEE
R9	0x200005F0
R10	0xFFFFEBFF
R11	0xEAFD7F7F
R12	0x00000000
R13 (SP)	0x1FFFF3F0
R14 (LR)	
R15 (PC)	
xPSR	
Banked	
MSP	0x1FFFF3F0
PSP	0xFFBFFEEC
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP

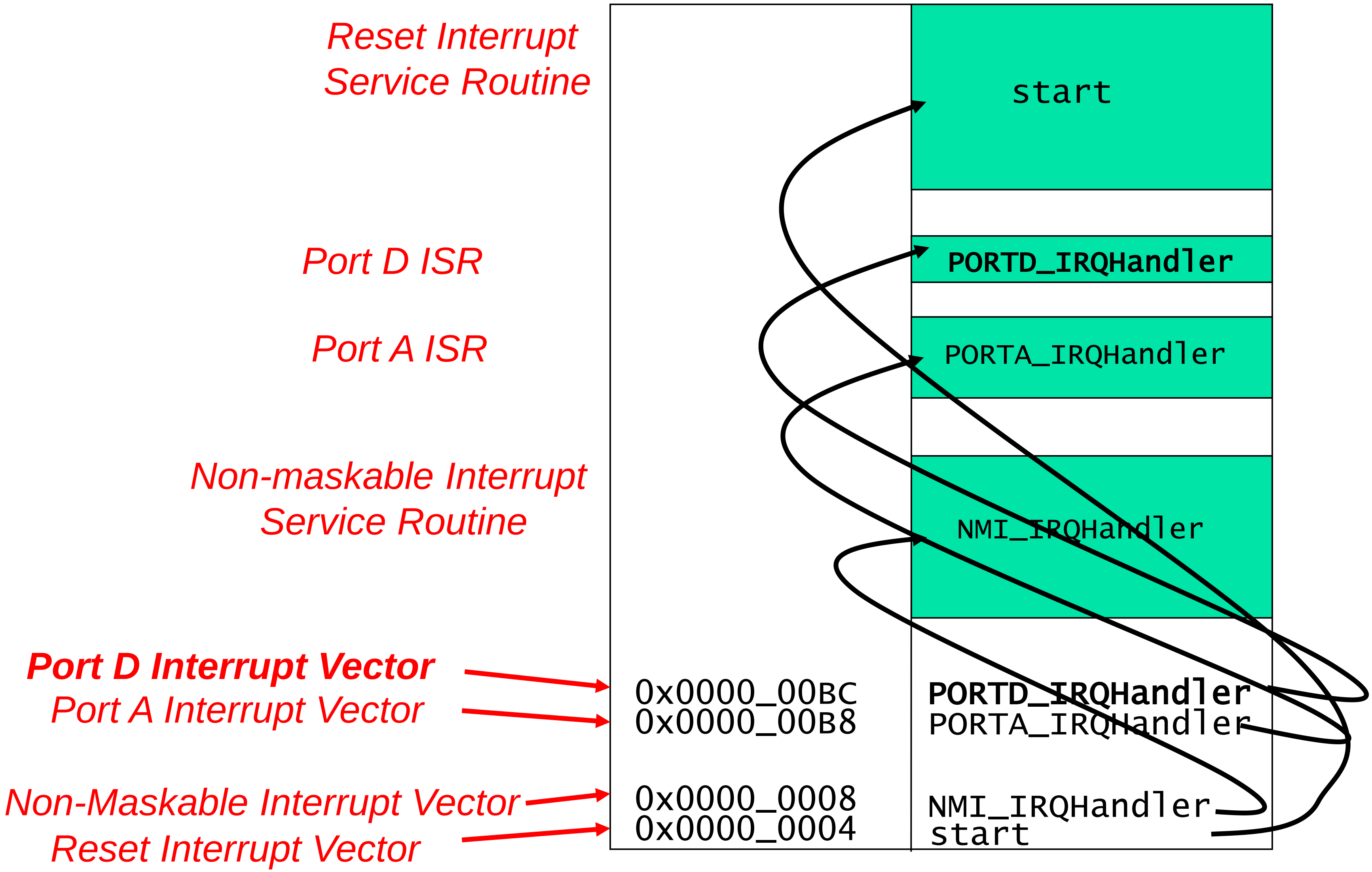
Mode changed to Handler. Was already using MSP and in Privileged mode

Update IPSR with Exception Number

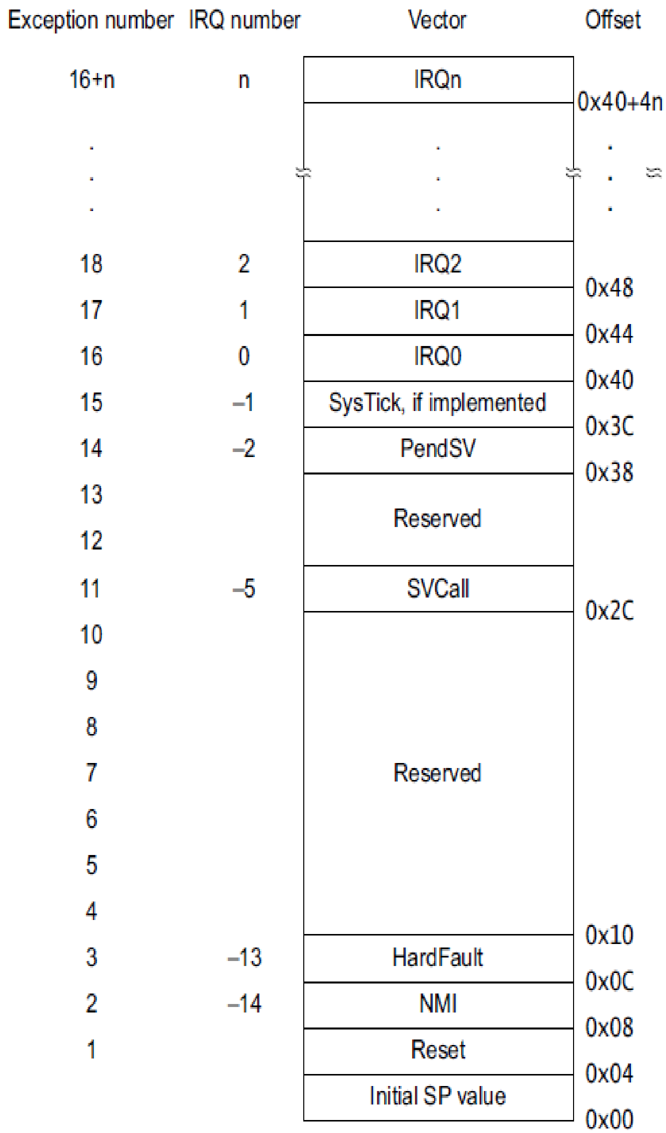
Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000001
R3	0x00000002
R4	0x00000462
R5	0x077A15DC
R6	0x000006CC
R7	0xFB3DFFFD
R8	0xBFFEFFEE
R9	0x200005F0
R10	0xFFFFEBFF
R11	0xEAFD7F7F
R12	0x00000000
R13 (SP)	0x1FFFF3F0
R14 (LR)	
R15 (PC)	
xPSR	0x0100002F
Banked	
MSP	0x1FFFF3F0
PSP	0xFFBFFEEC
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP

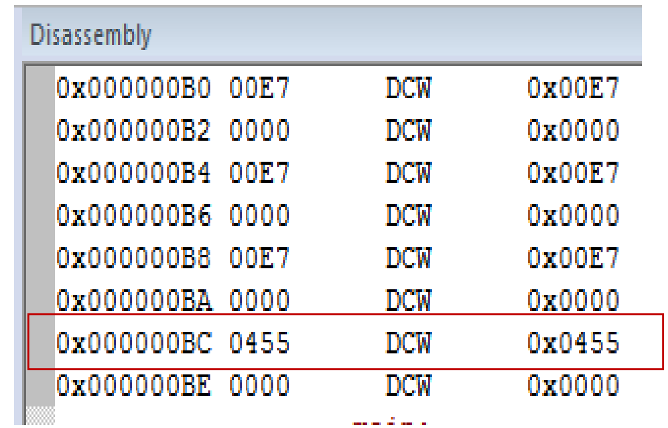
PORTD_IRQ is Exception number 0x2F (interrupt number + 0x10)

4. Load PC With Address Of Exception Handler



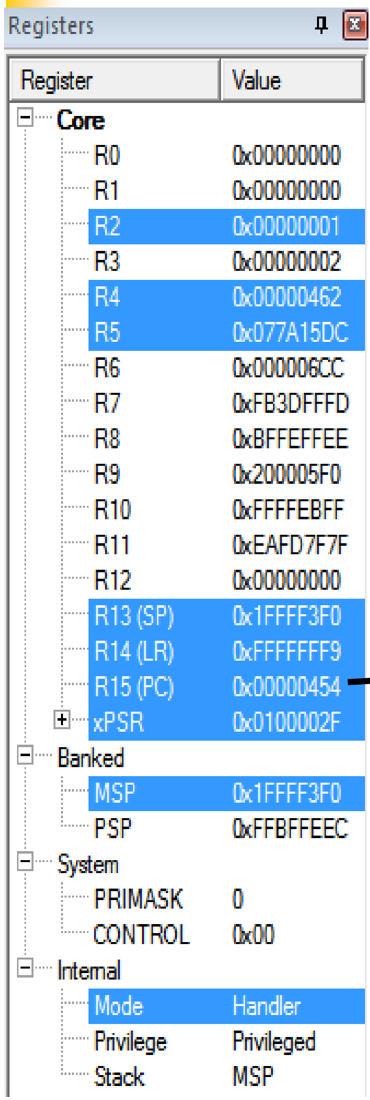
Can Examine Vector Table With Debugger

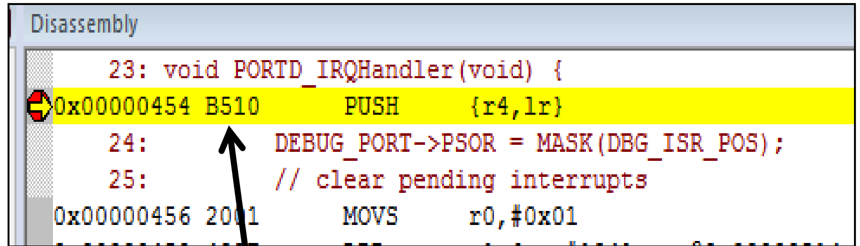




- PORTD ISR is IRQ #31 (0x1F), so vector to handler begins at **0x40+4*0x1F = 0xBC**
- Why is the vector odd? 0x0000_0455
- LSB of address indicates that handler uses Thumb code

Upon Entry to Handler





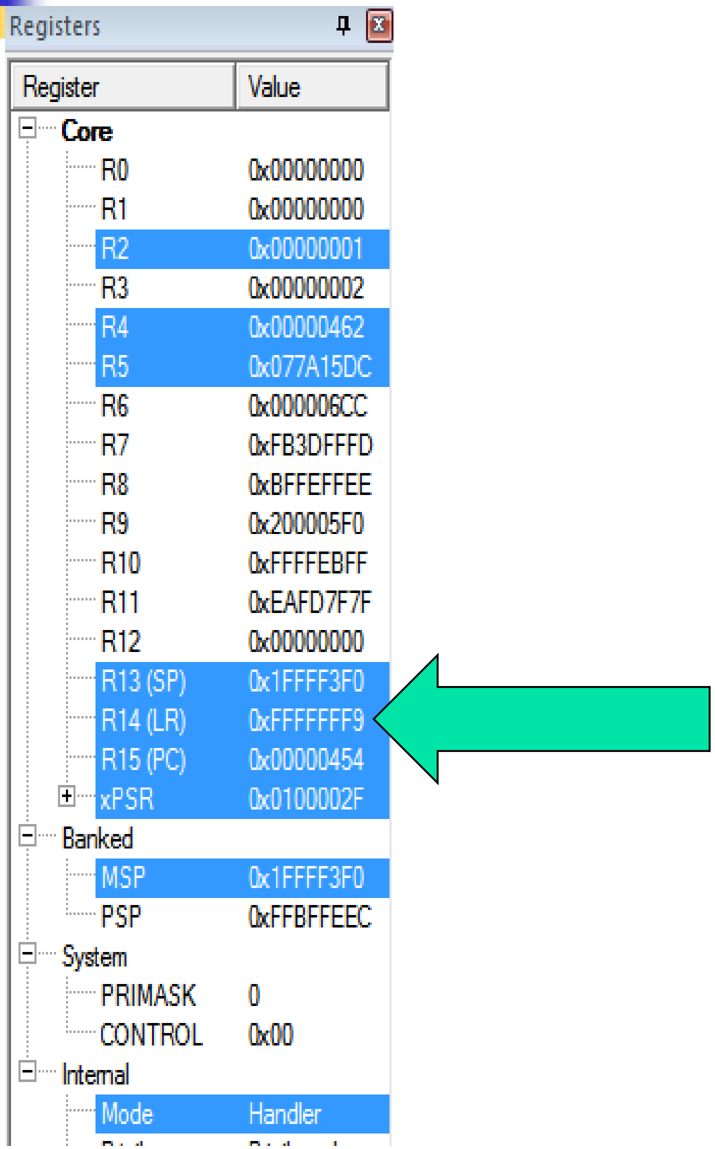
PC has been loaded with start address of handler

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



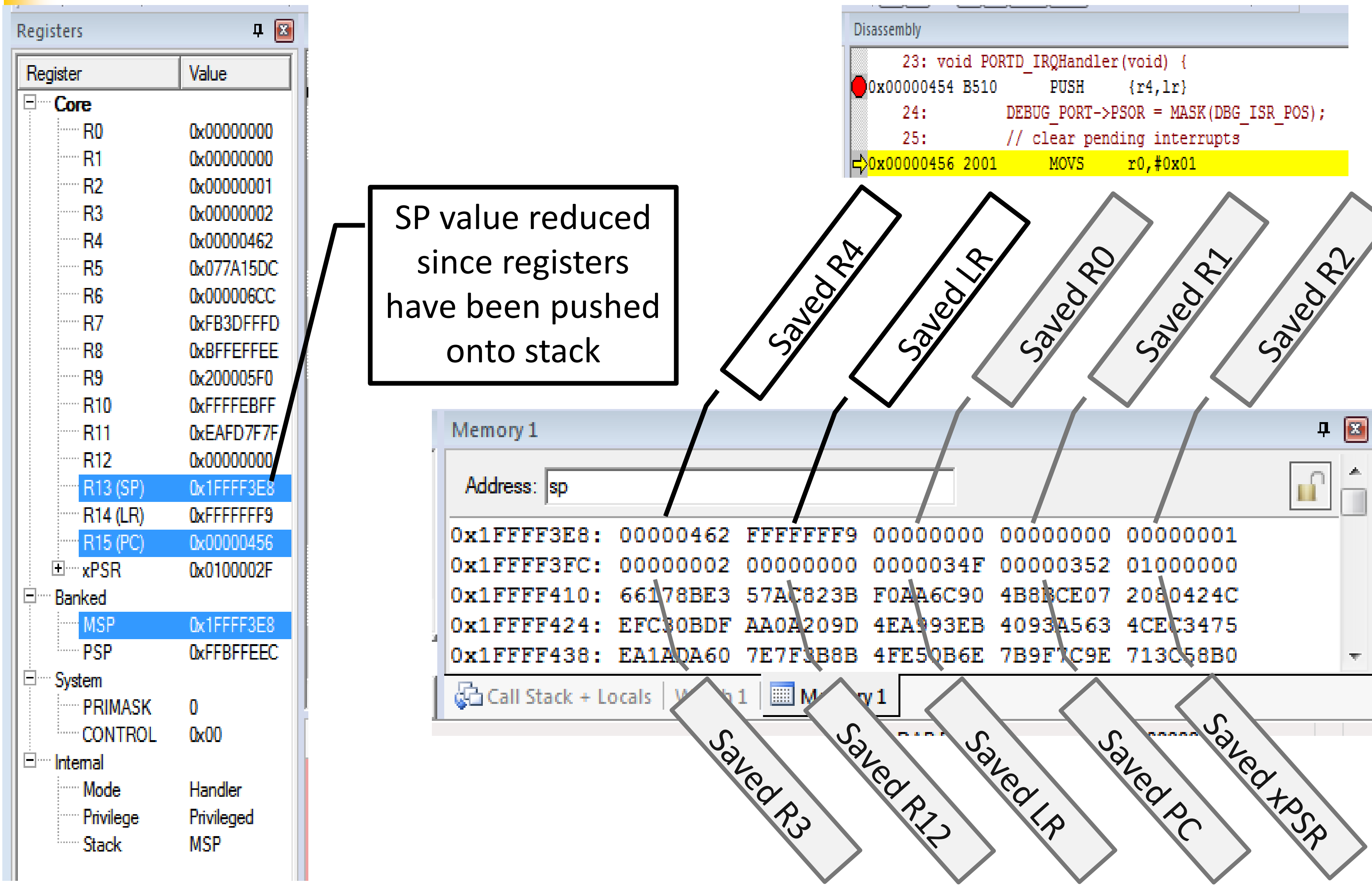
6. Start Executing Exception Handler

- 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

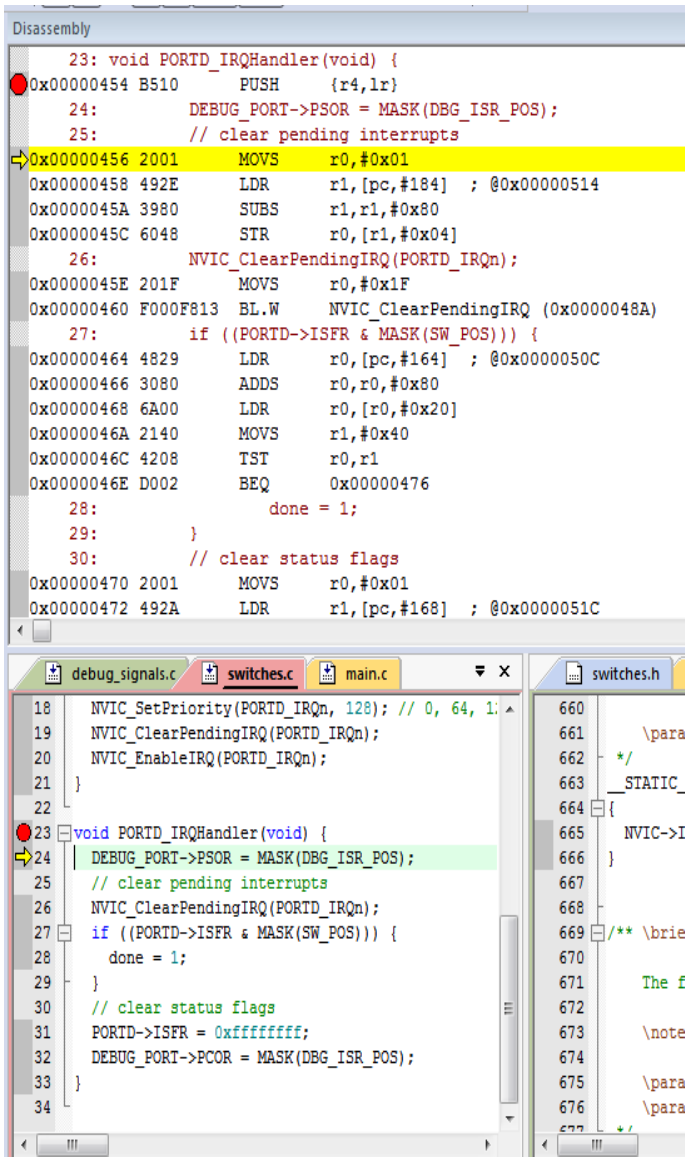
Disassembly

```
    23: void PORTD_IRQHandler(void) {
0x00000454 B510      PUSH     {r4,lr}
    24:         DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);
    25:         // clear pending interrupts
0x00000456 2001      MOVS     r0,#0x01
```

After Handler Has Saved More Context



Continue Executing Exception Handler



- Execute user code in handler

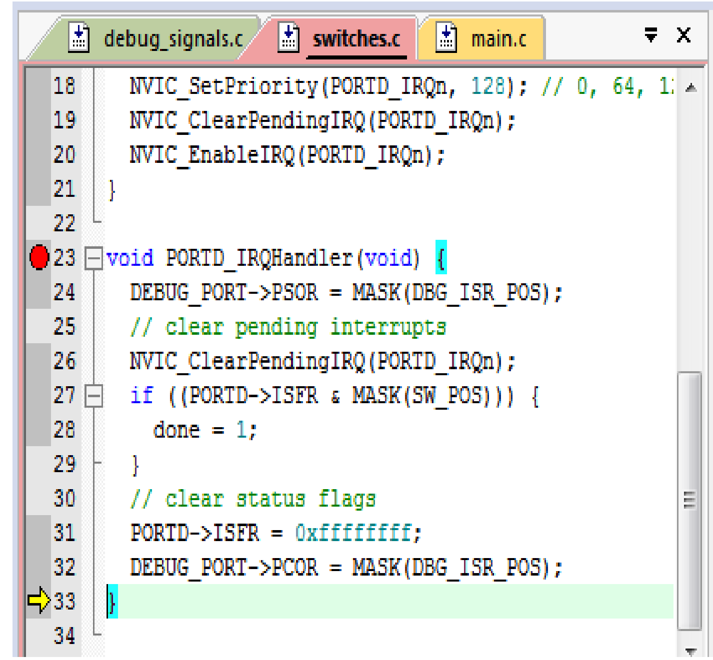
EXITING AN EXCEPTION HANDLER

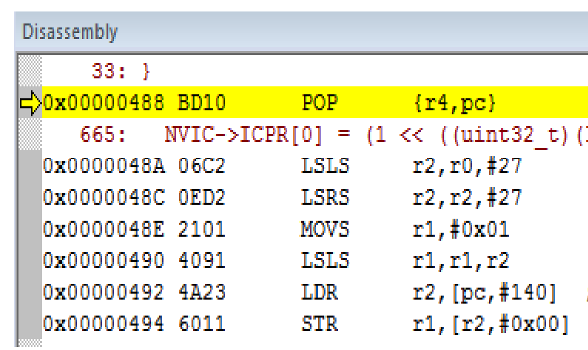
Exiting an Exception Handler

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

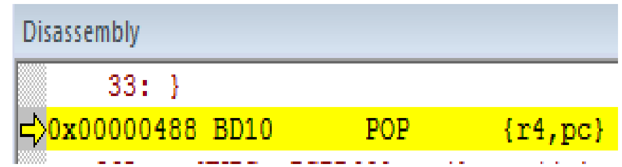
- No “return from interrupt” instruction
- Use regular instruction instead
  - BX LR - Branch to address in LR by loading PC with LR contents
  - POP …, PC - Pop address from stack into PC
- … with a special value EXC_RETURN loaded into the PC to trigger exception handling processing
  - BX LR used if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

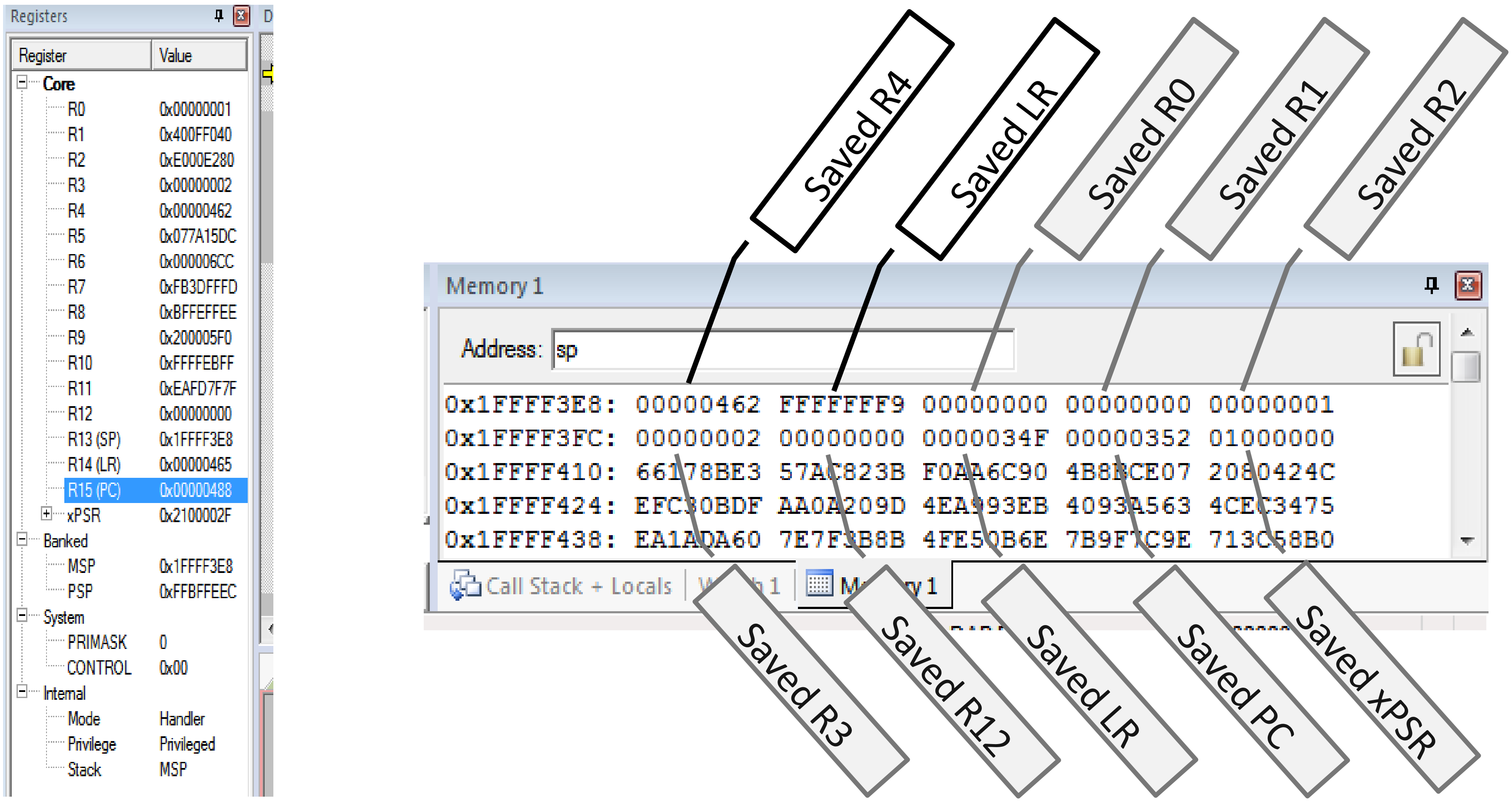




What Will Be Popped from Stack?

- R4: 0x0000_0462
- PC: 0xFFFF_FFF9



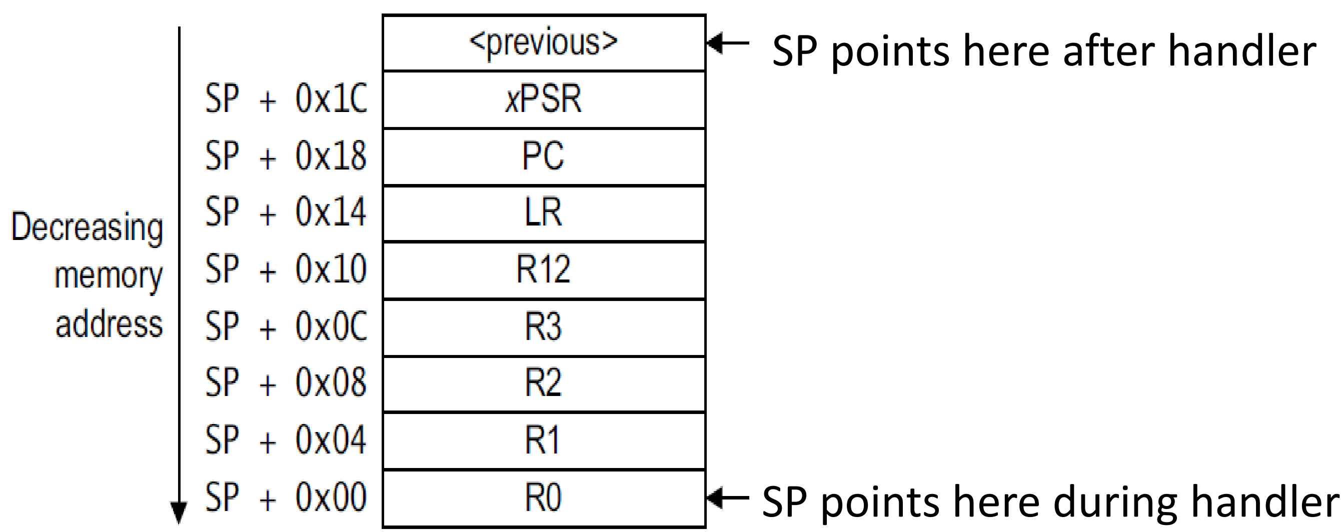


2. Select Stack, Restore Context

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

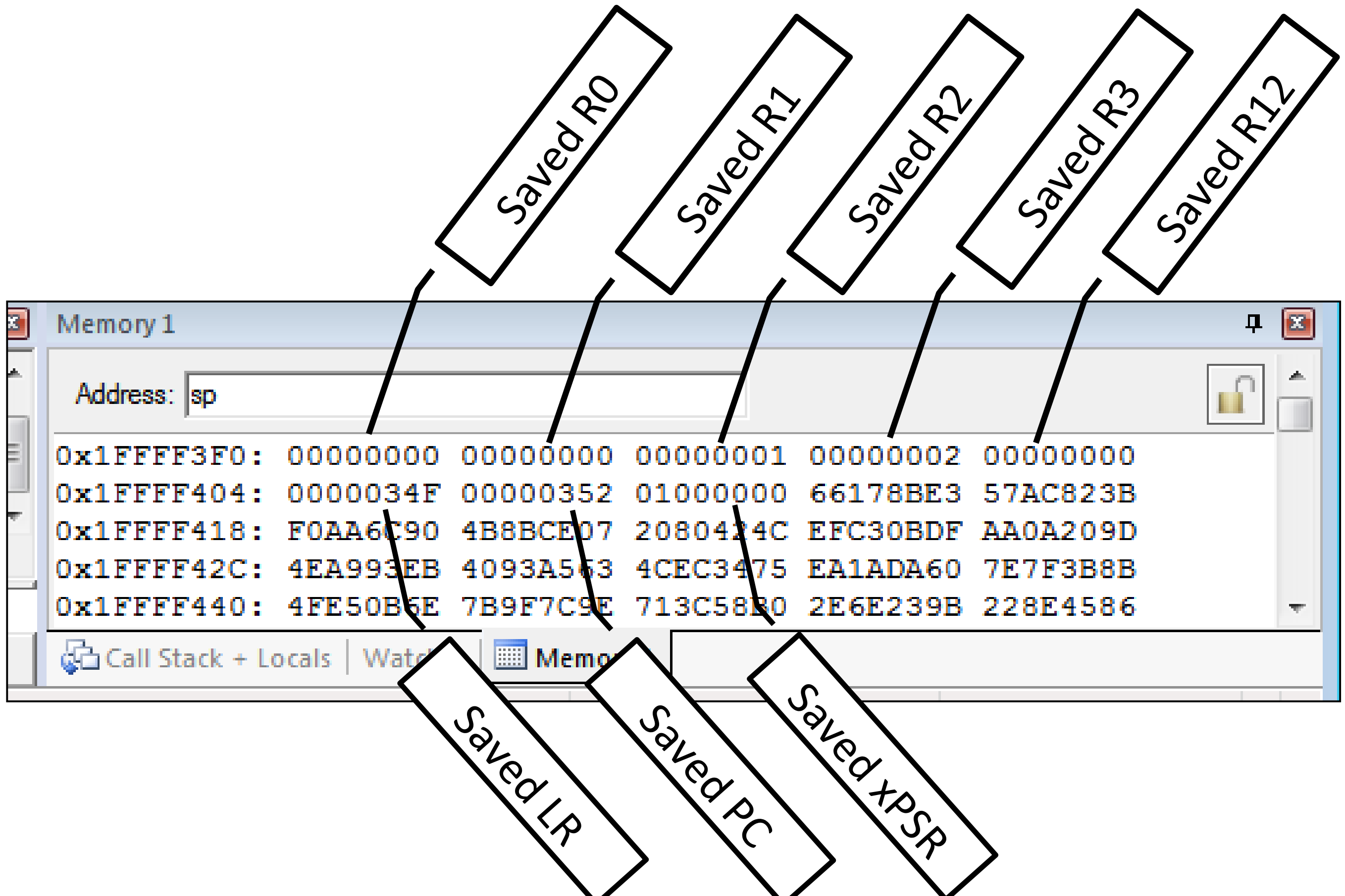
EXC_RETURN	Return Stack	Description
0xFFFF_FFF1	0 (MSP)	Return to exception handler with MSP
0xFFFF_FFF9	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	1 (PSP)	Return to thread with PSP

- Pop the registers from that stack



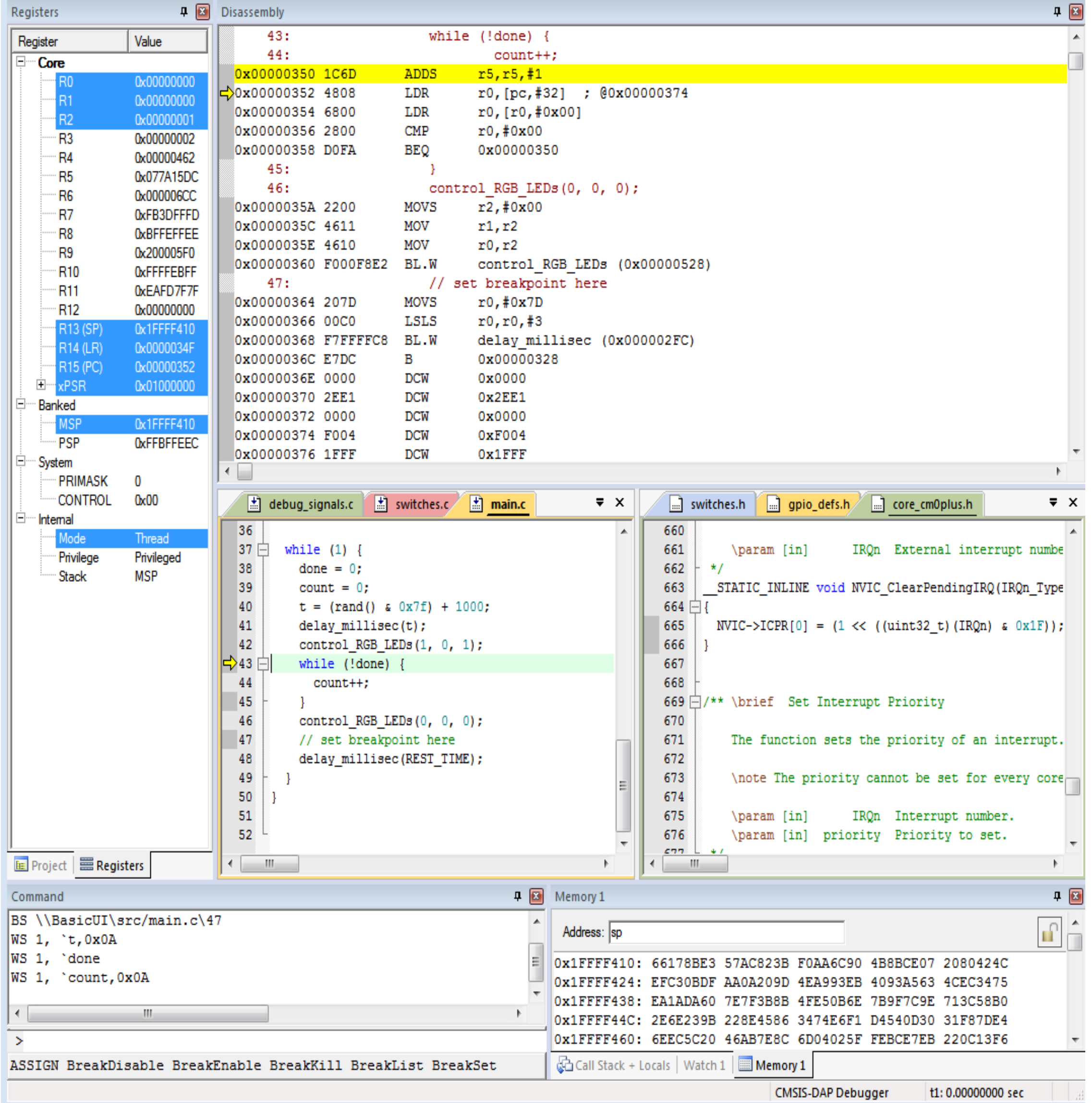
Example

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



3. 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

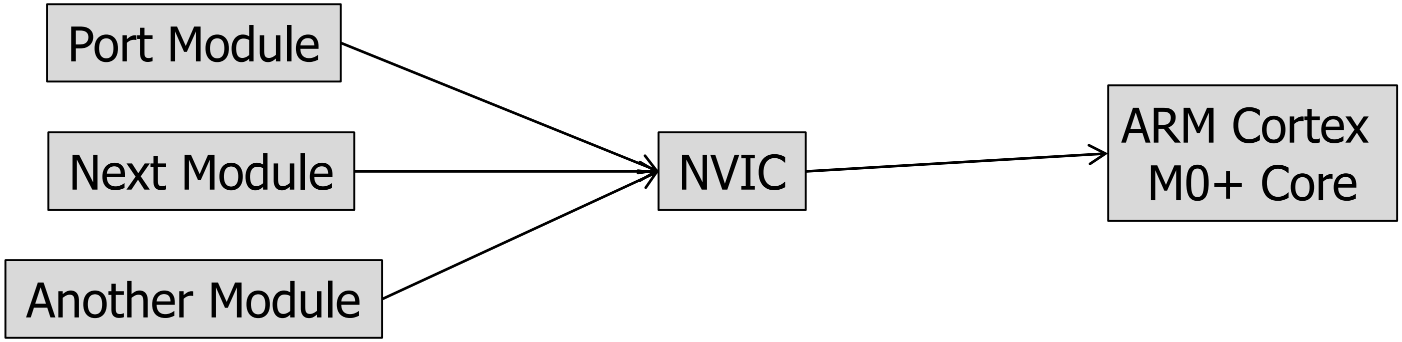


CORTEX-M0+ INTERRUPTS

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)



- NVIC manages and prioritizes external interrupts for Cortex-M0+
- 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

Some Interrupt Sources (Partial)

Vector Start Address	Vector #	IRQ	Source	Description
0x0000_0004	1		ARM Core	Initial program counter
0x0000_0008	2		ARM Core	Non-maskable interrupt
0x0000_0040-4C	16-19	0-3	Direct Memory Access Controller	Transfer complete or error
0x0000_0058	22	6	Power Management Controller	Low voltage detection
0x0000_0060-64	24-25	8-9	I^2C Modules	Status and error
0x0000_0068-6C	26-27	10-11	SPI Modules	Status and error
0x0000_0070-78	28-30	12-14	UART Modules	Status and error
0x0000_00B8	46	30	Port Control Module	Port A Pin Detect
0x0000_00BC	47	31	Port Control Module	Port D Pin Detect

Up to 32 non-core vectors, 16 core vectors
From KL25 Sub-Family Reference Manual, Table 3-6

NVIC Registers and State

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
  - Interrupt Priority Registers (IPR0-7 registers): two bits per interrupt source, four interrupt sources per register
  - Set priority to 0 (highest priority), 64, 128 or 192 (lowest)
  - CMSIS: NVIC_SetPriority(IRQnum, priority)

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 (Priority Mask Flag)
    - Set to 1 to prevent activation of all exceptions with configurable priority
    - Clear to 0 to allow activation of all exceptions
  - 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

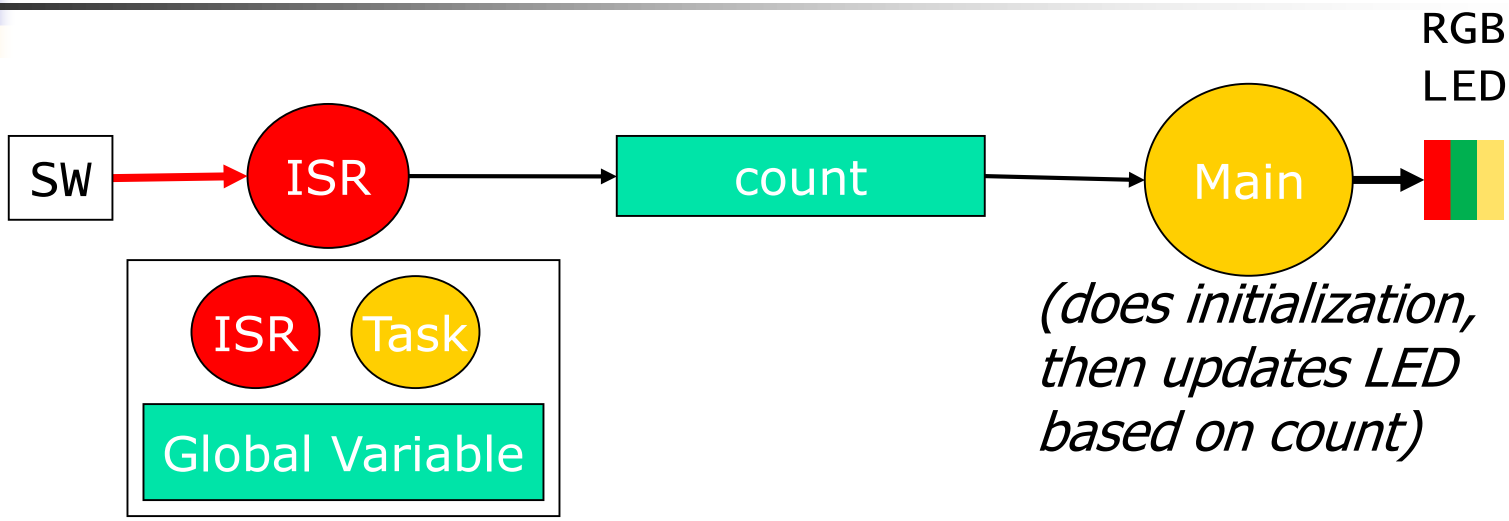
Special Cases of Prioritization

- 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

EXAMPLE USING PORT MODULE AND EXTERNAL INTERRUPTS

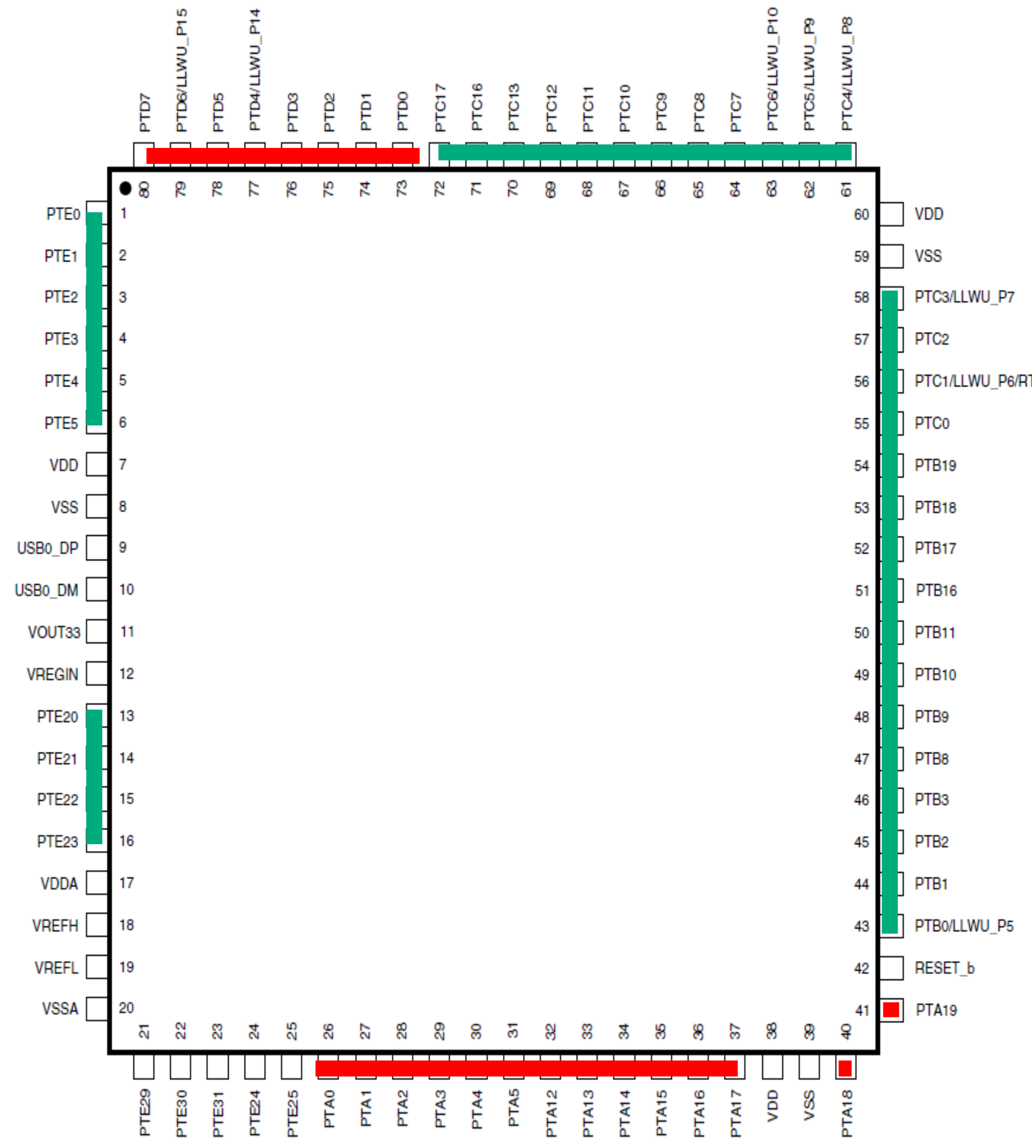
Refresher: Program Requirements & Design



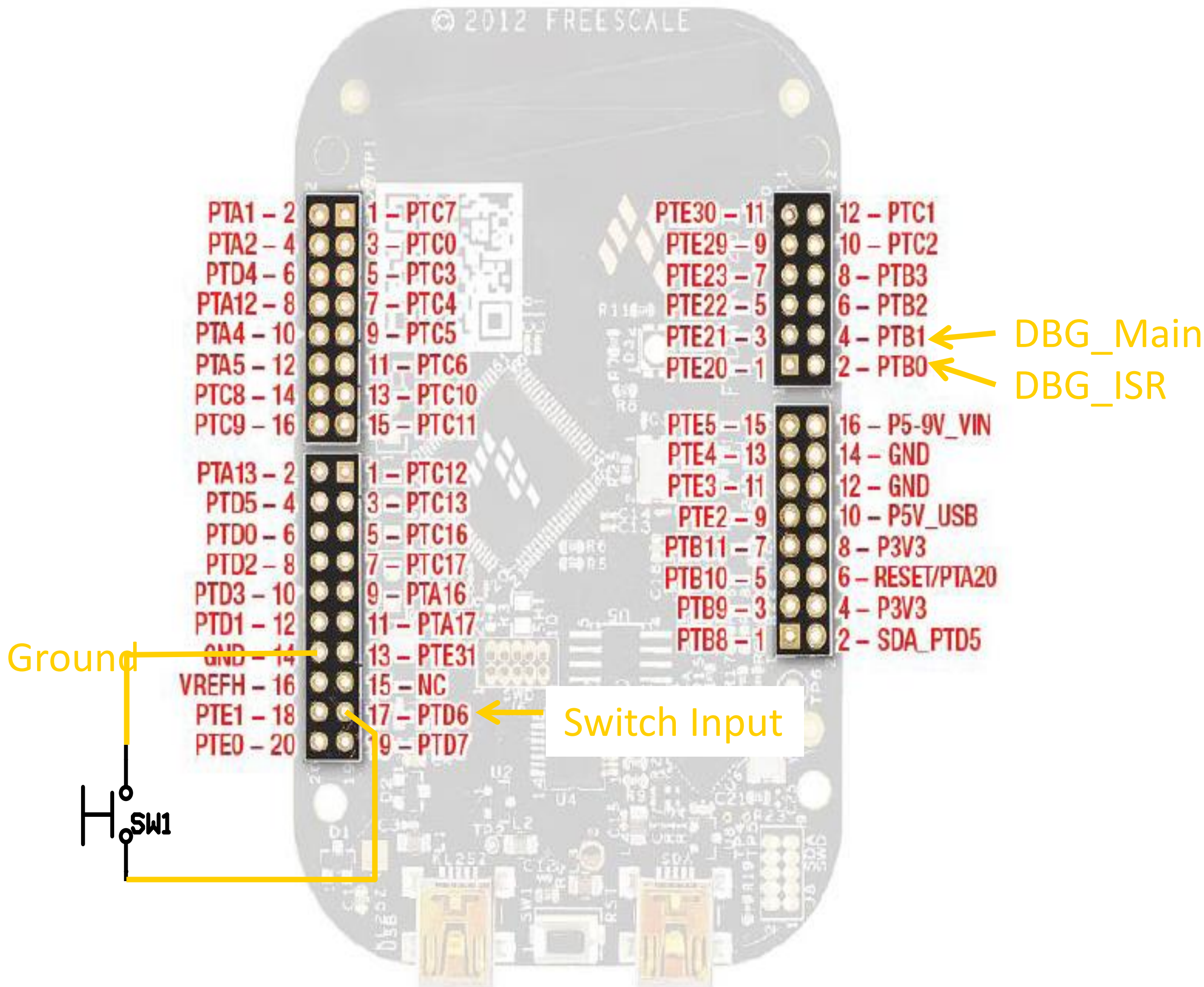
- Req1: When Switch SW is pressed, ISR will increment count variable
- Req2: Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1)
- Req3: Main code will toggle its debug line DBG_MAIN each time it executes
- Req4: ISR will raise its debug line DBG_ISR (and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

KL25Z GPIO Ports with Interrupts

- Port A (PTA) through Port E (PTE)
- Not all port bits are available (package-dependent)
- Ports A and D support interrupts



FREEDOM KL25Z Physical Set-up



Building a Program – Break into Pieces

- First break into threads, then break thread into steps
  - Main thread:
    - First initialize system
      - initialize switch: configure the port connected to the switches to be input
      - initialize LEDs: configure the ports connected to the LEDs to be outputs
      - initialize interrupts: initialize the interrupt controller
    - Then repeat
      - Update LEDs based on count
  - Switch Interrupt thread:
      - Update count

- Determine which variables ISRs will share with main thread
  - This is how ISR will send information to main thread
  - Mark these shared variables as *volatile* (more details ahead)
  - Ensure access to the shared variables is *atomic* (more details ahead)

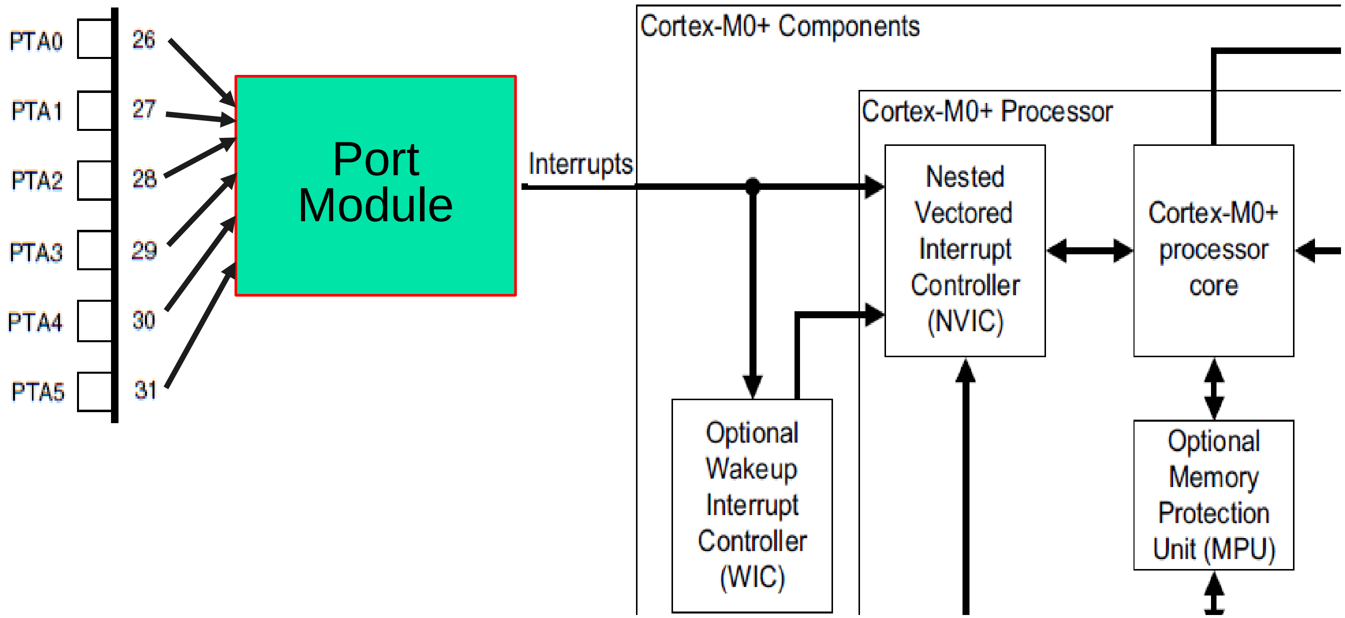
Where Do the Pieces Go?

- main
  - top level of main thread code
- switches
  - #defines for switch connections
  - declaration of count variable
  - Code to initialize switch and interrupt hardware
  - ISR for switch
- LEDs
  - #defines for LED connections
  - Code to initialize and light LEDs
- debug_signals
  - #defines for debug signal locations
  - Code to initialize and control debug lines

Configure MCU to Respond to the Interrupt

- Set up peripheral module to generate interrupt
  - We'll use Port Module in this example
- Set up NVIC
- Set global interrupt enable
  - Use CMSIS Macro __enable_irq();
  - This flag does not enable all interrupts; instead, it is an easy way to ***disable*** interrupts
    - Could also be called "don't disable all interrupts"

Port Module



- Port Module connects
  external pins to NVIC (and other devices)
- Relevant registers
  - PCR - Pin Control Register (32 per port)
    - Each register corresponds to an input pin
  - ISFR - Interrupt status flag register (one per port)
    - Each bit corresponds to an input pin
    - Bit is set to 1 if an interrupt has been detected

Pin Control Register

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0							ISF	0				IRQC			
W								w1c								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- ISF indicates if interrupt has been detected - different way to access same data as ISFR
- IRQC field of PCR defines behavior for external hardware interrupts
- Can also trigger direct memory access (not covered here)

IRQC	Configuration
0000	Interrupt Disabled
….	DMA, reserved
1000	Interrupt when logic zero
1001	Interrupt on rising edge
1010	Interrupt on falling edge
1011	Interrupt on either edge
1100	Interrupt when logic one
…	reserved

CMSIS C Support for PCR

- MKL25Z4.h defines PORT_Type structure with a PCR field (array of 32 integers)

```
/** PORT - Register Layout Typedef */
typedef struct {
  __IO uint32_t PCR[32];   /** Pin Control Register n, array 
offset: 0x0, array step: 0x4 */
  __O  uint32_t GPCLR;     /** Global Pin Control Low Register, 
offset: 0x80 */
  __O  uint32_t GPCHR;     /** Global Pin Control High Register, 
offset: 0x84 */
       uint8_t RESERVED_0[24];
  __IO uint32_t ISFR;            /** Interrupt Status Flag 
Register, offset: 0xA0 */
} PORT_Type;
```

CMSIS C Support for PCR

- Header file defines pointers to PORT_Type registers

```
/* PORT - Peripheral instance base addresses */
/** Peripheral PORTA base address */
#define PORTA_BASE           (0x40049000u)
/** Peripheral PORTA base pointer */
#define PORTA                ((PORT_Type *)PORTA_BASE)
```

- Also defines macros and constants

```
#define PORT_PCR_MUX_MASK      0x700u
#define PORT_PCR_MUX_SHIFT     8
#define PORT_PCR_MUX(x)
(((uint32_t)(((uint32_t)(x))<<PORT_PCR_MUX_SHIFT))
&PORT_PCR_MUX_MASK)
```

Switch Interrupt Initialization

```
void init_switch(void) {
    /* enable clock for port D */
    SIM->SCGC5 |=  SIM_SCGC5_PORTD_MASK;
    /* Select GPIO and enable pull-up resistors and
          interrupts on falling edges for pin
    connected to switch */
    PORTD->PCR[SW_POS] |= PORT_PCR_MUX(1) |
          PORT_PCR_PS_MASK | PORT_PCR_PE_MASK |
    PORT_PCR_IRQC(0x0a);
          /* Set port D switch bit to inputs */
    PTD->PDDR &= ~MASK(SW_POS);
    /* Enable Interrupts */
    NVIC_SetPriority(PORTD_IRQn, 128);
    NVIC_ClearPendingIRQ(PORTD_IRQn);
    NVIC_EnableIRQ(PORTD_IRQn);
}
```

Main Function

```
int main (void) {

	init_switch();
	init_RGB_LEDs();
	init_debug_signals();
	__enable_irq();

	while (1) {
		DEBUG_PORT->PTOR = MASK(DBG_MAIN_POS);
		control_RGB_LEDs(count&1, count&2,
count&4);
		__wfi(); // sleep now, wait for interrupt
	}
}
```

Write Interrupt Service Routine

- No arguments or return values – void is only valid type
- Keep it short and simple
  - Much easier to debug
  - Improves system response time
- Name the ISR according to CMSIS-CORE system exception names
  - PORTD_IRQHandler, RTC_IRQHandler, etc.
  - The linker will load the vector table with this handler rather than the default handler
- Clear pending interrupts
  - Call NVIC_ClearPendingIRQ(IRQnum)
- Read interrupt status flag register to determine source of interrupt
- Clear interrupt status flag register by writing to PORTD->ISFR

ISR

```
void PORTD_IRQHandler(void) {
	DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);
	// clear pending interrupts
	NVIC_ClearPendingIRQ(PORTD_IRQn);

	if ((PORTD->ISFR & MASK(SW_POS))) {
		count++;
	}
	// clear status flags
	PORTD->ISFR = 0xffffffff;
	DEBUG_PORT->PCOR = MASK(DBG_ISR_POS);
}
```

Evaluate Basic Operation

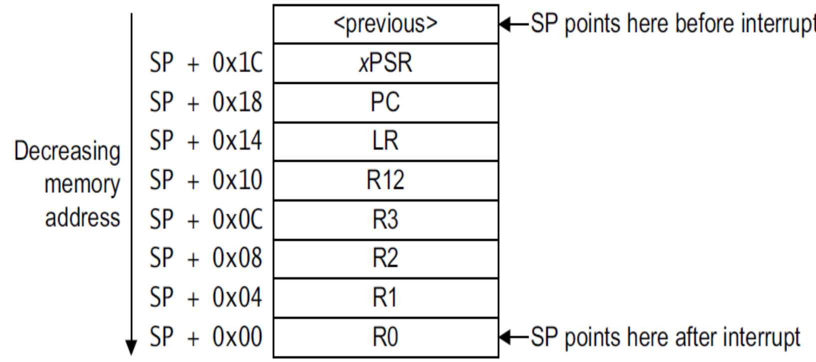
- Build program
- Load onto development board
- Start debugger
- Run
- Press switch, verify LED changes color

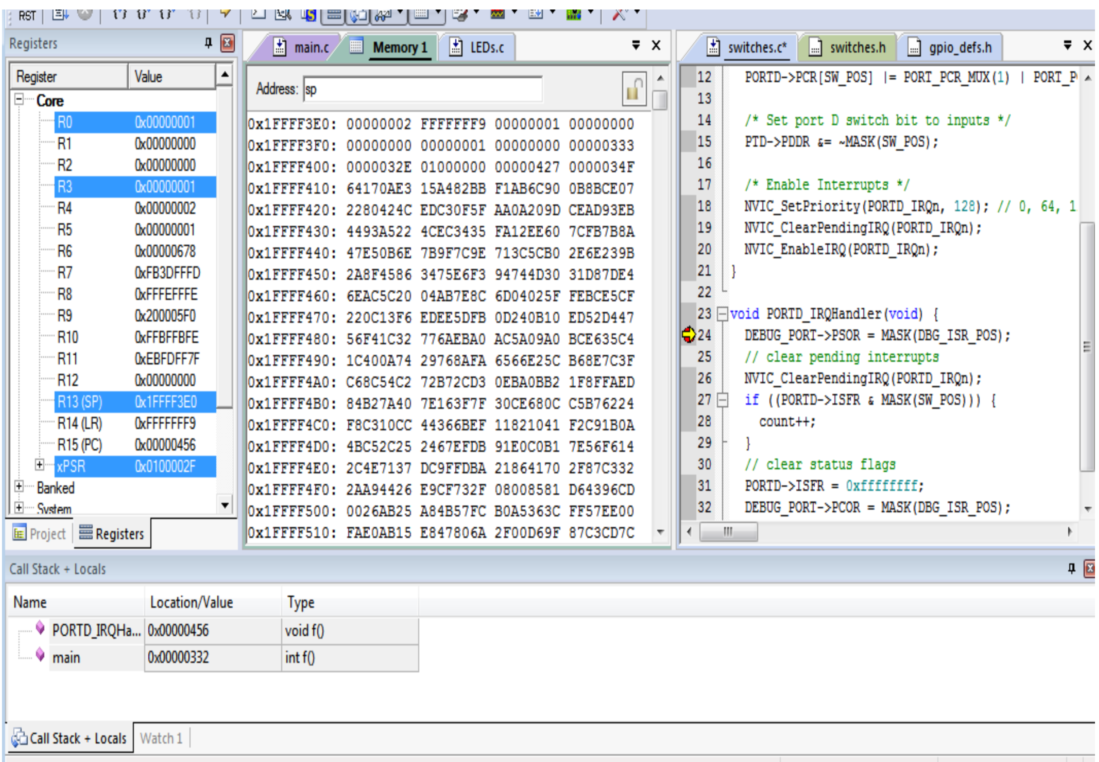
Examine Saved State in ISR

- Set breakpoint in ISR
- Run program
- Press switch, verify debugger stops at breakpoint
- Examine stack and registers

At Start of ISR

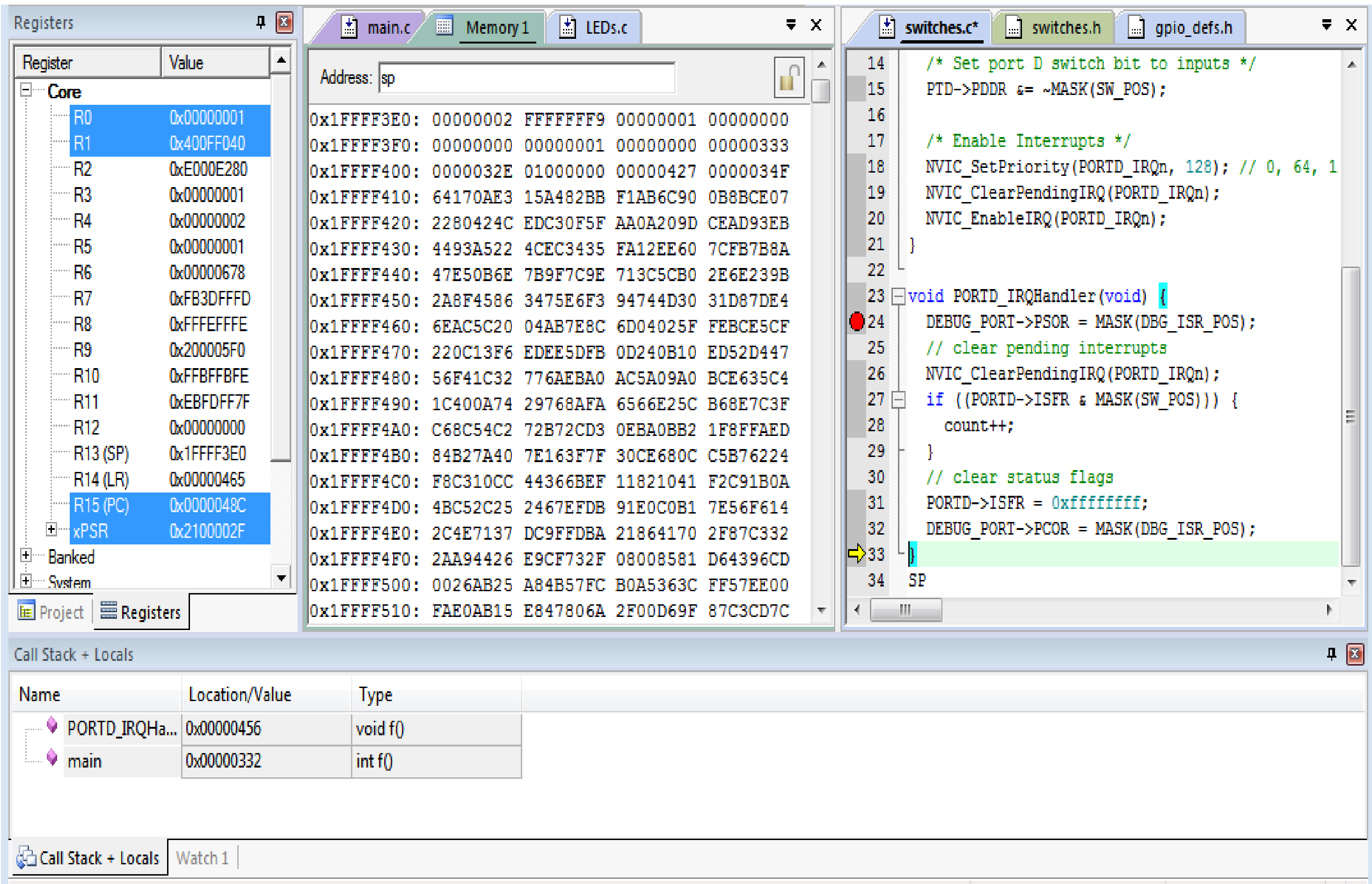
- Examine memory
- What is SP's value? See processor registers window





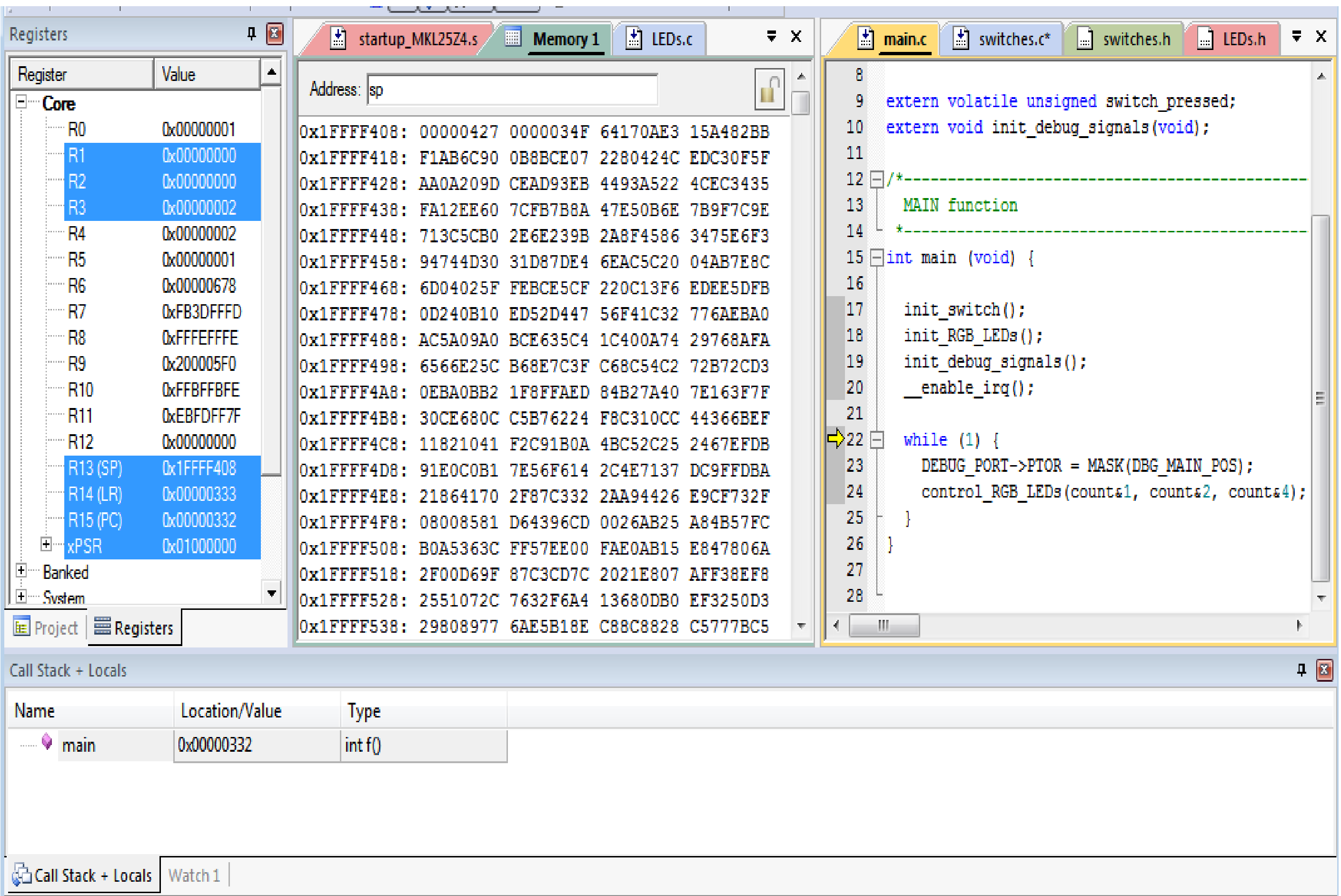
Step through ISR to End

- PC = 0x0000_048C
- Return address stored on stack: 0x0000_0333



Return from Interrupt to Main function

- PC = 0x0000_0332



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 $(1\text{-}U_{Int})*F_{CPU}$

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

Summary for Sharing Data

- In thread/ISR diagram, identify shared data
- Determine which shared data is too large to be handled atomically by default
  - This needs to be protected from preemption (e.g. disable interrupt(s), use an RTOS synchronization mechanism)
- Declare (and initialize) shared variables as volatile in main file (or globals.c)
  - ***volatile*** int my_shared_var=0;
- Update extern.h to make these variables available to functions in other files
  - ***volatile*** 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()

References

- Lecture Slides: Dr. Şule Gündüz Öğüdücü
- Lecture Slides: Dr. Erdem Matoğlu
- Lecture Slides: Dr. Feza Buzluca
- Lecture Slides: Dr. Bassel Soudan
- Lecture Slides: Dr. Gökhan İnce
- Lecture Slides: Dr. B.Berk Üstündağ