

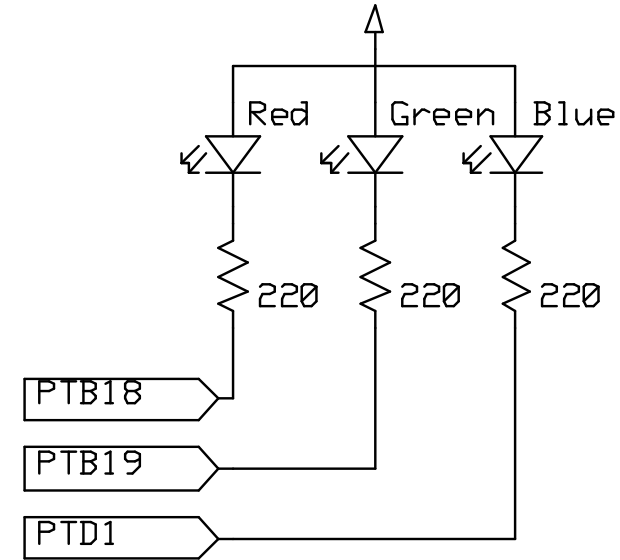
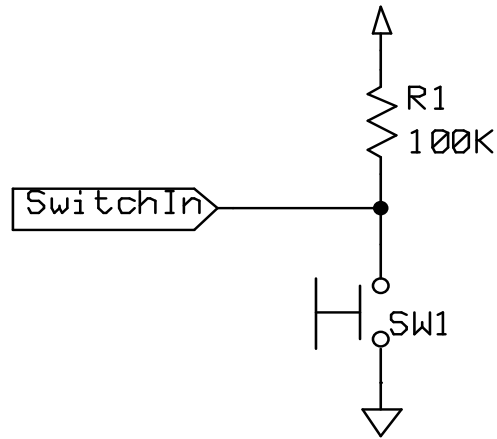
Cortex-M0+ Exceptions and Interrupts

Overview

- Exception and Interrupt Concepts
 - Entering an Exception Handler
 - Exiting an Exception Handler
- Cortex-M0+ Interrupts
 - Using Port Module and External Interrupts
- Timing Analysis
- 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

How to Detect Switch is Pressed?

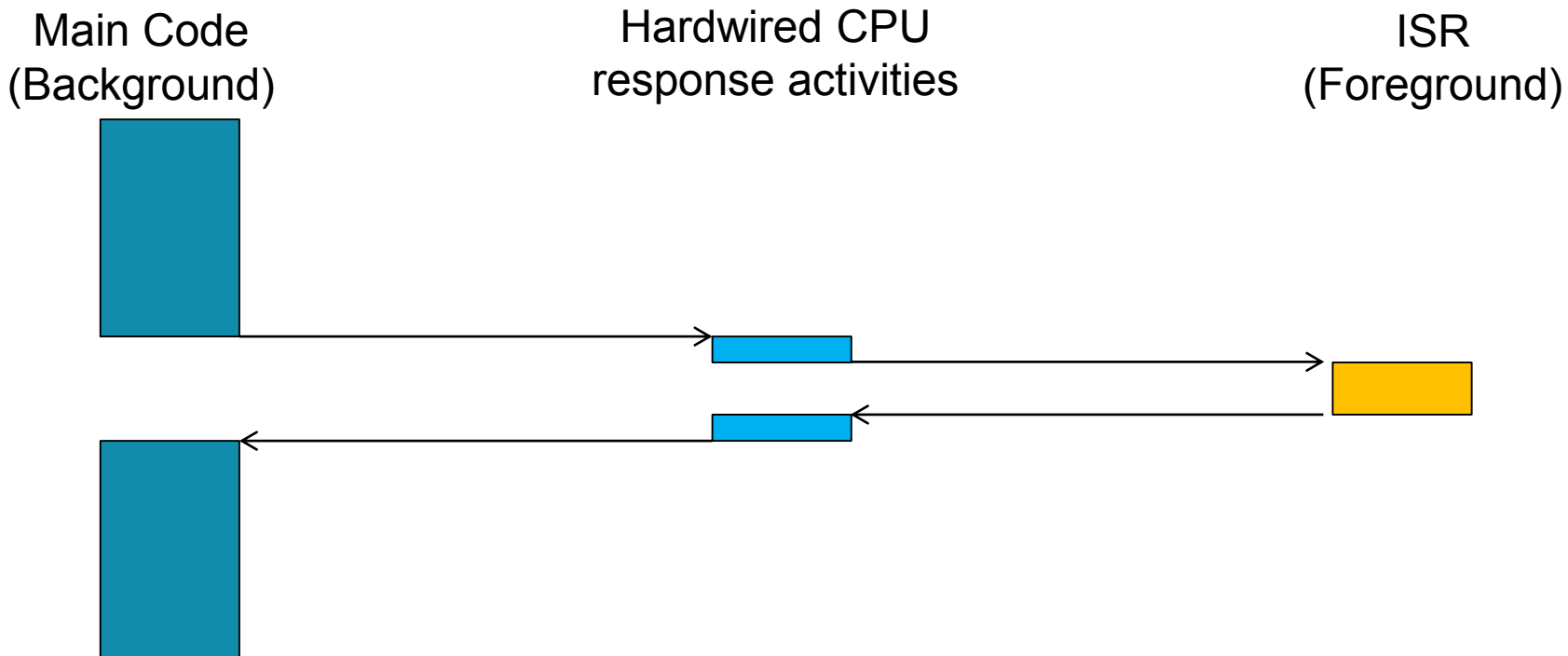
- Polling - use software to check it
 - Slow - need to explicitly check to see if switch is pressed
 - Wasteful of CPU time - the faster a response we need, the more often we need to check
 - Scales badly - difficult to build system with many activities which can respond quickly. Response time depends 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

```
main () {  
    while (1) {  
        if !sw1...  
    }  
}
```

Only use polling for one
task

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)

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

All 7 steps are done in hardware
(done very quickly)

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

I. Finish Current Instruction

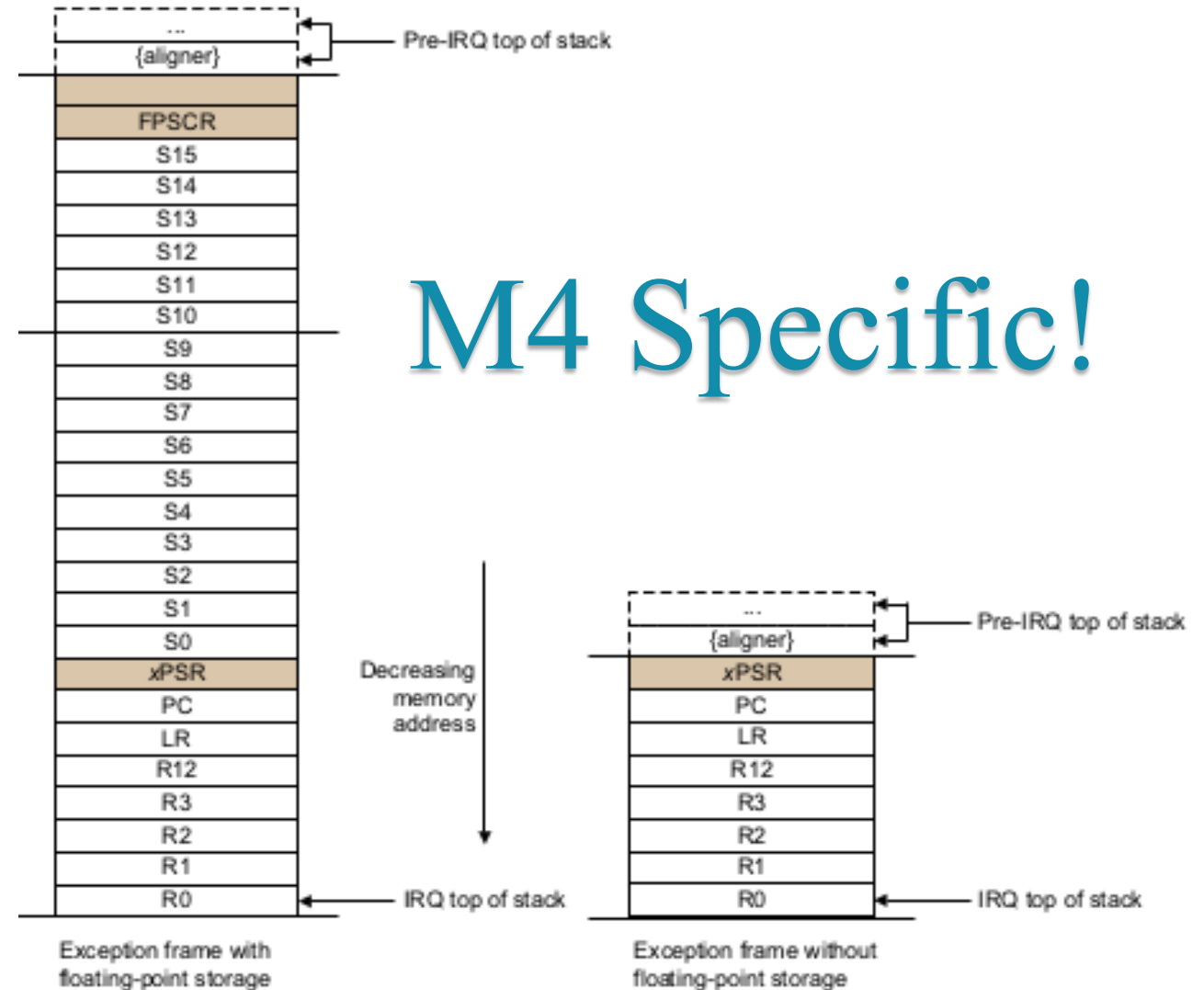
save scratch register and LR

- 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

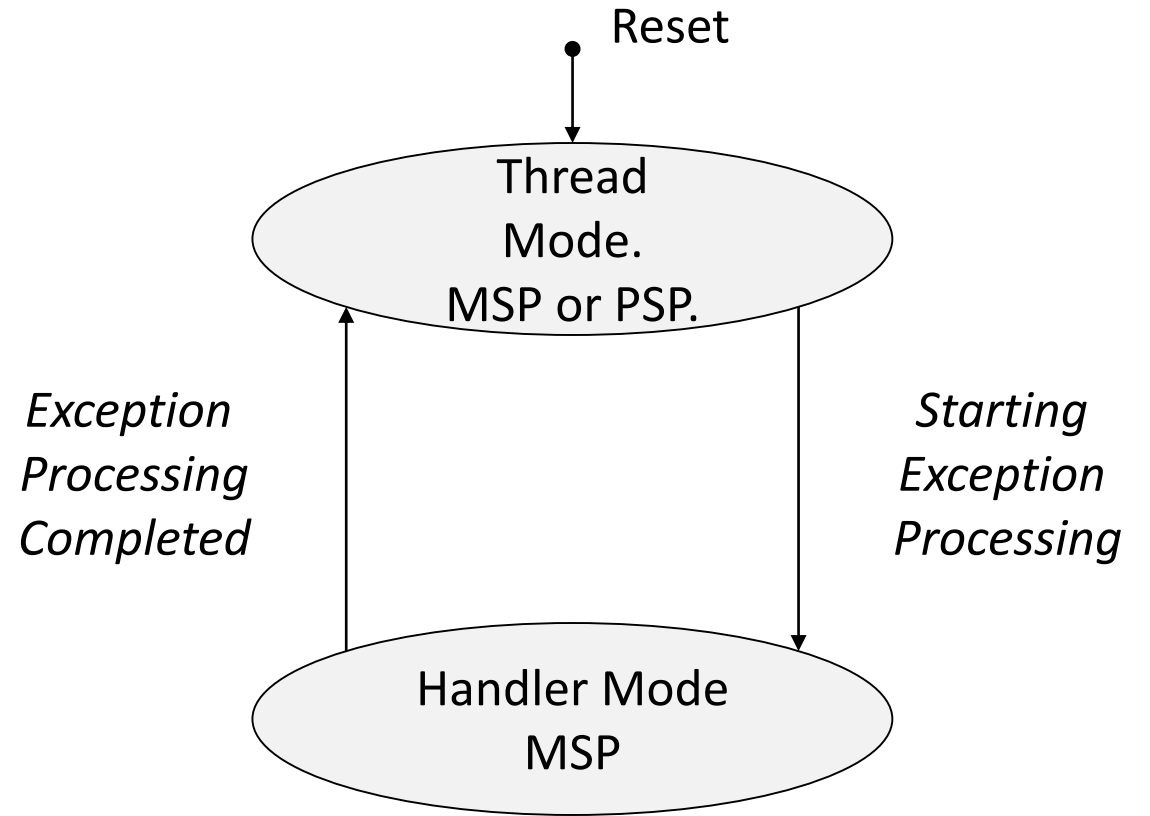
Usually using process stack
Handler mode = main stack

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



3. Switch to Handler/Privileged Mode

- Handler mode always uses Main SP



4. Load PC With Address Of Exception Handler

If there is every an interruption, PC will be loaded with PROTD_IRQHandler

*Reset Interrupt
Service Routine*

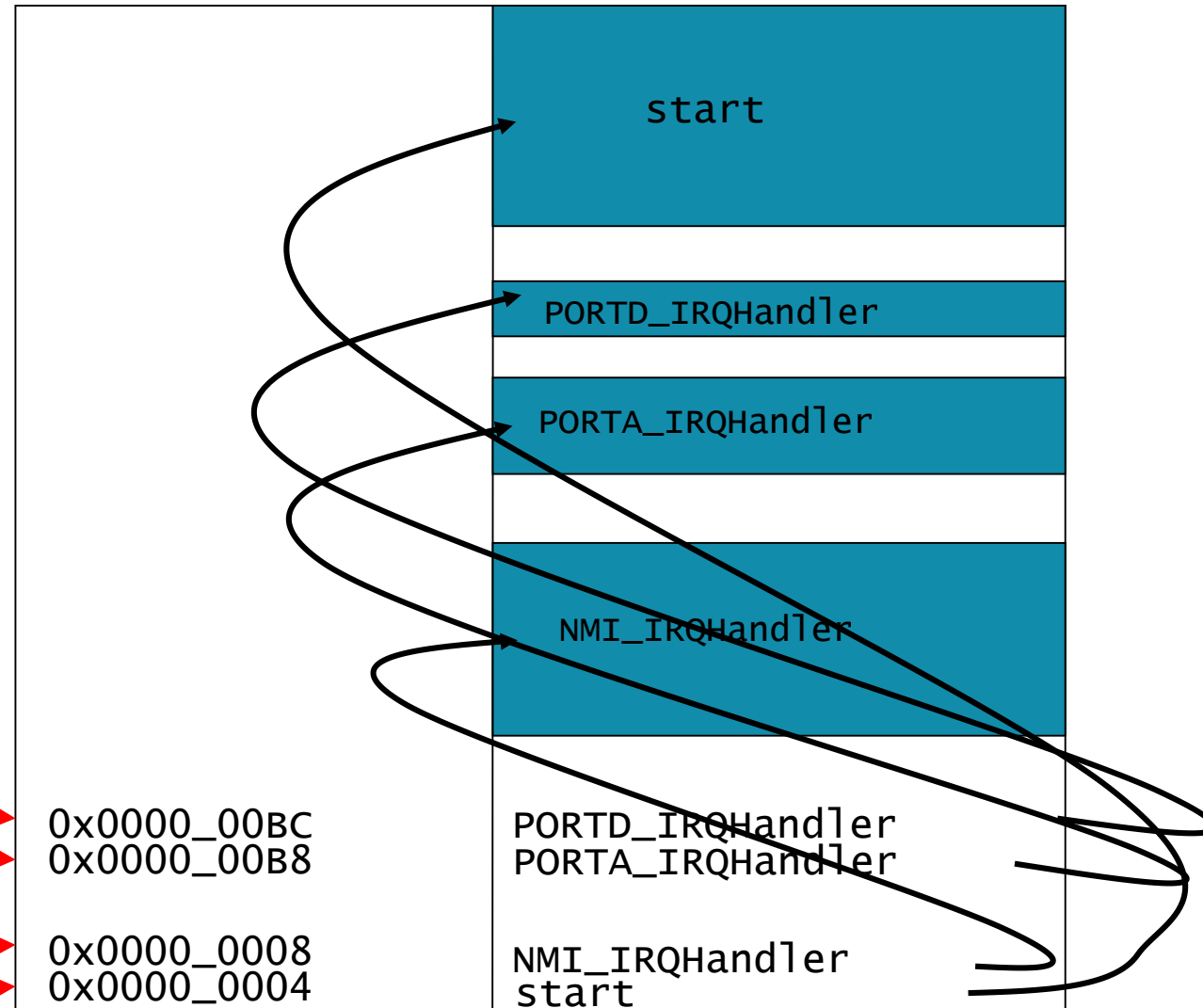
Port D ISR

Port A ISR

*Non-maskable Interrupt
Service Routine*

Port D Interrupt Vector
Port A Interrupt Vector

Non-Maskable Interrupt Vector
Reset Interrupt Vector



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			
9			
8			
7		Reserved	
6			
5			
4			
3	-13	HardFault	0x10
2	-14	NMI	0x0C
		Reset	0x08
1		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

IRQ# + 16 is your interrupt number

IRQ0 starts at 0x40

4*0x1F added to base

0xBC is that starting of the code to handle PROTID

5. Load LR With EXC_RETURN Code

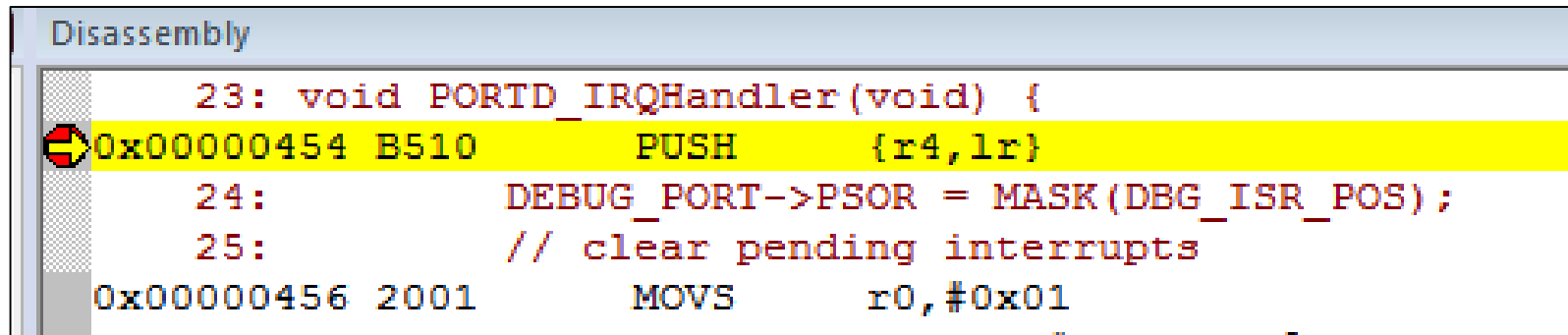
EXC_RETURN	FP Storage	Return Mode	Return Stack	Description
0xFFFF_FFF1	No	0 (Handler)	0 (MSP)	Return to exception handler
0xFFFF_FFF9	No	1 (Thread)	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	No	1 (Thread)	1 (PSP)	Return to thread with PSP
0xFFFF_FFE1	Yes	0 (Handler)	0 (MSP)	Return to exception handler
0xFFFF_FFE9	Yes	1 (Thread)	0 (MSP)	Return to thread with MSP
0xFFFF_FFED	Yes	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

M4 Specific

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

Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000001
R3	0x00000002
R4	0x00000462
R5	0x077A15DC
R6	0x000006CC
R7	0xFB3DFFFD
R8	0xBFFEFEFE
R9	0x200005F0
R10	0xFFFFEBFF
R11	0xEAFD7F7F
R12	0x00000000
R13 (SP)	0x1FFF3E8
R14 (LR)	0xFFFFFFF9
R15 (PC)	0x00000456
xPSR	0x0100002F
Banked	
MSP	0x1FFF3E8
PSP	0xFFBFEEC
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP

SP value reduced since registers have been pushed onto stack

```
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
```

Address	sp
0x1FFF3E8:	00000462 FFFFFFFF9 00000000 00000000 00000001
0x1FFF3FC:	00000002 00000000 0000034F 00000352 01000000
0x1FFF410:	66178BE3 57AC323B F0AA6C90 4B8BCE07 2080424C
0x1FFF424:	EFC30BDF AA0A209D 4EA993EB 4093A563 4CEC3475
0x1FFF438:	EA1ADA60 7E7F388B 4FE50B6E 7B9F7C9E 713C58B0

Saved R4

Saved LR

Saved R0

Saved R1

Saved R2

Saved R3

Saved R12

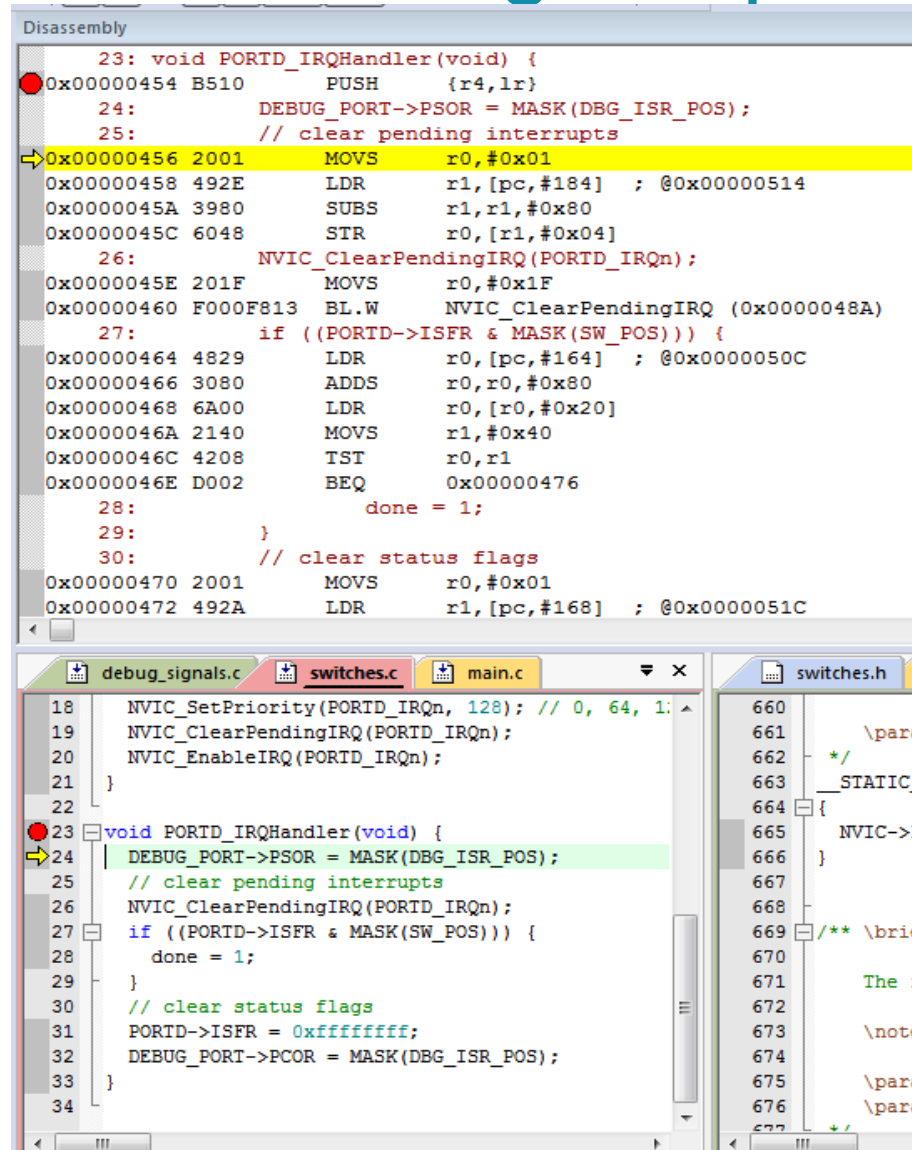
Saved LR

Saved PC

Saved xPSR

Continue Executing Exception Handler

- Execute user code in handler



The screenshot displays a debugger interface with two main panes. The top pane, titled 'Disassembly', shows the assembly code for the `PORTD_IRQHandler` function. The current instruction is highlighted in yellow: `0x00000456 2001 MOVS r0,#0x01`. The bottom pane shows the corresponding C code in `main.c`, with the `PORTD_IRQHandler` function definition visible. The current line of execution is `DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);`, which is highlighted in green. The right side of the bottom pane shows the `switches.h` header file.

```
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
0x00000458 492E LDR r1,[pc,#184] ; @0x00000514
0x0000045A 3980 SUBS r1,r1,#0x80
0x0000045C 6048 STR r0,[r1,#0x04]
26: NVIC_ClearPendingIRQ(PORTD_IRQn);
0x0000045E 201F MOVS r0,#0x1F
0x00000460 F000F813 BL.W NVIC_ClearPendingIRQ (0x0000048A)
27: if ((PORTD->ISFR & MASK(SW_POS))) {
0x00000464 4829 LDR r0,[pc,#164] ; @0x0000050C
0x00000466 3080 ADDS r0,r0,#0x80
0x00000468 6A00 LDR r0,[r0,#0x20]
0x0000046A 2140 MOVS r1,#0x40
0x0000046C 4208 TST r0,r1
0x0000046E D002 BEQ 0x00000476
28: done = 1;
29: }
30: // clear status flags
0x00000470 2001 MOVS r0,#0x01
0x00000472 492A LDR r1,[pc,#168] ; @0x0000051C

main.c
18 NVIC_SetPriority(PORTD_IRQn, 128); // 0, 64, 1;
19 NVIC_ClearPendingIRQ(PORTD_IRQn);
20 NVIC_EnableIRQ(PORTD_IRQn);
21 }
22
23 void PORTD_IRQHandler(void) {
24 DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);
25 // clear pending interrupts
26 NVIC_ClearPendingIRQ(PORTD_IRQn);
27 if ((PORTD->ISFR & MASK(SW_POS))) {
28 done = 1;
29 }
30 // clear status flags
31 PORTD->ISFR = 0xffffffff;
32 DEBUG_PORT->PCOR = MASK(DBG_ISR_POS);
33 }
34

switches.h
660
661 \para
662 */
663 __STATIC_
664 {
665 NVIC->I
666 }
667
668
669 /** \brie
670
671 The f
672
673 \note
674
675 \para
676 \para
677
```

EXITING AN EXCEPTION HANDLER

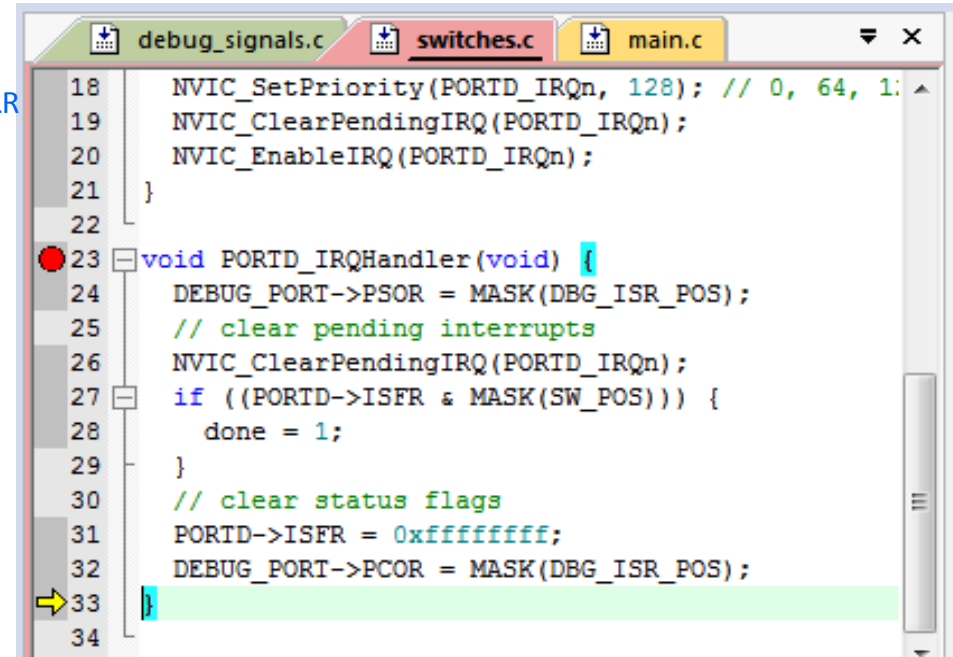
Exiting an Exception Handler

1. Execute instruction triggering exception return processing
2. Select return stack, restore context from that stack
3. Resume execution of code at restored address

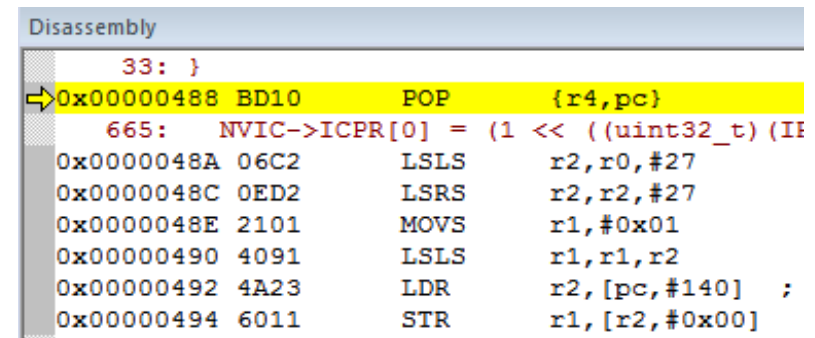
I. Execute Instruction for Exception Return

- 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

If no nested subroutines: BX LR
Else do both



```
debug_signals.c  switches.c  main.c
18  NVIC_SetPriority(PORTD_IRQn, 128); // 0, 64, 1
19  NVIC_ClearPendingIRQ(PORTD_IRQn);
20  NVIC_EnableIRQ(PORTD_IRQn);
21  }
22
23  void PORTD_IRQHandler(void) {
24      DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);
25      // clear pending interrupts
26      NVIC_ClearPendingIRQ(PORTD_IRQn);
27      if ((PORTD->ISFR & MASK(SW_POS))) {
28          done = 1;
29      }
30      // clear status flags
31      PORTD->ISFR = 0xffffffff;
32      DEBUG_PORT->PCOR = MASK(DBG_ISR_POS);
33  }
34
```



```
Disassembly
33: }
⇒ 0x00000488 BD10 POP {r4,pc}
665: NVIC->ICPR[0] = (1 << ((uint32_t) (I
0x0000048A 06C2 LSLS r2,r0,#27
0x0000048C 0ED2 LSRS r2,r2,#27
0x0000048E 2101 MOVS r1,#0x01
0x00000490 4091 LSLS r1,r1,r2
0x00000492 4A23 LDR r2,[pc,#140] ;
0x00000494 6011 STR r1,[r2,#0x00]
```

What Will Be Popped from Stack?

- R4: 0x0000_0462
- PC: 0xFFFF_FFF9

Registers	
Register	Value
Core	
R0	0x00000001
R1	0x400FF040
R2	0xE000E280
R3	0x00000002
R4	0x00000462
R5	0x077A15DC
R6	0x000006CC
R7	0xFB3DFFFD
R8	0xBFFEFEFE
R9	0x200005F0
R10	0xFFFFEBFF
R11	0xEAFD7F7F
R12	0x00000000
R13 (SP)	0x1FFF3E8
R14 (LR)	0x00000465
R15 (PC)	0x00000488
xPSR	0x2100002F
Banked	
MSP	0x1FFF3E8
PSP	0xFFBFFEEC
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP

Disassembly

```
33: }  
→ 0x00000488 BD10 POP {r4,pc}
```

Memory 1	
Address:	sp
0x1FFF3E8:	00000462 FFFFFFF9 00000000 00000000 00000001
0x1FFF3FC:	00000002 00000000 0000034F 00000352 01000000
0x1FFF410:	66178BE3 57AC823B F0AA6C90 4B8BCE07 2080424C
0x1FFF424:	EFC30BDF AA0A209D 4EA993EB 4093A563 4CEC3475
0x1FFF438:	EA1ADA60 7E7F3B8B 4FE50B6E 7B9F7C9E 713C58B0

Call Stack + Locals | Window 1 | Memory 1

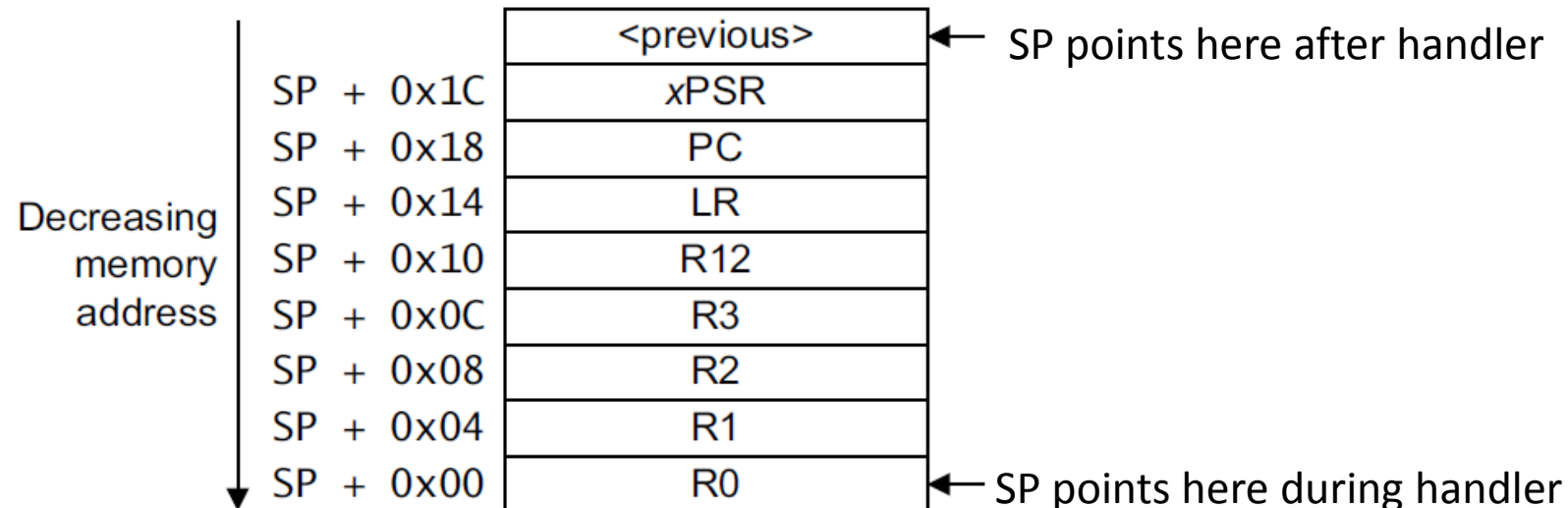
2. Select Stack, Restore Context

- 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

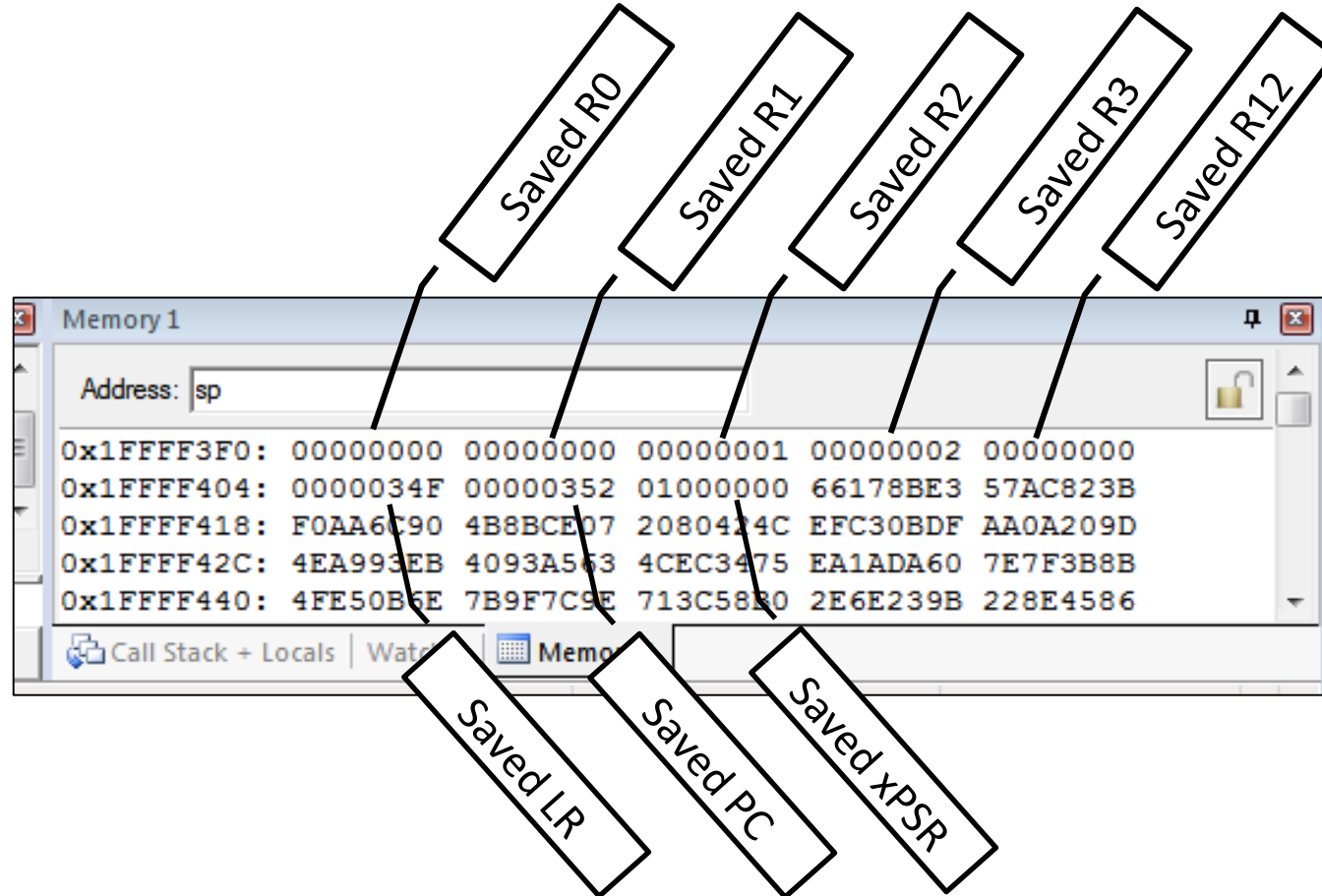
- Pop the registers from that stack

depending on second bit, you will be switching stacks



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

The screenshot displays the CMSIS-DAP Debugger interface with the following components:

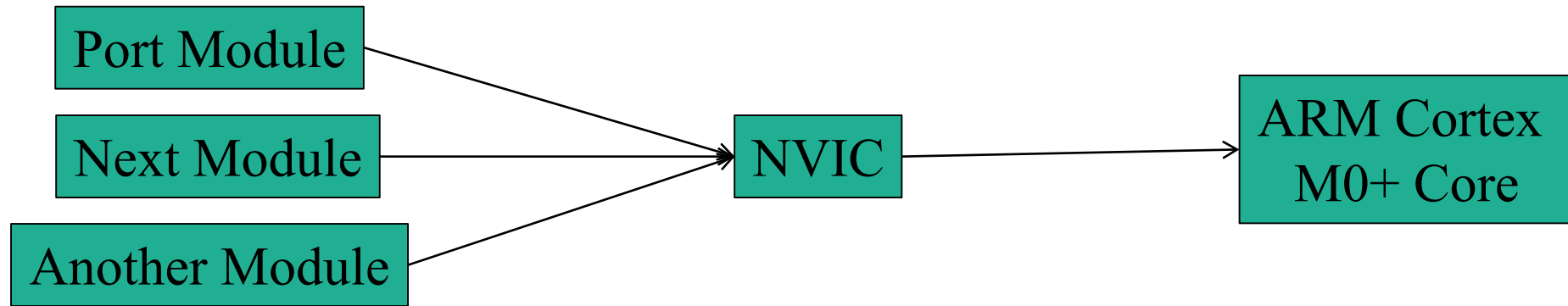
- Registers:** A list of registers (R0-R15, xPSR) and their values. R0, R1, R2, R3, R12, LR, PC, and xPSR are highlighted in blue, indicating they have been restored. R13 (SP) is also highlighted, showing its previous value.
- Disassembly:** A window showing the assembly code at the current instruction address (0x00000352). The instruction is `LDR r0, [pc, #32] ; @0x00000374`.
- Source Code:** A window showing the C source code in `main.c`. The current line is line 43, which is the start of a `while (!done) {` loop. The code includes comments and function calls like `control_RGB_LEDs` and `delay_millisec`.
- Memory:** A window showing the memory address `sp` (Stack Pointer) and its value, `0x1FFFF410`. Below this, a list of memory addresses and their corresponding values is shown.
- Command Window:** A window at the bottom showing the command `BS \\BasicUI\\src\\main.c\\47` and the status of the debugger.

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 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 ² C 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

Vector number is same as exception number
 Vector # = IRQ +16

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

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)

Two interrupts:

User-define and system preserved

At most 32 user defined interrupt (0-31)

Anything below 0 is system preserved (ex. RESET)

Core Exception Mask Register

- Similar to “Global interrupt disable” bit in other MCUs
- PRIMASK - Exception mask register (CPU core)
 - Bit 0: PM Flag Bit 0 : allow interrupt or not
 - 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 none of the interrupt will be serviced
 - `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

User defined interrupt have lower priority than system defined

Special Cases of Prioritization

New interrupt has exact same priority as current execution
Then the one with smaller interrupt number in the stack will be executed first

- 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

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;
- 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

maintain atomicity to prevent interrupts

```
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

competition between multiple tasks may result in mistakes

- **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()