Lecture 8: ARM Cortex-M4 Exceptions and Interrupts

Seyed-Hosein Attarzadeh-Niaki

Some slides due to ARM

Review

- STM32F401 General Purpose IO
  - Port Circuitry
  - Registers
- Cortex Microcontroller System Interface Standard (CMSIS)
  - Components
- GPIO programming using CMSIS
  - Accessing Hardware Registers in C
  - Clocking and Muxing
- Circuit Interfacing
  - Inputs
  - Outputs

Outline

- Exception and Interrupt Concepts
- Cortex-M4 Interrupts
  - NVIC
  - Priorities
- Entering an Exception Handler
- Exiting an Exception Handler

EXCEPTION AND INTERRUPT CONCEPTS

Example System

On board Switch and LEDS

- Goal: Change color of RGB LED when switch is pressed
- Could use on board switch/LED or add external switch/LED

Reviewing the Interrupt Concept: How to Detect if a Switch is Pressed?

Polling
use software to check it

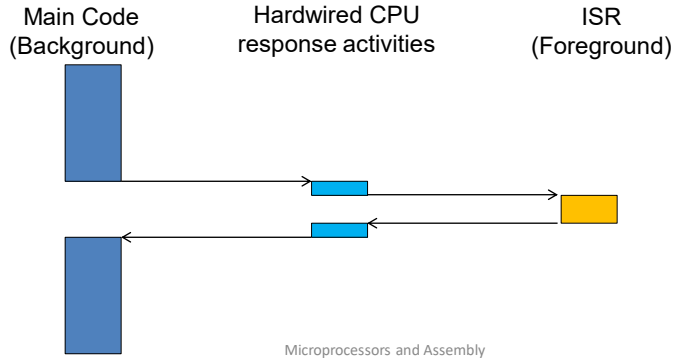
- **Slow** - need to explicitly check to see if switch is pressed
- **Wasteful of CPU time** - the faster a response we need, the more often we need to check
- **Scales badly** - difficult to build system with many activities which can respond quickly. Response time depends on all other processing.

Interrupt
Use HW to detect event and run ISR

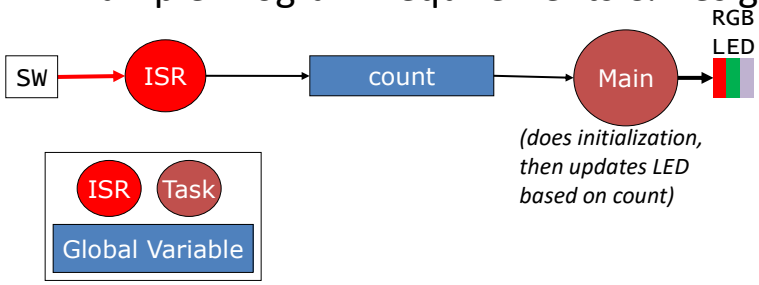
- **Efficient** - code runs only when necessary
- **Fast** - hardware mechanism
- **Scales well**
  - ISR response time doesn't depend on most other processing.
  - Code modules can be developed independently

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



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

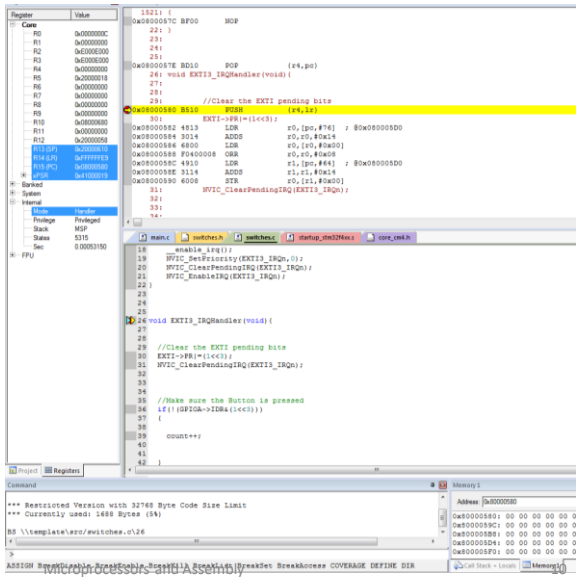
- We will examine processor's response to exception in detail

main.c | switches.h | switches.c*

```
#include "STM32F4xx.h"
#include "switches.h"

volatile unsigned count=0;

void Init_Switch(void){

    RCC->AHB1ENR|=RCC_AHB1ENR_GPIOAEN;


    GPIOA->PUPDR|=GPIO_PUPDR_PUPDR3_0;

    SYSCFG->EXTICR[0]&=SYSCFG_EXTICR1_EXTI3_

    EXTI->IMR |= (1<<3);//Interrupt Mask
    EXTI->FTSR|= (1<<3);//Falling trigger se

    __enable_irq();
    NVIC_SetPriority(EXTI3_IRQn,0);
    NVIC_ClearPendingIRQ(EXTI3_IRQn);
    NVIC_EnableIRQ(EXTI3_IRQn);
}

void EXTI3_IRQHandler(void){


  //Clear the EXTI pending bits
  EXTI->PR|=(1<<3);
  NVIC_ClearPendingIRQ(EXTI3_IRQn);
```


Use Debugger for Detailed Processor View

- Can see registers, stack, source code, disassembly (object code)
- Note: Compiler may generate code for function entry (see address 0x0000_0454)
- Place breakpoint on Handler function declaration line in source code (26), not at first line of function code (27)



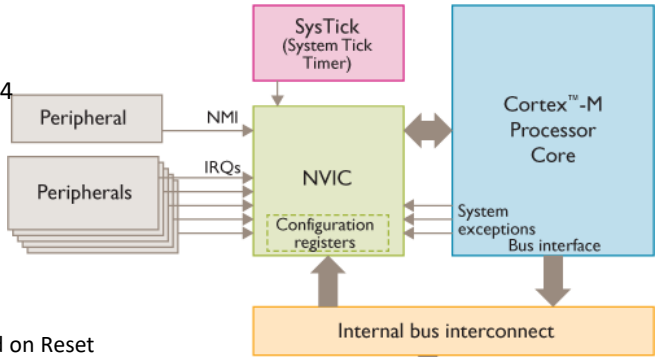
CORTEX-M4 INTERRUPTS

Microcontroller Interrupts

- Types of interrupts
  - Interrupt requests (IRQs)
    - ***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
  - Non-maskable Interrupt (NMI)
    - Similar to IRQs but cannot be disabled (non-maskable)
  - Exceptions, Faults, software interrupts (Generated by core)
    - ***Synchronous***: are the result of specific instructions executing
    - Examples: undefined instructions, overflow occurs for a given instruction
  - SysTick Timer
- 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-M4
- Interrupts are types of exceptions
  - CM4 supports up to 240 IRQs



- **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	No.	Priority	Exception Type	Description
0x0000_0004	1	-3(top)	Reset	Generated by core
0x0000_0008	2	-2	NMI	From on chip peripherals or external sources
0x0000_000C	3	-1	Hard Fault	All fault conditions
0x0000_0014	5	Settable	Bus Fault	Bus error
0x0000_0030	12	Settable	Debug Monitor	Software based debug
0x0000_003C	15	Settable	SYSTICK	Processor timer
0x0000_0040	16	Settable	WWDG	Windows watchdog
……	…	…	…	……
0x0000_0184	81	Settable	FPU	Floating Point interrupt

Check reference manual for complete list.

IRQ assignment in STM32F4xx

Partial Listing – For complete list see STM32F4 ref. manual

INT#	IRQ#	Vector location	Device
1-15		0800 0000 to 0800 003C	CPU Exception
16	0	0800 0040	Window Watchdog interrupt
17	1	0800 0044	PVD through EXTI line detection interrupt
18	2	0800 0048	Tamper and TimeStamp interrupts through the EXTI line
19	3	0800 004C	RTC Wakeup interrupt through the EXTI line
20	4	0800 0050	Flash global interrupt
21	5	0800 0054	RCC global interrupt
22	6	0800 0058	EXTI Line0 interrupt (EXTI0)
23	7	0800 005C	EXTI Line1 interrupt (EXTI1)
24	8	0800 0060	EXTI Line2 interrupt (EXTI2)
25	9	0800 0064	EXTI Line3 interrupt (EXTI3)
26	10	0800 0068	EXTI Line4 interrupt (EXTI4)
27	11	0800 006C	DMA1 Stream0 global interrupt
28	12	0800 0070	DMA1 Stream1 global interrupt
29	13	0800 0074	DMA1 Stream2 global interrupt
30	14	0800 0078	DMA1 Stream3 global interrupt
31	15	0800 007C	DMA1 Stream4 global interrupt
32	16	0800 0080	DMA1 Stream5 global interrupt
33	17	0800 0084	DMA1 Stream6 global interrupt
34	18	0800 0088	ADC1, ADC2 and ADC3 global interrupts
35	19	0800 008C	CAN1 TX interrupts
36	20	0800 0090	CAN1 RX0 interrupts
37	21	0800 0094	CAN1 RX1 interrupt
38	22	0800 0098	CAN1 SCE interrupt
39	23	0800 009C	EXTI9_5 (EXTI Line[9:5])
40	24	0800 00A0	TIM1_BRK_TIM9
41	25	0800 00A4	TIM1_UP_TIM10
42	26	0800 00A8	TIM1_TRG_COM_TIM11
43	27	0800 00AC	TIM1_CC
44	28	0800 00B0	TIM2 (TIM2 global interrupt)
45	29	0800 00B4	TIM3 (TIM3 global interrupt)
46	30	0800 00B8	TIM4 (TIM4 global interrupt)
47	31	0800 00BC	I2C1_EV (I2C1 event interrupt)
48	32	0800 00C0	I2C1_ER (I2C1 error interrupt)
49	33	0800 00C4	I2C2_EV (I2C2 event interrupt)
50	34	0800 00C2	I2C2_ER (I2C2 error interrupt)
51	35	0800 00CC	SPI1 (SPI1 global interrupt)

NVIC Registers

- Registers related to the interrupt of Cortex-M processors are usually inside the NVIC and System Control Block (SCB), both located inside System Control Space (SCS), range from 0xE000 with size of 4KB.
  - **ISER** Interrupt Set Enable Registers ([0] to [7])
  - **ICER** Interrupt Clear Enable Registers ([0] to [7])
  - **ISPR** Interrupt Set Pending Register ([0] to [7])
  - **ICPR** Interrupt Clear Pending Registers ([0] to [7])
  - **IABR** Interrupt Active bit Registers ([0] to [7])
  - **IP** Interrupt Priority Registers ([0] to [239])
  - **STIR** Software Trigger Interrupt Register
- And special registers **PRIMASK**, **FAULTMASK**, and **BASEPRI** are for interrupt masking.
- Most of these registers can be only accessed in privileged level.
- Direct access to registers is less practical for general application programming.
  - The more usual way is to use the CMSIS-Core access functions.

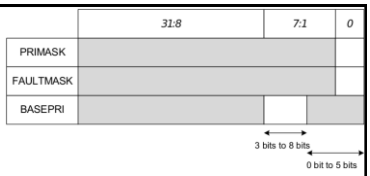
CMSIS support: IRQ type and functions

- CMSIS-Core provides a way to identify interrupt using the *interrupt enumeration* in which system exceptions use negative values. They are "typedefed" as "IRQn"

```
typedef enum IRQn
{
/******  Cortex-M4 Processor Exceptions Numbers ***********************************************************/
  NonMaskableInt_IRQn        = -14,    /*!< 2 Non Maskable Interrupt                                */
  MemoryManagement_IRQn      = -12,    /*!< 4 Cortex-M4 Memory Management Interrupt                 */
  BusFault_IRQn              = -11,    /*!< 5 Cortex-M4 Bus Fault Interrupt                         */
  UsageFault_IRQn            = -10,    /*!< 6 Cortex-M4 Usage Fault Interrupt                       */
  SVCall_IRQn                = -5,     /*!< 11 Cortex-M4 SV Call Interrupt                          */
  DebugMonitor_IRQn          = -4,     /*!< 12 Cortex-M4 Debug Monitor Interrupt                    */
  PendSV_IRQn                = -2,     /*!< 14 Cortex-M4 Pend SV Interrupt                          */
  SysTick_IRQn               = -1,     /*!< 15 Cortex-M4 System Tick Interrupt                      */
/******  STM32 specific Interrupt Numbers *****************************************************************/
  WWDG_IRQn                  = 0,      /*!< Window WatchDog Interrupt                               */
  PVD_IRQn                   = 1,      /*!< PVD through EXTI Line detection Interrupt               */
  TAMP_STAMP_IRQn            = 2,      /*!< Tamper and TimeStamp interrupts through the EXTI line   */
  RTC_WKUP_IRQn              = 3,      /*!< RTC Wakeup interrupt through the EXTI line              */
  FLASH_IRQn                 = 4,      /*!< FLASH global Interrupt                                  */
  RCC_IRQn                   = 5,      /*!< RCC global Interrupt                                    */
```

Functions	Usage
void NVIC_EnableIRQ (IRQn_Type IRQn)	Enable an external interrupt
void NVIC_DisableIRQ (IRQn_Type IRQn)	Disable an external interrupt
void NVIC_SetPriority (IRQn_Type IRQn, uint32_t priority)	Set the priority of an interrupt
void __enable_irq(void)	Clear PRIMASK to enable interrupts
void __disable_irq(void)	Set PRIMASK to disable all interrupts
void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)	Set priority grouping configuration

Core Exception Mask Registers



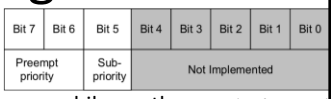
- Similar to "Global interrupt disable" bit in other MCUs
- **PRIMASK** - Exception mask register (CPU core)
  - Bit 0: PM Flag
    - Set to 1 to prevent activation of all exceptions with configurable priority
    - Clear to 0 to allow activation of all exception
  - Access using CPSIE/CPSID, MSR and MRS instructions
  - Use to prevent data race conditions with code needing atomicity
- **FAULTMASK** similar to PRIMASK but blocks hardfault as well
- CMSIS-CORE API
  - **`void __enable_irq()`** – Clear PRIMASK to enable interrupt
  - **`void __disable_irq()`** – Set PRIMASK to disable interrupts
  - **`uint32_t __get_PRIMASK()`** - returns value of PRIMASK
  - **`void __set_PRIMASK(uint32_t x)`** - sets PRIMASK to x
  - **`uint32_t __get_FAULTMASK()`** – returns value of FAULTMASK
  - **`void ___set_FAULTMASK(uint32_t x)`** – sets FAULTMASK 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 (up to 256 levels)
  - For STM32F4 families, only 4 higher-order bits of the priority level are implemented.
  - Which means 16 level of programmable priority levels
  - 0x10,0x20,0x30,0x40,0x50,0x60,0x70,0x80,0x90,0xA0,0xB0,0xC0,0XD0 ,0xE0,0xF0 (The larger number represents lower priority).
- **BASEPRI** masks exceptions based on priority level
  - `uint32_t __get_BASEPRI();`
  - `__set_PRIMASK(uint32_t basePri);`

Priority Grouping

- 4 Bits priority level could be split into two halves
  - Preempt Priority Field (upper half)
  - Subpriority Field (lower half)



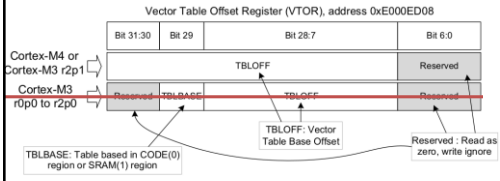
(not for stm32F4)

- Preempt Priority Level defines if one interrupt could happen while another interrupt handler is running
- Subpriority defines which interrupt will be served when several interrupts with the same Preempt Priority level happen at the same time
- By default the STM32F4 families have 4 bits preempt priority bits and 0 bit of subpriority field.

	Functions	Usage
void	NVIC_SetPriorityGrouping (uint32_t PriorityGroup)	Set priority grouping value
uint32_t	NVIC_GetPriorityGrouping (uint32_t PriorityGroup)	Get priority grouping value
uint32_t	NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t Pre emptPriority, uint32_t Sub priority)	Generate encoded priority value based on priority grouping, group priority and sub-priority
void	NVIC_DecodePriority (uint32_t Priority, uint32_t PriorityGroup, uint32_t *pPre emptPriority, uint32_t *pSub priority)	Extract, group priority and sub-priority from a priority value

Vector Table

- Upon accepting an exception request, CM4 needs to determine the starting address of the exception handler (or ISR)
- Stored in the vector table in the memory
  - by default, starts at memory address 0
  - arranged according to the exception number x 4
  - defined in the startup codes provided by the microcontroller vendors
- Vector Table Relocation
  - Via a programmable register called the Vector Table Offset Register (VTOR)
  - Use cases
    1. Devices with boot loader
    2. Applications load into RAM
    3. Dynamic changing of vector

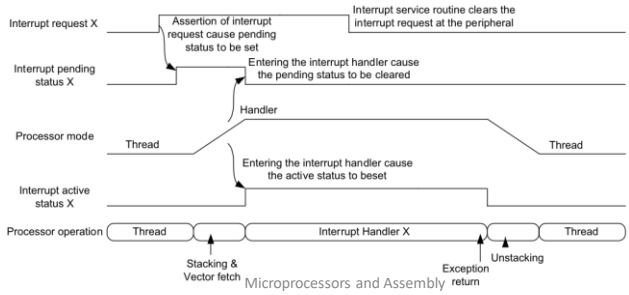


Memory Address		Exception Number
	1	
	1	
0x0000004C	Interrupt#3 vector 1	19
0x00000048	Interrupt#2 vector 1	18
0x00000044	Interrupt#1 vector 1	17
0x00000040	Interrupt#0 vector 1	16
0x0000003C	SysTick vector 1	15
0x00000038	PendSV vector 1	14
0x00000034	Not used	13
0x00000030	Debug Monitor vector 1	12
0x0000002C	SVC vector 1	11
0x00000028	Not used	10
0x00000024	Not used	9
0x00000020	Not used	8
0x0000001C	Not used	7
0x00000018	Usage Fault vector 1	6
0x00000014	Bus Fault vector 1	5
0x00000010	MemManage vector 1	4
0x0000000C	HardFault vector 1	3
0x00000008	NMI vector 1	2
0x00000004	Reset vector 1	1
0x00000000	MSP initial value	0

Note : LSB of each vector must be set to 1 to indicate Thumb state

Interrupt Status and Pending Behavior

- Three status attributes applicable to each interrupt
  - Disabled (default) or enabled
  - Pending (a request is waiting to be served) or not pending
  - Active (being served) or inactive
- Various possible combinations
- This design enables the pulsed interrupt request.
  - For traditional ARM processors, interrupts must be held while waiting to be served.
  - For CM4, pulsed interrupt will set the pending bit until explicitly cleared.
  - When processor starts to process an interrupt request, the pending status is cleared automatically
  - In many microcontroller designs, the peripherals operate with level-triggered interrupts
    - the ISR will have to clear the interrupt request manually
    - for example, by writing to a register in the peripheral



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

ENTERING AN EXCEPTION HANDLER

CPU's Hardwired Exception Processing

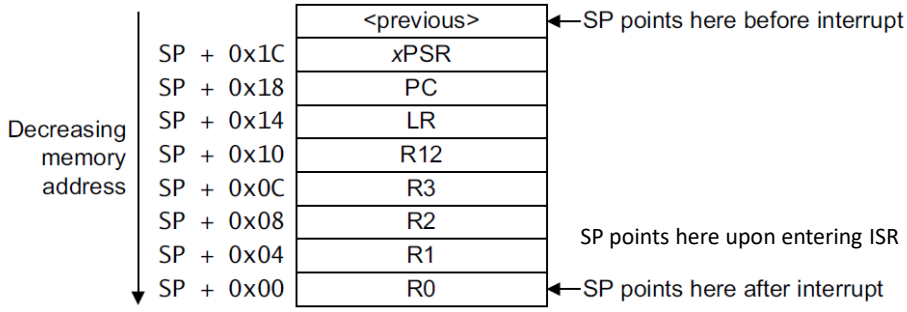
1. Finish current instruction (except for lengthy instructions)
2. Push context (8 32-bit words) onto current stack (MSP or PSP)
   - xPSR, Return address, LR (R14), R12, R3, R2, R1, R0
3. Switch to handler/privileged mode, use MSP
4. Load PC with address of exception handler
5. Load LR with EXC_RETURN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

Only 12 cycles from exception request to execution of first instruction in handler (interrupt latency) if zero memory system latency and bus supports vector fetch and stacking at the same time.

1. Finish Current Instruction

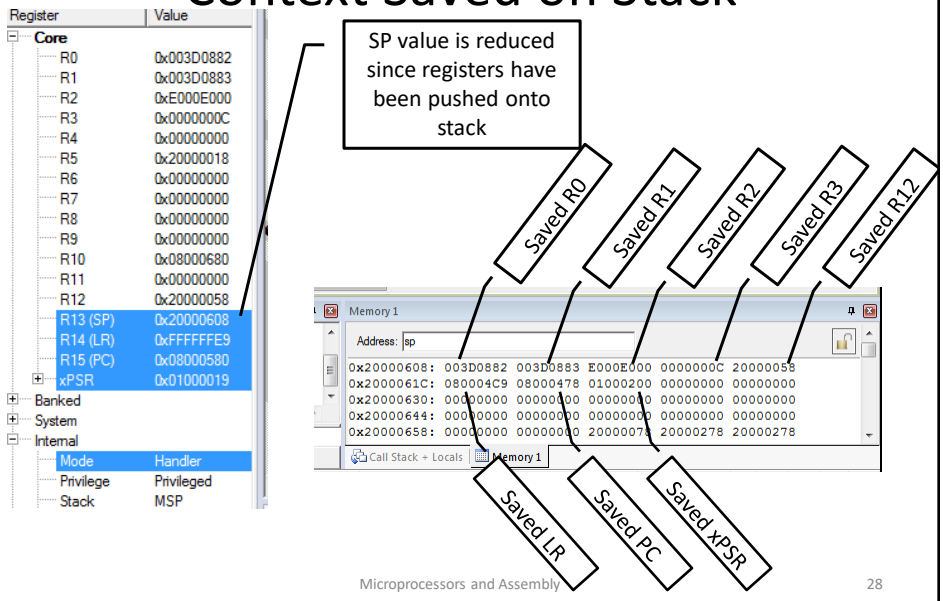
- Most instructions are short and finish quickly
- Some instructions may take many cycles to execute
  - Load Multiple (LDM or LDRD), Store Multiple (STM or STRD), Push, Pop, multiplication, division, etc.
- If one of these is executing when the interrupt is requested, the processor:
  - abandons the instruction
  - responds to the interrupt
  - executes the ISR
  - returns from interrupt
  - restarts the abandoned instruction
- Exception: if multiple load/store/push/pop is part of an IF-THEN instruction block, will be cancelled and restarted.

2. Push Context onto Current Stack



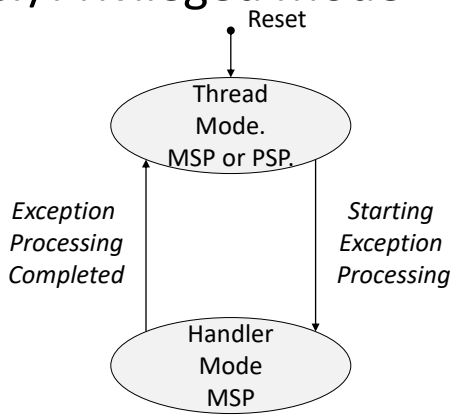
- The order that stack will be accessed is different from the stack frame order. E.g., PC is stored first so that it can be updated.
- Two SPs: Main (MSP), process (PSP)
- Which is active depends on operating mode, CONTROL register bit 1
- Stack grows toward smaller addresses

Context Saved on Stack

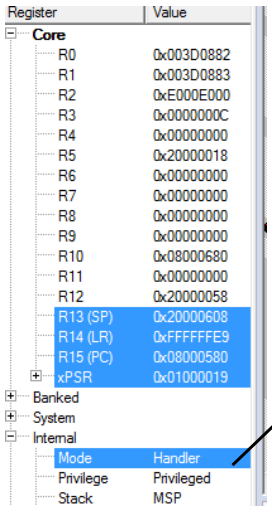


3. Switch to Handler/Privileged Mode

- Handler mode always uses Main SP

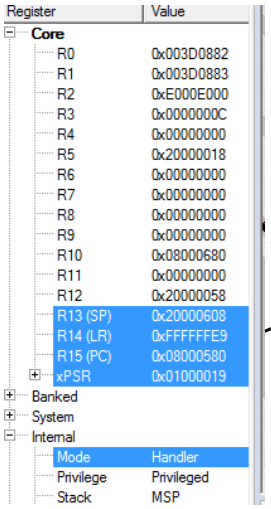


Handler and Privileged Mode



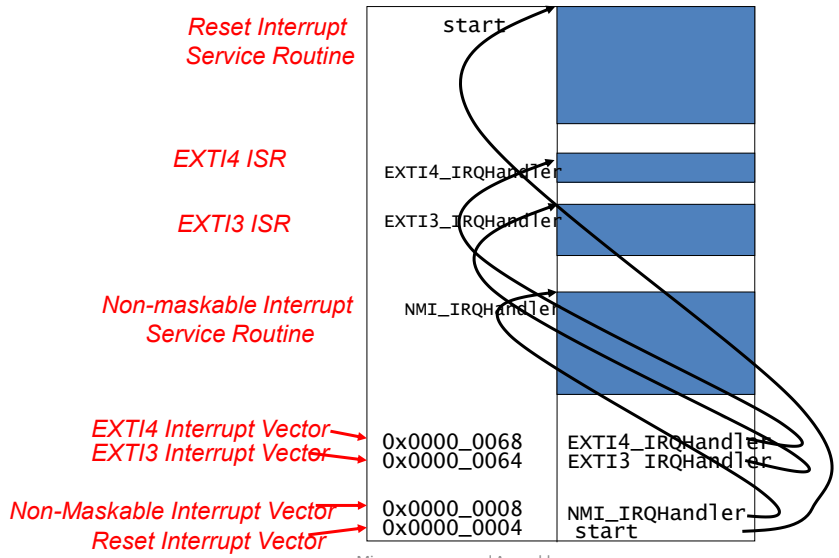
Mode changed to Handler. Was already using MSP and in Privileged mode

Update IPSR with Exception Number



EXTI3_IRQ is Exception number 0x19 (interrupt number + 0x10)

4. Load PC With Address Of Exception Handler



Can Examine Vector Table With Debugger

Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.		.	.
.		.	.
.		.	.
		0x004C	
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			
9			Reserved
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

Memory 1

Address: 0x00000064

```
0x00000064: 08000581 08000243 08000243 08000243 08000243
0x00000078: 08000243 08000243 08000243 08000243 08000243
0x0000008C: 08000243 08000243 08000243 08000243 08000243
0x000000A0: 08000243 08000243 08000243 08000243 08000243
0x000000B4: 08000243 08000243 08000243 08000243 08000243
```

- EXTI3 ISR is IRQ #9 (0x09), so vector to handler begins at 0x40+4*0x09 = 0x64
- Why is the vector odd? 0x0800_0581
- LSB of address indicates that handler uses Thumb code

Upon Entry to Handler

Register	Value
Core	
R0	0x003D0882
R1	0x003D0883
R2	0xE000E000
R3	0x0000000C
R4	0x00000000
R5	0x20000018
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x08000680
R11	0x00000000
R12	0x20000058
R13 (SP)	0x20000608
R14 (LR)	0xFFFFFFE9
R15 (PC)	0x0800058
xPSR	0x01000019
Banked	
System	
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP

```
29:         //Clear the EXTI pending bits
0x08000580 B510      PUSH        {r4,lr}
30:         EXTI->PR|=(1<<3);
0x08000582 4813      LDR         r0,[pc,#76]  ; @0x080005D0
0x08000584 3014      ADDS        r0,r0,#0x14
0x08000586 6800      LDR         r0,[r0,#0x00]
0x08000588 F0400008  ORR         r0,r0,#0x08
0x0800058C 4910      LDR         r1,[pc,#64]  ; @0x080005D0
0x0800058E 3114      ADDS        r1,r1,#0x14
0x08000590 6008      STR         r0,[r1,#0x00]
31:         NVIC_ClearPendingIRQ(EXTI3_IRQn);
```

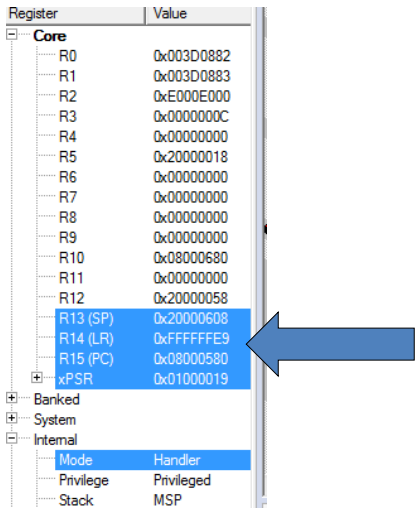
PC has been loaded with start address of handler

5. Load LR With EXC_RETURN Code

EXC_RETURN	Return Mode	Return Stack	Description
0xFFFF_FFE1	0 (Handler)	0 (MSP)	Return to handler mode (always Main Stack)
0xFFFF_FFE9	1 (Thread)	0 (MSP)	Return to thread with MSP
0xFFFF_FFED	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
  - With bit [4:7], using floating point unit or not (E, yes; F, no)

Updated LR With EXC_RETURN Code



- 0xFFFF_FFE9
  - Return to Thread mode and use the Main Stack for return
  - Floating Point Unit was used before interrupt (FPCA=1)

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

```
    29:          //Clear the EXTI pending bits
0x08000580 B510      PUSH     {r4,lr}
    30:          EXTI->PR|=(1<<3);
0x08000582 4813      LDR      r0,[pc,#76]  ; @0x080005D0
0x08000584 3014      ADDS     r0,r0,#0x14
0x08000586 6800      LDR      r0,[r0,#0x00]
0x08000588 F0400008  ORR      r0,r0,#0x08
```

Core

R0	0x0019DBF0
R1	0x0019DBF1
R2	0xE000E000
R3	0x0000000C
R4	0x00000000
R5	0x20000018
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x08000680
R11	0x00000000
R12	0x20000058
R13 (SP)	0x20000600
R14 (LR)	0xFFFFFFE9
R15 (PC)	0x08000582
xPSR	0x01000019
Banked	
System	
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP
States	1312739499
Sec	131.27394990
FPU	

SP value reduced since registers have been pushed onto stack

Saved R4

Saved LR

Saved R0

Saved R1

Saved R2

Memory 1

Address: sp

```
0x20000600: 00000000 FFFFFFE9 0019DBF0 0019DBF1 E000E000
0x20000614: 0000000C 20000058 080004B1 08000478 01000200
0x20000628: 00000000 00000000 00000000 00000000 00000000
0x2000063C: 00000000 00000000 00000000 00000000 00000000
0x20000650: 00000000 00000000 00000000 00000000 20000078
```

Saved R3

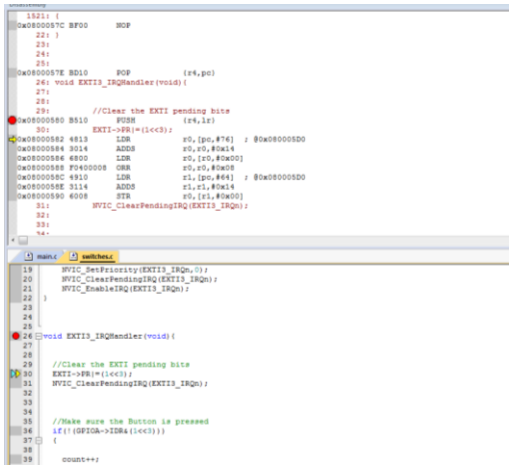
Saved R12

Saved LR

Saved PC

Saved xPSR

Continue Executing Exception Handler



```
  1521: {
0x080057C BF00      NOP
    22: }
    23:
    24:
    25:
0x080057E BD10      POP             {r4,pc}
    26: void EXTI3_IRQHandler(void){
    27:
    28:
    29:     //Clear the EXTI pending bits
0x0800580 B510      PUSH            {r4,lr}
    30:     EXTI->PR|=(1<<3);
0x0800582 4813      LDR             r0,[pc,#76]  ; @0x080005D0
0x0800584 3014      ADDS            r0,r0,#0x14
0x0800586 6800      LDR             r0,[r0,#0x00]
0x0800588 F0400008  ORR             r0,r0,#0x08
0x080058C 4910      LDR             r1,[pc,#64]  ; @0x080005D0
0x080058E 3114      ADDS            r1,r1,#0x14
0x0800590 6008      STR             r0,[r1,#0x00]
    31:     NVIC_ClearPendingIRQ(EXTI3_IRQn);
    32:
    33:
    34:
```

```
main.c  switches.c
19      NVIC_SetPriority(EXTI3_IRQn,0);
20      NVIC_ClearPendingIRQ(EXTI3_IRQn);
21      NVIC_EnableIRQ(EXTI3_IRQn);
22  }
23
24
25
26  void EXTI3_IRQHandler(void){
27
28
29    //Clear the EXTI pending bits
30    EXTI->PR|=(1<<3);
31    NVIC_ClearPendingIRQ(EXTI3_IRQn);
32
33
34
35    //Make sure the Button is pressed
36    if(!(GPIOA->IDR&(1<<3)))
37    {
38
39      count++;
```

- Then execute user code in handler

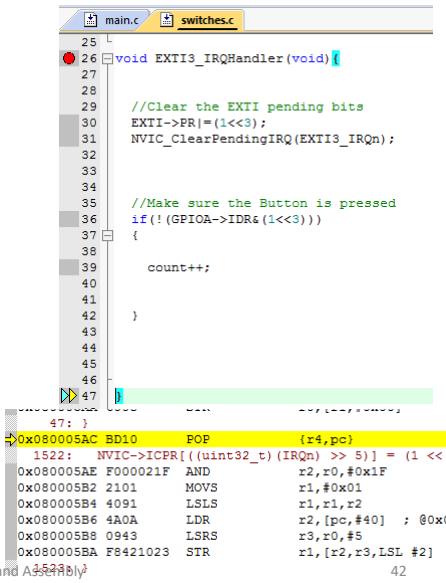
EXITING AN EXCEPTION HANDLER

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

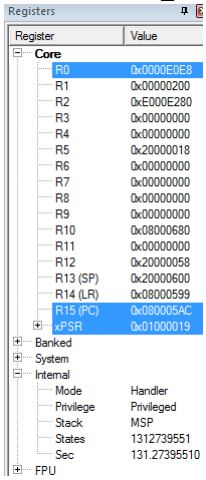
1. Execute Instruction for Exception Return

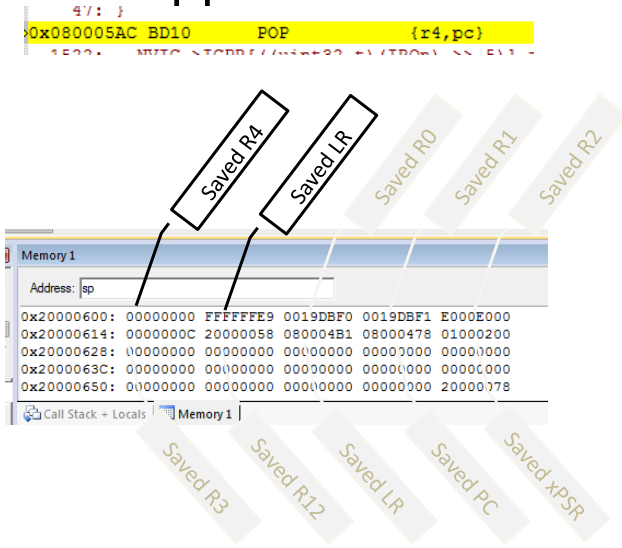
- No “return from interrupt” instruction
- Use regular instruction instead
  - BX LR - Branch to address in LR by loading PC with LR contents
  - POP …, PC - Pop address from stack into PC
- … with a special value EXC_RETURN loaded into the PC to trigger exception handling processing
  - BX LR used if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



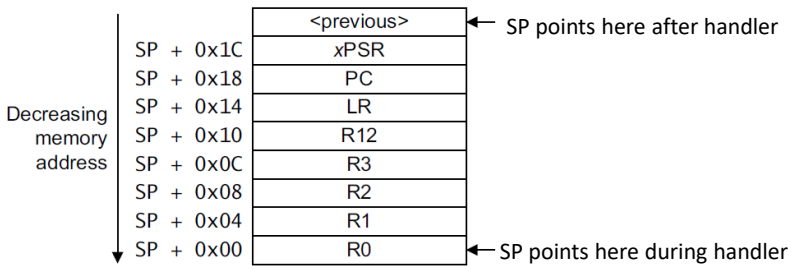


2. Select Stack, Restore Context

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

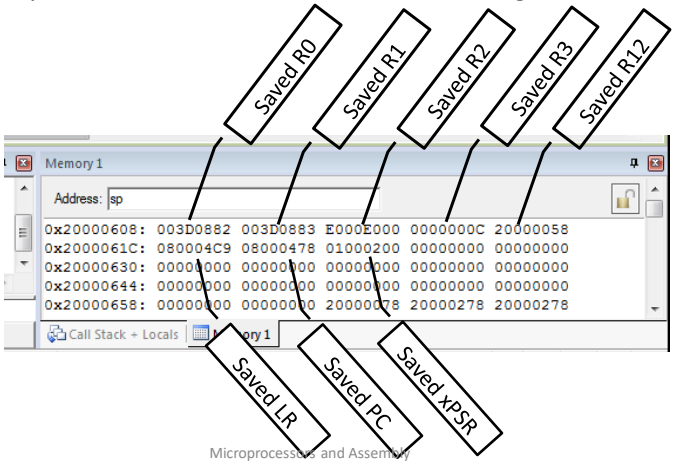
EXC_RETURN	Return Stack	Description
0xFFFF_FFE1	0 (MSP)	Return to exception handler with MSP
0xFFFF_FFE9	0 (MSP)	Return to thread with MSP
0xFFFF_FFED	1 (PSP)	Return to thread with PSP

- Pop the registers from that stack



Example

- PC=0xFFFF_FFE9, 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 0x0800_04D6