

# Module 2-B

Interrupts and Exceptions

# Reading

- Reading assignment:
  - STM32F0x1 Family Reference, Chapter 12, pages 217 – 228, "Interrupts and events"
  - Textbook, Chapter 11, "Interrupts", pages 237 – 268.
    - Note: The interrupt number ranges are all wrong since they are for the STM32L (ARMv7-M) rather than the STM32F0xx.
      - It's good for concepts rather than details.
  - Textbook, Chapter 15, "General Purpose Timers", pages 373 – 414.
    - If you just read section 15.1 you will do yourself a tremendous favor.

# Learning Outcome #2

“an ability to interface a microcontroller to various devices.”

**How?**

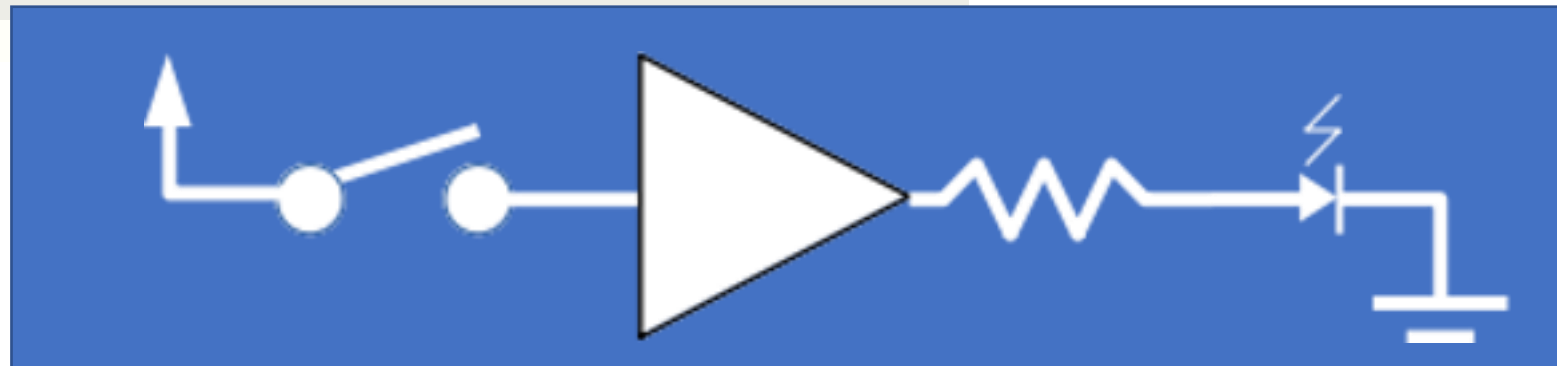
- ✓ A. General Purpose I/O
- B. Interrupts and Exceptions
- C. Basic Timers
- D. Embedded C
- E. Advanced Timers
- F. Debouncing and Multiplexing

## Recall

# Reading the switch value

Is this an efficient way to read the button value?

```
1 .equ GPIOA, 0x48000000 // Base address of GPIOA control
2 .equ GPIOC, 0x48000800 // Base address of GPIOC control registers
3 .equ IDR, 0x10          // Offset of the IDR
4 .equ ODR, 0x14          // Offset of the ODR
5
6     ldr r0, =GPIOA       // Load, into R0, base address of GPIOA
7     ldr r1, =GPIOC       // Load, into R1, base address of GPIOC
8     movs r3, #1
9 again:
10    ldr r2, [r0, #IDR]    // Load, into R2, value of GPIOA_IDR
11    ands r2, r3           // Look at only bit zero.
12    lsls r2, #8           // Translate bit 0 to bit 8
13    str r2, [r1, #ODR]    // Store value to GPIOC_ODR
14    b again
```



## Better way than a forever loop?

- Wastes CPU: processing time and energy
  - Most of the time, nothing changes (even with a 48MHz CPU)
  - Technique is also called **polling**.
- Better to just get notified when button is pushed
  - This is called an **interrupt**.
  - Interrupt: specific type of an **exception** (others: faults, traps, reset)
- **Exception**: A hardware-invoked subroutine call
  - Subroutine is called an **Interrupt Service Routine** (ISR), or **interrupt handler**.

# Interrupt Basics

- Peripherals can **raise** interrupts.
- CPU can be interrupted by more than one thing
  - Call a different function Interrupt Service Routine (ISR) for each thing
    - Stored in the interrupt **vector table**

Simplified example Vector Table	
Interrupt number	ISR
0	handle_button
1	handle_atd
2	handle_pwm
3	handle_reset

# Interrupt Basics

- Peripherals can **raise** interrupts.
- CPU can be interrupted by more than one thing
  - Call a different function Interrupt Service Routine (ISR) for each thing
    - Stored in the interrupt **vector table**
- Can be enabled/disabled

Interrupt number	ISR	EN
0	handle_button	0
1	handle_atd	0
2	handle_pwm	0
3	handle_reset	1

# Interrupt Basics

- Peripherals can **raise** interrupts.
- CPU can be interrupted by more than one thing
  - Call a different function Interrupt Service Routine (ISR) for each thing
    - Stored in the interrupt **vector table**
- Can be enabled/disabled
- Interrupts have priorities
  - Higher priority can **preempt** lower

Interrupt number	ISR	EN	Pri
0	handle_button	0	4
1	handle_atd	0	3
2	handle_pwm	0	2
3	handle_reset	1	1



# Interrupt Basics


- Peripherals can **raise** interrupts.
- CPU can be interrupted by more than one thing
  - Call a different function Interrupt Service Routine (ISR) for each thing
    - Stored in the interrupt **vector table**
- Can be enabled/disabled
- Interrupts have priorities
  - Higher priority can **preempt** lower
- Raised, but not yet handled interrupts are called **pending**.

Interrupt number	ISR	EN	Pri	Pend
0	handle_button	0	4	0
1	handle_atd	0	3	0
2	handle_pwm	0	2	1
3	handle_reset	1	1	0

# Interrupt Basics

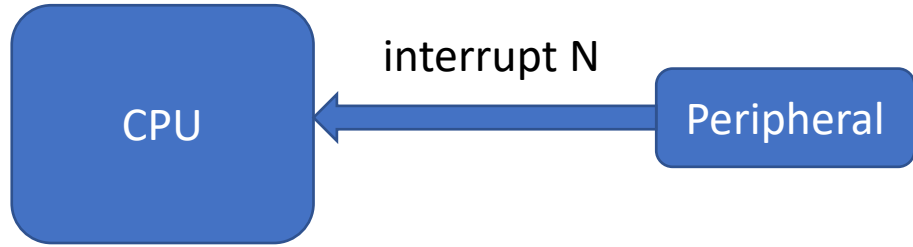
- Peripherals can **raise** interrupts.
- CPU can be interrupted by more than one thing
  - Call a different function Interrupt Service Routine (ISR) for each thing
    - Stored in the interrupt **vector table**
- Can be enabled/disabled
- Interrupts have priorities
  - Higher priority can **preempt** lower
- Raised, but not yet handled interrupts are called **pending**.
- Handler must tell peripheral it **acknowledges** its request.

Each column can be its own table, but conceptually each interrupt has these properties



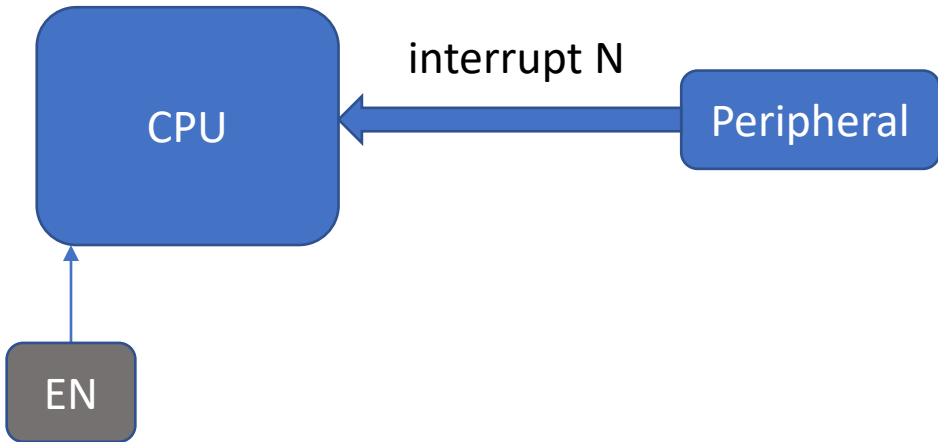
Interrupt number	ISR	EN	Pri	Pend
0	handle_button	0	4	0
1	handle_atd	0	3	0
2	handle_pwm	0	2	1
3	handle_reset	1	1	0

# Flow of an interrupt



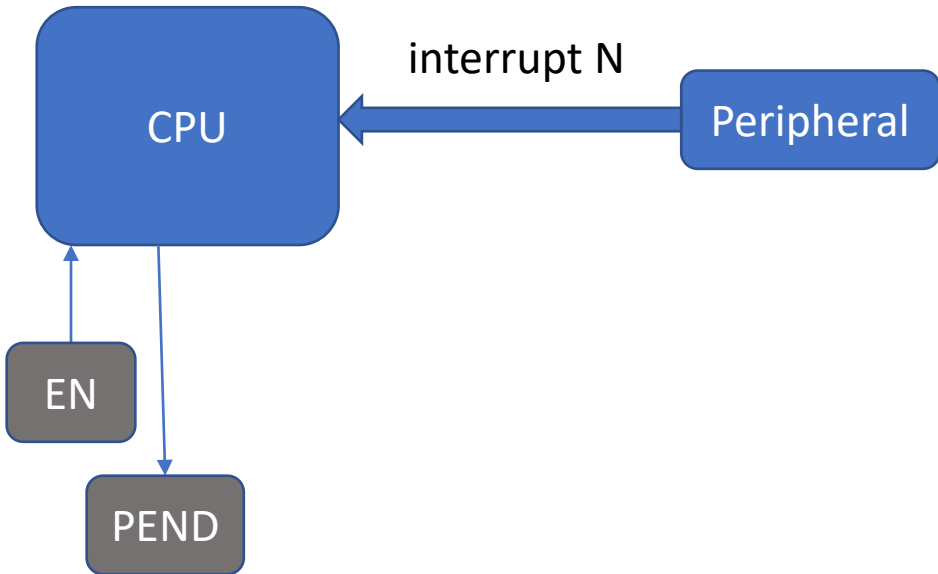
1. Peripheral **raises** a particular interrupt

# Flow of an interrupt



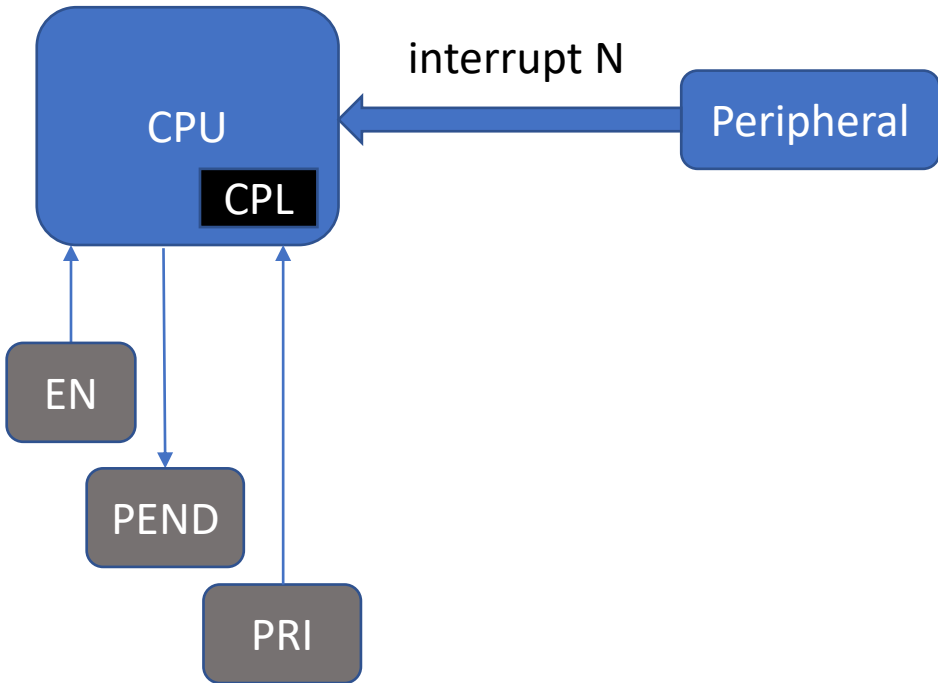
1. Peripheral **raises** a particular interrupt
2. CPU checks if "N" is enabled

# Flow of an interrupt



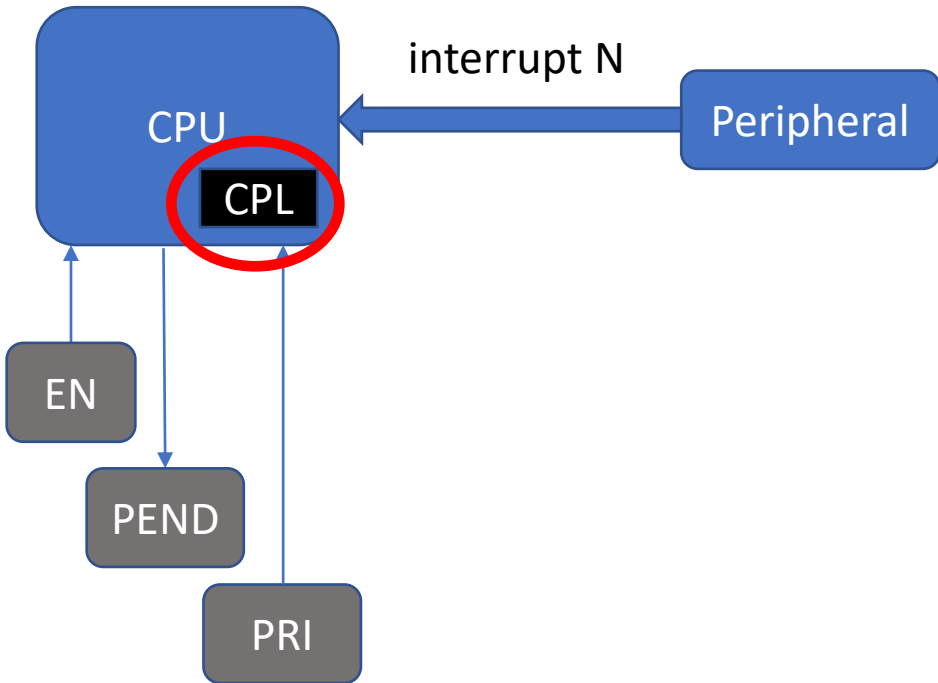
1. Peripheral **raises** a particular interrupt
2. CPU checks if "N" is enabled
3. If EN:  
Mark "N" as **pending**

# Flow of an interrupt



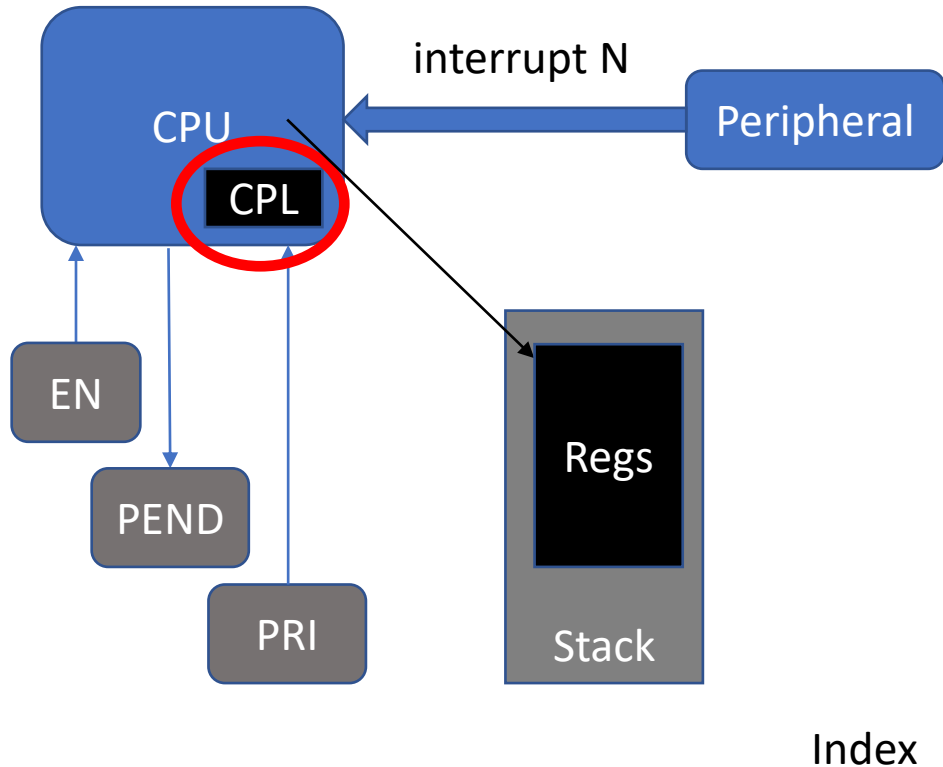
1. Peripheral **raises** a particular interrupt
2. CPU checks if "N" is enabled
3. If EN:  
Mark "N" as **pending**
4. CPU checks priority of "N" versus the current priority level (CPL) – as it might be already serving an interrupt.

# Flow of an interrupt



1. Peripheral **raises** a particular interrupt
2. CPU checks if "N" is enabled
3. If EN:  
Mark "N" as **pending**
4. CPU checks priority of "N" versus the current priority level (CPL) – as it might be already serving an interrupt.
5. If  $\text{Priority-N} > \text{CPL}$ :  
Update  $\text{CPL} = \text{Priority-N}$

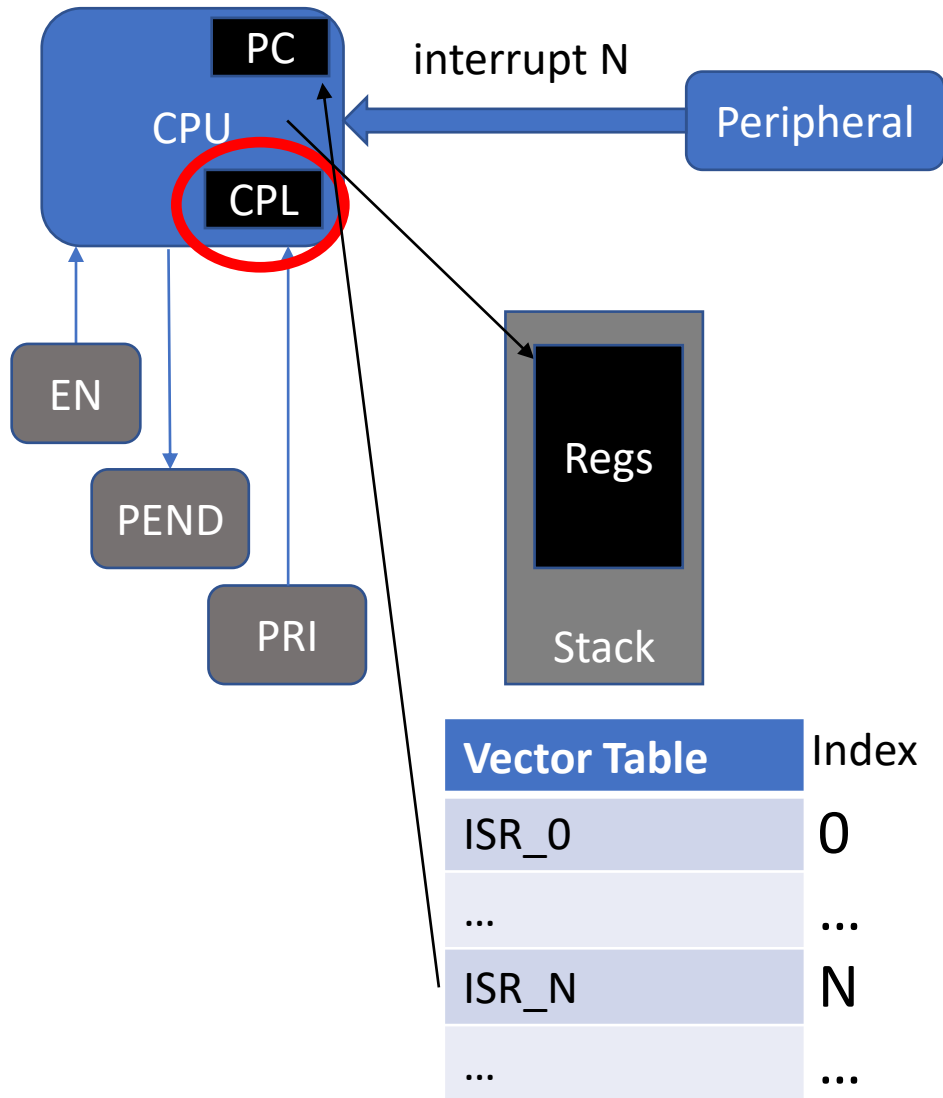
# Flow of an interrupt



1. Peripheral ***raises*** a particular interrupt
2. CPU checks if "N" is enabled
3. If EN:  
Mark "N" as ***pending***
4. CPU checks priority of "N" versus the current priority level (CPL) – as it might be already serving an interrupt.
5. If  $\text{Priority-N} > \text{CPL}$ :  
Update  $\text{CPL} = \text{Priority-N}$   
Push Registers to stack

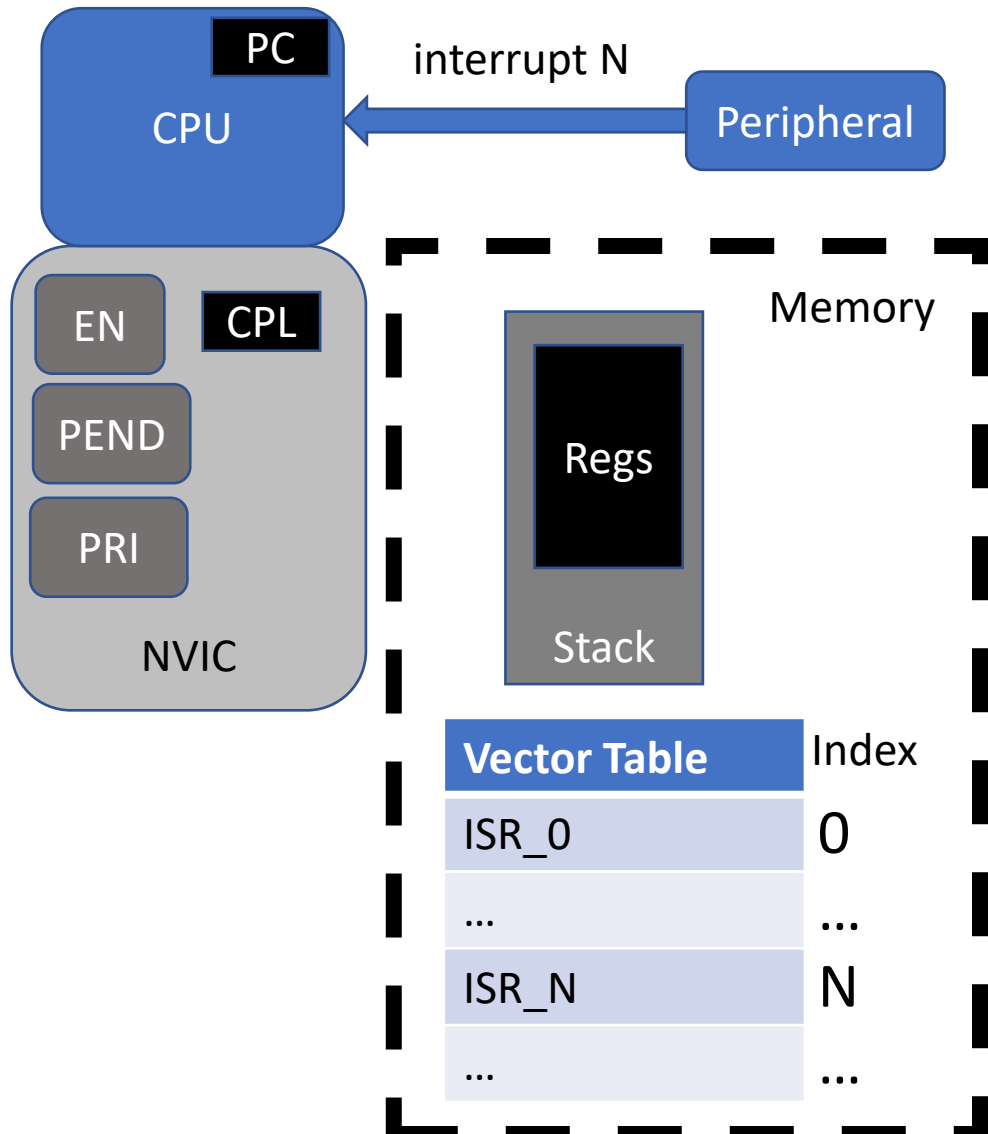


# Flow of an interrupt



1. Peripheral **raises** a particular interrupt
2. CPU checks if "N" is enabled
3. If EN:
  - Mark "N" as **pending**
4. CPU checks priority of "N" versus the current priority level (CPL) – as it might be already serving an interrupt.
5. If Priority-N > CPL:
  - Update CPL = Priority-N
  - Push Registers to stack
  - Put N<sup>th</sup> entry in the **vector table** into PC
6. Now the ISR is running!

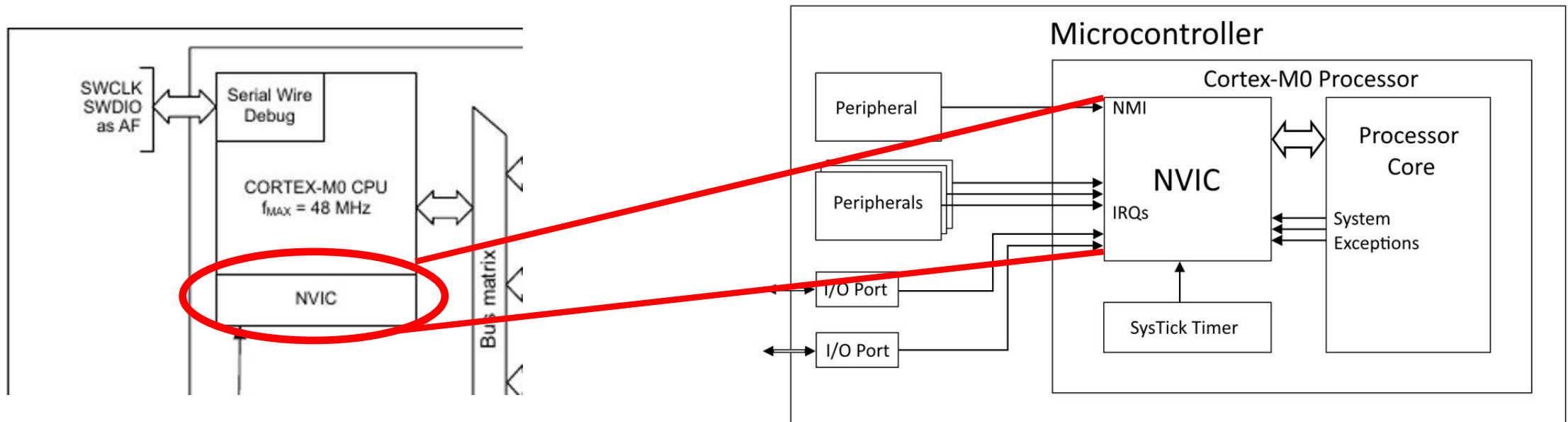
# Where stuff lives



- Most of the tables live inside the NVIC
- Enabled, Pending and Priority can be accessed via memory locations (like control registers) – but NVIC hardware has direct access.
- Only the vector table lookup and stack saving require memory accesses.

# Nested Vectored Interrupt Controller (NVIC)

- Unit on the STM32
  - Manages set of pre-defined exceptions
  - Supports priority levels, preemption, pending exceptions, masks, vectors, acknowledgement, etc.



# STM32 Exception Types

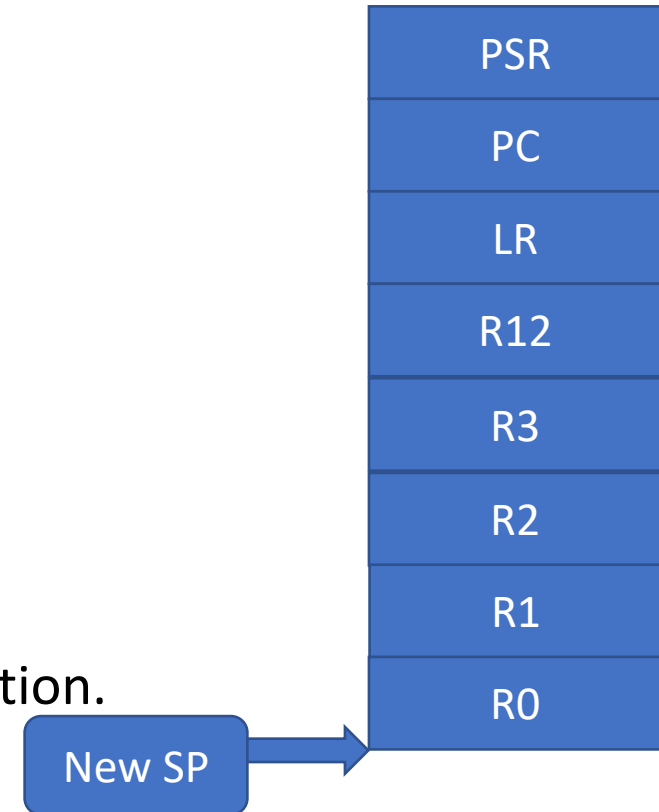
Table 37, page 217 of FRM

Position	Priority	Settable Priority?	Abbreviation	Description	Address
-	-	N	-	Reserved	0x0000 0000
-	-3	N	Reset	Reset	0x0000 0004
-	-2	N	NMI	Nom maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector	0x0000 0008
-	-1	N	HardFault	All class of fault	0x0000 000C
-	3	Y	SVCall	System service call via SWI instruction	0x0000 002C
-	5	Y	PendSV	Pendable request for system service	0x0000 0038
-	6	Y	SysTick	System tick timer	0x0000 003C
0	7	Y	WWDG	Window watchdog interrupt	0x0000 0040
1	8	Y	PVD_VDDIO2	PVD and V <sub>DDIO2</sub> supply comparator interrupt (+ EXTI lines 16 and 31)	0x0000 0044
2	9	Y	RTC	RTC interrupts (+ EXTI lines 17, 19, and 20)	0x0000 0048
3	10	Y	FLASH	Flash global interrupt	0x0000 004C
4	11	Y	RCC_CR5	RCC and CRS global interrupts	0x0000 0050
5	12	Y	EXTIO_1	EXTI Line[1:0] interrupts	0x0000 0054
6	13	Y	EXTI2_3	EXTI Line[3:2] interrupts	0x0000 0058
7	14	Y	EXTI4_15	EXTI Line[15:4] interrupts	0x0000 005C
8	15	Y	TSC	Touch sensing interrupt	0x0000 0060
9	16	Y	DMA_CH1	DMA channel 1 interrupt	0x0000 0064
10	17	Y	DMA_CH2_3 DMA2_CH1_2	DMA channel 2 and 3 interrupts DMA2 channel 1 and 2 interrupts	0x0000 0068
11	18	Y	DMA_CH4_5_6_7 DMA2_CH3_4_5	DMA channel 4, 5, 6, and 7 interrupts DMA2 channel 3, 4, and 5 interrupts	0x0000 006C

Position	Priority	Settable Priority?	Abbreviation	Description	Address
12	19	Y	ADC_COMP	ADC and COMP interrupts (+EXTI lines 21/22)	0x0000 0070
13	20	Y	TIM1_BRK_UP_TR G_COM	TIM1 break, update, trigger and commutation interrupt	0x0000 0074
14	21	Y	TIM1_CC	TIM1 capture compare interrupt	0x0000 0078
15	22	Y	TIM2	TIM2 global interrupt	0x0000 007C
16	23	Y	TIM3	TIM3 global interrupt	0x0000 0080
17	24	Y	TIM6_DAC	TIM6 global interrupt and DAC underrun interrupt	0x0000 0084
18	25	Y	TIM7	TIM7 global interrupt	0x0000 0088
19	26	Y	TIM14	TIM14 global interrupt	0x0000 008C
20	27	Y	TIM15	TIM15 global interrupt	0x0000 0090
21	28	Y	TIM16	TIM16 global interrupt	0x0000 0094
22	29	Y	TIM17	TIM17 global interrupt	0x0000 0098
23	30	Y	ISC1	I <sup>2</sup> C1 global interrupt (+ EXTI line 23)	0x0000 009C
24	31	Y	ISC2	I <sup>2</sup> C2 global interrupt	0x0000 00A0
25	32	Y	SPI1	SPI1 global interrupt	0x0000 00A4
26	33	Y	SPI2	SPI2 global interrupt	0x0000 00A8
27	34	Y	USART1	USART1 global interrupt (+ EXTI line 25)	0x0000 00AC
28	35	Y	USART2	USART2 global interrupt (+ EXTI line 26)	0x0000 00B0
29	36	Y	USART3_4_5_6_7_8	USART3/4/5/6/7/8 global interrupts (+ EXTI line 28)	0x0000 00B4
30	37	Y	CEC_CAN	CEC and CAN global interrupts (+ EXTI line 27)	0x0000 00B8
31	38	Y	USB	USB global interrupt (+ EXTI line 18)	0x0000 00BC

# On our micro, what happens on an exception

- NVIC decides things like:
  - Is exception enabled or should it be ignored?
  - Is the interrupt above current priority level?
  - Is the interrupt already pending?
- When an exception handler is invoked:
  - Current program stops running.
  - **R0-R3, R12**, LR, PC, PSR pushed onto stack.
    - These are the same registers ABI allows you to modify.
    - LR holds a special value that is not the PC of the program.
  - Priority level is set to that of the current invoked exception.
  - ISR address looked up in the vector table.
    - Branch to that. (i.e. set the PC to the address found there)



# Returning from an Interrupt

- Most CPUs have a special instruction (RTI) for this.
  - Needed because returning from an interrupt has to automatically restore the registers it saved when it was invoked
- However, Cortex-M0 does not have an RTI. So how does it know?
  - Special value placed in LR on invocation of the ISR.
  - When the special value is moved into the PC, it triggers a register reload.
  - The result: The ISR looks like any other subroutine (i.e. “***bx lr***” at the end).

# STM32 Vector Table

- Ranges from 0x0 – 0xC0 in memory.
  - This range is redundantly mapped into 0x0800 0000 (Flash ROM).
  - You can see it there in the debugger.
  - All the entries in the vector table are odd numbers???
    - The LSB set tells the CPU to execute code in "Thumb mode".
- Reset Vector (at 0x4) is special.
  - It doesn't push registers to the stack.
  - Stack pointer is initialized with contents at address 0x0.

# Updating Vector Table Entries

- Inside startup\_stm32.s

```
.section .isr_vector,"a",%progbits
.type g_pfnVectors, %object
.size g_pfnVectors, .-g_pfnVectors

g_pfnVectors:
    .word _estack
    .word Reset_Handler
    .word NMI_Handler
    .word HardFault_Handler
    .word 0
    ...
    .word SysTick_Handler
    ...
```

.weak is like .global except it is wimpy.  
It will be used if there are no other  
.global symbols of the same name.

```
/******
 * Provide weak aliases for each Exception
 * handler to the Default_Handler. As they
 * are weak aliases, any function with the
 * same name will override this definition.
 *****/

.weak HardFault_Handler
.thumb_set HardFault_Handler, Default_Handler

.weak SysTick_Handler
.thumb_set SysTick_Handler, Default_Handler
...
```

.thumb\_set is hard to explain.  
Takes the right-hand-side + 1.



# A weird aside: Thumb mode...

- Most ARM CPUs have two different instruction sets: ARM and Thumb.
  - You can switch between them with the BX (Branch and Exchange) instruction.
    - BX sets the 'T' bit in the ePSR (execution PSR) based on LSB of the register specified.
  - If the contents of the register specified for BX are odd, it forces the CPU to change to Thumb mode.
  - If the contents of the register specified for BX are even, it forces the CPU to change to ARM mode.
  - The ARM Cortex-M0 only supports Thumb mode. That's why the LR value from a BL instruction is always odd.
  - If you use an even address with BX, it will invoke the HardFault handler.
    - To maintain compatibility with all those other ARM CPUs you want to use.

# Bro, do you even BX?

```
1  .global main
2  main:
3      ldr    r0,=other
4      bx    r0
5
6
7  other:
8      ldr    r0,=main
9      bx    r0
```



HardFault!

# The “right way” to BX

```
1  .global main
2  main:
3      ldr    r0,=other
4      adds  r0,#1
5      bx    r0
6
7  other:
8      ldr    r0,=main
9      adds  r0,#1
10     bx    r0
11
```

Works, because the  
address is odd...

# To make sure addresses are odd

- The `.thumb_set` directive is used in the vector table:

`.thumb_set SysTick_Handler, Default_Handler`

- This creates a `SysTick_Handler` symbol (label) that is the `Default_Handler` value with the LSB set to 1.

- OR a label for a function could be declared as a function:

`.type SysTick_Handler, %function`

`.global SysTick_Handler`

`SysTick_Handler:`

- This makes all references to `SysTick_Handler` have the LSB set.
  - But the instructions that follow the label still start on an even address.
- A C compiler does this automatically for any function.

# Now, lets create an exception handler

- We'll work with easy ones first: **SVC\_Handler** and **SysTick\_Handler**.
  - These exceptions cannot be disabled. No work needed to enable them.
  - Fundamental hardware, so we don't need to enable RCC clocks.
  - We don't need to acknowledge the mechanisms that raise the interrupts.
- ISR should end with "BX LR" or a "POP" whose registers include PC.
  - If you call another subroutine in the ISR, that means the ISR is not a "leaf subroutine". You should definitely PUSH {LR} and return to user code with POP {PC}.

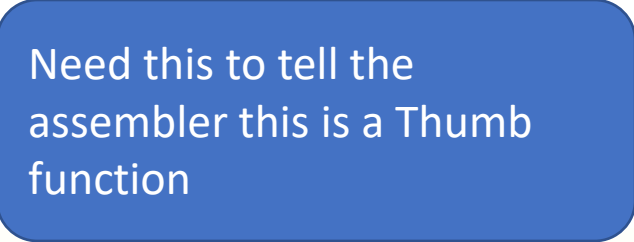
# SVC instruction

- SVC is the "Supervisor call" instruction.
  - Its intended to be like a "system call" instruction found on other systems, if you're familiar with that.
  - It must be assembled with an #imm8 argument, which becomes the low byte of the 2-byte instruction. (The only way to find this value is to read the instruction from memory.)
- When SVC is executed, it immediately raises an SVC exception, and invokes the handler for it.

# SVC Example

```
1  .global main
2  main:
3      movs r0,#0
4      movs r1,#1
5      movs r2,#2
6      movs r3,#3
7      svc #0xb1
8
9      nop
10     bkpt
```

```
1  .global SVC_Handler
2  .type SVC_Handler, %function
3  SVC_Handler:
4      movs r0,#6
5      movs r1,#6
6      movs r2,#6
7      movs r3,#6
8      ldr r0, [sp,#0]    // original R0 was pushed on stack
9      ldr r1, [sp,#4]    // original R1 was pushed on stack
10     ldr r2, [sp,#8]    // original R2 was pushed on stack
11     ldr r3, [sp,#12]   // original R3 was pushed on stack
12     ldr r0, [sp,#16]   // original R12 was pushed on stack
13     ldr r1, [sp,#20]   // original LR was pushed on stack
14     ldr r2, [sp,#24]   // original PC was pushed on stack
15     ldr r3, [sp,#28]   // original xPSR was pushed on stack
16
17     // R2 now points to the next instruction after the SVC call
18
19     subs r2,#2         // make R2 point to the SVC call
20     ldrb r0,[r2]       // load the SVC number
21     ldrb r1,[r2,#1]    // load the opcode (should be 0xdf)
22
23     bx lr             // return
```



# SysTick

- The SysTick exception is generated periodically by a 24-bit down-counting timer.
  - Set the reset value.
  - Set the clock source (either CPU clock: 48MHz or  $\div 8$ : 6MHz)
  - Enable the counter.
  - When the counter reaches zero, it flags an exception, and loads the reset value.
  - The interrupt cannot be ignored, so it does not need to be enabled.



# SysTick Control Registers

- See page 86 of the Cortex-M0 programming manual.
- STK: base address: 0xe000e010
  - CSR: (offset 0x0) Control/Status Register
  - RVR: (offset 0x4) Reset Value Register (24 bits)
  - CVR: (offset 0x8) Current Value Register

# STK CSR layout

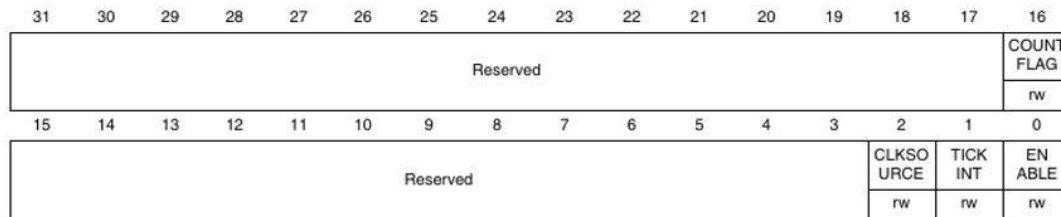
## 4.4.1 SysTick control and status register (STK\_CSR)

Address offset: 0x00

Reset value: 0x0000 0004

The SysTick CSR register enables the SysTick features.

- Set the low 3 bits to '1' to enable using the CPU clock.



Bits 31:17 Reserved, must be kept cleared.

**Bit 16 COUNTFLAG:**

Returns 1 if timer counted to 0 since last time this was read.

Bits 15:3 Reserved, must be kept cleared.

**Bit 2 CLKSOURCE:** Clock source selection

Selects the timer clock source.

0: External reference clock

1: Processor clock

**Bit 1 TICKINT:** SysTick exception request enable

0: Counting down to zero does not assert the SysTick exception request

1: Counting down to zero asserts the SysTick exception request.

**Bit 0 ENABLE:** Counter enable

Enables the counter. When ENABLE is set to 1, the counter starts counting down. On reaching 0, it sets the COUNTFLAG to 1 and optionally asserts the SysTick depending on the value of TICKINT. It then loads the RELOAD value again, and begins counting.

0: Counter disabled

1: Counter enabled

# Systick

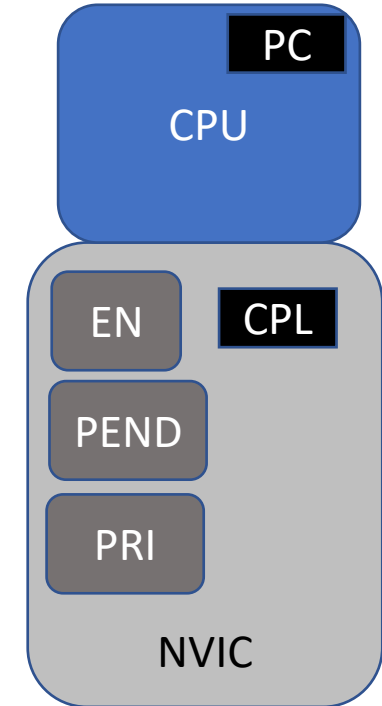
```
1  .equ STK, 0xe000e010
2  .equ CSR, 0x0
3  .equ RVR, 0x4
4      // Set up the SysTick timer...
5      ldr r3, =STK
6      ldr r0, =12000000-1 // 12M ticks = 1/4 second
7      str r0, [r3, #RVR] // Set the reset value
8
9      movs r0, #7
10     str r0, [r3, #CSR] // Enable using CPU clock
11 endless:
12     b endless|
```

- Simple example that does nothing on in the ISR

```
1  // The ISR...
2  .type SysTick_Handler, %function
3  .global SysTick_Handler
4  SysTick_Handler:
5      push {lr}
6      nop
7      pop {pc}|
```

# NVIC Control Registers

- 0xe000e100: ISER: Interrupt set-enable register
- 0xe000e180: ICER: Interrupt clear-enable register
- 0xe000e200: ISPR: Interrupt set-pending register
- 0xe000e280: ICPR: Interrupt clear-pending register
- 0xe000e400 – 0xe000e41c: (32 bytes)
  - IPR0-IPR7: Interrupt priority registers
- Look at Section 4.2 of the Cortex-M0 programming manual



# ISER and ICER

- Writing a '1' to any bit in ISER enables that interrupt, '0' does nothing.
- Writing a '1' to any bit in ICER disables that interrupt, '0' does nothing.
- Operationally similar to GPIOx BRR and BSRR
  - So you don't have to load, ORRS/BICS, store.
- An ISR will only be invoked if an interrupt is enabled.
  - You cannot disable exceptions -3 (HardFault), -2 (NMI), -1 (Reset).
    - That's why they have negative numbers.

# ISPR and ICPR

- Set or clear the pending bit for an interrupt.
- When any exception is raised, the pending bit for it is set.
  - If the exception is of lower priority than the current priority level, nothing happens until the currently executing exception handler (ISR) finishes.
  - You may also manually schedule an interrupt by setting its pending bit.
  - You can discard a pending interrupt by clearing its pending bit.
  - These also work like GPIOx BRR/BSRR.

# Priority Table

- 32 Interrupts with adjustable priority, 1 byte per interrupt = 32 bytes of priority registers
- Only the two most significant bits of the bytes are used.
  - Possible values: 0, 64, 128, 192
  - 0 is the "highest" (most important) priority
  - 192 is the "lowest" (least important) priority
  - Default priority for all exceptions is 0
- Must set all 32 bits of each priority word (4 entries) at once.
  - Load, shift, ORRS/BICS, store.
  - This is difficult, so we won't deal with interrupt priorities using assembly language.
- Priority should be set before interrupt is enabled.

# What About These Priorities?

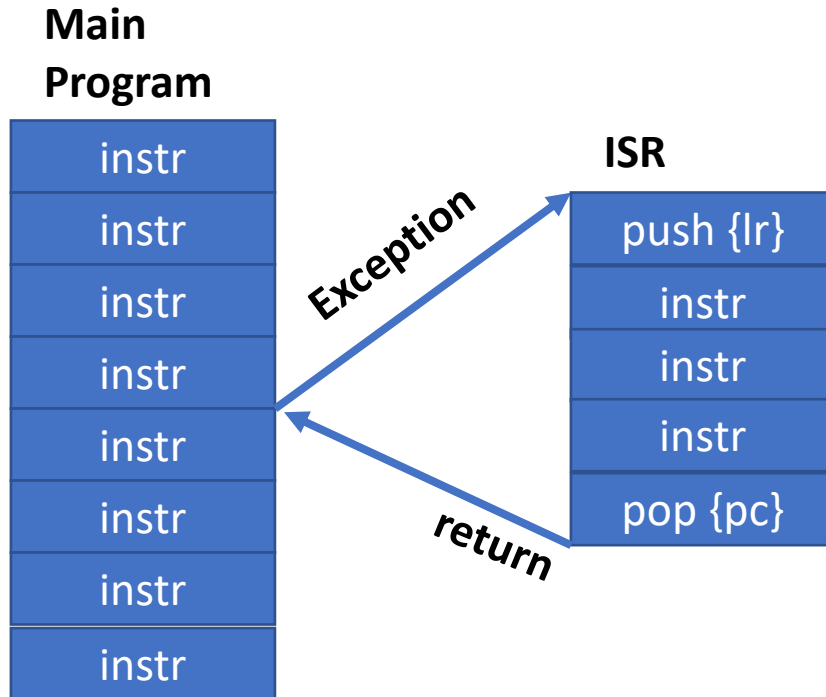
- If two interrupts with the same priority are raised, the one with the lowest number in this table will be invoked first.

Position	Priority	Settable Priority?	Abbreviation	Description	Address
-	-	N	-	Reserved	0x0000 0000
-	-3	N	Reset	Reset	0x0000 0004
-	-2	N	NMI	Nom maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector	0x0000 0008
-	-1	N	HardFault	All class of fault	0x0000 000C
-	3	Y	SVCall	System service call via SWI instruction	0x0000 002C
-	5	Y	PendSV	Pendable request for system service	0x0000 0038
-	6	Y	SysTick	System tick timer	0x0000 003C
0	7	Y	WWDG	Window watchdog interrupt	0x0000 0040
1	8	Y	PVD_VDDIO2	PVD and V <sub>DDIO2</sub> supply comparator interrupt (+ EXTI lines 16 and 31)	0x0000 0044
2	9	Y	RTC	RTC interrupts (+ EXTI lines 17, 19, and 20)	0x0000 0048
3	10	Y	FLASH	Flash global interrupt	0x0000 004C
4	11	Y	RCC_CR5	RCC and CRS global interrupts	0x0000 0050
5	12	Y	EXTIO_1	EXTI Line[1:0] interrupts	0x0000 0054
6	13	Y	EXTI2_3	EXTI Line[3:2] interrupts	0x0000 0058
7	14	Y	EXTI4_15	EXTI Line[15:4] interrupts	0x0000 005C
8	15	Y	TSC	Touch sensing interrupt	0x0000 0060
9	16	Y	DMA_CH1	DMA channel 1 interrupt	0x0000 0064
10	17	Y	DMA_CH2_3 DMA2_CH1_2	DMA channel 2 and 3 interrupts DMA2 channel 1 and 2 interrupts	0x0000 0068
11	18	Y	DMA_CH4_5_6_7 DMA2_CH3_4_5	DMA channel 4, 5, 6, and 7 interrupts DMA2 channel 3, 4, and 5 interrupts	0x0000 006C

Position	Priority	Settable Priority?	Abbreviation	Description	Address
12	19	Y	ADC_COMP	ADC and COMP interrupts (+EXTI lines 21/22)	0x0000 0070
13	20	Y	TIM1_BRK_UP_TR G_COM	TIM1 break, update, trigger and commutation interrupt	0x0000 0074
14	21	Y	TIM1_CC	TIM1 capture compare interrupt	0x0000 0078
15	22	Y	TIM2	TIM2 global interrupt	0x0000 007C
16	23	Y	TIM3	TIM3 global interrupt	0x0000 0080
17	24	Y	TIM6_DAC	TIM6 global interrupt and DAC underrun interrupt	0x0000 0084
18	25	Y	TIM7	TIM7 global interrupt	0x0000 0088
19	26	Y	TIM14	TIM14 global interrupt	0x0000 008C
20	27	Y	TIM15	TIM15 global interrupt	0x0000 0090
21	28	Y	TIM16	TIM16 global interrupt	0x0000 0094
22	29	Y	TIM17	TIM17 global interrupt	0x0000 0098
23	30	Y	ISC1	I <sup>2</sup> C1 global interrupt (+ EXTI line 23)	0x0000 009C
24	31	Y	ISC2	I <sup>2</sup> C2 global interrupt	0x0000 00A0
25	32	Y	SPI1	SPI1 global interrupt	0x0000 00A4
26	33	Y	SPI2	SPI2 global interrupt	0x0000 00A8
27	34	Y	USART1	USART1 global interrupt (+ EXTI line 25)	0x0000 00AC
28	35	Y	USART2	USART2 global interrupt (+ EXTI line 26)	0x0000 00B0
29	36	Y	USART3_4_5_6_7_8	USART3/4/5/6/7/8 global interrupts (+ EXTI line 28)	0x0000 00B4
30	37	Y	CEC_CAN	CEC and CAN global interrupts (+ EXTI line 27)	0x0000 00B8
31	38	Y	USB	USB global interrupt (+ EXTI line 18)	0x0000 00BC

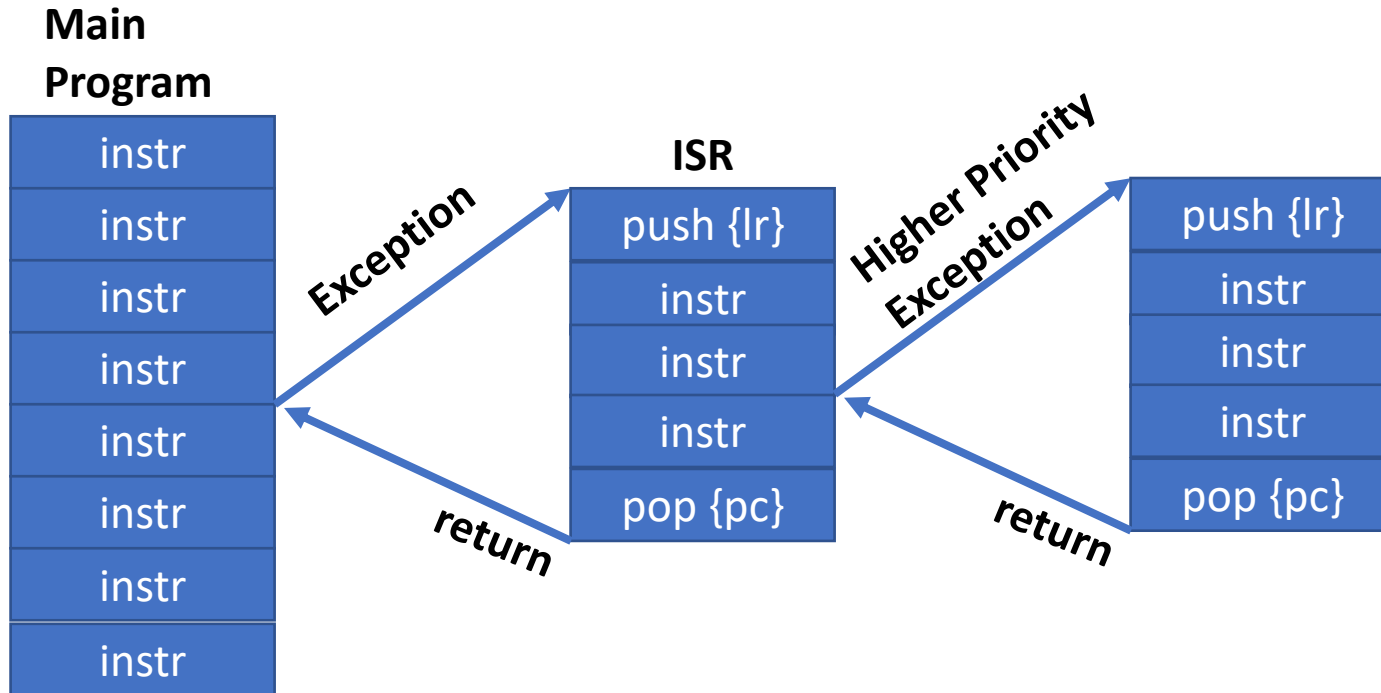


# Exception flow



- Interrupts are usually handled at the end of instructions.
  - Execution resumes at the beginning of the next instruction.
- Faults happen in the middle of an instruction.
  - Execution resumes at the same instr.
- ISR is expected to be quick.
- Can an Exception Handler be interrupted?

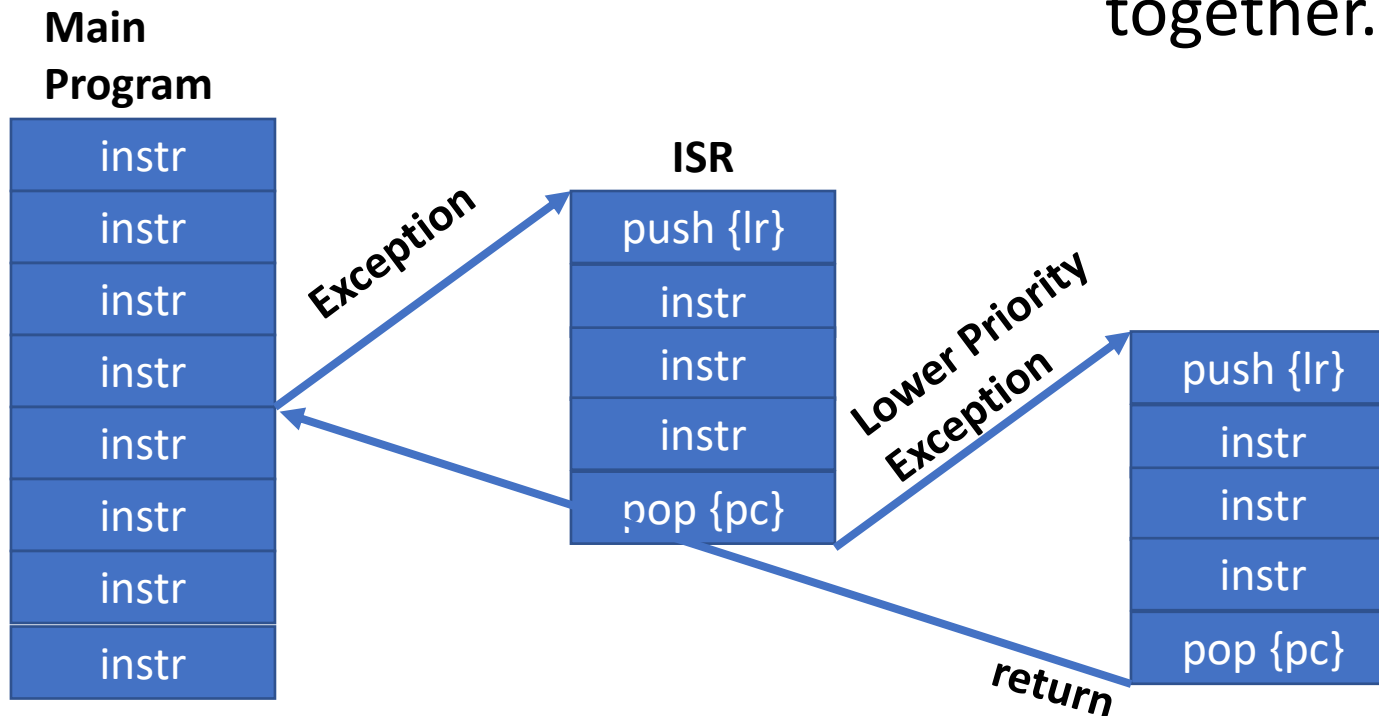
# Nested Exceptions



- Our priority system says which exception handlers can be **preempted**.

# Pending Exceptions

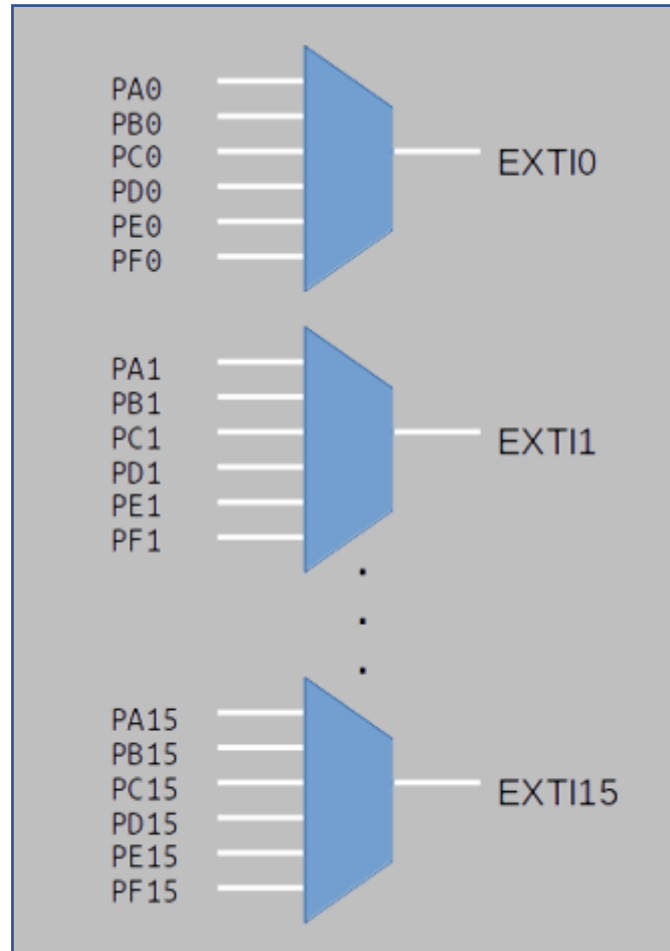
- Lower priority exceptions wait until earlier one finishes. **Chaining** them together.



# Peripherals generate exceptions

- But we don't know how to use those yet...
- GPIO pins can also be configured to generate exceptions.
- Some restrictions...
  - If PA0 is configured to generate an interrupt, PB0, PC0, PD0, etc cannot. Only one port per pin # can generate an interrupt.
  - Some of the pins are lumped together in the same interrupt number.
  - Pins 0,1 ==> Int 5; Pins 2,3 ==> Int 6; Pins 4–15 ==> Int 7
  - This is the reason we wired SW2 to PA0 and SW3 to PB2.

# Interrupts and Port/Pins



- For each pin # there is a selection for the port that a pin # can generate an interrupt for.
  - You might think it would be the other way around.
- SYSCFG EXTICR1 ... EXTICR4 select the ports.
  - Family Reference Manual p177.
- **Default for each pin is Port A.**

# Selecting a GPIO event type

- Pins can generate an interrupt upon a rising edge, a falling edge, or both.
  - See EXTI\_RTSTR and EXTI\_FTSR (rising/falling trigger select registers), p224 FRM.
  - 32-bit registers (lower 16 for pins, upper 16 for internal peripherals)
  - Writing a '1' to the appropriate bit position enables that pin # to trigger on the edge in question.

# (Un)masking External Interrupts

- The EXTI\_IMR (Interrupt Mask Register) configures which interrupt types are ignored.
  - FRM, page 223.
  - 32-bit register again.
  - Writing a '1' makes pin # not ignored.
  - Only internal peripherals (upper 16 bits) are enabled at reset time.

# Enabling an interrupt

- SVC and SysTick are always enabled.
- Other exceptions must be explicitly enabled.
- Turn on the interrupt number in NVIC\_ISER
- Section 4.2 of the Cortex-M0 programming manual

```
1  ldr  r0, =NVIC
2  ldr  r1, =NVIC_ISER
3  ldr  r2, =1<<30
4  str  r2, [r0, r1] // enable int 30
```

- Once configured and enabled, the ISR will be invoked when the event occurs.



# Summary: GPIO Interrupts

- To enable an interrupt for the rising edge of PA0:
  - Turn on GPIOAEN in RCC\_AHBENR
    - Set GPIOA\_PUPDR to pull down PA0.
  - The EXTI system is part of the SYSCFG subsystem.
    - We must turn on the clock to SYSCFG before it will do anything.
    - First set the SYSCFGCOMPEN bit in RCC\_APB2ENR
      - What?? (SYSCFG is grouped with a voltage comparator system)
  - Set bits 3:0 of SYSCFG\_EXTICR1 to zero.
  - OR a 0x1 into bit position 0 (for PA0) of EXTI\_RTSR.
  - OR a 0x1 into bit position 0 (for PA0) EXTI\_IMR.
  - Enable the interrupt by writing a 0x1 into bit 5 (interrupt #5 is for pins 0 and 1) of the NVIC\_ISER.

# Acknowledging an Interrupt

- Doing all those steps will invoke the interrupt when SW2 is pressed, but the moment the ISR exits, it will be immediately re-invoked.
  - Why?
  - Invoking the ISR does not acknowledge the interrupt.
  - We must write a '1' to bit 0 (for PA0) of the EXTI\_PR (pending register) to acknowledge the interrupt.
    - Then the NVIC knows we took care of the event.
    - Execution will go back to the main program.

# ISRs and non-atomicity

- Remember BRR/BSRR: They were **atomic**.
- Here is why that matters

End result: Pin 1 will get unintentionally turned off by the main program

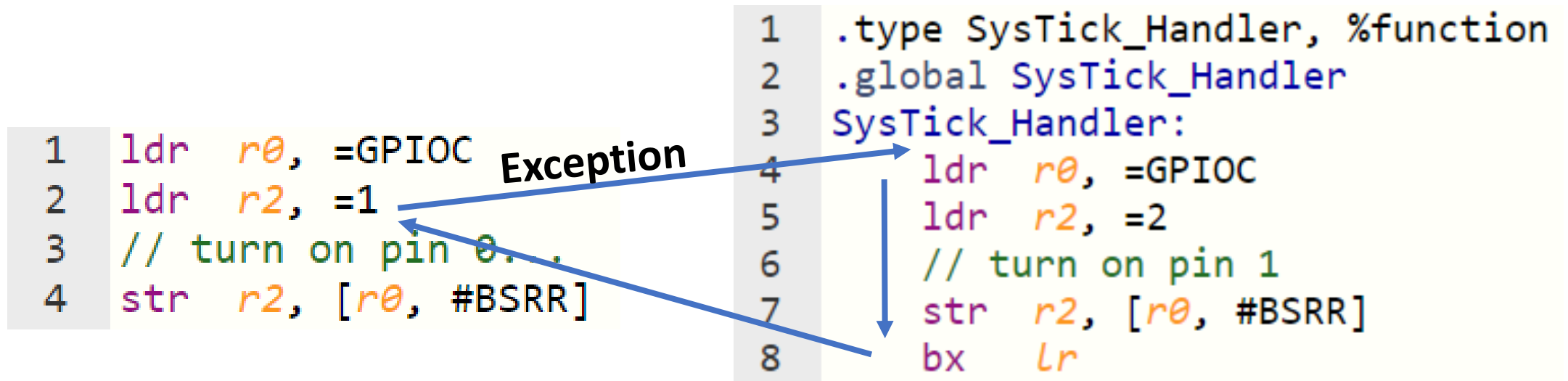
```
1  ldr  r0, =GPIOC
2  ldr  r1, [r0, #ODR]
3  ldr  r2, =1
4  orrs r1, r2 // turn on pin 0
5  str  r1, [r0, #ODR]
```

Exception

```
1  .type SysTick_Handler, %function
2  .global SysTick_Handler
3  SysTick_Handler:
4  ldr  r0, =GPIOC
5  ldr  r1, [r0, #ODR]
6  ldr  r2, =2
7  orrs r1, r2 // turn on pin 1
8  str  r1, [r0, #ODR]
9  bx  lr
```

# Using Atomics

- No temporary value stored in the main program – setting done in one instruction.
- No point at which the interrupt can mess this up.



# Other peripherals have no BSRR

- The GPIO ports are unique in that they have BSRR/BRR control registers. Other peripherals do not have such things.
- Another way to avoid corruption by an ISR that may run during a read-modify-write is to disable all interrupts.
  - The PRIMASK CPU register does this.
  - cpsid i: Sets PRIMASK to 0 to disallow all interrupts except NMI
  - cpsie i: Sets PRIMASK to 1 to allow interrupts to occur again

# Masking all interrupts

- Brute force: turn off all interrupts when read-modify-writing a modifying control register. **This is a bad general solution.**
- Better solution: Make sure each peripheral is only updated by just one ISR or the main program.

```
1  ldr  r0, =GPIOC
2  cpsid i // Disallow interrupts
3  ldr  r1, [r0, #ODR]
4  ldr  r2, =1
5  orrs r1, r2 // turn on pin 0
6  str  r1, [r0, #ODR]
7  cpsie i // Reallow interrupts
```

```
1  .type SysTick_Handler, %function
2  .global SysTick_Handler
3  SysTick_Handler:
4      ldr  r0, =GPIOC
5      ldr  r1, [r0, #ODR]
6      ldr  r2, =2
7      orrs r1, r2 // turn on pin 1
8      str  r1, [r0, #ODR]
9      bx  Lr
```

# Interrupts done

- So useful.
- So complicated.
- You'll get lots of practice using interrupts, and you will reach a point where you will think,
- "I have always thought these were simple."