Interrupts and Exceptions

ECE 362
https://engineering.purdue.edu/ece362/

Reading Assignment

- Reading assignment:
  - STM32F0x1 Family Reference, Chapter 12, pages 217 – 228, "Interrupts and events"
  - Textbook, Chapter 11, "Interrupts"
    - 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.

Remember this code from the last lecture?

```
.equ GPIOA, 0x48000000   // Base address of GPIOA control registers
.equ GPIOC, 0x48000800   // Base address of GPIOC control registers
.equ IDR, 0x10           // Offset of the IDR
.equ ODR, 0x14           // Offset of the ODR

    ldr  r0, =GPIOA      // Load, into R0, base address of GPIOA
    ldr  r1, =GPIOC      // Load, into R1, base address of GPIOC
    movs r3, #1
again:
    ldr  r2, [r0, #IDR]  // Load, into R2, value of GPIOA_IDR
    ands r2, r3          // Look at only bit zero.
    lsls r2, #8          // Translate bit 0 to bit 8
    str  r2, [r1, #ODR]  // Store value to GPIOC_ODR
    b    again
```
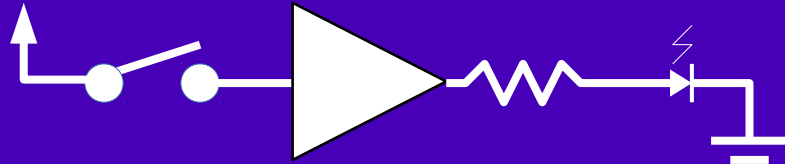
# A loop that reads PA0 and writes PC8, forever?

- A waste of the CPU.
- It would be excellent if the CPU could do something more useful and get <u>notified</u> if the button is pressed.
  - Such a thing is called an <u>interrupt</u>.
  - An interrupt is a type of <u>exception</u>.
    - There are other kinds of exceptions: faults, traps, and reset.
  - An exception is a hardware-invoked subroutine.
  - That subroutine is called an <u>Interrupt Service Routine</u> (ISR) or, sometimes, a <u>handler</u>.

Interrupt Fundamentals

- Peripherals can be configured to *raise* interrupts under certain circumstances.

- Most CPUs support many kinds of interrupts and a <u>*vector table*</u> in memory is used to look up the address of a *handler* (ISR) to invoke when a particular interrupt occurs.

- Most types of interrupts can be individually *enabled* (allowed to occur), or *disabled* (prevented from being invoked). A table in memory keeps track of which are enabled.

- Different interrupts can have different *priorities*. Higher priorities interrupts can *preempt* those of lower priorities. Priorities are encoded into yet another table in memory.

- Interrupts that have been *raised*, but not yet been handled, can be marked *pending*. A pending table (in memory) is used to keep track.

- Once an interrupt handler is invoked, it must tell the peripheral that raised the request that it is *acknowledging* its request. Otherwise, the peripheral may continue raising the interrupt.

Interrupt Action Description

- A peripheral *raises* an interrupt with a particular number.
- The computer checks if the particular interrupt is *enabled*.
- If so,
  - the computer marks the interrupt as *pending*, and
  - checks if the *priority* of the new interrupt is higher than the *current priority level*.
- If so,
  - the current priority level is updated to the priority of the invoked interrupt.
  - the registers are pushed into the stack.
  - the nth entry of the *vector table* is fetched and put into the PC.
- ...and the interrupt service routine is now running.

PC

int #n

periph

CPU

cpl++

regs

?

enabled

stack

pending

?

priority

vector table
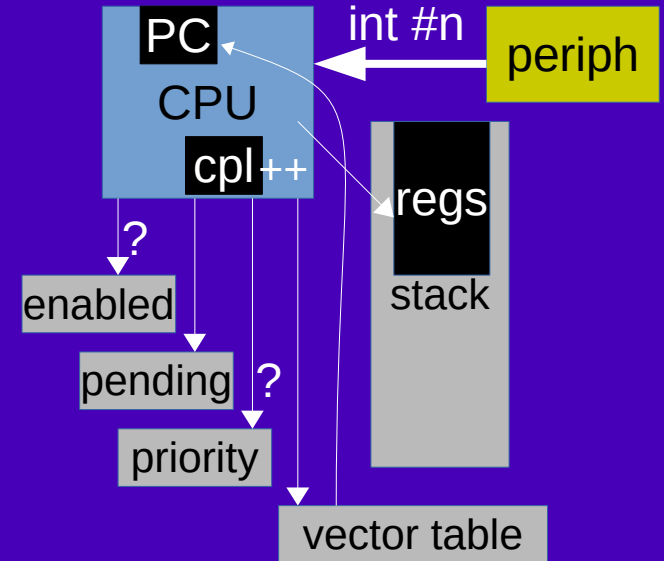
Most tables are built in to the CPU

- In reality, most of those table lookups are done inside something called the NVIC, which is right next to the CPU.

- The enable, pending, and priority tables are accessible and modifiable via memory locations, but the NVIC has direct access to them.

- The only memory references needed to invoke an interrupt service routine are the vector table lookup and saving the registers to the stack.

PC

int #n

periph

CPU

enabled   cpl

regs

pending

NVIC

stack

priority

vector table
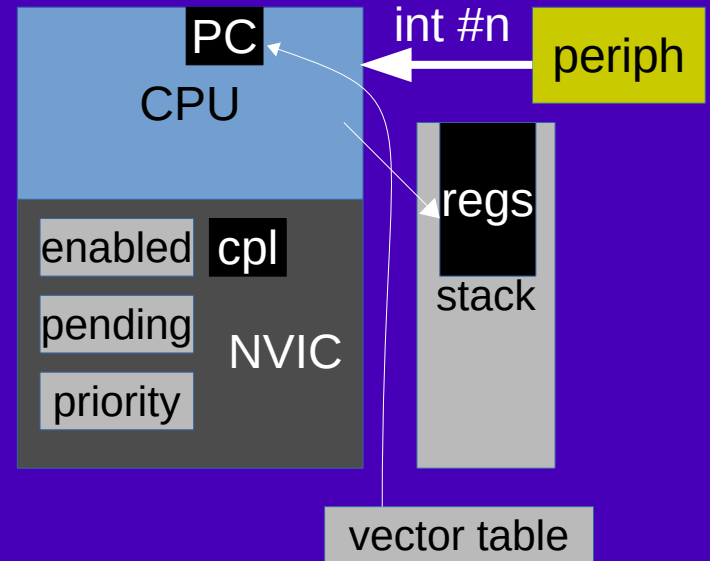
STM32 Exceptions: NVIC

- Nested Vectored Interrupt Controller
  - Manages several defined types of exceptions
  - Supports priority levels, preemption, pending exceptions, masks, vectors, acknowledgement, etc.

STM32F091xBC datasheet, pg 12
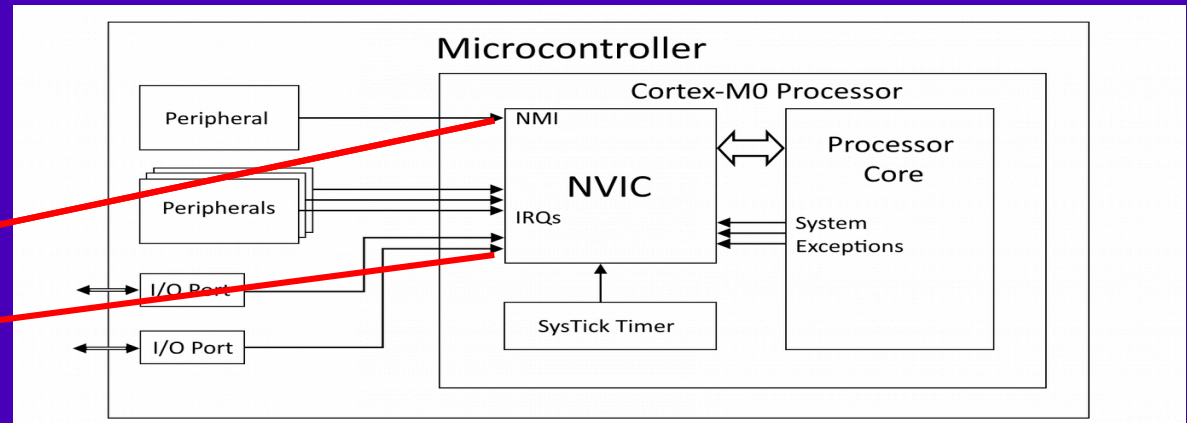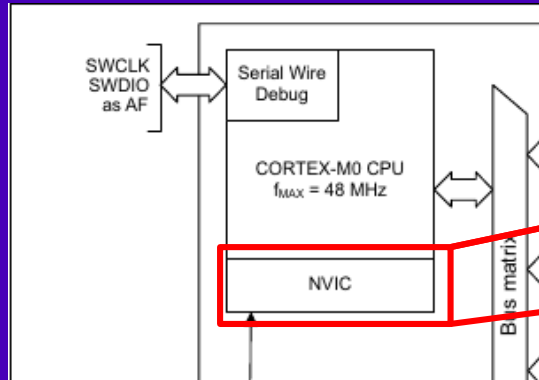
Types of exceptions
(Table 37, page 217 of FRM)

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_{DDIO2} 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_CRS	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 channgel 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_TRG_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^2C1 global interrupt (+ EXTI line 23)	0x0000 009C
24	31	Y	ISC2	I^2C2 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

What happens when an exception occurs?

- 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)

| PSR |
| PC |
| LR |
| R12 |
| R3 |
| R2 |
| R1 |
| R0 |

new
SP

Return From Interrupt

- Most types of CPUs have a "return from subroutine" (RTS) instruction as well as a "return from interrupt" (RTI) instruction.
  - Because the interrupt stack frame contains saved register values put there by the CPU.
  - RTI instruction knows how to restore the registers.
  - When you write your ISR, you must remember to do a RTI rather than an RTS.
- **ARM Cortex-M0 does not have these problems.**
  - 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.

The 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

- If you look at 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
    …
```

```
/****************************************
* 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
…
```

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

.thumb_set is hard to explain. Takes the right-hand-side + 1.

About 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.

Example With BX

- We should be able to load an address into R0 and jump to it, right?

```
.global main
main:
    ldr    r0,=other
    bx     r0

other:
    ldr    r0,=main
    bx     r0
```

HardFault

Working Example With BX

- An address for BX must be odd.

```
.global main
main:
    ldr   r0,=other
    adds  r0,#1
    bx    r0

other:
    ldr   r0,=main
    adds  r0,#1
    bx    r0
```

Address is odd. Works great!

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.

To create an exception handler

- Just create a subroutine with the proper name to replace the weak name.
  - If you are tired of getting stuck in the infinite loop when you make an unaligned memory reference, you can write your own version of HardFault_Handler that fixes the load or store and returns to the faulting program. (That's a lot of work though.)
  - 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" <u>or a "POP" whose registers include PC</u>.
    - 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 Program

```
// Why do we never need to declare "main"
// to be a Thumb function?
.global main
main:
        movs r0,#0
        movs r1,#1
        movs r2,#2
        movs r3,#3
        svc #0xb1

        nop
        bkpt
```

```
.global SVC_Handler
.type SVC_Handler, %function
SVC_Handler:
        movs r0,#6
        movs r1,#6
        movs r2,#6
        movs r3,#6
        ldr  r0, [sp,#0]   // original R0 was pushed on stack
        ldr  r1, [sp,#4]   // original R1 was pushed on stack
        ldr  r2, [sp,#8]   // original R2 was pushed on stack
        ldr  r3, [sp,#12] // original R3 was pushed on stack
        ldr  r0, [sp,#16] // original R12 was pushed on stack
        ldr  r1, [sp,#20] // original LR was pushed on stack
        ldr  r2, [sp,#24] // original PC was pushed on stack
        ldr  r3, [sp,#28] // original xPSR was pushed on stack

        // R2 now points to the next instruction after the SVC call

        subs r2,#2              // make R2 point to the SVC call
        ldrb r0,[r2]           // load the SVC number
        ldrb r1,[r2,#1]        // load the opcode (should be 0xdf)

        bx lr // return
```

Any reference to the label in a vector table will have the LSB set.

Without this, the address in the vector table would be even, and the CPU would consider it ARM code rather than Thumb code.

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 ÷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: 0xe000 e010
  - CSR: (offset 0x0) Control/Status Register
  - RVR: (offset 0x4) Reset Value Register (24 bits)
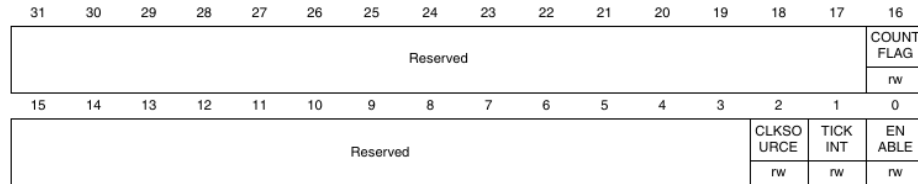 - CVR: (offset 0x8) Current Value Register

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.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
						Reserved									COUNT FLAG
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					Reserved								CLKSO URCE	TICK INT	EN ABLE
													rw	rw	rw

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 to 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

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

23

SysTick Program

```
// The ISR...
.type SysTick_Handler, %function
.global SysTick_Handler
SysTick_Handler:
    push {lr}
    nop
    pop {pc}
```

Any reference to the label in a vector table will have the LSB set.

Without this, the address in the vector table would be even, and the CPU would consider it ARM code rather than Thumb code.

```
.equ STK, 0xe000e010
.equ CSR, 0x0
.equ RVR, 0x4
    // Set up the SysTick timer...
    ldr r3, =STK
    ldr r0, =12000000-1    // 12M ticks = 1/4 second
    str r0, [r3, #RVR]     // Set the reset value

    movs r0, #7
    str r0, [r3, #CSR]     // Enable using CPU clock
endless:
    b endless
```

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 <u>bytes</u>)

    IPR0-IPR7: Interrupt priority registers

- Look at Section 4.2 of the Cortex-M0 programming manual.

25

ISER and ICER

- Writing a '1' to any bit in ISER <u>enables</u> that interrupt regardless of any others.

- Writing a '1' to any bit in ICER <u>disables</u> that interrupt regardless of any others.

- 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.

26

ISPR and ICPR

- Set or clear the <u>pending</u> 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 <u>all exceptions</u> 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 <u>before</u> interrupt is enabled.

What About These Priorities?

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_{DDIO2} 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_CRS	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 channgel 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

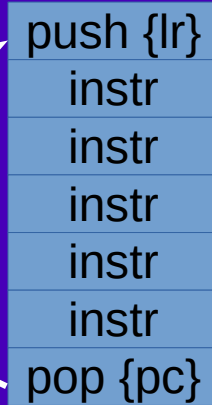| Position	Priority	Settable Priority?	Abbreviation	Description	Address
12	19	Y	ADC_COMP	ADC and COMP interrupts (+EXTI lines 21/22)	0x0000 0070
13	20	Y	TIM1_BRK_UP_TRG_COM	TIM1 break, update, trigger and commutation Interrupt	0x0000 0074
14	21	Y	TIM1_CC	TIM1 capture compare interrupt	0x0000 0078
15	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^2C1 global interrupt (+ EXTI line 23)	0x0000 009C
24	31	Y	ISC2	I^2C2 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

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

Execution Model of Exceptions

Main program

instr
instr
instr
instr
instr
instr
instr
instr
instr
instr
…

Exception Handler (ISR)

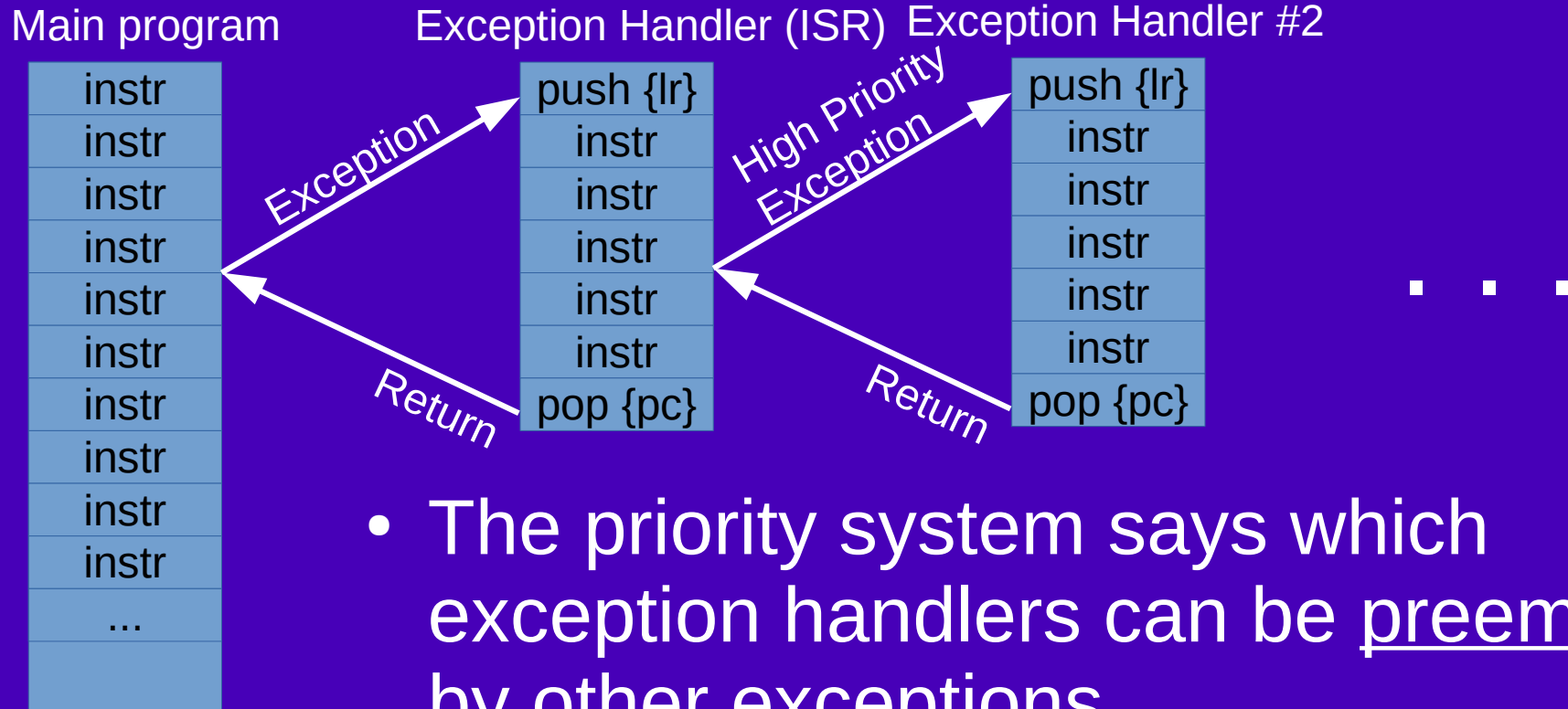| push {lr} |
|-----------|
| instr |
| instr |
| instr |
| instr |
| instr |
| pop {pc} |

Exception

Return

- Exceptions are usually handled at the end of instructions.
- Execution resumes at the beginning of the next instruction.
- Exception handler is expected to be quick.
- Can an Exception Handler be interrupted?

Nested Exceptions

Main program

Exception Handler (ISR)

Exception Handler #2

instr
instr
instr
instr
instr
instr
instr
instr
instr
instr
...

push {lr}
instr
instr
instr
instr
instr
pop {pc}

push {lr}
instr
instr
instr
instr
instr
pop {pc}

Exception

Return

High Priority Exception

Return

. . .

- The priority system says which exception handlers can be <u>preempted</u> by other exceptions.
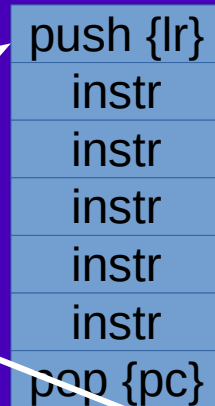
31

Pending Exceptions

- A lower priority exception will wait until the first one finishes, get <u>chained</u> in execution, and then return directly to the original progam.
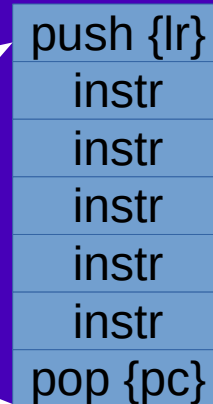
Main program

instr
instr
instr
instr
instr
instr
instr
instr
instr
...

Exception Handler (ISR)

push {lr}
instr
instr
instr
instr
instr
pop {pc}

Exception Handler #2

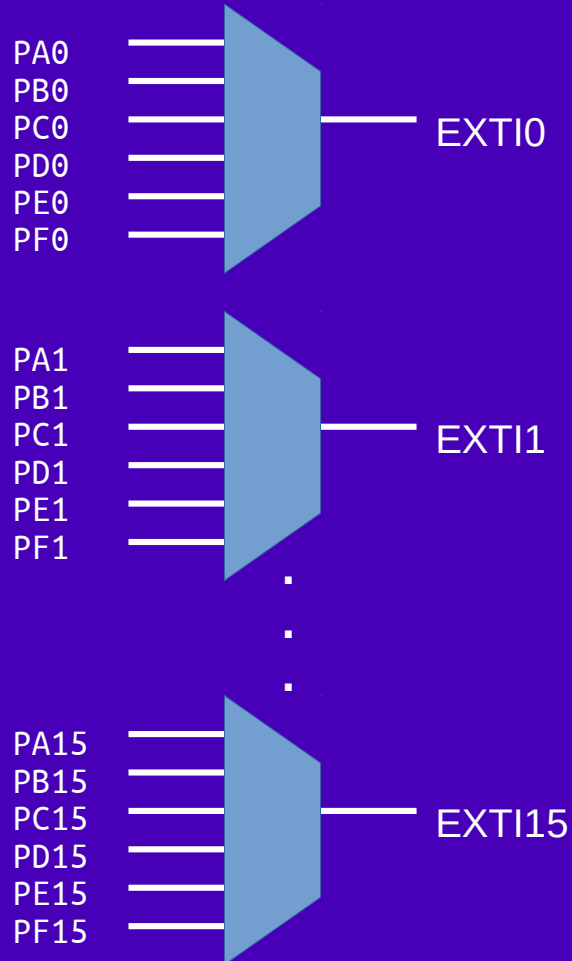| push {lr} |
|-----------|
| instr |
| instr |
| instr |
| instr |
| instr |
| pop {pc} |

Exception

Lower Priority Exception

Return

Most Built-In 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.

Figure 25 of FRM, p222

Pin/Port selection

PA0
PB0
PC0
PD0
PE0
PF0

EXTI0

PA1
PB1
PC1
PD1
PE1
PF1

EXTI1

PA15
PB15
PC15
PD15
PE15
PF15

EXTI15

- 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.
- <u>Default for each pin is Port A.</u>

34

Selecting a GPIO event type

- Pins can generate an interrupt upon a rising edge, a falling edge, or both.
  - See EXTI_RTSR 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

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

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

Recap: 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.
    - This is the default for PA0, so we don't have to do it.
  - 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 <u>acknowledge</u> 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.

Issues with ISRs and Non-Atomicity

- The lecture on GPIO claimed that BRR/BSRR are so wonderful because they are <u>ATOMIC</u>. What's the big deal?

If a program does this:

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

and it is interrupted and an ISR does this:

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

When the main program resumes, the value restored to R1 will not have bit 1 set.
It will turn pin 1 off.

40

Solution Using Atomic Operations

- We can read-modify-write ODR in one operation...

If a program does this:

```
ldr  r0, =GPIOC
ldr  r2, =1
// turn on pin 0...
str  r2, [r0, #BSRR]
```

and it is interrupted and an ISR does this:

```
.type SysTick_Handler, %function
.global SysTick_Handler
SysTick_Handler:
    ldr  r0, =GPIOC
    ldr  r2, =2
    // turn on pin 1
    str  r2, [r0, #BSRR]
    bx   lr
```

Since there are no longer multiple instructions to read-modify-write GPIOC ODR,
there is no point where the SysTick interrupt can occur that will corrupt the update.

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 <u>all</u> interrupts.
  - The PRIMASK CPU register does this.
  - cpsid i: Sets PRIMASK to 0 to disallow all interrupts except NMI
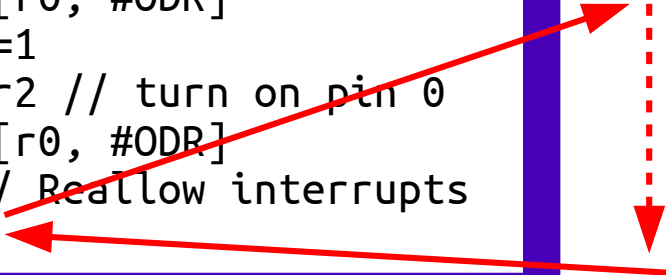 - cpsie i: Sets PRIMASK to 1 to allow interrupts to occur again

Masking All Interrupts

If a program does this:

The ISR can only run before cpsid or after cpsie:

```
ldr   r0, =GPIOC
cpsid i // Disallow interrupts
ldr   r1, [r0, #ODR]
ldr   r2, =1
orrs r1, r2 // turn on pin 0
str   r1, [r0, #ODR]
cpsie I // Reallow interrupts
```

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

- Concurrency always requires careful thought. This is a very brute-force method of dealing with the problem. If used as a general solution, it may lead to greater problems.

- A better solution is to ensure that a peripheral is only updated by a single ISR or by the main program.

43

Interrupts!

- 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."