Interrupts and Exceptions

ECE 362
https://engineering.purdue.edu/ee362/
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.
Why must I not modify R4 – R11?

- If you wrote everything in assembly language, you could do whatever you wanted.
- We do this just to get along with the C compiler.
- So what happens if you don’t?
Consider zero_to_n(int n)

```c
int zero_to_n(int n) {
    int x;
    int sum = 0;
    for(x=0; x<=n; x++) {
        sum += strange(x);
    }
    return sum;
}
```

```assembly
.global zero_to_n
zero_to_n:
push {r4,r5,r6,lr}
movs r4,r0
movs r5,#0
movs r6,#0
check:
cmp r5,r4
bgt done
movs r0,r5
adds r5,#1
bl strange
adds r6,r0
b check
done:
movs r0,r6
pop {r4,r5,r6,pc}
```
What if `strange(int x)` looks like...

```assembly
.global strange
strange:
    movs r4,#0 // because I feel like it.
    movs r5,#0 // la la la
    movs r6,#0 // I’m going to hose this register too.
    movs r7,#0 // Why not!
    muls r0,r0 // Okay, time to do some work.
    muls r0,r0 // More work.
    bx  lr     // I’m done now.
```
This is why we have conventions

- The ABI is a convention.
- Subroutines respect each other this way.
- That way we don’t have to code pessimistically.
  - If every subroutine could modify any register it wanted, then we would need to save registers before we called a subroutine.
Remember this code from a few days ago?

```asm
.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 notified if the button is pressed.
  - Such a thing would be called an interrupt.
  - An interrupt is a type of exception.
    - There are other exceptions: faults, reset, traps.
  - An exception is a hardware-invoked subroutine.
  - We can configure the hardware to invoke a subroutine called an Interrupt Service Routine (ISR) when certain conditions are detected.
Exceptions governed by NVIC

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

STM32F0x datasheet, pg 12
### Types of exceptions

(Table 37, page 217 of FRM)

<table>
<thead>
<tr>
<th>Position</th>
<th>Priority</th>
<th>Settable</th>
<th>Abbreviation</th>
<th>Description</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>N</td>
<td>Reserved</td>
<td></td>
<td>0x0000 0000</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>N</td>
<td>Reset</td>
<td>Reset</td>
<td>0x0000 0004</td>
</tr>
<tr>
<td>-2</td>
<td>N</td>
<td>N</td>
<td>NMI</td>
<td>Nom maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector</td>
<td>0x0000 0008</td>
</tr>
<tr>
<td>-</td>
<td>-1</td>
<td>N</td>
<td>HardFault</td>
<td>All class of fault</td>
<td>0x0000 000C</td>
</tr>
<tr>
<td>-3</td>
<td>Y</td>
<td>SVC</td>
<td>SVC</td>
<td>System service call via SWI instruction</td>
<td>0x0000 002C</td>
</tr>
<tr>
<td>-</td>
<td>5</td>
<td>Y</td>
<td>PendSV</td>
<td>Pendable request for system service</td>
<td>0x0000 0038</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>Y</td>
<td>SysTick</td>
<td>System tick timer</td>
<td>0x0000 003C</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>Y</td>
<td>WWDG</td>
<td>Window watchdog interrupt</td>
<td>0x0000 0040</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>Y</td>
<td>PVD_VDDIO2</td>
<td>PVD and VDDIO2 supply comparator interrupt (+ EXTI lines 16 and 31)</td>
<td>0x0000 0044</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>Y</td>
<td>RTC</td>
<td>RTC interrupts (+ EXTI lines 17, 19, and 20)</td>
<td>0x0000 0048</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>Y</td>
<td>FLASH</td>
<td>Flash global interrupt</td>
<td>0x0000 004C</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>Y</td>
<td>RCC_CRS</td>
<td>RCC and CRS global interrupts</td>
<td>0x0000 0050</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>Y</td>
<td>EXTI0_1</td>
<td>EXTI Line[1:0] interrupts</td>
<td>0x0000 0054</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>Y</td>
<td>EXTI2_3</td>
<td>EXTI Line[3:2] interrupts</td>
<td>0x0000 0058</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>Y</td>
<td>EXTI4_15</td>
<td>EXTI Line[15:4] interrupts</td>
<td>0x0000 005C</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>Y</td>
<td>TSC</td>
<td>Touch sensing interrupt</td>
<td>0x0000 0060</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>Y</td>
<td>DMA_CH1</td>
<td>DMA channel 1 interrupt</td>
<td>0x0000 0064</td>
</tr>
<tr>
<td>10</td>
<td>17</td>
<td>Y</td>
<td>DMA2_CH1_2</td>
<td>DMA channel 2 and 3 interrupts</td>
<td>0x0000 0068</td>
</tr>
<tr>
<td>11</td>
<td>18</td>
<td>Y</td>
<td>DMA2_CH3_4_5</td>
<td>DMA channel 4, 5, 6, and 7 interrupts</td>
<td>0x0000 006C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Position</th>
<th>Priority</th>
<th>Settable</th>
<th>Abbreviation</th>
<th>Description</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>19</td>
<td>Y</td>
<td>ADC_COMP</td>
<td>ADC and COMP interrupts (+ EXTI lines 21/22)</td>
<td>0x0000 0070</td>
</tr>
<tr>
<td>13</td>
<td>20</td>
<td>Y</td>
<td>TIM1_BRK_UP_TRG_COM</td>
<td>TIM1 break, update, trigger and commutation interrupt</td>
<td>0x0000 0074</td>
</tr>
<tr>
<td>14</td>
<td>21</td>
<td>Y</td>
<td>TIM1_CC</td>
<td>TIM1 capture compare interrupt</td>
<td>0x0000 0078</td>
</tr>
<tr>
<td>15</td>
<td>22</td>
<td>Y</td>
<td>TIM2</td>
<td>TIM2 global interrupt</td>
<td>0x0000 007C</td>
</tr>
<tr>
<td>16</td>
<td>23</td>
<td>Y</td>
<td>TIM3</td>
<td>TIM3 global interrupt</td>
<td>0x0000 0080</td>
</tr>
<tr>
<td>17</td>
<td>24</td>
<td>Y</td>
<td>TIM6_DAC</td>
<td>TIM6 global interrupt and DAC underrun interrupt</td>
<td>0x0000 0084</td>
</tr>
<tr>
<td>18</td>
<td>25</td>
<td>Y</td>
<td>TIM7</td>
<td>TIM7 global interrupt</td>
<td>0x0000 008B</td>
</tr>
<tr>
<td>19</td>
<td>26</td>
<td>Y</td>
<td>TIM14</td>
<td>TIM14 global interrupt</td>
<td>0x0000 008C</td>
</tr>
<tr>
<td>20</td>
<td>27</td>
<td>Y</td>
<td>TIM15</td>
<td>TIM15 global interrupt</td>
<td>0x0000 0090</td>
</tr>
<tr>
<td>21</td>
<td>28</td>
<td>Y</td>
<td>TIM16</td>
<td>TIM16 global interrupt</td>
<td>0x0000 0094</td>
</tr>
<tr>
<td>22</td>
<td>29</td>
<td>Y</td>
<td>TIM17</td>
<td>TIM17 global interrupt</td>
<td>0x0000 009B</td>
</tr>
<tr>
<td>23</td>
<td>30</td>
<td>Y</td>
<td>ISC1</td>
<td>FC1 global interrupt (+ EXTI line 23)</td>
<td>0x0000 009C</td>
</tr>
<tr>
<td>24</td>
<td>31</td>
<td>Y</td>
<td>ISC2</td>
<td>FC2 global interrupt</td>
<td>0x0000 00A0</td>
</tr>
<tr>
<td>25</td>
<td>32</td>
<td>Y</td>
<td>SPI1</td>
<td>SPI1 global interrupt</td>
<td>0x0000 00A4</td>
</tr>
<tr>
<td>26</td>
<td>33</td>
<td>Y</td>
<td>SPI2</td>
<td>SPI2 global interrupt</td>
<td>0x0000 00A8</td>
</tr>
<tr>
<td>27</td>
<td>34</td>
<td>Y</td>
<td>USART1</td>
<td>USART1 global interrupt (+ EXTI line 25)</td>
<td>0x0000 00AC</td>
</tr>
<tr>
<td>28</td>
<td>35</td>
<td>Y</td>
<td>USART2</td>
<td>USART2 global interrupt (+ EXTI line 26)</td>
<td>0x0000 00B0</td>
</tr>
<tr>
<td>29</td>
<td>36</td>
<td>Y</td>
<td>USART3_4_5_6_7_8</td>
<td>USART3/4/5/6/7/8 global interrupts (+ EXTI line 28)</td>
<td>0x0000 00B4</td>
</tr>
<tr>
<td>30</td>
<td>37</td>
<td>Y</td>
<td>CEC_CAN</td>
<td>CEC and CAN global interrupts (+ EXTI line 27)</td>
<td>0x0000 00B8</td>
</tr>
<tr>
<td>31</td>
<td>38</td>
<td>Y</td>
<td>USB</td>
<td>USB global interrupt (+ EXTI line 18)</td>
<td>0x0000 00BC</td>
</tr>
</tbody>
</table>
What happens when an exception occurs?

- NVIC decides things like:
  - Is exception to 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 exception.
  - ISR address looked up in the vector table.
    - Branch to that. (i.e. set the PC to the address found)
The Vector Table

- Ranges from 0x0 – 0xB8 in memory.
  - This is 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 of 0x0.
If you look at startup_stm32.s:

```assembly
.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
    ...

.thumb_set HardFault_Handler,Default_Handler
.thumb_set SysTick_Handler,Default_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 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.
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.
  - For now, we’re more interested in SysTick_Handler.
  - ISR should end with "BX LR" or a “POP” whose registers include LR.
    • If you call a subroutine in the ISR, you should first PUSH {LR} and return to user code with 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 regardless of any others.
- Writing a ‘1’ to any bit in ICER disables that interrupt regardless of any others.
- Operationally similar to GPIOx BRR and BSRR
  - So you don’t have to load, OR, mask, AND, store.
- An ISR will only be invoked if an interrupt is enabled.
  - You cannot disable exceptions -3 – -1.
    • 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 BSR/BSRR.
Priority

- 32 Interrupts
- 32 bytes of priority registers
- One byte per interrupt
- Only the two most significant bits of the bytes are used.
  - Possible values: 0, 64, 128, 192
- Must set all 32 bits of each priority word at once.
  - Load, shift, OR, mask, AND, store.
- Priority should be set before interrupt is enabled.
Simplest Example: SysTick

- The SysTick exception is generated periodically by a 24-bit down-counting timer.
  - Set the reset value.
  - Set the clock source (e.g. CPU clock: 8MHz, 48MHz)
  - Enable the counter.
  - When the counter reaches zero, it flags an exception, and loads the reset value.
  - Exception cannot be ignored. Does not need to be enabled.
SysTick Control Registers

- See page 86 of the Cortex-M0 programming manual.

- SYST: base address: 0xe000 e000
  - CSR: (offset 0x10) Control/Status Register
  - RVR: (offset 0x14) Reset Value Register (24 bits)
  - CVR: (offset 0x18) Current Value Register
SYST CSR layout

- Set the low 3 bits to ‘1’ to enable using the CPU clock.
Assembly Language ISR

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

// Setting up SysTick Exception...
    ldr r3, =SYST
    ldr r0, =16000000
    str r0, [r3, #RVR] // Set the reset value

    movs r0, #7
    str r0, [r3, #CSR] // Enable using CPU clock

What's this line???

This tells the linker that the label (address) is for a function! 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.
Example (continued)

// Full program
.cpu cortex-m0
.thumb
.syntax unified

.equ SYST, 0xe000e000
.equ CSR, 0x10
.equ RVR, 0x14

.global main
main:
  // Setting up...
  ldr r3, =SYST
  ldr r0, =16000000
  str r0, [r3, #RVR]  // reset
  movs r0, #7
  str r0, [r3, #CSR]  // Enable
loop:
  b loop  // Endless loop

.type SysTick_Handler, %function
.global SysTick_Handler
SysTick_Handler:
  push {lr}
  pop {pc}

Let's try it.
Execution Model of Exceptions

- 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?
The priority system says which exception handlers can be preempted by other exceptions.
Pending Exceptions

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

Main program

| instr | instr | instr | instr | instr | instr | ... |

Exception Handler (ISR)

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

Exception Handler #2

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

Return

Lower Priority Exception

Exception
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 generates 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
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.


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_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.
Masking the interrupts

- The EXTI_IMR (Interrupt Mask Register) tells the NVIC 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 the Interrupt

- SysTick is a non-maskable exception.
- Other exceptions must be explicitly enabled.
- Turn on the interrupt number in NVIC_ISER
  - Section 4.2 of the Cortex-M0 programming manual
  - Slide #16 of these lecture notes.
- Once configured and enabled, the ISR will be invoked when the event occurs.
Recap

- To enable an interrupt for the rising edge of PA0:
  - Set bits 3:0 of SYSCFG_EXTICR1 to zero.
    - This is the default, 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 (the interrupt for pins 0 and 1) of the NVIC_ISER.
Acknowledging an Interrupt

- Doing all those steps will invoke the interrupt, 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.