EL-GY 6483
Interrupts

- This homework assignment is not graded.
- You are encouraged to work in groups, and to use the Internet or any other tools available to learn the material in order to answer these questions.

---

1. Which of the following statements are true? Mark all that apply.
   (a) An interrupt is always an urgent, high-priority task.
   (b) Using interrupts is always faster than polling.
   (c) System latency is always larger than interrupt latency.
   (d) Global variables used within an ISR should be declared volatile.
   (e) The interrupt vector table has to be placed in a specific location in memory.
   (f) On the ARM Cortex-M4, if two interrupts with priority numbers 0 and 1 occur simultaneously, the interrupt controller (permanently) clears the one numbered 1 and passes the one numbered 0 to the CPU.
   (g) Level 0 is the highest (most urgent) interrupt priority on the ARM Cortex-M4.
   (h) The startup code for the STM32F4 Discovery includes assembly code to save registers r0-r3 to the stack before entering an ISR.

Solution:

(a) **An interrupt is *not* always an urgent, high-priority task.** For example, an asychnronous input from a peripheral that doesn't need an immediate response might still be interrupt-driven.

(b) **Using interrupts is *not* always faster than polling.** This tradeoff depends on the interrupt latency of the system, the frequency of the event you are polling/interrupting from, and possibly other factors.

(c) **System latency is always larger than* interrupt latency.** System latency is the interrupt latency plus the maximum amount of time that interrupts might be disabled. *It is at least as large as the interrupt latency.

(d) **Global variables used within an ISR should be declared volatile.** This is true; since ISRs are triggered asynchronously, the compiler cannot predict when they will occur (and so might "optimize" out operations involving these variables), and we always want the freshest data in an ISR (declaring as volatile requires the compiler to load the value again every time it is used, instead of caching it).

(e) **The interrupt vector table has to be placed in a specific location in memory.** True; the startup file lays out the interrupt vector table and marks it as such; the linker script describes where in memory the interrupt vector table should go. The processor will look for specific interrupt vectors at the expected locations in memory.

(f) **On the ARM Cortex-M4, if two interrupts with priority numbers 0 and 1 occur simultaneously, the interrupt controller *does not* (permanently) clear the one numbered 1 and passes the one numbered 0 to the CPU.** The interrupt with lower priority will remain in the pending state, and may still execute after the high-priority interrupt completes (if the other conditions enabling it to execute are true). See page 14 of lecture slides for further details.

(g) **Level 0 is the highest (most urgent) interrupt priority on the ARM Cortex-M4.** This is true. Lower numbers have higher priority on this processor.

(h) **The startup code for the STM32F4 Discovery *does not* include assembly code to save registers r0-r3 to the stack before entering an ISR.** The automatic context saving happens automatically on the Cortex-M4 and does not need to be added in the firmware. (This is the case for most microcontrollers.) "Extra" context saving (such as saving registers r4+) would have to happen in software.

2. Can an interrupt service routine ever return a value? Can an interrupt service routine take arguments? Why or why not?

   (You should be able to answer this question based on your understanding of interrupts, without having to explicitly look up the answer.)

Solution: An ISR is not called from program code, so we can not pass parameters to it via arguments. And because there is no calling code to read a return value, an ISR also cannot return a value.

3. Under what conditions may an interrupt service routine safely use an SPI bus that has multiple slaves on it?

   (You should be able to answer this question based on your understanding of interrupts and the SPI protocol, without having to explicitly look up the answer.)

Solution: SPI is not safe to interrupt. If a transaction has begun on an SPI bus, the bus may not be used for anything else until the transaction is complete.

For example, consider a program where the main loop sends data to slave 0 over an SPI bus, and an interrupt handler may use the same SPI bus to send data to slave 1. If an interrupt happens as main loop is using the SPI bus, the interrupt may set the CS line for slave 1 while the CS line for slave 0 is still flow. The interrupt handler would then unwittingly send its data with both slaves listening!

To safely share an SPI bus between an interrupt handler and a main task or other interrupt handler, you would have to disable the (other) interrupts that may use the SPI bus before you begin a transaction, and enable the interrupts again when your transaction is complete.

4. The startup code we have been using in the lab sets up the standard interrupt vector table for the Cortex-M3 and Cortex-M4 processors.

   Refer to this file (`startup_stm32f4xx.S`) and the STM32F4 Discovery Reference Manual to answer these questions (for the Discovery board and this particular startup file):

   (a) Write C code to define an ISR that's triggered by the USART1 peripheral. (You don't have to set up and enable the interrupt, just write the ISR.) The ISR should be a noop (i.e., do nothing).

Solution:
From the startup file, we can see that the interrupt handler should be named `USART1_IRQHandler`, so our function definition would look like this:

```
void USART1_IRQHandler(void){
  // do nothing
}
```

(b) If you enable an interrupt in your C code but don't define the ISR, what code will execute when the interrupt is triggered?

Solution:
In the startup file, we find that all of the interrupt handlers are initially mapped to a default interrupt handler with a "weak alias." A comment in the startup file explains:

```
/*******************************************************************************
*
* 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.
*
*******************************************************************************/
```

The default handler just executes an infinite loop:

```
/**
 * @brief  This is the code that gets called when the processor receives an
 *         unexpected interrupt.  This simply enters an infinite loop, preserving
 *         the system state for examination by a debugger.
 * @param  None
 * @retval None
*/
    .section  .text.Default_Handler,"ax",%progbits
Default_Handler:
Infinite_Loop:
  b  Infinite_Loop
  .size  Default_Handler, .-Default_Handler
```

So, if we don't define an interrupt handler in our program code, and the interrupt is triggered, it will execute the default handler and run an infinite loop.

(c) Write an `EXTI15_10_IRQHandler` (just the ISR, you don't have to enable the interrupt). Assume you do *not* have any peripheral library functions, but you do have the following `define` statements:

```
#define     __IO    volatile

typedef struct
{
  __IO uint32_t IMR;    /*!< EXTI Interrupt mask register,            */
  __IO uint32_t EMR;    /*!< EXTI Event mask register,                */
  __IO uint32_t RTSR;   /*!< EXTI Rising trigger selection register,  */
  __IO uint32_t FTSR;   /*!< EXTI Falling trigger selection register, */
  __IO uint32_t SWIER;  /*!< EXTI Software interrupt event register,  */
  __IO uint32_t PR;     /*!< EXTI Pending register,                   */
```

```
} EXTI_TypeDef;

#define PERIPH_BASE          ((uint32_t)0x40000000)
#define APB2PERIPH_BASE      (PERIPH_BASE + 0x00010000)
#define EXTI_BASE            (APB2PERIPH_BASE + 0x3C00)
#define EXTI                 ((EXTI_TypeDef *) EXTI_BASE)

#define EXTI_Line10      ((uint32_t)0x00400)     /*!< External interrupt line 10 */
#define EXTI_Line11      ((uint32_t)0x00800)     /*!< External interrupt line 11 */
#define EXTI_Line12      ((uint32_t)0x01000)     /*!< External interrupt line 12 */
#define EXTI_Line13      ((uint32_t)0x02000)     /*!< External interrupt line 13 */
#define EXTI_Line14      ((uint32_t)0x04000)     /*!< External interrupt line 14 */
#define EXTI_Line15      ((uint32_t)0x08000)     /*!< External interrupt line 15 */
```

- Remember, you do *not* have any peripheral library functions.
- Your ISR should increment a (previously declared) global variable named `firstCounter` if the interrupt was triggered by line 11, and increment a (previously declared) global variable named `secondCounter` if the interrupt was triggered by line 12.
- It should also clear the pending register for the line in both cases.
- To test if an interrupt was triggered by a particular line, you need to check whether the pending register flag is set for that line AND whether interrupts are enabled (i.e. not masked) for that line.
- Refer to the section beginning on page 378 of the Technical Reference Manual for more information.
- Your ISR should also handle the case where both lines trigger an interrupt.

Solution:

```
void EXTI15_10_IRQHandler(void) {
  // check pending register flag for line 11
  // check enable register status for line 11
  if ((EXTI->PR & EXTI_Line11) && (EXTI->IMR & EXTI_Line11)) {
    firstCounter++;

    // note that the reference manual says regarding the PR register:
    // "This bit is cleared by programming it to '1'."  Hence:
    EXTI->PR = EXTI_Line11;
  }

  // use a second if, not an else if, so we handle cases
  // where both interrupts are triggered

  // check pending register flag for line 12
  // check enable register status for line 12
  if ((EXTI->PR & EXTI_Line12) && (EXTI->IMR & EXTI_Line12)) {
    secondCounter++;

    // note that the reference manual says regarding the PR register:
    // "This bit is cleared by programming it to '1'." Hence:
    EXTI->PR = EXTI_Line12;
  }
}
```