Lab Assignment: Interrupt + Lookup Tables + Binary Semaphores

Objective

To learn how to create a single dynamic user defined interrupt service routine callback driver/library.

This lab will utilize:

- Semaphores
  - Wait on Semaphore Design pattern
- Lookup table structures
- Function pointers
- Interrupts
  - LPC supports rising and falling edge interrupts on certain pins
  - These port/pin interrupts are actually OR'd together and use a single CPU interrupt called EINT3 (External Interrupt 3)

Port Interrupts

You will configure GPIO interrupts. This is supported for Port0 and Port2 and the following registers are relevant.

Table 103. GPIO interrupt register map

Generic Name	Description	Access	Reset value[1]	PORTn Register Name & Address
IntEnR	GPIO Interrupt Enable for Rising edge.	R/W	0	IO0IntEnR - 0x4002 8090 IO2IntEnR - 0x4002 80B0
IntEnF	GPIO Interrupt Enable for Falling edge.	R/W	0	IO0IntEnR - 0x4002 8094 IO2IntEnR - 0x4002 80B4
IntStatR	GPIO Interrupt Status for Rising edge.	RO	0	IO0IntStatR - 0x4002 8084 IO2IntStatR - 0x4002 80A4
IntStatF	GPIO Interrupt Status for Falling edge.	RO	0	IO0IntStatF - 0x4002 8088 IO2IntStatF - 0x4002 80A8
IntClr	GPIO Interrupt Clear.	WO	0	IO0IntClr - 0x4002 808C IO2IntClr - 0x4002 80AC
IntStatus	GPIO overall Interrupt Status.	RO	0	IOIntStatus - 0x4002 8080

Assignment

Part 0: Simple Interrupt

The first thing you want to do is get a single Port/Pin's interrupt to work.

```
void interrupt_callback(void)
{
  // TODO: Clear the source of the EINT3 interrupt
  // Maybe uart0_puts() or blink and LED here to test your ISR
}
void main(void)
{
  // Register your callback for the EINT3
  isr_register(EINT3_IRQn, interrupt_callback);

  // Configure the registers to trigger Port2 interrupt (such as P2.0 rising edge)

  while (1)
  {
    continue; // Empty loop just to test the interrupt
  }}
```

Code Block 1. Basic Interrupt Test

Part 1: Design the **GPIOInterrupt** driver

You are designing a library that will allow the programmer using your library to be able to "attach" a function callback to any and each pin on port 0 or port 2.

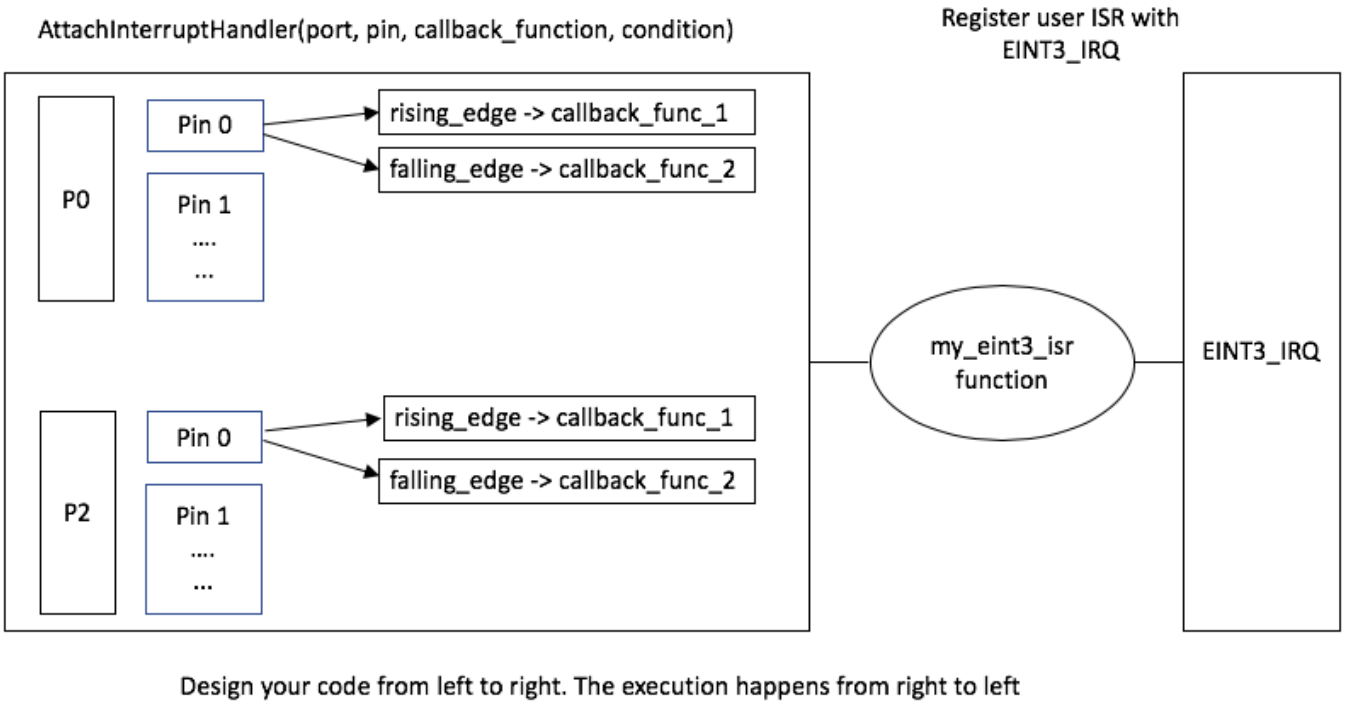
1. Implement ALL class methods.
2. All methods must function work as expected by their comment description.

```
enum InterruptCondition
{
    kRisingEdge,
    kFallingEdge,
    kBothEdges,
};
/**
 * Typdef a function pointer which will help in code readability
 * For example, with a function foo(), you can do this:
 * IsrPointer function_ptr = foo;
 * OR
 * IsrPointer function_ptr = &foo;
 */
typedef void (*IsrPointer)(void);
class LabGpioInterrupts
{
 public:
    /**
     * Optional: LabGpioInterrupts could be a singleton class, meaning, only one instance can exist at
     * Look up how to implement this. It is best to not allocate memory in the constructor and leave c
     * code to the Initialize() that you call in your main()
     */
    LabGpioInterrupts();
    /**
     * This should configure NVIC to notice EINT3 IRQs; use NVIC_EnableIRQ()
     */
    void Initialize();
    /**
     * This handler should place a function pointer within the lookup table for the HandleInterrupt()
```

```
     *
     * @param[in] port         specify the GPIO port, and 1st dimension of the lookup matrix
     * @param[in] pin          specify the GPIO pin to assign an ISR to, and 2nd dimension of the looku
     * @param[in] pin_isr      function to run when the interrupt event occurs
     * @param[in] condition    condition for the interrupt to occur on. RISING, FALLING or BOTH edges.
     * @return should return true if valid ports, pins, isrs were supplied and pin isr insertion was su
     */
    bool AttachInterruptHandler(uint8_t port, uint32_t pin, IsrPointer pin_isr, InterruptCondition cond

    /**
     * This function is invoked by the CPU (through Eint3Handler) asynchronously when a Port/Pin
     * interrupt occurs. This function is where you will check the Port status, such as IO0IntStatF,
     * and then invoke the user's registered callback and find the entry in your lookup table.
     *
     * VERY IMPORTANT!
     *  - Be sure to clear the interrupt flag that caused this interrupt, or this function will be call
     *    repetitively and lock your system.
     *  - NOTE that your code needs to be able to handle two GPIO interrupts occurring at the same time
     */
    void HandleInterrupt();
 private:
    /**
     * Allocate a lookup table matrix here of function pointers (avoid dynamic allocation)
     * Upon AttachInterruptHandler(), you will store the user's function callback
     * Upon the EINT3 interrupt, you will find out which callback to invoke based on Port/Pin status.
     */
    IsrPointer pin_isr_map[2][32];
};
/**
 * Unless you design Singleton class, we need a global instance of our class because
 * the asynchronous Eint3Handler() will need to invoke our C++ class instance callback
 * WARNING: You must use this same instance while testing your main()
 */
LabGpioInterrupts gpio_interrupt;
/* Since we have a C++ class handle an interrupt, we need to setup a C function delegate to invoke it
 * So here is the skeleton code that you can reference.
```

```
 * This function will simply delegate the interrupt handling to our C++ class
 * The CPU interrupt should be attached to this function through isr_register()
 */
void Eint3Handler(void)
{
    gpio_interrupt.HandleInterrupt();
}
/**
 * main() should register C function as callback for the EINT3
 * This is because we cannot register a C++ function as a callback through isr_register()
 *
 * There are workarounds, such as static functions inside a class, but that design is
 * not covered in this assignment.
 */
void main(void)
{
    // Init things once
    gpio_interrupt.Initialize();

    // Register C function which delegates interrupt handling to your C++ class function
    isr_register(EINT3_IRQn, Eint3Handler);

    // Create tasks and test your interrupt handler}
```

Code Block 2. GPIO Interrupt Driver Template Class



Design your code from left to right. The execution happens from right to left

Extra Credit: Use Driver to Optimize GPIO Application

As the title says, you will attempt to optimize your previous lab by utilizing interrupts and semaphores. This time, you will eliminate the ~~**vReadSwitch**~~ task, and utilize a interrupt service routine to send semaphores (fromISR) to the **vControlLED** task.

Requirements

- Should be able to specify a callback function for any port/pin for an exposed GPIO given a rising, falling, or both condition.
  - We may ask you to change which port and pin causes a particular callback to be executed in your code and then recompile and re-flash your board to and prove it works with any port 0 or port 2 pin.
- You will need to use two external switches for this lab.

> You cannot use printf() to print anything from inside an ISR (if FreeRTOS is running), but you can use the u0_dbg_printf() API from printf_lib.h.

> Note that printing 4 chars inside an ISR can take 1ms, and this is an eternity for the processor and should never be done (other than debug).

Skeleton Test Code:

```
void user_callback(void)
```

```
{
  // This is where you will "send" a Semaphore that the vControlLED task is waiting on
}
void main(void)
{
  // ISA team may modify Port/Pin to test your GPIO Interrupt class
  gpio_intr_instance.attachInterruptHandler(2, 3, user_callback, rising_edge);}
```

What to turn in:

- Place all relevant source files within a .pdf file.
- Turn in the **screenshots** of terminal output.

Revision #21
Created 1 year ago by Admin
Updated 4 months ago by Preet Kang