Microprocessor Systems Lab 2

Checkoff and Grade Sheet

Partner 1 Name: _____

Partner 2 Name: _____

Grade Component	Max.	Points Awarded Partner 1	Points Awarded Partner 2	TA Init.s	Date
Performance Verification: Task 1	10 %				
Task 2	15 %				
Task 3	15 %				
Task 4 [Depth]	10 %				
Documentation and Appearance	50 %				
Total:					

→ Laboratory Goals

By completing this laboratory assignment, you will learn to:

1. Add interrupt service routines to the C program,
2. Use Interrupt Requests (IRQs) and Timers on the STM32F769NI.

→ Reading and References

R1. Mastering STM32: Chapters 7 (Interrupts) ,11 (Timers)

R2. stm32f769ni_Datasheet.pdf: Skim Chapters 1 and 2

R3. RM0410-stm32f7_Reference_Manual.pdf: Chapters 5 (RCC),6 (GPIO),10 (NVIC), 11 (EXTI), 28 (Basic Timers)

R4. PM0253-stm32f7_Programming_Manual.pdf: Section 4.2 (NVIC)

R5. UM1905-stm32f7_HAL_and_LL_Drivers.pdf: Chapters 10 (CORTEX/NVIC) 26 (GPIO), 64 (Timers)

R6. Lab02_Interrupt_Timers_Template.zip: Project Template for Lab 2

STM32 Interrupts

An interrupt within a microcontroller is a request for service by a process that is running outside the direct control of the program, a *background* process. That is, the process is not a subroutine or other subprogram that is called by the program at a point in its execution. For example, a keyboard input is not controlled by the user's program but by a key being pressed. In real-time applications, interrupts are often used to perform periodic functions such as to sample data or to update the error input to a servo loop. Other functions are to measure the elapsed time from a reference starting time or to measure the time interval between external events. In this laboratory exercise, both time-based and external interrupts are used.

For any background process to request service, several conditions must be met, which depend on the module being used. For all modules, the interrupt service vector for the corresponding (sub)module must be enabled in the **Nested Vector Interrupt Controller (NVIC)**.

→ Nested Vector Interrupt Controller (NVIC)

All interrupts are controlled through the ARM standard NVIC system, even interrupts that are exclusive to a specific product line. Shown in Table 46 of R3 (p.313-318) is the complete list of interrupts available on the STM32F769NI. These interrupt vectors may be enabled and disabled independently. Additionally, the priority of each of these interrupt vectors is, by default, incremented with the position of the interrupt vector; for example, the interrupt at position 18 (ADC) has a priority of 25 while the interrupt at position 19 (CAN1_TX) has a priority of 26. These priorities may be changed through the NVIC registers.

To enable an interrupt vector in the NVIC, the `NVIC_ISERx` registers are used [R4]; though alternatively, functions provided from the ARM CMSIS driver may be used instead. There are eight `NVIC_ISERx` registers, allowing for a theoretical maximum of 256 interrupt vectors to be used; however, Cortex-M3/4/7 microcntrollers are limited to a maximum of 240 vectors. The mapping of the `NVIC_ISERx` bits is such that the interrupt vector at position 0 (WWDG) is controlled by bit 0 of `NVIC_ISER0`, position 31 (I2C1_EV) is controlled by bit 31 of `NVIC_ISER0`, position 32 (I2C1_ER) by bit 0 of `NVIC_ISER1` and so forth. This may be implemented in code as:

```
NVIC->ISER[IRQn / 32] = (uint32_t) 1 << (IRQn % 32);
```

where `IRQn`[1] is the interrupt vector position number listed in the table.

→ Extended Interrupts and Events Controller (EXTI)

The EXTI mainly supports interrupt functionality for external lines, but also some other sources (See Section 11.8 of R3). In order for a GPIO line to trigger an interrupt, it must be routed properly by the EXTI module as well as enabled in the NVIC.

→ Interrupt Request (IRQ) Handlers

Implementation of interrupt service routines (ISR) is done through IRQ Handlers. When an IRQ occurs, the microcontroller pauses the current code execution and saves the state. Then, the associated ISR is executed. The determination of the correct ISR is done through function handles stored in a location in memory reserved for the IRQ trigger, where the memory layout is specified by the file `startup_stm32f769xx.s`

[1] IRQ stands for "Interrupt Request" or "Interrupt Request Lines," the physical hardware lines that triggers the interrupt.

starting near line 140. In order for the correct ISR to be called, the name of the written ISR must match that of the one provided in the layout file. The function name is of the form {IRQ_NAME}_IRQHandler, where {IRQ_NAME} is given in the acronym column of Table 46 of R3. For example, an ISR written to service an ADC (IRQ number 25) should be written as:

```
void ADC_IRQHandler() {
        \\ Clear interrupt pending flag here
        \\ ISR Contents...
}
```

Care must be taken to ensure that the IRQ triggers which raised the IRQ Handler are cleared prior to exiting the function. Further, due to delays in the propagation of the IRQ triggers from the peripherals (e.g., a timer) to the NVIC system, it is possible for an IRQ Handler to be retriggered by the flag that was already cleared. A short delay can be added at the end of the IRQ Handler function to prevent this from happening. This delay may be implemented by adding several `asm("nop")` instructions.

STM32 Interrupts using HAL

Use of interrupts when implementing functionality using the HAL drivers is significantly different than when implementing with registers. This is due to the interrupt enabling, disabling and handling being mostly provided by the library of the module (e.g., GPIO), though direct enabling and disabling still needs to be done through `HAL_NVIC_EnableIRQ()`. Callback functions need to be written by the user that effectively serve as the ISR yet the standard {IRQ_NAME}_IRQHandler() function still needs to be called and pointed towards the HAL library via the function HAL_{IRQ_NAME}_IRQHandler(). For example, ADC interrupt support may be implemented by the following user provided functions:

```
void ADC_IRQHandler(){
        HAL_ADC_IRQHandler(&adc_handle);
}

void HAL_ADC_ConvCpItCallback(ADC_HandleTypeDef * hadc){
        \\ ISR Contents...
}
```

where `HAL_ADC_ConvCpItCallback()` is a "callback"[2] function which is written by the user and called by `HAL_ADC_IRQHandler()` (provided by HAL), if applicable; that is, the specific interrupt trigger source "Conversion Complete" occurred. Likewise, there may be a multitude of callback functions that exist for a module which are triggers by various different sources.

The functionality required to enable these interrupts in HAL varies between modules. For instance, for GPIO, the interrupt is enabled within the module by setting the mode of the pin to not simply `GPIO_MODE_INPUT` but instead using `GPIO_MODE_IT_`*xxx*. Of course this assumes that the interrupt is globally enabled within the NVIC.

[2]In general, HAL driver use entails the user calling the HAL functions; however in certain cases, the HAL will need to call user space functions. This leads to the name "callback," or functions existing in the user space which are called from the driver.

◇ Task 1: GPIO Interrupt

Write a simple C program that responds to two External Interrupts via the signals `EXTI0` and `EXTI8`. The external sources for `EXTI0` and `EXTI8` can be either Analog Discovery digital lines as connected in Lab 1 or physically wired pushbuttons using an protoboard (with appropriate pull-up/down resistors). The ISR code should set a global variable that can be seen by the main routine. Within the main routine, indication of when the pushbutton was pressed should be output to the terminal.

One of external interrupts should be entirely implemented using register configuration by one partner and the other using HAL by the other partner. Note that the partner who completed Task 3 of Lab 1 using HAL should do the registers portion of this task and vice versa. This task should result in only one program.

NOTES:

1. Do not forget to enable the RCC clocks for each GPIO port used.
2. Only certain pins are capable of being routed to each `EXTI`x signal. Make sure the correct pins are selected.
3. It may be easier to develop the two portions of this task in separate files then merge after completion.
4. The register implementation may suffer from being too efficient: clearing the interrupt flag that triggered the interrupt handler is *not immediate*. Therefore, if the flag is cleared and the interrupt ends within a few cycles, it is possible that the interrupt will be re-triggered incorrectly. This may be corrected by adding small wait at the end of the interrupt (e.g., an empty for loop that counts to 10).

Timers

The STM32F769NI has 14 programmable timer systems with three tiers of functionality. These tiers are labeled "Basic," "General Purpose," and "Advanced." The functional differences between these timer types are summarized in Section 11.1.1 of R1. Basic timers are 16-bit timers that are not exposed to an output; for example, pulsewidth modulation outputs, and are mainly only useful in producing time bases. General Purpose Timers may produce an output, can be 16- or 32-bit, and also have more advanced control features. The Advanced Timers have even more advanced control options in addition to the previous.

The 14 timers are labeled TIM0-TIM14, and a summary of their functionality is given in Table 6 of R2 (p.38). A simplified list is given below in Table 1.

→ Basic Counting

In this lab, only the basic timers should be used, although both the general purpose and advanced timers may be simplified to achieve the same result. One of the most common uses of a basic timer is to generate an interrupt after a certain time has passed. This allows for programs to incorporate a sense of time progression and have events occur at predetermined and consistent intervals. The basic timers are 16 bits in size, allowing for a maximum of 65536 tick counts to pass prior to the generation of an interrupt; however, if the auto-reload register is used (`TIM`n`_ARR`), the maximum count may be limited to the size

specified within this register. Further, the time required for each counted tick to occur is dependent on both the clock provided to the timer subsystem as well as the clock prescaling register `TIMn_PSC`.

For convenience, the overflow time calculation is given in Equation 1.

$$\text{Overflow Time} = (\texttt{TIM}n\texttt{\_ARR} + 1)\left(\frac{\texttt{TIM}n\texttt{\_PSC} + 1}{\texttt{TIM}n\texttt{CLK}}\right) \tag{1}$$

where `TIMnCLK` is the clock provided to the timer through the RCC system (see `RCC_APB1ENR`). For this lab, `TIMnCLK` has been preconfigured within `init.c` to provide a 108 MHz. For reference, the `SYSCLK`, named on this microcontroller as `HCLK`, is configured to 216 MHz.

→ Timer Interrupts

Each timer controls it's own interrupt generation and pending flags, unlike the GPIO, where all of the interrupts are handled by the EXTI. All of the interrupt sources available to a timer are contained within the `TIMn_DIER` register, along with additional functionality (e.g., DMA interactions). For basic timers, the only interrupt source available is the "Update interrupt," which, by another name, is the overflow interrupt. In general purpose and advanced timers, multiple capture and compare interrupts are also available, among others. Similarly, the pending flags for each one of those interrupts is contained within the `TIMn_SR` register.

◇ Task 2: Simple Timer: Register Implementation

Write a C program that responds to a timer interrupt to display elapsed time in tenths of seconds using one of the basic timers of the microcontroller. The timer should be implemented using registers. The timer should be as accurate as possible without changing the RCC subsystem, though it is allowed if desired.

NOTES:

1. Don't forget to turn the timer on.
2. Take care not to change any of the module's registers until at least two clock cycles have passed after turning on the modules corresponding clock in the RCC. Two clock cycles may be taken up by twice issuing the assembly command NOP (No OPeration) via the C command `asm("nop");` if needed.
3. If the `ARPE` bit in `TIMn_CR1` is true, a newly written value to `TIMn_ARR` will not be used until after the next overflow.

Table 1: Type and size of timers available on STM32F769NI

Timers	**Size**	**Type**
TIM1,TIM8	32-bit	Advanced
TIM3,TIM4	16-bit	General
TIM2,TIM5	32-bit	General
TIM6,TIM7	16-bit	Basic
TIM10,TIM11,TIM13,TIM14	16-bit	General (1-channel)
TIM9,TIM12	16-bit	General (2-channel)

→ HAL Timers

The HAL interface for timers is a bit more complex than that for the GPIO. The HAL timer functions generally require as the first argument a timer handle, defined with the type `TIM_HandleTypeDef *`; which has several custom struct types as fields. It is highly suggested to reference R1 for a clear discussion of how to use these structs.

For this lab's application, only base timers are requested and therefore the bulk of the HAL driver may be ignored; that is, only the `HAL_TIM_Base` configuration and control functions will be needed paired with `HAL_TIM_IRQHander()` and the associated callback function(s).

One complication in using the HAL to configure and use the timers is that the functions provided are not specific to each timer: the functions are not named `HAL_TIMx_` but instead `HAL_TIM_`. This implies that all of the timers are configured and controlled through the same set of functions. This also applies to the HAL IRQ handler and callback functions. Therefore, handles for each timer used within the HAL need to exist as global variables such that they may be passed to and used within these functions. For example, if two HAL timers are used to produce time bases, then both of these timers will triggers overflow interrupts which with both be handled by a single callback function. Within that callback function, it is necessary to determine the trigger source, provided through the timer handle passed to the callback function:

```
\\ Callback function contents
if (htim->Instance == TIM0) {
        \\ Do stuff pertaining to TIM0 interrupt
} else if (htim->Instance == TIM14) {
        \\ Do stuff pertaining to TIM14 interrupt
}
```

where `htim` is the timer handle passed to the function (user specified). Clearing of interrupt flags is not necessary when using the HAL, as it is directly taken care of by the `HAL_TIM_IRQHander()` function.

◇ Task 3: Simple Timer: HAL Implementation

Repeat Task 2 using the other basic timer available on the STM32F769NI but use a different configuration (e.g., should not use the same pre-scaler or auto reload register value) and implement using HAL.

◇ Task 4: [Depth] Number Entry

Create a program that uses a single pushbutton to enter a series of multi-digit numbers. A number's entry starts with setting the most significant digit of the number first then proceeding to the next digit (left-to-right). A short press of the pushbutton (<1s) will increment the digit that is currently being entered. Not pressing the button for 1s or longer should cause the entry to await the next digit's entry; although the next digit should not be added until the push button is pressed to provide a value. A long press of the pushbutton (>1s) should trigger the program to record the current number as complete and move to the next (blank) number. A very long press (>3s) should trigger the program to print out the recorded numbers and disallow further entry (a terminating `while(1);` is acceptable for this).

No code is allowed in the `while(1)` loop within the `main` function; all functional code, excluding initializations, should exist within the relavent interrupt(s).

NOTES:

1. Coming Soon...