Microprocessor Systems

ELE 271

Laboratory 5:

Pushbutton, LED, and Interrupts

Due 20MAR2023

By: Nick Bean ,Taylor Brookes, Toussaint Falangani

**Introduction**

The purpose of this lab is to learn about external interrupts. External interrupts are interrupts triggered by peripherals or devices, external to the microprocessor core, such as push buttons, keypads, etc. Interrupts allow for more efficient monitoring of external inputs by using dedicated hardware instead of software via the so-called busy-waiting or polling method. From our previous lab (lab 4) we were able to utilize the general purpose input/output (GPIO) pins for digital input and output. This concept will also be applied here because external interrupts are associated with GPIO pins.

* **In part 1** of our lab, we will configure our pins; where PA.5 is the output LED and PC.13 is the input pushbutton. We will configure the external interrupt; according to the textbook, there are 8 steps to follow in order to select the GPIO pin as the trigger source for an external interrupt. We found that not all were necessary for proper functioning of the program. We will then use an interrupt service routine to toggle the LED every time the pushbutton is pressed on either the rising or falling edge.
* **Part 2** of our lab toggles the green LED on every falling edge of a square wave input from a function generator and captures the input square wave and the output signal sent to the GPIO output pin PC.1 by making use of the logic analyzer. With this information we can deduce the latency delay resulting from the overhead in servicing the interrupt and record the frequency at which the microcontroller no longer keeps up with the input signal.

**Part 1**

To begin the experiment, we start with configuring the various ports utilized. The push button and LED configuration remains the same as previous experiments.

[Configuration of LED]

void configure\_LED\_pin(){

RCC -> AHB2ENR |= RCC\_AHB2ENR\_GPIOAEN;

RCC -> AHB2ENR |= 0x00000001;

GPIOA -> MODER &= ~(3UL<<10);

GPIOA -> MODER |= 1UL<<10;

GPIOA -> OTYPER &= ~(1UL<<5);

GPIOA -> OSPEEDR &= (1UL<<10);

GPIOA -> PUPDR &= ~(3<<10);

}

}

[Configuration of Push Button]

void configure\_BUTTON\_pin(){

RCC -> AHB2ENR |= RCC\_AHB2ENR\_GPIOCEN;

RCC -> AHB2ENR |= 0x00000001;

GPIOC -> MODER &= ~(3UL<<26);

GPIOC -> PUPDR &= ~(3<<26);

}

To begin our new code, we start by enabling the SYSCFG clock. This allows us to utilize the SYSCFG module. Next, we configure the system configuration external interrupt configuration register (SYSCFG EXTICR) to map GPIO PC.13 (pushbutton) to external interrupt input line 13. We use EXTICR[3] because 13 is the 3rd hex bit (0x1FFF), and EXTICR4 because it is 1-indexed instead of 0-indexed. We then select the active edge that can trigger EXTI13. The signal has to have a rising edge or a falling edge or else there will be no output. Both rising and falling can be used simultaneously in this part, but not in part 2. This is programmed via the rising edge trigger selection register (EXTI\_RTSRl) and the falling edge trigger selection register (EXTI\_FTSRl). We chose to enable the rising edge because part 2 asks us to enable the falling edge. We then enable EXTI13 by setting the 13th bit in EXTI interrupt mask register (EXTI\_IMRl) to 1. An interrupt can only be generated if the corresponding bit in the interrupt mask register is 1 (or called unmasked). This tells the program to not ignore the EXTI13 (the interrupt we are using). We then enable interrupt EXTI15\_10\_IRQn on NVIC (nested vectored interrupt controller) via NVIC\_EnableIRQ. This function enables the specified device specific interrupt IRQn. Each bit enables one interrupt. It is EXTI15\_10 instead of EXTI13 because EXTI15\_10 is the cluster of EXTIs that are between and including 10 and 15.

[Configuration of EXTI]

void configure\_EXTI(){

//enable SYSCFG clock

RCC->APB2ENR |= RCC\_APB2ENR\_SYSCFGEN;

//configure EXTICR to connect to PC.13

SYSCFG->EXTICR[3] |= SYSCFG\_EXTICR4\_EXTI13\_PC;

//configure EXTI registers to enable interrupts

// rising edge = 1

EXTI->RTSR1 |= EXTI\_RTSR1\_RT13;

// enable interrupt = unmasked (1) (tells program to not ignore this interrupt)

EXTI->IMR1 |= EXTI\_IMR1\_IM13;

//enable interrupt PC.13

NVIC\_EnableIRQ(EXTI15\_10\_IRQn);

}

Here we are changing / giving a definition to the EXTI15\_10IRQHandler (it is formatted similarly to defining a new function; however, because it is already defined in the imported .s file we are simply changing the existing function’s definition). We first start by checking that interrupt flag 13 has been tripped. This lets us know that we are using the interrupt correctly, otherwise the interrupt would always be on, such as calling the function in a dead while loop in the main method. We then toggle the LED by XOR’ing the ODR and waiting until the button has stopped being pressed by using a while loop and comparing IDR to 0x0000 (when the button is pressed, its state is 0). Afterwards, we clear the EXTI flag by resetting the flag to 1. This can either be done through decimal (1), hex (0x2), bit-shifting (1<<1), or, because we know that EXTI\_PR1\_PIF13’s value is already equal to 1 because it has been tripped, setting the value to pending bit 13. PR1 is Pending Register. We use PR1 instead of PR2 because we are interested in the first 32 registers. PIF13 is the pending bit of event 13.

[Configuration of EXTI15\_10\_IRQHandler]

//calling from the .s file, change this function by:

void EXTI15\_10\_IRQHandler(void){

if ((EXTI->PR1 & EXTI\_PR1\_PIF13) == EXTI\_PR1\_PIF13) {

GPIOA -> ODR ^= 1UL<<5;

while (GPIOC -> IDR == 0x0000);

EXTI->PR1 |= EXTI\_PR1\_PIF13;

}

}

In the main function we call all of the previously defined configuration functions and then execute an infinite while loop to ensure the program remains running. The primary difference to note here is that in previous experiments we would constantly be asking the processor if a case was true or false (a process referred to as busy-waiting or polling). By taking advantage of the interrupt register, we can just tell the program to execute the while loop forever and those conditions will always be implicitly checked by looking at the interrupt flags and executing the interrupt when one is triggered.

[Configuration of main method]

int main(void){

configure\_LED\_pin();

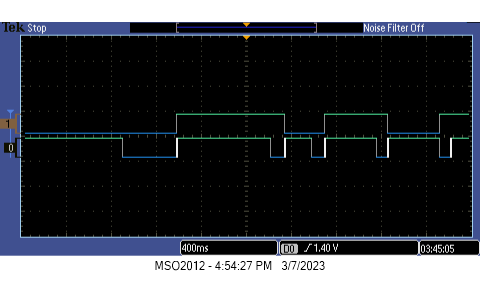
configure\_BUTTON\_pin();

configure\_EXTI();

while(1);

}

[Demonstration of LED (channel 1) changing with each rising edge of pushbutton (channel 0)]



**Part 2**

For part 2 the same LED Configuration is used, and in place of the push button we use an input port (PC1\_pin) to read an external square wave signal (provided by the PB505). To accomplish this, we change the name of the method configure\_BUTTON\_pin() to configure\_PC1\_pin() and address the 2nd bit in place of the 26th bit for MODER and PUPDR. The overall configuration, however, remains identical in that MODER is an input and PUPDR has no value for pull up or pull down.

[Configuration of PC1 pin]

void configure\_PC1\_pin(){

RCC -> AHB2ENR |= RCC\_AHB2ENR\_GPIOCEN;

RCC -> AHB2ENR |= 0x00000001;

GPIOC -> MODER &= ~(3UL<<2);

GPIOC -> PUPDR &= ~(3<<2);

}

For the EXTI configuration, most of the code serves the same function, however we are using PC.1 now instead of PC.13. We are also asked to trigger the LED on the falling edge rather than allowing us to choose either the falling edge or rising edge. The clock enable remains identical. We then configure the SYSCFG EXTICR to map GPIO PC.1 (the port that will receive the square wave function from PB505). We use [0] instead of [3] because we are interested in the first 0-indexed hex value, and use EXTI1 instead of EXTI13 because we are interested in PC.1 instead of PC.13. We specify that we want the program to trigger on the falling edge of the square wave by setting EXTI->FTSR1\_FT1 to high and EXTI->RTSR1 to low. When enabling interrupts it should be noted that either the rising or falling edge needs to have a value of ‘1’ to function, if both the rising and falling edge are set to a value of 1 then the interrupt will trigger on both edges and effectively double the frequency with which the LED is toggled. We then enable EXTI1 by setting the 1st bit in EXTI\_IMRl to 1, which allows an interrupt to be generated. This tells the program to not ignore EXTI1 (the interrupt). We then enable interrupt EXTI1\_IRQn on NVIC via NVIC\_EnableIRQ.

[Configuration of EXTI]

void configure\_EXTI(){

RCC->APB2ENR |= RCC\_APB2ENR\_SYSCFGEN;

SYSCFG->EXTICR[0] |= SYSCFG\_EXTICR1\_EXTI1\_PC;

// rising edge = 0

EXTI->RTSR1 &= ~EXTI\_RTSR1\_RT1;

// falling edge = 1

EXTI->FTSR1 |= EXTI\_FTSR1\_FT1;

EXTI->IMR1 |= EXTI\_IMR1\_IM1;

NVIC\_EnableIRQ(EXTI1\_IRQn);

}

To reconfigure the IRQHandler, we change the method name from EXTI15\_10\_IRQHandler to EXTI1\_IRQHandler. First we evaluate the new IMR flag to check for triggering conditions on pending bit 1. We no longer have a need for a while loop because we are not waiting for any condition to alternate states and reset interrupt flags. In this interrupt we simply want to toggle the state of the LED and then immediately reset the flag to resume normal program operation until the next occurrence of a falling edge. This means that for every two periods of the input signal, we will receive one period of the LED signal (because we are only interested in the falling edge).

[Configuration of EXTI1\_IRQHandler]

void EXTI1\_IRQHandler(void){

if ((EXTI->PR1 & EXTI\_PR1\_PIF1) == EXTI\_PR1\_PIF1) {

GPIOA -> ODR ^= 1UL<<5;

EXTI->PR1 |= EXTI\_PR1\_PIF1;

}

}

Our main while loop remains the same by initializing the new configuration values and maintaining an infinite while loop to continuously operate.

[Configuration of main method]

int main(void){

configure\_LED\_pin();

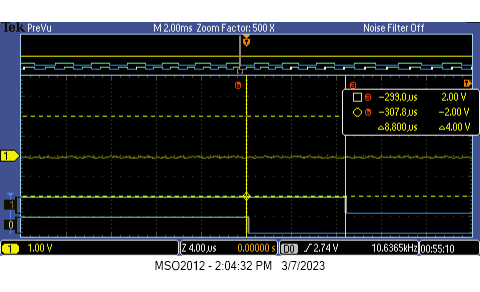
configure\_PC1\_pin();

configure\_EXTI();

while(1);

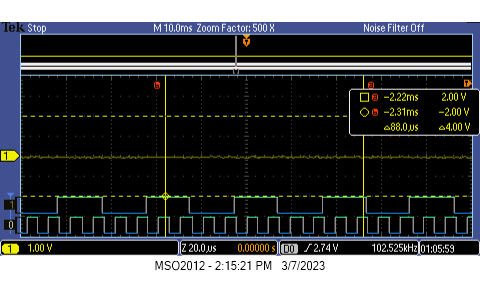
}

Our first task when analyzing how interrupts interact with a generated square wave was to measure the inherent latency present when using this type of input detection. Although we found that the present latency changed between experiments, the latency at 10MHz as requested was ~8.8us.

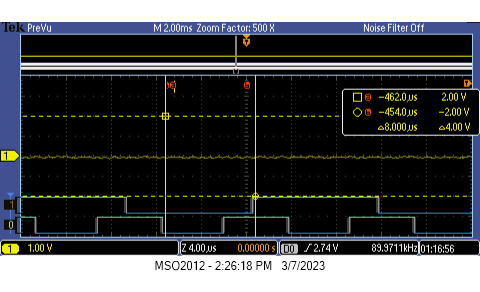
[a latency of 8.8us is present with a frequency of 10KHz]

We can see that the program stops functioning accurately when the input frequency becomes too fast for the interrupt to properly read the current signal; we have concluded that this happens at ~100KHz. In the proper operation of the program each toggle of input channel one will begin on the falling edge of channel zero and remain unchanged until channel zero has a subsequent falling edge. As can be seen below there are multiple falling edges between each toggle action displayed by channel one.

[ Program operation with a 100KHz input square wave ]



The highest frequency that allowed for proper operation of the program was concluded to be ~90KHz. As displayed by the logic analyzer below, we can see that the latency between the input falling edge and the action taken by the interrupt is becoming so significant that the pattern becomes less apparent. It should also be noted that the latency delay decreased from 8.8us at 10MHz to 8.0us at 90MHz.

[Program operation with a 90KHz input square wave ] 

**Conclusion**

Our goal in this lab was to learn how to configure the selected GPIO pin as the trigger source for an external interrupt. In part 1, we configured the various ports, used the blue pushbutton on PC.13 as input, and used the green LED on PA.5 as output. We then configured the system for the external interrupt configuration register, and we used the SYSCFG EXTICR function to map the external interrupt to PC.13. We then enabled the clock of SYSCFG and the corresponding GPIO port, then configure the GPIO pin as input, then select the active edge that can trigger our external interrupt (the code requires either a rising edge, falling edge, or both in order to work). We then enabled EXTI13’s IMR to allow the interrupt to execute. We then enabled EXTI13’s interrupt with EXTI15\_10\_IRQn. This was a successful strategy to monitor the input (push-button) and display the output state of the LED upon input, bypassing the need to constantly check to see if a condition had changed.

In part 2, we used the XOR bitwise operator to toggle our LED whenever an input square wave signal encountered a falling edge. We kept the same LED configuration but used PC.1 instead of PC.13 to allow for the analog input of the PB505’s square wave. We then configured the program to be able to capture the input square wave and the output signal sent to the GPIO output pin. With the help of our logic analyzer, we were able to observe that with a frequency of 10KHz, there is a latency of 8.8us. We ran frequencies up to 90KHz and found that that is the highest frequency that allows the program to function correctly. At values of >90KHz, we have observed that the program stops functioning correctly due to the input frequency being greater than the inherent latency that exists when executing the interrupt code.