Microprocessor Systems

ELE 271

Laboratory 4:

Pushbutton and GPIO Input

Due 09MAR2023

By: Nick Bean , Toussaint Falangani , [Taylor Brookes](mailto:taylor.brookes@maine.edu)

**Introduction**

The purpose of this lab is to learn how to utilize the general purpose input/output (GPIO) pins for digital input and output. In our previous lab, we used the GPIO pins for digital output only. GPIO pins are programmable input/output pins on microcontrollers that can be used for a variety of tasks such as sensing switches or buttons, controlling LEDs, or driving motors. Digital input and output involves reading the state of a switch or button and using that information to trigger a specific action or behavior in the system. Our designed program will be able to read the state of our switch and output data.

* **Part 1** of our lab is where we will configure our pins (PA.5 [output] and PC.13 [input]) to have the LED on by default and to then turn the LED off when the blue pushbutton is pressed. Afterwards, we toggle the LED to the opposite state of what it currently is by pressing the pushbutton.
* **Part 2** blinks the LED by default and turns the LED off when the blue pushbutton is pressed.
* **Part 3** counts how long the pushbutton is pressed, and then turns the LED on for however long the button was pressed for.
* **Part 4** counts the frequency of an input square wave from a function generator by taking advantage of PA.1 as an input pin and reading whether the pin is high or low, then counts how long it takes for one period to pass, then terminates the program.

**Part 1**

The goal of part 1 is to use the blue pushbutton (PC.13) to turn the green LED (PA.5) on and off. We are to configure GPIO PA.5 pin as output and GPIO PC.13 pin as input. Pressing the pushbutton should turn off the LED. Releasing the button should turn it back on.

The following code was previously used in the last lab (lab 3), so the discussion for this portion will be short, as it has already been discussed in detail. We first enable the clock to GPIO Port A, then set GPIO Port A (where the green LED is located (PA.5)) to output mode by first setting bits 10 and 11 to 0 and 0 then to 0 and 1 using bitwise operators. We then set GPIO Port A to push-pull, set GPIO speed to low, then set GPIO Port A to no pull-up or pull-down. This concludes the configuration of pin PA.5 as output.

void configure\_LED\_pin () {

// Enable the clock to GPIO Port A

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

RCC -> AHB2ENR |= 0x00000001;

// Set the GPIO Port A mode to output (01)

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

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

// Set the GPIO Port A output type to push - pull

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

// Set the GPIO speed to low

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

// Set GPIO Port A Push - Pull to no pull - up, no pull - down

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

}

The following code configures the blue pushbutton (PC.13) for input. First, we enable the GPIO Port C clock, which allows us to use Port C. Then, we set the GPIOC mode to input, which allows us to use the blue pushbutton as an input source for our data. Notice that through the reading, we learned that PUPDR’s pull-up or pull-down is only used with input pins, not output pins. In our case, we are setting the push-pull as no pull-up or pull-down. To better understand what this signifies, we have to understand what pull-up and pull-down does. Pull-up has the default value as 1, where pull-down has the default value as 0. These default values will be used when there is no input to the circuit. No pull-up or pull-down does not set a default value, and is referenced as high-impedance, floating, or analog. This means that the output will always be whatever the state of the input is. The default state for the pushbutton, however, is 1 when not pressed.

void configure\_Button\_pin (){

// Enable the clock to GPIO Port C

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

RCC -> AHB2ENR |= 0x00000001;

// GPIOC Mode: Input(00)

//sets bits 26 and 27 to 0 and 0

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

// GPIOC Push-Pull: No pull-up, no pull-down (00)

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

}

Turning the LED on and off is as simple as setting the ODR bit as high or low. We set it high (on\_LED) by using the bitwise OR operator to ensure that the bit is 1, and we set it low (off\_LED) by using the bitwise AND operator with the complementary operator to ensure that the bit is 0.

void on\_LED(){

GPIOA -> ODR |= 1<<5;

}

void off\_LED(){

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

}

In the main while loop, we check the GPIOC (port C) IDR (input data register) at bit 13 (PC.13) to see if the bit is high (the blue pushbutton is pressed) or low (the blue push-button is not pressed). If the button is pressed, we turn off the LED. Otherwise (when the button is not pressed), we make sure the LED is on. Knowing how the pushbutton works helps us to understand why this is the Case. When the value of IDR is 1, the button is not pressed. When the value of IDR is 0, the button is pressed, and the program executes the else statement.

while(1){

if(GPIOC -> IDR & (1<<13)) {

on\_LED();

} else {

off\_LED();

}

}

To toggle the LED, we first check that the pushbutton has been pressed by seeing if IDR == 0x0000. This is the same as the bit shift operation ~(1<<13), but in hex. The pushbutton is 0x0000 when being pressed and 0x2000 while not being pressed. When the button is pressed, we go to our toggle\_LED() method, which uses the XOR operator to switch the LED to the opposite of whatever state ODR (the LED, the output pin) currently is. We then wait until the pushbutton has finished being pressed with the while loop inside of the toggle\_LED() method. This is a necessary part of the program because it ensures that the LED doesn’t randomly switch between on and off. Without this, the method would just return back to the while loop and execute itself repeatedly until the button has stopped being pressed, which would result in a random outcome of the LED’s final state.

Our toggle\_LED() method is as follows:

void toggle\_LED(){

GPIOA -> ODR ^= (1<<5);

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

}

And our new main while loop is the following :

while(1){

if(GPIOC -> IDR == 0x0000) {

toggle\_LED();

}

}

**Part 2**

The goal of part 2 is to use the pushbutton to blink the LED or have it turned off depending on the state of the pushbutton. When the pushbutton is released the LED blinks, and when pressed the LED is off. If we recall from part 1, we can reuse the off\_LED() method when the button is pressed. We also reused the blink\_LED() method from the previous lab (lab 3), which blinks the LED while the pushbutton is not being pressed (when the pushbutton is 1). The following while loop is used inside the previously written main method in this lab, while the blink\_LED() method is outside of the main method and referenced within the while loop.

while(1){

if(GPIOC -> IDR & (1<<13)) {

blink\_LED();

} else {

off\_LED();

}

}

void blink\_LED() {

int i;

for (i = 0; i < 125000 / 2; i++);

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

}

**Part 3**

In part 3, our objective is to use a counter to capture the length of time the pushbutton is pressed. Once the pushbutton is released, we are to turn on the green LED for exactly the same period of time that it was pressed for. To accomplish this, we modified the main method to include a counter variable. By default, the LED is off. Within our if statement, if the button is not being pressed then the if clause will run. First we check to see if the counter variable is greater than zero, which tells us if the button has been previously pressed. If the button has been previously pressed, we define an increment variable “i”, turn on the LED, then count the increment variable until we reach the counter variable’s number. When the increment variable reaches the counter variable’s number, we reset the counter variable and then exit the if statement and turn the LED back off. Within our while loop, we have a statement that increments the counter variable for however long the button is pressed by using the increment (++) operator. One problem with our code, however, is that the counter variable will only start counting again after it has finished counting up to the counter variable. This means that if you press the pushbutton while the LED is on, the system will not put into memory how long the press is until it has turned the LED back off. A better method would include more memory to allow for this case.

int counter = 0;

while(1){

if(GPIOC -> IDR & (1<<13)) {

if (counter > 0) {

int i;

on\_LED();

for (i = 0; i < counter; i++);

counter = 0;

}

off\_LED();

} else {

counter++;

}

}

**Part 4**

In part 4, we are asked to use our program to determine the period of square waves by using the function generator as an input and record count values for multiple frequencies. To do this, we have to configure pin PC.1 as an input. We accomplish this by modifying the configure\_BUTTON\_pin() method by changing “26” to “2” as follows:

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

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

For the main function, we first want to make sure that we start at the rising edge of the square wave. To do this, we first check if we are currently in the high value of the square wave. For simplicity’s sake, the while loops will be referred to as While1, While2, While3, and While4. If we are at a high value of the square wave, we jump into While1, where we wait until we change to the low value of the square wave. If we started at the low value of the square wave, we skip over While1 and jump straight into While2. When we change to a new high value of the square wave, we move to While3. This ensures that we start our counting at the beginning of the square wave. As an indicator, we turn the LED on to signify that we are in the high part of the square wave. We add to the counter value pTime until we exit While3 when the square wave changes to its low value. We then execute While4, turn the LED off, and continue to increment pTime. When the square wave goes high again, we exit our while loop and complete the main method, terminating the program’s run state. This should give us the exact period of the square wave by counting both how long the high value lasts and how long the low value lasts, and ensures that we start at the beginning of the square wave’s period.

while (i){

while (GPIOC -> IDR & (1<<1);

while ((GPIOC -> IDR & (1<<1)) == 0x00);

while (GPIOC -> IDR & (1<<1)){

on\_LED();

pTime++;

}

while ((GPIOC -> IDR & (1<<1)) == 0x00){

off\_LED();

pTime++;

}

i = 0;

}

Our recorded values using the function generator of pTime are as follows:

* 10HZ = 0x2C22 = 11298 ≈ 12,500
* 100HZ = 0x4DB = 1243 ≈ 1,250
* 1kHz = 0x73 = 115 ≈ 125
* 10kHz = 0x5B = 91 ≈ 125 \*Explained below
* 100kHz = 0xA = 10 ≈ 12.5 \*Explained below

We do not receive exactly the values that we would expect to be getting, especially as pTime becomes smaller and smaller, making it more and more difficult for our main method to count exactly how long the period of the square wave is. Each time the program is run we receive a different value for each frequency we choose, but they all appear to be approximately equal to the values we should expect for how long the period should be. We know what these values should be from our previous lab 3 which tells us that counting to 125,000 in decimal or 0x1E848 in hex should take exactly 1 second, which should be 1HZ on the PB505. 10HZ should give us 12,500 in decimal or 0x30D4 in hex, and so on for the remaining frequencies, but that is not the case. For the very high frequencies, we sometimes do not even have a recorded pTime value, due to the square wave switching from high to low too quickly for us to record it. We suspected that this might be an issue with how we configured SPEEDR to low, but when we changed SPEEDR to high we still achieved the same results. As a result, we are unsure as to why this error occurs.

**Conclusion**

Our goal in this lab was to learn how to use the GPIO pins for digital input, to learn how to use the input to change the state of an LED, to count how long an LED has been on or off for, and to count the period of square waves. In part 1 we used the green LED on PA.5 as output and the blue pushbutton on PC.13 as input. We configured the program to have the LED on as default and turn off when the pushbutton is pressed. We accomplished this by checking the state of the input pin, then using an if-else statement to turn the LED on (with an OR bitwise operator equal to 1) and off (with an AND bitwise operator equal to the complementary of 1) contained in separate methods. Next, we configured the LED to turn on and off by the use of the pushbutton and the XOR operator, changing whatever state the LED was in to the opposite state when the toggle\_LED() method is run. In part 2, we configured the program to have the LED blink on and off as default and turn off when the pushbutton is pressed. We accomplished this by reviewing our code from the previous lab and using the XOR bitwise operator set equal to itself XOR 1 to blink the LED, much like the toggle\_LED() method. In part 3, we configured the program to capture the length of time the pushbutton is pressed. We accomplished this by using a counter variable “i” to count how long the button is pressed, then turn the LED on once the button is released and count up to the counter variable to keep the LED on for however long the push button was pressed, then turn off the LED. In part 4, we used while loops to ensure we started counting at the beginning of the PB505’s input square wave, then counted until the square wave started again, giving us the period of the square wave. The consistency in our results was affected by the inconsistency of the PB505’s output signal and when measuring higher frequencies, we suspect that this is due to latency within the code.