

# Microprocessor Systems Lab 2 Report

Weiheng Zhuang Ziyu Zhu

## Table of Contents

<b>Task 1: GPIO Interrupt</b>	<b>3</b>
Introduction	3
Low Level Description	4
Register Implementation	4
HAL Implementation	4
Results and Analysis	5
<b>Task 2: Simple Timer: Register Implementation</b>	<b>7</b>
Introduction	7
Low Level Description	8
Results and Analysis	9
<b>Task 3: Simple Timer: HAL Implementation</b>	<b>10</b>
Introduction and High-Level Description	10
Low Level Description	10
Results and Analysis	11
<b>Task 4: [Depth] Number Entry</b>	<b>12</b>
Introduction and High-Level Description	12
Low Level Description	13
Results and Analysis	16
<b>Conclusions</b>	<b>17</b>
GPIO interrupts	17
Timer interrupts	17
<b>Appendix</b>	<b>18</b>

# Task 1: GPIO Interrupt

## Introduction

This task is designed to write a program that runs on STM32F769NI that outputs the text whenever a pushbutton or an input high that is generated by the Analog Discovery digital line . This uses basic knowledge on GPIO Interrupt, and corresponding register and HAL functions.

An interrupt within a microcontroller is a request for a service from a process running directly outside the control program, a background process. Which means the process is not a subprogram or other subprogram called by the program at some point in its execution. For example, keyboard input is not controlled by the user's program, but by the key being pressed. In real-time applications, interrupts are often used to perform periodic functions such as sampling data or updating errors input to a servo loop. Other functions are to measure the elapsed time from a reference start time or to measure the time interval between external events. In this lab exercise, time-based interrupts and external interrupts are used.

The EXTI mainly supports interrupt functionality for external lines, but also some other sources<sup>1</sup>. 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.

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.

In this task, the main goal is to print the message on the terminal whenever external interrupt detects the input. The program is done by following steps: 1. initialize the EXTI and GPIO. 2. When the interrupt is triggered, clear the flag and set up a high signal. 3. When the signal is high, print the message, and setback the signal to low. The flowchart of this task is attached in the Appendix.[1]

---

<sup>1</sup> "RM0410-stm32f7 Reference Manual." [Online]. Available: [https://sites.ecse.rpi.edu/courses/static/ECSE-4790/Documents/RM0410-stm32f7\\_Reference\\_Manual.pdf](https://sites.ecse.rpi.edu/courses/static/ECSE-4790/Documents/RM0410-stm32f7_Reference_Manual.pdf).

# Low Level Description

## Register Implementation

For the Register Implementation, the function uses D5 as input, which corresponds to EXTI8. To initialize the GPIO and EXTI, the first thing to do is to turn on the RCC for the port and, which is discussed in Lab 1.

Then, the SYSCONFIG RCC module should be enabled to write EXTI registers.

```
RCC -> APB2ENR |= RCC_APB2ENR_SYSCFGEN;
```

Use bit masking to open the needed pin in EXTICR. It should be noticed that EXTICR is built with uint\_8 array, so the third register is actually 2 when it starts counting from 0.

```
SYSCFG -> EXTICR[2] |= 0x00000002; // make EXTI8 to PC8 -> D5 port
```

To enable an interrupt vector in the NVIC, the NVIC\_ISERx registers are used. To open the vector, the code needs to use the IRQn(interrupt vector position number). For the EXTI8, its IRQn is EXTI9\_5, which is 23.

```
NVIC -> ISER[23/32] = (uint32_t) 1 << (23%32);
```

## HAL Implementation

For the HAL Implementation, the function uses user pushbutton(A0) as input, which corresponds to EXTI0. HAL is similar to register in the general case, but it functions in a different coding method. Unlike bit masking for the register, HAL uses its own functions to reach the goal.

Initialize GPIO is the same as Lab 1, so there would be not much discussion. The difference is the Mode is both GPIO\_MODE\_INPUT and GPIO\_MODE\_IT\_RISING\_FALLING, so they are linked by OR operation. Another notice is that the register is pulldown.

For the EXTI IRQ handler function, HAL uses a certain function to call the corresponding GPIO pin for its EXTI, by the following code.

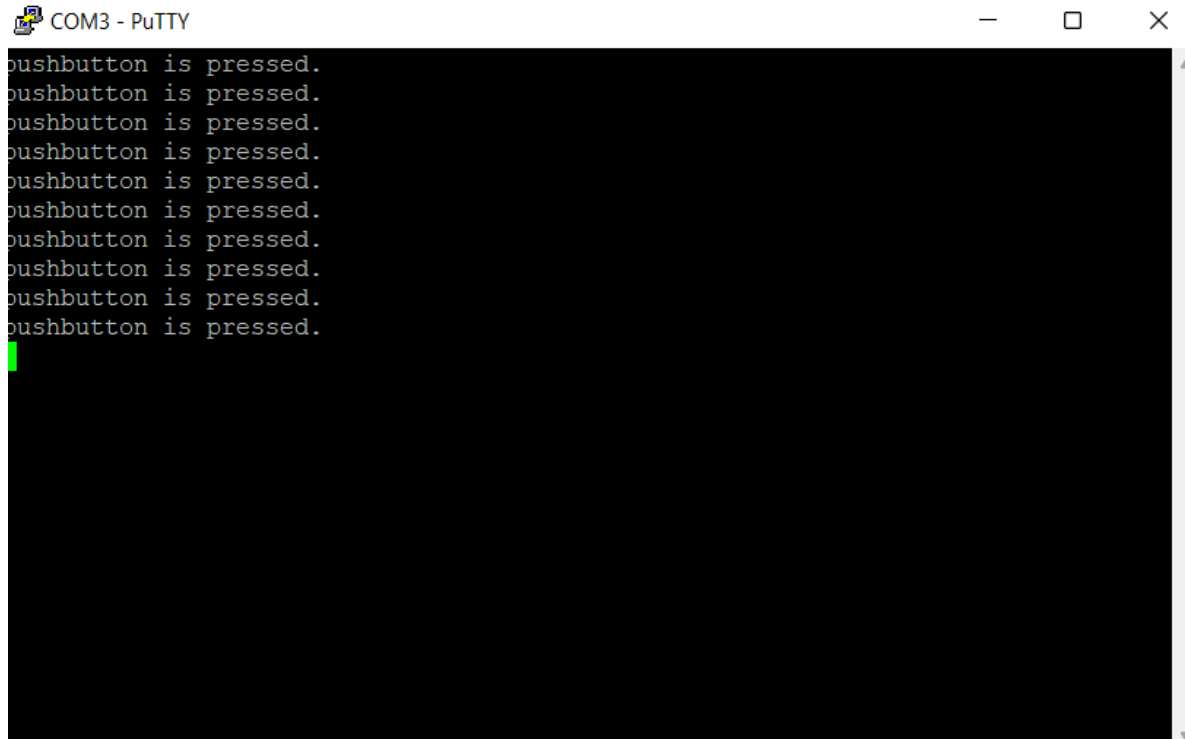
```
HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
```

After the IRQ handler is triggered, the callback functions will be called. It should clear the interrupt flag for the pin and set up a new signal. The function of clearing flag is followed.

```
__HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_0);
```

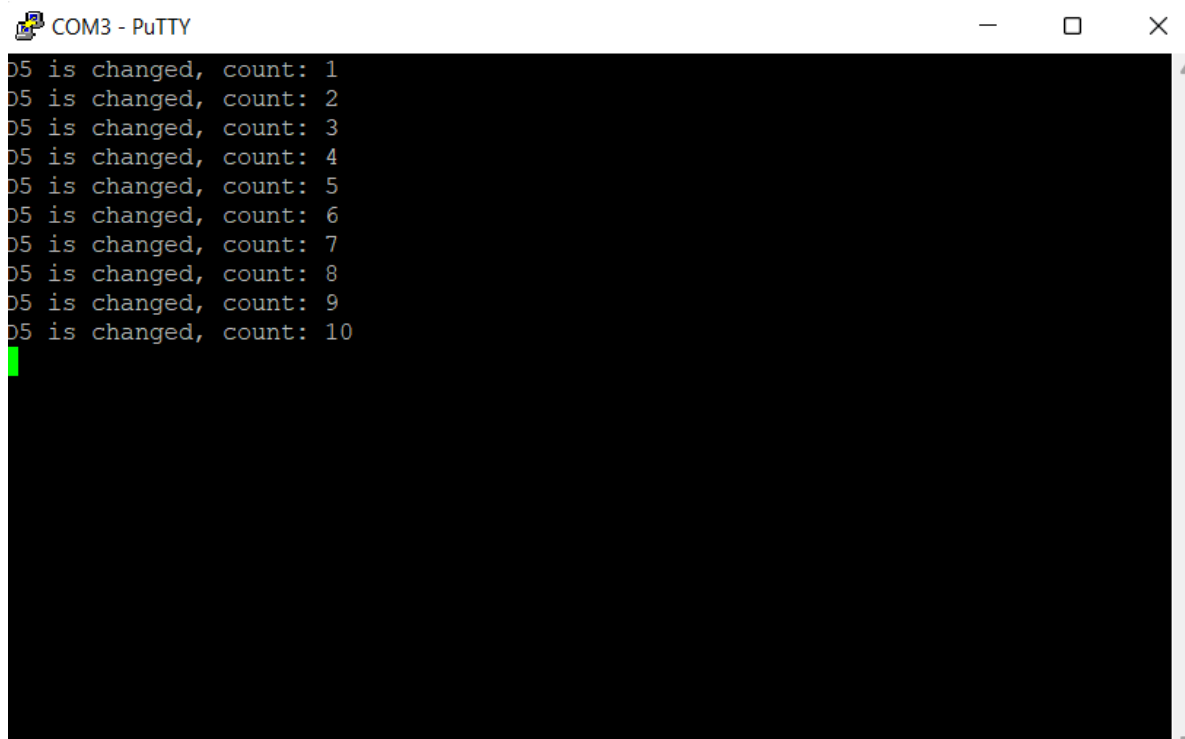
## Results and Analysis

The program functions as expected. When the pushbutton is pressed or released, a message would be printed.



```
COM3 - PuTTY
pushbutton is pressed.
pushbutton is pressed.
pushbutton is pressed.
pushbutton is pressed.
pushbutton is pressed.
pushbutton is pressed.
pushbutton is pressed.
pushbutton is pressed.
pushbutton is pressed.
pushbutton is pressed.
█
```

Figure 1. Task 1 GPIO Interrupt HAL



```
COM3 - PuTTY
05 is changed, count: 1
05 is changed, count: 2
05 is changed, count: 3
05 is changed, count: 4
05 is changed, count: 5
05 is changed, count: 6
05 is changed, count: 7
05 is changed, count: 8
05 is changed, count: 9
05 is changed, count: 10
█
```

Figure 2. Task 1 GPIO Interrupt Register

# Task 2: Simple Timer: Register Implementation

## Introduction

The requirements and results of this task are fairly concise. We only need to write a timer interrupt result display as required. What this timer has to do is to record the time every 0.1 seconds. This program is mainly to call the simple clock on our STM32F769NI into practice.

There are nearly 14 clocks that can be programmed on the STM32F769NI. And the simple clocks we're going to use now are the 6th and 7th. Because these two clocks have the most basic features and functions we need for this task. One of the main functions is to provide a clock interrupt, which means that when the timing function we want arrives, we can get the data we want according to the time. For this task, the base clock we used has a size of 16 bits, which is enough to count from 0 to 65536. However, if we want more functions, we need to use other methods to achieve it. There is a register called TIMn\_ARR in the clock, if it is used, its maximum flux will be limited by the size of this register. Each timer has its own interrupt and flags. But they are different from GPIO, all interrupts are taken over by EXTI. All interrupt sources are stored in their corresponding TIMn\_DIER registers.

For the task we are facing, we also need an additional register, TIMn\_PSC. This register needs to be used together with the clock provided to the timer subsystem to fully calculate the time required for each counted tick to occur. The most important equation used in this lab is that the overflow time is equal to the value of the clock prescaling register+1 divided by the clock provided to the timer and multiplied by the value of the auto-reload register+1. The flowchart of this task is attached in the Appendix.[2]

## Low Level Description

Since it is the first time to use the timer, the requirement of the task is basically to call the timer to print the data, and it is very simple to use in practice. Start by writing the Init\_Timer() equation. One of the first things we need to do is to open the interrupt of timer6. This step is actually very simple. In the table of the ISER register of NVIC, we can find that the number of timer6 is 54. Then turn on the clock of timer6

```
NVIC->ISER[54/32] = (uint32_t) 1 << (54%32);  
RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;
```

Then the main thing is to set up PSC and ARR. We know that the product of these two values plus one divided by TIMnCLK is the time we want. It seems simple, but we have to focus on a very important issue, that is overflow. As mentioned earlier, the upper limit of the count of the basic timer is 65536, which means that if we set the value of the PSC to exceed the upper limit of 65536, then our clock will be inaccurate. Knowing that the value of TIMn CLK is 108 MHz, after careful calculation, we set the value of PSC to 10800 - 1, and the value of ARR to 1000 - 1.

```
TIM6->PSC = 10800 - 1;  
TIM6->ARR = 1000 - 1;
```

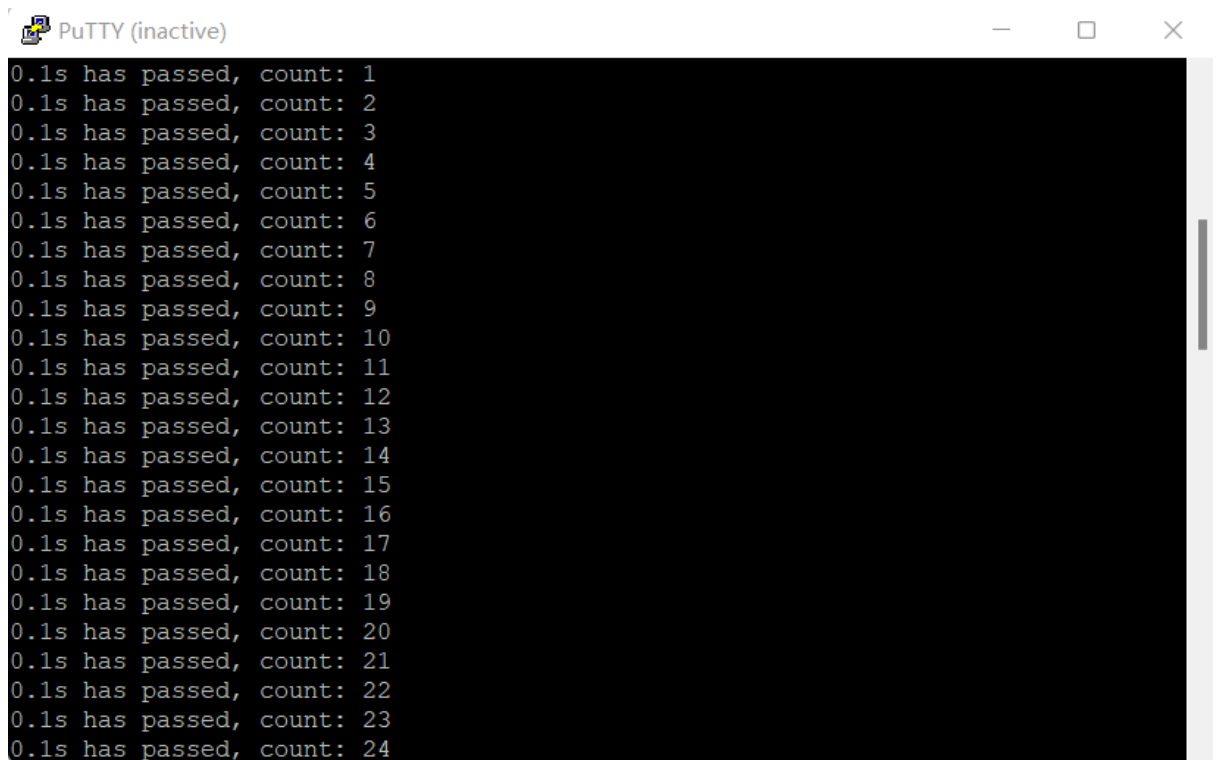
The last is to open some settings and open the timer, they generate update events to auto reload, enable Update Interrupts and start the timer.

Then we need to supplement the equation content of IRQHandler. First, we need to clear the interrupt bit and then use a variable to store the switch. In the main function, when the switch is toggled, we print the output.

```
TIM6->SR &= ~TIM_SR_UIF;  
elapsed = 1;  
printf("0.1s has passed, count: %d\r\n", count); // in the main function
```

## Results and Analysis

The results are as we expected.



```
PuTTY (inactive)
0.1s has passed, count: 1
0.1s has passed, count: 2
0.1s has passed, count: 3
0.1s has passed, count: 4
0.1s has passed, count: 5
0.1s has passed, count: 6
0.1s has passed, count: 7
0.1s has passed, count: 8
0.1s has passed, count: 9
0.1s has passed, count: 10
0.1s has passed, count: 11
0.1s has passed, count: 12
0.1s has passed, count: 13
0.1s has passed, count: 14
0.1s has passed, count: 15
0.1s has passed, count: 16
0.1s has passed, count: 17
0.1s has passed, count: 18
0.1s has passed, count: 19
0.1s has passed, count: 20
0.1s has passed, count: 21
0.1s has passed, count: 22
0.1s has passed, count: 23
0.1s has passed, count: 24
```

Figure 3. Task 2 Simple Timer Register Implementation



# Task 3: Simple Timer: HAL Implementation

## Introduction and High-Level Description

The purpose of this task is the same as Task 2, but using HAL to implement it. The HAL interface for timers is a bit more complex than that for the GPIO.

In order to open the timer, the prescaler and period to set the timer is required, but there can be no auto reloader for the HAL implement. Also, there will be a callback function for the interrupt, so the clearing the flag for the interrupt or setting up other signals can be done in the callback function, but not in the IRQ handler function. The flow of this Task is as same as Task 2, which is Appendix.[2]

## Low Level Description

The initialization of the timer for Task 3 is the same as Task 2. 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. The timer used in Task 3 is different to Task 2, Timer7 is used. After setting the Instance of the timer handle, just set the prescaler and period of the timer, just like in Task 2.

```
htim7.Init.Prescaler = 21599; //108MHz/21.6k = 5kHz  
htim7.Init.Period = 499; //5kHz / 0.5k = 10Hz = 0.1s
```

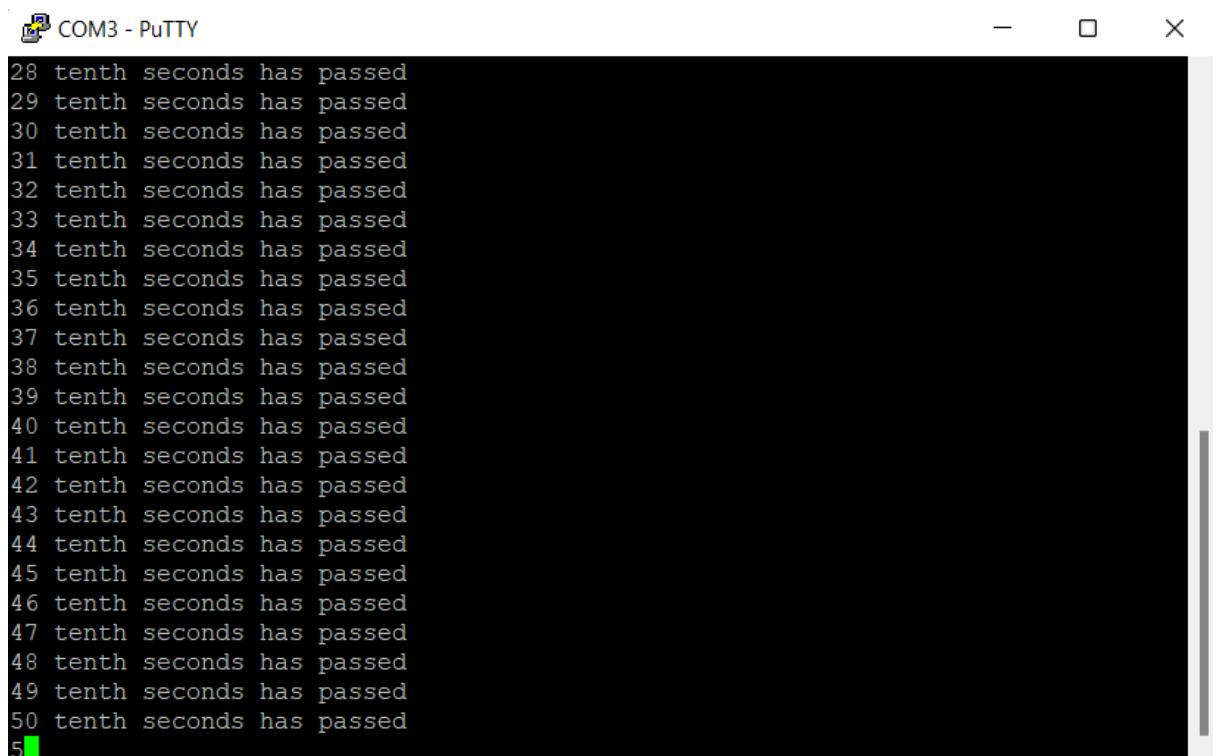
For this lab's application, only base timers are requested and therefore the bulk of the HAL driver may be ignored; that is, only the `HAL_TIM_Base` configuration and control functions will be needed to paired with the `HAL_TIM_IRQHandler()` and the associated callback functions.

In the `IRQHandler` function, like in Task 1, the handler only needs to link the timer to the timer handler that was set before. While in the callback function, the interrupt is triggered by the following code.

```
if (htim->Instance == TIM7)
```

## Results and Analysis

The program would output a message in tenths of seconds as expected.



```
COM3 - PuTTY
28 tenth seconds has passed
29 tenth seconds has passed
30 tenth seconds has passed
31 tenth seconds has passed
32 tenth seconds has passed
33 tenth seconds has passed
34 tenth seconds has passed
35 tenth seconds has passed
36 tenth seconds has passed
37 tenth seconds has passed
38 tenth seconds has passed
39 tenth seconds has passed
40 tenth seconds has passed
41 tenth seconds has passed
42 tenth seconds has passed
43 tenth seconds has passed
44 tenth seconds has passed
45 tenth seconds has passed
46 tenth seconds has passed
47 tenth seconds has passed
48 tenth seconds has passed
49 tenth seconds has passed
50 tenth seconds has passed
5
```

Figure 4: Simple Timer: HAL Implementation

# Task 4: [Depth] Number Entry

## Introduction and High-Level Description

This task 4 needs to be done significantly more difficult than the previous one. This task 4 also needs to complete the design of a small game, which can be divided into many different sub-steps. Structurally, it can be divided into four independent programs. They are: button travel not exceeding 1s at a time, idle time not pressing for more than 1s, key travel exceeding 1s at a time, and keystroke exceeding 3s at a time. The main function of the whole program is that players can input any number of numbers they want through the above four independent programs.

In this, each number is entered through each digit. Input from left to right means that the leftmost is the most significant digit. A keystroke of no more than 1s can increase the value by one digit. That is to say, if you keep entering a key input of no more than 1s at a time, then the value of that digit will keep increasing, but we know that the limit of a single digit should not exceed 9. After finishing the input of a digit, it is necessary to stop the input for more than 1s, then the digit will be saved. After completing all the numerical input of a number, it can complete and save the work through a keystroke of more than 1s at one time. This means that a new digital input loop starts. By continuously looping the first three steps, we can complete the input of many numbers. At the end, you can end the input of all the values and print all the numbers by a keystroke of more than 3s at a time. At this point, the game is over.

Finally, the most critical indicator of this task is that we cannot write any code in the while(1) wireless loop inside the main() equation. All functional code must be integrated inside the interrupt, we cannot write code elsewhere. This is an extremely difficult indicator, which means that we cannot write a linear process to easily accomplish this task. The flowchart of this task is attached in the Appendix.[3]

## Low Level Description

The first thing we can make clear is that all programs depend on the input of keys, so we can use this as a breakthrough for this task, and we can write the code of the theme in the equation `EXTI0_IRQHandler()`. Second, interrupts can be triggered at any time. In the first section, I split all the programs into 4 subroutines. In these four subroutines, there are three key inputs in total, and the difference between the three key inputs is determined by the length of the pressing time. Because interrupt uses edge to determine. Therefore, we should first consider when the `EXTI0_IRQHandler()` equation is called, whether it is a press or a rebound. I set a parameter flag to detect the timing of the trigger and start counting after an odd number of presses.

```
if (!flag) // button pressed.
{
    releasedtime = 0;
    count = 0;
    flag = 1;
    end_time = 0;
}
```

Then we need to go to the `TIM6_DAC_IRQHandler()` equation to make the corresponding settings. First of all, we need to judge that the button has been pressed. This judgment is very simple. As can be seen from the code above, if the button is pressed, the code in `EXTI0_IRQHandler()` will change the value of the flag from 0 to 1. So we only need to add a line of `if (flag)` judgment to the `TIM6_DAC_IRQHandler()` equation, and increase the value of count at the same time.

Next, let's consider the even-numbered case, which is what to do after a rebound. Likewise, the rebound will be called the `EXTI0_IRQHandler()` equation. We can use `else if (flag)` to determine this situation. After the rebound, we say that the flag is cleared and then start the judgment through the value of count. The highest priority is the case of more than 3s, because once the program exceeds 3s, it will end and it is also the longest case, followed by the case of more than 1s, and finally the case of less than 1s.

Judging over 3s is the best written, because it only has two functions. The first is to print out the numbers that are still not printed. The second is to end the program. There is an easy way to do numbers that haven't been printed out yet. If every time you press and hold for 1s, the number will be printed out, then there will be no output after long-pressing for 1s and 3s, so we only need a finish parameter. Set its value to 0 after pressing 1s.

```
if (finish)
{
    printf("number: %d\r\n", number);
    number = 0;
}
printf("program halts\r\n");
while(1); // program halts
```

The situation of long pressing 1s is very simple, it only needs him to print out the saved number, and then set the values of finish and number parameters to 0. The last is the case of the keystroke less than 1s, which is the most complicated. First we will increase the value of the parameter digit that holds the digit. Then the program will make a decision and drop the value back to 0 if the digit exceeds 9. Finally, the program will clear the count after all the checks are done.

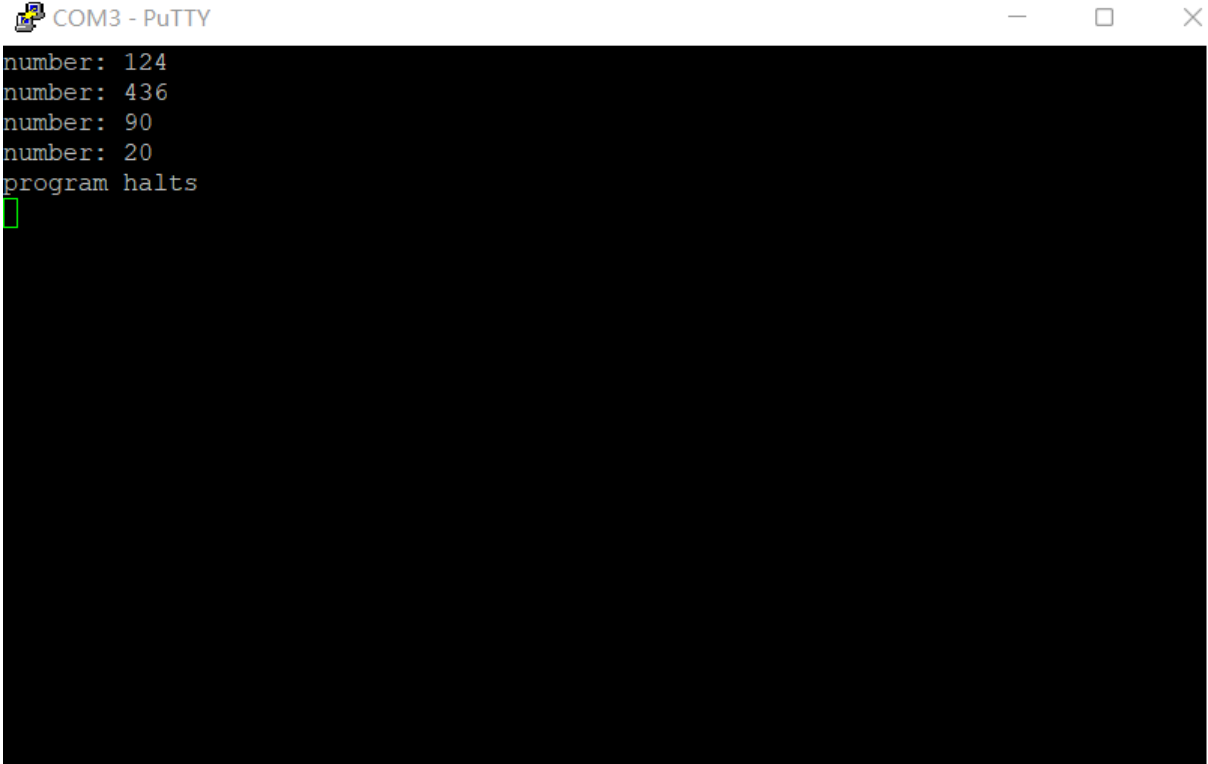
The last is to complete the case of not pressing the button for more than 1s. We will complete this function in TIM6\_DAC\_IRQHandler(). The first is to determine that the button has rebounded. We set the parameter end\_time in the EXTI0\_IRQHandler() equation. Then we increment the parameter *releasedtime* by one. Next, make a judgment of more than 10s. If it exceeds 10s, we will add the current digit to the number. Then clear the digit again.

```
if (releasedtime > 10)
{
    releasedtime = 0; // clear release time counter
    endcheck = 1;
    number = number * 10 + digit; // add the digit to the number
    digit = 0;
}
```

In order to ensure that this judgment will not be triggered all the time, we will add a new parameter endcheck. If the digit has been added to the number, we will turn on this switch, so that the parameter *releasedtime* will not continue to increase.

## Results and Analysis

The results are as we expected.

A screenshot of a PuTTY terminal window. The title bar at the top reads 'COM3 - PuTTY' and includes standard window control buttons (minimize, maximize, close). The terminal area has a black background with white text. The output consists of five lines: 'number: 124', 'number: 436', 'number: 90', 'number: 20', and 'program halts'. A green cursor is positioned at the start of the line following 'program halts'.

```
number: 124
number: 436
number: 90
number: 20
program halts
█
```

Figure 5: Number Entry

# Conclusions

Through this experiment, we learned how to use interrupt as the blocking cable of the peripheral hardware program to interrupt the running program. In the first part we learned how to use the interrupt method of GPIO, and in the second part we learned the time interrupt method using a timer. Here are some discussions of some real-world applications of these topics.

## GPIO interrupts

GPIO interrupts are very common in life. From the beverage buttons of vending machines, the settings of washing machines, the emergency braking of escalators, and even the open interface of home computers, GPIO interrupts are widely used.

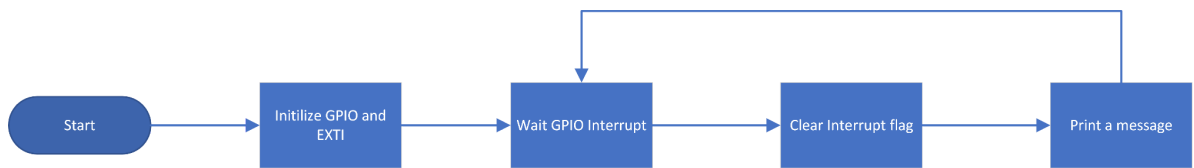
## Timer interrupts

Some common uses for GPIOs are to change the state of hardware devices, control LEDs, and read the state of switches. The interrupt instance of the timer is even more numerous in reality. First of all, most of the functions in our computer have to rely on the timer to solve, such as the visualization of timed shutdown and startup. There are more in daily life, the timing function in the mobile phone, the signal lights in the road, and even the alarm system of some agencies

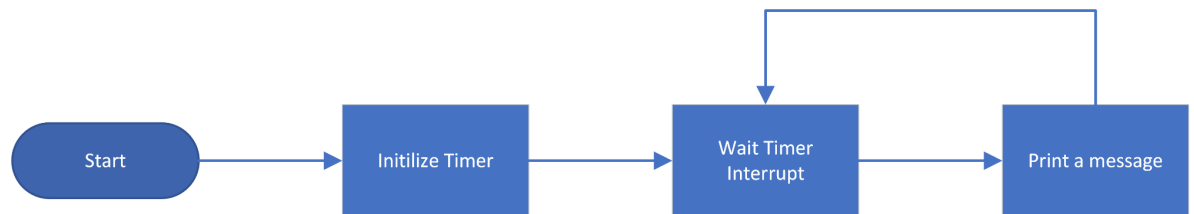


# Appendix

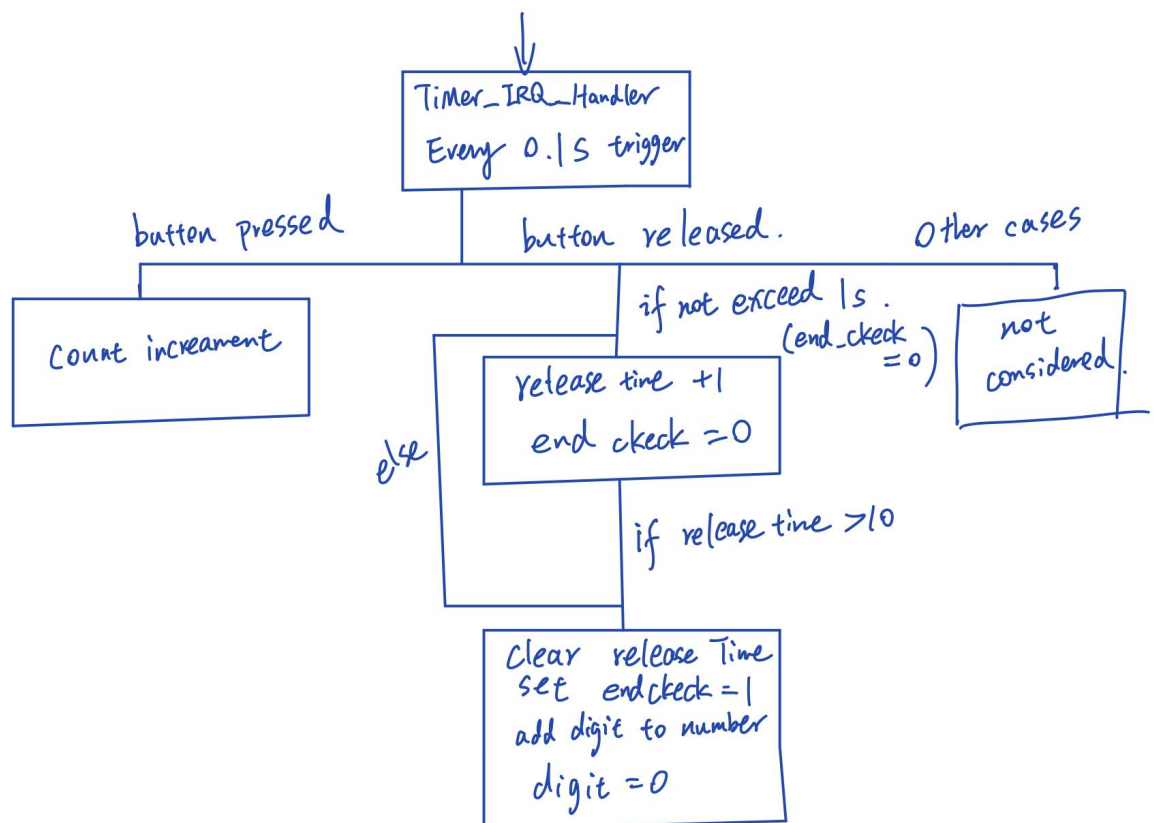
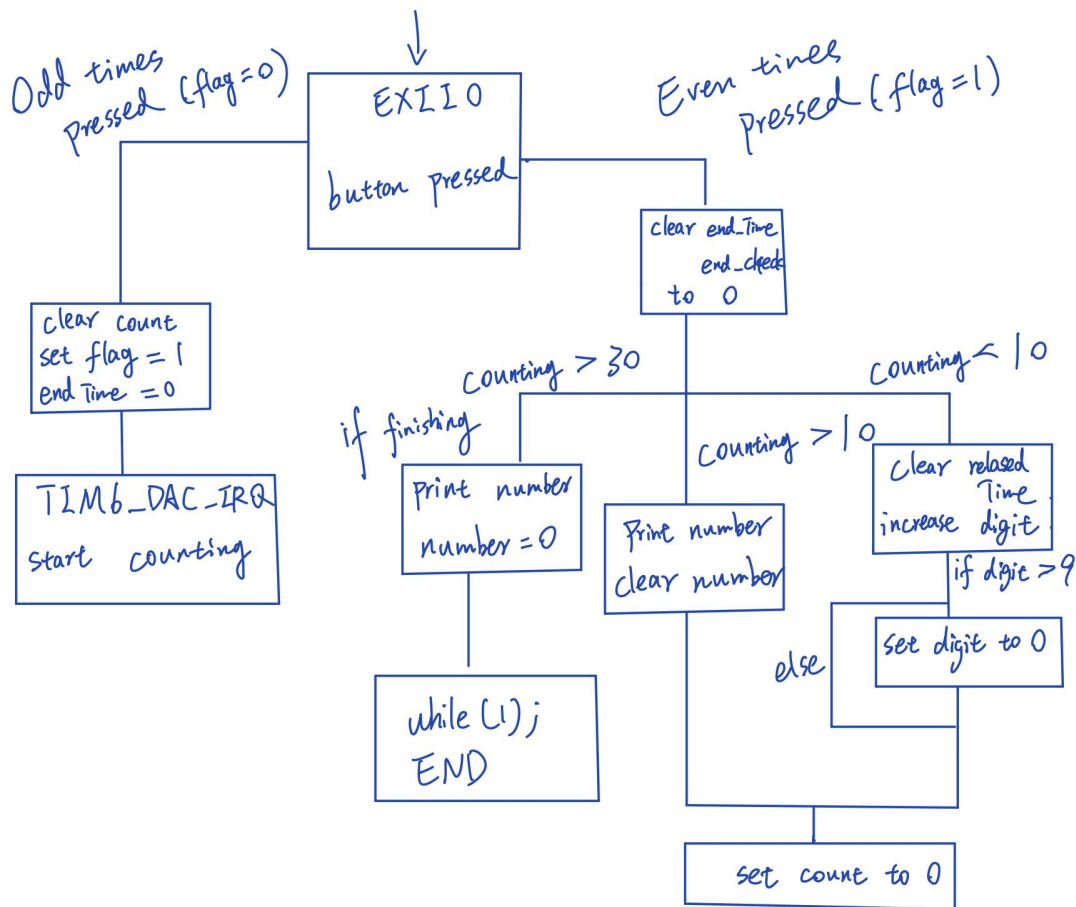
[1] Task 1 flowchart.



[2] Task 2&3 flowchart.



[3] Task 4 flowchart.



[4] Checkoff and Grade Sheet

MPS

Lab02: Interrupts and Timers

## Microprocessor Systems Lab 2

### Checkoff and Grade Sheet

Partner 1 Name: Ziyu Zhu

Partner 2 Name: Weiheng Zhuang

Grade Component	Max.	Points Awarded		TA Init.s	Date
		Partner 1	Partner 2		
Performance Verification: Task 1	10 %	10/10	10/10	kw	9/22
Task 2	15 %	15			
Task 3	15 %	15			
Task 4 [Depth]	10 %	8/9		kw	9/22
Documentation and Appearance	50 %				
Total:					