Embedded systems 3

Assignment 3

Interrupts

Abstract: This assignment is about the research and implementation of interrupts using an ARM Cortex chip on the STM32F303RE board.

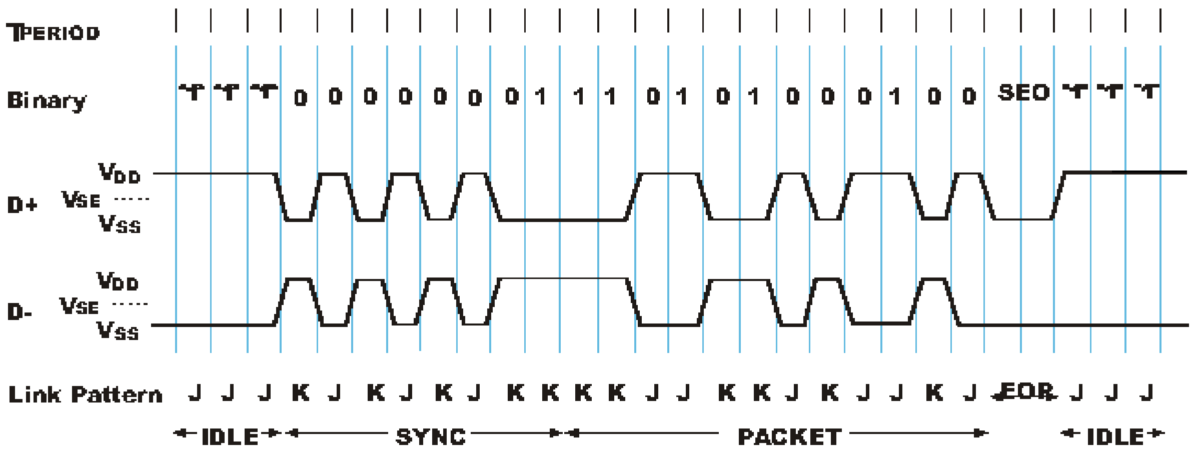
Made by Stefan Andonov 11/09/22

Introduction

In this assignment we are going to make use of two buttons and two LEDs on the STM32 board. We are going to re-use code from the second assignment, as the circuit is similar to what we are using in the current assignment. Concisely, when doing a short press, a button must turn on the corresponding LED. Conversely, when doing a long press on said button, the corresponding LED should turn off. Both buttons function in the same fashion. A further requirement is to exclusively use interruptions and not polling.

Note: A short press is defined as pressing the button for more than 20ms and less than 500ms. A long press is more than 500ms. A press shorter than 20ms should be ***ignored****.*

Polling is the action of constantly checking if a condition has been fulfilled. For example, the way USB keyboards work is they constantly check if there has been a change in the input. This action of constantly checking is called polling.



TPERIOD
Binary
'1' '1' '1' 0 0 0 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 0 0 SEO '1' '1' '1'
D+
VDD
VSE
VSS
D-
VDD
VSE
VSS
Link Pattern J J J K J K J K J K K K K J J K K J K J J K J EOR J J J
IDLE
SYNC
PACKET
IDLE

Fig. 0 The signal of a keyboard under an oscilloscope

Interrupts on the other hand are triggered based on input from the core peripherals – the general-purpose input/output (GPIO) pins. Those are called external interrupts. There is also one more type of interrupts which are based on built-in hardware into the microcontroller. They are called exceptions. In this assignment we are going to be using external interrupts, referred to only as interrupts hereafter in the document.

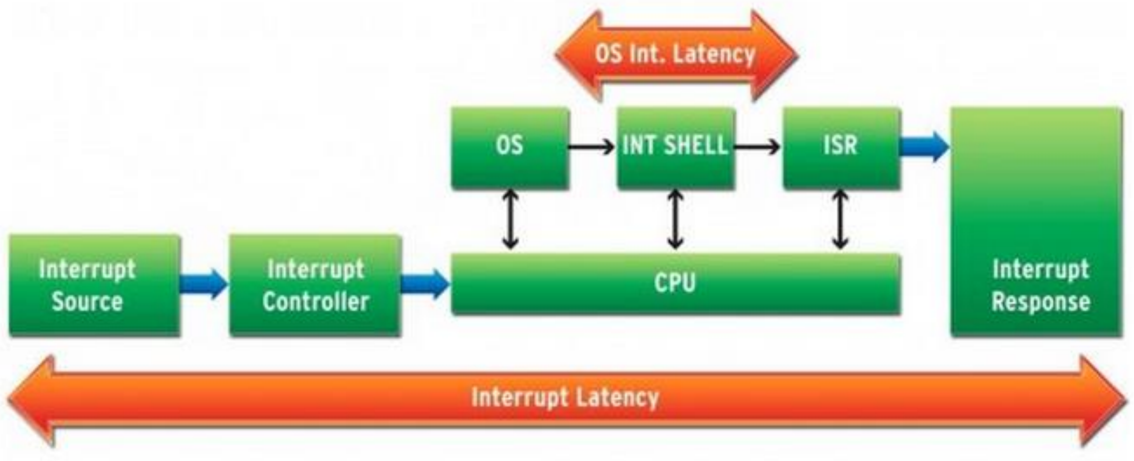


Fig. 1 Interrupt latency in a system

The NVIC (Nested Vectored Interrupt Controller) is the next step after an interrupt request. The interrupt controller stops the main chip from executing whichever task it is executing at the time and tells it to jump to the interrupt handler. The interrupt controller can be thought of as a complex table with multiplexers connecting the input of different pins to the same output. **Only 1 interrupt can be handled at a single time.**

Figure 51. External interrupt/event GPIO mapping

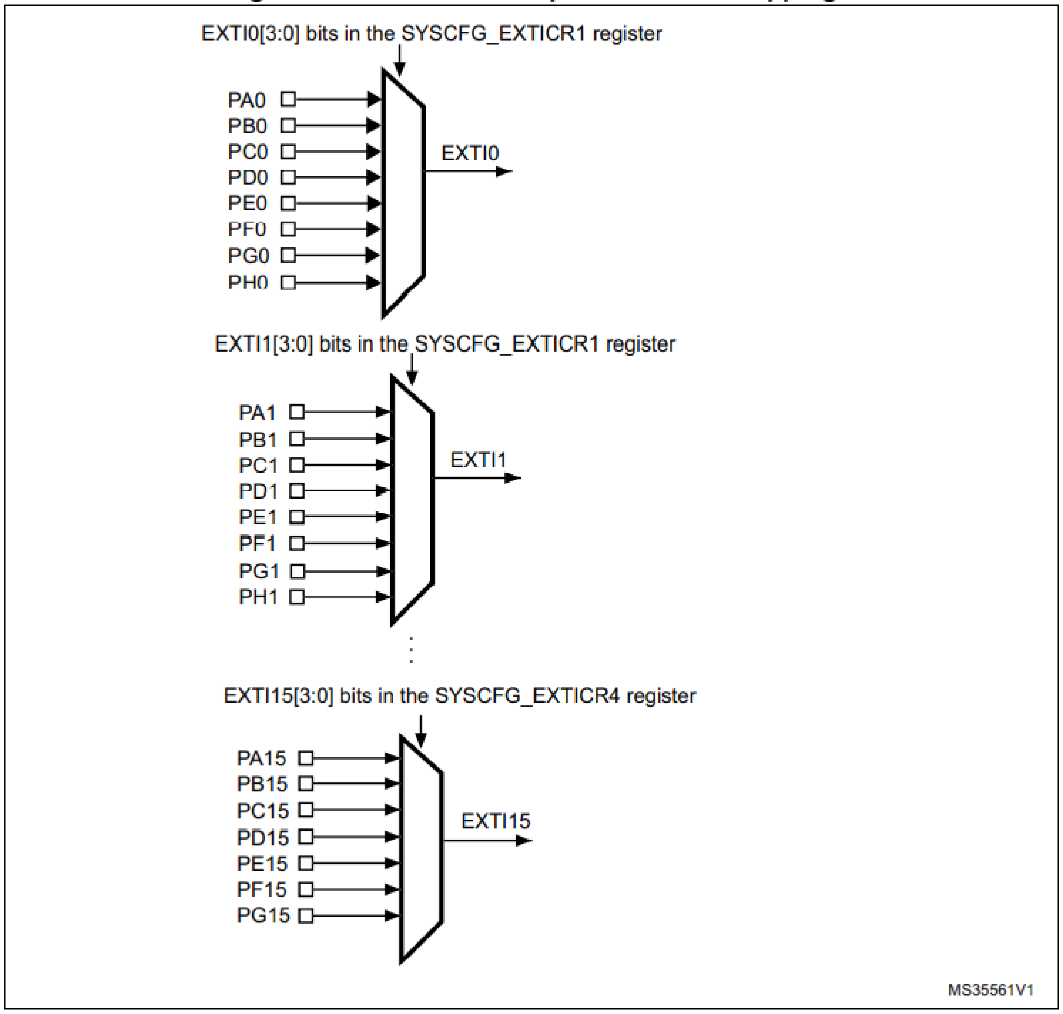

EXTI0[3:0] bits in the SYSCFG_EXTICR1 register
PA0
PB0
PC0
PD0
PE0
PF0
PG0
PH0
EXTI0
EXTI1[3:0] bits in the SYSCFG_EXTICR1 register
PA1
PB1
PC1
PD1
PE1
PF1
PG1
PH1
EXTI1
EXTI15[3:0] bits in the SYSCFG_EXTICR4 register
PA15
PB15
PC15
PD15
PE15
PF15
PG15
EXTI15
MS35561V1

Fig. 2 Correlation between pin number and interrupt lines in the STM32F303RE. /Pg. 295 from the Reference manual/

System design

In this section we are going to build our approach to tackle the problem, create schematics and look at corner cases of our system.

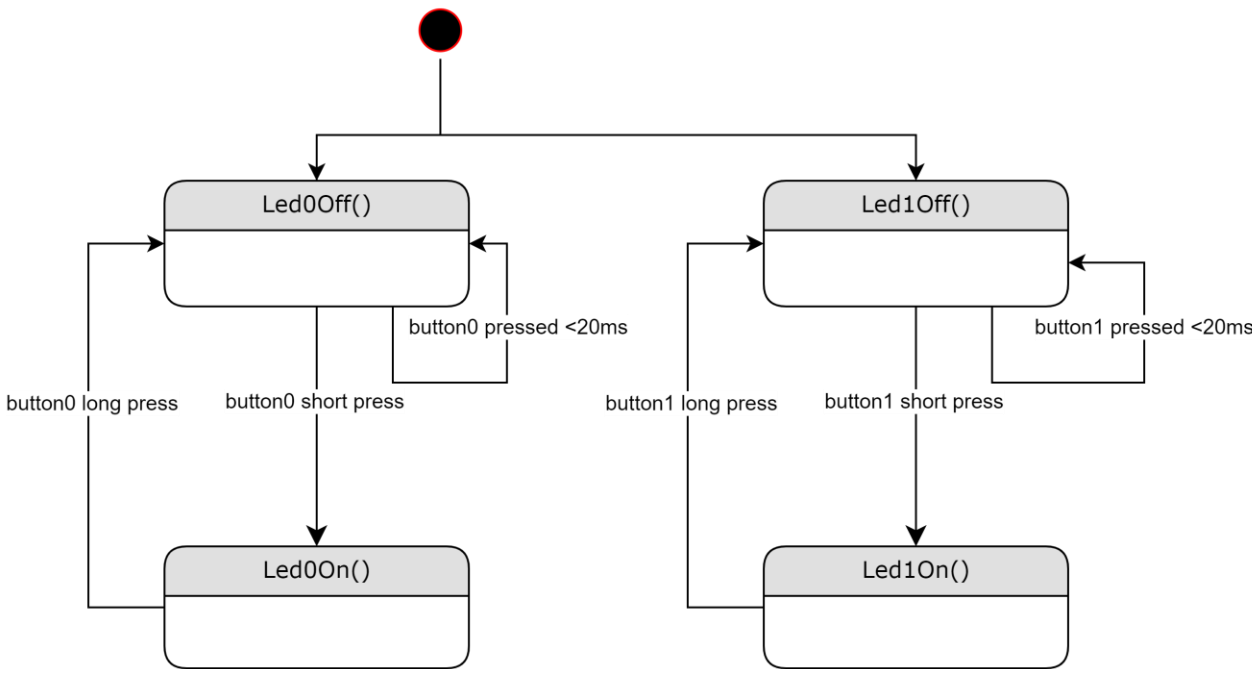


Fig. 2.1 State machined diagram of our system

Button handling

In this part I will explain the reasoning behind the programming techniques used when handling the button. I am going to look at all possible cases that arise from the assignment requirements and consider possible solutions. To achieve this, I am going to also look at the hardware that we are using to build the circuit and examine possible problems tied to physical limitations of the hardware parts.

Case 1: Button pressed for less than 20ms

The job of our system is simple – the input of each button is connected to the output of one respective LED. An LED should only react if the button press is longer than 20ms. A potential point of failure for our system would be not correctly detecting a button press. Firstly, let's investigate how a button works. The button can be thought of as a switch that connects VCC to ground. By default, the switch is open, and only when you physically press the button does the circuit close as shown on Fig 3.

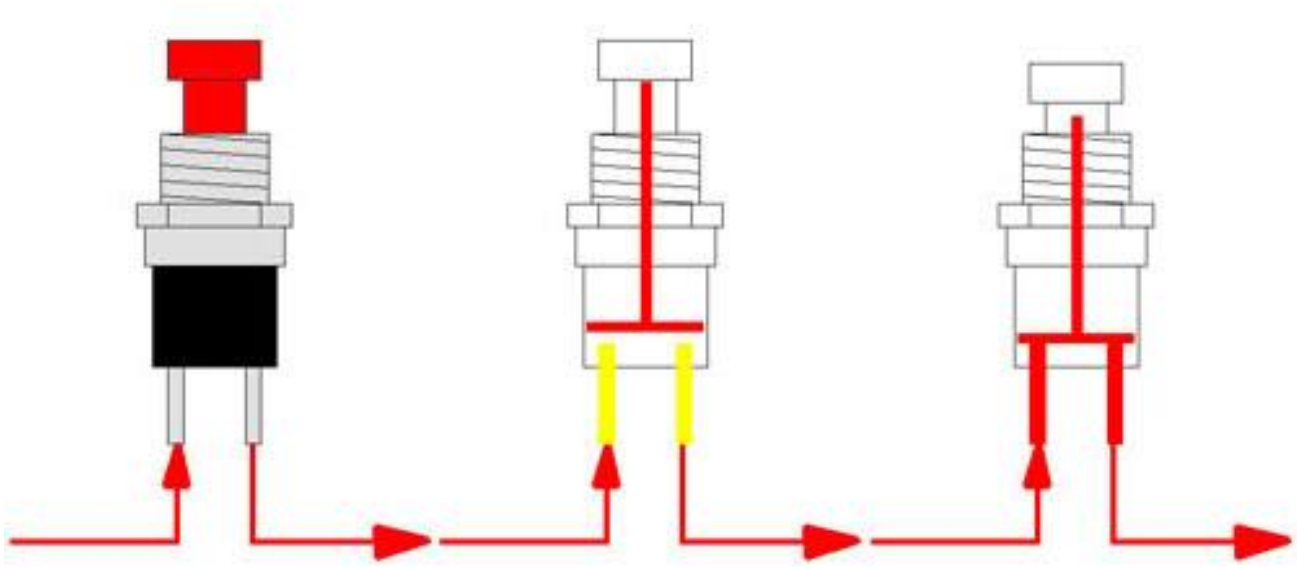


Fig. 3 Mechanical button

Because of mechanical imperfections in each button, there sometimes occurs 'bouncing' of the spring. This bouncing can be seen under an oscilloscope in Fig. 4.

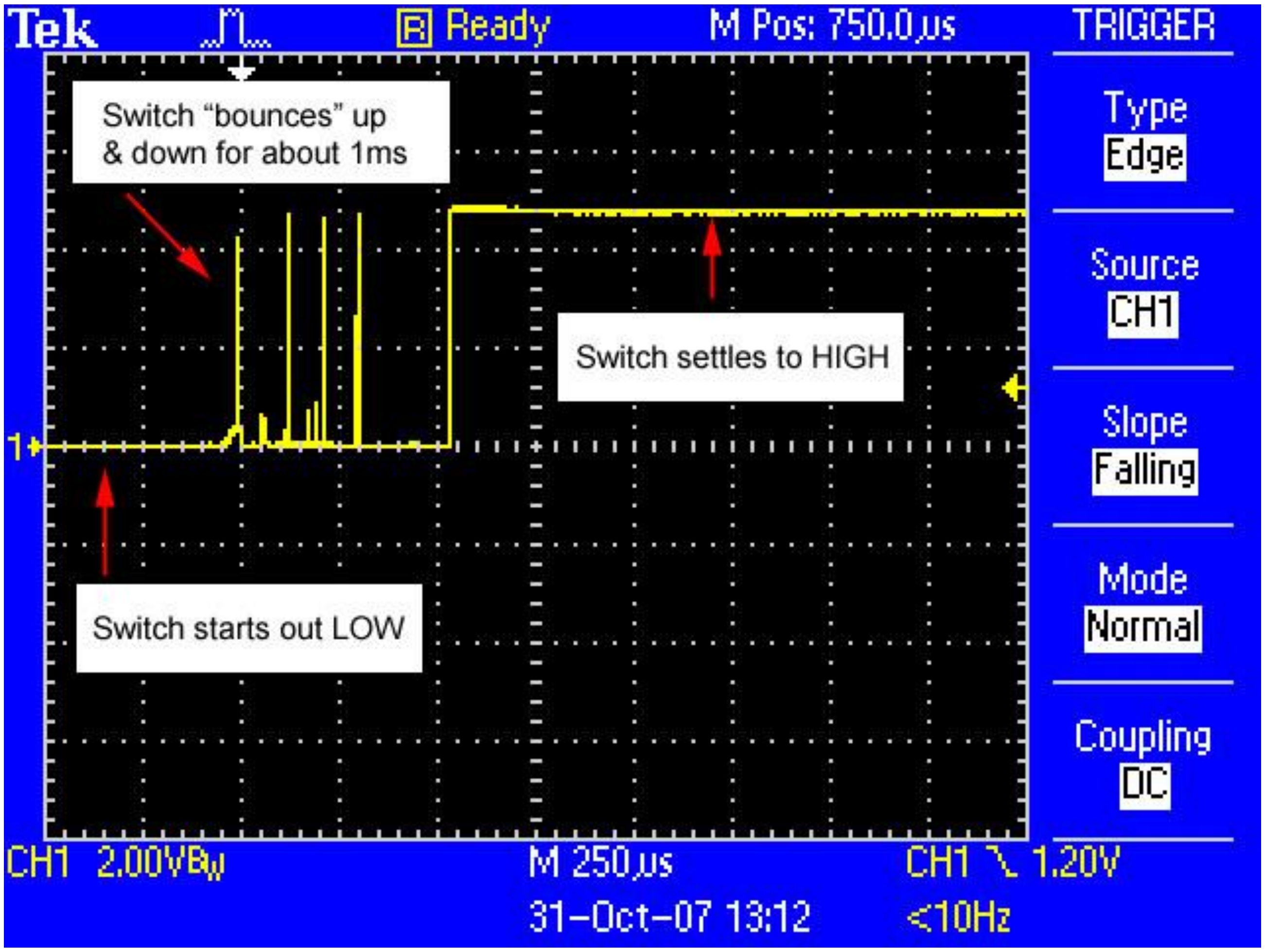


Fig. 4 Button bouncing

In practice, when this happens a single button press is sometimes registered as two separate button presses. To counteract this, we are going to debounce the button in software. For this purpose, we are going to use our own blocking delay using the internal system timer to debounce the button. The reason we are using a blocking delay is due to the nature of interrupts. We are re-using code from Assignment 2 – Systick. Because we are implementing a debounce solution, one of the requirements is already fulfilled – a button press shorter than 20ms will not be registered.

Blocking delay reasoning

Since we're only entering the interrupt handler twice per button press – we must wait for 20 milliseconds. If we use a non-blocking delay the microcontroller will process the entire function in less than 20 milliseconds in all cases. A non-blocking debouncing example could look like:

```
void interruptHandler(void) {
    if (buttonState == HIGH) {
        if (timeElapsed > 20) {
            if (buttonState == HIGH) {
                TurnOnLed();
            …
```

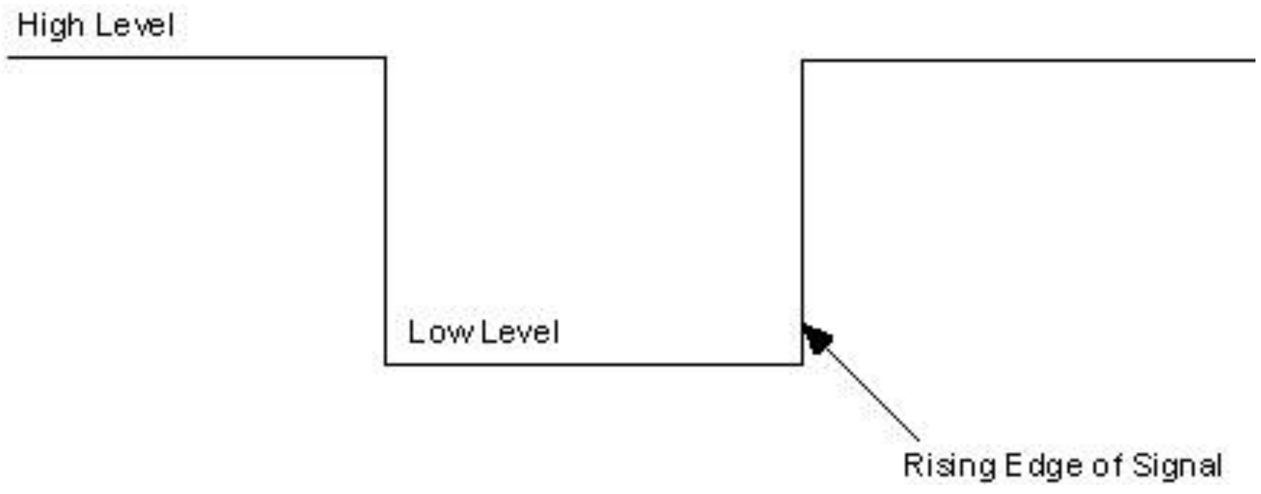
Since we are not **polling**, the code is executed only once, and the non-blocking timer condition always evaluates to false. Therefore, we are going to use a blocking delay instead and debounce the button.

Case 2: Short press

In this part we are going to turn on the LED by pressing the button for a short time (20ms<button press<500ms). The way we are going to implement this is by generating an interrupt upon a button press. We will wait for 20ms (with a blocking delay) and see if the button is still pressed. At that point we know for certain that the button press has actually happened, and we are going to turn on the LED. We are also going to set up the code for the long press by resetting the long-press duration counter.

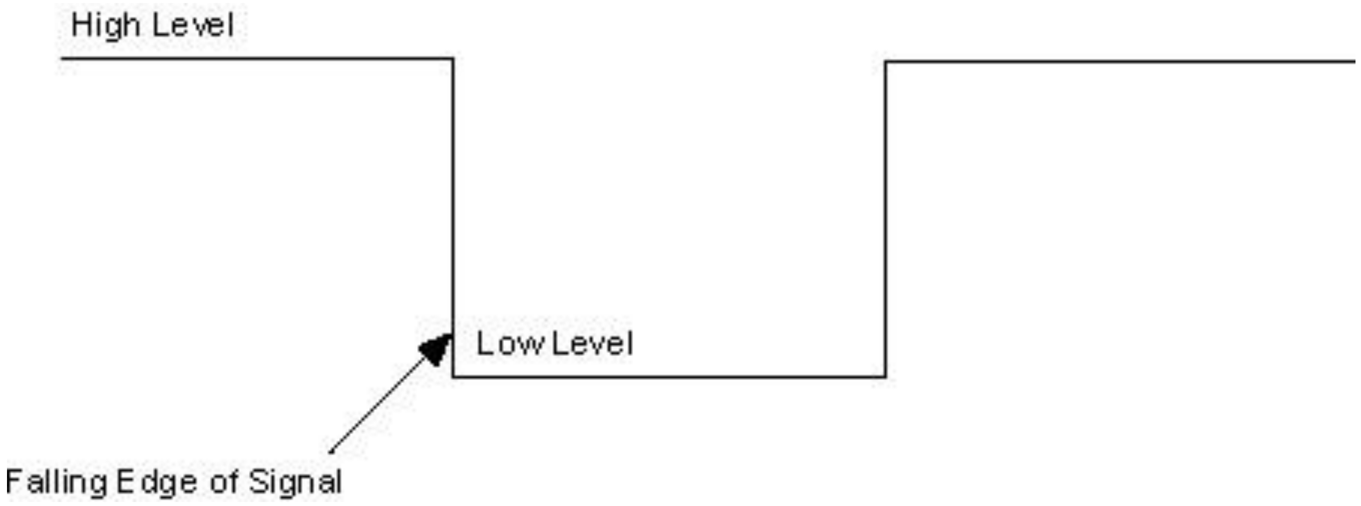
There is a situation where the user turns on the button by holding down the button for a short press, and then keeps holding even after the button has been turned on. By our design, we are going to turn off the LED if the user keeps holding the button.

Since we're using pull-down resistors, a logical 0 means that the button has not been pressed and a logical 1 means that there has been a button press. Following that logic, to detect a button press we need to detect the rising edge in order to toggle the LED.



Case 3: Long press

In this state the LED has been turned on and the user wants to turn it off by holding down the button for longer than 500ms. The way we are going to achieve this is by detecting the falling edge of the button.



Since we're working with the internal pull-down resistor in both external buttons, once the button has been let go of, we are going to observe a falling edge of our signal. Since we're using 1 interrupt handler per pin, we need to program our button so that we detect both rising and falling edge in the same function. The way we're going to achieve this is by using a simple if/else statement. This is the pseudocode for it:

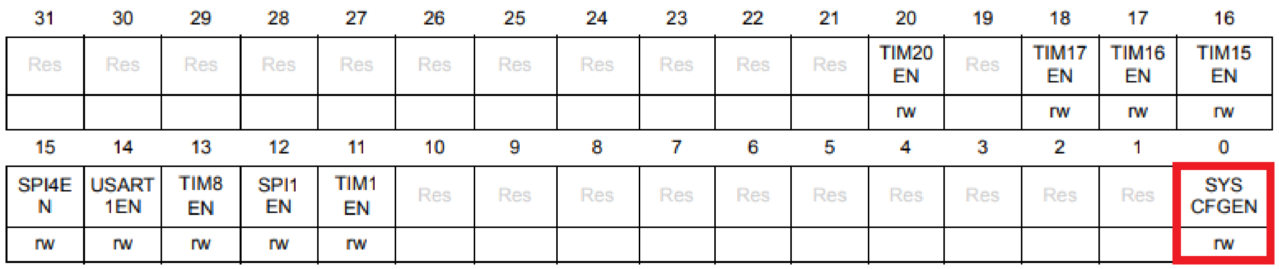
```
if (buttonState == HIGH) {
    TurnOnLed();
}
else if ((buttonState == LOW) & (timePassed > 500)) {
    TurnOffLed();
}
```

Procedure

In this section I shall discuss all the required registers to configure external interrupts on the STM32 board for the required task.

RCC_APB2ENR

The first register we need is the RCC_APB2ENR. We have already used this register in previous assignments, and it enables or disables the peripheral clock. In order to use external interrupts, we need to enable ‘SYSCFGEN’.

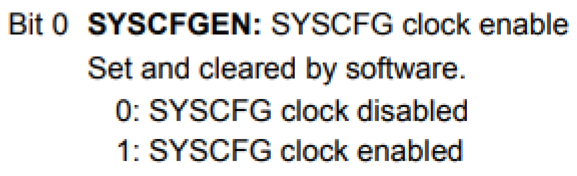


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	TIM20 EN	Res	TIM17 EN	TIM16 EN	TIM15 EN
											rw		rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI4E N	USART 1EN	TIM8 EN	SPI1 EN	TIM1 EN	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	SYS CFGEN
rw	rw	rw	rw	rw											rw

Fig. 5 RCC_APB2ENR register

To enable SYSCFG, we need to write ‘1’ to b0.



Bit 0 **SYSCFGEN:** SYSCFG clock enable

Set and cleared by software.

0: SYSCFG clock disabled

1: SYSCFG clock enabled

SYSCFG_EXTICR1/SYSCFG_EXTICR4

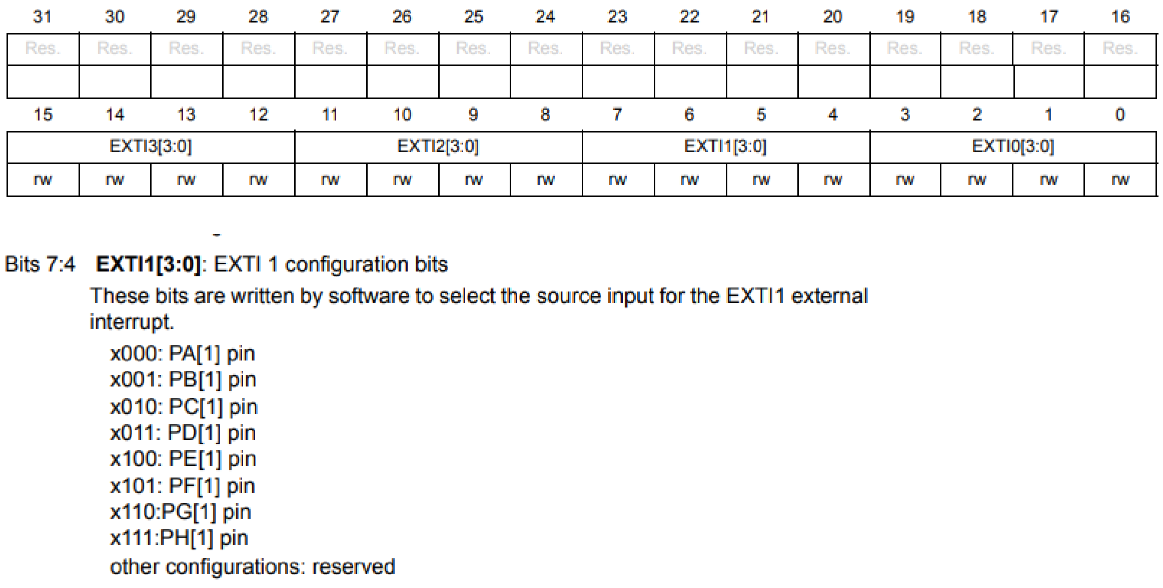
This register is used for configuring different interrupt lanes. In this task we are going to use the internal button B1, mapped on PC13, and an external button connected to pin A1, mapped on PA1.

	Pin	**Port**
Internal button	B1	PC13
External button	A1	PA1

Fig. 6 Button pin numbers and ports

External button

To enable external interrupts for the pins we need to look at Fig. 7



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:4 **EXTI1[3:0]**: EXTI 1 configuration bits

These bits are written by software to select the source input for the EXTI1 external interrupt.

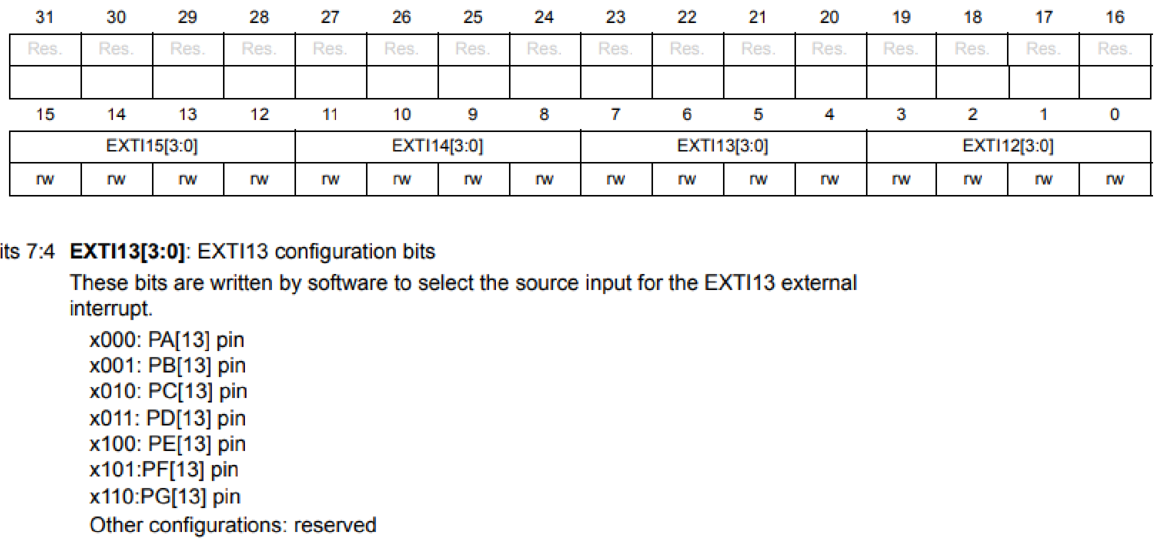
x000: PA[1] pin
x001: PB[1] pin
x010: PC[1] pin
x011: PD[1] pin
x100: PE[1] pin
x101: PF[1] pin
x110:PG[1] pin
x111:PH[1] pin
other configurations: reserved

Fig. 7 SYS_CFG_EXTICR1 interrupt lane bit mapping

Since we're using pin PA1 for the external button, we need to write 0x000, shifted 4 times to the left to the SYS_CFG_EXTICR1 register.

Internal button

To enable internal interrupts for the pins we need to look at Fig. 8



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI15[3:0]				EXTI14[3:0]				EXTI13[3:0]				EXTI12[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:4 **EXTI13[3:0]**: EXTI13 configuration bits

These bits are written by software to select the source input for the EXTI13 external interrupt.

x000: PA[13] pin
x001: PB[13] pin
x010: PC[13] pin
x011: PD[13] pin
x100: PE[13] pin
x101:PF[13] pin
x110:PG[13] pin
Other configurations: reserved

Fig. 8 SYS_CFG_EXTICR4 interrupt lane bit mapping

To enable interrupt requests for the internal button, we are going to use PC13. According to Fig. 8 we need to write 0x010 shifted 4 times to the left.

EXTI_FTSR/EXTI_RTSR

These two registers are for edge detection. We will be using pull-down resistors configured by software. This means that the input value will be a Boolean '0' when there is no input. This is because the resistor and input are connected to ground, and the resistor 'pulls down' the input to a low. When the button is pressed the circuit is closed and the input becomes '1'.

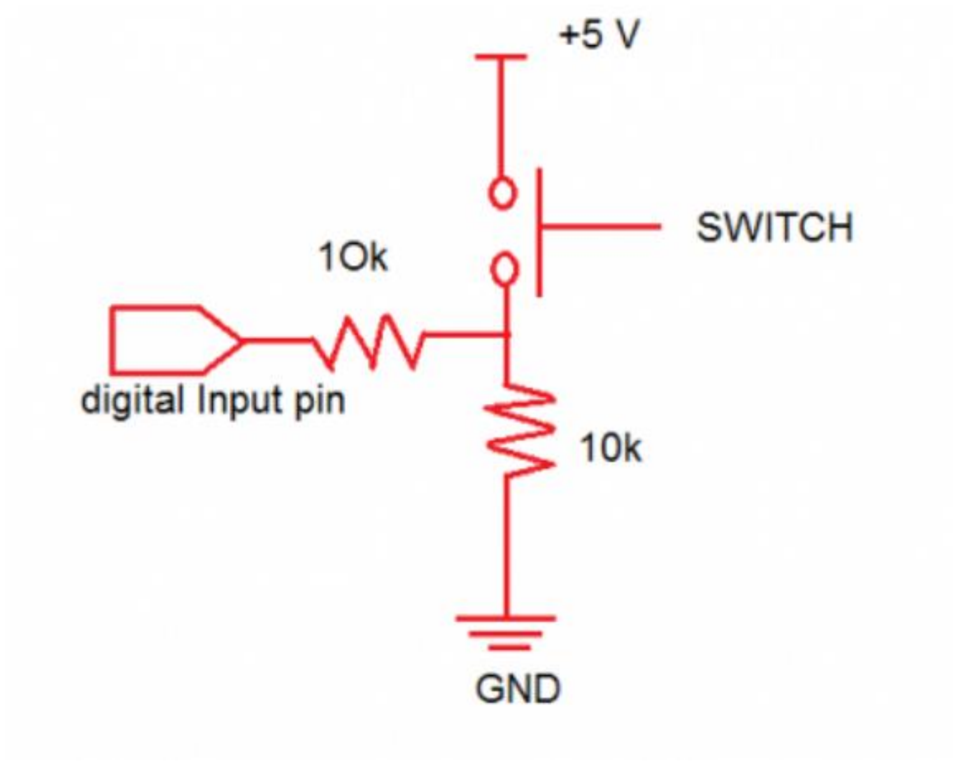
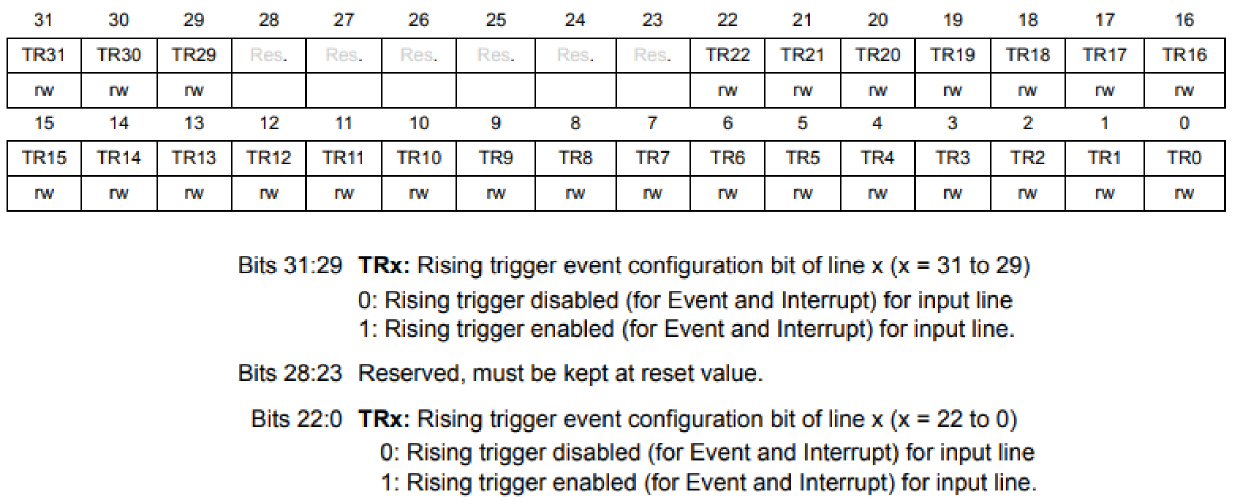


Fig. 9 Schematic of a circuit pull-down resistor

In our task we are going to use a pull-down resistor for both buttons. We will still need to enable both falling and rising edge detection for our interrupts. This is because we want to keep track of when the user presses the button, as well as when they let go of it.



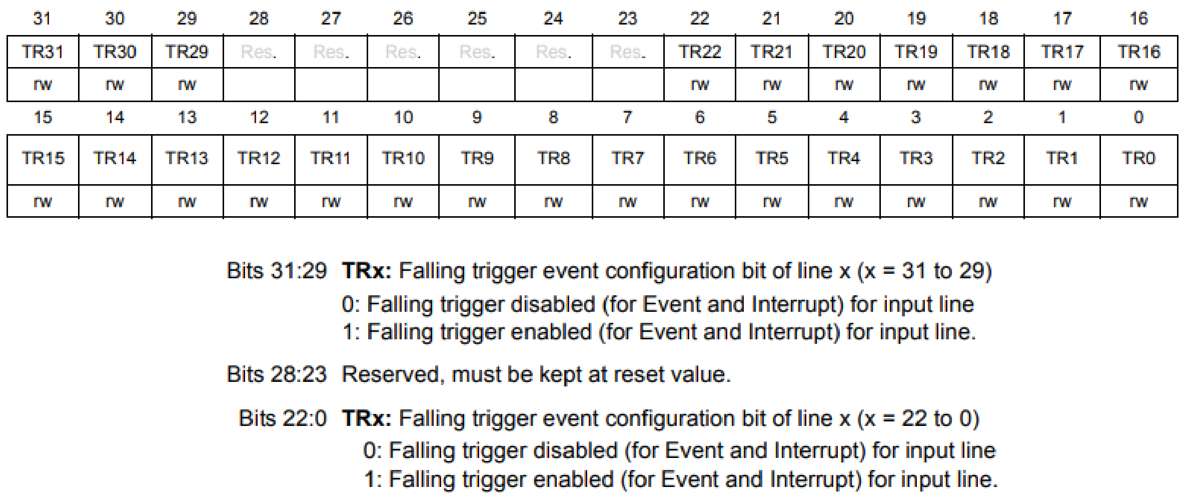
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TR31	TR30	TR29	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	TR20	TR19	TR18	TR17	TR16
rw	rw	rw							rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:29 **TRx:** Rising trigger event configuration bit of line x (x = 31 to 29)
0: Rising trigger disabled (for Event and Interrupt) for input line
1: Rising trigger enabled (for Event and Interrupt) for input line.

Bits 28:23 Reserved, must be kept at reset value.

Bits 22:0 **TRx:** Rising trigger event configuration bit of line x (x = 22 to 0)
0: Rising trigger disabled (for Event and Interrupt) for input line
1: Rising trigger enabled (for Event and Interrupt) for input line.

Fig. 10 Rising detection register



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TR31	TR30	TR29	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	TR20	TR19	TR18	TR17	TR16
rw	rw	rw							rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:29 **TRx:** Falling trigger event configuration bit of line x (x = 31 to 29)
0: Falling trigger disabled (for Event and Interrupt) for input line
1: Falling trigger enabled (for Event and Interrupt) for input line.

Bits 28:23 Reserved, must be kept at reset value.

Bits 22:0 **TRx:** Falling trigger event configuration bit of line x (x = 22 to 0)
0: Falling trigger disabled (for Event and Interrupt) for input line
1: Falling trigger enabled (for Event and Interrupt) for input line.

Fig. 10 Falling detection register

	Mask	**Register**
Enable rising detection	0xD	EXTI_RTSR1
Enable falling detection	0xD	EXTI_FTSR1

*Fig 11. Masks required to enable falling and rising detection for **internal** button*

	Mask	**Register**
Enable rising detection	0x2	EXTI_RTSR1
Enable falling detection	0x2	EXTI_FTSR1

*Fig 12. Masks required to enable falling and rising detection for **external** button*

EXTI_EMR1

This register is for enabling the interrupt handler. It must be enabled, so that a ‘1’ is passed onto the AND gate and the interrupt request is passed onto the NVIC.

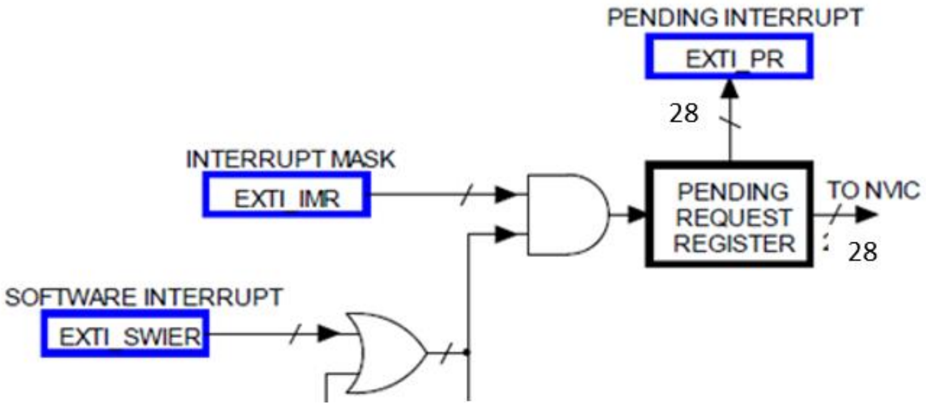
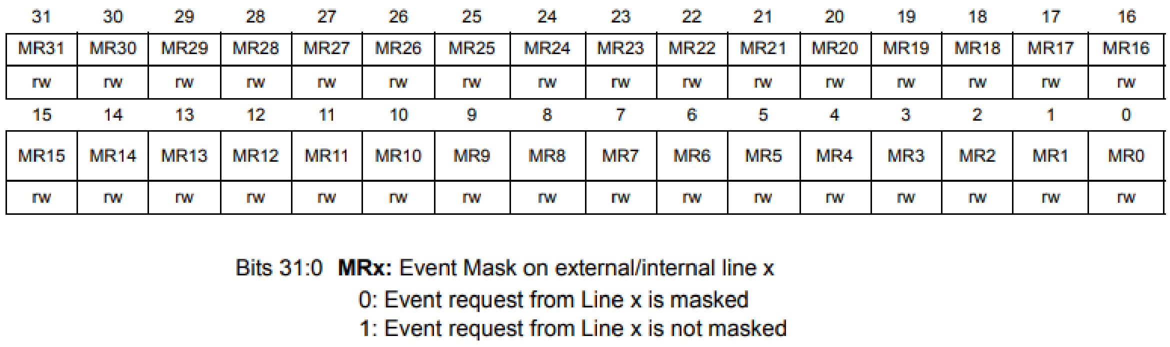


Fig. 13 Interrupt mask allows the interrupt to be passed



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MR31	MR30	MR29	MR28	MR27	MR26	MR25	MR24	MR23	MR22	MR21	MR20	MR19	MR18	MR17	MR16
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **MRx:** Event Mask on external/internal line x
0: Event request from Line x is masked
1: Event request from Line x is not masked

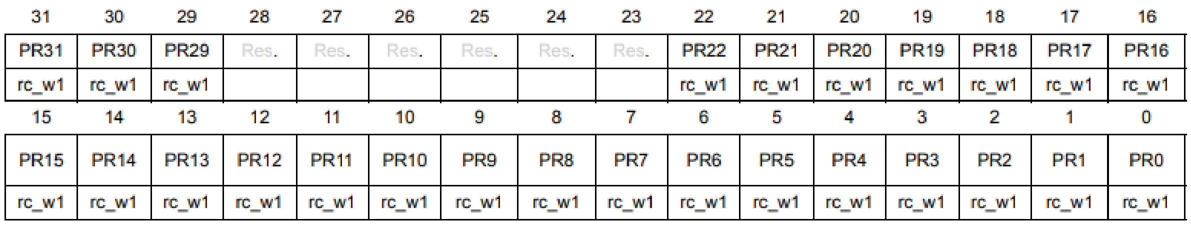
Fig. 14 Event mask register

	Mask	Register
Internal button mask enable	0xD	EXTI_EMR1
External button mask enable	0x2	EXTI_EMR1

Fig. 15 Masks required to enable the internal mask

EXTI_PR1

This is the register for our flag. In the implementation, we are going to reset it immediately after an interrupt request has been generated.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PR31	PR30	PR29	Res.	Res.	Res.	Res.	Res.	Res.	PR22	PR21	PR20	PR19	PR18	PR17	PR16
rc_w1	rc_w1	rc_w1							rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

Fig 16. Pending register

	Mask	Register
Flag reset internal button	0xD	EXTI_EMR1
Flag reset external button	0x2	EXTI_EMR1

Fig. 17 Masks required to reset flag

Conclusion

The importance of interrupts should not be underestimated when programming embedded systems. They are a powerful tool to trigger immediate actions based on important events. Imagine there's an aircraft and the temperature or altitude sensors read critical values – the system should handle the disturbances with immediate attention. In order to make use of interrupts there is additional hardware knowledge required – which makes the learning curve quite steep. For example, there's no way to know how to program a button for falling or rising edge, unless you know what falling and rising edge are. But to know what falling and rising edge are, you also need to know how a switch closes an electric circuit and so on.

All in all, this specific task was handled with care and many small incremental steps – from carefully setting up and testing each I/O pin, to programming and setting up each interrupt register. Mostly everything worked from the first attempt – except for one small mistake interrupts initialization. All the setup for the interrupts was correct, except for the NVIC_EnableIRQ() function. I had chosen the wrong line to enable, which was frustrating as it's a very small and hard to detect mistake.

Other examples of overcoming adversity in this assignment are resetting the external interrupt flag. I noticed that if I reset the flag at the very end of the interrupt handler, I am not able to detect the falling edge, since no interrupt can be triggered until the flag has been reset.

All in all, the execution of the assignment felt very intuitive to me. Even though the task at hand was as simple as toggling an LED, it required a lot of background knowledge, which I highly appreciate.

References

Stm32 Peripheral Drivers from Scartch : GPIO Programming: Interrupts

Interrupts

Lab_02_SysTick

Embedded Systems Programming: Interrupt, Interrupt Latency, and IRQ - 2020

STM32 Interrupts External Interrupts Tutorial NVIC & EXTI ARM Exceptios

Detecting Rising and Falling edges

STM32 rising and falling button interrupt detection