GEBZE TECHNICAL UNIVERSITY

ELECTRONICS ENGINEERING

ELEC 334

Microprocessors Lab

LAB #3

Prepared by
1) 1801022037 Ömer Emre POLAT

1. Introduction

In this lab we will look into exceptions, fault generation and handling, external interrupts and priorities.

2. Problem 1

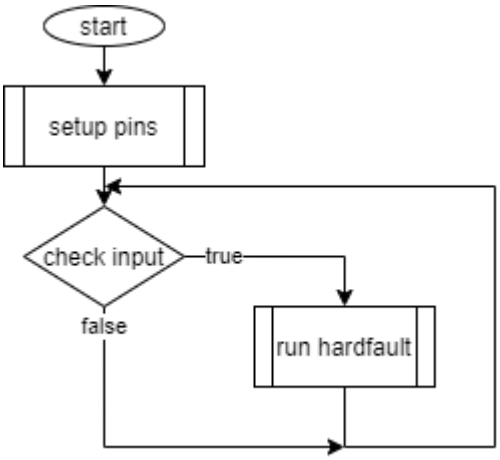
For this problem we first have to look up for the hardfault handler's name from the startup_stm32g031k8tx.s file to override it.

```
190     .weak   HardFault_Handler
191     .thumb_set HardFault_Handler,Default_Handler
```

After finding the name of the weak function we can override it in the program code. Now we have to look for the reset handler's name from the same file.

```
 56   .weak Reset_Handler
 57   .type Reset_Handler, %function
```

Now that both of the weak defined functions are found we can write the first code that will reset the stack pointer and then call the default reset handler.



After the flowchart is created we can write the code.

```c
#include "stm32g0xx.h"

void HardFault_Handler(void)
{
	__ASM("LDR r0, =_estack"); /* resetting the stack pointer */
	__ASM("MOV SP, r0");
	__ASM("B Reset_Handler"); /* calling the reset handler */
}

int main(void)
{
    while(1)
    {
        HardFault_Handler(); /* calling the hardfault handler for testing */
```

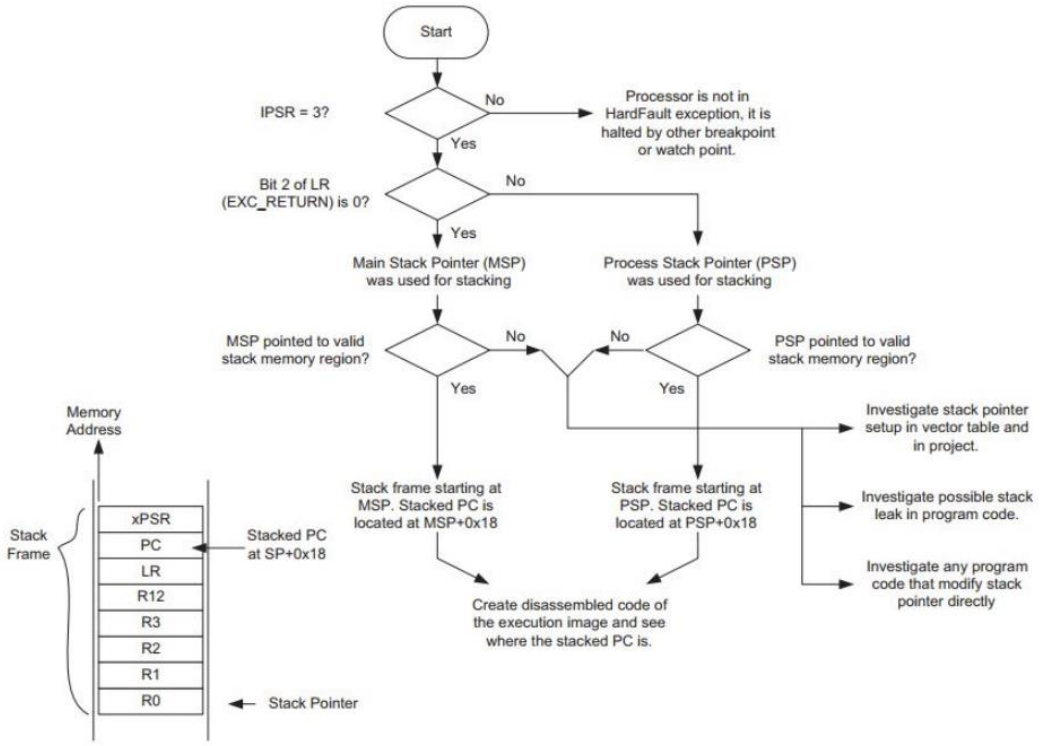
```
    }
    return 0;
}
```

After the code above is ran, we can see that the stack pointer is reset back and the reset handler is called to reset the board.

After running the hardfault handler once the SP is returned back to its starting value and is equal to MSP at the start of the program.

Possible hardfaults may include:

- **Memory Related**
  - An attempt to access an invalid address.
  - Attempt to execute a program from non-executable memory region.
- **Program Errors**
  - Attempting to run an undefined instruction.
  - Switching from Thumb state to ARM state.
  - Trying to access unaligned memory.

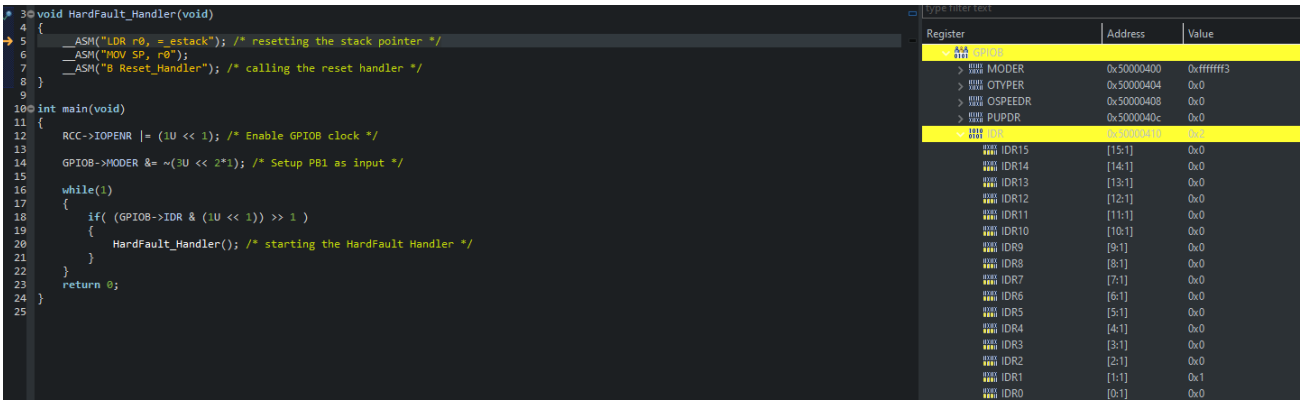


These errors can be identified using the table above.

Now we can add a button to the system and run the HardFault handler according to the button press.

```c
#include "stm32g0xx.h"

void HardFault_Handler(void)
{
	__ASM("LDR r0, =_estack"); /* resetting the stack pointer */
	__ASM("MOV SP, r0");
	__ASM("B Reset_Handler"); /* calling the reset handler */
}

int main(void)
{
	RCC->IOPENR |= (1U << 1); /* Enable GPIOB clock */

	GPIOB->MODER &= ~(3U << 2*1); /* Setup PB1 as input */

    while(1)
    {
      if( (GPIOB->IDR & (1U << 1)) >> 1 )
      {
		HardFault_Handler(); /* starting the HardFault Handler */
      }
    }
    return 0;
}
```

With the button added we can test the board if it jumps to the hardfault handler when the button is pressed.

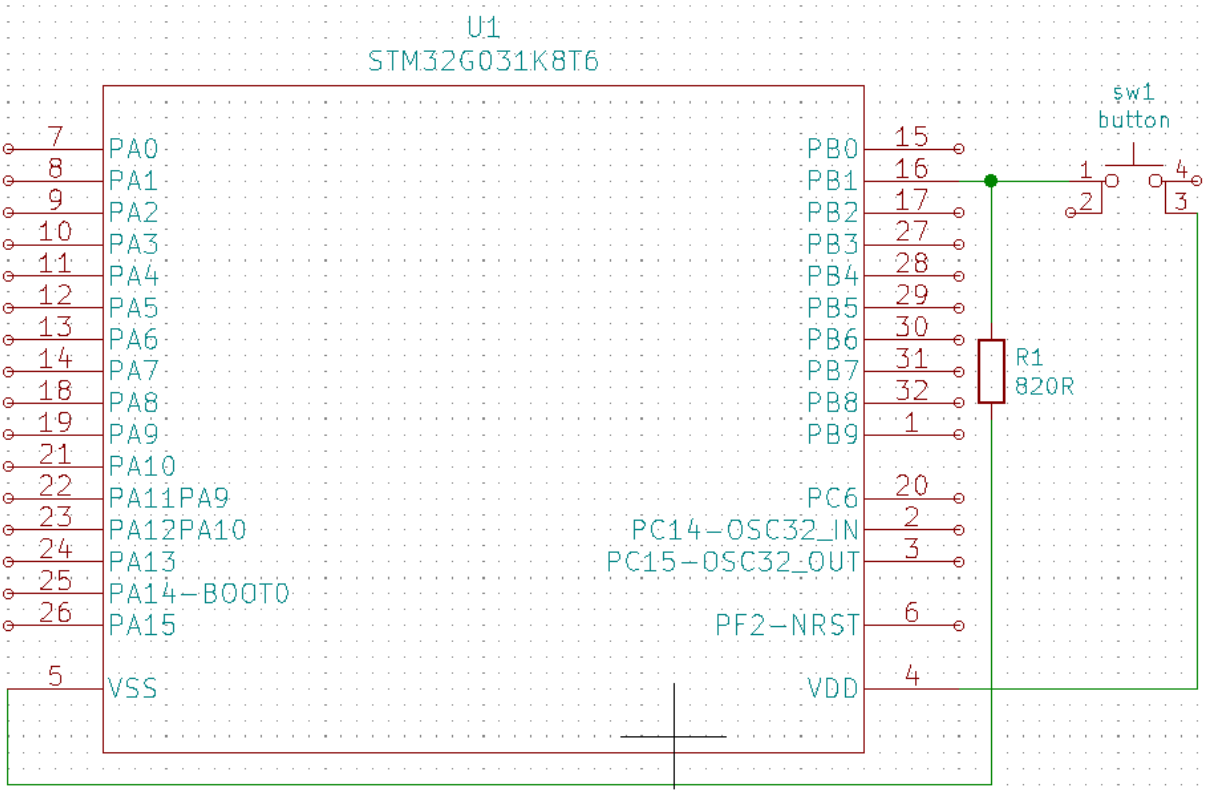


As we can see when the button is pressed and the IDR is updated the hardfault handler is run.

3. Problem 2

In this problem we will implement the code in the problem 1 but instead of using polling we will use an interrupt routine to execute the same routine. First lets choose a pin for the button input. We can choose the PB1 pin as an input to the button.

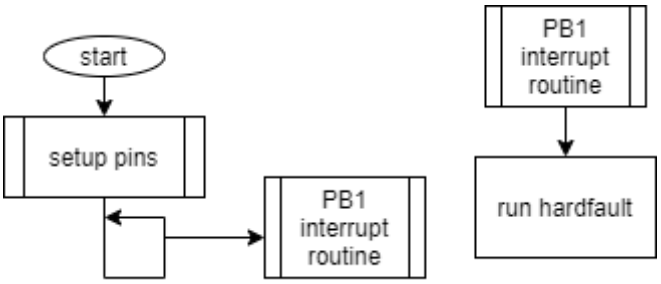
Now we can draw the block diagram for the connections.



Using the block diagram we can create a part list.

	PART NAME	PART COUNT
1	820 ohm resistor	1
2	4 pin button	2
3	STM32G031K8T6	1
4	Breadboard	1
5	Jumper cables	5

We can create the flowchart for the code.

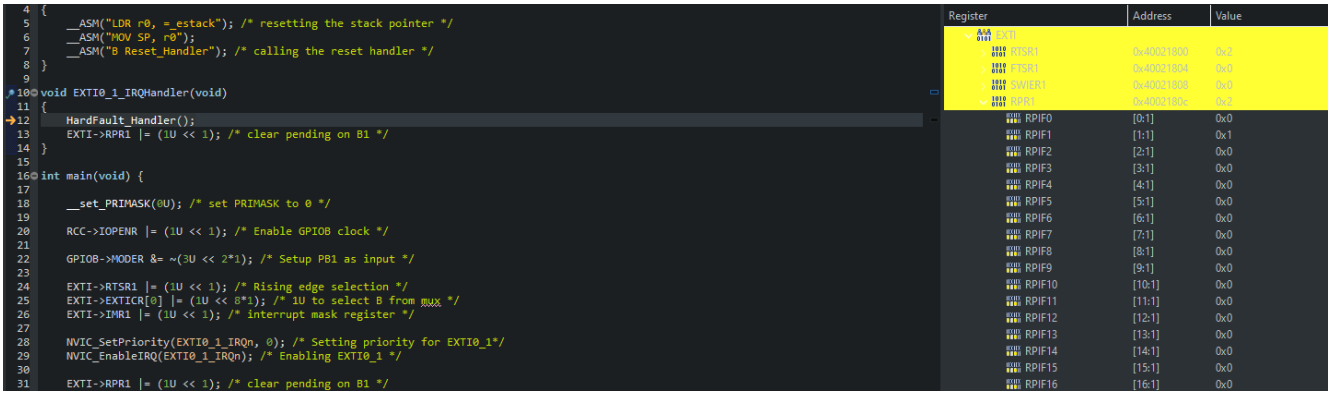


Now the code can be written using the pins we have chosen.

```
#include "stm32g0xx.h"

void HardFault_Handler(void)
{
	__ASM("LDR r0, =_estack"); /* resetting the stack pointer */
	__ASM("MOV SP, r0");
	__ASM("B Reset_Handler"); /* calling the reset handler */
}

void EXTI0_1_IRQHandler(void)
{
	HardFault_Handler();
	EXTI->RPR1 |= (1U << 1); /* clear pending on B1 */
}

int main(void) {

	__set_PRIMASK(0U); /* set PRIMASK to 0 */

    RCC->IOPENR |= (1U << 1); /* Enable GPIOB clock */

    GPIOB->MODER &= ~(3U << 2*1); /* Setup PB1 as input */

    EXTI->RTSR1 |= (1U << 1); /* Rising edge selection */
    EXTI->EXTICR[0] |= (1U << 8*1); /* 1U to select B from mux */
    EXTI->IMR1 |= (1U << 1); /* interrupt mask register */

    NVIC_SetPriority(EXTI0_1_IRQn, 0); /* Setting priority for EXTI0_1*/
    NVIC_EnableIRQ(EXTI0_1_IRQn); /* Enabling EXTI0_1 */

    EXTI->RPR1 |= (1U << 1); /* clear pending on B1 */

    while(1)
    {
```

```
    }
    return 0;
}
```

Now we can debug this code on STM32 to check if it works as intended.



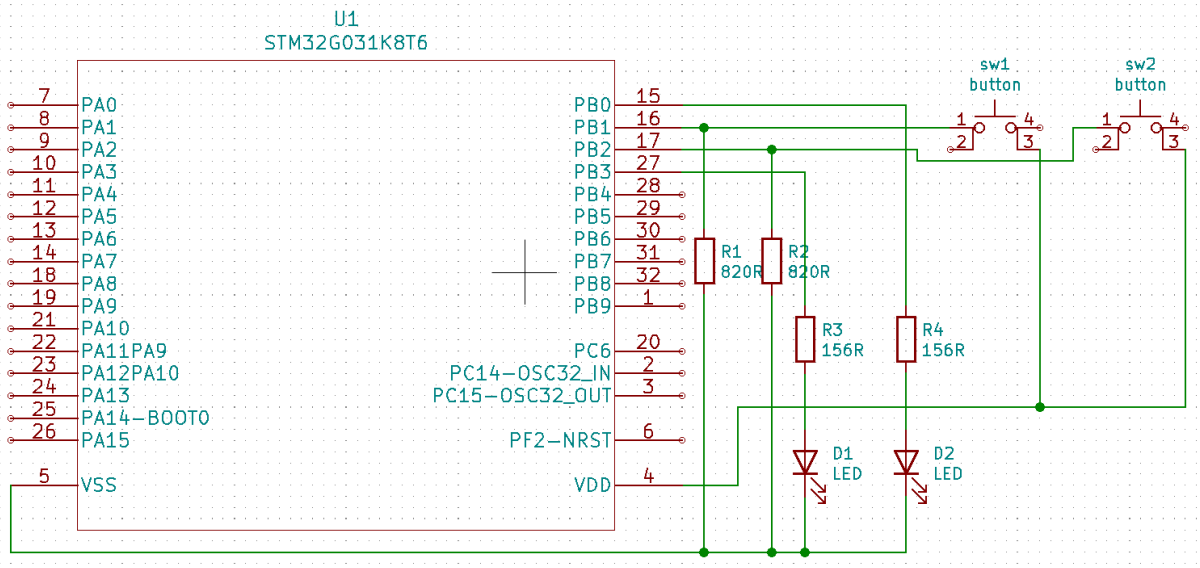
As we can see when the button is pressed the interrupt function is run and the hardfault handler is run.

4. Problem 3

For this problem first we have to implement two buttons that trigger two different interrupt handlers and inside the handlers we will turn on and off the LEDs accordingly. First lets choose the pins for the LEDs and buttons.

We can use B1 and B2 pins for the buttons since we need to use two different handler functions. As for the LED pins we can use B0 and B3 so that it would be easier to setup the pins.

Now for drawing the block diagram to show the connections on the board.

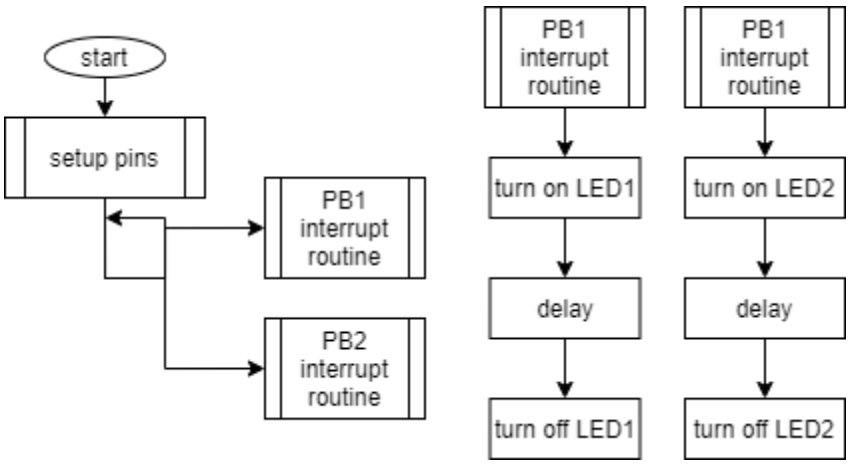


The connection diagram is given above.

Using the block diagram we can create a part list.

	PART NAME	PART COUNT
1	156 ohm resistor	2
2	820 ohm resistor	2
3	Red LED	2
4	4 pin button	2
5	STM32G031K8T6	1
6	Breadboard	1
7	Jumper cables	10

After the part list is created we can create the flowchart to write the main code.

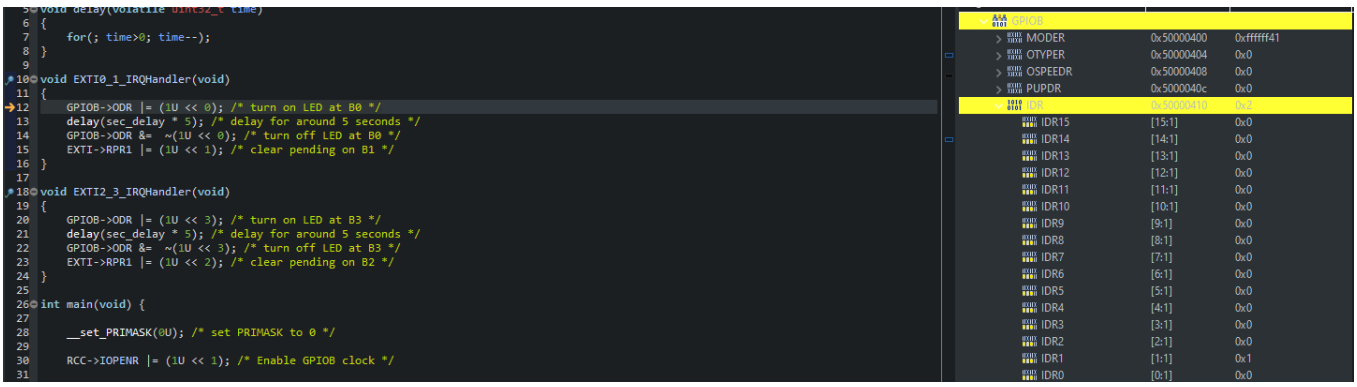


Now for the code we can write the code using the pins we've defined. The code will use the two weakly defined interrupt handler functions and redefine them for the functionality that we want.

```
#include "stm32g0xx.h"

#define sec_delay 1600000

void delay(volatile uint32_t time)
{
    for(; time>0; time--);
}

void EXTI0_1_IRQHandler(void)
{
      EXTI->RPR1 |= (1U << 1); /* clear pending on B1 */
      GPIOB->ODR |= (1U << 0); /* turn on LED at B0 */
      delay(sec_delay * 5); /* delay for around 5 seconds */
      GPIOB->ODR &=  ~(1U << 0); /* turn off LED at B0 */
}

void EXTI2_3_IRQHandler(void)
```

```
{
	EXTI->RPR1 |= (1U << 2); /* clear pending on B2 */
	GPIOB->ODR |= (1U << 3); /* turn on LED at B3 */
	delay(sec_delay * 5); /* delay for around 5 seconds */
	GPIOB->ODR &=  ~(1U << 3); /* turn off LED at B3 */
}

int main(void) {

	__set_PRIMASK(0U); /* set PRIMASK to 0 */

	RCC->IOPENR |= (1U << 1); /* Enable GPIOB clock */

    GPIOB->MODER &= ~(15U << 2*1); /* Setup PB1 and PB2 as input */

    GPIOB->MODER &= ~(0xC3U << 2*0); /* Setup PB0 and PB3 as output */
    GPIOB->MODER |= (0x41U << 2*0);

    EXTI->RTSR1 |= (3U << 1); /* Rising edge selection B1 and B2 */
    EXTI->EXTICR[0] |= (1U << 8*1); /* 1U to select B1 from mux */
    EXTI->EXTICR[0] |= (1U << 8*2); /* 1U to select B2 from mux */
    EXTI->IMR1 |= (3U << 1); /* interrupt mask register B1 and B2 */

    EXTI->RPR1 |= (1U << 1); /* clear pending on B1 */
    EXTI->RPR1 |= (1U << 2); /* clear pending on B2 */

    NVIC_SetPriority(EXTI0_1_IRQn, 0x0U); /* Setting priority for EXTI0_1 */
    NVIC_EnableIRQ(EXTI0_1_IRQn); /* Enabling EXTI0_1 */

    NVIC_SetPriority(EXTI2_3_IRQn, 0x2U); /* Setting priority for EXTI2_3 */
    NVIC_EnableIRQ(EXTI2_3_IRQn); /* Enabling EXTI2_3 */

    while(1)
    {

    }
    return 0;
}
```

With the code written we can test the code on STM32 to check if it behaves as expected on debug mode.



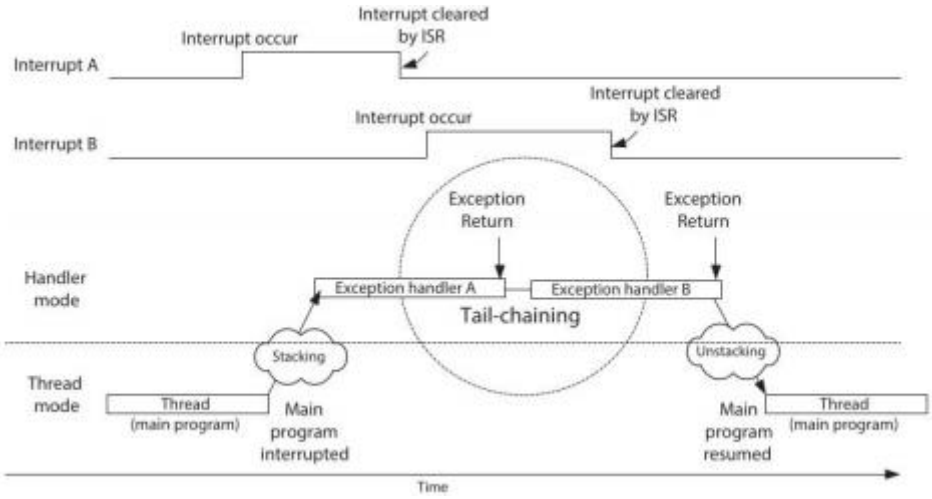
As we can wee when the GPIOB IDR takes input from the PB1 pin the program jumps to the EXTI0_1_IRQ handler function.

```
18  void EXTI2_3_IRQHandler(void)
19  {
20      GPIOB->ODR |= (1U << 3); /* turn on LED at B3 */
21      delay(sec_delay * 5); /* delay for around 5 seconds */
22      GPIOB->ODR &=  ~(1U << 3); /* turn off LED at B3 */
23      EXTI->RPR1 |= (1U << 2); /* clear pending on B2 */
24  }
25
26  int main(void) {
27
28      __set_PRIMASK(0U); /* set PRIMASK to 0 */
29
30      RCC->IOPENR |= (1U << 1); /* Enable GPIOB clock */
31
32      GPIOB->MODER &= ~(15U << 2*1); /* Setup PB1 and PB2 as input */
33
34      GPIOB->MODER &= ~(0xC3U << 2*0); /* Setup PB0 and PB3 as output */
35      GPIOB->MODER |= (0x41U << 2*0);
36
37      EXTI->RTSR1 |= (3U << 1); /* Rising edge selection B1 and B2 */
38      EXTI->EXTICR[0] |= (1U << 8*1); /* 1U to select B1 from MUX */
39      EXTI->EXTICR[0] |= (1U << 8*2); /* 1U to select B2 from MUX */
40      EXTI->IMR1 |= (3U << 1); /* interrupt mask register B1 and B2 */
41
42      EXTI->RPR1 |= (1U << 1); /* clear pending on B1 */
43      EXTI->RPR1 |= (1U << 2); /* clear pending on B2 */
44
```

```
GPIOB
  MODER      0x50000400   0xffffff41
  OTYPER     0x50000404   0x0
  OSPEEDR    0x50000408   0x0
  PUPDR      0x5000040c   0x0
  IDR        0x50000410   0x4
    IDR15    [15:1]       0x0
    IDR14    [14:1]       0x0
    IDR13    [13:1]       0x0
    IDR12    [12:1]       0x0
    IDR11    [11:1]       0x0
    IDR10    [10:1]       0x0
    IDR9     [9:1]        0x0
    IDR8     [8:1]        0x0
    IDR7     [7:1]        0x0
    IDR6     [6:1]        0x0
    IDR5     [5:1]        0x0
    IDR4     [4:1]        0x0
    IDR3     [3:1]        0x0
    IDR2     [2:1]        0x1
    IDR1     [1:1]        0x0
    IDR0     [0:1]        0x0
```

Same goes with the PB2 pin. When the PB2 pin takes and input to IDR the EXTI2_3IRQ handler is run.

Observations show us that when a button is pressed the interrupt routine is executed, the LED turns on and the delay function delays the led turning of for 5 seconds. If the button is pressed in that 5 seconds the interrupt routine keeps running and a pending routine is called. Tail chaining event occurs and the pending interrupt routine is run after the first interrupt routine is over.



If the higher priority interrupt has come while the lower priority interrupt routine is running the higher priority interrupt routine is run inside the lower priority interrupt routine. This preemption occurs and creates a nested interrupt routine.

Now that we have tested the program we can wire up the components to the board and test the board on real time.

The test video will be posted on Youtube and can be watched with the link given below.

5. Results and General Comments

In this lab we have looked into hardfault handlers, interrupts and priorities. In the first problem we have used a hardfault handler to handle faults that could be generated. In the second problem we used interrupts instead of polling to create faults and handle them using the hardfault. In the last third problem we used two interrupts to light up 2 LEDs. These two

handlers had different priorities. When the lower priority interrupt is pending and the higher priority interrupt is pending the higher priority interrupt handler is run inside the lower priority handler creating a nested interrupt. This preemption is observed on the video.

6. References

[1]. ELEC 335 Lecture Slides (5-6-7).