1. INRODUCTION:

The objective of this lab is to give a thorough understanding of exception operation, fault generation and handling, external interrupt and managing multiple sources, and priority. We use C language fort he problems unless some parts require inline assembly.

2. PROBLEMS:

2.1. Problem 1:

In this problem, you will work on fault generation and handling.

- Create a hardfault handler that will detect possible faults. The handler should properly restore stack pointer, and call reset handler afterwards. After compiling, examine the routine to make sure there are no overheads. The exact name of this function can be seen in startup_xxx file under include directory.

- Explain the type of faults that we discussed in the class and implement a routine for each case. Observe the hardfault operation, check the context after a fault occur, and expkain the behavior.

- Attach an external button to your board, and using polling method execute these hardfaults randomly. For this you can use rand() C library function.

- Optionally you can implement a simple toggle LED mechanism in the begginning of your code to see the reset opreration.

2.1.1. Theoretical Research:

Writing inline assembly code:

The compiler provides an inline assembler that enables you to write assembly code in your C or C++ source code, for example to access features of the target processor that are not available from C or C++. The __asm keyword can incorporate inline assembly code into a function using the GNU inline assembly syntax. For example:

```
#include <stdio.h>

int add(int i, int j)

{

  int res = 0;

  __asm ("ADD %[result], %[input_i], %[input_j]"
```

```
#include <stdio.h>

int add(int i, int j)

{

  int res = 0;

  __asm ("ADD %[result], %[input_i], %[input_j]"

    : [result] "=r" (res)

    : [input_i] "r" (i), [input_j] "r" (j)

  );

  return res;

}


int main(void)

{

  int a = 1;

  int b = 2;

  int c = 0;


  c = add(a,b);


  printf("Result of %d + %d = %d\n", a, b, c);

}
```

The general form of an __asm inline assembly statement is:

```
__asm [volatile] (code); /* Basic inline assembly syntax */

/* Extended inline assembly syntax */

__asm [volatile] (code_template

       : output_operand_list

      [: input_operand_list

      [: clobbered_register_list]]

  );
```

Use the volatile qualifier for assembler instructions that have processor side-effects, which the compiler might be unaware of. The volatile qualifier disables certain compiler optimizations, which may otherwise lead to the compiler removing the code block. The volatile qualifier is optional, but you should consider using it around your assembly code blocks to ensure the compiler does not remove them when compiling with -O1 or above.code is the assembly instruction, for example "ADD R0, R1, R2". code_template is a template for an assembly instruction, for example "ADD %[result], %[input_i], %[input_j]". If you specify a code_template rather than code then you must specify the output_operand_list before specifying the optional input_operand_list and clobbered_register_list. output_operand_list is a list of output operands, separated by commas. Each operand consists of a symbolic name in square brackets, a constraint string, and a C expression in parentheses. In this example, there is a single output operand: [result] "=r" (res). The list can be empty.

For example:

```
__asm ("ADD R0, %[input_i], %[input_j]"
    :  /* This is an empty output operand list */
    : [input_i] "r" (i), [input_j] "r" (j)
  );
```

2.1.2 Code:

```
/*
 * main.c
 *
 * author: Mehmet Akif Gümüş
 *
 */

#include "stm32g0xx.h"
#include "cmsis_device.h"
#include "stdio.h"
void Hard_Fault_Handler(uint32_t *sp);

__asm void HardFault_Handler(void){
      asm volatile();

}
enum { r0, r1, r2, r3, r12, lr, pc, psr};

int main(void);
void delay(volatile uint32_t);
```

```c
int RETARGET_ReadChar(void)
{
    return 0;
}
int RETARGET_WriteChar(char c)
{
    return ITM_SendChar(c);
}


int main(void) {

      printf("Start\n");

          /* Create an invalid function pointer and call it.
           * This will trigger a Hard Fault.  */
          void (*fp)(void) = (void (*)(void))(0x00000000);
          sp();


    while(1) {

    return 0;
}

      void Hard_Fault_Handler(uint32_t *sp)
      {
          uint32_t cfsr  = SCB->CFSR;
          uint32_t hfsr  = SCB->HFSR;
          uint32_t mmfar = SCB->MMFAR;
          uint32_t bfar  = SCB->BFAR;

          uint32_t r0  = sp[0];
          uint32_t r1  = sp[1];
          uint32_t r2  = sp[2];
          uint32_t r3  = sp[3];
          uint32_t r12 = sp[4];
          uint32_t lr  = sp[5];
          uint32_t pc  = sp[6];
          uint32_t psr = sp[7];

          printf("HardFault:\n");
          printf("SCB->CFSR   0x%08lx\n", cfsr);
          printf("SCB->HFSR   0x%08lx\n", hfsr);
          printf("SCB->MMFAR  0x%08lx\n", mmfar);
          printf("SCB->BFAR   0x%08lx\n", bfar);
          printf("\n");

          printf("SP          0x%08lx\n", (uint32_t)sp);
          printf("R0          0x%08lx\n", r0);
          printf("R1          0x%08lx\n", r1);
          printf("R2          0x%08lx\n", r2);
          printf("R3          0x%08lx\n", r3);
          printf("R12         0x%08lx\n", r12);
```

```
        printf("LR          0x%08lx\n", lr);
        printf("PC          0x%08lx\n", pc);
        printf("PSR         0x%08lx\n", psr);

        while(1);
    }
```

2.2 PROBLEM-2

In this problem, you will work with external interrups.

- Setup and implement an external interrupt routine with highest possible priority, and tie it to your external button. Upo pressing, your intterrupt routine should execute the previously implemented hardfaults randomly.
- When interrupt happens, check out the context and explain each section.
- Your main while loop should be empty for this problem since everthing will be taken care of in the interrupt handler routine.

2.2.1 Theoretical Research:

Interrupts basics CMSIS:

Interrupt is a processor request to stop doing everything what was going on in the system and execute some “interrupt routine”. After that, the system would go back to continue the instructions which were interrupted. That way, the important data would be received because the interrupt would trigger the interrupt routine (special function called when an interrupt occurs).

In this post we will try to write the code which would toggle an LED when the button is pressed.

The LED configuration is the same as in the previous post. Now let’s configure the button. On my board it’s located on pin PA0.

The configuration of LED IO:

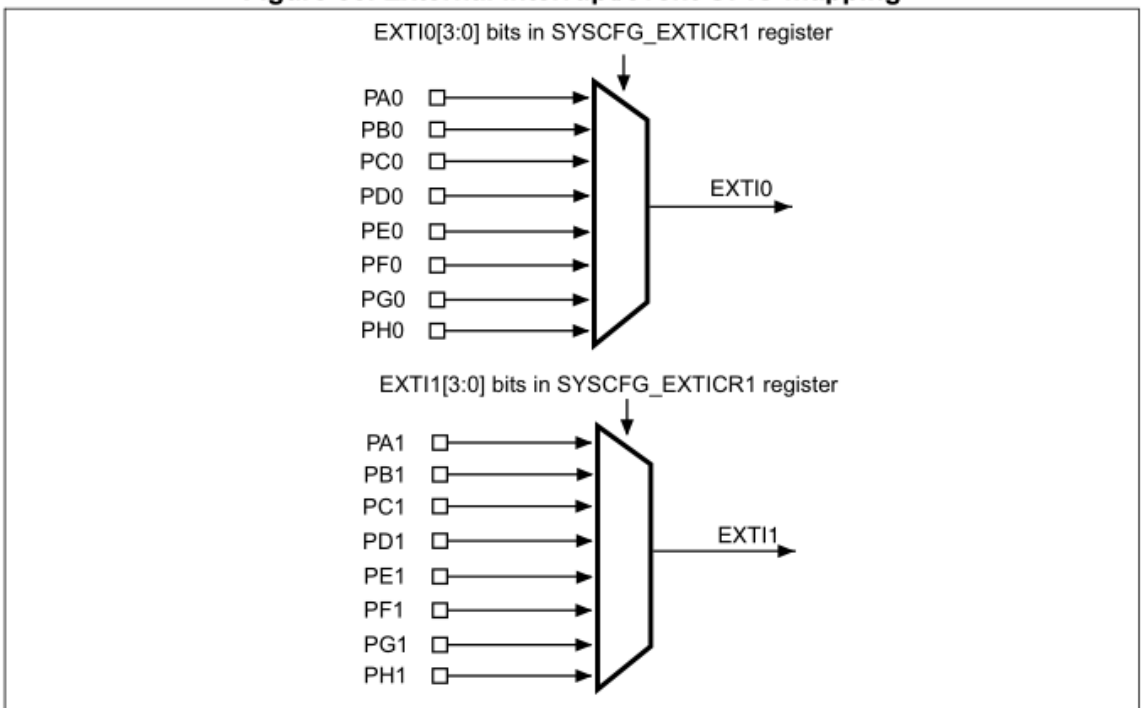
1. We enabled the GPIOA clock
2. We set the pin0 as input (00). “&= ~” operation clears the selected bits (sets them to zero).
3. We set the pin0 as push-pull option (0) – also by clearing the corresponding bit. It is not necessary, but most of the time it is done to to make sure that everything is configured correctly.
4. We set the speed to “very high speed” (11).
5. We set the pin to “No pull-up, pull-down” – it is done physically on the board.

As you can see, the configuration is very similar to the button configuration.

Now, let's configure the interrupts. To do that, let's open the datasheet on the chapter 10 "Interrupts and events". The thing which is the most interesting for us at the moment is EXTI – External Interrupt. External interrupts are the interrupts which are triggered be a specific IO line – for example SPI data receiving, some sensor signal or our button press. Also, there are internal interrupts, like clocks overflows.

Each line can be be configured to have its interrupt, however they mapping is not 1 to 1. The IOs are grouped to EXTI interrupts, so we have to choose which one we are going to use. It is shown on the picture in section 10.2.5 "External interrupt/event line mapping":

Figure 33. External interrupt/event GPIO mapping



2.3 PROBLEM-3

In this problem, you will work with multiple sources for external interrupts and prioritie.

- Connect two buttons and two LEDs to your board. Enable external interrupt fot both of them. Each button should toggle one LED.
- Connect one button in one of the following groups: {0,1}{2,3,}{4,-15} and the other in the remainin group.
- You will use two external interrupt handlers. Make sure to check using the pending bit if the correct interrupt is received.
- Assign different priorities to your buttons, and inside the handlers create a long delay that will last for at least 5-10 seconds
- You can implement this behavior by turning on the associated LED in the begibbing, then an empty for loop, then turning off the associated LED.
- Observe the preemption by trying to interrupt the current interrupt routine. Explain your observations.

3. CONCLUSION

The purpose of the experiment are understanding of exception operation, fault generation and handling, external interrupt and managing multpile sources, and priority. STM32 board is used for the first time. In order to use the pins on the card, a clock signal was sent to those pins first. After the pins were opened for use, GPIO IDR and ODR registers were assigned according to the datasheet. Necessary connections have been made on the card. Connections were made according to the information form that came with the card.

Though, Due to the lack of sufficient competence; This experiment was not successful. Even though we repeated what was explained in the course, it was not enough and the necessary online research could not be done due to the busy schedule.

4. References

- https://www.keil.com/pack/doc/CMSIS/Core/html/group__NVIC__gr.html
- https://blog.feabhas.com/2013/02/developing-a-generic-hard-fault-handler-for-arm-cortex-m3cortex-m4/
- https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2014/05/26/debug_a_hardfault-78gc