Bern University
of Applied Sciences

Exercise - 05

Basic Interrupt

Andreas Habegger | Adrian Steiner
BTS3230 | Version 1.0.0 of 17.03.2025

In this exercise you will learn about hardware interrupts and how to handle them. Reading an input periodically is also called polling. This is very easy to implement, but also results in unnecessary register reads and CPU load. With interrupts, the CPU interrupts the current programme sequence and executes the processes from the interrupt until they are finished. You are implementing a toggle LED application without polling the state of an input pin.

Objectives

- Nested Vectored Interrupt Controller “NVIC”
- External Interrupt/Event Controller “EXTI”
- Implement “Rising edge” interrupt on SW centre button
- Code modularization with callback functions

Outcomes

- Handling hardware interrupts
- Understand Nested Vectored Interrupt Controller “NVIC”
- Understand the advantage of an interrupt over polling
- Using code modularization for higher abstraction
- Run a program without any superloop control logic

Description

To detect a change in the physical state of an input pin, there is a concept called “polling”. It is based on repeatedly looking up the current state in the register as the “superloop” is traversed. However, this can introduce overhead and is not well suited to deterministic systems and systems that need to react to input changes. Hardware interrupts offer a way to overcome the disadvantages of polling. More specifically, the External Interrupt Controller, also known as the Event Controller “EXTI”, in conjunction with the Nested Vectored Interrupt Controller “NVIC”, are the sub-circuits on ARM-based MCUs that do this. The NVIC will call the Interrupt Service Routine (ISR) based on its configuration to handle interrupts. The procedure is triggered by an Interrupt Service Request (“IRQ”), which is the detection of the hardware event in this scenario.

Note

Only the NULCEO-64 board and the mbed Application Shield are used in this exercise, but the temperature logger board has no effect on this exercise and can remain connected.

Tasks

Toggle the red RGB LED on the mbed application shield on a **rising edge** of the SW centre button by using an interrupt on this input. Implement everything with an empty superloop.

Example

To get started quick, we provide an example for the USER button and the LED on the NULCEO-64 board. The LED gets toggled with an interrupt on a **falling edge**.

Important

You will have to analyze and rewrite the code to use the red RGB LED and SW centre button on the mbed application shield.

Main function with empty superloop

Main function with placeholder for the initialization and empty super loop.

```
int main(void){
  /* Initialisation */

  while (1){
    /* user code executed repeatedly, empty in this exercise */
  }
}
```

Initialization of the GPIO and the EXTI interrupt peripheral

Code is executed once and before the super loop.

```
/* Enable GPIOA and GPIOC clocks */
SET_BIT(RCC->AHB1ENR, RCC_AHB1ENR_GPIOAEN);
SET_BIT(RCC->AHB1ENR, RCC_AHB1ENR_GPIOCEN);
/* Clock Init SYSCFG  */
SET_BIT(RCC->APB2ENR, RCC_APB2ENR_SYSCFGEN);

/* Configure GPIO (Green LED on Board [PA5]) */
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODE5_Msk, 0b01 << GPIO_MODER_MODE5_Pos);

/* Configure EXTI for the push button on dev board (PC13) */
MODIFY_REG(GPIOC->MODER, GPIO_MODER_MODE13_Msk, 0b00 << GPIO_MODER_MODE13_Pos);
/* Set IMR (Interrupt Mask Register) */
SET_BIT(EXTI->IMR, EXTI_IMR_MR13);
/* Set FTSR (Falling Trigger Selection Register) */
SET_BIT(EXTI->FTSR, EXTI_FTSR_TR13);
/* Set SYSCFG_EXTICR2 (System Configuration Controller External Interrupt
Configuration Register) */
SET_BIT(SYSCFG->EXTICR[3], SYSCFG_EXTICR4_EXTI13_PC);
/* Set (Clear) PR (Pending Register) before enabling NVIC */
SET_BIT(EXTI->PR, EXTI_PR_PR13);
/* Set NVIC_IPRx (Interrupt Priority Register) */
NVIC_SetPriority(EXTI15_10_IRQn, 1);
/* Set NVIC_ICPRx (Interrupt Clear-Pending Register) */
NVIC_ClearPendingIRQ(EXTI15_10_IRQn);
/* Set NVIC_ISERx (Interrupt Set-Enable Register) */
NVIC_EnableIRQ(EXTI15_10_IRQn);
```

Note

The example shown next is one way of handling an interrupt. This method focuses on modularisation, mainly used in HAL libraries, to decouple low-level handling from user-specific actions triggered by an interrupt. We will use the same approach even though we don’t use the HAL library.

Reimplementation of weak function “EXTI15_10_IRQHandler()”

The function `EXTI15_10_IRQHandler()` is a valid symbol defined and registered in the vector table implementation (see stm32_startup.c in the template). If there is no implementation of this function on an EXTI15_10 IRQ, the default IRQ handler will be called instead (weak linking). This function calls a user defined EXTI IRQ handler.

```
void EXTI15_10_IRQHandler(void){
    custom_GPIO_EXTI_IRQHandler(1 << 13);
}
```

User defined EXTI IRQ Handler

The custom handler checks where an interrupt is pending and clears it. It then calls the user callback for the pending line.

```
void custom_GPIO_EXTI_IRQHandler(uint32_t pendingLine){
    if (0 != READ_BIT(EXTI->PR, pendingLine)) {
        SET_BIT(EXTI->PR, pendingLine);
        custom_GPIO_EXTI_Callback(pendingLine);
    }
}
```

Interrupt callback function

This function implements the action for the current pending interrupt line.

```
void custom_GPIO_EXTI_Callback(uint32_t pendingLine){
    if ((1 << 13) == pendingLine) {
        if (READ_BIT(GPIOA->ODR, GPIO_ODR_OD5))
            CLEAR_BIT(GPIOA->ODR, GPIO_ODR_OD5);
        else
            SET_BIT(GPIOA->ODR, GPIO_ODR_OD5);
    }
}
```

Inside the ISR we check which IRQ has been fired. Then we clear the IRQ and do whatever we need to do based on the functional requirements. The name of the IRQ handler is defined by ARM and must match the entry in the vector table. To decouple the standardised names from domain specific implementations, we call our custom interrupt handler in the “IRQ” handler.

Question

Analyze the last three functions above and answer following questions:

a. What happens in each function?

b. What is the advantage of this decoupling implementation?

c. What is important about interrupt service routines?

Implementation

Implement the interrupt on a rising edge of the SW centre button using the example above. Start with the initialisation part.

Exercise

1. Initialise the GPIO that drives the red RGB LED as output.
2. Initialise the SW centre button as an input as in the previous exercises.
3. Configure the “EXTI” on the desired “line” by writing to the interrupt mask register “IMR”.
4. Configure the “EXTI” to the desired “edge” by writing to the rising trigger selection register “RTSR”.
5. Map the “EXTI” line to the “NVIC” by writing to the System Configuration Controller external interrupt configuration register “EXTICRx”.
6. Set (=> clear) the pending register “PR” before enabling the NVIC
7. Set a priority for the “IRQ” sent by “EXTI” to the “NVIC” using the CMSIS function `NVIC_SetPriority()`, we recommend the priority 1.
8. Clear the NVIC Interrupt Clear-Pending Register “ICPR” by using the CMSIS function `NVIC_ClearPendingIRQ()`
9. Finally, enable the “IRQ” by using the CMSIS function `NVIC_EnableIRQ()`.

After initialisation, implement the callback functions using the modularisation from the example.

Exercise

10. Implement the callback function that is defined in the startup.c file for the SW centre button as a low level stage.

> **Hint**
>
> By setting a breakpoint in this function, you can test your initialisation. When the SW centre button is pressed, the debugger should jump to that function and stop the program there.

11. If all works, implement the next level `custom_GPIO_EXTI_IRQHandler()` and clear the IRQ there.
12. Finally, implement a high-level `custom_GPIO_EXTI_Callback()` with the action to toggle the red RGB LED.