Lab 5: Interrupts

COEN-4720 Embedded Systems

Cris Ababei
Dept. of Electrical and Computer Engineering, Marquette University

1. Objective

The objective of this lab is to learn about interrupts. We will look at interrupts related to GPIOs, SysTick, and Timers.

2. Description

Prerequisites for this Lab
Read Chapters 7 (not 7.4.2) and 11 from the textbook.

Introduction to Interrupts
There are multiple ways of describing what an **interrupt** is.

An **interrupt** is the automatic transfer of software execution in response to a hardware event that is **asynchronous** with the current software execution. This hardware event is called a **trigger**. The hardware event can either be a *busy to ready transition* in an external I/O device (i.e., peripheral, like for example the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer). When the hardware needs service, signified by a busy to ready state transition, it will request an **interrupt** by setting its trigger flag.

So, interrupts can originate both by the hardware and the software itself. ARM architecture distinguishes between the two types: interrupts originated by the hardware, exceptions by the software (e.g., an access to invalid memory location). In ARM terminology, an interrupt is a type of exception.

An **interrupt** can be seen as an asynchronous event that causes stopping the execution of the current code on a priority basis (the more important the interrupt is, the higher its priority; this will cause that a lower-priority interrupt is suspended). The code that services the interrupt is called an **Interrupt Service Routine (ISR)**.

A **thread** is defined as the path of action of software as it executes. The execution of the **interrupt service routine (ISR)** is called as a background thread, which is created by the hardware interrupt request and is killed when the ISR returns from interrupt. A new thread is created for each interrupt request. In a **multi-threaded** system, threads normally cooperate to perform an overall task.

A **process** is also defined as the action of software as it executes. Processes do not necessarily cooperate towards a common shared goal. Threads share access to I/O devices, system resources, and global variables, while processes have separate global variables and system resources. Processes do not share I/O devices.

Interrupts in Cortex-M processors are controlled by the **Nested Vector Interrupt Controller (NVIC),** which is a unit dedicated to exceptions management. Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located. Vectors are stored in ROM at the beginning of the memory. The relationship between NVIC, Cortex-M processor, and peripherals is shown in Figure 1 below [1].

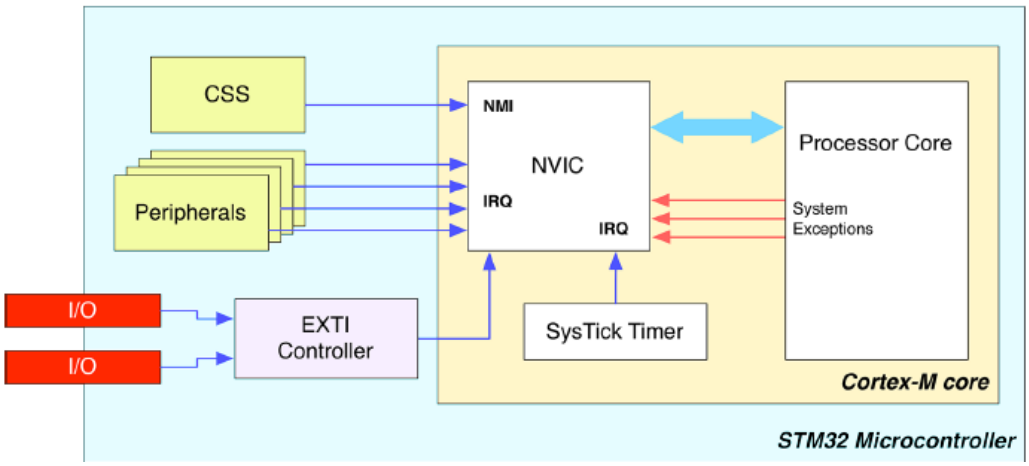

Figure 1: Relationship between NVIC, MCU, and peripherals [1].

All Cortex-M processors reserve a fixed set of 15 (first ones in the vector table) exceptions common to all Cortex-M families. See Section 7.1.1 in the textbook for a description of them. You can see them also inside the file /Startup/startup_stm32l053r8tx.s of a project in STM32CubeIDE.

The remaining exceptions that can be defined for a given MCU are related to IRQ handling. Cortex-M0/0+ cores allow up to 32 external interrupts, Cortex-M3/4/7 cores allow silicon manufacturers to define up to 240 interrupts while Cortex-M33 cores up to 480 IRQ lines.

When an STM32 MCU boots up, only Reset, NMI and Hard Fault exceptions are **enabled** by default. The rest of exceptions and peripheral interrupts are **disabled**, and they **must be enabled on request**. To enable an IRQ, the CubeHAL provides the following function:

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
```

where the IRQn_Type is an *enumeration* of all exceptions and interrupts defined for that specific MCU. IRQn is the interrupt number in the NVIC table.
You can see details of IRQn_Type enum inside:
Drivers/CMSIS/Device/ST/STM32XXxx/Include/stm32l053xx.h
For example, the IRQn value for the SysTick internal exception is SysTick_IRQn = -1, while the IRQn value for the SPI1 interrupt is SPI1_IRQn = 25.

Example 1

In this example, we use interrupts to toggle the LD2 LED every time we press the user-programmable button, which is connected to the PC13 pin. So, create a new project called **lab5_ex1**. During the creation process, when CubeMX is launched, configure the clock tree so that the high speed internal (HSI) clock source is used. To do that select the corresponding input of the system clock mux, as shown in Figure 2. This is done here only to illustrate also the clock configuration to use the internal HSI clock source vs. the external source HSE. Usually the HSE clock source is better as it is higher frequency and more stable.

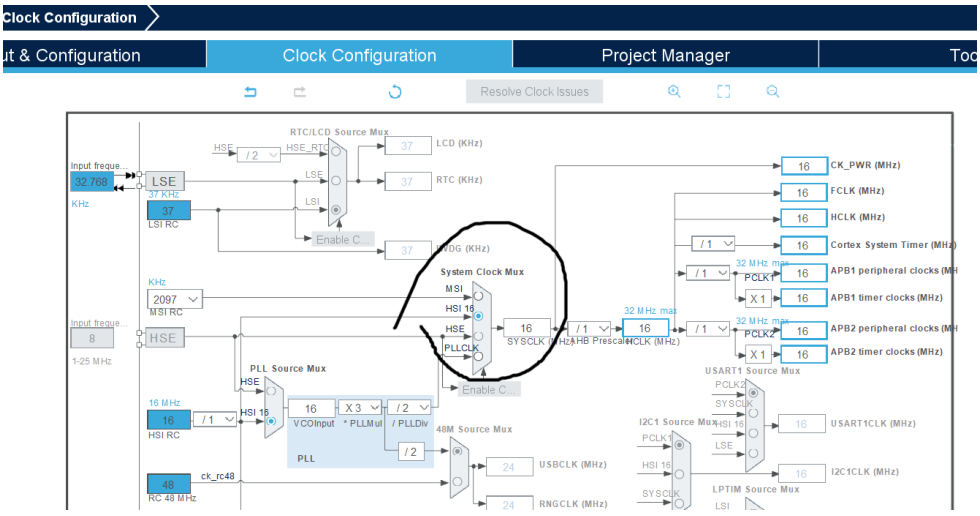

Figure 2: Select HSI clock source.

The source code of this example is provided inside the files archive for this lab.
We configure the GPIO PC13 to fire an interrupt every time it goes from the low level to the high one (i.e., rising edge). This is accomplished by setting GPIO. Mode to GPIO_MODE_IT_RISING:

```
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
```

For the complete list of available interrupt related modes see Table 6.2 in the textbook [1].

Next, we **enable** the interrupt of the EXTI line associated with the Px13 pins, that is *EXTI4_15_IRQn*:

```
HAL_NVIC_EnableIRQ(EXTI4_15_IRQn);
```

We do that because EXTI lines 4 to 15 share the same IRQn inside the NVIC (and hence are serviced by the same ISR). This can be seen inside file:
Drivers/CMSIS/Device/ST/STM32XXxx/Include/stm32l053xx.h
Where, we see:

```
EXTI4_15_IRQn     = 7,     /*!< EXTI Line 4 to 15 Interrupts */
```

Please also see Table 55 and Figure 30 from the MCU Reference Manual to double check this fact!

While the above code could be used directly inside the main() function – as shown in the textbook (listing "Filename: src/main-ex1.c" on page 150) - we prefer to have it done inside the function:

```
static void MX_GPIO_Init(void)
```

which, is then called inside the main() function.

The ISR associated with the IRQ for the EXTI4_15_IRQn line needs then to be described. In our case, it is as simple as follows:

```
void EXTI4_15_IRQHandler(void)
{
  if(__HAL_GPIO_EXTI_GET_IT(GPIO_PIN_13) != RESET) {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_13);
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
  }
}
```

NOTE: It is declared as a handler inside the file:

Core/Startup/startup_ stm32l053r8tx.s
Here, we only provide a description of what the actual code of the corresponding ISR routine is.

This routine will be called and executed whenever the event of rising edge happens on the pins associated with EXTI lines 4 to 15. Then, inside this ISR, we **check** if the event happened on the PC13, case in which we clear it and **toggle** pin PA5.
The above **check** is done with the macro:

```
/**
  * @brief  Checks whether the specified EXTI line is asserted or not.
  * @param  __EXTI_LINE__ specifies the EXTI line to check.
  *          This parameter can be GPIO_PIN_x where x can be(0..15)
  * @retval The new state of __EXTI_LINE__ (SET or RESET).
  */
#define __HAL_GPIO_EXTI_GET_IT(__EXTI_LINE__) (EXTI->PR & (__EXTI_LINE__))
```

Which you can find inside (please open this file and search that inside it!):
Drivers/STM32L0xx_HAL_Driver/Inc/stm32l0xx_hal_gpio.h

The above **toggle** is done with the function:

```
/**
  * @brief  Toggles the specified GPIO pins.
  * @param  GPIOx Where x can be (A..E and H) to select the GPIO peripheral for
STM32L0xx family devices.
  *               Note that GPIOE is not available on all devices.
  *               All port bits are not necessarily available on all GPIOs.
  * @param  GPIO_Pin Specifies the pins to be toggled.
  * @retval None
  */
void HAL_GPIO_TogglePin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)
{
  uint32_t odr;

  /* Check the parameters */
  assert_param(IS_GPIO_PIN_AVAILABLE(GPIOx, GPIO_Pin));

  /* get current Output Data Register value */
  odr = GPIOx->ODR;

  /* Set selected pins that were at low level, and reset ones that were high */
  GPIOx->BSRR = ((odr & GPIO_Pin) << GPIO_NUMBER) | (~odr & GPIO_Pin);
}
```

Described inside:
Drivers/STM32L0xx_HAL_Driver/Src/stm32l0xx_hal_gpio.c

After studying the above codes, build the project. Program the Nucleo board. Observe operation.

Example 2
This is a modified version of example 1 and is also included in the files of this lab as **lab5_ex2**. A more common way to implement example 1 with interrupts is to use the so called **callback functions** approach.

Create a new project **lab5_ex2**, as a modified version of example 1; follow the discussion from Section 7.2.1 in the textbook – more specifically as shown in listing "Filename: src/main-ex2.c" on

page 151-152. Note however that the code from the textbook is not directly usable; in this case, the ISR and the callback functions are:

```
void EXTI4_15_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  // this is where user code should go, inside the callback function
  // rather than in EXTI4_15_IRQHandler(); also, as a rule, we should
  // minimize the amount of code inside ISRs; put as little code inside
  // ISRs; ok, to put more code inside main();
  if (GPIO_Pin == GPIO_PIN_13) {
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
  }
}
```

In this case, when the EXTI4_15_IRQHandler() ISR is called, the control is transferred to the HAL_GPIO_EXTI_IRQHandler() function inside the HAL. This will perform for us all the interrupt related activities, and it will call the HAL_GPIO_EXTI_Callback() routine passing the actual GPIO pin that generated the IRQ.

Introduction to Timers

A timer is basically a counter with a counting frequency that is a fraction of its source clock. Depending on the timer type, it can be clocked by the internal clock, by an external clock source, or by another timer used as "master".

Usually, a timer counts from zero up to a given value, which cannot be higher than the maximum unsigned value for its resolution. For example, a 16-bit timer overflows when the counter reaches 65535. Timers can be configured though to count downwards, from a pre-specified value down to zero.

Depending on the timer type, a **timer can generate interrupts or DMA** (dynamic memory access) requests when the following events occur:

- Update events
  - Counter overflow/underflow
  - Counter initialized
  - Others
- Trigger
  - Counter start/stop
  - Counter Initialize
  - Others
- Input capture/Output compare

Different STM32 MCU families have (or not) available different types of timers. See Table 11.2 from the textbook [1] for a summary. Check out in that table the types of timers the STM32L0 has. You should see that there are available for example two basic timers (**TIM6 and TIM7**) and one general-purpose 16-bit timer **TIM2**.

A timer is referenced by using an instance of the C struct **TIM_HandleTypeDef**, which is defined as follows:

```
typedef struct {
    TIM_TypeDef *Instance; /* Pointer to timer descriptor */
    TIM_Base_InitTypeDef Init; /* TIM Time Base required parameters */
    HAL_TIM_ActiveChannel Channel; /* Active channel */
    DMA_HandleTypeDef *hdma[7]; /* DMA Handlers array */
    HAL_LockTypeDef Lock; /* Locking object */
    __IO HAL_TIM_StateTypeDef State; /* TIM operation state */
} TIM_HandleTypeDef;
```

Check it out inside this file:
Drivers/STM32L0xx_HAL_Driver/Inc/stm32l0xx_hal_tim.h
And read more about each member of that structure in Section 11.2 of the textbook [1]!

Note the member variable `TIM_Base_InitTypeDef Init` in the above structure.
It is an instance of the C struct TIM_Base_InitTypeDef, and it is where all the timer configuration activities are performed. This structure is defined as follows:

```
typedef struct {
    uint32_t Prescaler; /* Specifies the prescaler value used to divide the TIM clock. */
    uint32_t CounterMode; /* Specifies the counter mode. */
    uint32_t Period; /* Specifies the period value to be loaded into the active
    Auto-Reload Register at the next update event. */
    uint32_t ClockDivision; /* Specifies the clock division. */
    uint32_t RepetitionCounter; /* Specifies the repetition counter value. */
} TIM_Base_InitTypeDef;
```

Check it out inside this file:
Drivers/STM32L0xx_HAL_Driver/Inc/stm32l0xx_hal_tim.h
And read more about each member of that structure in Section 11.2 of the textbook [1]!

Example 3

In this example, we use the basic timer **TIM6** in interrupt mode to blink the green LED once per second. It is included in the files archive provided for this lab.

We will use a basic timer, for which we are will need to configure several parameters:

- **Period.** This specifies the value up to which the counter will count starting from zero. It is one of the variables of TIM_Base_InitTypeDef, which can assume the maximum value of 0xFFFF for a 16-bit timer (0xFFFF FFFF for 32-bit timers).
- **Prescaler** register in the initialization structure. It is used to lower the counting frequency (that depends on the speed of the bus where the timer is connected).

When the timer reaches the **Period** value, it overflows, and the **Update Event (UEV)** flag is set; then, the timer automatically restarts counting from the initial value (which is zero for basic timers).

The **Period** and **Prescaler** registers determine the frequency of the timer. The inverse of the frequency is the time duration that it takes for the timer to count from zero to the set value and it overflows – when the **Update Event** is generated. The formula for the frequency of generation of **Update Event (UEV)** is:

$$UpdateEvent_{freq} = \frac{Timer_{clock-freq}}{(\mathbf{Prescaler} + 1)(\mathbf{Period} + 1)}$$

For example, given a timer connected to the APB1 bus in an STM32L058 MCU, with the HCLK set to 16MHz, a Prescaler value of 15999 and a Period value of 499. The timer will overflow after a duration of 0.5 s, which is of course: $1/UpdateEvent_{freq}$ which in turn is:

$$UpdateEvent_{freq} = \frac{16.000.000\ Hz}{(\mathbf{15.999}+1)(\mathbf{499}+1)} = 2\ \text{Hz}$$

To do this example, create a new project, **lab5_ex3**.
Right after you created the project, inside CubeMX, configure the clock to use HIS.
Also, configure the timer TIM6 with the above Period and Prescaler values – as shown in Figure 3 below. Or, just port the necessary code from the provided files for this lab.

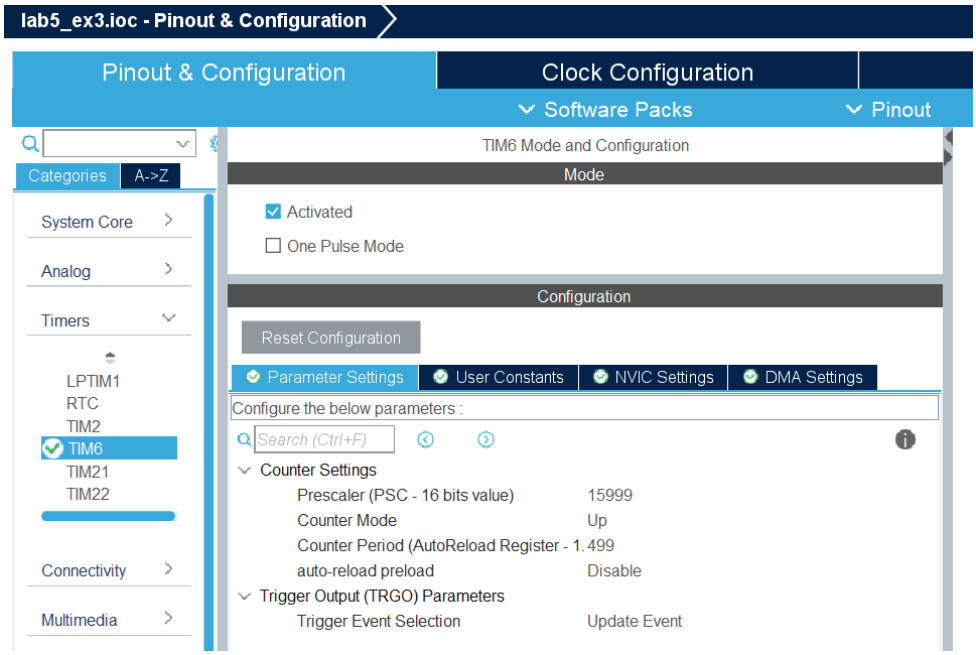

Figure 3: Configuration of timer TIM6 inside CubeMX.

Once that is done you should see that main.c file has defined the function:
`static void MX_TIM6_Init(void)`
which does the timer initialization and configuration.

Note also that file:
Core/Src/stm32l0xx_it.c
Has defined the **`TIM6_DAC_IRQHandler(void)`** ISR that will be called each time to service the generated interrupt:

```
void TIM6_DAC_IRQHandler(void)
{
  HAL_TIM_IRQHandler(&htim6);
}
```

So, when the ISR `TIM6_DAC_IRQHandler`() will be called, the control will be given to `HAL_TIM_IRQHandler`() routine inside the HAL. This will perform all the *interrupt related activities*, and it will call the `HAL_TIM_PeriodElapsedCallback(htim)` routine to which the object `htim` is passed.
If you wanted to see all those *interrupt related activities*, look inside the file:
Drivers/STM32L0xx_HAL_Driver/Src/stm32l0xx_hal_tim.c

Moreover, if we want additional stuff to be done at this time, we can include that inside `HAL_TIM_PeriodElapsedCallback()` routine, which I have defined/described inside main.c. This is left to you to investigate and see what is done inside it.

Please take time to look into and understand the provided code of the **lab5_ex3** example.
Build the project. Program the Nucleo board. Observe operation.

3. Lab Assignment

You must develop an application with the following functionality and requirements. The blue button of the Nucleo board must control the blinking of the green LED such that when the button is pressed, the LED is blinked at a rate of once per second. When the button is pressed again, the LED is blinked at a rate of 10 times per second. And so on. You must use the interrupts from examples 1 or 2 for the button and from example 3 for controlling the LED blinking.

4. References and Credits

[1] Textbook:
Carmine Noviello, Mastering STM32 - Second Edition, 2022, Available to purchase online:
https://leanpub.com/mastering-stm32-2nd <--- Buy from
https://github.com/cnoviello/mastering-stm32-2nd <--- Code examples