Politecnico di Torino

Digital Systems Electronics

04OIHNX - A.A. 2024/2025

Laboratory assignment n°. 9:

Interrupts

Due date: 27/05/2025 Delivery date: 26/05/2025

Group 08 - Contributions:

310779, Simone Viola - 33,3%
300061, Viorel Ionut Bohotici - 33,3%
308299, Daniele Becchero - 33,3%

The members of the group listed above declare under their own responsibility that no part of this document has been copied from other documents and that the associated code is original and has been developed expressly for the assigned project.

Contents

1 Interrupt-Based Variable-Frequency Square Waveform Generator

In microcontroller applications, polling is not always the most efficient method for handling events. The polling approach continuously checks for event conditions, occupying the processor in waiting loops. In contrast, interrupts allow the processor to execute other tasks until a specific event occurs. When triggered, an interrupt temporarily halts the current execution flow to handle the event via a dedicated Interrupt Service Routine (`ISR`).

1.1 Application Description

The objective of this exercise is to replicate the functionality developed in the final project of the previous laboratory session, with one key difference: the toggling of the output pin must now be initiated using an interrupt. Specifically, the `PA10` pin must be toggled explicitly within the `ISR` using an appropriate statement.

1.2 Code Implementation

Starting with STM32CubeIDE, a new project is created. The `PA0` pin is configured as `ADC1_IN0` to sample the voltage from the potentiometer voltage divider using the microcontroller's internal `ADC`. In the ADC1 parameter settings, the resolution is set to 8 bits (11 ADC clock cycles). Timer `TIM3` is enabled, and since no output channel is required, Channel 1 of `TIM3` is configured as "Output Compare No Output". The `TIM3` global interrupt is then enabled. Finally, pin `PA10` is configured as `GPIO_Output` and is used to generate the output square waveform. The code starts by defining two constants, `MIN_FREQ_INC` and `MAX_FREQ_INC`, which set the maximum and the minimum value of timer cycles to generate waveforms with frequency that matches the specifications. For further details on how these values have been obtained, check the last section of the previous lab report. These values are later used to represent a timing interval based on the analog potentiometer voltage in input.

```
#define MIN_FREQ_INC 625
#define MAX_FREQ_INC 111
uint16_t timeInterval;
```

Next up, there's a global variable called `timeInterval`, which will hold that calculated interval. It's defined as a 16 bit unsigned integer to match the dimension of `TIM3_CCR1`, and its value will change depending on analog input applied to the ADC pin `PA0`. Inside the main function, a list of instructions is set in order to ensure the correct working of the code: the timer TIM3 prescaler is configured at 83 (meaning 84 cycles starting from 0). Remember that

$$\text{Timer clock} = \frac{\text{System clock}}{\text{PSC} + 1}$$

The auto-reload register is set to `0xFFFF`, that is equal to 65535 in decimal. When the counter reaches this value, it overflows and resets to zero while rising an IRQ (if enabled).

```
LL_TIM_WriteReg(TIM3, PSC, 83);
LL_TIM_WriteReg(TIM3, ARR, 0xFFFF);
LL_TIM_WriteReg(TIM3, DIER, LL_TIM_ReadReg(TIM3, DIER) | 0b010);
LL_TIM_WriteReg(TIM3, CCER, LL_TIM_ReadReg(TIM3, CCER) | 0x01);
LL_TIM_WriteReg(TIM3, CCR1, 0);
```

With `DIER`, we indicate that the timer has to generate an interrupt when an event occurs in capture/compare Channel 1. With `CCER`, Channel 1 of `TIM3` is enabled to allow it to generate output signals or perform compare operations. The mode of the channel has already been set in CubeMX. The output compare register for channel 1 `CCR1` is set to 0, meaning the compare event will occur immediately at the start of the counting cycles.

Setting bit 0 of the `ADC_CR2` register turns on the ADC, allowing it to begin conversions. Bit 30 of `CR2` is then set, corresponding to the "start conversion" command (`SWSTART` bit). In CubeMX, the conversion mode has been configured as continuous conversion mode. Finally, setting bit 0 of the `TI3_CR1` register (`CEN` or "counter enable") starts the timer, allowing it to begin counting.

```
LL_ADC_WriteReg(ADC1, CR2, LL_ADC_ReadReg(ADC1,CR2) | 0x01);
LL_ADC_WriteReg(ADC1, CR2, LL_ADC_ReadReg(ADC1,CR2) | (1 << 30));
uint8_t voltage;
LL_TIM_WriteReg(TIM3, CR1, LL_TIM_ReadReg(TIM3,CR1) | 0x01);
```

A an infinite loop where the code polls the ADC for conversion complete status. Upon conversion, the 8-bit ADC value is read and mapped to a time interval using linear interpolation. This allows for dynamic adjustment of `timeInterval` based on the analog input.

```
if ((LL_ADC_ReadReg(ADC1,SR) & 0x02) == 0x02) {
 voltage = LL_ADC_ReadReg(ADC1, DR) & 0x00FF; // Get converted value
    // Compute interval
        timeInterval = MIN_FREQ_INC
        + ((MAX_FREQ_INC - MIN_FREQ_INC) * voltage) / 255;
    // Clear EOC (End Of Conversion)
 LL_ADC_WriteReg(ADC1, SR, LL_ADC_ReadReg(ADC1,SR) & (~0x02));
```

The other crucial part of the code is the ISR of `TIM3`, which is located in the file "`stm32f4xx_it.c`". The IRQ handler first checks the flags of `TIM3_SR` to know who requested the interrupt, and executes a set of instructions depending on the interrupt source. In this exercise, the routine checks flag `CC1IR`. If true, the routine toggles `PA10`, updates the register `CCR1` and clears the flag.

```
// Check for CC1IF
if (LL_TIM_IsActiveFlag_CC1(TIM3)) {
    LL_GPIO_WriteReg(GPIOA, ODR,
            LL_GPIO_ReadReg(GPIOA,ODR) ^ (1 << 10));// Toggle PA10
    LL_TIM_WriteReg(TIM3, CCR1,
            LL_TIM_ReadReg(TIM3,CCR1) + timeInterval);// Update CC1R
    LL_TIM_ClearFlag_CC1(TIM3);        // Clear CC1IF
 }
```

1.3 Verification and testing

After compiling and uploading the code to the Nucleo board, we measured the output frequency of the square wave to verify the correct functioning of the overall system. The waveform was observed using an oscilloscope. The results are consistent with the ones obtained with Laboratory 8, exercise 1.5. This indicates that the code works correctly: by adjusting the potentiometer, the waveform frequency changes as expected. By measuring the frequency at the potentiometer's minimum and maximum positions, we achieved a maximum frequency of approximately 4.05 kHz and a minimum frequency of 805 Hz. Observation of the trace of the square wave with the scope shows no noticeable jitter, indicating an output stable both in frequency and amplitude . Additionally, modifying the preemptive interrupt priority revealed that at a level of `0`, the system responds more quickly to changes in the potentiometer position than to higher pre-empitive priority values, which result in lower interrupt priority. This suggests improved responsiveness with lower numerical pre-empitive settings. The pre-empitive priority have been edited in `main.c` code, in the function `MX_TIM3_Init`, by editing the following instruction:

```
/* TIM3 interrupt Init */
  //NVIC_SetPriority(TIM3_IRQn,
  //    NVIC_EncodePriority(NVIC_GetPriorityGrouping(),0, 0)); // PE 0
  NVIC_SetPriority(TIM3_IRQn,
        NVIC_EncodePriority(NVIC_GetPriorityGrouping(),1, 0)); // PE 1
  NVIC_EnableIRQ(TIM3_IRQn);
```

2 Multiple interrupts

In the following projects, unlike to the previous one, multiple tasks on the MCU will be managed using different interrupt service routines (ISRs).

2.1 Three interrupts

2.1.1 Application description

This project implement a 3-bit clock process which produces three square waves having a period of 1 ms, 2 ms and 4 ms. Each wave must be generated starting from `TIM3` relying on OC capability with automatic pin toggling.

2.1.2 Code implementation

In CubeMX, `TIM3` has been configured and three channels have been set as "Output compare Channel x" with "automatic pin toggling" mode. From the GUI, the pin mapping may be observed: Channel 1 of `TIM3` is hardwired to `PA6`, Channel 2 is connected to `PA7` and Channel 3 is driving `PB0`. Global interrupt for `TIM3` has been enabled.The first part of the code is about the configuration of the registers of `TIM3`. Prescaler has been set to 83 to achieve an increment frequency of the timer's counter at a rate of 1 MHz, while ARR has been set to its maximum value. Then, register `DIER` is configured to enable the interrupts of capture/compare channels 1, 2 and 3.

```
LL_TIM_WriteReg(TIM3, PSC, 83);
LL_TIM_WriteReg(TIM3, ARR, 0xFFFF);
LL_TIM_WriteReg(TIM3, DIER, LL_TIM_ReadReg(TIM3,DIER) | 0b01110);
LL_TIM_WriteReg(TIM3, CCER, LL_TIM_ReadReg(TIM3, CCER) | 0x0111);

LL_TIM_WriteReg(TIM3, CCR1, INTERVAL3 -1);
LL_TIM_WriteReg(TIM3, CCR2, INTERVAL2 -1);
LL_TIM_WriteReg(TIM3, CCR3, INTERVAL1 -1);
LL_TIM_WriteReg(TIM3, CR1, LL_TIM_ReadReg(TIM3,CR1) | 0x01);
```

Register `CCER` has been configured to enable the used capture/compared channels and registers `CCRx` have been filled with the initial value specified within a `#define` directive. Last task to do before entering the infinite loop is to enable the timer by set to '1' the LSB of `CR1`. The infinite loop is empty because all the tasks are time-based, and so they are executed within ISRs. All the peripherals interrupt handler functions are located in `stm32f4xx_it.c`; the one useful for this application is `TIM3_IRQHandler`. Within this function, the first thing to do is understand who generates an interrupt request. This operation is done by checking `CCxIF` flags in `TIM3_SR`. Within each conditional statement, two tasks are done: first, register `CCRx` is updated (based on what channel requested the interrupt) and the corresponding interrupt flag is cleared. There is no need to toggle the output pin, since it has already been set to toggle automatically.

2.1.3 Verification and testing

This project had been tested with an oscilloscope. In details:

- Channel 2 of the scope (cyan trace) is connected to `PA6` (CC1)
- Channel 3 (magenta trace) is connected to `PA7` (CC2)
- Channel 4 (purple trace) is connected to `PB0` (CC3)

The three waveforms and their periods are shown in figure 1 and All of them appear stable.

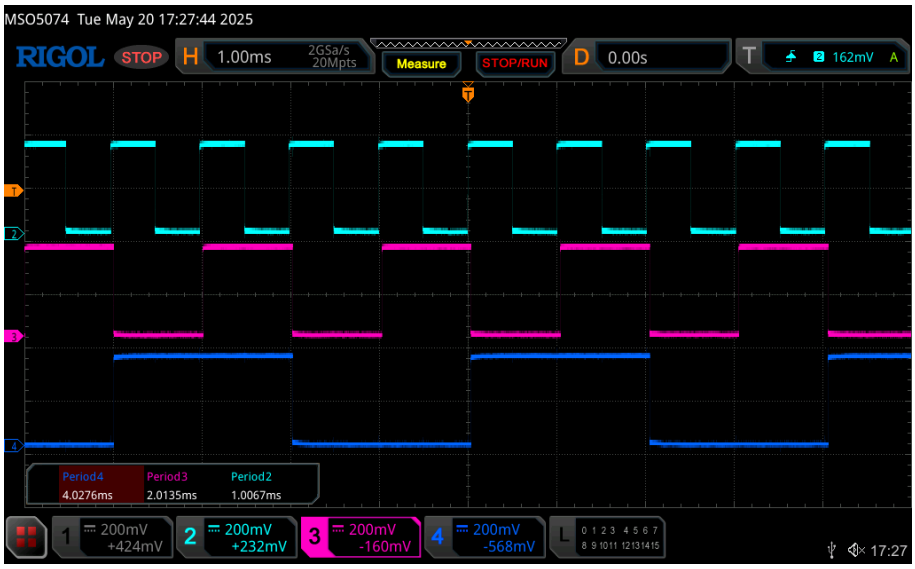

Figure 1: Result of Exercise 2.1

2.2 Four interrupts

2.2.1 Application description

This section extends the features of the previous project by handling another interrupt-based task. The user pushbutton, which is connected to `PA7`, must generate an interrupt request whose ISR toggles the onboard LED.

2.2.2 Code implementation

Even though this project is based on the previous one, STM32CubeIDE does not have a native "clone" feature, so it has to be generated from scratch. The first part of the implementation is equal to the one described in the last section. Within CubeMX, the EXTI line [10:15] interrupt has been enabled in the GPIO window. This modification adds one function in file `stm32f4xx_it.c` that handles the IRQ coming from EXTI line [10:15].

The function `EXTI15_10_IRQHandler` behaves similarly to `TIM3_IRQHandler`: an `if...else` statement is used to determine which pin has sent an interrupt request, and the corresponding ISR is executed if the condition is met.

```
void EXTI15_10_IRQHandler(void) {
 /* USER CODE BEGIN EXTI15_10_IRQn 0 */

 /* USER CODE END EXTI15_10_IRQn 0 */
 if (LL_EXTI_IsActiveFlag_0_31(LL_EXTI_LINE_13) != RESET) {
  LL_EXTI_ClearFlag_0_31(LL_EXTI_LINE_13);
  /* USER CODE BEGIN LL_EXTI_LINE_13 */
        // Toggle PA5
  LL_GPIO_WriteReg(GPIOA,ODR,LL_GPIO_ReadReg(GPIOA,ODR) ^ (1<<5));
  /* USER CODE END LL_EXTI_LINE_13 */
 }
```

This time, the ISR contains only the toggling operation of the LED (which is connected to `PA5`). Notice that the ISR already includes the flag-clearing instruction.

2.2.3 Verification and testing

The testing setup is exactly equal to the previous one, since the other devices are placed onboard. The oscilloscope captures the same waveforms viewed in Figure 1, with almost equal period measurements. In addition, the pushbutton correctly toggles the LED with each press; however, occasional missed toggles occur. This may be due to a rare synchronization issue where another ISR is being handled at the moment the pushbutton is pressed. More likely, the missed toggles could be caused by a user error or mechanical instability known as "bouncing", leading to an inconsistent signal. Anyway, the period of the three outputs is not affected by the pushbutton, meaning that the system and the interrupts are well-managed.

2.3 Five interrupts

2.3.1 Application description

This project extends the functionality of the system described in Section 2.2. It enables control over the output signal periods using a potentiometer within a defined range. Specifically, the selectable period ranges are:

- Channel 1: 0.1 ms – 1 ms
- Channel 2: 0.2 ms – 2 ms
- Channel 3: 0.4 ms – 4 ms

The potentiometer acts as a sort of scaling factor, dividing each period by a value between 1 and 10.

2.3.2 Code implementation

As in the previous section, the first part of the implementation is equal. `TIM3` has been enabled, with clock source set as "internal clock". Channels 1, 2 and 3 have been configured as "Output compare Channel x", with "automatic pin toggling" behaviour. Also `TIM4` has been enabled because it is needed to generate a time-based event every 500 ms. Channel 2 of `TIM4` has been configured as an "Output compare no output", in frozen mode. The global interrupt for both `TIM3` and `TIM4` has been enabled in NVIC settings. Interrupt functionality has been enabled for the user pushbutton as well. Also `ADC1` has been enabled as it is required to convert the analog voltage provided by the potentiometer into a digital value to control the scaling factor. The configuration of the ADC in CubeMX follows the same steps described in Section 1. In addition to that, also global interrupt for `ADC1` has been enabled. Finally, NVIC priority group has been configured as specified in the assignment.

The generated code includes ISR handling functions the different peripherals for which the interrupt has been enabled. These functions are located in the file named "`stm32f4xx_it.c`", under `Src` folder in `Core`. In `main`, before the infinite loop, there is the series of configuring instructions of the peripherals' registers. First part is about configuring registers of `TIM3`, which has not changed since the previous projects (apart from the comments and the location of the define directives, which have been moved to file `main.h`). Then, the same has been done for `TIM4`:

```
LL_TIM_WriteReg(TIM4, PSC, 41999); // Prescaler value
LL_TIM_WriteReg(TIM4, ARR, 0xFFFF); // ARR value

// Enable interrupt for CC2
LL_TIM_WriteReg(TIM4, DIER, LL_TIM_ReadReg(TIM4,DIER) | 0b0100);

// Enable CC2
LL_TIM_WriteReg(TIM4, CCER, LL_TIM_ReadReg(TIM4, CCER) | 0x0010);

// Init CCR2
LL_TIM_WriteReg(TIM4, CCR2, ADCINTERVAL -1);
```

- Prescaler has been set to 41999 (timer clock of 2 kHz.
- Auto-reload value has been set to 0xFFFF.
- Interrupt for CC2 has been enabled.
- Capture/Compare channel 2 has been activated.
- CCR2 register has been initialized with `ADCINTERVAL` - 1 (value of `ADCINTERVAL` has been declared with a define directive in `main.h`; its value is computed using the formula provided in the last laboratory report).

Finally, registers for`ADC1` have to be configured. The peripheral has to be enabled by set `ADON` bit to '1', and EOC interrupt functionality must be activated by setting `EOCIF` to '1'.

The last operations to do before entering the infinite loop is to start the two timers `TIM3` and `TIM4`. The application code relies on an infinite loop which checks for a positive flag to update variable `scalingFactor`. All other instructions are located within ISRs, which are defined in the other source file mentioned above.

The ISR for `TIM3` and for the pushbutton are equal to the ones described in previous sections. The one for `TIM4` has the same structure, but has only one conditional statement because only one interrupt flag has to be checked. If condition is met, the routine updates the output compare register, clears the flag who had requested the interrupt and enable bit `SWSTART` of `ADC_SR` in order to start the conversion. The last ISR is dedicated to the ADC, which generates an interrupt request when the conversion has ended. Within the routine, first we have to check for `EOC` flag to be '1', and if true, the conversion result is stored into a variable and the scaling factor is computed using an arithmetic expression. Before exiting the routine, `EOC` have to be cleared.

2.3.3 Verification and testing

Using the circuit realized in Section 1 and an oscilloscope, the output waveform generated by the Nucleo board has been observed in the time domain. Automatic measurements of the periods are shown in the bottom part of the scope screen captures.

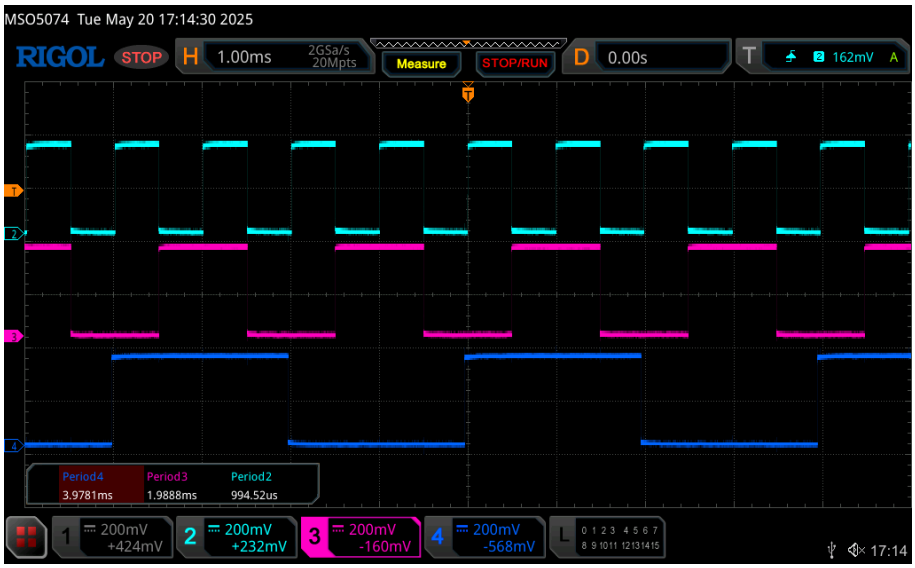

Figure 2: Result of Exercise 2.3, maximum periods selected

By rotating the potentiometer at one end, the original waveforms of Section 2.2 are generated.

Adjusting the potentiometer shaft will not have instantaneous impact on the waveform frequency, but they are modified at a rate of 2 Hz, which corresponds to the ADC sampling frequency. All intermediate periods -within specifications- may be generated and successfully observed with the oscilloscope. When the potentiometer is fully rotated to to its limit, the scaling factor reaches its maximum value of 10, and the observed waveforms are shown in Figure 3.

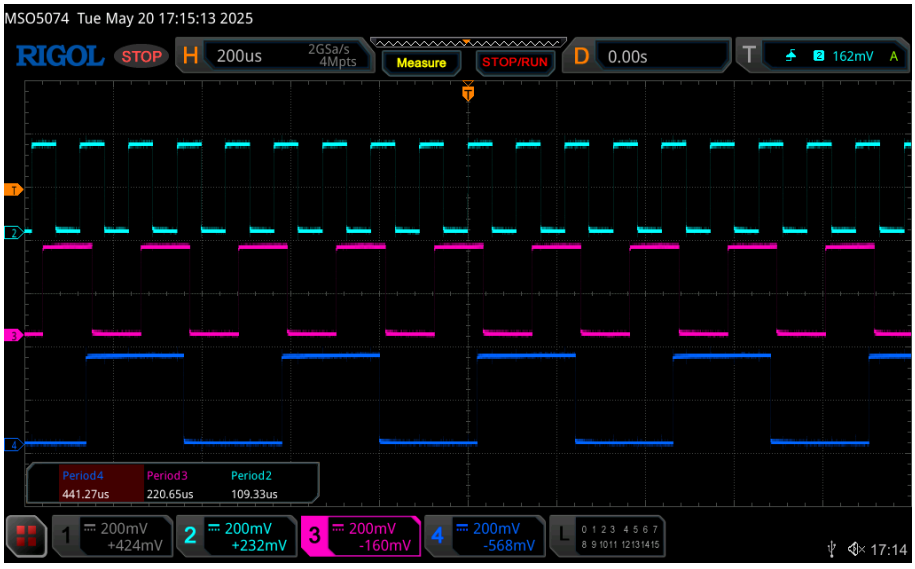

Figure 3: Result of Exercise 2.3, minimum periods selected

The result obtained is consistent with the specifications, and the deviation between the experimental result and the expected value is quite low. This error may be reduced by relying on precision external oscillators, which provides a more stable and precise frequency reference than the internal RC oscillator used in this projects. All the waveforms

appear stable on the oscilloscope, apart from the third which is showing almost random flipping along the horizontal axis, but no jitter in period is present.

The change in the ADC sampling period may be executed by commenting and uncommenting the appropriate line in the `#define` section of "include" file `main.h`. With an ADC sampling time of 100 ms, the overall behaviour of the system has not changed, but the responsiveness had increased.

3 Conclusion

The use of interrupts in MCU applications significantly optimizes computational resources by executing tasks only when necessary, rather than relying on a sequential and resource-intensive polling approach. Additionally, interrupts enhance the system's response time, as an interrupt request is prioritized over standard application code execution. Most interrupts are associated with a programmable priority, ensuring that critical tasks are handled first.

4 Sources and notes

All the projects in this document have been developed from scratch using STM32CubeIDE software, with reference to the STM32 MCU's Reference Manual and User Manual, as well as the Nucleo board's User Manual—all provided by STMicroelectronics. These specific resources can be found on `https://www.st.com/` by searching for the Nucleo-F401RE board and navigating to the documentation section. They are also available within the Digital System Electronics course material.

The members of the group have used GitHub to collaborate in the developing of the delivered codes. For each project, only the essential files have been provided. However, the project does not always work with just these files alone. For exercises requiring the STM32Cube hardware configuration, it is necessary to start a new project in STM32CubeIDE using the delivered .ioc file. After generating the C code, the `main.c` file should be replaced with the delivered version. Only after completing these steps will the project be ready to be built.