Politecnico di Torino

Digital Systems Electronics

04OIHNX - A.A. 2024/2025

Laboratory assignment n°. 7:

LEDs, Pushbuttons and Square Waves

Due date: 13/05/2025 Delivery date: 12/05/2025

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 Program example

The first project is based on an application that uses a low-layer approach to control the built-in LED using the on-board user pushbutton. The application code is provided in the assignment.

A debug session has started, with a breakpoint set at line 30, as specified in the assignment. When execution begins, the code runs until it reaches the instruction just before the breakpoint. At this point, the register values (and thus the device's behavior) are frozen until execution is resumed by selecting one of the available debugging options: "Step Into," "Step Over," or "Resume."

In this example, after selecting "Resume," the code proceeds to toggle a bit in the GPIOA output data register, executes a loop for delay, and then stops again.

1.1 Low-level approach

In this exercise, a program was developed for the STM32 Nucleo board using a low level approach, meaning direct access to the MCU registers without the use of abstract libraries *HAL* or *LL*. The goal is to control the LED (connected to GPIO pin `PA5`) based on the state of the user button (connected to `PC13`), turning the LED on only when the button is pressed.

2 Controlling the LED

2.1 Low-level approach

In this exercise, a program was developed for the STM32 Nucleo board using a low level approach, meaning direct access to the MCU registers without the use of abstract libraries *HAL* or *LL*. The goal is to control the LED (connected to GPIO pin `PA5`) based on the state of the user button (connected to `PC13`), turning the LED on only when the button is pressed.

2.1.1 Application description

This approach requires a detailed understanding of GPIO register mapping, pin configuration as input/output, and managing the AHB1 bus clock to enable peripherals.

The code has been written in C, without external library dependencies, and loaded directly on the MCU using the ST-link interface integrated into the Nucleo board.

2.1.2 Code implementation

At the beginning of the program, a series of pointers to memory-mapped registers are declared. They are directly used to control the behavior of the GPIO which are General Purpose Input/Output ports.

```
//PORT REGISTERS
volatile unsigned int *GPIOA_MODER =
                            (unsigned int*) (0x40020000 + 0x00);
volatile unsigned int *GPIOC_MODER =
                            (unsigned int*) (0x40020800 + 0x00);
volatile unsigned int *GPIOA_ODR =
                            (unsigned int*) (0x40020000 + 0x14);
volatile unsigned int *GPIOC_IDR =
                            (unsigned int*) (0x40020800 + 0x10);
```

In order for the GPIO peripherals of the STM32 microcontroller to operate, it is necessary to enable the clock signal associated with them. By default, all peripheral clocks are disabled to save power and resources. The clock for each peripheral is activated through the RCC (Reset and Clock Control) module, specifically by writing to the `AHB1ENR` register, which controls the clocks for all peripherals connected to the AHB1 bus, including the GPIO ports.

```
   //CLOCK REGISTERS
volatile unsigned int *RCC_AHB1ENR =
                            (unsigned int*) (0x40023800 + 0x30);
   //ENABLE PORT CLOCK
*RCC_AHB1ENR |= 0x05U;
```

Once the clock has been enabled for GPIO ports, the I/O pins have to be configured. `GPIOx_MODER` is a 32-bit register used to configure the behavior of the corresponding pins in `PORTx`. Each pin is controlled by two bits, determining its mode. In this case, `PA5` (pin 5 of `PORTA`) must be configured as an output GPIO, while `PC13` (pin 13 of `PORTC`) must be set as an input GPIO. The configuration is applied by writing the appropriate value to the designated bit positions within the register, using masks to ensure that no unintended bits are modified.

```
//CONFIGURE PORT: set MODER[11:10] = 0x1
*GPIOA_MODER = *GPIOA_MODER | 0x400;
*GPIOC_MODER = *GPIOC_MODER | 0x000;
```

`GPIOx_ODR`, refers to the Output Data Register of port x; the address for each port's ODR may be found in the reference manual. Writing a 1 to a specific bit in this register sets the corresponding GPIO pin to logic high, enabling control of output devices such as LED.

The second register, `GPIOx_ODR`, refers to the Input Data Register of port x. Reading this register allows the program to detect the digital input state of each pin, for example reading the status of a pushbutton connected to `PC13`.

In the code, the pointers for the needed registers are declared as volatile to inform the compiler that the values at these memory addresses can change at any time and also to tell to the compiler to not simplify the code.

Before entering the main infinite loop, the program includes an instruction intended to initialize the state of the LED connected to pin `PC13`. This instruction is not manda-

tory, since at reset (or power-up) the content of each register is its reset value, and for `GPIOx_ODR`, the reset value is `0x00000000`. The infinite loop is shown below.

```
// If button is not pressed (USER BUTTON is active-low)
if ((*GPIOC_IDR & 0x02000) != 0x00) {
    *GPIOA_ODR = *GPIOA_ODR & 0xFFFFFFDF;// Set bit in pos 5 at 0
    }
    else {
    *GPIOA_ODR = *GPIOA_ODR | 0x020;     // Set bit in pos 5 at 1
    }
```

During each loop cycle, the state of the user pushbutton is monitored, and the LED output state is updated accordingly. The `GPIOC_IDR` register is read to obtain the current logic level on pin `PC13`, where the user pushbutton is connected. Pressing the button pulls `PC13` to logic 0, as the pushbutton is active-low.

To process this input, the register value is first masked to ensure that only the relevant bits are considered, then it is compared to the expected value within an `if...else` statement. If the condition evaluates to true (indicating the button is pressed), the LED on `PA5` is turned on by setting bit 5 of the `GPIOA_ODR` register to 1.

Subsequently, the code updates the `GPIOA_ODR` register to assign the correct logic level to bit position 5, based on the state of `GPIOC_IDR` at bit position 13. Two different masks are applied to ensure that only the intended bit in the register is modified.

2.1.3 Verification and testing

The STM32 Nucleo board supports debugging in STM32CubeIDE via the Serial Wire Debug (SWD) interface, allowing users to place multiple breakpoints in the code. During debugging, the program runs until it reaches a breakpoint, at which point execution halts. The IDE then displays the contents of internal registers and the current values of variables, enabling the user to verify the program's behavior and identify any errors in its functionality. Execution can then be resumed in several ways: the program may continue until the next breakpoint, step into a function for detailed analysis, or step over a function to proceed without entering it.

In the initial version, the pushbutton was assumed to be active-high, causing the code to function opposite to expectations. As a result, adjustments were required. The revised application code has been successfully loaded onto the MCU, and a debugging session has been initiated in the IDE. The overall functionality meets the specifications, so the code has been built and programmed into the MCU for real-time testing on the Nucleo board.

2.2 STM32Cube approach

The objective of this exercise is the same as the previous one, but the result has to be obtained using a modern and modular approach by employing the STM32CubeIDE environment and the LL (Low Layer) libraries provided by the STM32 manufacturer.

2.2.1 Application description

Since the objective has not changed from the previous project, the application is described properly in 2.1.

2.2.2 Code implementation

The project was created using a different approach, leveraging the graphical configuration tool to set up the required peripherals. Through STM32CubeMX, pin `PA5` was configured as a digital output, while pin `PC13` was set as a digital input. All other peripherals were left in their default mode.

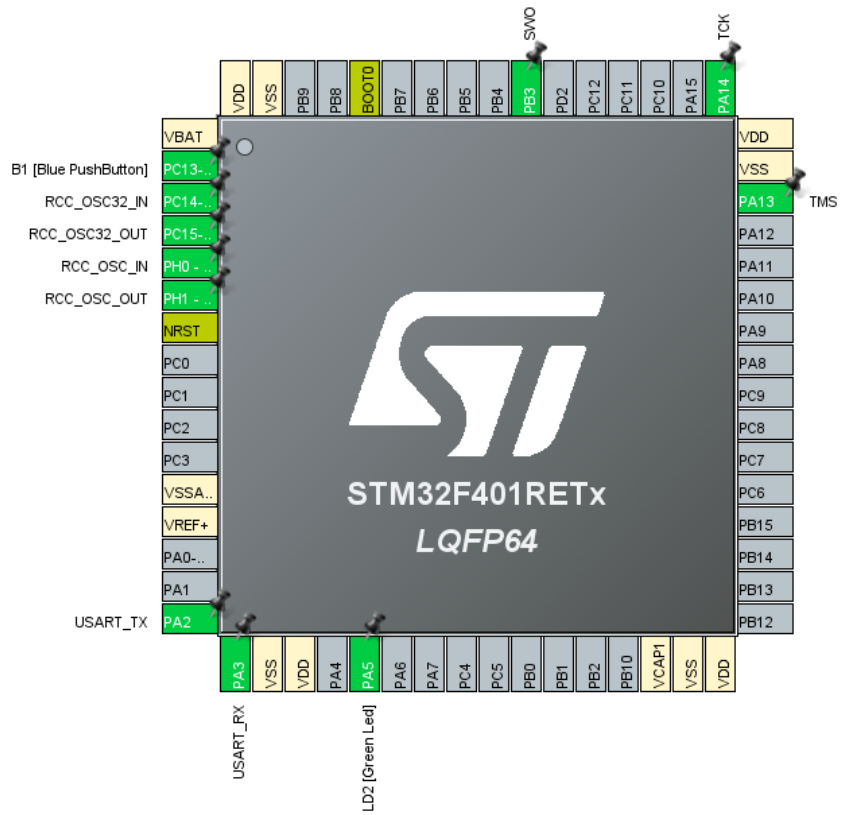

Figure 1: STM32-F401RE (just the MCU IC) pinout view, STM32CubeMX perspective

The automatically generated code initializes all configured peripherals, enables the necessary clock sources, and provides the required LL drivers—implemented as macros—for interacting with them. For GPIO peripherals, two key macros are used: `LL_GPIO_WriteReg` and `LL_GPIO_ReadReg`. These macros are described in detail in the user manual and the LAB7_STM32_document.pdf included in the course material.

The user must insert their code only between the designated commented lines in the generated `main.c` file. Any modification to the `.ioc` file, followed by the regeneration of the C code, will overwrite all content outside these commented sections, replacing it with newly generated code.

The application code in the infinite loop has the same structure of the previous exercise. The main difference is in the use of LL functions to read and write the GPIO registers instead of dealing with pointers.

```
// If button is not pressed (USER BUTTON is active-low)
if ((LL_GPIO_ReadReg(GPIOC, IDR) & 0x02000) != 0x00) {
    LL_GPIO_WriteReg(GPIOA, ODR,
            (LL_GPIO_ReadReg(GPIOA, ODR) & 0xFFFFFFDF));
  }
  else{
    LL_GPIO_WriteReg(GPIOA, ODR,
            (LL_GPIO_ReadReg(GPIOA, ODR) | 0x020));
  }
```

2.2.3 Verification and testing

Following the same steps described in the previous section, the overall project has been tested and debugged with success.

2.3 Varying the blinking frequency

2.3.1 Application description

This project builds upon the functionality developed in the previous section. Initially, the LED blinks at approximately 0.25 Hz, and with each press of the pushbutton, the blinking frequency doubles.

The application utilizes the LED `LD2` on the `NUCLEO-F401RE` development board, which is connected to the I/O pin `PA5` of the STM32 microcontroller. Additionally, the user pushbutton `B1`, connected to the I/O pin `PC13`, serves as the input trigger for adjusting the blinking frequency.

2.3.2 Code implementation

First, an STM32 project was created in `STM32CubeIDE` using the board selector to specify the board name. The hardware configuration was then completed selecting `STM32Cube` and opening the `STM32CubeMX` perspective, where the microcontroller pinout, the clock tree, and peripherals driver selection (HAL or LL) were defined according to the assignment. All peripherals remained in their default mode, utilizing only `LD2` and `USER BUTTON` as GPIOs, while the clock tree was left unchanged. The driver settings were explicitly configured to LL. Once the configuration was finalized, the code was generated using the appropriate command in the IDE.

The command systematically generates multiple files and folders, organizing the project structure. Within the `Core` folder, two subdirectories are created: `Inc`, which contains all header files, and `Src`, which includes the source files. Each generated file follows a structured format and is enriched with detailed comments indicating where specific parts of the code should be placed within each subsection. The `main.c` file serves as the central location for integrating the application code. After all the initialization commands generated by the tool, the output register of `GPIOA` was set to `0x00`. Although the reset value of this register is `0x00000000`, explicitly assigning `0x00` ensures that the register is forced to this value.

Following this, a new variable named `timeInterval` was initialized with the value $20,000,000$, determined empirically to achieve an initial blinking frequency of 0.25 Hz.

During each cycle of the infinite loop, the microcontroller reads the status of the pushbutton by accessing the value in ‘`GPIOC_IDR`‘. If the pushbutton is pressed, the ‘`timeInterval`‘ variable is shifted one position to the right, effectively performing a division by 2 in binary arithmetic. If the pushbutton is not pressed, the variable remains unchanged.

The second part of the loop manages the blinking operation, utilizing a delay mechanism implemented through an empty `for` loop. This loop runs for approximately half of the blinking period, after which the bit corresponding to the pin connected to the LED is toggled. The application code is shown below. However, since the application does not employ timers or interrupts, its performance is very limited and inaccurate.

To begin with, the initial time interval lacks precision, as accurately determining the execution time of each instruction is very challenging and out of the scope of this laboratory. Additionally, the polling of the pushbutton status occurs immediately after the

toggle operation, making the frequency-doubling process less responsive at lower frequencies. A more efficient system could be designed using timers and interrupts to enhance both frequency accuracy and responsiveness.

```
// If button is pressed (USER BUTTON is active-low)
if ((LL_GPIO_ReadReg(GPIOC, IDR) & 0x02000) == 0x00) {
    timeInterval = timeInterval >> 1; // Decrease the time-interval
    }
for (int i = 0; i < (timeInterval / 2); i++); // Delay statement
LL_GPIO_WriteReg(GPIOA, ODR,
    (LL_GPIO_ReadReg(GPIOA, ODR) ^ 0x020));
```

2.3.3 Verification and testing

For this project, the development effort was significantly greater because we needed to achieve the initial blinking operation at a roughly tuned value. This required determining an appropriate value for the delay statement. However, the name `timeInterval` may lead to misunderstanding, as it represents a sort of number of cycles to wait rather than an actual time duration. Specifically, it represents to the number of iterations of an empty `for` loop to realize a delay of approximately 2 s—half of the blinking period, which is 4 s. The testing and debugging processes have been successful. The threshold at which a noticeable change in frequency occurs is approximately 64 Hz. Beyond this value, the button press duration exceeds the toggling time interval, resulting in a very rapid increase in frequency.

3 Generating a square wave

3.1 Application description

The approach used in the previous section to blink a LED can be similarly applied to generate a square wave on a GPIO pin. This section aims to implement a C program that produces a 1 kHz square wave using the D2 pin on the NUCLEO-F401RE development board.

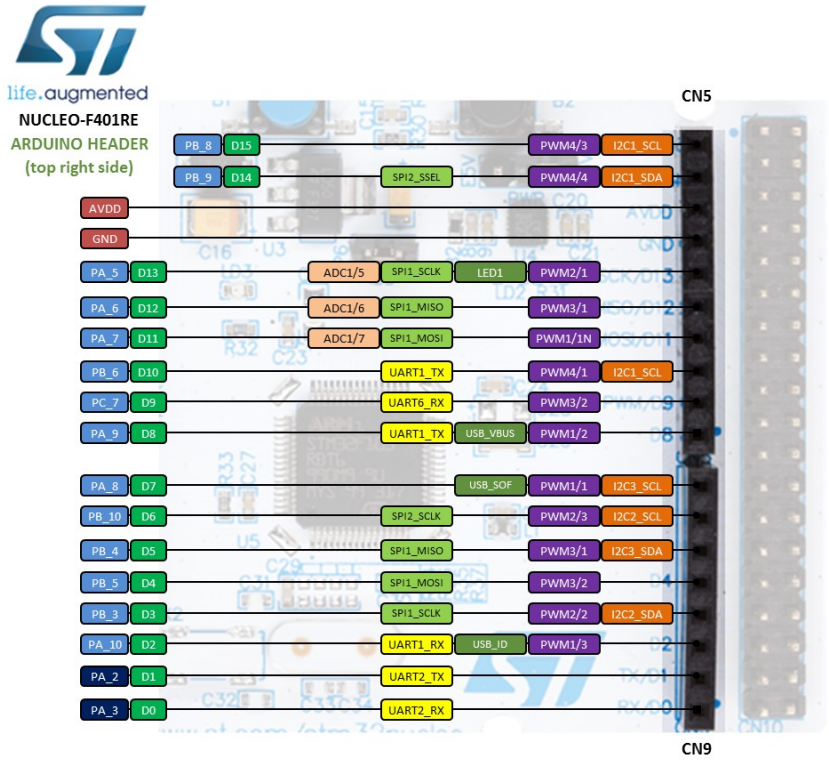

Figure 2: NUCLEO-F401RE Arduino header (right) connections. Property of arm Mbed.

3.2 Code implementation

As suggested by the assignment, also in this section, a new project was created using STM32CubeIDE, selecting the appropriate configuration for the D2 pin. We then proceeded with the hardware configuration using STM32CubeMX and analyzed the generated code.
First, a variable named `INTERVAL` was initialized using `#define` with a random initial value of `1000`. Initializing this variable will be very useful later on, when we need to change this value multiple times in order to achieve the desired frequency.
The laboratory assignment provides the line:

```
SysTick_Config(SystemCoreClock / 1000)
```

which is intended to be placed just before the `while(1)` infinite loop. This function sets up the SysTick timer to generate an interrupt every 1 millisecond. Within the stm32f4xx_it.c file, the interrupt handler function includes the following code:

```
static int x = 0x12C;
for (int i = 0; i < x; i++);
x = (x >> 2) | (((x & 1) ^ (x & 2)) << 4);
```

This additional code is designed to emulate a more realistic workload on the microcontroller. However, the specific purpose and behavior of this code will be analyzed in future reports. Fow now, this section of code can be enabled or disabled (by commenting or uncommenting) to compare the behavior and performance of the system with and without the simulated application load. The results of the comparison are discussed in the Verification and testing section.
Since the desired frequency will be obtained after several tests and adjustments to the delay, we initialized a variable named `INTERVAL` with a random initial value. This approach is convenient because it allows for quicker modifications of the delay value in order to find the desired frequency.

```
#include <stm32f4xx_ll_bus.h>
#include <system_stm32f4xx.h>

#define INTERVAL 8500
```

As done in the previous exercises, to blink the LED connected to PA10, we started by modifying the code generated by STM32Cube within the infinite loop `while(1)`, writing a `for` loop parameterized by the variable `INTERVAL`, which had been previously initialized to create a delay in each cycle.

Next, using the function `LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SYSCFG)`, we enable the clock for the `SYSCFG` peripheral, which is located on the APB2 bus. Similarly, with the function `LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_PWR)`, we enable the clock for the `PWR` peripheral, which is located on the APB1 bus.

```
int main(void){

  LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SYSCFG);
  LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_PWR);

  NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);

  SystemClock_Config();

  MX_GPIO_Init();
  MX_USART2_UART_Init();
```

```
  int i;

 SysTick_Config(SystemCoreClock / 1000);

 while (1) {
  for (i = 0; i < (INTERVAL / 2); i++);
  LL_GPIO_WriteReg(GPIOA, ODR, LL_GPIO_ReadReg(GPIOA, ODR) ^ 0x0400);
 }
}
```

Then, using the function `LL_GPIO_WriteReg(GPIOA, ODR, LL_GPIO_ReadReg(GPIOA, ODR) ^ 0x0400)`, we write to the `ODR` (Output Data Register) of GPIOA by reading its current value and applying a bitwise XOR with the value `0x0400`. The operator "`^`" performs a bitwise XOR, and the hexadecimal value `0x0400` (which is `0000 0100 0000 0000` in binary) has bit 10 set, which corresponds to pin `PA10`. This operation toggles the state of the LED, since applying XOR with 1 inverts the bit's value.

3.3 Verification and testing

The verification and testing phase involves using an oscilloscope connected via a probe to the `PA10` pin to observe and measure the output frequency. This allows for fine-tuning by adjusting the value of the `INTERVAL` variable. As shown in the figure 3 below, the output clock exhibits a correct waveform with a relatively stable frequency of approximately 1.01 kHz.

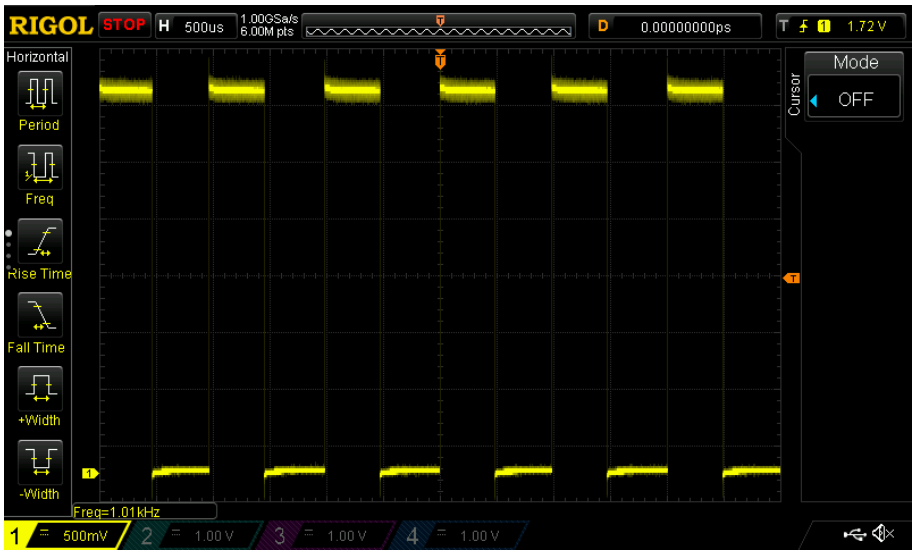

Figure 3: Oscilloscope view of the generated square wave

The waveform displayed on the oscilloscope appears stable, with no strange or unexpected behavior. Then, we have tried modifying the main code by commenting out the line `SysTick_Config(SystemCoreClock / 1000)` placed just before the while loop, no significant changes were observed in the stability or frequency of the generated waveform, contrary to expectations.

4 Conclusion

Throughout this lab, the configuration of the peripherals' registers via pointers has provided valuable hands-on experience, emphasizing the significance of understanding the physical addresses of the involved registers and their meaning for the behaviour of the system. Moreover, the use of LL drivers has proven effective in maintaining strong control over hardware resources while significantly improving code readability.

5 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.