Software Design for Embedded Systems
Laboratory exercise 1
GPIO and TIM peripherals

November 9, 2022

Contents

1 Required hardware, software and additional literature

For this laboratory exercise, you will need the following:
Hardware:

- STM32F4DISCOVERY development kit,
- USB-mini cable for power supply and programming.

Software:

- cross compiler for STM32F4 (see subsection 2.1),
- ST-link for flashing binary to hardware (see subsection 2.2),
- STM32CubeMX [1] (see subsection 2.5).
  - Graphical tool for configuration of STM32 microcontrollers and initialization code generation

Literature and materials:

- user guide covering the microcontroller family [2],
- datasheet for specific microcontroller part [3],
- programming manual [4],
- description of STM32F4 HAL and low-layer drivers [5].

Additional literature includes the following:

- STM32F4DISCOVERY board user manual (*UM1472 - Discovery kit for STM32F407.pdf*),
- STM32F4DISCOVERY board training materials (*STMicroelectronics Cortex-M4 Training STM32F407.pdf*),
- STM32CubeMX user manual (*STM32CubeMX User manual.pdf*).

Additional materials for this lab exercise are packed conveniently in the archive *Lab1_Pack.zip*, available from FER web page, under the section *Laboratory exercises*. This lab guideline will address above references at some points and it is advisable to read referenced parts of the literature whenever further clarifications and instructions are needed.

2 Setting up development environment

2.1 Setting up cross-compiler for STM32F4

2.1.1 Linux OS

To compile C source code for STM32F4 microcontrollers on our PCs, we need a cross-compiler for ARM Cortex-M processors. The `gcc-arm-none-eabi` cross-compiler can be installed from the package manager,

[1] Available from ST webpage
[2] STM32F4 Reference manual.pdf
[3] STM32F407VG.pdf
[4] STM32F3 and STM32F4 Series Cortex-M4 programming manual.pdf.
[5] UM1725 - STM32F4 HAL Manual.pdf

using the following command:

```
$ sudo apt install gcc-arm-none-eabi
```

2.1.2 Windows OS

Prebuilt arm-gcc toolchain for Windows can be downloaded directly from ARM webpage. It is recommended to fetch the last build, for example:

```
gcc-arm-none-eabi-9-2020-q2-update-win32.exe
Windows 32-bit Installer (Signed for Windows 10 and later) (Formerly SHA2 signed binary)
MD5: 62d2b385da1550d431c9148c6e06bd44
```

Note: Legacy prebuilt toolchain versions are available here. However, this source is not recommended since later builds are managed directly by ARM, as denoted at ARM's webpage "Recent releases are available on this page. You can download older releases from Launchpad, and view a timeline of older releases on Launchpad").

Run the installer. Toolchain will install in the following default target folder e.g.:

```
C:\Program Files (x86)\GNU Arm Embedded Toolchain\9 2020-q2-update
```

Notes:

- Check the *Add path to environment variable* option before you click the *Finish* button for the installation,
- You may select a different folder but the installation instructions use the default values.

2.2 Setting up flashing tool

ST-Link is an open-source software interface manufactured by STMicroelectronics for programming STM32 microcontrollers. The tool offers a wide range of features to program STM32 internal memories (Flash, RAM and others), external memories, to verify the programming content and to automate STM32 programming. ST-Link utility is delivered as a graphical user interface and a command line interface. The following text gives an overview of the ST-Link commands under Linux OS.

The `st-info` [6] command provides information about connected STM32 devices. For example, if we connect our STM32F407 board, we can determine the summarized information about the device (e.g. size of flash/SRAM memory, chip ID) by running the following command:

```
$st-info --probe
```

Flashing binary file to STM32 device is done by `write` option. For example, the command:

[6] http://manpages.ubuntu.com/manpages/cosmic/man1/st-info.1.html

```
$ st-flash write firmware.bin 0x8000000
```

writes the `firmware.bin` binary file to flash memory at address `0x8000000`.

The `read` option is used for reading firmware from device. The command:

```
$ st-flash read firmware.bin 0x8000000 0x1000
```

reads 4096 bytes from flash memory, starting at address `0x8000000`.

If it is necessary to erase entire firmware from the device, the `erase` command is used:

```
$ st-flash erase
```

2.2.1 Linux OS

For setting up ST-Link, we need to install dependencies and build it from sources. For building sources, the following tools are required:

- `cmake` - software tool for building ST-Link source files,
- `udev` - Linux subsystem for device management,
- `build-essential` - package which includes GCC compiler, libraries and other utilities,
- `libusb-1.0-0-dev` - cross-platform library for access to USB devices.

All of the required tools can be installed by running the following command:

```
$ sudo apt install cmake udev build-essential libusb-1.0-0-dev
```

We shall build ST-Link tool from sources which are available from ST-Link GitHub page. For our laboratory exercises, we are using a fork [7]. of the original repository.

The first step is to download ST-Link sources from GitHub:

```
$ git clone https://github.com/OPPURS-laboratory-exercises/stlink
$ cd stlink
```

Next, we shall build the *Release* version of ST-Link tools by running:

```
$ make release
```

By running the command above, we invoked the top-level *Makefile* which created a new directory `build/Release` and set the `cmake` environment for building the binaries. The binaries are built automatically beacuse the top-level *Makefile* recursively runs the *Makefile* in the `build/Release` directory.

[7] A copy of an original Git repository in a separate namespace

We can now copy the binary files to `/usr/local/bin/` directory. This will allow us to simply run ST-Link tools as commands, without writing the full path to the local binary:

```
$ cd build/Release
$ sudo cp bin/st-* /usr/local/bin
```

This command copies all files prefixed with `st-` (ST-Link tools) to `/usr/local/bin`, which is a common directory for placing user-installed software.

The next step is to install `udev` rules. Udev is the device manager for Linux kernel and it is used for dynamically creating or removing device nodes. The `udev` rules determine how to identify devices and match the configured rules to corresponding device upon a device event (e.g. connecting a USB device). Custom `udev` rules are located in the `/etc/udev/rules.d` directory.

The `udev` rules provided by ST-Link are used for managing access to USB device and defining the adequate permissions.

We shall install `udev` rules by copying the rules provided by ST-Link to custom rules directory:

```
$ sudo cp config/udev/rules.d/49-stlinkv* /etc/udev/rules.d/
```

Since `udev` rules are not re-triggered automatically on already existing devices, it might be necessary to reload the rules:

```
$ sudo udevadm control --reload
```

2.2.2 Windows OS

Download STM32 ST-LINK flash programming utility from ST website and follow installation instructions.

Although STM32 ST-LINK flash programming utility offers intuitive and easy to use GUI interface for flashing the microcontroller, it is often convenient to use command line interface (CLI) version of utility. Guidelines for using CLI flash programming utility can be found here.

Here is a simple example of flashing microcontroller using CLI:

```
st-link_cli -c SWD UR -P "main.bin" 0x08000000 -Rst -Run
```

where the parts of CLI command are:

- `st-link_cli` - ST-Link command line utility executable `C:\..\st-link\_cli.exe` (provide full path or just a filename if installation path is included in `PATH` system environmental library),
- `-c SWD UR` – connection interface definition - Serial Wire Debug (SWD) (JTAG is another option); connect under reset (UR),
- `"main.bin"` – last built binary file,
- `0x8000000` – start address in FLASH memory where binary must be programmed to (FLASH memory starts at this address),

– `-Rst` – reset target after programming,

– `-Run` – run newly loaded program.

2.3 Setting up GNU Make environment

2.3.1 Linux OS

The GNU Make tool is included in the `build-essential` package and was installed by steps described in subsection 2.2.

2.3.2 Windows OS

GNU make utility for Windows is available here. The prebuilt binary installer is available here. Run the installer, default installation folder is as follows:

```
C:\Program Files (x86)\GnuWin32
```

Add installation folder to `PATH` environmental variable for easier access from Windows CLI system-wide. **Note:** When you run make from command line, the messages will not be in English for OS with different locale. To force English command line interface it is sufficient to rename or delete locale folder in installation path:

```
C:\Program Files (x86)\GnuWin32\share\locale
```

Important: Please refer to Appendix Appendix A for information about GNU Make tool and a simple example of writing Makefiles.

2.4 Running a simple example

Clone example from github repository (example of some working blinky implementation):

```
git clone https://github.com/OPPURS-laboratory-exercises/STM32F407_test_example.git
```

Copy everything under stm32f4-discovery folder to some location. In root folder there is a Makefile with all necessary information needed to build example.

2.4.1 Linux OS

Change working directory to the project root directory. Connect the STM32F4DISCOVERY board to computer and run the following commands:

```
$ make
$ /opt/stlink/st-flash write firmware.bin 0x8000000
```

In this case, it is assumed that ST-Link is installed in `/opt/stlink` directory. If that is not the case, determine the path to ST-Link by running:

```
$ whereis st-flash
```

2.4.2 Windows OS

Assuming that ARM GCC toolchain and Make utility are accessible via `PATH` environmental variable, run make from command line (make sure that the current working directory is the one that contains the Makefile):

```
$ cd STM32F407_test_example\stm32vl-discovery
$ make
```

The result of compilation are output files ready for flash programming: `main.bin`, `main.hex`. Either of the files can be used for flash programming. Connect STM32F4DISCOVERY board to computer. If you are using ST-Link CLI utility, follow the instructions given in the previous subsection. If you are using the ST-Link GUI, click on *Target-Connect* to check connection with target board. Run *Target-Program*, select generated hex or bin file, run *Start* to program the development board.

2.5 Setting up STM32CubeMX

STM32CubeMX is a tool for generating the hardware abstraction layer for STM32F4 family. This layer is vendor-specific (STMicroelectronics) and provides high-level access to microcontroller resources and peripherals without a need for individual register access programming.

2.5.1 Installation

The installation package must be downloaded from ST webpage. Extract the `stm32cubemx.zip` to some directory.

To make the installation file executable, we must change the file permissions. Run the following command:

```
$ sudo chmod 777 SetupSTM32CubeMX-5.0.0.linux
```

Now the file can be run, which will launch the installation wizard:

```
$ ./SetupSTM32CubeMX-5.0.0.linux
```

To run STM32CubeMX application, launch the executable from the installation directory.

2.5.2 Installing software packages for STM32F4

Before creating a STM32CubeMX project, the software packages for the target microcontroller must be installed. The available packages can be accessed under the *Embedded Software Packages Manager*. In

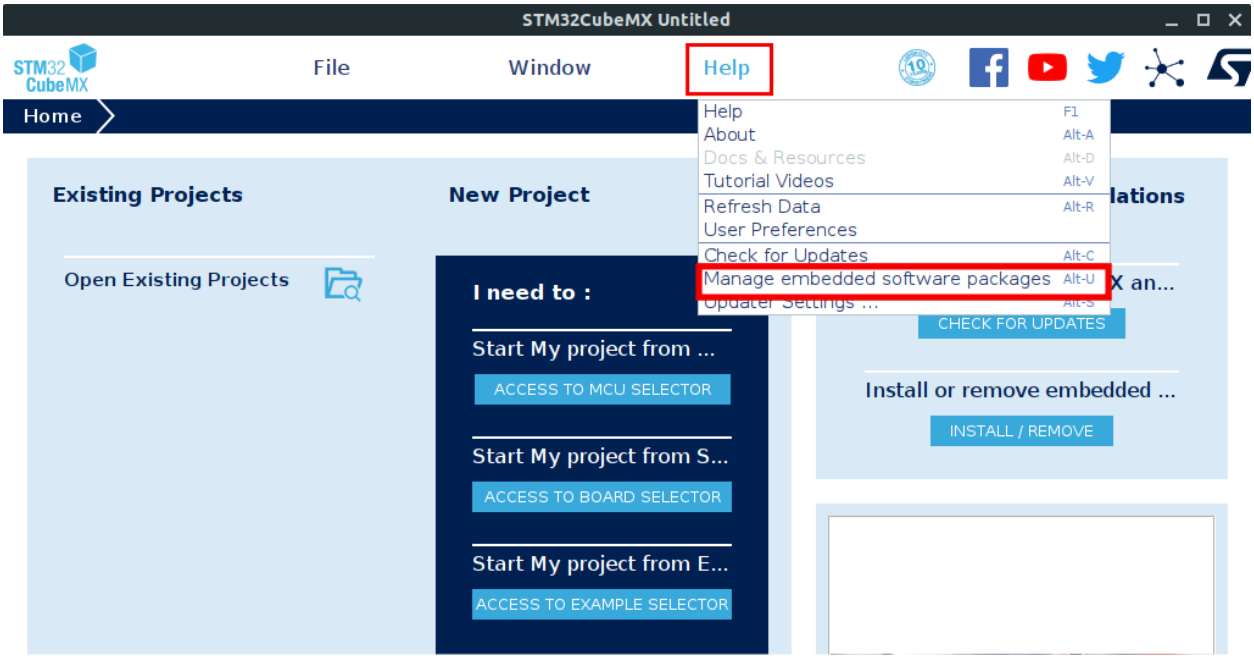

Figure 1: Running the STM32CubeMX packages manager.

STM32CubeMX application, select *Help* - *Manage embedded software packages* (see Fig. 2).
Expand the packages under STM32F4 entry and select the newest version. The listed packages consist of library packages and library patches. The patches are not complete libraries and they consist of updated library files. The patched files go on top of the original package, e.g. STM32Cube_FW_F7_1.25.1 complements STM32Cube_FW_F7_1.25.0 package. Start the download by clicking *Install Now*.

3 Creating a new project

3.1 Configuring the MCU and generating initialization code by STM32CubeMX

In STM32CubeMX application, create new project by clicking *File* - *New Project*. In the list of available MCUs, select STM32F407VG, as shown in Fig. 3.

Once the target board is selected, the project page opens, showing the following set of views:

- Pinout & Configuration,
- Clock Configuration,
- Project Manager,
- Tools.

In the Project Manager tab, configure the project settings as shown in Fig. 4.

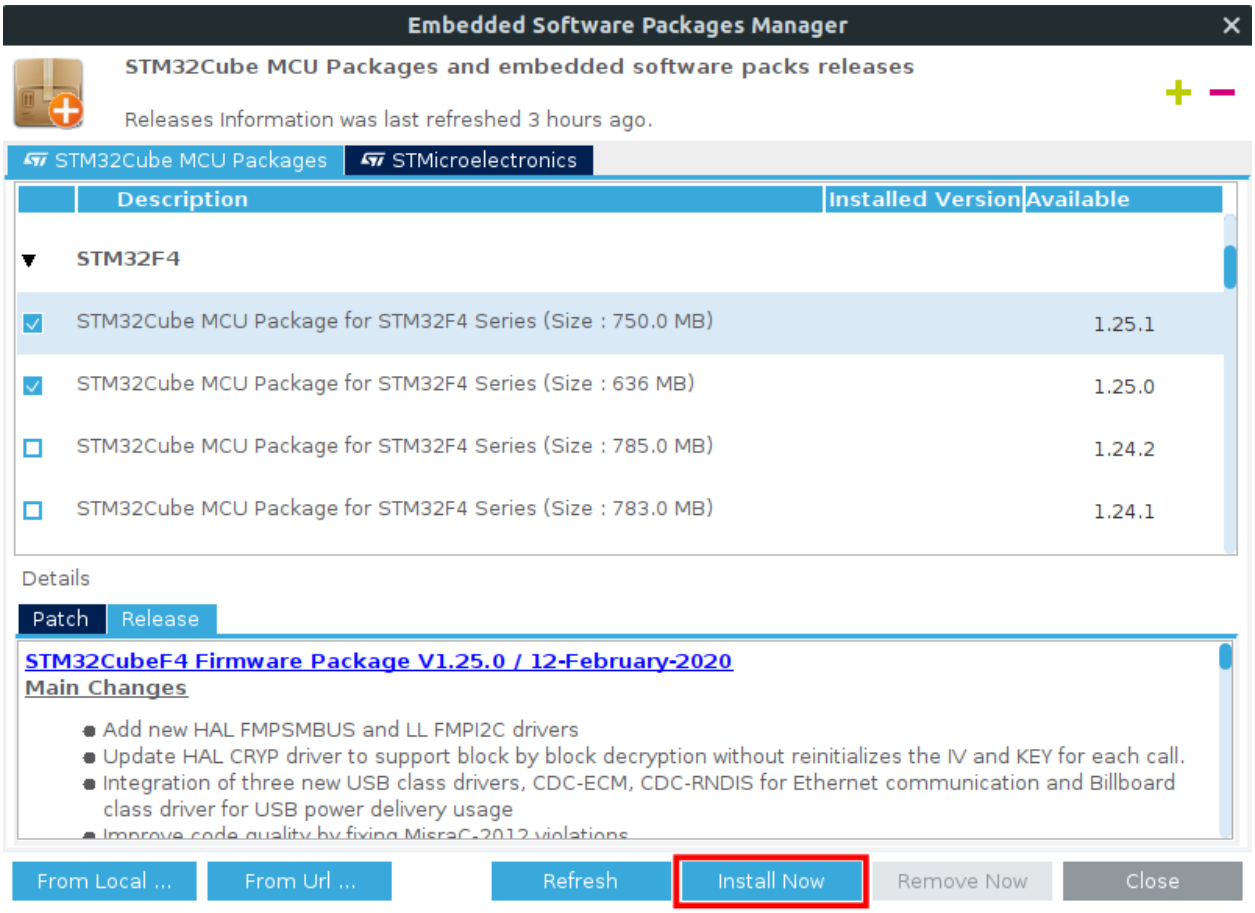

Figure 2: Selecting the software package.

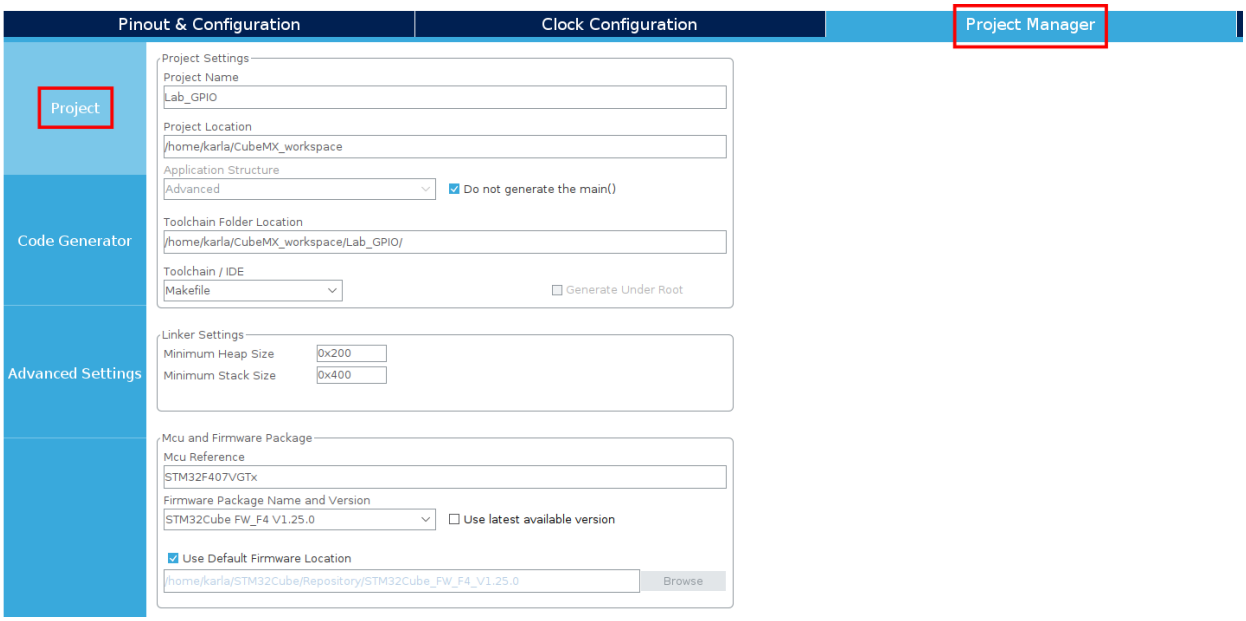

Figure 4: Project settings in STM32CubeMX.

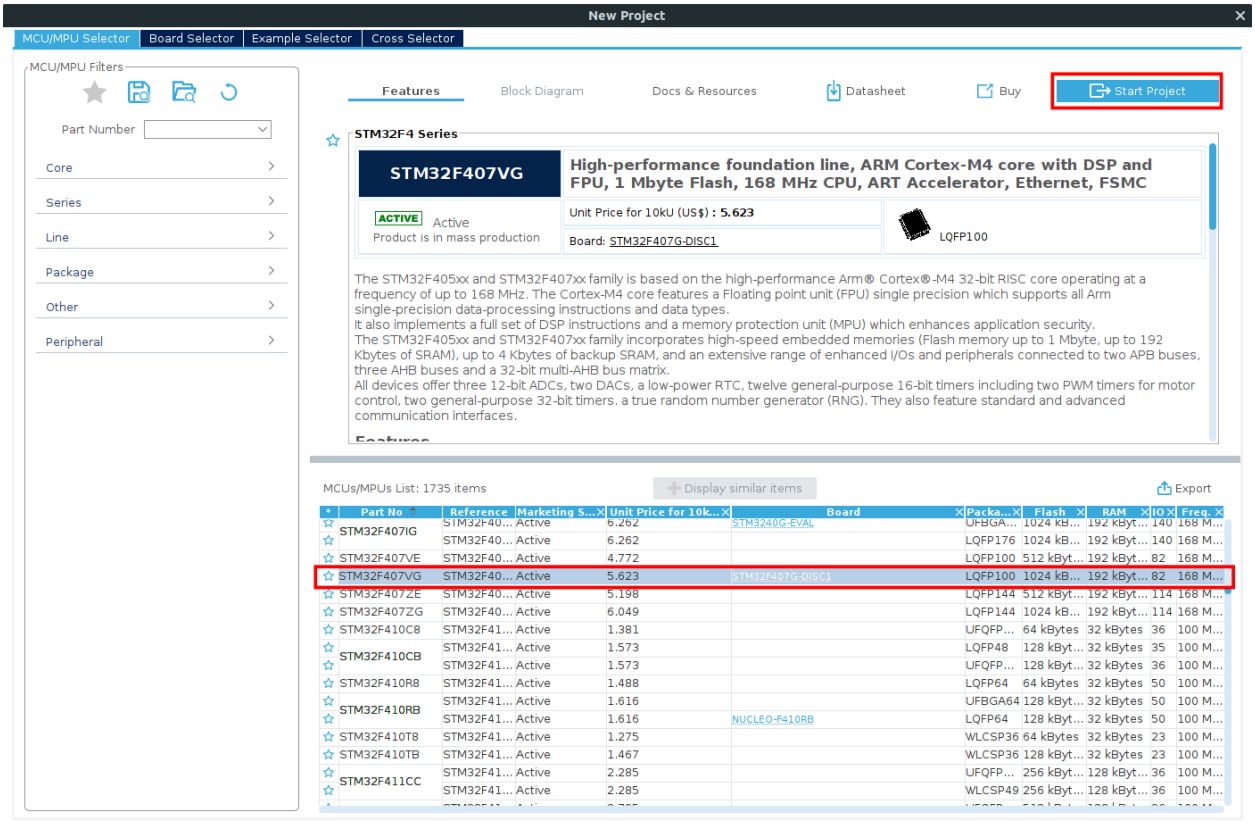

Figure 3: Selecting the MCU.

Select the project name, e.g. *Lab_GPIO* and set the project location to your STM32CubeMX workspace. Select *Makefile* as Toolchain/IDE. Turn on the option *Do not generate the main()*.

Click on *Code Generator* tab and turn on the *Generate peripheral initialization as a pair of '.c/.h' files* option (see Fig. 5).

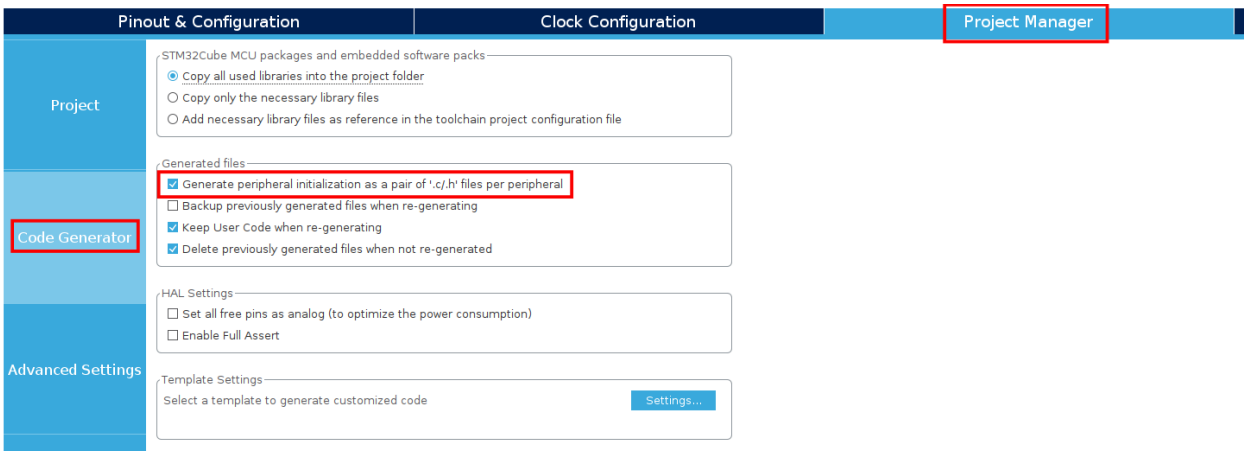

Figure 5: Code generator settings in STM32CubeMX.

Once the configuration is finished, click on *Generate Code*.

The source tree of the generated code should look like this:

```
Lab_GPIO/
  Core/
    Inc/
      gpio.h
      main.h
      stm32f4xx_it.h
      stm32f4xx_hal_conf.h
    Src/
      gpio.c
      main.c
      stm32f4xx_it.c
      stm32f4xx_hal_msp.c
      system_stm32f4xx.c
  Drivers/
  startup_stm32f407xx.s
  STM32F407VGTx_FLASH.ld
  Makefile
```

The `stm32f4xx_hal_conf.h` file defines the enabled HAL modules and sets some parameters (e.g. External High Speed oscillator frequency) to predefined default values or according to user configuration.

The `stm32f4xx_it.h` file contains the prototypes of the interrupt handlers, while the handlers are defined in the `stm32f4xx_it.c` file.

File `system_stm32f4xx.c` defines the default system settings and adequate startup routines.

All initialization functions used to configure the peripherals according to user configuration are defined in `stm32f4xx_hal_msp.c` file.

The generated `main.c` file contains the function for system clock configuration - `SystemClock_Config()`. This function will be explained in subsection 4.2. Let's write a simple `main()` function. In `main.c` add the following lines:

```
int main(void)
{
   while(1);
}
```

Each microcontroller program must not exit the `main()` function because there is no other program environment (e.g. OS) where this function should "return" to.

At this point, we have a functional project source code. The process of compiling the code shall be described in the following text.

3.2 Using Makefile for building the project

The Makefile generated by STM32CubeMX is located in the project source tree. The following text gives a brief description of the Makefile generated by STM32CubeMX and it assumes that the content of Appendix A is read and understood. For additional information, please refer to GNU Make documentation.

The target application is defined by the `TARGET` variable.

```
TARGET = Lab_GPIO
```

It is often useful to keep non-source files separate from the source files and therefore we define a build directory, where the compiled files shall be placed:

```
BUILD_DIR = build
```

The C and assembler source files are defined by `C_SOURCES` and `ASM_SOURCES` variables:

```
C_SOURCES =  \
Core/Src/main.c \
Core/Src/gpio.c \
Core/Src/stm32f4xx_it.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_ll_gpio.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_ll_rcc.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_ll_utils.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_ll_exti.c \
Core/Src/system_stm32f4xx.c

ASM_SOURCES =  \
startup_stm32f407xx.s
```

Cross-compiler used for compiling C and asm source files and linking is set in the following lines. The compiler is also used for generating the binary file which will eventually be flashed to STM32F4 board.

```
PREFIX = arm-none-eabi-
...
CC = $(PREFIX)gcc
AS = $(PREFIX)gcc -x assembler-with-cpp
CP = $(PREFIX)objcopy
SZ = $(PREFIX)size
...
HEX = $(CP) -O ihex
BIN = $(CP) -O binary -S
```

The following snippet defines compiler flags which include project headers, platform-specific flags and macros for gcc:

```
# cpu
CPU = -mcpu=cortex-m4

# fpu
FPU = -mfpu=fpv4-sp-d16

# float-abi
FLOAT-ABI = -mfloat-abi=hard

# mcu
MCU = $(CPU) -mthumb $(FPU) $(FLOAT-ABI)

# macros for gcc
# AS defines
AS_DEFS =

# C defines
C_DEFS =  \
-DUSE_FULL_LL_DRIVER \
-DSTM32F407xx

# AS includes
AS_INCLUDES =

# C includes
C_INCLUDES =  \
-ICore/Inc \
-IDrivers/STM32F4xx_HAL_Driver/Inc \
```

```
-IDrivers/STM32F4xx_HAL_Driver/Inc/Legacy \
-IDrivers/CMSIS/Device/ST/STM32F4xx/Include \
-IDrivers/CMSIS/Include

# compile gcc flags
ASFLAGS = $(MCU) $(AS_DEFS) $(AS_INCLUDES) $(OPT) -Wall -fdata-sections \
-ffunction-sections

CFLAGS = $(MCU) $(C_DEFS) $(C_INCLUDES) $(OPT) -Wall -fdata-sections \
-ffunction-sections
```

Linker script that shall be used is defined as follows, along with the corresponding libraries and linker flags:

```
LINKER_SCRIPT = STM32F407VGTx_FLASH.ld
# libraries
LIBS = -lc -lm -lnosys
LIBDIR =
LDFLAGS = $(MCU) -specs=nano.specs -T$(LDSCRIPT) $(LIBDIR) $(LIBS) -Wl, \
-Map=$(BUILD_DIR)/$(TARGET).map,--cref -Wl,--gc-sections
```

The default target is denoted by `all` and it builds `.elf`, `.hex` and `.bin` files.

```
all: $(BUILD_DIR)/$(TARGET).elf $(BUILD_DIR)/$(TARGET).hex \
$(BUILD_DIR)/$(TARGET).bin
```

List of required objects is generated by wildcard expressions:

```
# list of objects
OBJECTS = $(addprefix $(BUILD_DIR)/,$(notdir $(C_SOURCES:.c=.o)))
vpath %.c $(sort $(dir $(C_SOURCES)))
# list of ASM program objects
OBJECTS += $(addprefix $(BUILD_DIR)/,$(notdir $(ASM_SOURCES:.s=.o)))
vpath %.s $(sort $(dir $(ASM_SOURCES)))
```

The following snippet contains the targets and corresponding rules to generate object files and executables:

```
$(BUILD_DIR)/%.o: %.c Makefile | $(BUILD_DIR)
        $(CC) -c $(CFLAGS) -Wa,-a,-ad, \
        -alms=$(BUILD_DIR)/$(notdir $(<:.c=.lst)) $< -o $@

$(BUILD_DIR)/%.o: %.s Makefile | $(BUILD_DIR)
        $(AS) -c $(CFLAGS) $< -o $@

$(BUILD_DIR)/$(TARGET).elf: $(OBJECTS) Makefile
        $(CC) $(OBJECTS) $(LDFLAGS) -o $@
        $(SZ) $@

$(BUILD_DIR)/%.hex: $(BUILD_DIR)/%.elf | $(BUILD_DIR)
        $(HEX) $< $@

$(BUILD_DIR)/%.bin: $(BUILD_DIR)/%.elf | $(BUILD_DIR)
        $(BIN) $< $@
```

The `clean` target is used to delete all non-source files:

```
clean:
        -rm -fR $(BUILD_DIR)
```

Our project is now ready to be compiled. Run `make` in the command line. If you followed all the steps correctly, you should get the following output:

```
...
arm-none-eabi-size build/Lab_GPIO.elf
   text    data     bss     dec     hex filename
   4396      20    1572    5988    1764 build/Lab_GPIO.elf
arm-none-eabi-objcopy -O ihex build/Lab_GPIO.elf build/Lab_GPIO.hex
arm-none-eabi-objcopy -O binary -S build/Lab_GPIO.elf build/Lab_GPIO.bin
```

Now we have our binary file ready to be flashed. If you are using ST-Link for flashing executables, you can add an adjustment to the Makefile to simplify the flashing process. We will add an additional target to the Makefile which invokes ST-Link application, defines the binary to be flashed and the starting address. Add the following lines to the Makefile:

Linux OS

```
flash: $(BUILD_DIR)/$(TARGET).bin
    st-flash write $^ 0x8000000
```

Windows OS

```
flash: $(BUILD_DIR)/$(TARGET).bin
    st-link_cli -c SWD UR -P "$^" 0x8000000 -Rst -Run
```

This will create `flash` target, so you can flash your binary to the microcontroller by simply running `make flash`. Check out st-flash man pages for more details.

At this point, we have a functional project template and we are ready to implement *Hello World* program and run it on our STM32F4 microcontroller.

4 General-purpose input/output (GPIO) ports

The introductory *Hello World* programming example counterpart in a microcontroller world is usually called "Blinky". Considering that there is no readily available console output for your `printf` statements to check the program behaviour (until you make youself one), it is usually the first step to light some LED(s) on and off, by changing the logic levels at GPIO pins. In this part of exercise we shall learn how to control LED states (GPIO levels) by using STM32F4 HAL library.

Assignment:
Make Blinky program that will use HAL library to blink on-board LEDs.

4.1 Determine the GPIO pins connected to on-board LEDs

In order to control the on-board LEDs, it is first necessary to determine to which GPIO pins these LEDs are connected. STM32F4DISCOVERY board user manual [8] defines LEDs positions on page 16. We are concerned with four user-controlled LEDs:

[8] UM1472 - Discovery kit for STM32F407.pdf

User LED ID	Color	Port pin
LD3	Orange	PD13
LD4	Green	PD12
LD5	Red	PD14
LD6	Blue	PD15

Table 1: Port pins corresponding to user LEDs.

4.2 Initialization of GPIO interface

In STM32CubeMX, open the project created in section 3. Select the Pinout & Configuration tab. The pins corresponding to user LEDs (described in 1) are configured in the following way:

- click on the pin (e.g. PD13) to expand the drop-down menu,
- set the pin mode to *GPIO_Output*.

After all the pins are configured, the pinout display should look like this:

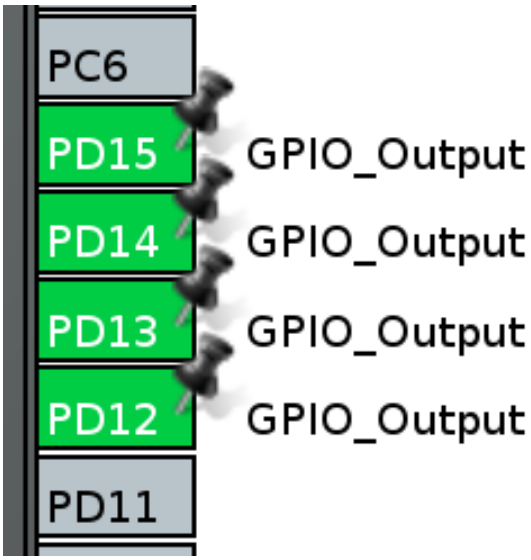

Figure 6: Configured GPIO pins.

To configure additional GPIO settings, open GPIO Configuration window. From the menu on the left side, select *System Core* and click on *GPIO* (see Fig. 7). For each modified pin, select *Very High* under *Maximum output speed* column.

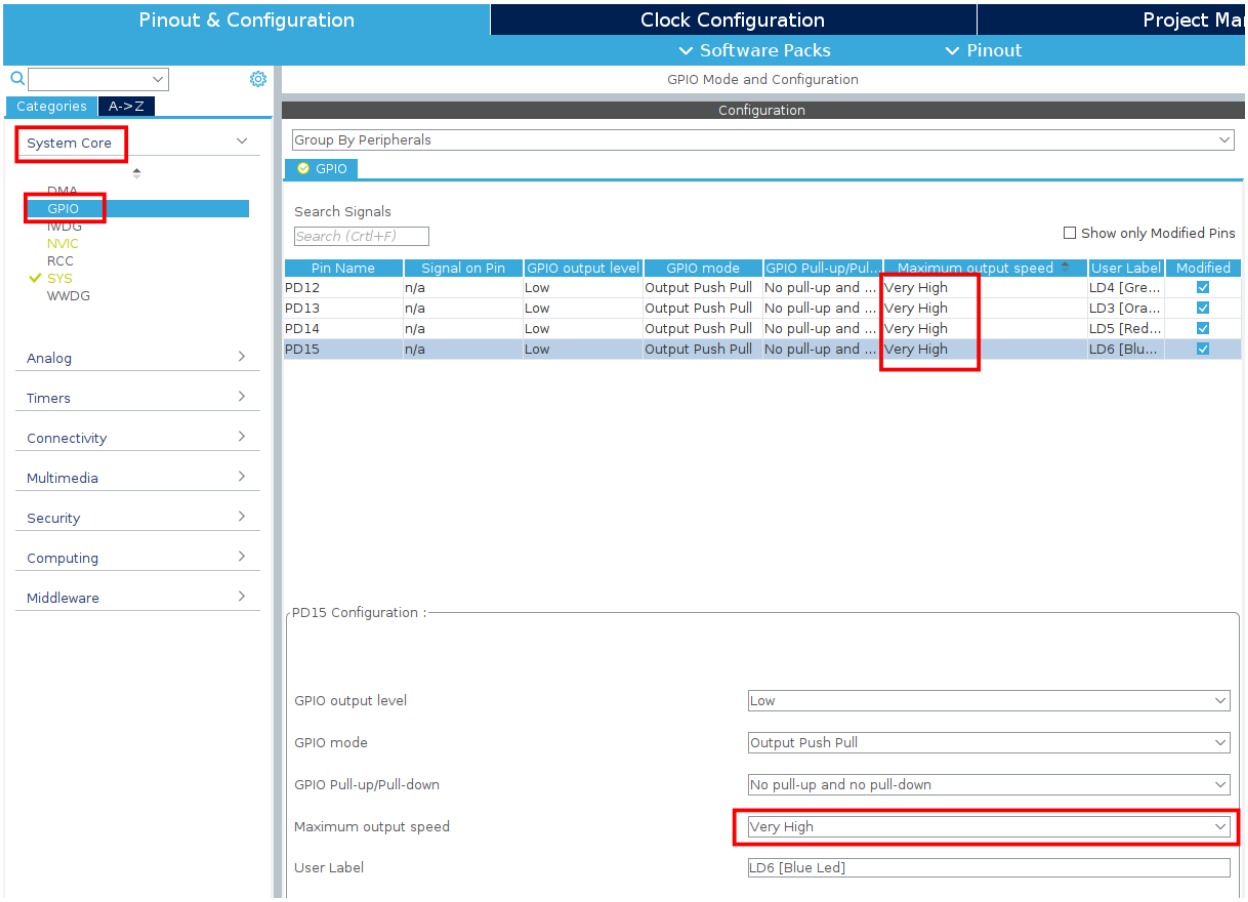

Figure 7: Gpio configuration window.

It is also necessary to initialize Reset and Clock Control (RCC) peripheral, since each GPIO port (more generally, every peripheral) has to be explicitly assigned a clock to be functional. To initialize RCC, select *RCC* under *System Core* menu. Set High Speed Clock (HSE) to *Crystal/Ceramic Resonator* (see Fig. 8).

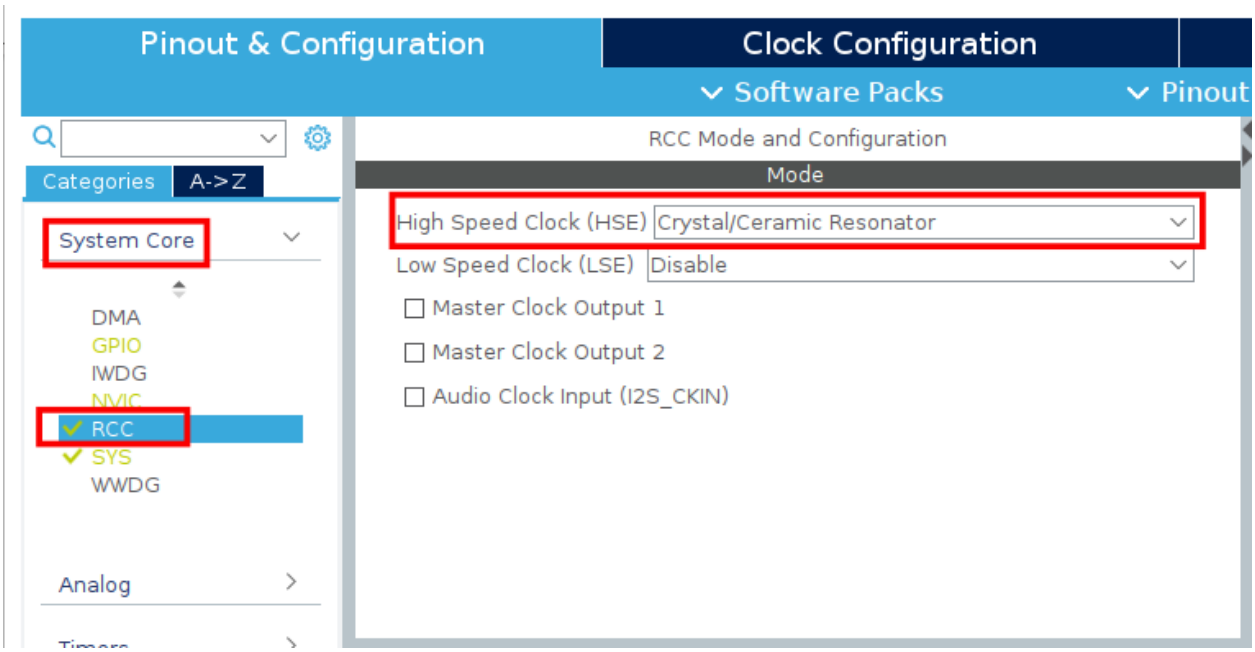

Figure 8: RCC configuration.

Then go to *Clock Configuration* window to configure clock source on each peripheral. Type "8" to input frequency. On PLL source Mux choose HSE. On *System Clock Mux* choose PLLCLK. On the right side we can choose clock for each peripheral. Set APB1 peripheral clock to "36" and APB2 peripheral clock to "72". The required configuration in shown in Fig. 9. Click *Resolve Clock Issues* and CubeMX will automatically resolve any issues with chosen clock values. If it cannot be resolved automatically, then you must choose other values for clock.

Click on *Generate Code*. After the code is generated, examine the changes made in the existing files.

Each module should have well defined functionality and consistent interface for easy interfacing with other modules. The generated code for GPIO initialization is a good example of decoupling functional module from the main program.

The code for initialization of the GPIO module is generated in separate files: `gpio.c` and `gpio.h`.

In `gpio.c`, there is a function `MX_GPIO_Init()` for initializing LED GPIO pins. After the reset, all GPIO pins are in inactive state (open drain) and GPIO ports are not clocked. We have to:

- provide clock for the chosen GPIO port,
- configure the chosen pin settings.

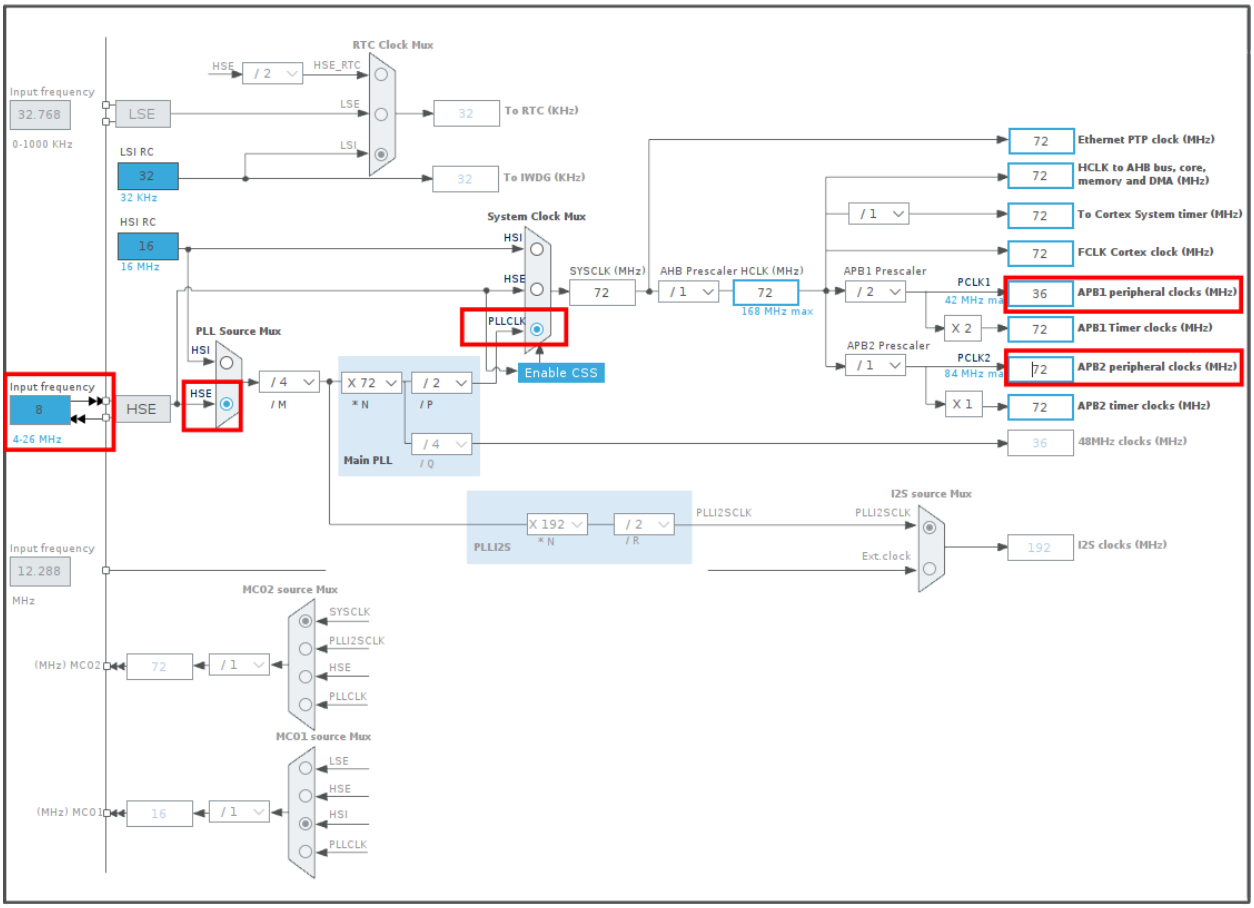

Figure 9: Clock parameters configuration.

This is achieved by the following code:

```
/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOD_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15, GPIO_PIN_RESET);

  /*Configure GPIO pins : PD12 PD13 PD14 PD15 */
  GPIO_InitStruct.Pin = GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
}
```

Let us see what each part of the code above means. As one can notice, we do not access MCU registers directly, neither we write any "magic" hex number constants to these registers, and we use the high level HAL approach instead. To ensure that the code is understandable without reading the corresponding datasheet that contains specifications of each peripheral register, we use a human readable HAL functions and structures that are provided by MCU manfacturer, in this particular case, ST HAL library for STM32 MCUs.

Firstly, let us see the definition of `GPIO_InitTypeDef` structure (in header file `stm32f4xx_ll_gpio.h`), which is rather self-explanatory:

```
typedef struct
{
  uint32_t Pin;       /*!< Specifies the GPIO pins to be configured.
                           This parameter can be any value of @ref GPIO_pins_define */
  uint32_t Mode;      /*!< Specifies the operating mode for the selected pins.
                           This parameter can be a value of @ref GPIO_mode_define */
  uint32_t Pull;      /*!< Specifies the Pull-up or Pull-Down activation for the selected
                           pins.
                           This parameter can be a value of @ref GPIO_pull_define */
  uint32_t Speed;     /*!< Specifies the speed for the selected pins.
                           This parameter can be a value of @ref GPIO_speed_define */
  uint32_t Alternate; /*!< Peripheral to be connected to the selected pins.
                           This parameter can be a value of
                           @ref GPIO_Alternate_function_selection */
} GPIO_InitTypeDef;
```

Before considering how to initialize `GPIO_InitStruct`, let us see the macro function `__HAL_RCC_GPIOD_CLK_ENABLE()`.

```
#define __HAL_RCC_GPIOD_CLK_ENABLE() do { \
                                          __IO uint32_t tmpreg = 0x00U; \
                                          SET_BIT(RCC->AHB1ENR, RCC_AHB1ENR_GPIODEN);\
                                          /* Delay after an RCC peripheral clock enabling */ \
                                          tmpreg = READ_BIT(RCC->AHB1ENR, RCC_AHB1ENR_GPIODEN);\
                                          UNUSED(tmpreg); \
                                          } while(0U)
```

The function is defined in `stm32f4xx_hal_rcc_ex.h` file and it serves to enable clock for some peripheral connected to Port D. Port D peripherals are connected to AHB1 bus interface, so the `RCC_AHB1ENR_GPIODEN` bit of the RCC AHB1 peripheral clock register must be set (page 180 of STM32F4 Reference Manual).
Bonus question: Why is `do {...} while(0)` syntax used in this macro function?

The `GPIO_InitTypeDef` structure references defined values for pins, operating mode, speed and output type. The pin values used are defined via pin masks. The pin masks are defined in `stm32f4xx_ll_gpio.h` header:

```
/** @defgroup GPIO_pins_define GPIO pins define
  * @{
  */
#define GPIO_PIN_0                 ((uint16_t)0x0001)  /* Pin 0 selected    */
#define GPIO_PIN_1                 ((uint16_t)0x0002)  /* Pin 1 selected    */
#define GPIO_PIN_2                 ((uint16_t)0x0004)  /* Pin 2 selected    */
#define GPIO_PIN_3                 ((uint16_t)0x0008)  /* Pin 3 selected    */
#define GPIO_PIN_4                 ((uint16_t)0x0010)  /* Pin 4 selected    */
#define GPIO_PIN_5                 ((uint16_t)0x0020)  /* Pin 5 selected    */
#define GPIO_PIN_6                 ((uint16_t)0x0040)  /* Pin 6 selected    */
#define GPIO_PIN_7                 ((uint16_t)0x0080)  /* Pin 7 selected    */
#define GPIO_PIN_8                 ((uint16_t)0x0100)  /* Pin 8 selected    */
#define GPIO_PIN_9                 ((uint16_t)0x0200)  /* Pin 9 selected    */
#define GPIO_PIN_10                ((uint16_t)0x0400)  /* Pin 10 selected   */
#define GPIO_PIN_11                ((uint16_t)0x0800)  /* Pin 11 selected   */
#define GPIO_PIN_12                ((uint16_t)0x1000)  /* Pin 12 selected   */
#define GPIO_PIN_13                ((uint16_t)0x2000)  /* Pin 13 selected   */
```

```
#define GPIO_PIN_14                 ((uint16_t)0x4000)  /* Pin 14 selected   */
#define GPIO_PIN_15                 ((uint16_t)0x8000)  /* Pin 15 selected   */
#define GPIO_PIN_All                ((uint16_t)0xFFFF)  /* All pins selected */
```

The rest of the pin mask macros can be found in `stm32f4xx_hal_gpio.h` file.

```
#define GPIO_MODE_OUTPUT_PP        0x00000001U   /*!< Output Push Pull Mode */
#define GPIO_MODE_OUTPUT_OD        0x00000011U   /*!< Output Open Drain Mode */
...
#define GPIO_SPEED_FREQ_LOW        0x00000000U  /*!< IO works at 2 MHz, please refer
                                                     to the product datasheet */
#define GPIO_SPEED_FREQ_MEDIUM     0x00000001U  /*!< range 12,5 MHz to 50 MHz,
                                                     please refer to the product datasheet */
#define GPIO_SPEED_FREQ_HIGH       0x00000002U  /*!< range 25 MHz to 100 MHz,
                                                     please refer to the product datasheet */
#define GPIO_SPEED_FREQ_VERY_HIGH 0x00000003U  /*!< range 50 MHz to 200 MHz,
                                                     please refer to the product datasheet */
...
#define GPIO_NOPULL                0x00000000U  /*!< No Pull-up or Pull-down activation  */
#define GPIO_PULLUP                0x00000001U  /*!< Pull-up activation                  */
#define GPIO_PULLDOWN              0x00000002U  /*!< Pull-down activation                */
```

In this case we want to initialize pins as output, push-pull type (instead of open-drain), with no internal pull-up or pull-down resistor. We set fast I/O speed (50 MHz setting). Slower speeds are desirable in cases when speed is not needed to mitigate electromagnetic intereference (EMI) issues due to fast rise time of output signal.

After the settings in GPIO descriptor structure are configured, we apply them to the chosen GPIO port (Port D in this case) using the call to `HAL_GPIO_Init` API function:

```
HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
```

The prototype definition of functio `HAL_GPIO_Init` can be found in `stm32f4xx_hal_gpio.h`:

```
void HAL_GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_Init);
```

The first argument is of type `GPIO_TypeDef`. Data type `GPIO_TypeDef` can be found in header `stm32f412cx.h`:

```
typedef struct
{
  __IO uint32_t MODER;   /*!< GPIO port mode register,
                              Address offset: 0x00      */
  __IO uint32_t OTYPER;  /*!< GPIO port output type register,
                              Address offset: 0x04      */
  __IO uint32_t OSPEEDR; /*!< GPIO port output speed register,
                              Address offset: 0x08      */
  __IO uint32_t PUPDR;   /*!< GPIO port pull-up/pull-down register,
                              Address offset: 0x0C      */
  __IO uint32_t IDR;     /*!< GPIO port input data register,
                              Address offset: 0x10      */
  __IO uint32_t ODR;     /*!< GPIO port output data register,
                              Address offset: 0x14      */
  __IO uint16_t BSRRL;   /*!< GPIO port bit set/reset low register,
                              Address offset: 0x18      */
  __IO uint16_t BSRRH;   /*!< GPIO port bit set/reset high register,
                              Address offset: 0x1A      */
  __IO uint32_t LCKR;    /*!< GPIO port configuration lock register,
                              Address offset: 0x1C      */
  __IO uint32_t AFR[2];  /*!< GPIO alternate function registers,
                              Address offset: 0x20-0x24 */
} GPIO_TypeDef;
```

The C structure `GPIO_InitTypeDef` contains logical fields which are other typedefs, enums or defines, and has not direct connection with microcontroller hardware registers. The meaning of `GPIO_TypeDef` structure is slightly different as it maps directly to hardware addresses of registers controlling GPIO port. The `GPIO_TypeDef` structure contains an "addressing template" with appropriate offsets and enables access to hardware registers conveniently through the C struct fields. For example, since all port control register banks have the same structure in memory address space, it is sufficient to provide base address of each port control register bank to map appropriate port with symbolic identifier. For example, the port D is accessed through the `GPIO_TypeDef`-typed pointer on predefined address:

```
#define GPIOD                ((GPIO_TypeDef *) GPIOD_BASE)
```

In code, the symbol `GPIOD` refers to the typed pointer pointing to the address `GPIOD_BASE`, which is defined as:

```
/*!< Peripheral base address in the alias region */
#define PERIPH_BASE           ((uint32_t)0x40000000UL)
#define AHB1PERIPH_BASE       (PERIPH_BASE + 0x00020000UL)
#define GPIOD_BASE            (AHB1PERIPH_BASE + 0x0C00UL)
```

4.3 Avoiding hard-coded constants via `#define` statements

If we take a look on the code

```
void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOD_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15, GPIO_PIN_RESET);

  /*Configure GPIO pins : PD12 PD13 PD14 PD15 */
  GPIO_InitStruct.Pin = GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
}
```

we can see that it would be rather tedious and error-prone to change e.g. LED location from PD13 to PA10. The problem is even more emphasized when the number of GPIO pins rises. Therefore, it is a good programming practice to avoid any hard-coded constants in your application code and move all hardware-specific references to `#define` statements. Let us consider the following code:

```
...
#define LED_GPIOx                    GPIOD
#define LED4_GREEN_PinNumber         GPIO_PIN_12
#define LED3_ORANGE_PinNumber        GPIO_PIN_13
#define LED5_RED_PinNumber           GPIO_PIN_14
#define LED6_BLUE_PinNumber          GPIO_PIN_15

void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOD_CLK_ENABLE();
```

```
14
15    /*Configure GPIO pin Output Level */
16    HAL_GPIO_WritePin(LED_GPIOx, LED4_GREEN_PinNumber|LED3_ORANGE_PinNumber|LED5_RED_PinNumber
          |LED6_BLUE_PinNumber, GPIO_PIN_RESET);
17
18    /*Configure GPIO pins : PD12 PD13 PD14 PD15 */
19    GPIO_InitStruct.Pin = LED4_GREEN_PinNumber|LED3_ORANGE_PinNumber|LED5_RED_PinNumber|
          LED6_BLUE_PinNumber;
20    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
21    GPIO_InitStruct.Pull = GPIO_NOPULL;
22    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
23    HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
24  }
```

The code is almost the same like in the previous version but nothing in application code is now hard-coded. There is a single place where you change your port pin definitions. For example, if you want to move orange LED to some external LED connected on PA10, this can be achieved without affecting the application code at all, only `#define` statements:

```
1  #define LED_GPIOx                    GPIOD
2  #define LED4_GREEN_PinNumber         GPIO_PIN_12
3  #define LED3_ORANGE_PinNumber        GPIO_PIN_13
4  #define LED5_RED_PinNumber           GPIO_PIN_14
5  #define LED6_BLUE_PinNumber          GPIO_PIN_15
```

The application code is unaffected by this change and program is now easy to modify and maintain.

Code in header files and C modules is given below (comments generated by CubeMX are left out):

```
1  /* gpio.h */
2  #ifndef __gpio_H
3  #define __gpio_H
4  #ifdef __cplusplus
5   extern "C" {
6  #endif
7
8  #include "main.h"
9
10 #define LED_GPIOx                    GPIOD
11 #define LED4_GREEN_PinNumber         GPIO_PIN_12
12 #define LED3_ORANGE_PinNumber        GPIO_PIN_13
13 #define LED5_RED_PinNumber           GPIO_PIN_14
14 #define LED6_BLUE_PinNumber          GPIO_PIN_15
15
16 void MX_GPIO_Init(void);
17
18 #ifdef __cplusplus
19 }
20 #endif
21 #endif
```

```
1  /* main.h */
2  #ifndef __MAIN_H
3  #define __MAIN_H
4
5  #ifdef __cplusplus
6  extern "C" {
7  #endif
8
9  #include "stm32f4xx_hal.h"
10
11 void Error_Handler(void);
12
13 #ifdef __cplusplus
```

```
}
#endif

#endif /* __MAIN_H */
```

```
/* gpio.c */
#include "gpio.h"

void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOD_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LED_GPIOx, LED4_GREEN_PinNumber|LED3_ORANGE_PinNumber|LED5_RED_PinNumber
      |LED6_BLUE_PinNumber, GPIO_PIN_RESET);

  /*Configure GPIO pins : PD12 PD13 PD14 PD15 */
  GPIO_InitStruct.Pin = LED4_GREEN_PinNumber|LED3_ORANGE_PinNumber|LED5_RED_PinNumber|
      LED6_BLUE_PinNumber;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
}
```

```
/* main.c */
#include "main.h"
#include "gpio.h"

void SystemClock_Config(void);

int main(void) {
        HAL_Init();
        MX_GPIO_Init();
        while(1);
}

void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
```

```
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
  {
    Error_Handler();
  }
}
```

The header file `gpio.h` contains all definitions and function prototype declarations that we want to use in external modules. The structure

```
#ifndef __gpio_H
#define __gpio_H
...
#endif __gpio_H
```

is called include guards, which prevents recursive inclusion of the same header file within a single translation unit (C file). By convention, at the beginning of the file it is checked whether the symbol `__<filename>_H` has been already encountered in the current translation unit, thus preventing the collision with already parsed identifiers. Lack of include guards would require very careful use of include files, and make it even impossible to resolve complex include schemes in larger libraries and programs in many cases. Next, we include header file `main.h`.

The block of `#define` statements:

```
#define LED_GPIOx                   GPIOD
#define LED4_GREEN_PinNumber        GPIO_PIN_12
#define LED3_ORANGE_PinNumber       GPIO_PIN_13
#define LED5_RED_PinNumber          GPIO_PIN_14
#define LED6_BLUE_PinNumber         GPIO_PIN_15
```

extends the definition of the on-board LEDs, to include all four LEDs present on STM32F4DISCOVERY development board. Finally, the line

```
void MX_GPIO_Init(void);
```

is not only the forward declaration of function `MX_GPIO_Init()` in `gpio.c`, it also serves to any module that includes `gpio.h` to know declaration of `MX_GPIO_Init()` function. The actual function code is not necessary for compilation (only function declaration from header), and actual machine code for the function will be resolved by linker either from the source code (if C file is present in a project) or precompiled static library (depending on project configuration).

The module file `gpio.c` needs only to include `gpio.h`:

```
#include "gpio.h"
```

to automatically reference all needed STM32F4 HAL functions, peripheral definitions (provided by `#define` statements) and forward function declarations (this is useful on a module-level only if some module functions uses them before function body implementations in C source file).

The include file main.h has include guards and the following include statement:

```
#include "stm32f4xx_hal.h"
```

The header file `stm32f4xx_hal.h` contains all the functions prototypes for the HAL module driver.

The structure of the main.c program is now much more clear and oriented to application logic:

```
int main(void) {
        HAL_Init();
        SystemClock_Config();
        MX_GPIO_Init();
        while(1);
}
```

The `HAL_Init()` function is used to initialize HAL library. Configuration of system clock according to the parameters we set in CubeMX is done in the `SystemClock_Config()` function:

```
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 8;
  RCC_OscInitStruct.PLL.PLLN = 72;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 4;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
  {
    Error_Handler();
  }
}
```

The `RCC_OscInitTypeDef` and `RCC_ClkInitTypeDef` types are defined in `stm32f4xx_hal_rcc.h` header:

```
...
typedef struct
{
  uint32_t OscillatorType;      /*!< The oscillators to be configured.
                                     This parameter can be a value of
                                     @ref RCC_Oscillator_Type                    */
  uint32_t HSEState;            /*!< The new state of the HSE.
                                     This parameter can be a value of
                                     @ref RCC_HSE_Config                         */
  uint32_t LSEState;            /*!< The new state of the LSE.
                                     This parameter can be a value of
                                     @ref RCC_LSE_Config                         */
  uint32_t HSIState;            /*!< The new state of the HSI.
```

```
                                    This parameter can be a value of
                                    @ref RCC_HSI_Config                          */
  uint32_t HSICalibrationValue; /*!< The HSI calibration trimming value
                                    (default is RCC_HSICALIBRATION_DEFAULT).
                                    This parameter must be a number between
                                    Min_Data = 0x00 and Max_Data = 0x1F           */
  uint32_t LSIState;            /*!< The new state of the LSI.
                                    This parameter can be a value of
                                    @ref RCC_LSI_Config                          */
  RCC_PLLInitTypeDef PLL;       /*!< PLL structure parameters                     */
} RCC_OscInitTypeDef;
...
typedef struct
{
  uint32_t ClockType;             /*!< The clock to be configured.
                                       This parameter can be a value of
                                       @ref RCC_System_Clock_Type                */
  uint32_t SYSCLKSource;          /*!< The clock source (SYSCLKS) used as system clock.
                                       This parameter can be a value of
                                       @ref RCC_System_Clock_Source              */
  uint32_t AHBCLKDivider;         /*!< The AHB clock (HCLK) divider.
                                       This clock is derived from the system clock (SYSCLK).
                                       This parameter can be a value of
                                       @ref RCC_AHB_Clock_Source                 */
  uint32_t APB1CLKDivider;        /*!< The APB1 clock (PCLK1) divider.
                                       This clock is derived from the AHB clock (HCLK).
                                       This parameter can be a value of
                                       @ref RCC_APB1_APB2_Clock_Source           */
  uint32_t APB2CLKDivider;        /*!< The APB2 clock (PCLK2) divider.
                                       This clock is derived from the AHB clock (HCLK).
                                       This parameter can be a value of
                                       @ref RCC_APB1_APB2_Clock_Source           */
} RCC_ClkInitTypeDef;
...
```

The values configured in CubeMX are assigned to the structure members. The RCC oscillators are then initialized according to the parameters specified in `RCC_OscInitStruct` by calling the `HAL_RCC_OscConfig()` function (implemented in `stm32f4xx_hal_rcc.c`). The `HAL_RCC_ClockConfig()` function is used to initialize the CPU, AHB and APB buses clocks according to the parameters specified in `RCC_ClkInitStruct`.

It is good practice to decouple peripheral access from program logic and contain all code accessing the peripherals in separate C modules to keep application logic clean of platform-specific code, making it less error-prone and easier to maintain and port. This approach of building applications by decoupling hardware access from program logic will be maintained in the rest of lab exercises.

4.4 Adding the *blinky* functionality:

At this point we have the code for initializing GPIO to control our four on-board LEDs. In the next step we shall make the program to periodically turn on and off all four LED diodes. Before we start modifying the code in any of the project files, it is important to emphasize that you should only add code between `/* USER CODE BEGIN */` and `/* USER CODE END */` comments. The code that you add between those comments will stay there after we regenerate the code in CubeMX, while the rest of the code will be deleted.

In `gpio.h` add the following constants:

```
#define LED3_ORANGE_ID      1
#define LED4_GREEN_ID       2
#define LED5_RED_ID         3
#define LED6_BLUE_ID        4
```

These constants are user-defined identifiers of each on-board LED. Each LED will be turned on and off by referencing the constant to driver function, instead of hard-coding individual pins to what the LEDs are actually wired. Add the following function to `gpio.h`:

```
void gpio_led_state(uint8_t LED_ID, uint8_t state);
```

In module `gpio.c` add the function implementation code:

```
void gpio_led_state(uint8_t LED_ID, uint8_t state) {
GPIO_PinState pinState;

  pinState = (state == 1) ? GPIO_PIN_SET : GPIO_PIN_RESET;
  switch(LED_ID) {
        case LED3_ORANGE_ID:
          HAL_GPIO_WritePin(GPIOD, LED3_ORANGE_PinNumber, pinState);
          break;
        case LED4_GREEN_ID:
          HAL_GPIO_WritePin(GPIOD, LED4_GREEN_PinNumber, pinState);
          break;
        case LED5_RED_ID:
          HAL_GPIO_WritePin(GPIOD, LED5_RED_PinNumber, pinState);
          break;
        case LED6_BLUE_ID:
          HAL_GPIO_WritePin(GPIOD, LED6_BLUE_PinNumber, pinState);
          break;
  }
}
```

This function will turn ON and OFF the chosen LED, referenced by LED ID define. It uses HAL function:

```
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState);
```

where `pinState` specifies the value to be written to the selected bit (`Bit_RESET` clears the port pin, and `Bit_SET` sets the port pin). Note that any changes in pin assigment to LEDs can be changed in defines in `gpio.h` without affecting a single line of code in `gpio.c` implementation file.

Note that for integer values an explicit signed and size type identifier `uint8_t` is used. It is good practice to clearly define for each integer value whether it is signed or unsigned, and what size is it. Definitions of integer types are contained in `stdint.h`, which is already included in `stm32f407xx.h`.

Commonly used integer types defined in `stdint.h` are:

```
/* exact-width signed integer types */
typedef   signed          char int8_t;
typedef   signed short     int int16_t;
typedef   signed           int int32_t;
typedef   signed       __int64 int64_t;

/* exact-width unsigned integer types */
typedef unsigned          char uint8_t;
typedef unsigned short     int uint16_t;
typedef unsigned           int uint32_t;
typedef unsigned       __int64 uint64_t;
```

Main module now looks like this:

```
/* main.c */
#include "main.h"
int main(void) {
  int i;

```

```
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  while(1) {
    for(i=0;i<1000000;i++);

      gpio_led_state(LED3_ORANGE_ID, 1);     // turn on
      gpio_led_state(LED4_GREEN_ID, 1);      // turn on
      gpio_led_state(LED5_RED_ID, 0);        // turn off
      gpio_led_state(LED6_BLUE_ID, 0);       // turn off

    for(i=0;i<1000000;i++);
      gpio_led_state(LED3_ORANGE_ID, 0);     // turn off
      gpio_led_state(LED4_GREEN_ID, 0);      // turn off
      gpio_led_state(LED5_RED_ID, 1);        // turn on
      gpio_led_state(LED6_BLUE_ID, 1);       // turn on
  }
}
```

The code will alternatively turn on and off pairs of on-board LEDs. Application code in `main.c` is completely decoupled from any platform-specific hardware calls. Any changes in hardware wiring to LEDs are configured in `#define` statements in `gpio.h` at a single point, without affecting both function implementation code in `gpio.c` and application logic in `main.c`. Pauses for blinking LEDs are implemented by simple for loops that provide no strict timings but this shortcoming will be corrected soon in the next chapter.

4.5 Lab outcomes

Students must individually accomplish the following outcomes:

- working Blinky project, following previously elaborated guidelines,
- the project must be ready to be transferred to STM32F4 development board connected to a demonstration computer and ready for practical demonstration,
- students must understand all steps in building the solution and must be able to demonstrate individual steps on demand

Copying other students' solutions without understanding how they work will be properly sanctioned!

5 Timers and interrupts

Assignment:
Make on-board LEDs blink in programmable time intervals. Use timer peripheral to measure time periods accurately and interrupts to trigger actions on timer overflow event.

Guidelines:
The previous chapter described in details how to structure a program and use HAL libraries. The following chapter(s) will describe only parts of solutions related to each specific topic, not repeating all the steps for building the solution as explained earlier.

5.1 Create new project from template

Following the description in previous chapter(s), create new HAL-based project. The easiest way to accomplish this is to copy the project from the previous chapter in a separate folder. In STM32CubeMX, load the new project by option *File - Load Project*.
Important: Assignments in each chapter must be implemented in separate projects! This means that "General-purpose input/output (GPIO) ports" part of the lab exercise is one project, while "Timers and interrupts" is another. You may use parts of the previous solutions for assignments in subsequent chapters.

5.2 Configure timer

STM32F4 contains multiple types of timers with various advanced capabilities (see RM0090 Reference manual for STM32F4 microcontroller family for more details). In this exercise we shall use only simple timer feature, which generates an overflow event that can be either polled or trigger an interrupt. It is sufficient to use one of the general-purpose timers (TIM2-TIM5), which are described in chapter 18 of RM0090 Reference manual.

To configure TIM2 timer, select *Timers* under *Categories* menu and click on TIM2. Set the Clock Source option to Internal Clock.

To calculate Prescaler and Period we must know timer clock and desired frequency. The timer clock frequencies are set by hardware. There are two cases:

- If the APB prescaler is 1, the timer clock frequencies are set to the same frequency as that of the APB domain to which the timers are connected.
- Otherwise, they are set to twice (x2) the frequency of the APB domain to which the timers are connected.

As we can see in Clock Configuration window in CubeMX, APB1 prescaler is 2 so timer clock is twice the frequency of APB domain. Now that we know the timer clock frequency, we can calculate period and prescaler based on desired frequency of 1 kHz. Timer will overflow and cause interrupt each 1 ms. Formula for frequency calculation with prescaler set to 1 is:

$$\text{desired frequency} = \frac{\text{clock frequency}}{\text{Period}}.$$

Therefore, period value is calculated as follows:

$$\text{Period} = \frac{\text{clock frequency}}{\text{desired frequency}}$$

$$\text{Period} = \frac{36 \cdot 10^6}{1 \cdot 10^3} = 36000.$$

If the period value is 36000, the auto-load register should be set to $36000 - 1$. The timer will count from 0 to 35999 and trigger an interrupt, then start counting from 0 again.

Enter these values in the *Categories* menu (see Fig. 10).

Under the *NVIC Settings* tab, enable TIM2 global interrupt (Fig. 11).

Click on *Generate Code*.

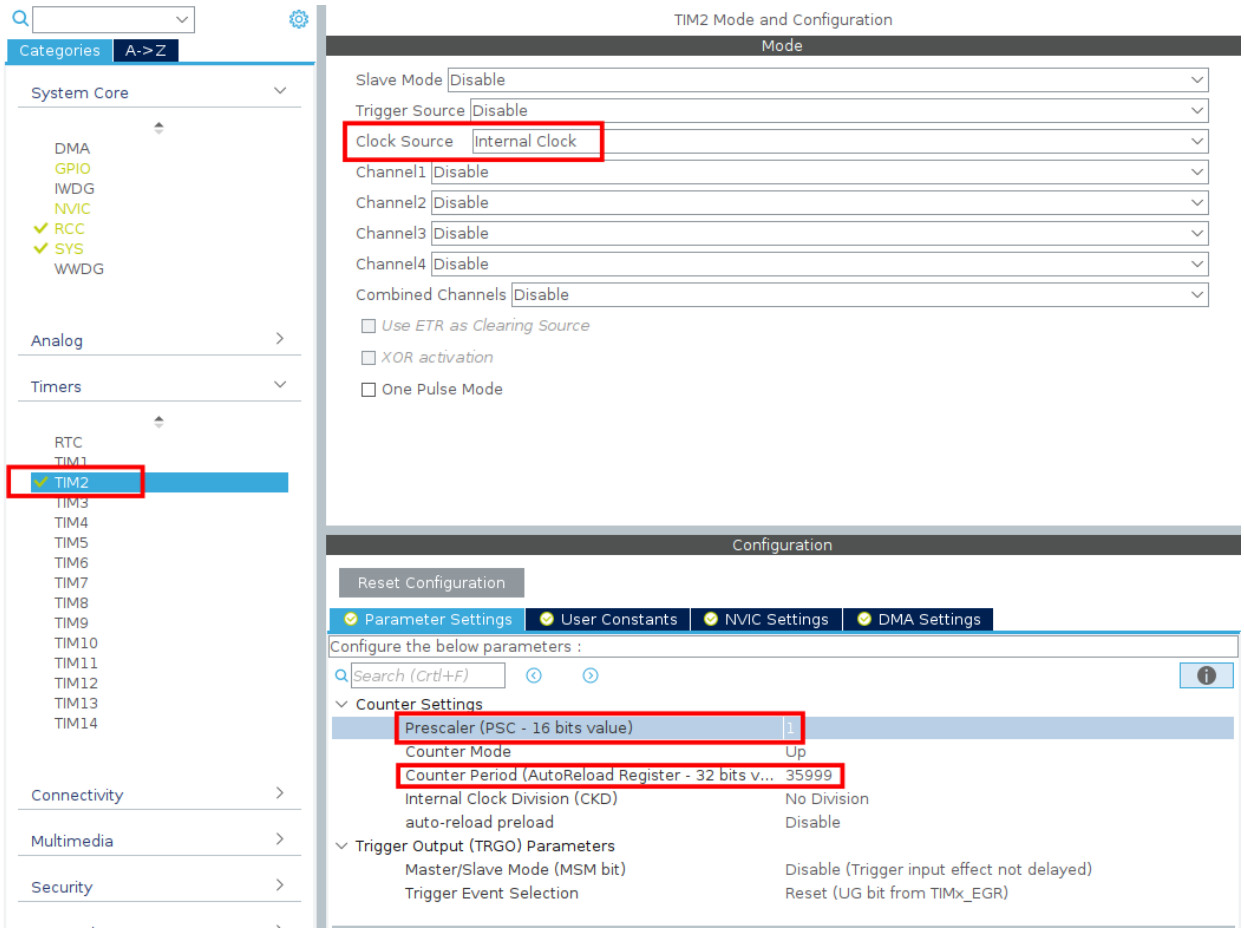

Figure 10: Configuration of timer and prescaler for Timer 2.

5.3 Timer API implementation

Open the generated header file `tim.h`. Under user protoypes, add the following lines:

```
uint32_t timer2_get_millisec(void);
void timer2_wait_millisec(uint32_t ms);
```

This header file will provide an interface to our high-level timer API to be readily used in the main program module, decoupling hardware implementation details from the application logic. Our tiny timer wrapper API will expose the following functions to other modules:

- `uint32_t timer2_get_millisec(void)` - number of milliseconds elapsed since the last call of `MX_TIM2_Init()` function,
- `void timer2_wait_millisec(uint32_t ms)` - blocking call for waiting predefined number of milliseconds.

There are multiple ways to achieve the described functionality of the proposed API. In this example we shall do the following:

- initialize TIM2 timer to overflow every 1 ms (one millisecond software timer resolution) and generate

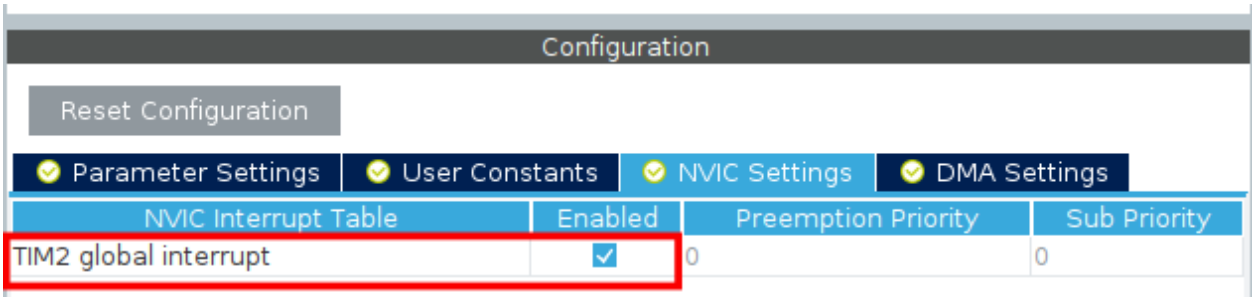

Figure 11: Enabling the TIM2 interrupt.

an interrupt at each overflow,

- use module-level global variable to keep track of number of overflows; the variable will be incremented in TIM2 interrupt service routine (ISR) each millisecond; this will enable to keep track of time and provide an easy implementation of software timer for measuring arbitrary time intervals with millisecond resolution,
- enable the main module to read the elapsed time since the last timer restart in a non-blocking fashion,
- enable the main module to block execution for exact number of milliseconds.

Let us first take a look at the generated function for timer configuration. We need to add the global variable:

```
uint32_t timer2_Ticks_Millisec;
```

This variable is global within the `tim.c` module (i.e. it is visible to all functions contained in a `tim.c` compilation unit) but it will not be available to functions outside of `tim.c` module (because we did not put definition of `timer2_Ticks_Millisec` in `tim.h` header). This is intended behaviour because we do not want external module to directly access the variable that keeps track of time within `tim.c` module.

However, if we define global variable with the same name in some other module (e.g. in `main.c`), the program will not compile and linker will complain about multiple variable definitions with the same name. There are situations when we want to let other modules to access the global variable contained in some compilation unit. If we wanted to export `timer2_Ticks_Millisec` variable and let some other module to use it, in `tim.h` header file we should declar eit like this:

```
/* tim.h */
#ifndef __tim_H
#define __tim_H

...
extern uint32_t timer2_Ticks_Millisec;
...

#endif
```

All modules that use `tim.h` header will now see global variable `timer2_Ticks_Millisec` and will be able to access it from their function code. It is very important to use extern keyword in header file because it will prevent allocating storage for variable - the `extern` keyword serves only to declare the symbol name. Although it would be possible to manually add the line

```
extern uint32_t timer2_Ticks_Millisec;
```

in each module that wants to see `timer2_Ticks_Millisec` variable, this is not a good programming practice because all symbol name exports (whether they are variables or functions) should be kept in the accompanying header file. Another rule is that header files should never allocate any storage, what is achieved by enforcing extern keyword for each variable export declaration.

Next we consider the void MX_TIM2_Init(void) TIM2 initialization routine. Just like in GPIO example, we shall use helper struct of custom type `TIM_HandleTypeDef` to initialize the TIM2 module:

```
...
TIM_HandleTypeDef htim2;
...
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 3;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 35999;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
...
```

`TIM_HandleTypeDef` type is defined as:

```
/* stm32f4xx_hal_tim.h */
...
typedef struct
{
  TIM_TypeDef                 *Instance;     /*!< Register base address             */
  TIM_Base_InitTypeDef        Init;          /*!< TIM Time Base required parameters */
  HAL_TIM_ActiveChannel       Channel;       /*!< Active channel                    */
  DMA_HandleTypeDef           *hdma[7];      /*!< DMA Handlers array
                                                  This array is accessed by a @ref
                                                      DMA_Handle_index */
  HAL_LockTypeDef             Lock;          /*!< Locking object                    */
  __IO HAL_TIM_StateTypeDef   State;         /*!< TIM operation state               */
} TIM_HandleTypeDef;
...
```

The timer instance `TIM2` refers to the typed pointer pointing to the address `TIM2_BASE`, which is defined as:

```
#define AHB1PERIPH_BASE       (PERIPH_BASE + 0x00020000UL)
#define TIM2_BASE             (APB1PERIPH_BASE + 0x0000UL)
#define TIM2                ((TIM_TypeDef *) TIM2_BASE)
```

Counter modes are defined in the `stm32f4xx_hal_tim.h` header, as well as the clock division and auto-reload constants:

```
...
#define TIM_COUNTERMODE_UP             0x00000000U   /*!< Counter used as up-counter   */
#define TIM_COUNTERMODE_DOWN           TIM_CR1_DIR   /*!< Counter used as down-counter */
#define TIM_COUNTERMODE_CENTERALIGNED1 TIM_CR1_CMS_0 /*!< Center-aligned mode 1        */
#define TIM_COUNTERMODE_CENTERALIGNED2 TIM_CR1_CMS_1 /*!< Center-aligned mode 2        */
#define TIM_COUNTERMODE_CENTERALIGNED3 TIM_CR1_CMS   /*!< Center-aligned mode 3        */
...
#define TIM_CLOCKDIVISION_DIV1         0x00000000U   /*!< Clock division: tDTS=tCK_INT   */
#define TIM_CLOCKDIVISION_DIV2         TIM_CR1_CKD_0 /*!< Clock division: tDTS=2*tCK_INT */
#define TIM_CLOCKDIVISION_DIV4         TIM_CR1_CKD_1 /*!< Clock division: tDTS=4*tCK_INT */
...
#define TIM_AUTORELOAD_PRELOAD_DISABLE 0x00000000U /*!< TIMx_ARR register is not buffered */
#define TIM_AUTORELOAD_PRELOAD_ENABLE  TIM_CR1_ARPE  /*!< TIMx_ARR register is buffered */
...
```

The HAL_TIM_Base_Init() then initializes the TIM module according to the specified parameters in the TIM_HandleTypeDef structure.

Clock source is configured by HAL_TIM_ConfigClockSource() function, based on the sClockSourceConfig structure of type TIM_ClockConfigTypeDef:

```
/* stm32f4xx_hal_tim.h */
typedef struct
{
  uint32_t ClockSource;     /*!< TIM clock sources
                                 This parameter can be a value of @ref TIM_Clock_Source */
  uint32_t ClockPolarity;   /*!< TIM clock polarity
                                 This parameter can be a value of @ref TIM_Clock_Polarity */
  uint32_t ClockPrescaler;  /*!< TIM clock prescaler
                                 This parameter can be a value of @ref TIM_Clock_Prescaler
                                     */
  uint32_t ClockFilter;     /*!< TIM clock filter
                                 This parameter can be a number between Min_Data = 0x0 and
                                     Max_Data = 0xF */
} TIM_ClockConfigTypeDef;
```

The TIM_MasterConfigTypeDef structure is used for configuring the timer in slave mode, but we will not use that in this exercise.

The RCC peripheral connected to TIM2 module is initialized in HAL_TIM_Base_MspInit() function. TIM2 interrupt is configured by calling the HAL_NVIC_SetPriority() and HAL_NVIC_EnableIRQ() functions.

```
HAL_NVIC_SetPriority(TIM2_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(TIM2_IRQn);
```

Priority level is set to maximum by setting the preemption priority and subpriority to zero. Now the TIM2 interrupt request to NVIC will actually generate interrupt service routine (ISR) call upon TIM2 overflow.

Before running the timer we shall reset global variable millisecond counter for later use:

```
timer2_Ticks_Millisec = 0;
```

How to write ISR routine and connect the code with interrupt vector table?
Interrupt vector table is defined in startup file startup_stm32f407xx.s (assembly code). Let us consider important parts of startup file:

```
...
/* External Interrupts */
  .word     WWDG_IRQHandler                   /* Window WatchDog              */
  .word     PVD_IRQHandler                    /* PVD through EXTI Line detection */
  .word     TAMP_STAMP_IRQHandler             /* Tamper and TimeStamps through the EXTI line
      */
  .word     RTC_WKUP_IRQHandler               /* RTC Wakeup through the EXTI line */
  .word     FLASH_IRQHandler                  /* FLASH                        */
  .word     RCC_IRQHandler                    /* RCC                          */
  .word     EXTI0_IRQHandler                  /* EXTI Line0                   */
  .word     EXTI1_IRQHandler                  /* EXTI Line1                   */
  .word     EXTI2_IRQHandler                  /* EXTI Line2                   */
  .word     EXTI3_IRQHandler                  /* EXTI Line3                   */
  .word     EXTI4_IRQHandler                  /* EXTI Line4                   */
  .word     DMA1_Stream0_IRQHandler           /* DMA1 Stream 0                */
  .word     DMA1_Stream1_IRQHandler           /* DMA1 Stream 1                */
  .word     DMA1_Stream2_IRQHandler           /* DMA1 Stream 2                */
  .word     DMA1_Stream3_IRQHandler           /* DMA1 Stream 3                */
  .word     DMA1_Stream4_IRQHandler           /* DMA1 Stream 4                */
  .word     DMA1_Stream5_IRQHandler           /* DMA1 Stream 5                */
```

```
  .word     DMA1_Stream6_IRQHandler           /* DMA1 Stream 6                */
  .word     ADC_IRQHandler                    /* ADC1, ADC2 and ADC3s         */
  .word     CAN1_TX_IRQHandler                /* CAN1 TX                      */
  .word     CAN1_RX0_IRQHandler               /* CAN1 RX0                     */
  .word     CAN1_RX1_IRQHandler               /* CAN1 RX1                     */
  .word     CAN1_SCE_IRQHandler               /* CAN1 SCE                     */
  .word     EXTI9_5_IRQHandler                /* External Line[9:5]s          */
  .word     TIM1_BRK_TIM9_IRQHandler          /* TIM1 Break and TIM9          */
  .word     TIM1_UP_TIM10_IRQHandler          /* TIM1 Update and TIM10        */
  .word     TIM1_TRG_COM_TIM11_IRQHandler     /* TIM1 Trigger and Commutation and TIM11 */
  .word     TIM1_CC_IRQHandler                /* TIM1 Capture Compare         */
  .word     TIM2_IRQHandler                   /* TIM2                         */
  .word     TIM3_IRQHandler                   /* TIM3                         */
  .word     TIM4_IRQHandler                   /* TIM4                         */
  .word     I2C1_EV_IRQHandler                /* I2C1 Event                   */
  .word     I2C1_ER_IRQHandler                /* I2C1 Error                   */
  .word     I2C2_EV_IRQHandler                /* I2C2 Event                   */
  .word     I2C2_ER_IRQHandler                /* I2C2 Error                   */
  ...
  .weak TIM2_IRQHandler
  .thumb_set TIM2_IRQHandler,Default_Handler
  ...
```

The startup file contains some important initialization settings that must be taken into account to properly write the application code. Very important setting is the stack size:

```
Stack_Size      EQU     0x00000400
```

The stack size is set to 1024 bytes by default. This size must provide enough space to hold nested function call stack frames, each containing function parameters, local variables etc. Although the default setting is sufficient for most cases, in some situations the stack size must be manually adjusted to accomodate need for deeply nested function calls, functions with large amount of local variables storage etc.

Next important thing is the interrupt vector table (IVT), that is placed at the memory address `0x00000000` after reset. Each DCD directive allocates (consecutive) 4 bytes of memory, starting with `__initial_sp` on address `0x00000000`, `Reset_Handler` ISR function address at `0x00000004` etc. The DCD directives follow the structure of the interrupt vector table as defined by Cortex-M processor architecture. For our case the interesting entry is

```
                DCD     TIM2_IRQHandler                 ; TIM2
```

This positional IVT entry is given symbolic name `TIM2_IRQHandler`. If we define in any included compilation unit (i.e. C source file) a function with exactly the same name, the linker will insert the address of that function into interrupt vector table entry at this place. We can edit startup file and write another name for TIM2 ISR if we wish, but then we also need to call TIM2 ISR function that name in our code.

The `TIM2_IRQHandler()` function is defined in `stm32f4xx_it.c`:

```
void TIM2_IRQHandler(void)
{
  HAL_TIM_IRQHandler(&htim2);
}
```

The `HAL_TIM_IRQHandler()` function calls the user-implemented function `HAL_TIM_PeriodElapsedCallback()`:

```
__weak void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(htim);
```

```
  /* NOTE : This function should not be modified, when the callback is needed,
            the HAL_TIM_PeriodElapsedCallback could be implemented in the user file
   */
}
```

This function is defined as `__weak` function, which means that it can be implemented in a user file if needed. Let's add an implementation of the `HAL_TIM_PeriodElapsedCallback()` function to `tim.c`:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if(htim->Instance == TIM2) {
        timer2_Ticks_Millisec++;
  }
}
```

This function is called every time the timer overflows and causes an interrupt. If TIM2 caused the interrupt, the global variable `timer2_Ticks_Millisec` is incremented, enabling the application code to keep track of elapsed time since the last TIM2 initialization.

How can the application program read the value of `timer2_Ticks_Millisec`? Although the application code could simply read the content of `timer2_Ticks_Millisec` global variable, it is not desirable because this variable could be changed at any time by interrupt (this is a shared resource). Each access to the shared resource must be done within critical section and protected against the non-atomic behavior. It would be advisable not to read this variable directly but to use helper function which contains critical section. Therefore, we do not export `timer2_Ticks_Millisec` for outside modules and use helper function to read the value:

```
uint32_t timer2_get_millisec()
{
  uint32_t value;
  NVIC_DisableIRQ(TIM2_IRQn);
  value = timer2_Ticks_Millisec;
  NVIC_EnableIRQ(TIM2_IRQn);
  return value;
}
```

Since the variable `timer2_Ticks_Millisec` is a shared resource that could be changed by TIM2 interrupt, it should be protected by granting exclusive access to function `timer2_get_millisec()`. At the beginning of critical section, the TIM2 interrupt must be disabled. Then we read the content of `timer2_Ticks_Millisec` variable into local copy and enable TIM2 interrupt again. Since the variable value is locally visible, it is safe to enable interrupt after copying the global variable into local storage and return local copy to the outside caller function. The application code will use this function to read elapsed number of milliseconds at any moment in atomic way.

Another function that can be called from the user application code is `timer2_wait_millisec()`. This function will produce a blocking delay for supplied number of milliseconds, using the previously implemented function `timer2_get_millisec()`. It will first read the elapsed number of milliseconds and wait in a loop until the predefined delay elapses. The atomicity of access to the global variable is satisfied by calling a `timer2_get_millisec()` function that already provides critical section for reading the shared resource.

The `timer2_wait_millisec()` function is implemented like this:

```
void timer2_wait_millisec(uint32_t ms)
{
  uint32_t t1, t2;
  t1 = timer2_get_millisec();

```

```
  while(1) {
    t2 = timer2_get_millisec();
    if ((t2 - t1) >= ms)
      break;

    if (t2 < t1)
      break; // almost never occurs, once in 49 days
  }
}
```

5.4 Modify the main module to make LEDs blink with strict timing

Now that we have defined `tim.h` and `tim.c` files, we can easily modify the `main.c` module from the previous chapter to enable strictly timed LED blinking.

We define a constant `DELAY_MS` which determines the LED blinking interval. To change this interval it is sufficient only to change the `#define` statement, without affecting the code in `main.c`.

```
#define DELAY_MS      1000
```

The main function is given in the following code listing:

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_TIM2_Init();
  HAL_TIM_Base_Start_IT(&htim2);
  MX_GPIO_Init();
  while(1)
  {
    timer2_wait_millisec(1000);
    gpio_led_state(LED3_ORANGE_ID, 1);     // turn on
    gpio_led_state(LED4_GREEN_ID, 1);      // turn on
    gpio_led_state(LED5_RED_ID, 0);        // turn off
    gpio_led_state(LED6_BLUE_ID, 0);       // turn off

    timer2_wait_millisec(1000);
    gpio_led_state(LED3_ORANGE_ID, 0);     // turn off
    gpio_led_state(LED4_GREEN_ID, 0);      // turn off
    gpio_led_state(LED5_RED_ID, 1);        // turn on
    gpio_led_state(LED6_BLUE_ID, 1);       // turn on
  }
}
```

The code is compact and easy to understand. First we initialize GPIO by calling the function `MX_GPIO_Init()`. Then we initialize TIM2 peripheral and NVIC by calling the function `MX_TIM2_Init()`. The following function must be called in order to start the timer in interrupt mode:

```
HAL_TIM_Base_Start_IT(&htim2);
```

The blocking delay is achieved by calling the function `timer2_wait_millisec(DELAY_MS)`, where delay can be changed in `main.h` header.

5.5 Lab outcomes

Students must individually accomplish the following outcomes:

- working project, following the guidelines for this part of lab exercise, with the same remarks like in the previous chapter regarding the project building and transfering to STM32F4DISCOVERY development board.

A Makefile examples

GNU Make is a tool for building and installing programs from source files. Make gets the instructions on how to build executables from a file called *Makefile* which lists all of the non-source files and how to generate them from other files. The-non source files that need to be generated are called *targets*, while the instructions on how to build the targets are called *rules*. All files which are used as inputs to the commands in the rule are referred to as *dependencies*.

The syntax of a simple rule looks like this:

```
target: dependencies
  commands
```

For example, if we wanted to build the `OPPURS_app` program from `OPPURS_app.c` file, we would run the following command:

```
$ gcc -c OPPURS_app.c OPPURS_app.o
$ gcc OPPURS_app.o -o OPPURS_app
```

This compiles the `OPPURS_app.c` file and names the executable `OPPURS_app`.

Instead, we can compile the `OPPURS_app.c` file by using this simple Makefile:

```
OPPURS_app: OPPURS_app.o
  gcc OPPURS_app.o -o OPPURS_app

OPPURS_app.o: OPPURS_app.c
  gcc -c OPPURS_app.c -o OPPURS_app.o
```

If we have a more complex project, it is efficient to use variables. Let's say we want to add another source file to our project and we want to compile with `-g` compiler flag.

```
CC = gcc
CFLAGS = -g

OPPURS_app: OPPURS_app.o OPPURS_bsp.o
  $(CC) OPPURS_app.o OPPURS_bsp.o -o OPPURS_app

OPPURS_app.o: OPPURS_app.c
  $(CC) $(CFLAGS) -c OPPURS_app.c -o OPPURS_app.o

OPPURS_bsp.o: OPPURS_bsp.c
  $(CC) $(CFLAGS) -c OPPURS_bsp.c -o OPPURS_bsp.o
```

Additional simplifications can be made by using wildcards `%, *, ?` and automatic variables (`$@, $^, $<`). The `%` wildcard captures zero or more of any character and it can be used to match all source (`%.c`) or object files (`%.o`). The automatic variables `$@` and `$^` match the target and the dependencies, respectively. Automatic variable `$<` matches the first dependency. Here is an example of the Makefile with included wildcards and special characters:

```
CC = gcc
CFLAGS = -g

SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)

OPPURS_app: $(OBJ)
  $(CC) $^ -o $@

%.o: %.c
  $(CC) $(CFLAGS) -c $^ -o $@
```