ENSEA
Ecole Nationale Supérieure de l'Electronique et de ses Applications

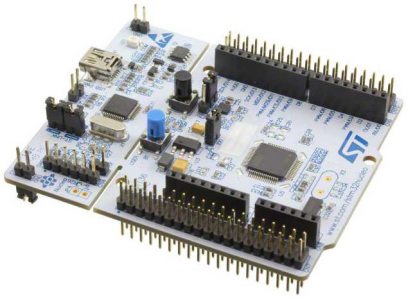
You have questions to answer before coming in lab!

MICROCONTROLLER ARCHITECTURE

BARE METAL

V04

LAB sessions
Lab #2 and #3



The Main Goal:

- *To discover the architecture of a Microcontroller.*

1 Goals and means

1.1 Means

In these lab sessions, you are controlling the green LED and the blue button of your STM32L476-Nucleo Board using very low-level methods also called "Bare Metal Programming". The goal is to understand the architecture of the microcontroller by writing low-level code. Asking an IA to generate this code for you is a very poor idea because you won't understand the architecture.

The assembly code to write must be restricted to the list of instructions given at the end of this document.

A STM32CubeIDE project is required to program the STM32.

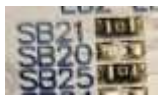
1.2 Documentations

You can find PDF versions of guides on Moodle: "STM32 (ressources)" such as "Microprocessor Basics".

The Input/Output architecture of the STM32L4xxx family is described in the Chapter 8 (GPIO) of the "Reference Manual" (it remains valid for other architectures). More broadly, this document is also valid for other parts of the architecture.

To get the addresses of the Peripherals, have a look at "Memory map" in the datasheet.

For the NUCLEO board itself, have a look at the documents about the board itself: User manual of Nucleo. It is a common document for NUCLEO boards with various STM32 microcontrollers. It is the document to read if you want to know where the LEDs or buttons are connected for any kind of Nucleo 64 boards like yours. To understand this documentation, you must be aware of the use of solder bridges. A solder bridge is connection that can be ON like SB21 and SB25 (0 ohm resitor soldered) or OFF like SB20 (no resistor soldered) .



1.3 Goals

At the end of this first part, you'll obtain an LED animation equivalent to the one you implemented during the first semester with a deep understanding of what is "under".

2 Green LED

In this first part of the lab, you are going to write very low-level code (also called "bare metal" code) to control an LED. Finally, you should better understand the architecture of the microcontroller.

2.1 LED and STM32

The LED is connected to a pin of the STM32. We can distinguish 3 steps to use the LED:

1. First step: **enable** the peripheral on which is connected the LED.
2. Second step: **configure** the pin as an "output Push Pull" (equivalent to Digital Output).
3. Third step: **control** the LED by writing a 1 or a 0 in the corresponding bit at the right place in the memory (peripheral registers).

This is why, you are going to write the following functions: LED_GreenEnable, LED_GreenConfig, LED_GreenDisplay in your LED.s file (do not forget to create a LED.h for the prototypes if you want to call these functions in C, later).

These steps are done by writing code executed by the microprocessor. It will consist in reading and writing in the registers of the STM32: RCC_AHBxENR, GPIOX_MODER or GPIOX_ODR where x and X have been replaced by the right number or letter.

The methods to create code for the peripherals of the STM32 are numerous:

A. Reading and writing, in ASM, directly in the memory (ASM Bare Metal coding).
B. Reading and writing in C, directly in the memory, using pointers (`int *pt;`).
C. Reading and writing in C directly in the memory using already defined data structures in libraries (`RCC->AHPB1ENR=1;`).
D. Reading and writing in the memory by calling HAL (Hardware Abstract Layer) functions: `HAL_GPIO_WritePin(GPIOA,GPIO_PIN_3, GPIO_PIN_RESET);`.
E. Generating code from STM32CubeIDE configurator ("ioc" file).

Q1 : LED (Preparation: so … to be done before coming)

Draw a schematic to light an LED with a "1" from an output pin. In other words: how to connect an LED to a pin of a microcontroller to light it with a 1 on the pin and extinguish it with a 0?

Find in the documentation about the Nucleo board: where is connected the user LED (User Green LED) of your L476RG-Nucleo board: ports and numbers?

Other preparations are to be done next in the text.

2.2 Your STM32CubeIDE project

Today, you can use your previous project or open a new one.

Warning: when using "myasm.s" code initializations are "short-circuited" so the frequency of the CPU clock is not 80MHz but 4 MHz ("reset value" to check in the documentation).

Q2 : Opening previous project (or create a new one)

Launch STM32CubeIDE and use your previous project. If you decide to create a new one, use the BARE_METAL ioc file as previously, to create it and do not forget to insert "myasm.s".

You will add your *.s files (for ASM) and *.c files in "Core/Src". Next, it will be possible to add your own prototypes in H files in "Core/Inc". You can rename a project if you wish: remember, you can't open 2 projects with the same name.

2.3 Pattern to create ASM function in an ASM file (from previous lab)

To create a function in assembly language, it must be in an *.s file starting with:

```
// COMMON DIRECTIVES to Use ASM Language
  .syntax unified
  .cpu cortex-m4
  .fpu softvfp
  .thumb
```

The function must be written according to this pattern:

```
//////////////////
// Here is a pattern to write your own functions
// Do not forget to replace
// all 3 occurences of YOURFUNCT
//////////////////
      .global YOURFUNCT
      .type  YOURFUNCT, %function
      .text
// a label must be followed by :
YOURFUNCT:  PUSH {R4,LR}
// R0 is always the first entering parameter then R1,R2,R3
// R4 is always the first variable then R5,R6..
// insert your code here

// the return value must be copied in R0 before return
                  POP   {R4,PC}
//////////////////////
// To call this code in ASM: BL YOURFUNCT
//////////////////////
// To call this function in C: YOURFUNCT();
// add the right prototype: void YOURFUNCT(void); in header file (*.h)
//////////////////////
```

The starting point of your code (remember previous lab) is given by this label:

```
Reset_Handler:
```

2.4 Structuration of the code for the LEDs (and other peripherals)

The initializations are written in two functions: LED_Enable and LED_Config. The LED_Enable function is here to enable all the ports required for all the LEDs (not only the green one). Then the LED_Config runs the code to configure the LEDs as "General Purpose Output Push-Pull".

The LED must be initialized by **enabling** its GPIO and then **configuring** the corresponding pin as a digital output. This "law" is the same for all peripherals: "Enable" first and then "Configure".

Then it will be possible to light or extinguish the LED with such a "control" function:

```
LED_DriveGreen(int val);
```

Have in mind!

- The functions are written in "*.s" (/Startup) or "*.c" files (/Src).
- The prototypes are written in "*.h" files (/Inc).
- Insert the prototype to use functions with #include …

2.5 New files for LED

You are going to add your own code in ASM files (along with their header files).

Q3 : Creation of files for your green LED and other LEDs

Create LED.c in "Core/Src". To add functions in C.

Create LED.s in "Core/Startup". Yes, in "Startup" (not "Src"), because LED.c and LED.s must not be in the same directory.

Create LED.h in "Core/Inc".

Tip 1: To create the LED.c for your project use the right click on "/Core/Src" (New - Source File). Add the extension: the name must be LED.c.
Tip 2: To add an existing C file to your project use the right click (Add Existing Files).
Tip 3: You will call your own LED functions from other files: do not forget to include the header file when necessary.

2.6 Initialization of the green LED

We are going to use the green LED on the kit. First add a function in ASM to enable the port for this LED: `LED_Enable`. You are going to write this code in ASM in "LED.s".

Enable GPIO

Q4 : Enabling GPIO port (Preparation)

Which register must be modified to enable a GPIO port (GPIOA or GPIOC…)?

We must find the register to enable your GPIO port. Have look at the slides of the course or the Reference Manual (RM). Search for RCC chapter. At the end of the chapter, RCC registers are described. You have all the details to enable all the peripherals. Tip: for GPIOA, the bit is called GPIOAEN, for GPIOB, GPIOBEN...

Next, we will call this port GPIO_GREEN_LED.

In the reference manual, the Memory Organization is described: especially there is a chapter about the memory map and the addresses in the memory of the STM32 where to find the Peripheral associated with peripherals: it is the memory map. What is the address of the register to enable GPIO_GREEN_LED (amongst other GPIOs)? Which bit must be modified for the port of green LED?

Q5 : Enabling GPIO port (in Bare Metal Programming: Assembly language)

Create the LED_Enable function in LED.s.

Add the prototype in LED.h: "void LED_Enable(void);".

You will write the code or call functions to enable and configure all your LEDs in this function. We start with the green LED.

Write the code to enable the port of the green LED in LED_Enable (in "LED.s") using all you know about the ways to read and write in memory in ASM (LDR, STR).

To test it, call `LED_Enable`, from the starting point in "myasm.s" as you did in previous lab, compile your code and enter debug session. Insert breakpoints.

Check with Memory windows (use "unsigned int" as format) that you write the right value at the right place. You can also display RCC peripheral thanks to SFR view in debugger mode. Take snapshots of your screen to show it works.

Configuration of GPIO pin for green LED

Q6 : Configuration for the Green LED (Preparation)

Why is it advised to configure the pin of the LED as GP output push-pull without PU/PD resistors?

Give the value and name of the 7 bits to configure the pin as GP output push-pull in low speed without pull-up or pull-down resistors.

Give the addresses of the following registers: `GPIOX_MODER`, `GPIOX_OTYPER`, `GPIOX_PUPDR`, `GPIOX_OSPEEDR`, `GPIOX_ODR` and `GPIOX_IDR` (where X must be replaced by the right letter).

Where are located the concerned bits in these registers for the green LED?

What are their default values after Reset?

Why is it required to modify MODER? And not the other registers?

Tip: for next question, read how to use logical masks to set or clear bits without modifying others in the course and in this page https://moodle.ensea.fr/mod/resource/view.php?id=4796).

Q7 : Configuration of the Green LED

Add LED_Configure in LED.s (as you did previously).

Write the code in LED_Configure, to configure the pin to use the green LED.

Start with the code to modify the mode.

Is it necessary to modify other settings such as OTYPE, PUPD and OSPEED? Why?

Compile and enter Debug mode. View the register values `RCC_AHBXENR` and then `GPIOX_MODER`… Check the value of the register: compare with expected values. Don't forget to add screenshot of these views for your report.

Test with `GPIOX_ODR`: in debug mode, modify bit n (n is the number of the pin: PXn) of `GPIOX_ODR` after execution of the initialization (stop the execution) and see if green LED is ON or not.

In your report, explain the way GPIOX is enabled and the configuration of GPIOX registers.

Control of the Green LED

We need a "`LED_DriveGreen(int val)`" function to display the green LED ON if the LSB of `val` is 1 else OFF. This function will be written in ASM in LED.s

The algorithm is the following one:

```
val=val&1
if val=1
        write a 1 in GPIOX_ODR at the right place
else
        write a 0 in GPIOX_ODR at the right place
```

Q8 : Display the Green LED

Write `LED_DriveGreen(int val)` in LED.s.

Test this function by calling `LED_DriveGreen(int val)` many times in your code. Do a robust test. Insert at least one breakpoint to test your function, then you can use the functionalities of the debugger session.

Q9 : Green LED switching state

Create "utils.c" in "Core/src".

Add 2 functions in "utils.c" : "void setup(void)" and "void loop(void)". They will be completed next.

Create "utils.h" in "Core/inc".

Add the prototypes of "utils.c" functions in "utils.h".

Call "setup" once from "myasm.s" (the call must be written at the right location. The "setup" function contains your initializations or the actions you want to be executed once.

Call "loop" in an endless loop from "myasm.s". The "loop" function is here to contain a code that is repeated endlessly by the microprocessor.

Add an "int" global variable in utils.c to contain the state of the LED (0 or 1).

Initialize the value in "setup".

Use this variable and `LED_DriveGreen` to make the green LED blink in "loop" function.

You should obtain something like that in "loop":

```
LED_DriveGreen(GreenLED_state);
GreenLED_state =1- GreenLED_state;
```

Add a breakpoint to check it works. What does it happen if there is no breakpoint?

Where is in memory your global variable? What is the maximum size for an "int gtab[Nmax]" array to be declared as a global variable?

Give the ASM codes for the lines in "loop", explain them and show how they follow the rules of the compiler.

We are going to add a function to wait for N times 10 ms. This function can be inserted in “utils.c”.

```
void UTILS_WaitN10ms(int N){
      int n,i,s=0;
      for(n=1;n<=N;n++){
            for(i=0;i<2500;i++){
                  s=s+i;
            }
      }
}
```

Q10 : Green LED blinking

Insert the call of the waiting function in “loop” and make the LED blink thanks to it.

Add a global variable containing the time of the blinking LED in ms.

Where are declared in memory the global variables you use (give the addresses). Use “Build Analyzer- Memory Details” (after Refresh).

In our configuration, with “myasm.s” the CPU works at 2MHz.

Try with a blinking LED at 1000 ms: that is to say 1000ms/2 =500 ms to wait between a 0 to 1 (or 1 to 0).

Is the waiting function giving a satisfying result (at more or less 20%)? How does it work (roughly)?

2.7 Initializations from ioc file

In this first part, the clock of the CPU is not initialized. Its default value is 2Mhz.

We need to use the initializations from ioc file next. That’s why we are going to modify our project and modify the clock of the CPU.

Q11 : modification of the startup file

Create an “OutOfCompilaton” folder in your project, at the same level as “Core”, Drivers” or “Debug”.

Drag and drop “myasm.s” file in “OutOfCompilation”.

Now, the startup_stm32l476xx.s file is used for initializations. After the initializations of this file, the main function is called. That’s why, you need to add the calls to your functions in main.c. It means that you can add initialization code with ioc view.

The CPU clock is now set to 80MHz. Do not forget to launch the setup and loop functions from main.c. Launch your code and explain what you see.

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
setup();
while (1)
{
 loop();
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

Q12 : modification of the waitN10ms

How to modify (in a simple way) the waitN10ms function to obtain the same behavior as before for the blinking of the LED. Explain this modification.

3 Beautiful Blue Button

Create and insert in the correct folders of your project: BUTTON.c and BUTTON.h. These two files will be used for the blue LED of the Nucleo board and the buttons of the lab card.

Q13 : Creation of files for the buttons

Create button.c and button.h files at the right places (you should know where now) (if you don't know, read previous questions).

Q14: Initializations and use of the Blue Button (preparation)

Find on which pin is connected the USER push-button on your Nucleo board by reading the right document. According to the guide, how is connected the USER push-button? Is there already a pull-up or a pull-down resistor? The push-button must be seen as a digital signal. What is the configuration for the Button: MODE, PUPD… ? What are the registers involved in these initializations? What are their addresses.

You must deduce from the previous question, the way to configure the pin on which is connected the User-push-button using masks and « stm32l4xx.h » functions. In the next questions, you will have to write `BUTTON_Enable` and `BUTTON_Configure` to initialize blue button. Next you will write the code of `BUTTON_GetBlueLevel` in order to read its level : it returns 1 if the button is pushed else 0 (not pushed).

At the end of this part, you will use `BUTTON_GetLevel` to modify the speed of the blinking LED (and next the speed of the animation).

At the end of this chapter your "loop" function may look like this:

```
LED_DriveGreen(GreenLED_state);
GreenLED_state =1- GreenLED_state;
UTILS_WaitN10ms(nt_ms/10);
PushedButton=BUTTON_GetBlueLevel();
if(PushedButton==1){
      modify nt_ms
      …
}
```

To obtain that, you need to write functions for your button. Let's start with the initializations!

Q15: Write BUTTON_Enable (in C with home made pointers)

Use "home made" pointers (that is to say "`int * pt;`") in C language to achieve your goal to set the mode of the blue button pin. Write the code of "BUTTON_Enable" (so in BUTTON.c). Add the call of your function at the right place (you should know now).

Test your code with it. Show the effect of this code in memory: which register is modified?

Was it necessary to call "BUTTON_Enable"?

Q16: Write BUTTON_Config (data structures from stm32l4xx.h)

Use data structures such as "`GPIOX->MODER`" (and others) defined in libraries (#include "stm32l4xx.h") to write `BUTTON_Config()`.

Write the code of "BUTTON_Config" and test it.

Q17: Write BUTTON_GetBlueLevel (HAL functions) and test!

Use the right HAL function (you can find the HAL functions for GPIOs in the "stm32l4xx_hal_gpio.c") to write `BUTTON_GetBlueLevel()`.

Write the code of "BUTTON_GetBlueLevel" and test it. Write a simple and dedicated test (not in the previous loop function) of your own for this function to prove it works well. Use the debugger. The goal, here, is not only to convince people (the professor) that your function works well: you must show the register and the bit(s) read by the HAL function. Take screenshots for your report.

Q18: Control the speed of the LED blinking

Insert now your "BUTTON_GetBlueLevel" function in loop function as mentioned before. We want to modify the speed of the blinking, each time you push the blue button. Try to obtain a behavior such as you switch the speed at each time you push the button: one fast speed at 5 (more or less) blinkings per second, one regular speed at 1 per second. Maybe it won't work perfectly but the goal is not to make it run perfectly at this point: we want you to analyze what happens.

What are the main drawbacks and limits of such an implementation for detecting a push on the button? How could you improve it? What was the solution used in the first semester to solve it? We will improve it in the next part if we have time.

Q19: Summary of your work

Now it's time to list the different ways (and therefore ways of coding) of implementing STM32 device initializations. Give the advantages and disadvantages.

By analyzing the assembly code corresponding to the call of your functions from setup and loop functions in utils.c, show the compliance of your ASM code with the rules of the ARM compiler.

4 Restricted list of the instructions to use

Mnemonics	Actions	Limits or remarks
ADD Rd,Rn,#imm3		
ADD Rd,#imm8		
ADD Rd,Rn,Rm		
AND Rd,Rm		
B label		Branch
BXX label		Branch with conditions
CMP Rn,#imm8		
SDIV or UDIV		Signed or Unsigned integer division
EOR Rd,Rm		Exclusive OR
LDR Rt,[Rn]		Read memory
LSL Rd,Rm,#imm5		
MOV Rd,#imm8		
MOV Rd,Rm		
MUL		Multiplication
ORR Rd,Rm		
MVN		NOT

POP		To retrieve data from the stack
PUSH		To save data on the stack
STR Rt,[Rn]		Write in memory
SUB Rd,Rn,#imm3		
SUB Rd,#imm8		
SUB Rd,Rn,Rm		

Table of contents