STM32 Installation and Programming Guide

Low Voltage System Subteam

April 1, 2018

1 Introduction

1.0.1 CMSIS is a software framework for embedded applications that run on Cortex-M based microcontrollers. CMSIS enables consistent and simple software interfaces to the processor and the peripherals, simplifying software reuse, reducing the learning curve for microcontroller developers. The CMSIS-Driver interfaces to connect microcontroller peripherals with standardized middleware and generic user applications. It is common interface across different vendors which is good because it is a standard when Cortex-M devices are being considered.

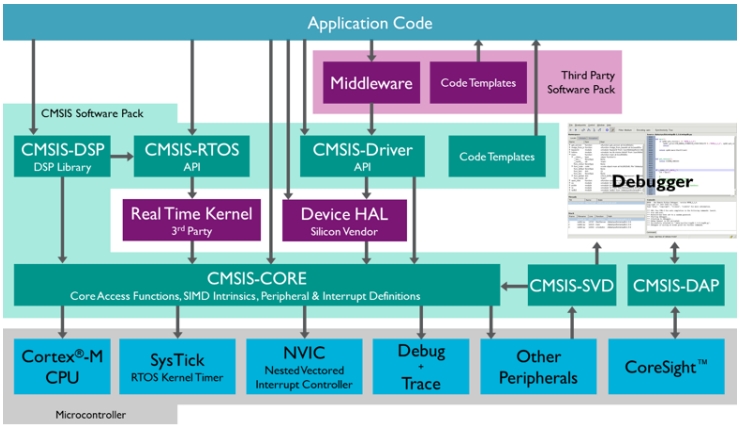


Figure 1. CMSIS Structure

1.0.2 SPL is a set of libraries created by ST company which use the CMSIS framework to program their Cortex-M microcontrollers easier. They use the CMSIS framework to implement the functions of its libraries. These libraries are not so commonly used now because ST dropped support for them in order to implement the new and more powerful HAL Libraries using the STM32CubeMx IDE.

1.0.3 HAL has a lot more features than SPL. The main advantage is that there is a tool from ST that can automatically generate code in HAL and is called STM32CubeMx. One main advantage is that you can reuse your code between different chip families based

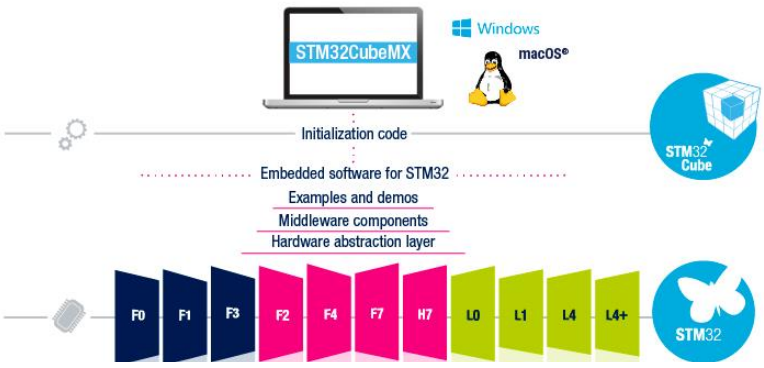


Figure 2. STM32CubeMx from ST

on Cortex-M core from this vendor. HAL has a lot more features such as FreeRTOS, FatFS and a lot more capabilities that weren't before in the SPL.

1.0.4 RTOS Design Involves running several threads with a Real Time Operating System (RTOS). The RTOS provides inter-thread communication and time management functions. A pre-emptive RTOS reduces the complexity of interrupt functions, because high-priority threads can perform time-critical data processing

1.0.5 JTAG, SWD, ST-Link are parts of the process that is used for debugging the uController Unit (MCU), i.e. uploading/flashing new code, and running debugging functions to investigate the flow of the program. Essentially, it's the interface between the microcontroller and the programming computer.

1.1 Quick Overview

1.1 Prerequisites

1.1.1 Keil uVision 5: The **uVision IDE** combines project management, run-time environment, build facilities, source code editing, and program debugging in a single powerful environment. uVision is easy-to-use and accelerates the embedded software development. uVision supports multiple screens and allows us to create individual window layouts anywhere on the visual surface. The **uVision Debugger** provides a single environment in which you may test, verify, and optimize the application code. The debugger includes traditional features like simple and complex breakpoints, watch windows, and execution control and provides full visibility to device peripherals.

How to download it:

- Go to http://www.keil.com/ and click download.
- Then click "Product Downloads".
- After click "MDK-Arm".
- Complete the form and then click submit.

1.1.2 Keil uVision Legacy Support: Because of the fact that MDK Version 5 uses Software Packs to support a microcontroller device and to use middleware, so in order to maintain backward compatibility with MDK Version 4 you may install Legacy Support. This might is necessary in order to:

- Maintain projects created with MDK Version 4 without migrating Software Packs.
- Use older devices that are not supported by a Device Family Pack.

1.1.3 STM32CubeF4 Sample Code

1.1.4 STSW-LINK009 USB Drivers

1.1.5 STSW-LINK007

1.2 uVision Project Manager and Run-Time Environment

With the uVision Project Manager and Run-Time Environment you create software application using pre-build software components and device support from Software Packs. The software components contain libraries, source modules, configuration files, source code templates, and documentation. Software components can be generic to support a wide range of devices and applications.

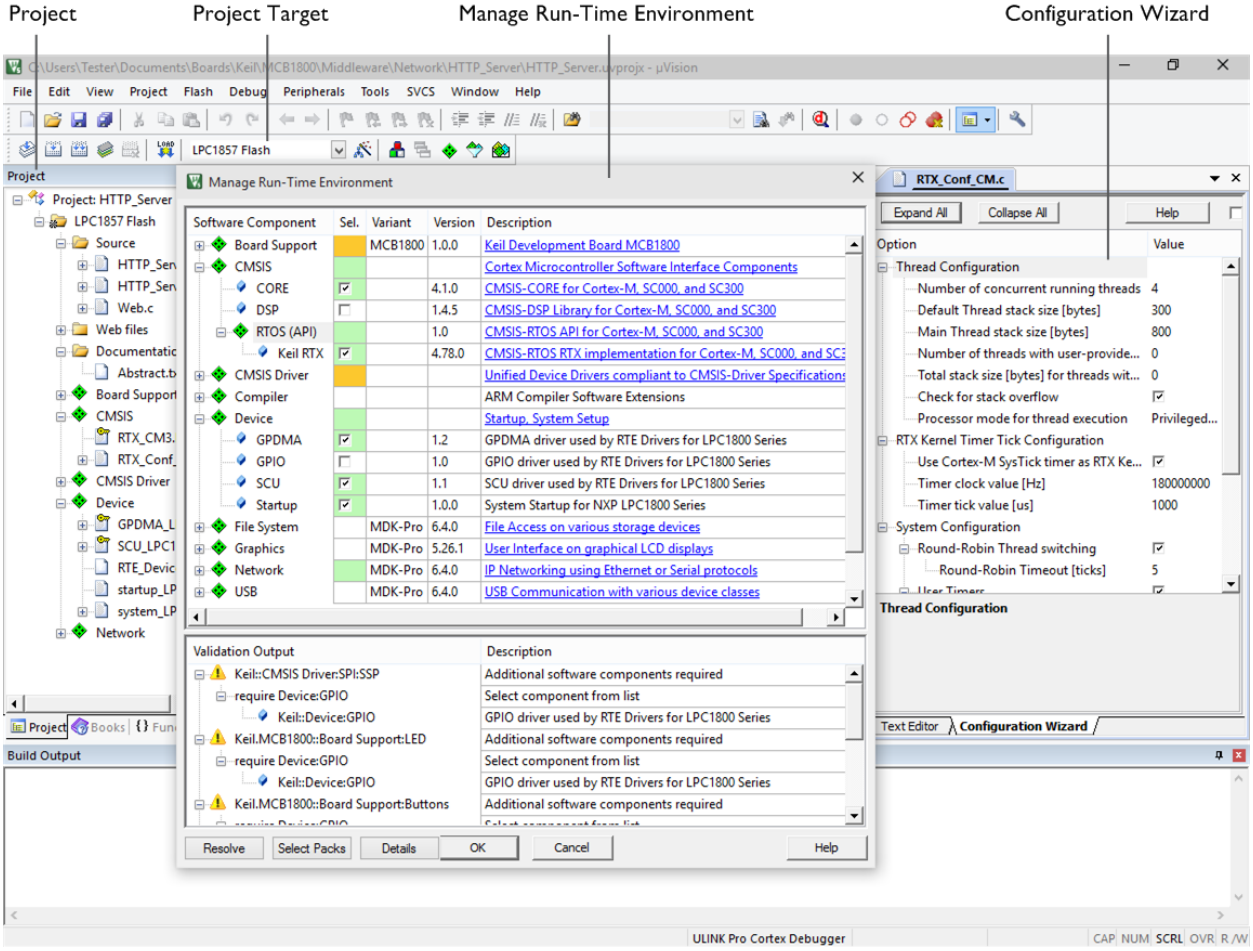


Figure 3. Project Manager & Runtime Environment

- The **Project** window shows application source files and selected software components. Below the components you can find corresponding library and configuration files.
- **Projects** support multiple **targets**. They ease configuration management and may be used to generate debug and release builds or adoptions for different hardware platforms.
- The **Manage Run-Time Environment** window shows all software components that are compatible with the selected device. Inter-dependencies of software components are clearly identified with validation messages.
- The **Configuration Wizard** is an integrated editor utility for generating GUI-like configuration controls in assembler, C/C++, or initialization files.

1.3 uVision Editor

The integrated uVision Editor includes all standard features of a modern source code editor and is also available during debugging. Color syntax highlighting, text indentation, and source outlining are optimized for C/C++.

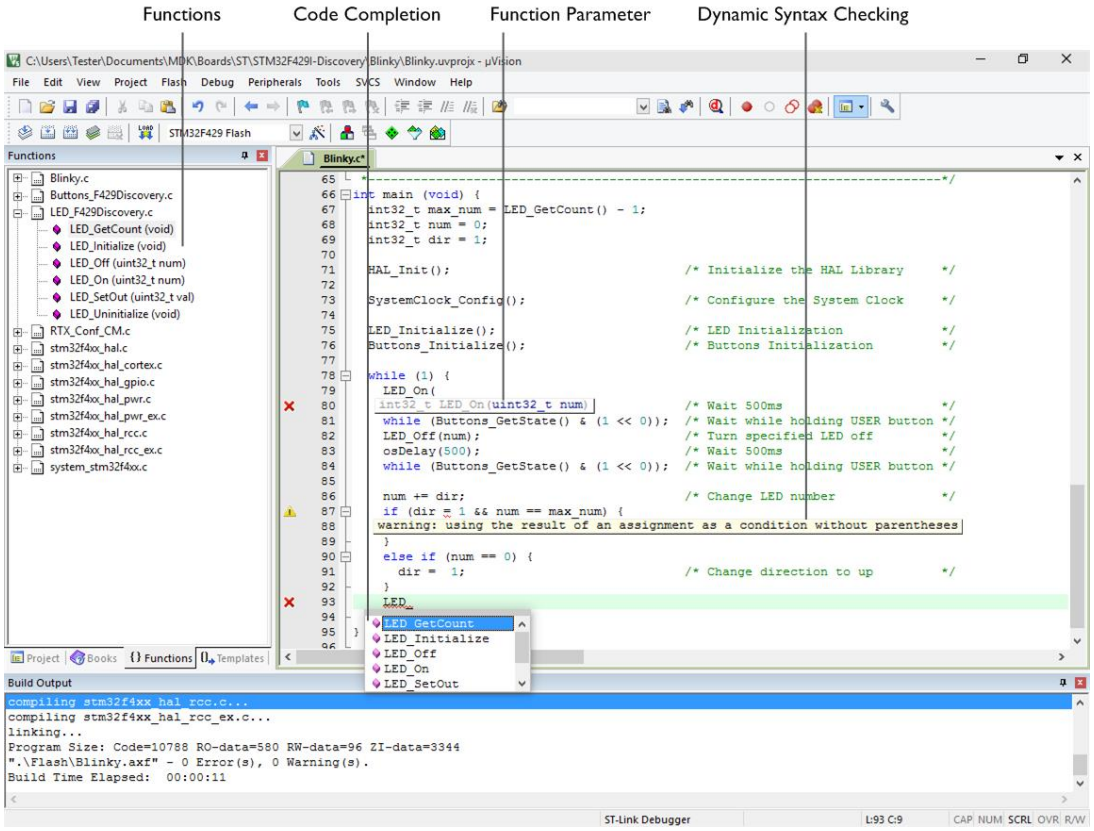


Figure 4. uVision Editor

- The **Functions** window gives fast access to the functions in each C/C++ source code module.
- The **Code Completion** list and **Function Parameter** information helps you to keep track of symbols, functions, and parameters.
- **Dynamic Syntax Checking** validates the program syntax while you are typing and provides real-time alerts to potential code violations before compilation.

2 Part Specifications

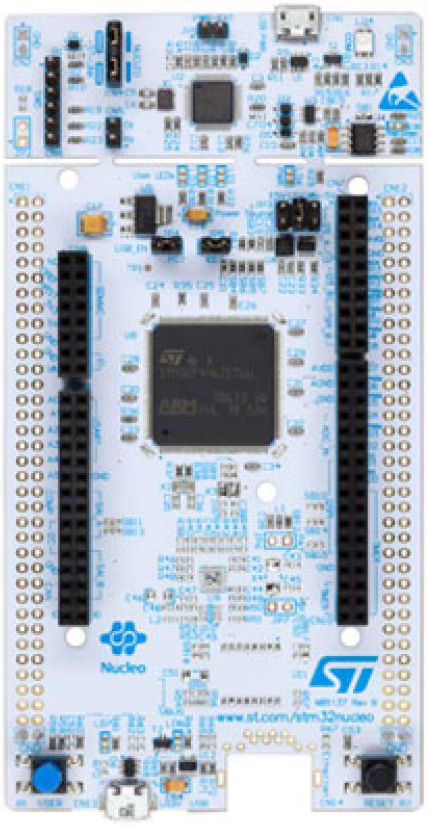


Figure 5. Aristurtle uController Development Board

Kit Name	STM32F4 Nucleo Board
Part Number	STM32F446 Nucleo Board
MCU Name	STM32F446ZET6U
Memory	512kB

3 Component Overview

3.1 Instruction Set Architecture (ISA)

The MCU is based on the **ARM Cortex M4** architecture.

3.1.1. Features

Word length	32 bits
Floating Point Unit (FPU)	No
Nested Vector Interrupt Controller (NVIC)	Yes
JTAG/SW debug	Yes
Embedded Trace Macrocell (ETM)	Yes
Memory Protection Unit (MPU)	Yes
Digital Signal Processing Unit (DSP)	Yes

3.1.2 Resources

3.1.2.1 Documents

- User Guide
- Technical Reference

3.1.2.2 Links

- ARM Website
- ST MCU Website
- ST Board Website

3.2 STM32F446ZET6U Microcontroller

3.2.1 Features

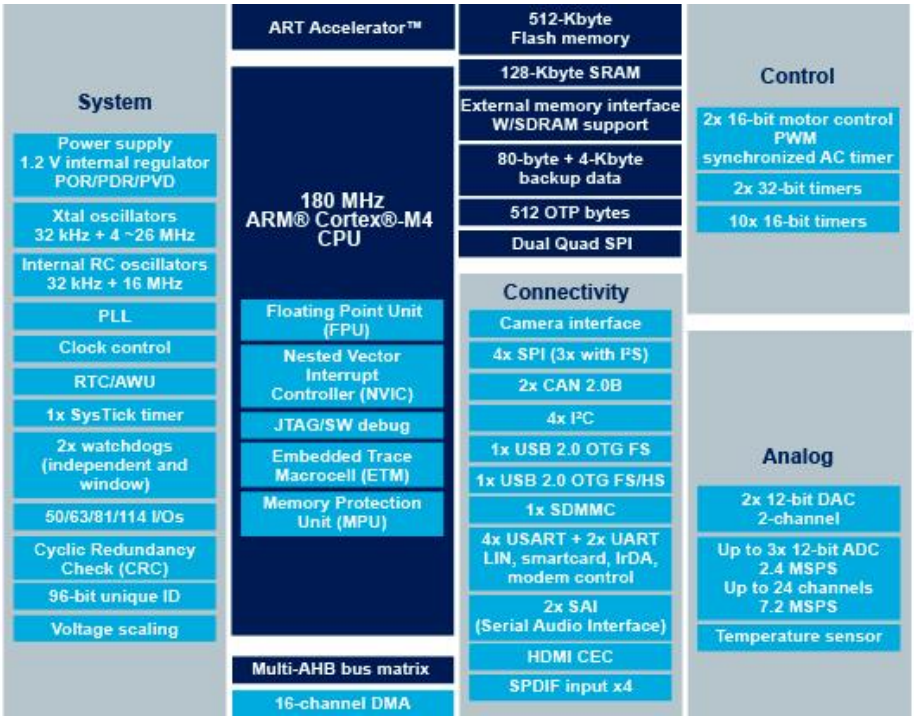


Figure 6. uController internal structure

Core: ARM® 32-bit Cortex®-M4 CPU with FPU, Adaptive real-time accelerator (ART Accelerator™) allowing 0-wait state execution from Flash memory, frequency up to 180 MHz, MPU, 225 DMIPS/1.25 DMIPS/MHz (Dhrystone 2.1), and DSP instructions.

Package	LQFP144
Flash Memory	512kB
SRAM	128kB
External Memory Controller	Yes
Dual Mode Quad SPI	Yes
LCD Parallel Interface	Yes
Frequency	26MHz Crystal (up to 180MHz)
Low Power Modes	Sleep, Stop, Standby Modes
ADC	3x12-bit, 2.4 MSPS ADC: up to 24 channels and 7.2 MSPS in triple interleaved mode
DAC	2x12-bit D/A converters
DMA	16-stream DMA controller with FIFOs
Timers	2xWatchdog, 1xSysTick timer, 12 16-bit & 2 32-bit timers up to 180MHz

Debug Mode	SWD & JTAG Interface, Cortex®-M4 Trace Macrocell™
I/O Ports	114 with Interrupt Capability
I^2C	4
USART	4
SPI	4
I^2S	3
SAI (serial audio interface)	2
CAN (2.0B Active)	2
SDIO Interface	1
USB 2.0 full-speed device/host/OTG controller with on-chip PHY	1
8-14 bit parallel camera interface up to 54Mbytes/s	1
CRC calculation unit	
RTC	
96-bit unique ID	

3.2.2 Resources

3.2.2.1 Documents

- Datasheet
- Application note
- Errata Sheet
- Programming Manual
- Reference Manual
- Flash memory programming manual
- Nucleo Board Manual

Figure 7. Chip Package

3.3 STM32F446 Nucleo Board

3.3.1 Features

USB	Yes
On-Chip Programmer	Yes
Buttons	**2**

LEDs	**3**
LCD Display	**No**
Accelerometer	**No**
Microphone	**No**

3.3.2 Getting Started

- We install the STSW-LINK007 for our PC.
- We conduct a firmware update of our board (STSW-LINK009).

3.3.2 Processor Interconnection

- Red LED – LD3: PB14
- Blue LED – LD2: PB7
- Green LED – LD1:
- User Button (Blue): PC13
- Reset Button (Black) – NRST

4 Keil MDK

For the programming of the ARM Microcontroller we are going to use **Keil MDK 5** of ARM which includes an IDE, Debugger, compiler, source files and other features that we are going to use. It runs only on windows platform and its free version allows a maximum of 32Kb Flash Memory. Before we use Keil we need to know some things about the programming process of the microcontroller which is:

1. **Build** of the code (compile & link). For this we need:
    - Our code.
    - *Header files* for our processor (contain declaration of constants and other values like special registers in the program memory).
    - *Startup files* for our processor (contain source code which initialize our processor such as clock configuration).
    - *Driver files* if we want to use special libraries (SPL or HAL or any other library) for some functions.
2. **Download/Flash** of the executable code in the microcontroller. Specifically, this stage requires:
    - **Erase** of the existing code.
    - **Programming** the new code.
    - **Reset & Run** of the loaded program.

The download is a complicated procedure and so we are going to do it in detail in a next chapter. This procedure is being done by the ST-Link Debugger, which is a smaller Cortex-M3 microcontroller in the upper place of the main microcontroller board which is responsible for the programming of the Cortex-M4 main microcontroller.

4.1 Installation

4.1.1 Keil-ARM MDK IDE v5

First, we need to download the Integrated Development Environment for the programming of the microcontroller with the procedure that we mentioned before.

4.1.2 MDK Legacy Support

Second, MDK Legacy Support is needed only if you want to open projects created with previous versions of the Keil uVision.

4.1.3 STM32CubeF4

Thirdly, we need the STM32CubeF4 in order to open examples of sample projects to check the operation of our IDE. To do that we need to go to st.com and type “cubef4” in the search bar and then download the file. Or else use this link.

4.1.4 STSW-LINK009

Moreover, we need to download STSW-LINK009 which is a USB driver for ST-LINK/V2 and ST-LINK/V2-1 boards and derivatives of the boards (Nucleo etc.).

4.1.5 STSW-LINK007

In addition to this we need to download the STSW-LINK007 which is an application which is used to upgrade the firmware of the ST-LINK, ST-LINK/V2 and ST-LINK/V2-1 boards through the USB port.

After all the main downloads we are going to test if our program can download programs to the microcontroller. To do this go to STM32CubeF4\Projects\STM32F446ZE-Nucleo\Examples\GPIO\GPIO_EXTI\ MDK-ARM and then open the Project.uvprojx file.

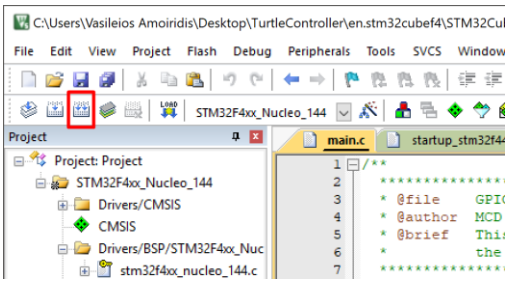


Figure 8. Compiling of the code.

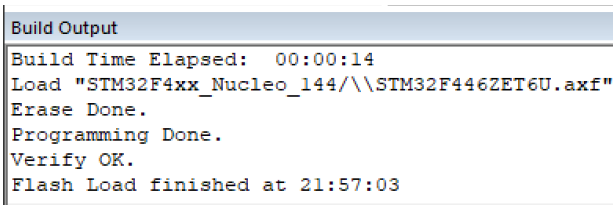


Figure 9. Build Output of successful program.

For the program to be uploaded in the microcontroller the following message must be shown in the **Build Output.** This means that the setup is correct.

<u>If it does not upload, then some of the following must be wrong.</u>

4.1.6 Options for Target "STM32F4xx_Nucleo_144"

In options you need to go to the device tab and choose the STM32F446ZETx. Except from this you need to download the right software pack for your device which is stated a little bit right from the search bar.

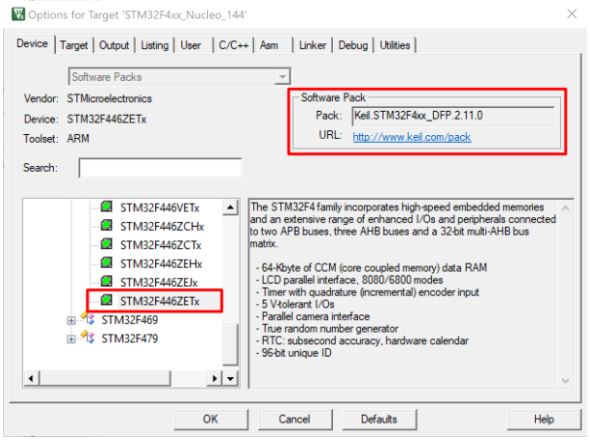


Figure 10. Device and Pack Selection.

4.1.7 Pack Installer

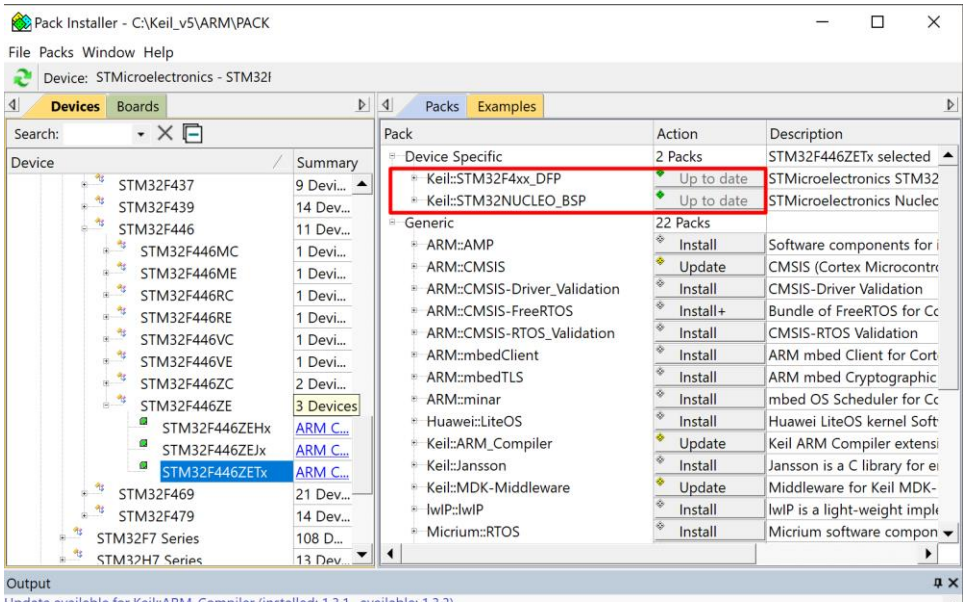


Figure 11.Selections in the pack Installer.

4.1 Create a new Project

For the creation of a new project we need to take care of all the above details in the programming of the microcontroller. In order to create a new project, you have to click the 'Project' tab and then click the "New uVision Project…' option. Save the project in a local file and continue the installation.

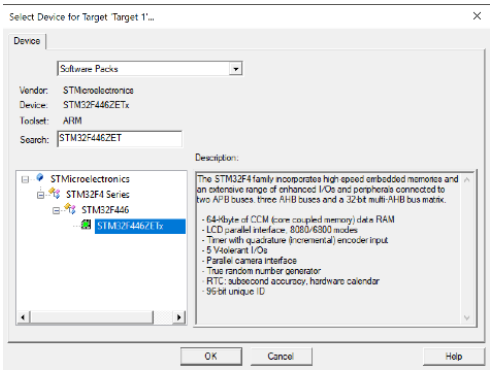


Figure 12. Device Selection

After this we must set the 'Target Board' for our program to be downloaded so in the **Device Selection** we choose STM32F446ZETx.

Subsequently we need to choose the drivers for our project. This is done with the help of the Run-Time Environment that we mentioned before. This is a very important step because if we don't choose the right drivers for our project we are not going to be able to program it in the way that we want because there are a lot of different things that do almost the same thing.

First of all, there is the Board Support option. The Board Support option is valid only for certain discovery boards, so we are never going to use something from it.

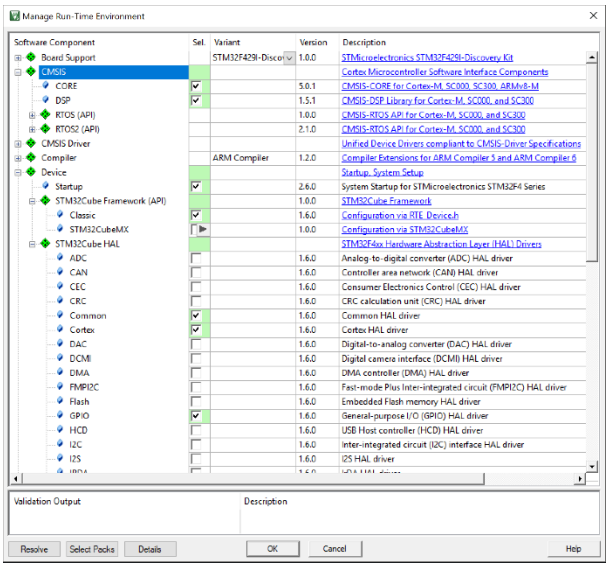


Figure 13. Run-Time Environment Configuration

Secondly, there is the *CMSIS* option. In this option we have the ability to choose 4 options: *CORE*, *DSP*, *RTOS*, *RTOS2*. We must certainly choose the *CORE API* because this is the final layer of abstraction before the hardware. The *DSP* consists of libraries for Digital Signal Processing Features that we don't currently need. The other 2 options are for RTOS implementation.

The next 2 options are *CMSIS Driver* and *Compiler*. From the *CMSIS Driver* we are not going to use something ever because there ST's HAL Libraries. In the *Compiler* option we may choose the *Event Recorder* in case that we need to debug our code.

Finally, in the *Device* option we must definitely check the *Startup* option otherwise our microcontroller will never start to work and from the *STM32Cube Framework* and *STM32Cube HAL* we should always choose the following: Common, Cortex, GPIO, PWR, RCC. If we don't use one of these then the green tick-box will become yellow and hit us with a warning.

Now that our project is open in the *Project Toolbar* we need to add the file that we are going to write our code in it. To do this we select the *'Source Group 1'* and create a new C file with the name **"main.c"**. After this step, there are some minor configurations that need to be done in order to set the environment for your project.

In the *'Options for Target'* Tab we need to do the last modifications. The first one, is to set the right frequency of our microcontroller in order to work properly. The right frequency is 8MHz because it uses the external Crystal Oscillator. In the same tab all the memories of the microcontroller are shown, and one can modify them in every way that he wants (not recommended if you don't know what you do).

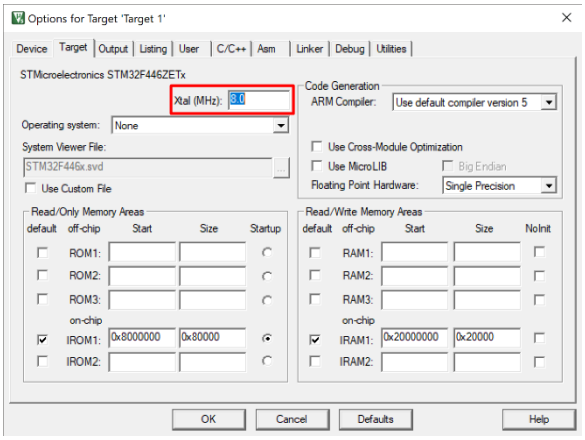


Figure 14. Target1 Configuration

In addition to this we need to set the *Debug Options*. In the *Debug* tab we need to choose the ST-Link Debugger instead of the standard one but most importantly in its settings we need to be sure that **Port** is set to **SW** and not to **JTAG**.

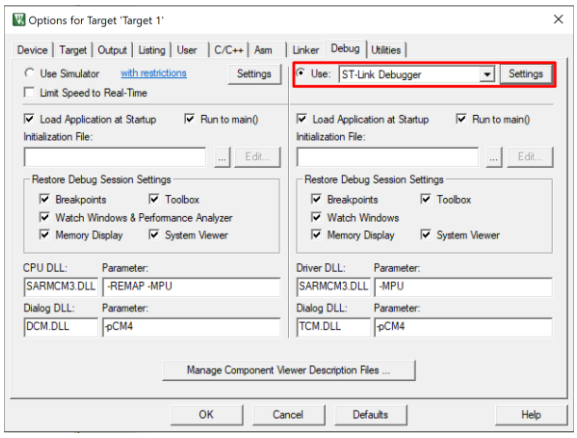


Figure 15. Debug Options and Settings.

Finally, in the *Utilities* Tab we need to uncheck the *Use Debug Driver* and select once again the ST-Link Debugger. Furthermore, if we want our program to run immediately after it is downloaded in the microcontroller and don't require a reset, we need to check the *'Reset and Run'* checkbox in the settings of the *Target Driver for Flash Programming*. After all this work our Integrated Development Environment is ready to be used in order to program our microcontroller.

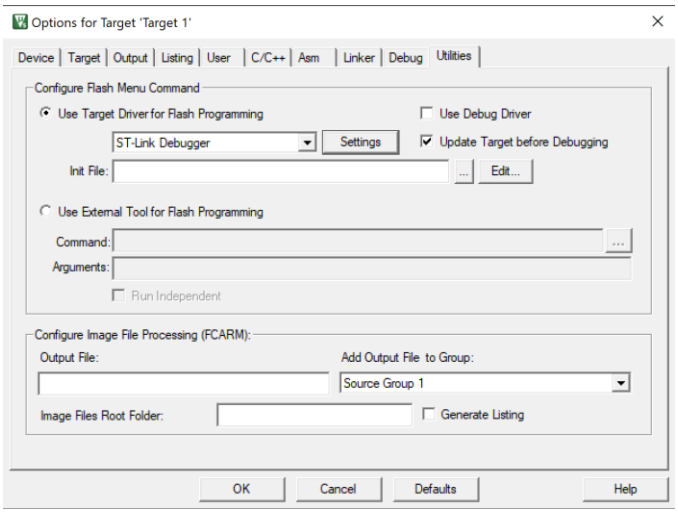


Figure 16. Utilities Configuration

5 STM32CubeMx

5.1 Introduction

STM32CubeMx is a graphical tool that allows a very easy configuration of STM32 microcontrollers and the generation of the corresponding initialization C code through a step-by-step process. It embeds comprehensive STM32Cube MCU Packages for STM32 microcontroller series (STM32CubeF4 in our case). These packages include the:

- STM32Cube HAL which is an abstraction layer embedded software ensuring maximized portability across the STM32 portfolio.
- STM32Cube LL which is low-layer APIs, a fast, light-weight, expert-oriented layer
- A consistent set of middleware components such as RTOS, USB, TCP/IP and graphics.

All the embedded software utilities are delivered with a full set of examples. The program generates C code which is ready to be used in the Keil uVision Environment.

5.2 Installation

First, we need to download the STM32CubeMx from here which is located in the bottom of the page. We click the get software option and it is automatically downloaded in a special folder.

Part Number	Software Version	Marketing Status	Supplier	Order from ST
STM32CubeMX	4.23.0	Active	ST	Get Software

Figure 17. Location of the download link

In addition to this, we unzip the file and we run the SetupSTM32CubeMX-4.xx.0.exe and we click next in every option that comes. After that, we need to download the software packages. For this we should go to Program Files\ STMicroelectronics\STM32Cube\STM32CubeMx and we run the STM32CubeMX.exe and this window opens. After this we click this option and choose the latest version of the STM32CubeF4 Package and we are almost done.

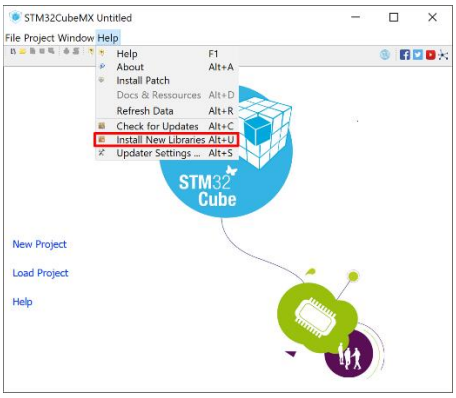


Figure 18. New software packages

5.3 Create new Project

For the creation of a new project with the STM32CubeMx we select the option "New Project". After this, we need to select our board or our MCU. For our case, it's easier to choose the Board Selector option and choose the STM32F446ZE board with a double click.

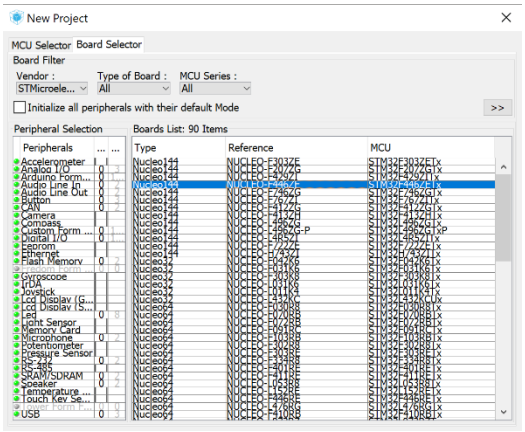


Figure 19. Board Selection for our Project

If we did everything correct then this must be showed in our screen.

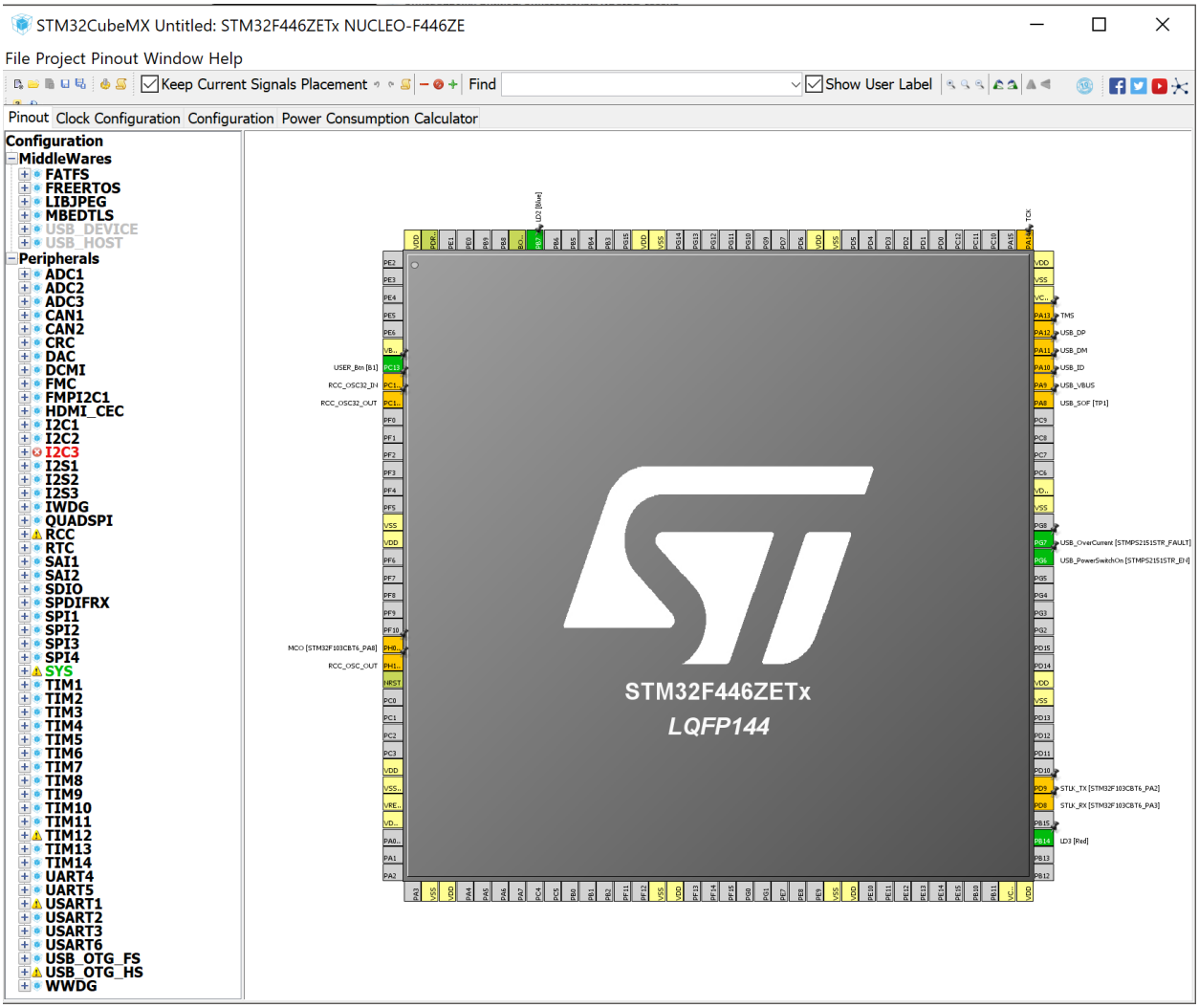


Figure 20. Startup screen in STM32F446ZE

Now we choose the functions on the left that we want to use, and then after we have set everything we click on the *Project* bar. In the settings option we find the destination folder for our project and then we click the option *Generate Code*. Then our code is uploaded to the Keil uVision and we write the appropriate functions before we upload it to our uController.

6. First Project – LED Blink

The purpose of this project is to learn the process of uploading a program to our microcontroller and also, to learn some basics about it's programming. In order to achieve this, we are going to blink 2 LEDs opposite to one another.

6.1 STM32CubeMx configuration

We open the STM32CubeMx and we create a new project with our microcontroller. After doing this, we are going to clear the pinout because it has some predefined functions that we don't want to deal with right now. In the *Pinout* bar we choose the option *Clear Pinout* and our microcontroller's pins will be cleared. So, we are ready to start configurating our peripherals.

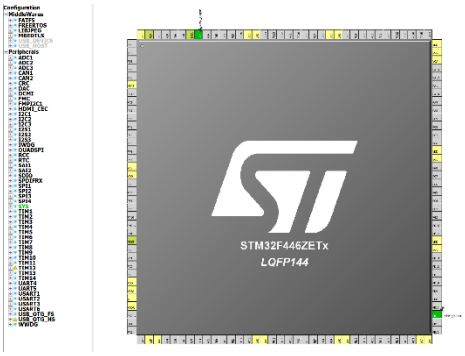


Figure 21. CubeMx main window

The peripherals that we are going to use for our project are LED_Blue in PB7 and LED_Red in PB14. To set those pins as outputs we need to click on them and choose the *GPIO_Output* option. After this we go to the *Configuration* bar and we select the GPIO option. We find PB7 (or PB14) and we click on it in order to set its *GPIO Output level* to HIGH. With this we set it's first state to HIGH while the PB14 is LOW, so they are going to flash in the opposite way.

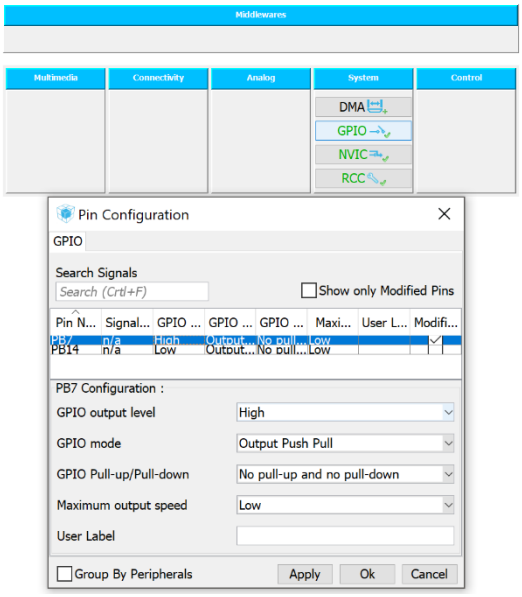


Figure 22. Pin Configuration

Now, after we chose our peripherals we need to configure some things about the coding part. In the taskbar we click on the *Project* bar and we click on the *Settings*. In the *Project* bar we give a name to our project and choose the destination folder. Then we need to choose the destination IDE for our program to be generated. We choose MDK-ARM V5. In the Code Generator bar, we choose the "Add necessary library files as reference in the toolchain proj…" and then click ok. Now we are ready to generate our Code. We click again in the Project bar and then we click "Generate Code". Now our work is transferred to the Keil uVision IDE.

6.2 Keil uVision MDK-ARM v5

All the appropriate code for our peripherals is generated and now it's time to give action to our peripherals. We want to toggle them together at the same time so we are going to use a specified Library for this:

```
HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

In this function the first parameter is the Port of our Pin and the second is the number of our pin. Another function that we are going to use is a simple delay function for the blinking:

```
void HAL_Delay(__IO uint32_t Delay);
```

This function uses only one parameter which is the milliseconds of the delay that you want to achieve.

Finally, we get to write our code inside the endless loop of our code which is the **while(1)** loop inside the **main()** function. After we do this we build our code and then upload it to the microcontroller. If we want to change something in our code we need to rebuild it before uploading it again in the microcontroller.

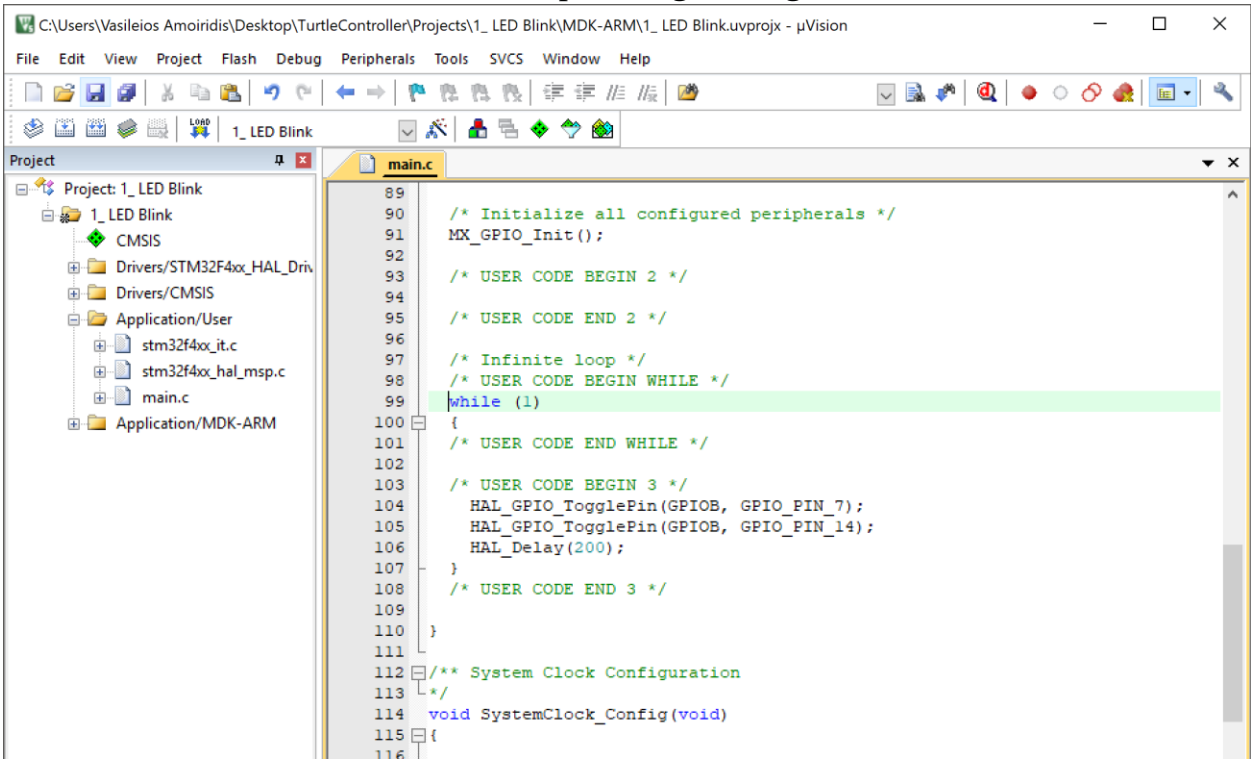


Figure 23. Final Code

7. LED Blink v2

The purpose of this project is to learn how to read digital values from external sources such as buttons and how to configure time delays without blocking the rest of the code to be executed. In order to achieve this, we are going to blink an LED continuously as before, but we are going to do it with a function called

```
HAL_GetTick();
```

Which is exactly the same as millis(); function in the Arduino Software. Moreover, we are going to light an LED when we press the user button in PC13 and to achieve this we need a non-blocking delay function.

7.1 STM32CubeMx

As before we need to configure our 2 LEDs as outputs but this time we also need to set PC13 as input. Because of the fact that we have a button we need to know what resistance should we use for understanding when the button is pressed or not. In the schematic files of the microcontroller we can see that the User Button in PC13 is directly connected in the V_{DD} so we need to use the pull down resistance of the current pin in order to sense the press of the button.

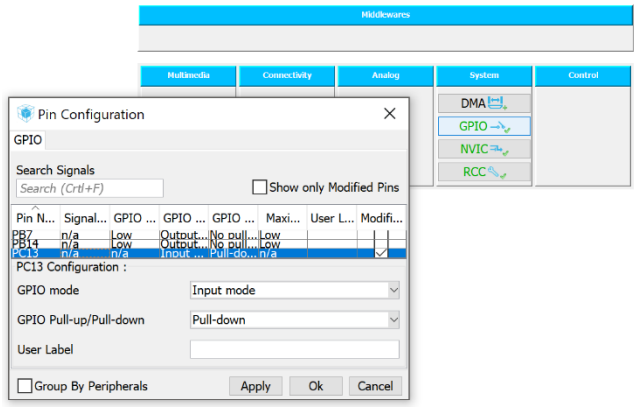


Figure 24. Set pull-down resistor

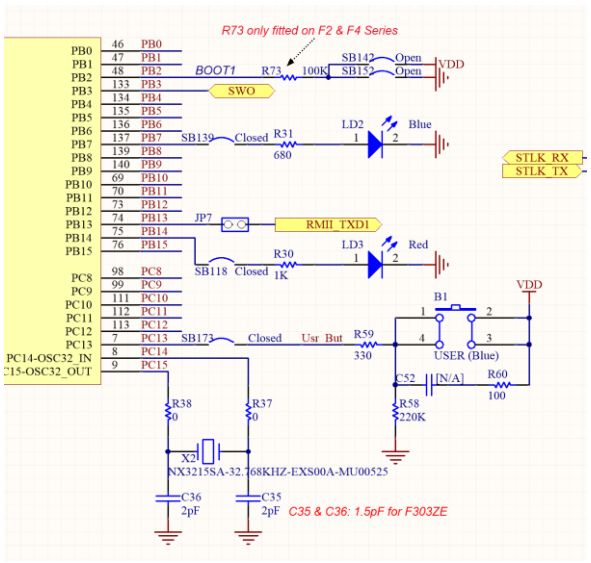


Figure 25. User Button Schematic

7.2 Keil uVision MDK-ARM v2

In order to read the state of the button we are going to use this function:

```
HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

If the button is pressed (short circuit to supply) then its binary value is 1 or in the definition in the STM32 HAL Libraries is equal to `GPIO_PIN_SET`. On the contrary if the button is not pressed (pull down resistor) its binary value is 0 or in the definition in the STM32 HAL Libraries is equal to `GPIO_PIN_RESET`.

Now in order to light an LED we need to be able to give it a digital value except for just toggling it. The function needed for this is the:

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

As before, if we want to light the LED which means to set the pin we need to assign to it the `GPIO_PIN_SET` value or else if we want to clear it the `GPIO_PIN_RESET` value. So the code must look like this:

```
 99   while (1)
100   {
101   /* USER CODE END WHILE */
102
103   /* USER CODE BEGIN 3 */
104     if(HAL_GetTick() - cnt > 3000)
105     {
106       HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_7);
107       cnt = HAL_GetTick();
108     }
109
110     if(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_SET)
111     {
112       HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET);
113     }
114     else
115     {
116       HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_RESET);
117     }
118   /* USER CODE END 3 */
119
120   }
```

Figure 26. Final Code

8. LED Blink v3

The purpose of this project is to learn how to use the external interrupt of pins. To do this we are going to toggle one LED continuously and we are going to toggle another one each time there is an external interrupt in the PC13 (User Button).

8.1 STM32CubeMx

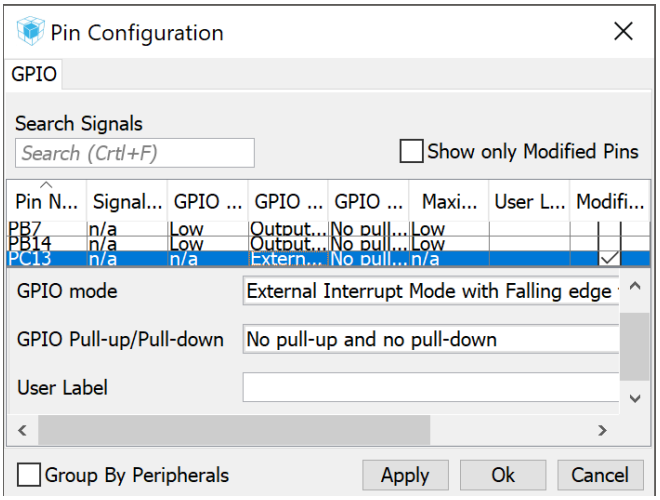


Figure 27. Selection for triggering the Interrupt

Like before, we are going to configure our 2 LEDs as outputs. In addition to this, the User Button in PC13 is going to be used as an external interrupt pin. We are going to configure it as a falling edge detection interrupt because we want to change the state of the LED exactly when the button is released.

Except for this, this time in the program we need to enable the External Interrupt Vector in order to be able to use the Interrupt in PC13. The Interrupt that we need to enable is the EXTI line15:01 interrupts.

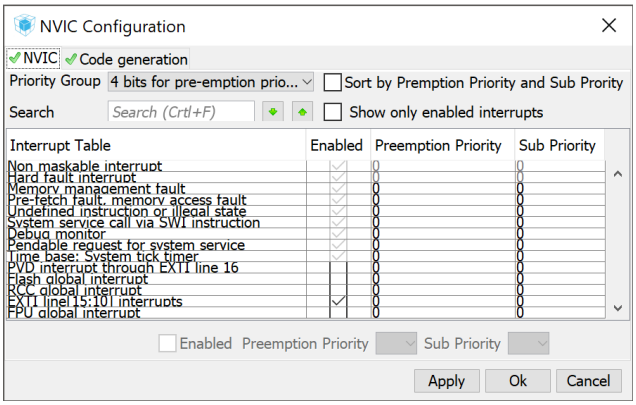


Figure 28. Enable of the External Interrupt

8.2 Keil uVision MDK-ARM v5

As before we configure one LED to flash continuously and we take care of the secondary action. In order to handle the Interrupt, we need to open the stm32f4xx_it.c and to find the

```
void EXTI15_10_IRQHandler(void);
```

Inside this function we are going to write our code in case of an Interrupt. In our case we want to flash an LED.

9. LED Blink v4

The purpose of this project is to learn some basic things about the Timer of the microcontroller. In order to do this, we are going to blink our LEDs with the help of the Timer1 every 1 second.

9.1 STM32CubeMx

Like before we configure our 2 LEDs as outputs and then we have to set Timer1 for our needs. In order to do this firstly we choose the clock source of the *TIM1* to be the *Internal Clock*. In the *Configuration* tab we configure our pins as outputs.

Moreover, in the TIM1 tab in the Parameter Settings we need to set the Prescaler value in 1024-1, configure the Counter in Up mode, set the Period to 15625-1. This happens because of the fact that our Clock Frequency is set to 16MHz so we need to set a Prescaler to stretch the clock cycles to our needs. The counter period is set exactly for 1 second. Moreover, we need to enable the TIM1 update interrupt in the NVIC Settings.

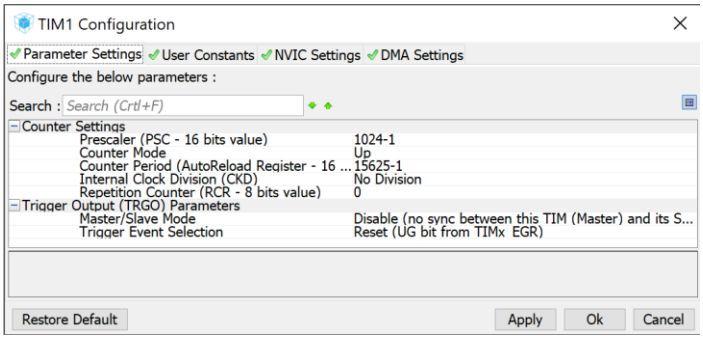


Figure 29. Timer Configuration

9.2 Keil uVision – MDK-ARM v5

First of all, we need to enable the timer. To do this we must call this function:

```
HAL_TIM_Base_Start_IT(&htim1);
```

If we wanted to enable the 2nd Timer we would write &htm2. Into the while(1) we are not going to write anything because we are going to use the Update Interrupt of the TIM1. To do this we are going to use the:

```
HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim);
```

Inside this function we are going to toggle our pins.

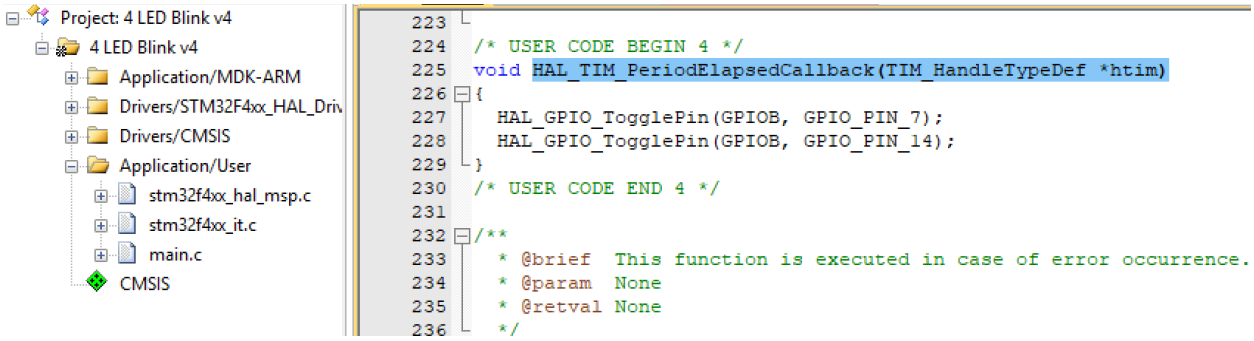


Figure 30. Code Implementation

10. ADC (!!!Your attention here please!!!)

The purpose of this project is to learn how to use the ADC of the microcontroller in single and interrupt mode. In order to achieve this, we are going to read the analog value of a potentiometer from the 3 different channels that our microcontroller offers.

10.1 STM32CubeMx

This time in our implementation we are going to use the PA3 pin of our microcontroller or the A0 pin of the pinout in the development board. This confusion is happening because these pins are Arduino Compatible. In order to configure PA3 for analog input, we need to enable from ADC1, ADC2, ADC3 the IN3 input. We are also going to configure all ADCs for single continuous conversion mode. This is happening because we want to continuously measure the value of the input in the selected pin.

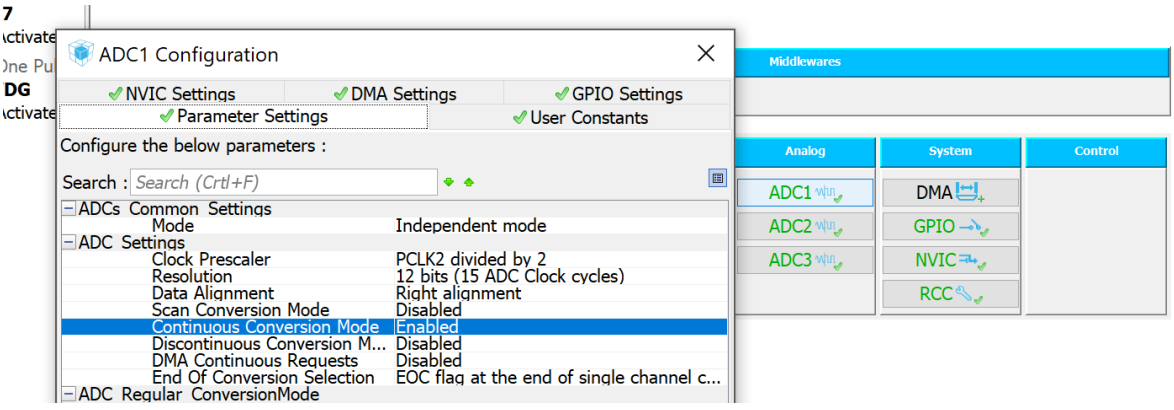


Figure 31. Configuration of the ADCs.

10.2 Keil uVision

After our generated code, we must initiate 3 values of the **uint16_t** type for the 3 Channels of the ADCs. Then into the while(1) function we are going to implement our code for reading analog values. The first function that we are going to use is the:

```
HAL_ADC_Start(&hadcx);
```

This function enables the x ADC Channel.

After this function we need to wait after the conversion is completed in order to read the value from the ADC register. To achieve this we need to call the:

```
HAL_ADC_PollForConversion(&hadcx,y);
```

In this function x defines the channel of the ADC that we chose and y is the time delay in ms that we wait for the ADC to make the conversion.

Now that we are ready to read our analog value we are going to read it with the

```
adcValuex = HAL_ADC_GetValue(&hadcx);
```

Now that we have our value we need to stop the conversion and let the next conversion start.

```
HAL_ADC_Stop(&hadcx);
```

Now with the exact same functions we can continue reading the values from different channels.

```
110   while (1)
111   {
112   /* USER CODE END WHILE */
113
114   /* USER CODE BEGIN 3 */
115     HAL_ADC_Start(&hadc1);
116     HAL_ADC_PollForConversion(&hadc1,1);
117     adcValue1 = HAL_ADC_GetValue(&hadc1);
118     HAL_ADC_Stop(&hadc1);
119
120     HAL_ADC_Start(&hadc2);
121     HAL_ADC_PollForConversion(&hadc2,1);
122     adcValue2 = HAL_ADC_GetValue(&hadc2);
123     HAL_ADC_Stop(&hadc2);
124
125     HAL_ADC_Start(&hadc3);
126     HAL_ADC_PollForConversion(&hadc3,1);
127     adcValue3 = HAL_ADC_GetValue(&hadc3);
128     HAL_ADC_Stop(&hadc3);
129   }
```

Figure 32. Code Implementation

11. STMStudio

STMicroelectronics offers a very useful piece of software in order to monitor the values of your system. First of all, the program is downloaded from here in the bottom of the page. The program is very easy to be installed. In order to display a value, we right click in this area and then we click import.

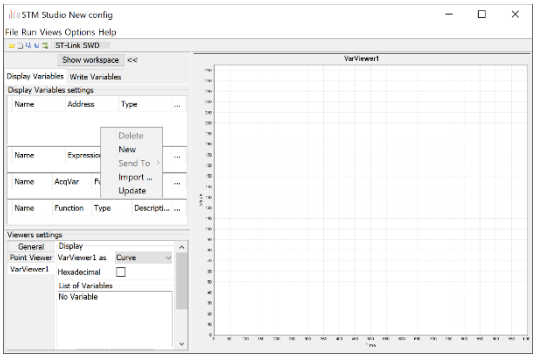


Figure 33. STMStudio Configuration

After we click import we need to select the executable file that contains the value that we want to monitor. This file has a .axf extension and is located inside MDK-ARM/ProjectName/ folder. After this, we find our value in the drop down list

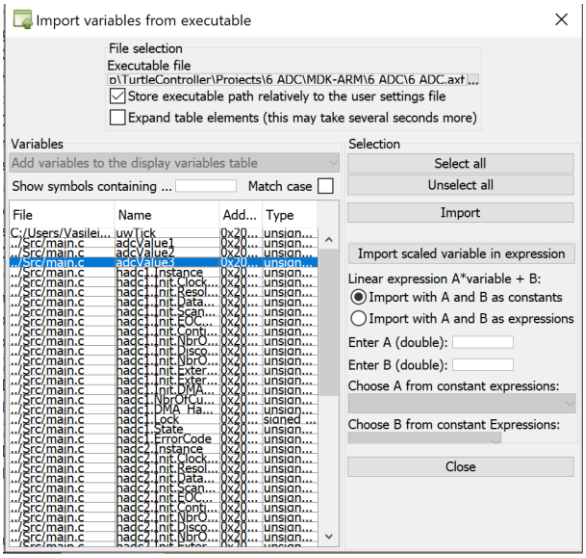
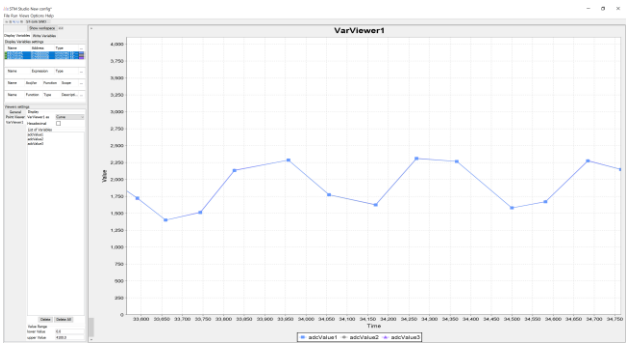


Figure 34. Selection of Values.

After you selected the values that you want to monitor you need to click import in order to import the values to the main screen of the program. Then you right click the values, select the option *Send To* and then the option *VarViewer*. Then you can monitor your values in the main screen in 3 different ways, either graph, bar or bare value.



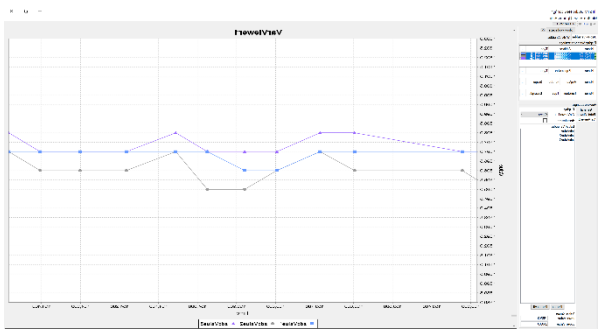


Figure 35 & 36 Visualization of data, & Zoom in for Actual data

12. CAN Communication

This purpose of this project is to learn the basics about CAN protocol implementation in our microcontroller. In order to achieve this, we are going to use the Loopback Mode that the microcontroller offers in CAN mode so as to not need another node in the CAN Bus.

12.1 STM32CubeMx

The main thing here is the initialization of the CAN Interface and then the bit timing of the CAN message. In order to keep things simple, we are going to leave the transmission and reception of messages as simple as possible without any interrupts or complicated stuff like this. The new thing here is the bit timing.

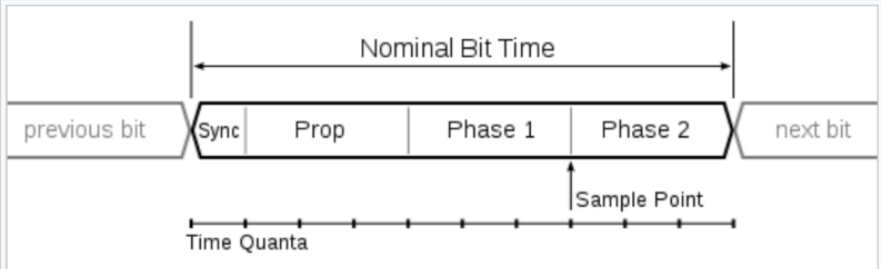


Figure 36. Bit Subdivision.

Bit timing is the separation of the CAN bit in smaller time pieces called “quanta”. In the pictures is clearly marked that the bit is divided in 4 pieces. The sync phase that the 2 nodes try to sync together, the propagation of the message and then phase 1 and phase 2. The sample point is set exactly between phase 1 and phase 2. In the STM32 CAN bus you need to set the SJW time quanta, BS1 and BS2 which are the 2 phases. The propagation phase is included inside phase 1. It is most commonly suggested to put the sample point near to 75% of the bit. It is also proposed to use only 1 synchronization time quanta. The way to calculate the appropriate values is the following. First of all, you need to know the clock speed in the peripheral that CAN gets its clock. In our case it is the APB1 Clock peripheral. So, you take that clock speed you divide it with the Prescaler that you set in your CAN initialization and then divide it with your desired baud rate. The result must be an integer. For example, the speed in the APB1 Peripheral is 16MHz, the CAN Prescaler is set to 2 and you want a 500kbps baud rate, so the result is 16 time quanta. A good division

for each bit time operator is SJW = 1, BS1 = 11, BS2 = 4. With this initialization the result is exactly 75%.

12.2 Keil uVision MDK-ARM 5

After our code is generated from Cube, we get to write the appropriate functions to test the CAN interface.

First of all, inside the `MX_CAN1_Init(void)` we set the parameters for the CAN filter. Because of the fact that we don't want to make things more complex right now and filter our messages IDs with hardware rather than software we are going to use the default settings of the CAN filter. After this, we need to create some variables of a certain message type which are going to hold all the required "peripheral" data along with the actual data of a CAN message.

```
sFilterConfig.FilterNumber = 0;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0000;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = 0;
sFilterConfig.FilterActivation = ENABLE;
sFilterConfig.BankNumber = 14;
```

Figure 37. CAN Filter Configuration.

```
static CanTxMsgTypeDef        TxMessage;
static CanRxMsgTypeDef        RxMessage;
hcan1.Instance = CAN1;
hcan1.pTxMsg = &TxMessage;
hcan1.pRxMsg = &RxMessage;
```

So, after we created the two variables `(TxMessage, RxMessage)` which hold the data of a CAN message we assign their addresses to the Transmit and Receive pointers of the respected CAN Instance in order to be able to be Transmitted or Received with the appropriate CAN format.

Inside the main() function we "build" the CAN message, giving the values of each field of the

```
hcan1.pTxMsg->StdId = 0x100;
hcan1.pTxMsg->RTR = CAN_RTR_DATA;
hcan1.pTxMsg->IDE = CAN_ID_STD;
hcan1.pTxMsg->DLC = 2;
```

Figure 38. CAN message fields' configuration

message by reading the reference manual given by ST. In our case the appropriate values are shown in Figure 38. Now in order to send a 2 byte message we just need to assign the bytes to the data field of the can message in the same way we did before.

```
hcan1.pTxMsg->Data[0] = 0xCA;
hcan1.pTxMsg->Data[1] = 0xFE;
```

After we framed our message we want to transmit it. This action is implemented by calling the function:

```
HAL_CAN_Transmit(&hcan1, 10);
```

The first argument is the address of the CAN Instance that we use and the second argument is the timeout value for the specific action. The final thing that needs to be done to test our code is a way to read all these transmitted messages, check if they are correctly transmitted. To achieve this a very simple example is given. We compare the values of the associated message with the expected values and if they are all the same we just toggle an LED for every correct message that we receive just for a visual confirmation.

```
if(HAL_CAN_Receive(&hcan1,CAN_FIFO0,10) == HAL_OK)
{
  if(hcan1.pRxMsg->StdId == 0x100)
  {
      if(hcan1.pRxMsg->IDE == CAN_ID_STD)
      {
        if(hcan1.pRxMsg->DLC == 2)
        {
          if(hcan1.pRxMsg->Data[0] == 0xCA)
          {
            if(hcan1.pRxMsg->Data[1] == 0xFE)
            {
              HAL_GPIO_TogglePin(GPIOB,GPIO_PIN_7);
              HAL_Delay(100);
            }
          }
        }
      }
  }
```

Figure 39. Message Reception.