How to STM32

A how-to-guide on software development with the STM32F411E-DISCO board in C/C++ using STM32CubeMX and STM32CubeIDE.

Special Course
Spring 2022

Member:	**Student ID:**
Christian Jannik Metz	(s202460)
Mikkel Haugaard-Hansen	(s170608)

Technical University of Denmark
Kongens Lyngby
DTU Photonics Engineering

Date: 28/05-2022

Contents

1 Introduction

This is a tutorial guide on how to program STM32 microcontrollers. We focus on introducing the basic concepts of microcontroller programming with the STM32CubeMX and STM32CubeIDE pipeline.

This guide is not requiring any prior knowledge in microcontroller programming, but proficiency in C programming is beneficial. We are going to use the HAL-library (Hardware Abstraction Layer) from STM32, instead of 'bare-metal' programming.

You will learn how to setup your project with STM32CubeMX, configuring the pins and clock to your demands. Further, we will show you how to use STM32CubeIDE to program different interface protocols, timers, ADC, GPIOs and much more.

The guides structure follows the Udemy course: "Mastering STM32CubeMX 5 and CubeIDE - Embedded Systems"[1] and book: " Programming with STM32 - Getting Started with the Nucleo Board and C/C++." [2].

We are going to use the STM32F411-DISCO board for all exercises. It is also possible to use other STM32 development boards, however, the pin configuration might be different. To follow along with the presented projects, we created a github repository. Try to solve the problems by yourself, but if you get stuck, you can get inspired by the code.
Link to repository: https://github.com/CM134/STM32_HowTo

1.1 Microcontroller

Microcontrollers are a compact integrated circuits in an embedded system. A typical microcontroller includes a processor, memory and input/output (I/O) peripherals which are all integrated on a single chip.

Sometimes referred to as an embedded controller or microcontroller unit (MCU), microcontrollers are found everywhere. In vehicles, robots, IoT devices, medical devices, mobile radio transceivers, vending machines and home appliances, among other devices. They are designed to control small features of a larger component, without a complex graphical operating system (OS).

A mircocontroller is usually embedded on a printed circuit board with different components. To interface with these components the MCU needs to interpret data it receives from its I/O peripherals using its central processor unit (CPU). The temporary information that the microcontroller receives is stored in its data memory, where the processor accesses it and uses instructions stored in its program memory to decipher and apply the incoming data. It then uses its I/O peripherals to communicate and enact the appropriate action.

In many cases, several microcontrollers are integrated and work together to control a larger, complex system. A car for example has more than 50 MCUs [3] to control various individual systems within, such as the anti-lock braking system, traction control, fuel injection or suspension control. All the microcontrollers communicate with each other to inform the correct actions.

A microcontroller consist of a few essential elements. The core elements of a microcontroller are:

[1] https://www.udemy.com/course/stm32cubemx-5-and-cubeide/

- The processor (CPU): A processor can be thought of as the brain of the device. It processes and responds to various instructions that direct the microcontroller's function. This involves performing basic arithmetic, logic and I/O operations. It also performs data transfer operations, which communicate commands to other components in the larger embedded system.
- Memory: A microcontroller's memory is used to store the data that the processor receives and uses to respond to instructions that it's been programmed to carry out. A microcontroller has two main memory types:
  - Program memory, which stores long-term information about the instructions that the CPU carries out. Program memory is non-volatile memory, meaning it holds information over time without needing a power source.
  - Data memory, which is required for temporary data storage while the instructions are being executed. Data memory is volatile, meaning the data it holds is temporary and is only maintained if the device is connected to a power source.
- I/O peripherals: The input and output devices are the interface for the processor to the outside world. The input ports receive information and send it to the processor in the form of binary data. The processor receives that data and sends the necessary instructions to output devices that execute tasks external to the microcontroller.

While the processor, memory and I/O peripherals are the defining elements of the microprocessor, there are other elements that are frequently included. The term I/O peripherals itself simply refers to supporting components that interface with the memory and processor. There are many supporting components that can be classified as peripherals. Having some manifestation of an I/O peripheral is elemental to a microprocessor, because they are the mechanism through which the processor is applied.

Other supporting elements of a microcontroller include:

- Analog to Digital Converter (ADC): An ADC is a circuit that converts analog signals to digital signals. It allows the processor at the center of the microcontroller to interface with external analog devices, such as sensors.
- Digital to Analog Converter (DAC): A DAC performs the inverse function of an ADC and allows the processor at the center of the microcontroller to communicate its outgoing signals to external analog components.
- System bus: The system bus is the connective wire that links all components of the microcontroller together.
- Serial port: The serial port is one example of an I/O port that allows the microcontroller to connect to external components. It has a similar function to a USB or a parallel port but differs in the way it exchanges bits.

1.2 ARM

The microcontroller used in this guide is a based on the ARM architecture.

The ARM microcontroller stands for Advance RISC Machine; it is one of the extensive and most licensed processor cores in the world. The first ARM processor was developed in the year 1978 by Cambridge University, and the first ARM RISC processor was produced by the Acorn Group of Computers in the year 1985. These processors are specifically used in portable devices like digital cameras, mobile phones, home networking modules and wireless communication technologies and other embedded systems due to the benefits, such as low power consumption, reasonable performance, etc. This article gives an overview of ARM architecture with each module's principle of

working.

The ARM architecture processor is an advanced reduced instruction set computing (RISC) machine with a 32-bit (RISC) microcontroller. It was introduced by the Acron computer organization in 1987. This ARM is a family of microcontroller developed by makers like ST Microelectronics, Motorola, and so on.

The ARM cortex is a complicated microcontroller within the ARM family that has ARMv7 design.

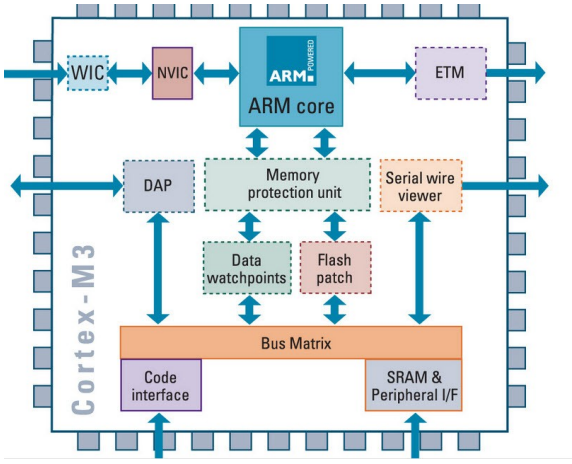


Figure 1: The ARM architecture

The ARM architecture consist of some essential parts explained below:

- Arithmetic Logic Unit (ALU): The ALU has two 32-bits inputs. The primary comes from the register file, whereas the other comes from the shifter. Status registers flags modified by the ALU outputs. The V-bit output goes to the V flag as well as the Count goes to the C flag. Whereas the foremost significant bit really represents the S flag, the ALU output operation is done by NORed to get the Z flag. The ALU has a 4-bit function bus that permits up to 16 opcode to be implemented.

- Booth Multiplier Factor: The multiplier factor has three 32-bit inputs and the inputs return from the register file. The multiplier output is barely 32-Least Significant Bits of the merchandise. The entity representation of the multiplier factor is shown in the above block diagram. The multiplication starts whenever the beginning 04 input goes active. Fin of the output goes high when finishing.

- Booth algorithm: Booth algorithm is a noteworthy multiplication algorithmic rule for 2's complement numbers. This treats positive and negative numbers uniformly. Moreover, the runs of 0's or 1's within the multiplier factor are skipped over without any addition or subtraction being performed, thereby creating possible quicker multiplication. The figure shows the simulation results for the multiplier test bench. It's clear that the multiplication finishes only in16 clock cycle.

- Control Unit: For any microprocessor, control unit is the heart of the whole process and it is responsible for the system operation,so the control unit design is the most important part within the whole design. The control unit is sometimes a pure combinational circuit design. Here, the control unit is implemented by easy state machine. The processor timing is additionally included within the control unit. Signals from the control unit are connected to each component within the processor to supervise its operation.

Figure 2: STM32 microcontroller chip

1.3 STM32

STM32 is a family of 32-bit microcontroller integrated circuits by STMicroelectronics. The STM32 chips are grouped into related series that are based around the same 32-bit ARM processor core. The STM32F411-DISCO uses an Arm Cortex-M4 with FPU core. Internally, each microcontroller consists of the processor core, static RAM, flash memory, debugging interface, and various peripherals.

1.3.1 STM32 F4

The STM32 F4-series is the first group of STM32 microcontrollers based on the ARM Cortex-M4F core. The F4-series is also the first STM32 series to have DSP and floating-point instructions. The F4 is pin-to-pin compatible with the STM32 F2-series and adds higher clock speed, 64 KB CCM static RAM, full-duplex I^2S, improved real-time clock, and faster ADCs. The summary for this series is:

- Core: ARM Cortex-M4F core at a maximum clock rate of 84 / 100 / 168 / 180 MHz.
- Memory: Static RAM consists of up to 192 KB general-purpose, 64 KB core-coupled memory (CCM), 80 bytes battery-backed with tamper-detection erase. Flash consists of 512 / 1024 / 2048 KB general-purpose, 30 KB system boot, 512 bytes one-time programmable (OTP), 16 option bytes. Each chip has a factory-programmed 96-bit unique device identifier number.
- Peripherals: Common peripherals included in all IC packages are USB 2.0 OTG HS and FS, CAN 2.0B, SPI or full-duplex I^2S, USART, 16-bit timers and 32-bit timers, watchdog timers, temperature sensor, ADCs, DACs, GPIOs, DMA, improved real-time clock (RTC), cyclic redundancy check (CRC) engine, random number generator (RNG) engine.

1.3.2 Discovery board

The following Discovery evaluation boards are sold by STMicroelectronics to provide a quick and easy way for engineers to evaluate their microcontroller chips. These kits are available from various distributors for less than 20€ .

Each board includes an on-board ST-LINK for programming and debugging via a Mini-B USB connector. The power for each board is provided by a choice of the 5 V via the USB cable, or an external 5 V power supply. They can be used as output power supplies of 3 V or 5 V (current must be less than 100 mA). All Discovery boards also include a voltage regulator, reset button, user button, multiple LEDs, SWD header on top of each board, and rows of header pins on the bottom.

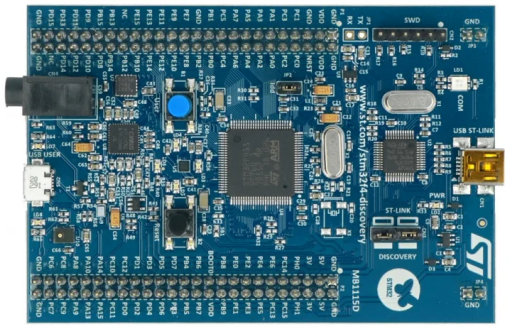


Figure 3: STM32F411e-DISCO board

An open-source project was created to allow Linux to communicate with the ST-LINK debugger.

2 Installation

Before we can start working with the STM32 board we need to install the development tools. For this how-to guide we are using the software development tools from ST[2].
All the software tools are available for Windows, Linux and Mac. This how-to-guide is using Windows, but any other OS is suitable.

2.1 STM32CubeMX

The STM32CubeMX is a graphical tool used to configure the STM32 microcontrollers and microprocessors. It makes it possible to initialize a project suiting to the corresponding development board, and creates corresponding initialization C code, which is the foundation for developing applications.

The software can be downloaded from the ST webpage[3]. 'Get the Software' and follow the installation instructions.

2.2 STM32CubeIDE

The generated C code can be opened in STM32CubeIDE. It is an advanced C/C++ development environment with peripheral configuration, code generation, code compilation, and debug features for STM32 microcontrollers and microprocessors. It uses GCC toolchain for development and GDB for the debugging. It is not needed to download any additional C/C++ compiler.

As in the previous step, you can find the software on the ST website[4] and install it on your machine.

2.3 Additional Resources

- Installation guide Windows: https://www.digikey.com/en/maker/projects/\protect\@normalcr\ relaxgetting-started-with-stm32-introduction-to-stm32cubeide/6a6c60a670c447abb90fd0fd78008697

[2]ST32 Software Development Tools: https://www.st.com/en/development-tools/stm32-software-development-tools.html
[3]STM32CubeMX: https://www.st.com/en/development-tools/stm32cubemx.html
[4]STM32CubeIDE: https://www.st.com/en/development-tools/stm32cubeide.html#overview&secondary=st-clickmap-diagram

3 Getting Started

3.1 Background

This section will teach you how to start a new project, enable pins, rename pins and save your project.

3.2 Prerequisites

- STM32CubeMX installed

3.3 Steps

After installing, we need install some more packages. Head to "Help" in the top bar and find "Manage embedded software packages". See figure 4.

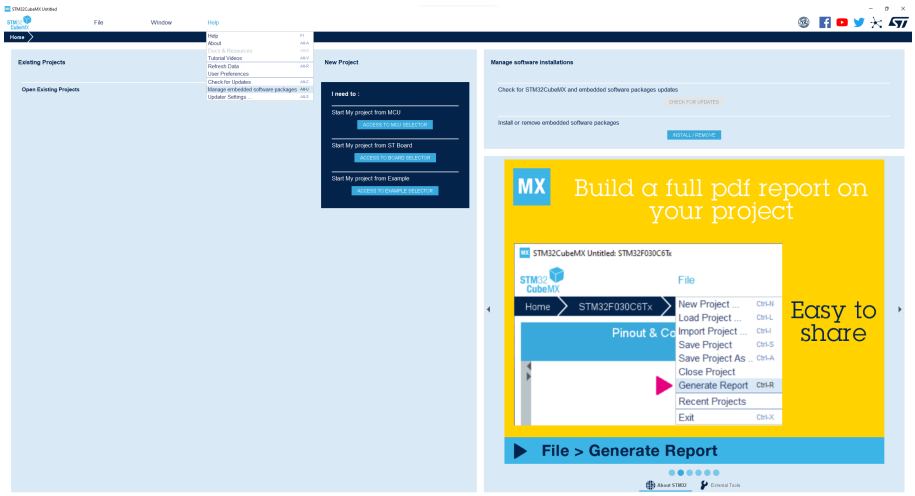


Figure 4: Manage embedded software packages

A pop-up window should appear. In this find "STM32Cube MCU Package for STM32F4 Series". Install the latest version.

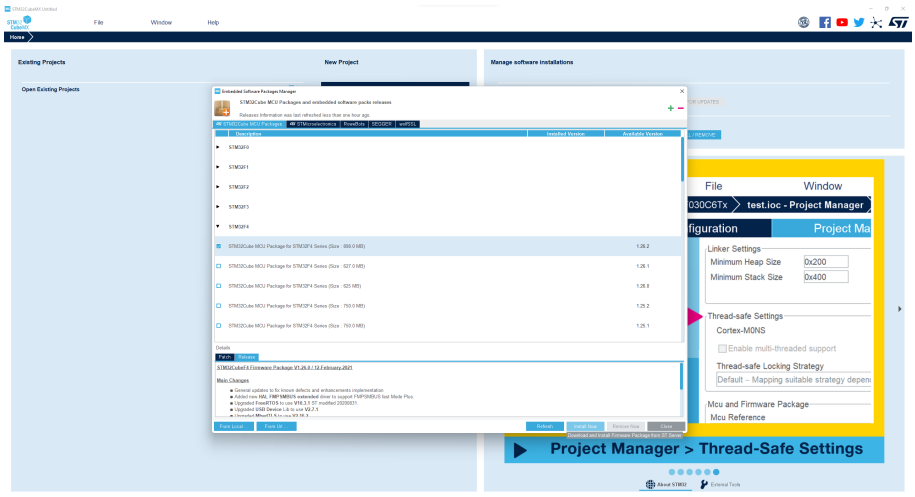


Figure 5: Find STM32F4 in the list

STM32CubeMX is now ready for creating projects with STM32FxxE-DISCO boards.

3.3.1 Creating a project

To create a project using the STM32F411E-DISCO board, open STM32CubeMX. Inside the darkblue box, you have the option to create a new project based the MCU, the board or using an example project. As we are using a specific board, we will start a new project from ST board. See figure 6.

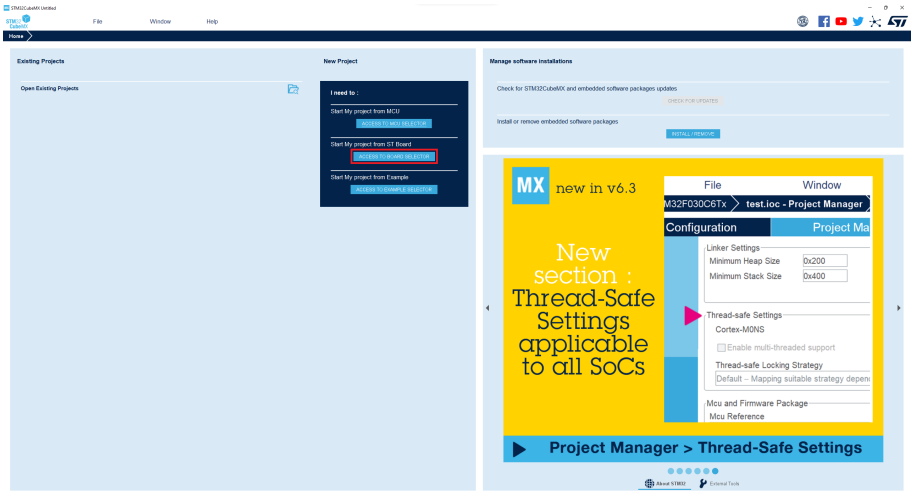


Figure 6: Create your first project in ST32CubeMX

In the board selector, you can scroll through the different boards available, or you can use the search box in the top left of the pop-up window.

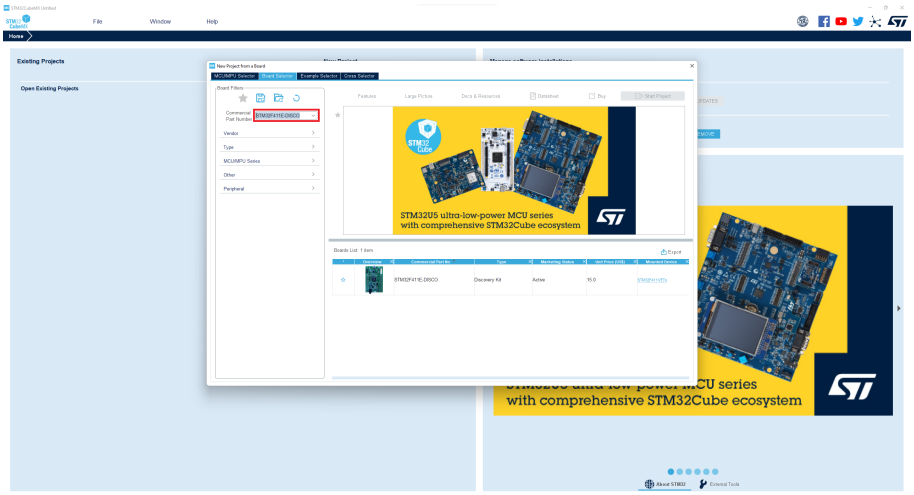


Figure 7: ST Board Selector

When the board is found, click it. You now have the opportunity to download the data sheet, pictures etc. from the top. Go ahead and pick up the data sheet, it will useful later. Start the project using the blue button.

When pressing "Start Project", you will get a prompt asking if you would like to initialize with the default values. This is not a big deal whether to do this or not, as when doing a project you will change most values anyway. For the purpose of learning, we will use the default values in the rest of this section.

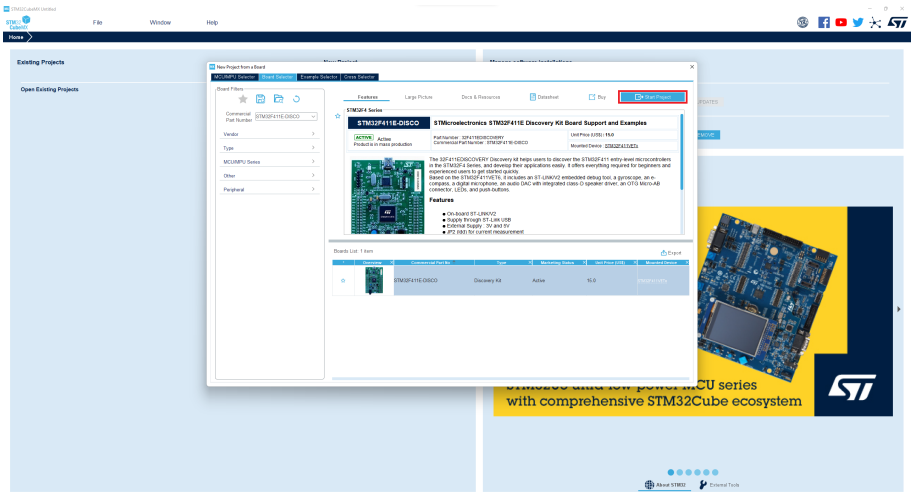


Figure 8: Start STM32CubeMX Project

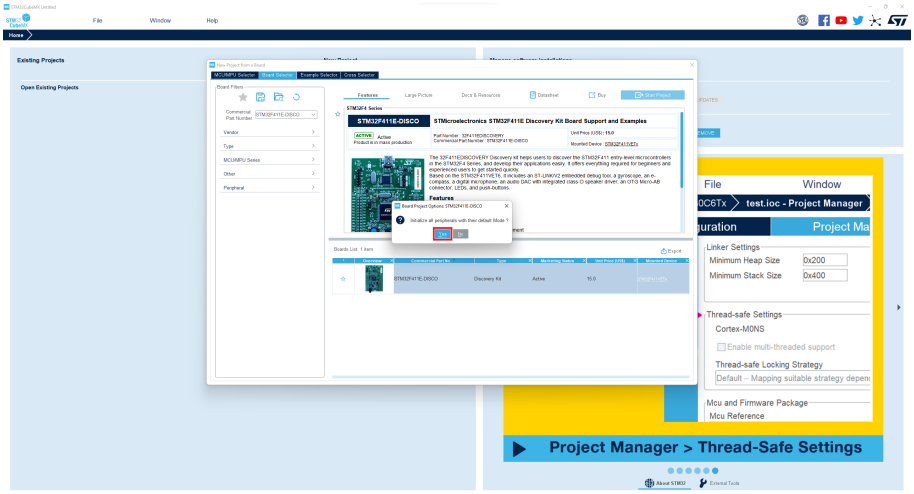


Figure 9: Prompt asking to initialize peripherals with default values or not.

If you decided to initialize with default values, you should now see the pinout & configurator as shown in figure 11.

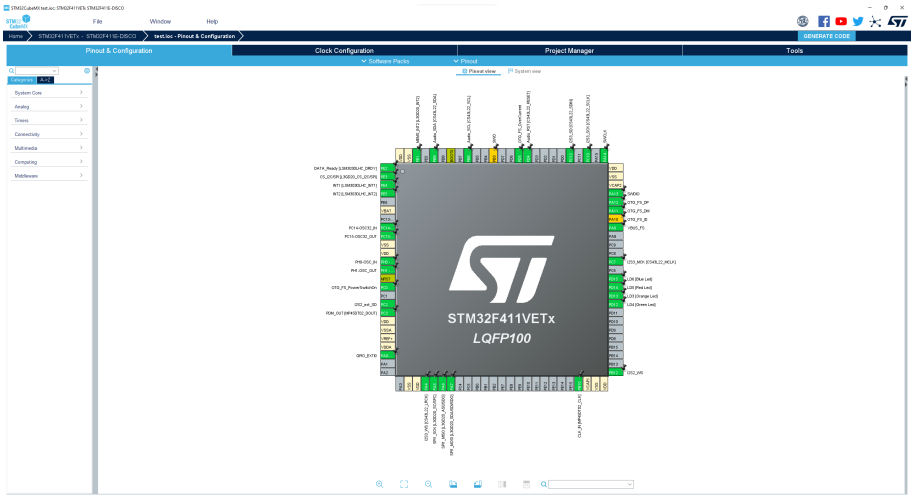


Figure 10: Pinout & Configurator tab with default values

In the pinout & configurator tab, you see the pinouts, their status (green means enabled, yellow means disabled, grey means reset). On the left panel you have the different configurations, which will be covered in later sections.

As an example, lets say we wanted to use pin PA8. We therefore have to enable PA8.

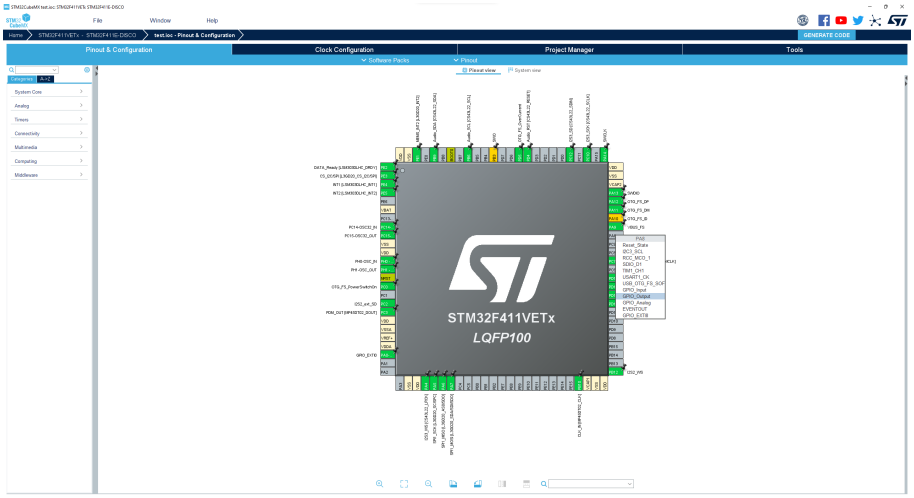


Figure 11: Left-click PA8

To enable PA8, you left-click the PA8 pin, and choose which type you want the pin to be. We choose it as an output for now. Choose "GPIO_Output". To change the name of pin, right-click and choose "Enter User label", then write what you would like to call the pin.

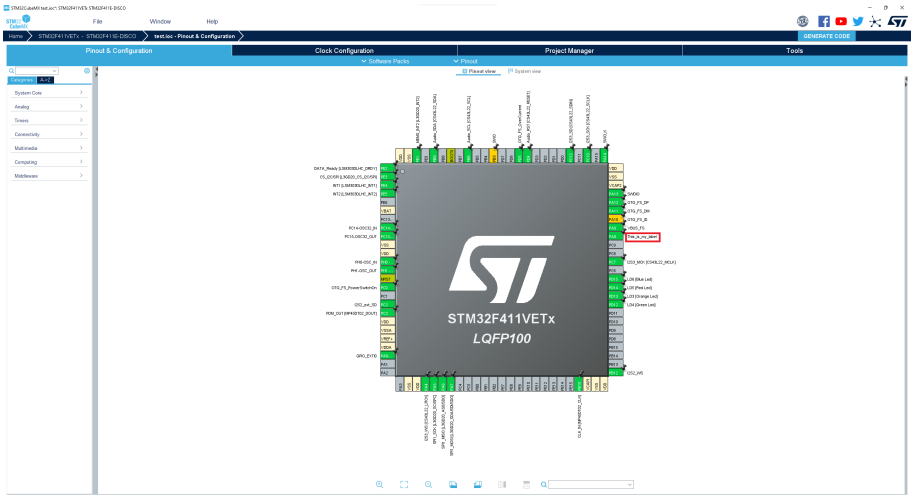


Figure 12: Pin PA8 changed to output with a new name.

To save your project, go to "Project Manager" and choose where to save the project. **It is important that you pick STM32CubeIDE in the "Toolchain/IDE" box**

The final step is to press "Generate Code". This will prompt you if want to "Open Project" or "Open Folder". Pressing "Open Folder" will take you to STM32CubeIDE.

3.4 Exercise

1. Configure a project using from an empty pinout. Clear the default pinout in the 'Pinout & Configurator tab' (fig. 11), in the 'Pinout' drop-down menu select 'Clear Pinout'.
   Configure the blue button as an input and the LEDs as an output.

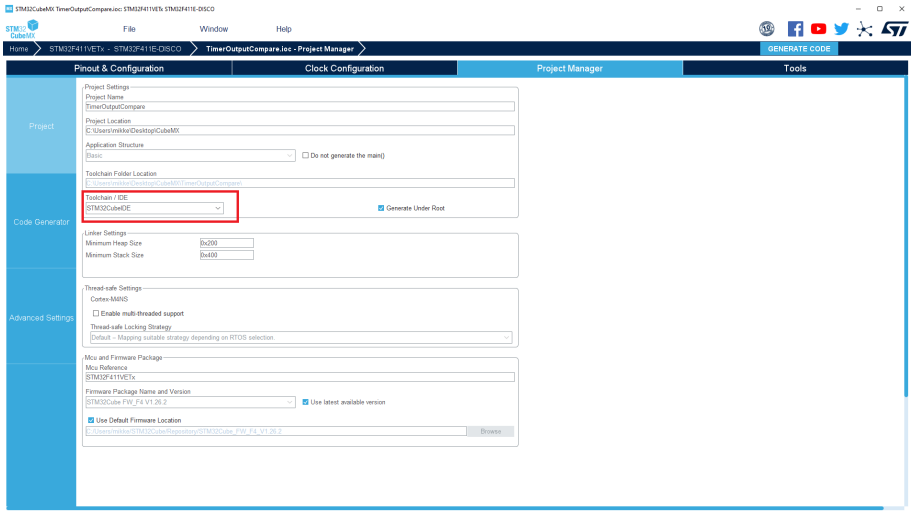


Figure 13: Project Manager tab

Get comfortable with reading the documentation. There you will find the pinout specifications for the DISCO board.

3.5 Summary

We created our first project, where we enabled a pin and used the project manager tab to save our project and generate code.

3.6 Additional Resources

- CubeMX User-Manual: https://www.st.com/resource/en/user_manual/dm00104712-stm32cubemx-for-stm32\-configuration-and-initialization-c-code-generation-stmicroelectronics.pdf
- Beginner's Guide: https://www.microcontrollertutorials.com/2020/07/beginners-guide-for-stm32cubeide-and.html

4 GPIO & HAL

4.1 Background

Though out this guide we will continousely use the HAL library to work with the DISCO-board.

The STM32 Hardware Abstraction Layer (HAL) an STM32 abstraction layer embedded software ensuring maximized portability across the STM32 microcontroller. The HAL is available for all the hardware peripherals. It originates from STMCube to ease and accelerate the development cycle of embedded products.

The low-layer APIs (LL) offering a fast light-weight expert-oriented layer that is closer to the hardware than the HAL. The LL APIs are available only for a set of peripherals.

The HAL and LL are complementary and cover a wide range of applications requirements:

- The HAL offers high-level and feature-oriented APIs, with a high-portability level. They hide the MCU and peripheral complexity to end-user.
- The LL offers low-level APIs at registers level, with better optimization but less portability. They require deep knowledge of the MCU and peripherals specifications.

The HAL driver layer provides a simple, generic multi-instance set of APIs (application programming interfaces) to interact with the upper layer (application, libraries and stacks). The HAL driver APIs are split into two categories: generic APIs, which provide common and generic functions for all the STM32 series and extension APIs, which include specific and customized functions for a given line or part number. The HAL drivers include a complete set of ready-to-use APIs that simplify the user application implementation. For example, the communication peripherals contain APIs to initialize and configure the peripheral, manage data transfers in polling mode, handle interrupts or DMA, and manage communication errors. The HAL drivers are feature-oriented instead of IP-oriented. For example, the timer APIs are split into several categories following the IP functions, such as basic timer, capture and pulse width modulation (PWM). The HAL driver layer implements run-time failure detection by checking the input values of all functions. Such dynamic checking enhances the firmware robustness. Run-time detection is also suitable for user application development and debugging.

This section introduce you how to use General Purpose Input Output (GPIO) and the STM Hardware Abstraction Layer (HAL) on the STM32F411E-DISCO board. Figure 14 shows the a screenshot from the HAL library reference manual. During this course you will need to get familiar with finding functions and documentation in this manual. (fig. 14)

HAL_GPIO_Init

Function name	**void HAL_GPIO_Init (GPIO_TypeDef * GPIOx, GPIO_InitTypeDef * GPIO_Init)**
Function description	Initialize the GPIOx peripheral according to the specified parameters in the GPIO_Init.
Parameters	• **GPIOx:** where x can be (A..H) to select the GPIO peripheral for STM32L4 family • **GPIO_Init:** pointer to a GPIO_InitTypeDef structure that contains the configuration information for the specified GPIO peripheral.
Return values	• **None:**

Figure 14: HAL reference manual description of the `HAL_GPIO_Init` function

Each of the general-purpose I/O (GPIO) ports has two 32-bit configuration registers, two 32-bit data registers, a 32-bit set/reset register, a 16-bit reset register, and a 32-bit locking register. Each

I/O port bit is freely programmable, however, the I/O port registers have to be accessed as 32-bit words (half-word or byte accesses are not allowed). The purpose of the set/reset registers is to allow atomic read/modify accesses to any of the GPIO registers. This way, there is no risk that an IRQ occurs between the read and modify access.

Here is a digital diagram for the internal structure of a typical GPIO pin. It shows the diode protection, internal pull-up or down enable/disable, and also the push-pull output driver, output enable/disable for switching between input/output pin modes, Schmitt-triggered digital input, analog input. (fig. 15

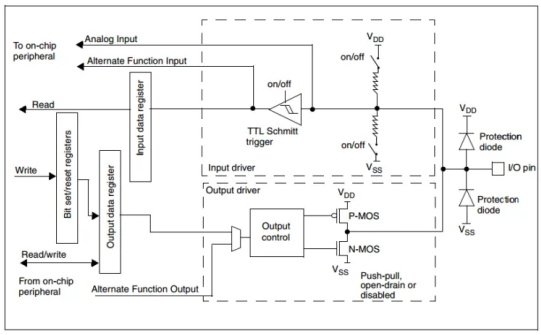


Figure 15: Basic structure of a five-volt tolerant I/O port bit (from Reference Manual)

You have to assume that all GPIO pins are not 5V tolerant by default until you find in the datasheet that a specific pin is 5V tolerant, only then you can use it as a 5V pin. The pins are mostly 3.3V and can be damaged when hooked up to 5v digital input lines. Level shifting may be mandatory in many cases.

So you have to be careful with the voltage level for input pins. And also you have to pay attention to the output current when you set GPIO output pins. The maximum current that could be sourced or sunk into any GPIO pin is 25mA as per the datasheet. And you have to check it for the specific target microcontroller you are dealing with.

4.2 Prerequisites

- STM32CubeMX installed
- STM32CubeIDE installed
- STM32F411E-DISCO Board
- STM32F HAL Manual
- STM32F411E-DISCO User Manual

4.3 STM32CubeMX

First, start a new project, find the STMF411E-DISCO board and open the Pinout & Configurator tab as shown in section 3. This section Initializes peripherals with default settings, doing without will not make any difference.

Second, go ahead and enable PD12, PD13, PD14 and PD15 and rename after their color. The User manual will tell you which pin is the LED's and their colors.

Open the drop-down menu "System core" in the left side of the window. Open the "RCC" menu and choose "BYPASS Clock Source" in the "High Speed Clock (HSE)" tab and choose "Crystal/-Ceramic Resonator" in the "Low Speed Clock (LSE)". Pin PC14-OSC32_IN, PC15-OSC32_OUT, PH0-OSC_IN and PH1-OSC_OUT should now be enabled.

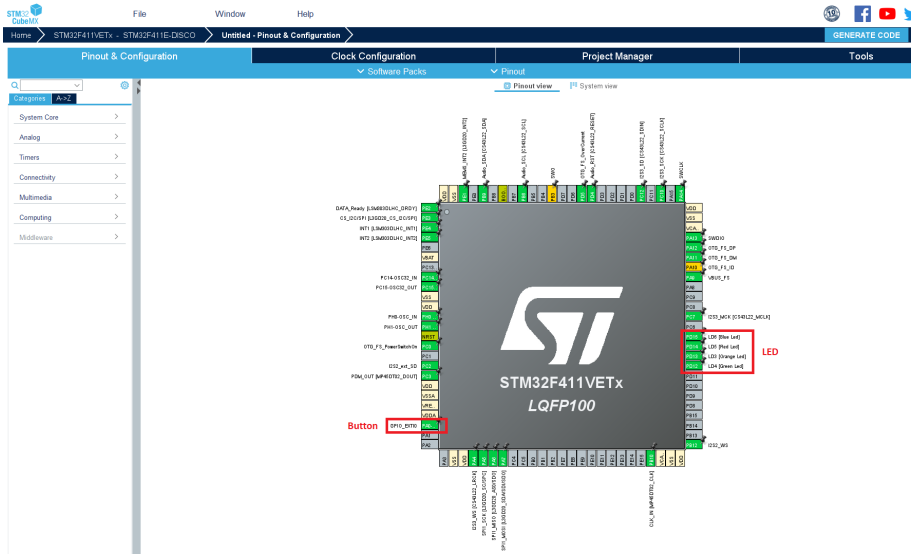


Figure 16: Configure the LED Pins and the button

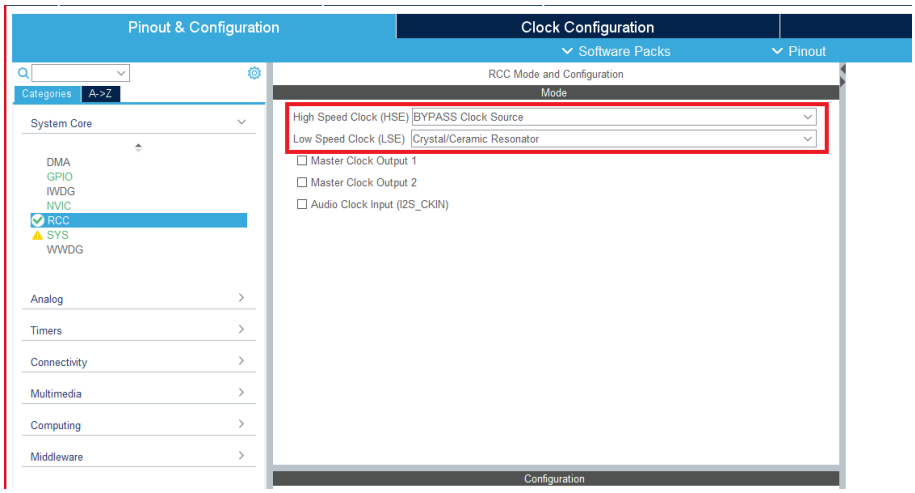


Figure 17: In System Core configuring RCC Clocks

Head to the "SYS" tab, and choose "Serial Wire" in the debug drop-down menu.

We are now ready to save the project and generate the code for STM32CubeIDE.

4.4 STM32CubeIDE

After the code generated, we can directly open the STM32CubeIDE with the newly generated project. On the lefthand-side all the project within the workspace folder are shown in the "Project Explorer". Unfold the just created project and take a look a the folder structure. You can intestate the different, automatically created files. All the HAL-drivers are in the "Drivers/STM32F4xx_HAL_Driver" folder. If you want, you can find all the HAL functions, implemented in these files. It is however easier to search in the HAL manual then than in the source files.

4.5 Code

In the source directory we can find the main.c file. Here we will implement the majority of our code.

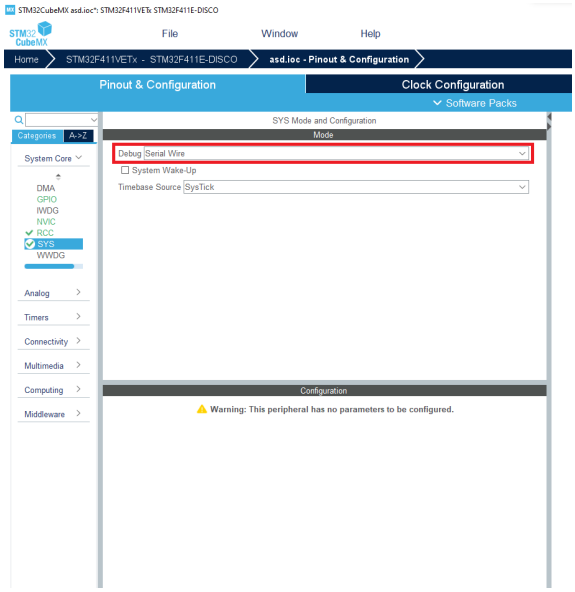


Figure 18: Serial Wire Debug

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

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/

/* USER CODE BEGIN PV */
int BTN_state = 1;
int cnt = 0;

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
/* USER CODE BEGIN PFP */
void SystemClock_Config(void);

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
```

```
46   */
47
48
49 int main(void)
50 {
51   /* USER CODE BEGIN 1 */
52
53   /* USER CODE END 1 */
54
55   /* MCU Configuration--------------------------------------------------------*/
56
57   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
58   HAL_Init();
59
60   /* USER CODE BEGIN Init */
61
62   /* USER CODE END Init */
63
64   /* Configure the system clock */
65   SystemClock_Config();
66
67   /* USER CODE BEGIN SysInit */
68
69   /* USER CODE END SysInit */
70
71   /* USER CODE BEGIN 2 */
72   /* Initialize all configured peripherals */
73   MX_GPIO_Init();
74   /* USER CODE END 2 */
75
76   /* Infinite loop */
77   /* USER CODE BEGIN WHILE */
78
79   while (1)
80   {
81     /* USER CODE END WHILE *
82
83     /* USER CODE BEGIN 3 */
84     BTN_state = HAL_GPIO_ReadPin(BTN_GPIO_Port, BTN_Pin);
85     HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, BTN_state);
86   HAL_Delay(100);
87
88   }
89   /* USER CODE END 3 */
90 }
```

Listing 1: main.c full code until end of infinite while-loop

4.6 Examine the code

In this section we are going to take a closer look at the code.

4.6.1 Comments

Taking a look at the generated file, it is possible to see that many comments were automatically created. This might seem irritating in the beginning but can be rather helpful. The code tells us, where we should put our code and the IDE knows where it can put automatically generated code.

Lets assume that we set up our project in STM32CubeMX and start coding in STM32CubeIDE. During the coding process we notice that we require a additional hardware configuration. In order to not have to start from scratch, we can open CubeMX inside of CubeIDE by pressing the ¡project_name¿.ioc file. Now we can make our hardware configuration adjustments in the known environment and generate the new code. Everything which was coded inside of the specified comments, will stay unchanged.

4.6.2 Includes

The beginning of the file we can see the automatically created includes 'main.h' and 'gpio.h'. Let us take a closer look at them.

```
1 /* Includes ------------------------------------------------------------------*/
2 #include "main.h"
3 #include "gpio.h"
```

Listing 2: Includes

In the main.h are the labels defined that we assigned to the pins in CubeMX.

```
1 /* Private defines -----------------------------------------------------------*/
2 #define BTN_Pin GPIO_PIN_0
3 #define BTN_GPIO_Port GPIOA
4 #define LED_G_Pin GPIO_PIN_12
5 #define LED_G_GPIO_Port GPIOD
```

Listing 3: main.h

In the gpio.c file is including a struct that sets the GPIOs up as we defined them in CubeMX. It is possible to do fast changes of the configurations in this file, if we do not want to change the whole project in CubeMX.

```
1  void MX_GPIO_Init(void)
2  {
3
4    GPIO_InitTypeDef GPIO_InitStruct = {0};
5
6    /* GPIO Ports Clock Enable */
7    __HAL_RCC_GPIOC_CLK_ENABLE();
8    __HAL_RCC_GPIOH_CLK_ENABLE();
9    __HAL_RCC_GPIOA_CLK_ENABLE();
10   __HAL_RCC_GPIOD_CLK_ENABLE();
11
12   /*Configure GPIO pin Output Level */
13   HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, GPIO_PIN_RESET);
14
15   /*Configure GPIO pin : PtPin */
16   GPIO_InitStruct.Pin = BTN_Pin;
17   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
18   GPIO_InitStruct.Pull = GPIO_NOPULL;
19   HAL_GPIO_Init(BTN_GPIO_Port, &GPIO_InitStruct);
20
21   /*Configure GPIO pin : PtPin */
22   GPIO_InitStruct.Pin = LED_G_Pin;
23   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
24   GPIO_InitStruct.Pull = GPIO_NOPULL;
25   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
26   HAL_GPIO_Init(LED_G_GPIO_Port, &GPIO_InitStruct);
27
28 }
```

Listing 4: gpio.c

The GPIO is initialized in line 73 of main.c.

4.7 Turn on the LED when the button is pressed

In order to toggle the LED, we need to know the current state of the LED and then we want to Write the new state to the right port and pin.

Looking at the HAL manual we can find the `HAL_GPIO_ReadPin()` and `HAL_GPIO_WritePin()` functions.

We want to read the state of the button and write the state to the LED. The button is 1 when pressed.

```
while (1)
  {
    BTN_state = HAL_GPIO_ReadPin(BTN_GPIO_Port, BTN_Pin);
    HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, BTN_state);
    HAL_Delay(100);
  }
```

Listing 5: Turning on LED with button

We put it in the while loop in order to keep it continuously running.

4.8 Exercise

- Instead of turning on the LED when the button is pressed, try to toggle it, like a light switch you have at home
  Tipp: In order to debounce button set delay time up.
- Use `HAL_GPIO_TogglePin` to toggle the LED every 1 second.

4.9 Summary

We created our first project, where we enabled a pin and used the project manager tab to save our project and generate code.

4.10 Additional Resources

- Information about GPIOs: https://deepbluembedded.com/stm32-gpio-tutorial/
- Code example: https://github.com/CM134/STM32_HowTo/tree/4_GPIO_0

5 Interrupts

5.1 Background

Interrupts are a widely used tool in microcontroller programming. When a specified event happens it triggers an request that interrupts the current process and executes an interrupt service routine (ISR). The ISR handles the event and after execution the microcontroller resumes with previous interrupted activities.
Interrupts are used for hardware or software triggers that need immediate attention. In this tutorial external interrupts are introduced. For this the blue button triggers an interrupt which interacts with the LED.

5.2 Prerequisites

- Reference manual (RM0383)[5]
- STM32CubeMX
- STM32CubeIDE

On page 201 off the reference manual the nested vectored interrupt controller (NVIC) is explained. For this tutorial we focus on external interrupts. From figure 30 'external interrupt/event GPIO mapping' it is possible to see which pins share the same register. For the graph we can see that the blue button PA0 is using EXTI0.

5.3 Start Project in STM32CubeMX

Do the usual steps as the two times before. First, find the board we are using. Then delete the pins and start the configuration of the clocks. Click on RCC in 'System Core' menu and change 'HSC' to 'BYPASS' and 'LSC' to 'Crystal'. In the 'SYS' tab change 'Debug' to Serial Wire'.

5.3.1 Pinout Configuration

In the 'Pinout view', where you see the chip, select the pin for the blue button PA0-WKUP (bottom left corner) and configure it to be 'GPIO_EXTI0' (fig. 19). This sets the external event register EXTI0 for the button.

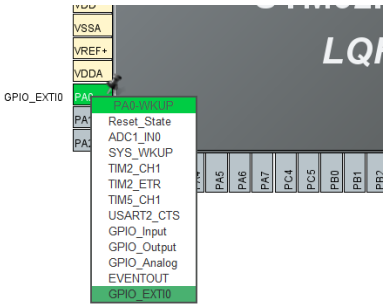


Figure 19: Configure button to be external interrupt

Next activate a LED to be a 'GPIO_Output' as in the last weeks exercise. You can change the labels of the button and the LED by right clicking the pin and 'enter a user label'. Here we chose 'BTN' and 'LED' for LED on port PD12.

[5]Reference manual and other documents can be found here: https://www.st.com/en/microcontrollers-microprocessors/stm32f411ve.html#documentation

5.4 Configure Interrupt

Change to 'NVIC'-view in the left and enable 'EXTI-line 0 interrupt' (fig. 20).

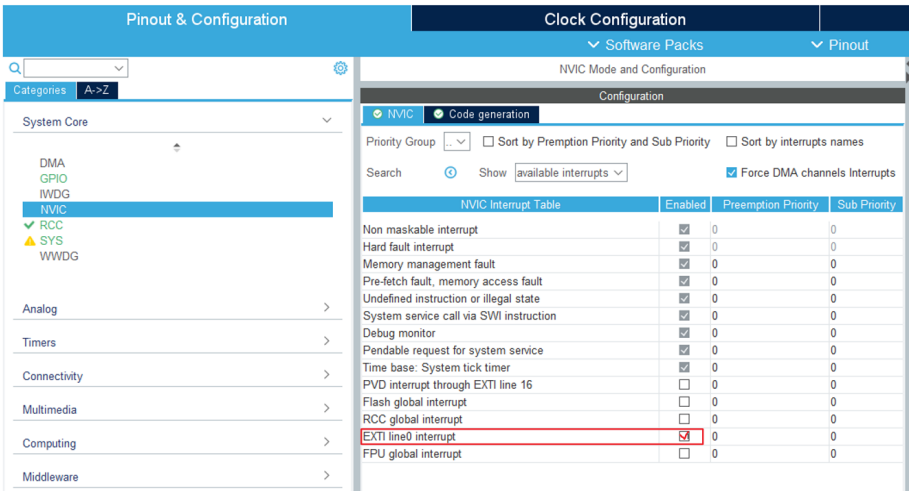


Figure 20: Enable EXTI0

In the NVIC-view change to code generation and enalbe the init sequence for the interrupt (fig. 21)

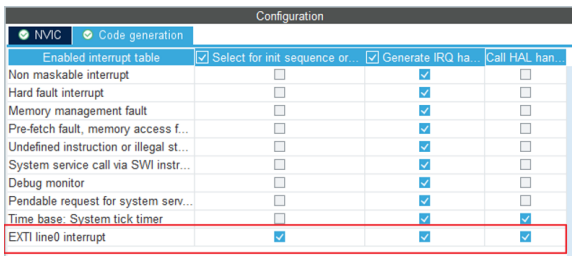


Figure 21: Enable code generation

In the 'GPIO'-view it is possible to change the trigger detection. Select the button pin in the top, then you can see the different GPIO modes in the bottom. For no we want to leave the mode on 'External Interrupt Mode with Rising edge trigger detection'. (fig. 22)

5.5 Finish the project and generate code

As in the last 2 sessions before, select fan 'Project Manager' and name the project, select the right toolchain and in the 'Code Generator' view under 'Generated files' check 'Generate peripheral initialization as a pair of '.c/.h' files per peripheral'. Then 'Generate Code' and open STM32CubeIDE.

5.6 Code Inspection

In the IDE lets see what CubeMX has generated. You can either find different functions related to interrupts in the HAL and reference manuals or by scanning the code.

If you do not want to look up all the implemented functions in the manual, try to find it in the generated header files. Since we are dealing with GPIOs find the HAL GPIO header and take a look at the file's outline. You can spot the interrupt functions and declarations here. For the function definitions you can either go in the header and find the declaration and CTRL+Right-Click to get to the declaration OR you can find it in the /Drivers/STM32F4xx_HAL_Driver/src/stm32f4xx_hal_gpio.c

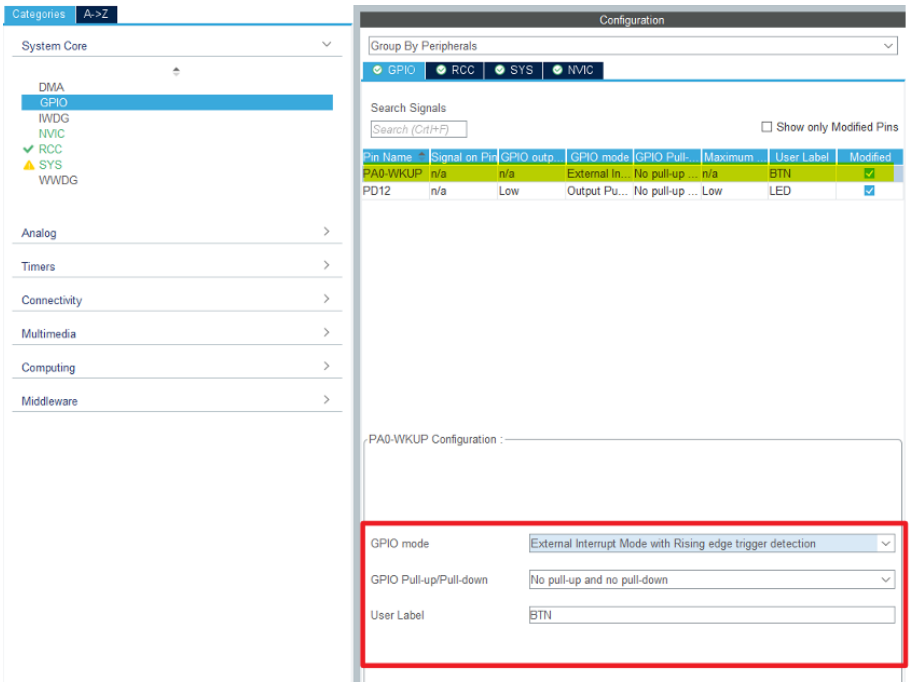


Figure 22: Configure GPIO interrupt mode

file. In the source file on line 507 you find: `__weak void HAL_GPIO_Callback(uint16_t GPIO_Pin)`. The weak-keyword is useful when overloading the function. If a second declaration of the function is defined, it the compiler chooses the one which is not weak.

```
1  /* Initialize all configured peripherals */
2  MX_GPIO_Init();
3
4  /* Initialize interrupts */
5  MX_NVIC_Init();
```

Listing 6: Init NVIC

In main.c is the initialization of the NVIC, which enables the interrupt requests (IRQ). In gpio.c we can see the GPIO settings we set in the CubeMX. Here we could manually change the mode of the interrupt.

In stm32f4xx_it.c is the `EXTI0_IRQHandler` function defined which is dealing with the event. Here we can put the instructions to execute during the ISR. The function call `HAL_GPIO_EXTI_IRQHandler(Pin)` resets the interrupt to prevent it from getting executed again.

5.7 Modify code in STM32CubeIDE

5.7.1 LED should blink in variable time

In main.c while declare a integer for the time variable time: `uint16_t t_delay_`. In the while-loop toggle the LED and set delay.

```
1  while (1)
2  {
3    /* USER CODE END WHILE */
4    HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
5    HAL_Delay(t_delay_);
6    /* USER CODE BEGIN 3 */
```

```
7   }
```

Listing 7: Infinite while-loop

5.7.2 Change delay with interrupt

In stm32f4xx_it.c import the external delay variable: `extern uint16_t t_delay_`. Then write in the IRQ-Handler a logic that changes the delay.

```
1  void EXTI0_IRQHandler(void)
2  {
3    /* USER CODE BEGIN EXTI0_IRQn 0 */
4
5    if (t_delay_ == 250){
6      t_delay_ = 5000;
7    }
8    else{
9      t_delay_ = 250;
10   }
11
12   /* USER CODE END EXTI0_IRQn 0 */
13   HAL_GPIO_EXTI_IRQHandler(BTN_Pin);
14   /* USER CODE BEGIN EXTI0_IRQn 1 */
15
16   /* USER CODE END EXTI0_IRQn 1 */
17 }
```

Listing 8: IQR-Handler logic

5.7.3 Compile and download code

Now you should be able to download the code to the disco-board and see the LED blinking. Press the button to change the frequency. Working as expected? Well done!

5.8 Exercise

1. Use `HAL_EXTI_Callback()` instead of IRQ-Handler.
   (a) Check which pin called the callback. (Needed when several interrupts are used. Callback works as central interrupt logic)
2. Configure multiple interrupts to happen.

5.9 Summary

We saw how to setup and code external interrupts.

5.10 Additional Resources

- https://www.geeksforgeeks.org/interrupts/
- https://www.codeinsideout.com/blog/stm32/interrupt/
- Code example: https://github.com/CM134/STM32_HowTo/tree/5_Interrupt, https://github.com/CM134/STM32_HowTo/tree/5_Interrupt_MultipleInterrupts

6 Timers

6.1 Background

Timers are one of the most often used features in microcontroller programming. They are used to measure execution times, create non-blocking code, control pin timing or even run operating systems. [4]

A timer is basically a free-running clock counter, which increments by 1 with each clock. The source clock can be reduced using a dedicated prescaler for each timer. A timer can have additional pre-load register, therefore, timer will count from 0 to the pre-load value, and then go back to count again from 0.
There are different types of timer types like basic timers, low-power timers, advanced timers or high-resolution timers. However, this section is focusing on using general-purpose timers. General purpose timers are 16/32-bit timers with multiple purposes. They come with four-programmable input/output channels.
The output compare is used in different applications like timing and delay generation, One-Pulse Mode, input capture (for external signal frequency measurement), sensor interface (encoder, hall sensor), PWM generation.
Here the timers are used to simply blink an LED. It is shown how to configure them using the STM32CubeMX and CubeIDE pipeline.

6.2 Prerequisites

- Datasheet
  - on p.27 see section '3.20 Timers and watchdogs'
  - on p.15 see figure 3 ' STM32F411xC/xE block diagram' to see what is connected to the timers.
- Reference Manual
  - Section 13: 'General-purpose Timers'

6.3 Start Project in STM32CubeMX

In this tutorial, the LED blinking is triggered by using timers.

Like the other times, start the CubeMX project as in the weeks before. Use the system defaults for the clocks.

6.4 Configure LED

As in the last section, enable one of the LEDs as an GPIO output. Do that by selecting one of the pins PD12-PD15 as 'GPIO_Output' in the chip configurator.

6.5 Select timer

We would like to work with general purpose timers (TIMx). To see which timer is connected to which bus, open the datasheet and look a figure 3 or read about the different timers in section 3.20. A deeper descriptions of the timers can be found in the reference manual. The general-purpose timers for the STM32F411 are TIM2, TIM3, TIM4 and TIM5. For now we are using the 16-bit

auto-reload up/downcounter and 16-bit prescaler of TIM3 (or TIM4).
From figure 3 of the datasheet we can see that TIM3 is connected to APB1. The clock configuration shows that the 'APB1 Timer clocks (MHz)' is 48.

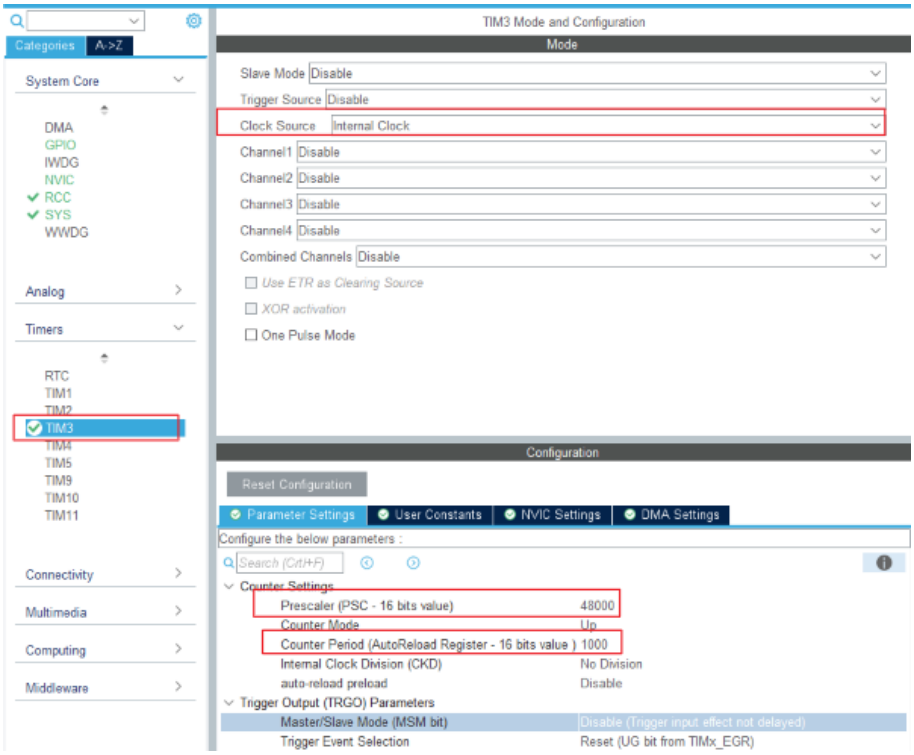


Figure 23: Configure TIM3

We would like to get a timer frequency of 1 Hz. Therefore we set the Prescaler (PSC) to 48000 and the Counter Period to 1000 (fig. 23. That means we downscale the 48 MHz by 48000 to 1 kHz. Next we set the counter up to 1000. With every 1 kHz tick the counter increments and when it reaches 1000, the timer rolls over and executes an interrupt.
To enable the interrupt change to the 'NVIC'-tab and enable the global interrupt (fig. 24).

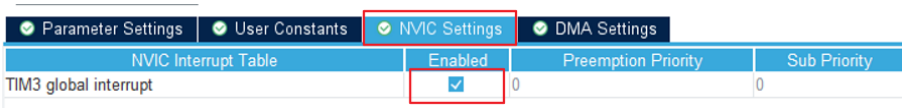


Figure 24: Enable TIM3 Interrupt

6.6 Generate Code

Proceed with the code generation as in the last sections. Select the CubeIDE toolchain and then open STM32CubeIDE.

6.7 Code Inspection

In the generated main.c, is apart from the usual init functions, there is the timer init, with the in CubeMX specified values.
To start programming it is always recommended to take a first look into the HAL manual or by checking the generated source code in the Drivers/Src/stm32f4xx_hal_tim.c file.

6.8 Modify code in STM32CubeIDE

The first step is to start the interrupt timer. Use the `HAL_Base_Start_IT` function to do that. In the top of the main.c, it is possible to find the name of the name of the timer. In our case it is `htim3`.
Now the timer is started, and will run and call raise an interrupt after every second. That means every second a callback function is executed. To find the structure of this callback we can either take the HAL manual or find it in the generated source code. Find the function 'HAL_TIM_PeriodElapsedCallback' and put it in the main function. Best practice is to put the function definition in one of the places marked with 'USER CODE'. E.g. after the superloop in USER CODE 4.
`HAL_TIM_Base_Start_IT(&htim3)`, which starts the timer in interrupt mode. The while superloop is blank! Our blinky code will be entirely handled by interrupts.
In the callback, check that the interrupt was triggered by the right interrupt and then toggle the LED, as shown below.

```
/* USER CODE BEGIN 4 */

// Callback: timer has rolled over
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  // Check which version of the timer triggered this callback and toggle LED
  if (htim == &htim16 )
  {
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
  }
}

/* USER CODE END 4 */
```

Listing 9: Timer interrupt callback

6.9 Exercise

- Output compare functionality
  - Use 1 timer to control several LEDs by using the 'Output Compare Channels' of a general purpose timer.
  - Toggle each LED successively. Meaning the first toggles after 250ms, the next after 500ms and so on.
  - Use timer as a input event counter.

6.10 Summary

We showed how to use timers and introduced different ways of timer configurations.

6.11 Additional Resources

- Nucleo tutorial with video: https://www.digikey.dk/en/maker/projects/getting-started-with-stm32-timers-and-time d08e6493cefa486fb1e79c43c0b08cc6
- STM32 timer tutorial: https://deepbluembedded.com/stm32-timers-tutorial-hardware-timers-explained/
- Code example:
  https://github.com/CM134/STM32_HowTo/tree/6_Timers_compare,
  https://github.com/CM134/STM32_HowTo/tree/6_Timers_count,
  https://github.com/CM134/STM32_HowTo/tree/6_Timers_1sec

7 Pulse-Width Modulation (PWM)

7.1 Background

In microcontroller programming pulse-width modulation (PWM) is used for controlling analog circuits with a digital output. It generates pulses of variable width to represent the amplitude of an analog signal (see fig. 25 [6]). PWM is used in many applications like electric motor control or dimming an lightbulb.
A PWM signal is a square wave with a period P, an on time T_{on} where the output is high and an off time T_{off} where the output is low. The duty cycle D, is the ration of T_{on} and T_{off}, i.e. $D = \frac{T_{on}}{T_{off}}$. The average output voltage is dependent on the duty cycle and the pin output voltage V_{on}, i.e. $V_{avg} = D * V_{on}$.

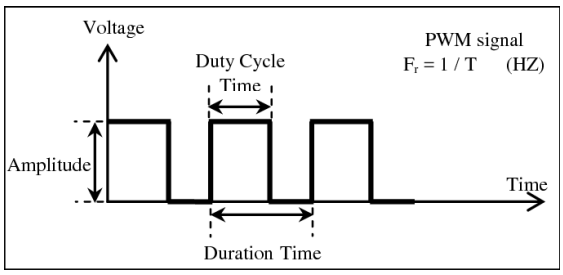


Figure 25: PWM signal

7.1.1 Configure Board

The goal of this project is to control a LED using PWM.

In CubeMX set the usual settings for the "System Core".

Configure the LED to be a timer. In this example we use the red led PD14.

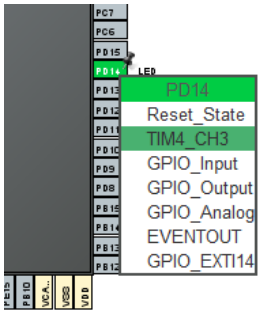


Figure 26: LED configuration

Afterward, go to "Timers" and select TIM4 in the left. Select the Clock Source to come from the Internal Clock and Channel 3 to PWM Generation. (fig. 27)

Leave the clock configuration at the default. The HCLK should 16 MHz as well as the APB2 peripheral clock.

To set 1 sec as the counter period, configure the Prescaler to be 1000 and counter period is 16000.

[6] https://www.researchgate.net/figure/PWM-signal-with-its-two-basic-time-periods_fig4_271437313

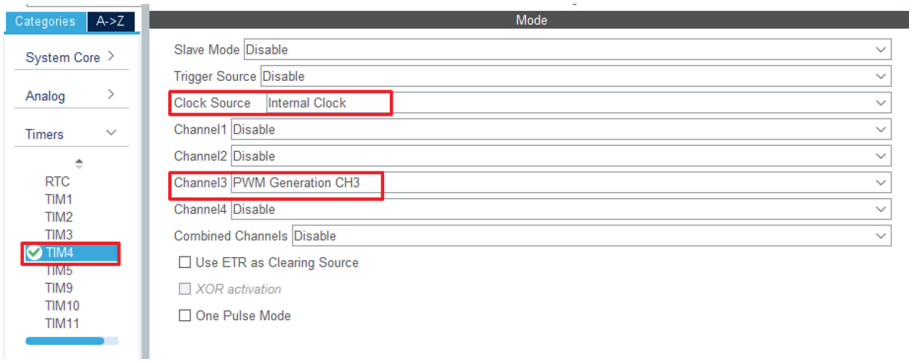


Figure 27: Timer TIM4 Mode settings

The duty cycle is set by the "Pulse" setting. To get a 50% duty cycle we set it to 8000. The LED will turn on for 50% of a second. (fig. 28)

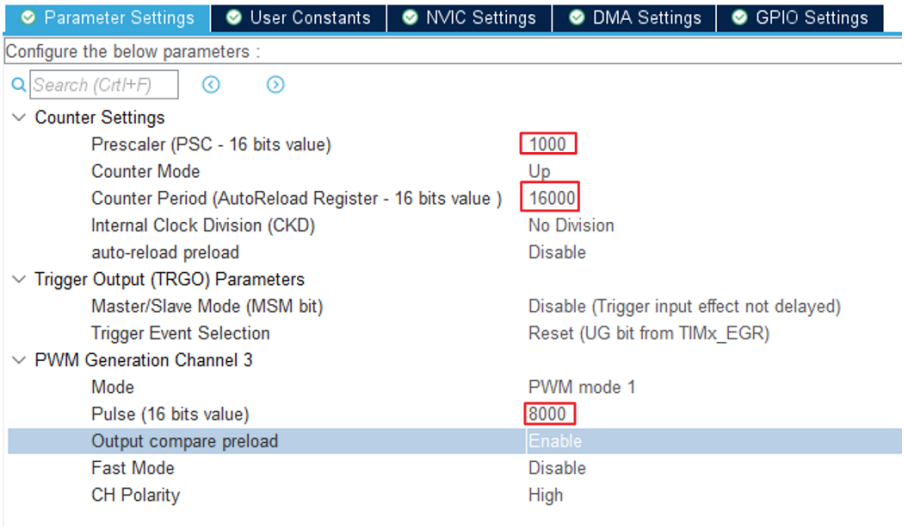


Figure 28: Timer parameter settings for 2Hz duty cycle.

We can calculate the timer period by knowing the Prescaler = 1000, counter period = 16000, APB2 clock = 16 MHz and Pulse = 8000.

$$f = \frac{16\,\text{MHz}}{1000} = 16\,\text{kHz}$$
$$T_{period} = \frac{1}{16\,\text{kHz}} * 16000 = 1\,\text{sec}$$
$$D = \frac{\text{Pulse}}{\text{Period}} = \frac{8000}{16000} = 50\,\%$$

Finish the CubeMX project and generate code.

7.1.2 Code Inspection

In CubeIDE you can find the tim.c source file. In here the timer configuration struct and PWM configuration can be found. It is possible to find the documentation of the sturcts in the HAL library handbook, and change the configuration fast during the project.

```
htim4.Instance = TIM4;
htim4.Init.Prescaler = 1000;
htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
htim4.Init.Period = 16000;
htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
{
  Error_Handler();
}
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
{
  Error_Handler();
}
if (HAL_TIM_PWM_Init(&htim4) != HAL_OK)
{
  Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
{
  Error_Handler();
}
```

Listing 10: tim.c: Timer configuration struct

```
sConfigOC.OCMode = TIM_OCMODE_PWM1;
sConfigOC.Pulse = 8000;
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_PWM_ConfigChannel(&htim4, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
{
  Error_Handler();
}
/* USER CODE BEGIN TIM4_Init 2 */

/* USER CODE END TIM4_Init 2 */
HAL_TIM_MspPostInit(&htim4);
```

Listing 11: PWM configuration

7.1.3 Start the PWM timer

In main.c start the timer after the initialization of the peripherals. Start the timer with the line `HAL_TIM_PMW_Start(&htim4, TIM_CHANNEL_3);`

7.1.4 Build & Run the code

You should see the red led toggle in a 2 Hz frequency.

7.2 Exercise

1. Change the duty cycle of the LED to 25%
2. Change the frequency to 100 Hz
3. Dim the LED by pressing the button. (should start at $D = 100\,\%$ and go down to $D = 0\,\%$ and then back to 100 %)
4. Control the brightness of an external LED. (Requires a breadboard, cables, resistor and an LED)

7.3 Summary

We showed how to control the an GPIO using PWM. We can use our knowledge to regulate the brightness of LEDs, control servo motors, etc.

7.4 Additional Resources

- Extensive timer explanation with additional examples:
  https://deepbluembedded.com/stm32-pwm-example-timer-pwm-mode-tutorial/
- Nucleo tutorial with video: https://controllerstech.com/pwm-in-stm32/
- Code example: https://github.com/CM134/STM32_HowTo/tree/7_PMW

8 UART

8.1 Background

UART stands for *Universal Asynchronous Receiver Transmitter* and is one the most used device-to-device communication protocols.[1]

By definition, UART is a hard communication protocol that uses asynchronous serial communication. Asynchronous meaning there is no signal clock to synchronize the output bits from the transmitter to the receiver.[1]

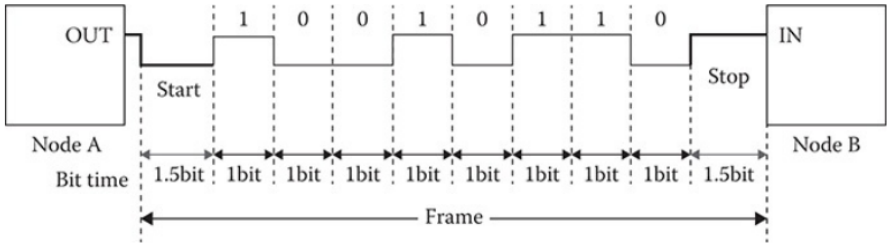


Figure 29: UART Block Diagram.[2]

Figure 29 shows how one device (Node A) sends information to another devide (Node B). Notice, how the stop and start bit are 50% longer. This is how Node B knows it is receiving a start or stop bit.

8.2 Prerequisites

- Reference manual (RM0383)[7]
- STM32CubeMX
- STM32CubeIDE
- USB-to-Serial bridge[8]
- A terminal emulator (We use Tera Term)

Because the STM32 DISCO is not able to make a virtual USB port, we need to use a USB-to-Serial bridge. Figure 31[9] shows how to connect it to the DISCO board.

When plugging the USB-to-Serial converter into your PC, it should show up under ports in "Device Manager".

Is this not the case, a driver update should do the trick.

8.3 Start Project in STM32CubeMX

Do the usual steps as the two times before. First, find the board we are using. Then delete the pins and start the configuration of the clocks. Click on RCC in 'System Core' menu and change 'HSC' to 'BYPASS' and 'LSC' to 'Crystal'. In the 'SYS' tab change 'Debug' to Serial Wire'.

[7]Reference manual and other documents can be found here: https://www.st.com/en/microcontrollers-microprocessors/stm32f411ve.html#documentation

[8]Not needed for NUCLEO boards

[9]https://microcontrollerslab.com/uart-usart-communication-stm32f4-discovery-board-hal-uart-driver/

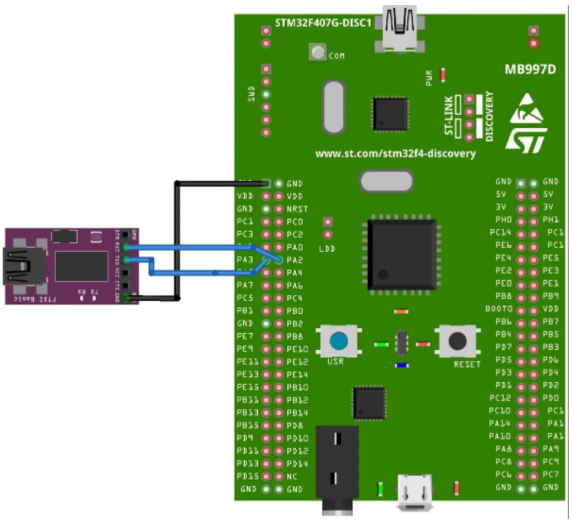


Figure 30: How to connect USB-to-Serial bridge

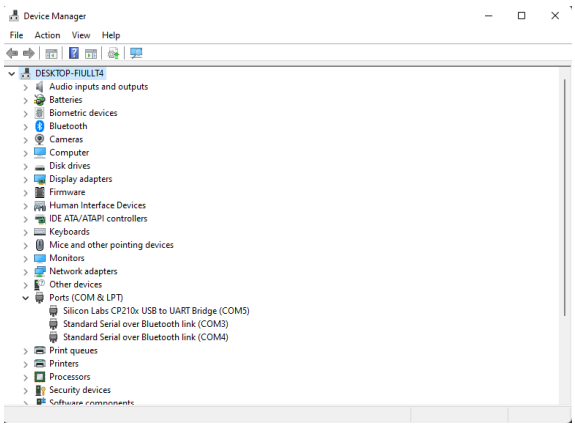


Figure 31: USB-to-Serial is plugged into the COM5 port.

8.3.1 Pinout Configuration

In the 'Pinout view', where you see the chip, select the pin for the USART2_TX PA2 and USART2_RX PA3 and configure it to be 'USART2_TX' and 'USART2_RX' respectively.

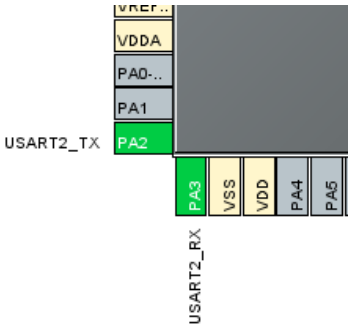


Figure 32: Configure pinout to USART2

Next, to enable USART, go to the connectivity tab and find USART2. In the 'Mode' dropdown menu, choose 'Asynchronous'.

8.3.2 Clock Configuration

The UART requires a precise clock configuration. Therefore in the clock configuration tab, we need to change to use HSI as it has a precision of about 1%.

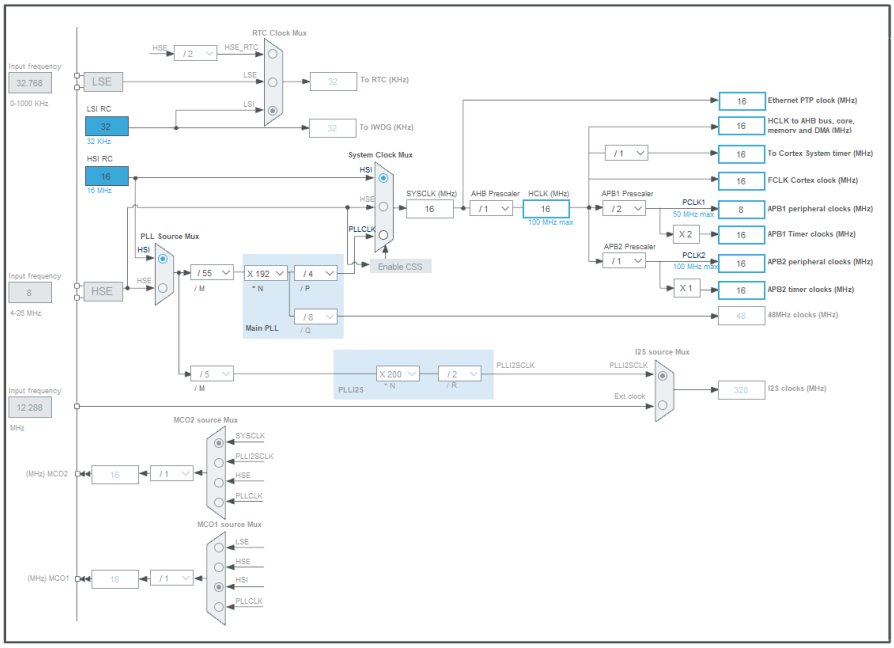


Figure 33: Clock configuration

You can now create project and generate code as explained in earlier section.

8.4 Code

In the generated code, you will see that in the USART has already been initialised.
Create a 'message' array, and use 'HAL_UART_Transmit(&huart2, (uint8_t*) message, 30, 100);' to transmit the message to via UART. 'HAL_UART_Transmit' takes the address of the UART, a pointer to the message, length of the data transmitted and TimeOut duration.

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
char message[30] = "Hello World!\r\n";
/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

```

```
26   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
27   HAL_Init();
28
29   /* USER CODE BEGIN Init */
30
31   /* USER CODE END Init */
32
33   /* Configure the system clock */
34   SystemClock_Config();
35
36   /* USER CODE BEGIN SysInit */
37
38   /* USER CODE END SysInit */
39
40   /* Initialize all configured peripherals */
41   MX_GPIO_Init();
42   MX_USART2_UART_Init();
43   /* USER CODE BEGIN 2 */
44
45   /* USER CODE END 2 */
46
47   /* Infinite loop */
48   /* USER CODE BEGIN WHILE */
49   while (1)
50   {
51     /* USER CODE END WHILE */
52     HAL_UART_Transmit(&huart2, (uint8_t*) message, 30, 100);
53     HAL_Delay(200);
54     /* USER CODE BEGIN 3 */
55   }
56   /* USER CODE END 3 */
57 }
```

Listing 12: UART

8.5 Demo

To see if the code is working, open your terminal emulator. When opening Tera term, you will be prompted as in figure 34. Choose "Serial", and from the dropdown choose the port the USB-to-Serial bridge is connected.

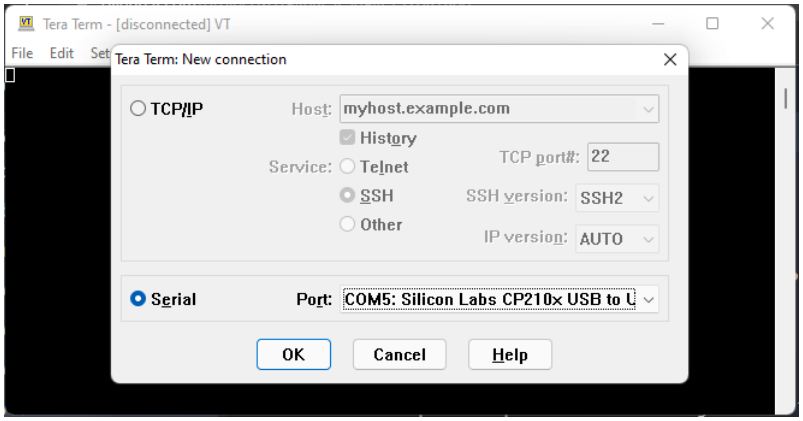


Figure 34: Prompt when opening Tera Term

After pressing "OK", you are taken to a terminal. At this point, you may see the terminal printing gibberish. This is due to Baud-rate being misaligned. in Tera term go "Setup → Serial port" and change the speed to match the Baud-rate in the CubeMX.

Tera term should now display the message.

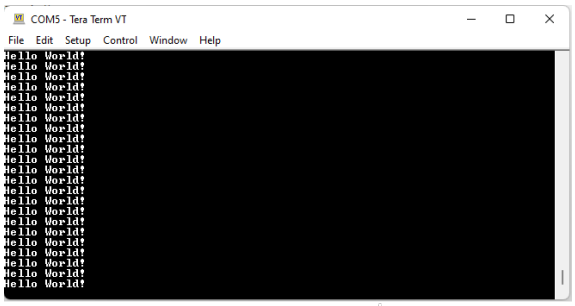
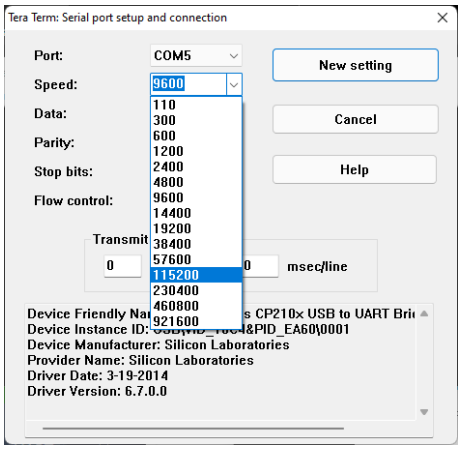
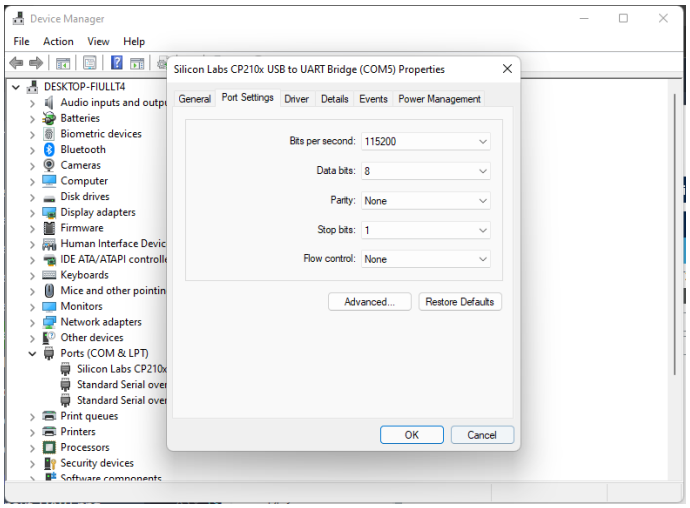


Figure 35: Hello World in the Tera Term

(a) Baud Rate in CubeMX



(b) Baud Rate in Tera Term



(c) Baud Rate in Device Manager

Figure 36: All the Baud Rates must be aligned.

8.6 Summary

We saw how to setup UART and transmit a message to your PC.

8.7 Additional Resources

- Code example: https://github.com/CM134/STM32_HowTo/tree/8_UART

9 ADC

9.1 Background

Analogue-to-digital converters (ADC) are commonly used peripherals in modern micro-controllers. They read an analog voltage and transform it into a binary value. Usually the value 0 corresponds to 0 V and the max reference voltage to the maximum bit resolution of the peripheral. Assume a 8-bit ADC port with 3.3 V reference the resolution is linearly quantized by $\Delta V = \frac{3.3\,V}{2^8\,\text{bit}} = 12.89\,\frac{\text{mV}}{\text{bit}}$. The way the ADC is implemented depends on the STM32 chip. Some use the successive-approximation register (SAR) technique where as others rely on sigma-delta modulation for better resolution in trade off to lower speeds.

The STN32F411E has one 12-bit ADC embedded which shares up to 16 external channels. The ADC can be served by the DMA controller. An interrupt is generated when the converted voltage is outside the programmed thresholds. Read the user manual for more detailed information about the ADC and see the HAL library documentation for function descriptions.

9.2 Prerequisites

- Potentiometer
- Breadboard
- Cables

9.3 Steps

9.3.1 Start Project in STM32CubeMX

The initial project configuration is rather simple. Start by configuring the system core as usual. Then select the only ADC-module of the STM32 by "Analog": "ADC1" on the left-hand side and set a check mark for channel IN0. The STM32 has 16 channels, but for now we only use a single one. All the parameter settings of the input channel can stay at the default.

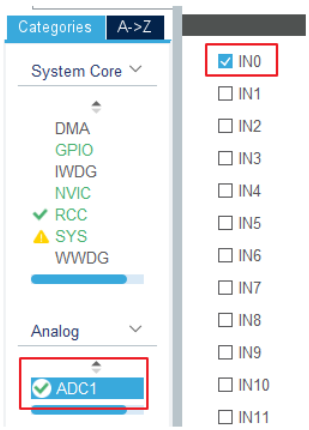


Figure 37: ADC configuration

9.4 Code

In the adc.c we can see there was a handle for the ADC created. Take a look at the initialization of the ADC.

From CubeMX it was possible to see that the ADC channel 0 is located at pin PA0. Connect a potentiometer to power and the 3rd pin to PA0 as shown in fig. 38.

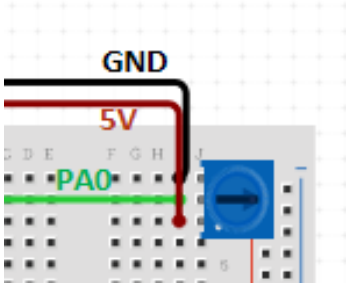


Figure 38: ADC pin configuration

To be able to read data from the ADC port we need to implement the following workflow: First: Start the ADC; Second: Poll for conversion; Last: Get the conversion.

In order to start working with the ADC we have to start it. You can find the HAL function from the documentation or by looking into the driver source file of the ADC and search it for start. There are 2 versions, one with interrupt and one without. We use the function without the interrupt and pass the ADC handle to it in our main.c file: `HAL_ADC_Start(&hadc1);`

Now we started the ADC. Next thing we want to do is convert the voltage to an integer value. Therefore we can use the function `HAL_ADC_PollForConversion(&hadc1,1)` for that. The second argument is the timeout in milliseconds.

Create a global variable to store the converted digital value and use `uint32_t HAL_ADC_GetValue(&hadc1)` to return the value.

```
while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

    // 1. Start ADC
    HAL_ADC_Start(&hadc1);

    // 2. Poll Conversion
    HAL_ADC_PollForConversion(&hadc1, 1);

    // 3. Get conversion
   sensor_value = HAL_ADC_GetValue(&hadc1);

  }
```

Listing 13: Single conversion mode: infinite while-loop

Now you can build the code and run it. In order to observe the ADC value, start the debug view, by pressing on the bug symbol next to the run symbol. Add the global variable to the "Live Expressions" view (fig. 39), and observe how the value changes when you turn the potentiometer.

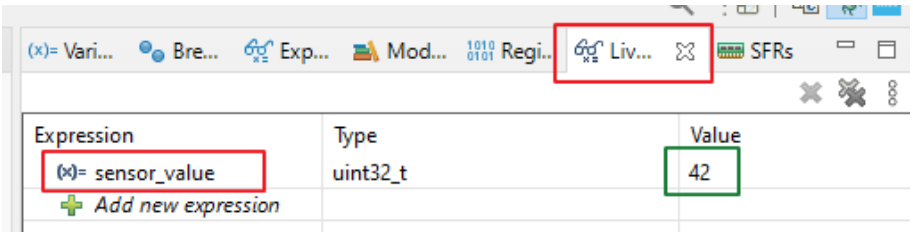


Figure 39: CubeIDE debug view showing the live sensor value.

9.4.1 Continuous Conversion

We only have to change a small thing in the ADC configuration to run the conversion continuously, which means there is no need to poll for conversion.
In CubeMX we need can use the same configuration as before, but change the default parameters and enable the continuous conversion mode. fig. 40

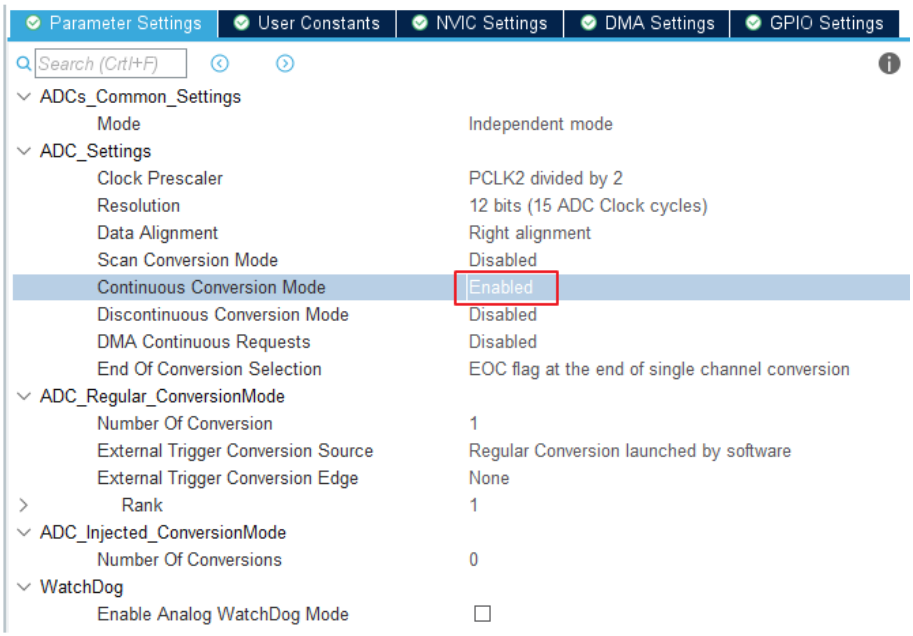


Figure 40: ADC enabled continuous conversion mode

In the IDE we can remove the poll for conversion function and use get value directly after starting the ADC. The output is the same as before.

```
1  // 1. Start ADC
2  HAL_ADC_Start(&hadc1);
3
4  while (1)
5  {
6    //2. Get Conversion
7    sensor_value = HAL_ADC_GetValue(&hadc1);
8  }
```

Listing 14: Continuous Conversion

9.5 Exercise

1. Read the on the DISCO board integrated temperature sensor and convert it to degrees Celsius. (Read manual for conversion)

In this exercise we would like to take a deeper look at direct access memory (DMA). DMAs are extremely helpful when working with large amounts of data, that has to be handled with out blocking the CPU. It connects peripherals data register directly to other peripherals or memory.
Setting up the DMA can be a bit cumbersome, but once it is done it is extremely helpful when working with constant data streams and large buffers such as strings.
It is possible to set up ping pong buffers, which enable to process data for one 'partition' of the memory, while the DMA stores data in another 'partition'.[10]

2. Use DMA instead of CPU to handle the memory from ADC. See the digikey tutorial [11] on how to do so.

9.6 Summary

9.7 Additional Resources

- Extensive ADC explanation with additional examples:
  https://deepbluembedded.com/stm32-adc-tutorial-complete-guide-with-examples/
- ST Application note: "How to get the best ADC accuracy in STM32 microcontrollers"
  https://www.st.com/resource/en/application_note/cd00211314-how-to-get-the-best-adc-accuracy-in-stm32-microco
  pdf
- Code example:
  https://github.com/CM134/STM32_HowTo/tree/9_ADC_SingleModeConversion
  https://github.com/CM134/STM32_HowTo/tree/9_ADC_ContinousConversion

[10] https://embedded.fm/blog/2017/3/21/ping-pong-buffers
[11] https://www.digikey.com/en/maker/projects/getting-started-with-stm32-working-with-adc-and-dma/
f5009db3a3ed4370acaf545a3370c30c

10 Real-Time Operating System

10.1 Background

A General Purpose Operating System (GPOS) such as Windows, IOS and Android focuses on maximizing throughput. This means GPOS will slow down priority processes to ensure a maximum number of processes are completed as fast as possible. GPOS are therefore especially efficient at multitasking, which makes them a great choice for everyday regular use. However, as the time of which a GPOS completes a process varies depending on the number of processes happening at the same time, GPOS can not guarantee a process to be completed in real time.
Therefore when you need to guarantee a process to be completed within certain amount of time, you need a Real-Time Operating Systems (RTOS). RTOS focuses on prioritizing the processes, so should a higher priority task enter the system, e.g. an airbag during a car crash, the high priority task does not share bandwidth with the other tasks. RTOS are therefore able to complete task within a predictable amount of time, every time.

10.2 Prerequisites

- Reference manual (RM0383)[12]
- STM32CubeMX
- STM32CubeIDE
- STM32F4xxE-DISCO

10.3 Introduction to FreeRTOS

10.3.1 Start Project in STM32CubeMX

Like the other times, start the CubeMX project as in the weeks before. Use the system defaults for the clocks. Clear all pinouts.
Set the green LED as GPIO output and the push button as GPIO input.

10.3.2 Pinout Configuration

In the 'Pinout view', where you see the chip, select the pin for the blue button PA0-WKUP (bottom left corner) and configure it to be 'GPIO_EXTI0' (fig. 19). This sets the external event register EXTI0 for the button.

Next activate a LED to be a 'GPIO_Output'. You can change the labels of the button and the LED by right clicking the pin and 'enter a user label. Here we chose 'BTN' and 'LED' for LED on port PD12. Also, go ahead and enable USART2 and set it to asynchronous.

10.3.3 Setting up FreeRTOS

First we need to enable a timer. This is done in "System Core" under "SYS". Go ahead and choose "TIM3".

To enable FreeRTOS, head to "Middleware" and choose "FreeRTOS". To enable FreeRTOS, choose "CMSIS_V1". CMSIS is an API for RTOS application that is used across many RTOS platforms.

[12]Reference manual and other documents can be found here: https://www.st.com/en/microcontrollers-microprocessors/stm32f411ve.html#documentation

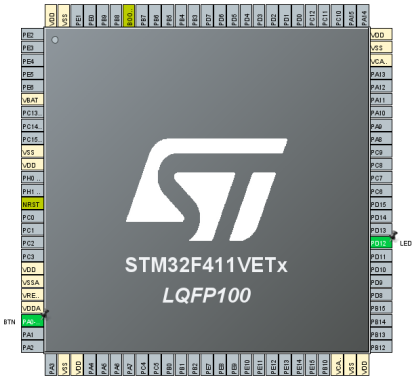


Figure 41: Pin out configuration

All the configurations seen in fig. 42 will be available in a generated "FreeRTOSconfig.h" file. So we will leave them as they are for now. Head into "Tasks and Queues" tab. Here you can see the tasks that are created. You should only see a default task with normal priority at the current point in time. However, let go ahead and add a couple more tasks. When creating a new task, you also have the option to choose the priority of the task. For now, we will leave both at "osPriorityIdle". We will see how to change this later.

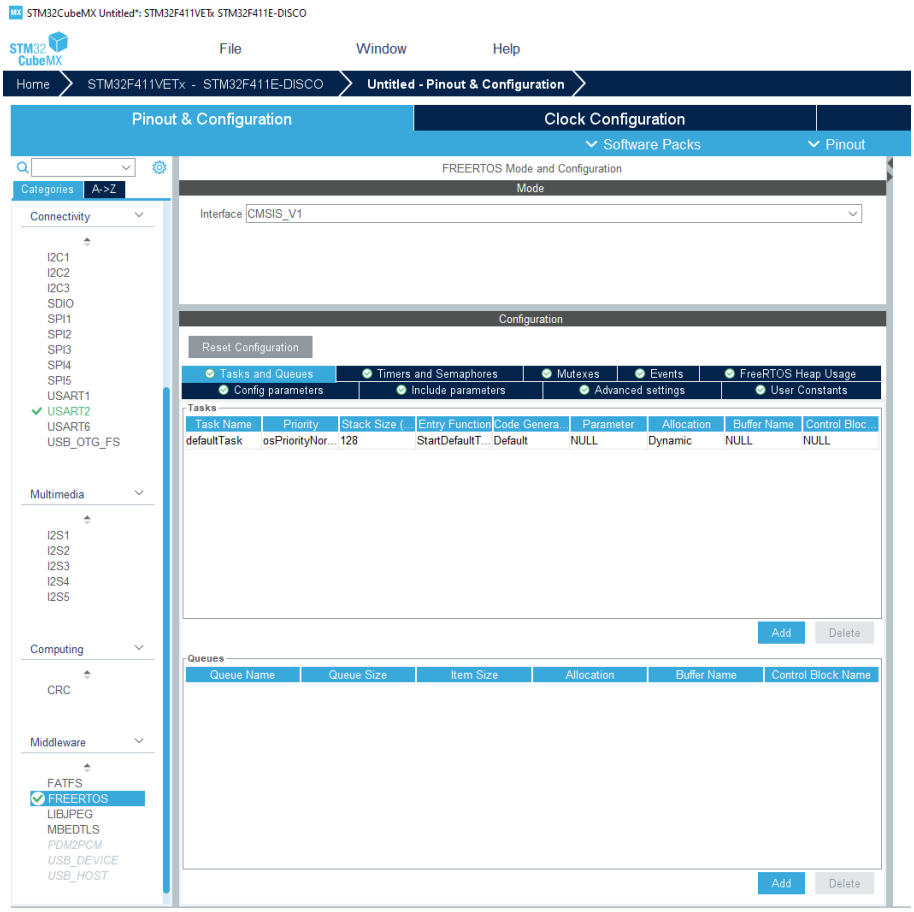


Figure 42: FreeRTOS enabled

Next we will add a "Mutex", which means mutual exclusives. This will allow us to use a shared

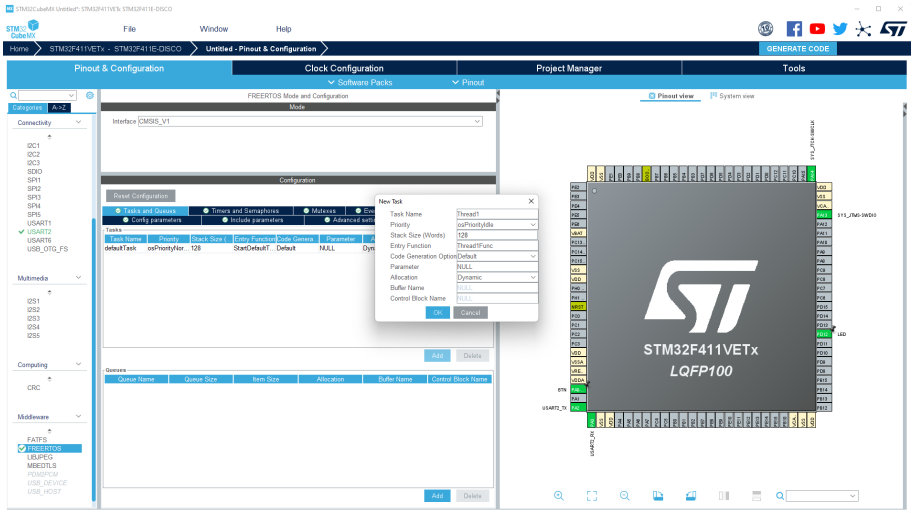


Figure 43: Adding a new task

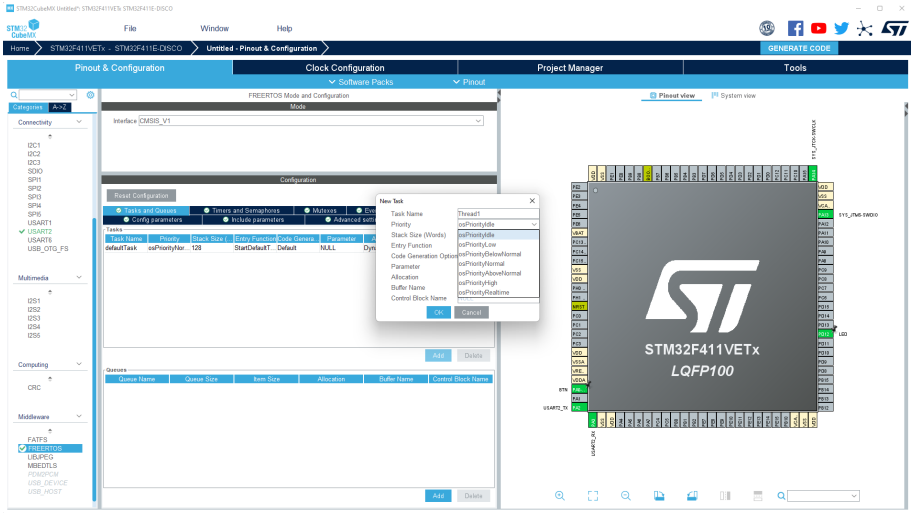


Figure 44: Task priorities

resource effectively. This is important, because if you have threads[13] trying to write out UART at the same time, the UART might crash. Same thing with peripherals like an LCD screen. If you write two things at the same time, the peripherals may crash. We need to set up a "Mutex" so only one thread accesses it at a time. When the first thread is done, the second can access it. In our example, we create a "mutex" for our UART.

When this is done, go ahead and generate your code.

[13] tasks being executed

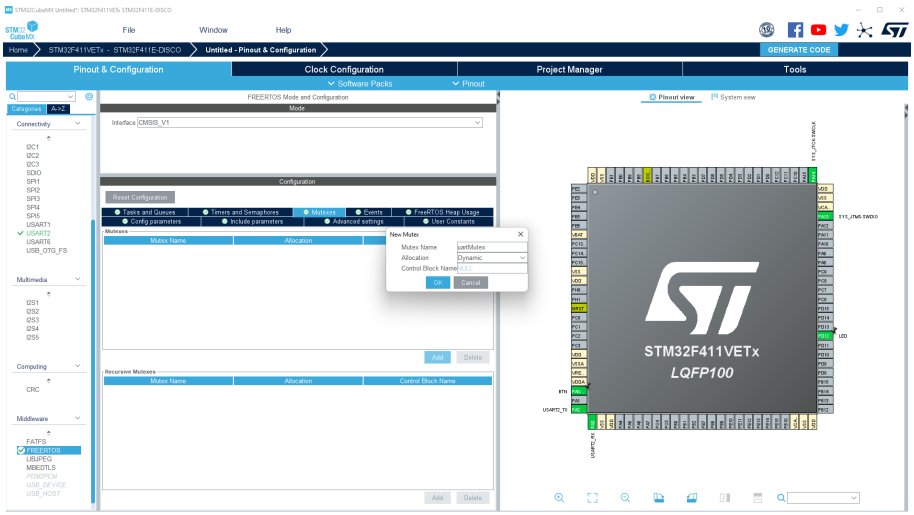


Figure 45: Creation of UART Mutex

10.3.4 Multiple Threads

Each thread and Mutex created in the previous section has been given an ID as can be seen in listing 15. A thread is basically a "main" function, an infinite loop. This type of function has been created for each of threads created.

```
1  osThreadId defaultTaskHandle;
2  osThreadId Thread1Handle;
3  osThreadId Thread2Handle;
4  osMutexId uartMutexHandle;
5
6  ...
7  ...
8  ...
9
10 void StartDefaultTask(void const * argument)
11 {
12   /* USER CODE BEGIN */
13   /* Infinite loop */
14   for(;;)
15   {
16     /*
17     CODE
18     */
19   }
20   /* USER CODE END */
21 }
22
23 void Thread1Func(void const * argument)
24 {
25   /* USER CODE BEGIN Thread1Func */
26   /* Infinite loop */
27   for(;;)
28   {
29     /*
30     CODE
31     */
32   }
33   /* USER CODE END Thread1Func */
34 }
35
36 void Thread2Func(void const * argument)
37 {
38   /* USER CODE BEGIN Thread2Func */
39   /* Infinite loop */
40   for(;;)
41   {
42     /*
```

```
    CODE
    */
  }
  /* USER CODE END Thread2Func */
}
```

Listing 15: Generated code

To visualize each function running, we will create three variable to profile that the function is running using a counter variable.

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
void StartDefaultTask(void const * argument);
void Thread1Func(void const * argument);
void Thread2Func(void const * argument);


typedef uint32_t TaskProfiler;

TaskProfiler Thread1Profiler, Thread2Profiler, DefaultThreadProfiler;
```

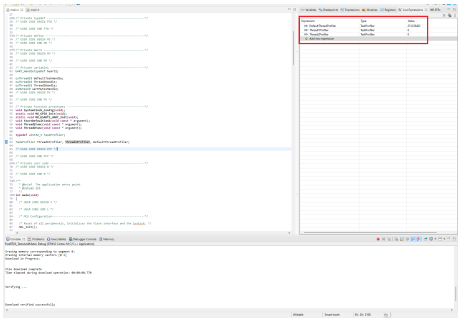
Listing 16: TaskProfilers

After uploading the code to the board, add each variable to the "Live expressions". When you run the code, you should see that only the "DefaultThreadProfiler" is counting, and the others are staying at 0. This is due to the priority of each task. Remember thread1 and thread2 are at idle, while the default thread is at normal priority. Lets put them all at equal priority.

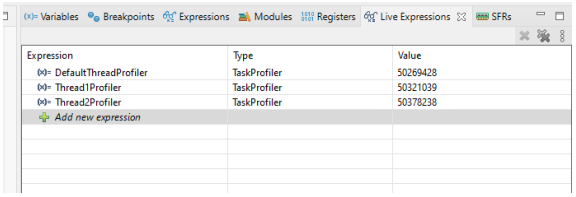
```
  /* Create the thread(s) */
  /* definition and creation of defaultTask */
  osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 128);
  defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);

  /* definition and creation of Thread1 */
  osThreadDef(Thread1, Thread1Func, osPriorityNormal, 0, 128);
  Thread1Handle = osThreadCreate(osThread(Thread1), NULL);

  /* definition and creation of Thread2 */
  osThreadDef(Thread2, Thread2Func, osPriorityNormal, 0, 128);
  Thread2Handle = osThreadCreate(osThread(Thread2), NULL);
```

Listing 17: All threads at equal priority

As can be seen in fig. 46b, the values are not exactly the same. This is because the threads are not running simultaneously, but rather is running an equal amount of time before being put on hold. So one thread may run 10ms, then be put on hold until the other threads have run for 10ms.



(a) Default thread running



(b) All threads are running

Figure 46: Example of threads running at different priorities.

10.3.5 FreeRTOSConfig.h

In the "Inc" folder, you find the FreeRTOSConfig.h file. This file gives the opportunity to change all the parameters needed to fit to your application. The documentation can be found at FreeRTOS' website: https://freertos.org/a00110.html

10.4 Accessing Hardware Drivers from Multiple Threads

Lets try toggle the LED and print to Tera Term using thread1. Use the function in listing 18

```
void Thread1Func(void const * argument)
{
  /* USER CODE BEGIN Thread1Func */
  /* Infinite loop */
  uint8_t message[24] ="Greetings from Thread1\r\n";
  for(;;)
  {
    HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12);
    HAL_UART_Transmit(&huart2,message,24,10);
    osDelay(1000);
  }
  /* USER CODE END Thread1Func */
}
```

Listing 18: Thread1Func

To make it a bit more advanced, lets add something thread2. listing 19 shows the function for thread2, which prints "Greetings from thread2" when button is pressed while the function runs.

```
void Thread2Func(void const * argument)
{
  /* USER CODE BEGIN Thread1Func */
  /* Infinite loop */
  uint8_t message[24] ="Greetings from Thread1\r\n";
  for(;;)
  {
    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) != GPIO_PIN_RESET){
      HAL_UART_Transmit(&huart2,message,24,10);
    }
    osDelay(1000);
  }
  /* USER CODE END Thread2Func */
}
```

Listing 19: Thread2Func

Running to threads to the same shared resource may cause conflicts. However, not in this case due to the large amount of delay in each thread. To make sure there is no conflict, we use the "mutex" we created in the beginning.
The "mutex" works as a key to using the resource. A key that is taken if available, and then given back whenever there is no need for a function to use the shared resource anymore. Below is an example of how it is used.

```
  xSemaphoreTake(uartMutexHandle,portMAX_DELAY);
        /*
            CODE TO BE EXECUTED USING A SHARED RESOURCE.
        */
  xSemaphoreGive(uartMutexHandle);
```

Listing 20: xSemaphoreTake and xSemaphoreGive

xSemaphoreTake takes the "uartMutexHandle" key if it is available. "portMAX_DELAY" makes the function wait an infinite amount of time until the key is available. If a number is provided instead, the function will wait that amount of time, and then use the resource no matter what.

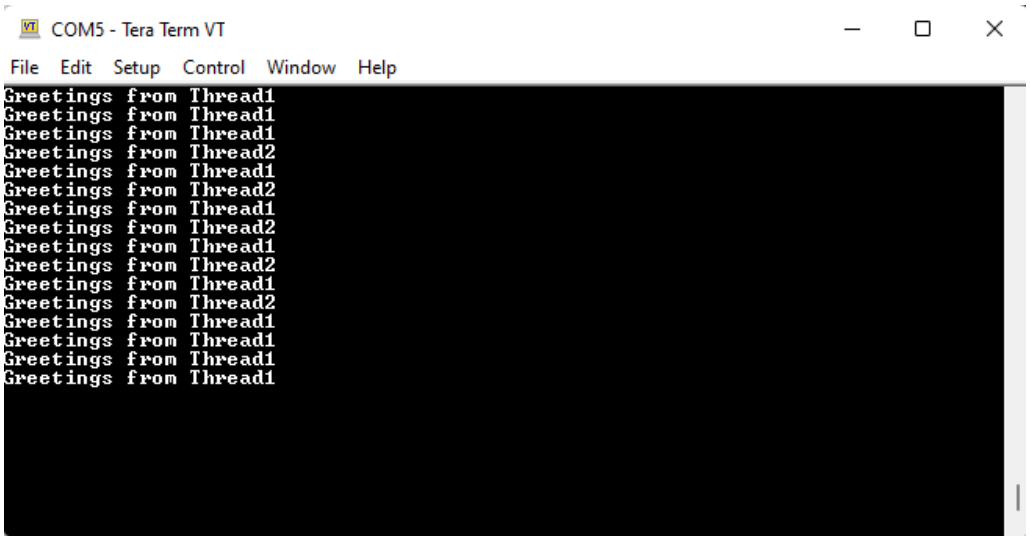


Figure 47: Thread1 and Thread2 in Tera Term.

10.4.1 Sending Notifications between threads

It is possible to send notifications between threads, and use these notifications to perform actions.

```
void Thread1Func(void const * argument)
{
  /* USER CODE BEGIN Thread1Func */
  /* Infinite loop */
  uint8_t message[24] ="Greetings from Thread1\r\n";
  uint32_t notifyValue;
  for(;;)
  {
    xTaskNotifyWait(pdFALSE,0xFF,&notifyValue,portMAX_DELAY);
    if ((notifyValue & 0x01) != 0x00){
      xSemaphoreTake(uartMutexHandle,portMAX_DELAY);
      HAL_UART_Transmit(&huart2,message,24,10);
      osDelay(1000);
      xSemaphoreGive(uartMutexHandle);
    }
  }
  /* USER CODE END Thread1Func */
}


void Thread2Func(void const * argument)
{
  /* USER CODE BEGIN Thread2Func */
  /* Infinite loop */
  uint8_t message[24] ="Greetings from Thread2\r\n";
  uint8_t message1[24] ="Thread1 Notified!\r\n";
  for(;;)
  {
    xSemaphoreTake(uartMutexHandle,portMAX_DELAY);
    HAL_UART_Transmit(&huart2,message,24,10);
    xSemaphoreGive(uartMutexHandle);

    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) != GPIO_PIN_RESET){
      osDelay(1000);
      //HAL_UART_Transmit(&huart2,message,24,10);
      xSemaphoreTake(uartMutexHandle,portMAX_DELAY);
      HAL_UART_Transmit(&huart2,message1,24,10);
      xSemaphoreGive(uartMutexHandle);
      xTaskNotify(Thread1Handle,0x01,eSetBits);

    }
    osDelay(10);
  }
```

```
44   /* USER CODE END Thread2Func */
45 }
```

Listing 21: xTaskNotify and xTaskNotifyWait

The code above shows two threads, where thread2 is sending out a message to UART, but when the pushbutton is set, a notification is sent to thread1.
This is done using *xTaskNotify*, which sets the bit 0 in the NotifyValue. *xTaskNotifyWait* waits for a notification and sets the bit in the NotifyValue.

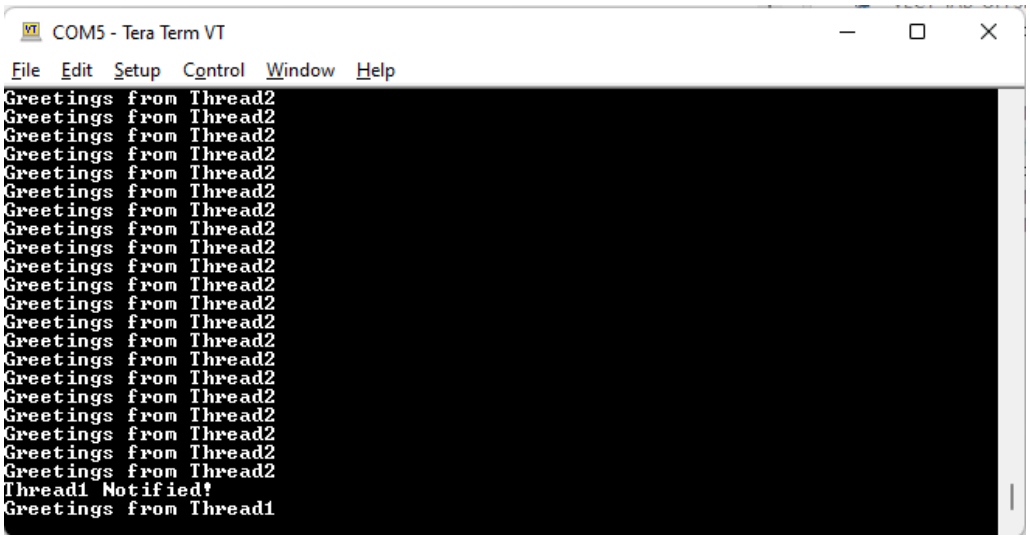


Figure 48: Notification sent from thread2 to thread1

10.5 Summary

In this section we introduced the Real-time operating system kernel, FreeRTOS. We showed how to set up FreeRTOS using the STM32CubeMX. Then, we looked at how to run different threads at different priorities, and how to use a mutex to ensure no conflicts between shared resources. Finally, we showed how to sent notifications between threads.

10.6 Additional Resources

- FreeRTOS: https://freertos.org/
- About FreeRTOS: https://www.highintegritysystems.com/rtos/what-is-an-rtos/#:~:text=A%20Real%20Time%20Operating%20System,on%20a%20single%20processing%20core.
- Code example: https://github.com/CM134/STM32_HowTo/tree/10_RTOS_TasksAndMutex

References

[1] AnalogDialogue, 2022. [online] Available at: <h20a,going%20to%20the%20receiving%20end.>[Accessed 10 March 2022].

[2] Norris, Donald J., 2018. Programming with STM32 - Getting Started with the Nucleo Board and C/C++. McGraw-Hill Education, ISBN: 978-1-26-003131-7.

[3] Fleming, B. (2011). Microcontroller Units in Automobiles [Automotive Electronics]. IEEE Vehicular Technology Magazine, 6, 4-8.

[4] DigiKey, 2022. [online] Available at: <https://www.digikey.dk/en/maker/projects/getting-started-with-stm32-timers-and-timer-interrupts/d08e6493cefa486fb1e79c43c0b08cc6>[Accessed 10 March 2022].