# Laboratory 1: Basic MCU development on STM32

## *Objectives*

1. Understand the STM32 microcontroller
2. Introduce the basic of C programming using STM32CubeIDE
3. Understand the step debugging for MCU on Eclipse platform
4. Understand the System clock configuration
5. Understand the STM32Cube embedded software

## *Embedded System*

"An embedded system is one that has embedded software and computer-hardware, which makes it a system dedicated for an application(s) or specific part of an application or product or a part of a larger system." – Raj Kamal

## *Microcontroller*

A microcontroller (MCU) is a System-On-Chip (SoC) that contain a processor core, memory, and some peripherals Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications consisting of various discrete chips.

Modern embedded systems are often based on microcontrollers (i.e. CPUs with integrated memory or peripheral interfaces), but ordinary microprocessors (using external chips for memory and peripheral interface circuits) are also common, especially in more-complex systems. In either case, the processor(s) used may be types ranging from general purpose to those specialize in certain class of computations, or even custom designed for the application at hand. A common standard class of dedicated processors is the digital signal processor (DSP).

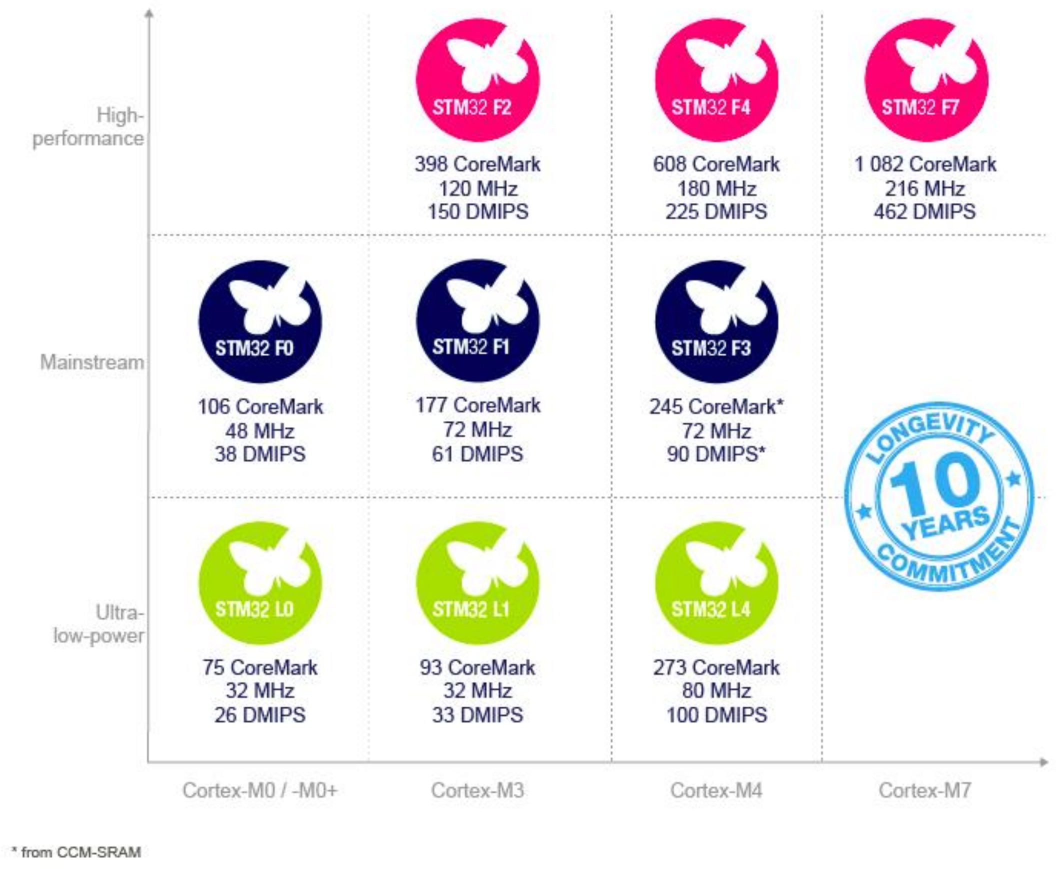
https://en.wikipedia.org/wiki/Embedded_system

https://en.wikipedia.org/wiki/Microcontroller

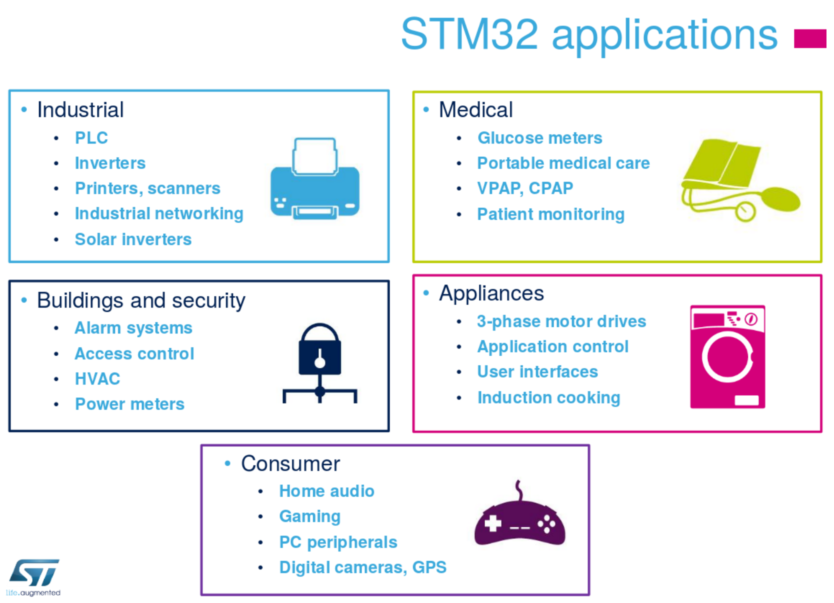
## *STM32 32-bit ARM Cortex MCUs*

The STM32 family of 32-bit Flash microcontrollers based on the ARM® Cortex®-M processor is designed to offer new degrees of freedom to MCU users. It offers a 32-bit product range that combines very high performance, real-time capabilities, digital signal processing, and low-power, low-voltage operation, while maintaining full integration and ease of development.

The unparalleled and large range of STM32 devices, based on an industry-standard core and accompanied by a vast choice of tools and software, makes this family of products the ideal choice, both for small projects and for entire platform decisions.



http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus.html?querycriteria=productId=SC1169



## STM32 applications

- Industrial
  - PLC
  - Inverters
  - Printers, scanners
  - Industrial networking
  - Solar inverters
- Medical
  - Glucose meters
  - Portable medical care
  - VPAP, CPAP
  - Patient monitoring
- Buildings and security
  - Alarm systems
  - Access control
  - HVAC
  - Power meters
- Appliances
  - 3-phase motor drives
  - Application control
  - User interfaces
  - Induction cooking
- Consumer
  - Home audio
  - Gaming
  - PC peripherals
  - Digital cameras, GPS

life.augmented

Coffee machine
Blenders
Class B
Vacuum cleaner
Induction cooking
Consumer appliance
User interface
Gaming
USB
CEC
Digital TV
Wireless charging
Computer and peripherals
IoT
Battery charger
Industrial
Street light
Smart power
CEC
Power tools
Motor control

## STM32Cube Embedded Software

With STM32Cube, STMicroelectronics provides a comprehensive software tool, significantly reducing development efforts, time and cost.

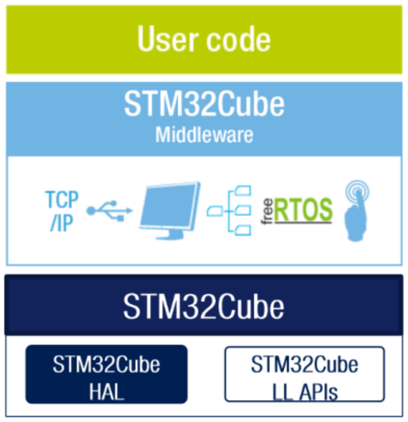
STM32Cube consists of (usable together or independantly):

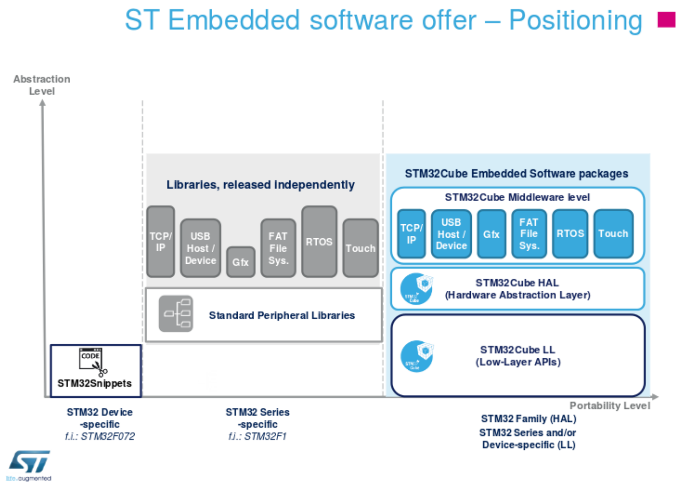
The STM32CubeMX, featuring

- Configuration C code generation for pin multiplexing, clock tree, peripherals and middleware setup with graphical wizards
- Generation of IDE ready projects for a integrated development environment tool chains
- Power consumption calculation for a user-defined application sequence
- Direct import of STM32 Cube embedded software libraries from st.com
- Integrated updater to keep STM32CubeMX up-to-date

STM32Cube embedded software libraries, including:

- The HAL hardware abstraction layer, enabling portability between different STM32 devices via standardized API calls
- The Low-Layer (LL) APIs, a light-weight, optimized, expert oriented set of APIs designed for both performance and runtime efficiency
- A collection of Middleware components, like RTOS, USB library, file system, TCP/IP stack, Touch sensing library or Graphic Library (depending on the MCU series)




ST Embedded software offer – Positioning
Abstraction Level
Libraries, released independently
TCP/IP
USB Host / Device
Gfx
FAT File Sys.
RTOS
Touch
Standard Peripheral Libraries
STM32Cube Embedded Software packages
STM32Cube Middleware level
STM32Cube HAL (Hardware Abstraction Layer)
STM32Cube LL (Low-Layer APIs)
CODE
STM32Snippets
Portability Level
STM32 Device -specific f.i.: STM32F072
STM32 Series -specific f.i.: STM32F1
STM32 Family (HAL) STM32 Series and/or Device-specific (LL)

[http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-embedded-software.html](http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-embedded-software.html)

***STM32CubeMX – STM32Cube initialization code generator***

STM32CubeMX is part of STMicroelectronics STMCube™ original initiative to ease developers life by reducing development efforts, time and cost. STM32Cube covers STM32 portfolio.

STM32Cube includes the STM32CubeMX which is a graphical software configuration tool that allows generating C initialization code using graphical wizards.

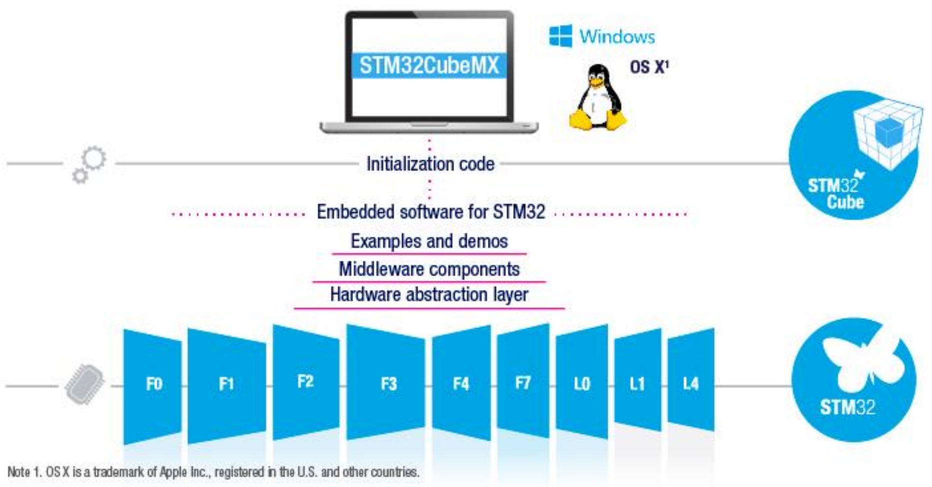
It also embeds a comprehensive software platform, delivered per series (such as STM32CubeF4 for STM32F4 series). This platform includes the STM32Cube HAL (an STM32 abstraction layer embedded software, ensuring maximized portability across STM32 portfolio), plus a consistent set of middleware components (RTOS, USB, TCP/IP and graphics). All embedded software utilities come with a full set of examples.

STM32CubeMX is a graphical tool that allows configuring STM32 microcontrollers very easily and generating the corresponding initialization C code through a step-by-step process.

Step one consists in selecting the STMicroelectronics STM32 microcontroller that matches the required set of peripherals.

The user must then configure each required embedded software thanks to a pinout-conflict solver, a clock-tree setting helper, a power-consumption calculator, and an utility performing MCU peripheral configuration (GPIO, USART, ..) and middleware stacks (USB, TCP/IP, ...).

Finally, the user launches the generation of the initialization C code based on the selected configuration. This code is ready to be used within several development environments. The user code is kept at the next code generation.


STM32CubeMX
Windows
OS X¹
Initialization code
Embedded software for STM32
Examples and demos
Middleware components
Hardware abstraction layer
F0
F1
F2
F3
F4
F7
L0
L1
L4
STM32 Cube
STM32
Note 1. OS X is a trademark of Apple Inc., registered in the U.S. and other countries.

http://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-configurators-and-code-generators/stm32cubemx.html

**STM32CubeIDE**

STM32CubeIDE is an advanced C/C++ development platform with IP configuration, code generation, code compilation, and debug features for STM32 microcontrollers. It is based on the ECLIPSE™/CDT framework and

GCC toolchain for the development, and GDB for the debugging. It allows the integration of the hundreds of existing plugins that complete the features of the ECLIPSE™ IDE.
STM32CubeIDE integrates all STM32CubeMX functionalities to offer all-in-one tool experience and save installation and development time. After the selection of an empty STM32 MCU or preconfigured microcontroller from the selection of a board, the project is created and initialization code generated. At any time during the development, the user can return to the initialization and configuration of the IPs or middleware and regenerate the initialization code with no impact on the user code.
STM32CubeIDE includes build and stack analyzers that provide the user with useful information about project status and memory requirements.
STM32CubeIDE also includes standard and advanced debugging features including views of CPU core registers, memories, and peripheral registers, as well as live variable watch, Serial Wire Viewer interface, or fault analyzer. (From ST website)

### *STM32F4*

STM32F4 series of high-performance MCUs with DSP and FPU instructions

The ARM® Cortex®-M4-based STM32F4 series MCUs leverage ST's NVM technology and ST's ART Accelerator™ to reach the industry's highest benchmark scores for Cortex-M-based microcontrollers with up to 225 DMIPS/608 CoreMark executing from Flash memory at up to 180 MHz operating frequency.

STM32F4XX

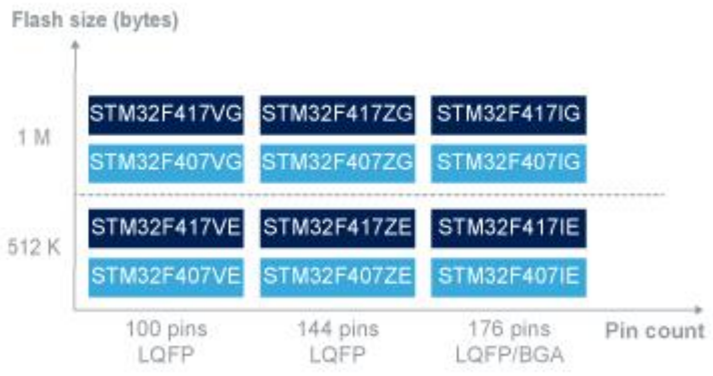
The STM32F4XX lines are designed for medical, industrial and consumer applications where the high level of integration and performance, embedded memories and rich peripheral set inside packages as small as 10 x 10 mm are required.
The STM32F407/417 offers the performance of the Cortex™-M4 core (with floating point unit) running at 168 MHz.

Performance: At 168 MHz, the STM32F4XX deliver 210 DMIPS/566 CoreMark performance executing from Flash memory, with 0-wait states using ST's ART Accelerator. The DSP instructions and the floating point unit enlarge the range of addressable applications.

Power efficiency: ST's 90 nm process, ART Accelerator and the dynamic power scaling enables the current consumption in run mode and executing from Flash memory to be as low as 238 µA/MHz at 168 MHz.

Rich connectivity: Superior and innovative peripherals: Compared to the STM32F4x5 series, the STM32F4XX product lines feature Ethernet MAC10/100 with IEEE 1588 v2 support and a 8- to 14-bit parallel camera interface to connect a CMOS camera sensor.

http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f4-series/stm32f407-417.html?querycriteria=productId=LN11



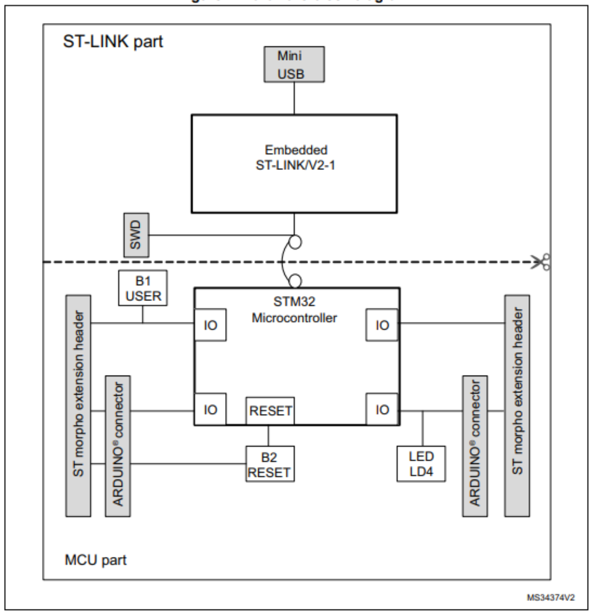
NUCLEO-F411

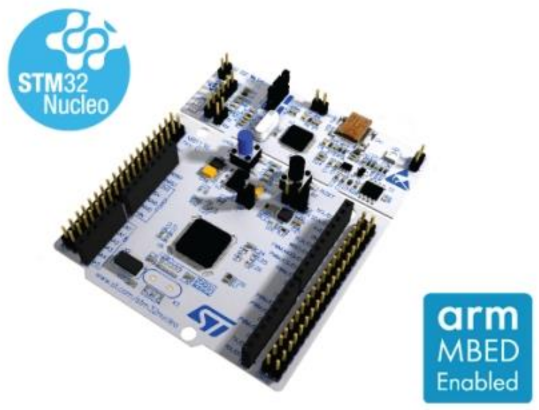
The STM32 Nucleo-64 board provides an affordable and flexible way for users to try out new concepts and build prototypes by choosing from the various combinations of performance and power consumption features, provided by the STM32 microcontroller.

Traditionally, Microcontroller board requires an external board for programming and debugging. In case of Nucleo board family, ST provides this functionality through the Mini-USB through ST-LINK protocol. See more details about the board here.

https://www.st.com/en/evaluation-tools/nucleo-f411re.html

**Figure 2. Hardware block diagram**





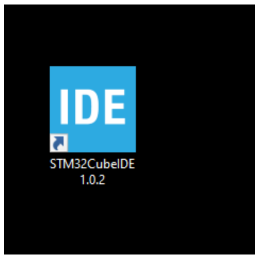
## *Digital Logic*

***Digital Logic*** is an alternative term for ***Digital Electronic (or Digital Circuits)***. Digital is a way for representing the electronic signals as High (True) and Low (False) rather than the real analog values (voltages). Thus, small changes in the analog levels (voltages), such as noise from circuits or communication channels, does not effect the digital signal. This allows electronic devices to switch to known states than producing specific values (voltages).

Usually, a signal level (voltage) near a reference ground level is called Low or False or 0. A signal level (voltage) near the supply voltage is called High or True or 1. With this notion in our hands, we can simply refer to our digital value as high or low.

*Getting Started*

1 Run System Workbench for STM32.



It will ask you to create a workspace. Choose work space that you would like your program to be in. **However, make sure that there is no space in any of the pathname.** For example

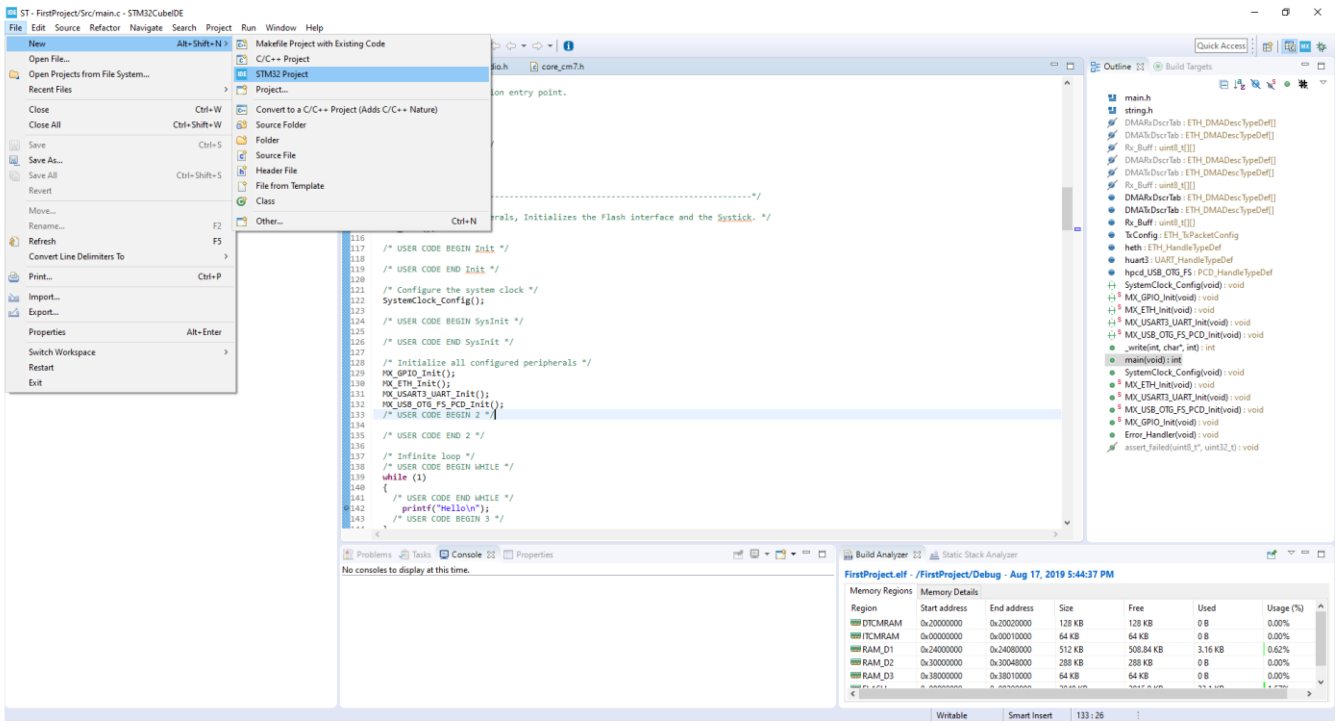
C:\ST\STM32

Is ok, but

C:\my work\STM32

Will lead to some problems later on.

2 Choose File -> New -> STM32 Project



```
ion entry point.

-------------------------------------------------------*/

erals, Initializes the Flash interface and the Systick. */

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_ETH_Init();
  MX_USART3_UART_Init();
  MX_USB_OTG_FS_PCD_Init();
  /* USER CODE BEGIN 2 */

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
      printf("Hello\n");
    /* USER CODE BEGIN 3 */
```

```
main.h
string.h
DMARxDscrTab : ETH_DMADescTypeDef[]
DMATxDscrTab : ETH_DMADescTypeDef[]
Rx_Buff : uint8_t[][]
DMARxDscrTab : ETH_DMADescTypeDef[]
DMATxDscrTab : ETH_DMADescTypeDef[]
Rx_Buff : uint8_t[][]
DMARxDscrTab : ETH_DMADescTypeDef[]
DMATxDscrTab : ETH_DMADescTypeDef[]
Rx_Buff : uint8_t[][]
TxConfig : ETH_TxPacketConfig
heth : ETH_HandleTypeDef
huart3 : UART_HandleTypeDef
hpcd_USB_OTG_FS : PCD_HandleTypeDef
SystemClock_Config(void) : void
MX_GPIO_Init(void) : void
MX_ETH_Init(void) : void
MX_USART3_UART_Init(void) : void
MX_USB_OTG_FS_PCD_Init(void) : void
_write(int, char*, int) : int
main(void) : int
SystemClock_Config(void) : void
MX_ETH_Init(void) : void
MX_USART3_UART_Init(void) : void
MX_USB_OTG_FS_PCD_Init(void) : void
MX_GPIO_Init(void) : void
Error_Handler(void) : void
assert_failed(uint8_t*, uint32_t) : void
```

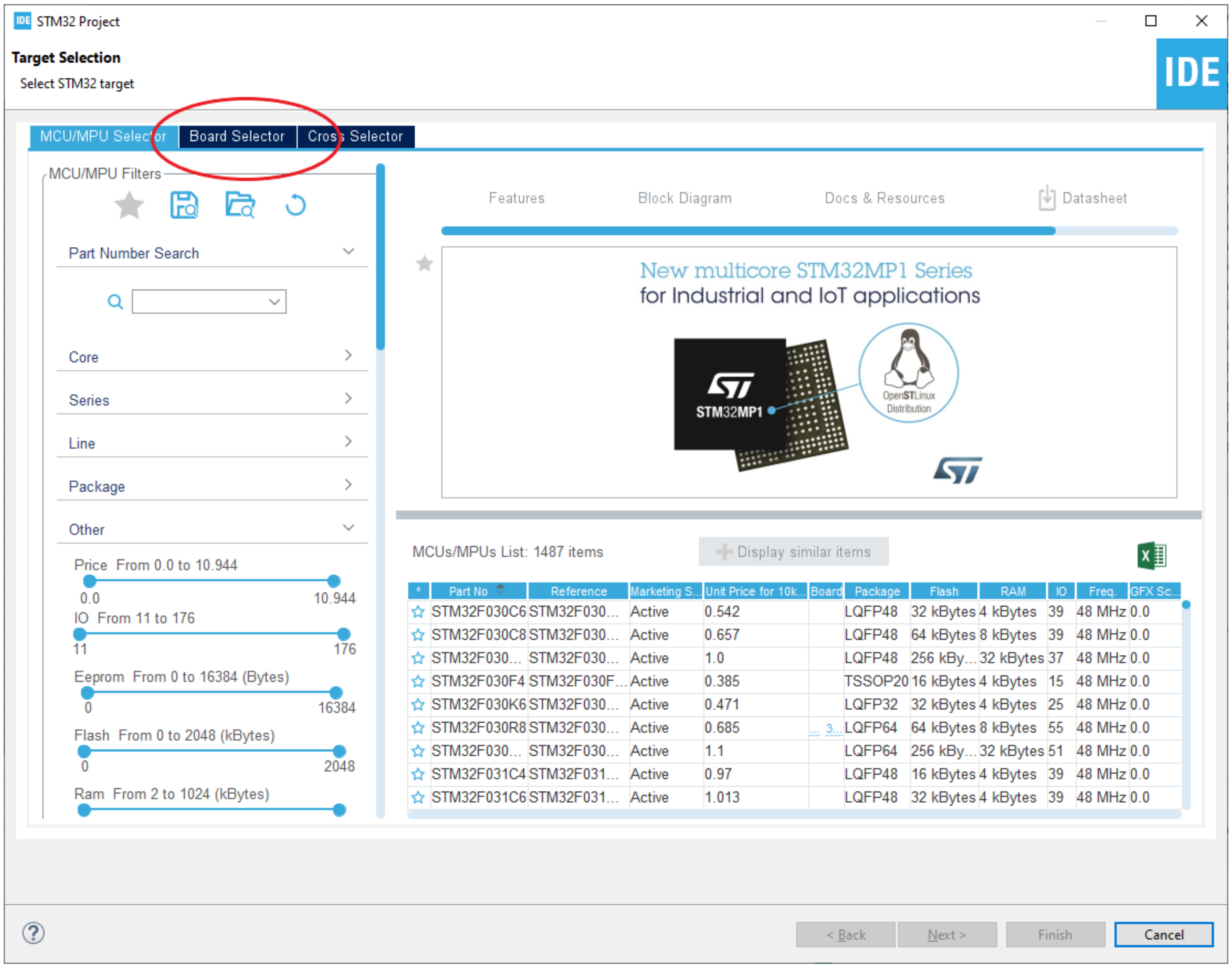
No consoles to display at this time.

FirstProject.elf - /FirstProject/Debug - Aug 17, 2019 5:44:37 PM

Memory Regions | Memory Details

| Region | Start address | End address | Size | Free | Used | Usage (%) |
|---|---|---|---|---|---|---|
| DTCMRAM | 0x20000000 | 0x20020000 | 128 KB | 128 KB | 0 B | 0.00% |
| ITCMRAM | 0x00000000 | 0x00010000 | 64 KB | 64 KB | 0 B | 0.00% |
| RAM_D1 | 0x24000000 | 0x24080000 | 512 KB | 508.84 KB | 3.16 KB | 0.62% |
| RAM_D2 | 0x30000000 | 0x30048000 | 288 KB | 288 KB | 0 B | 0.00% |
| RAM_D3 | 0x38000000 | 0x38010000 | 64 KB | 64 KB | 0 B | 0.00% |

This window will appear



Price From 0.0 to 10.944 (0.0 – 10.944)
IO From 11 to 176 (11 – 176)
Eeprom From 0 to 16384 (Bytes) (0 – 16384)
Flash From 0 to 2048 (kBytes) (0 – 2048)
Ram From 2 to 1024 (kBytes)

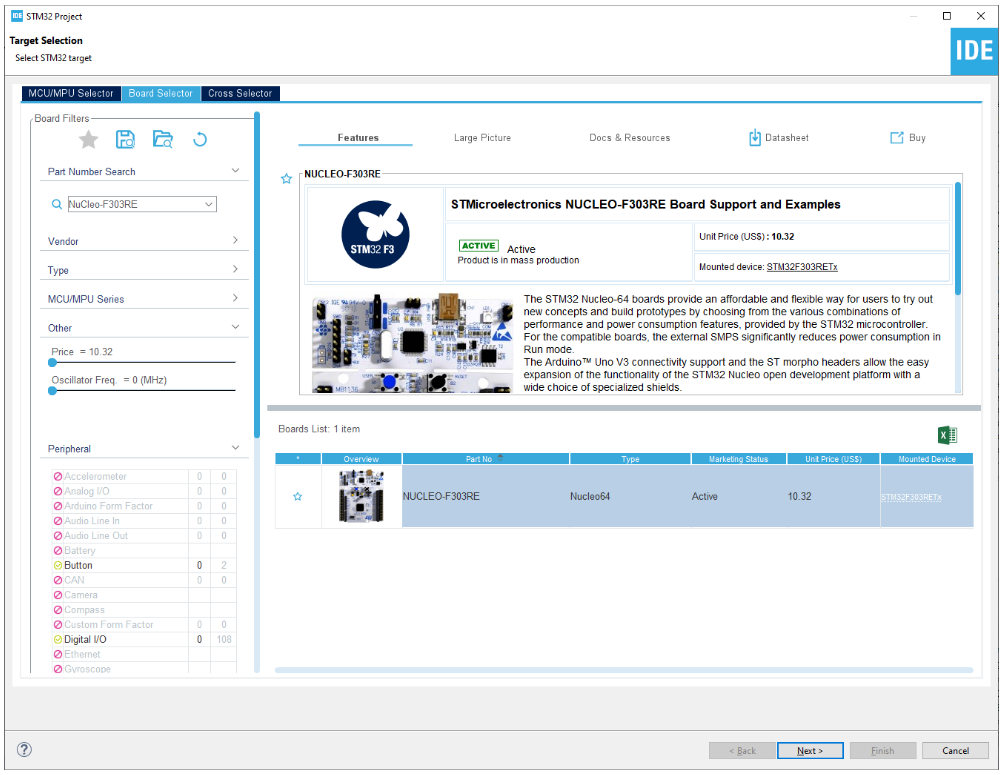
MCUs/MPUs List: 1487 items

| * | Part No | Reference | Marketing S... | Unit Price for 10k... | Board | Package | Flash | RAM | IO | Freq. | GFX Sc... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ☆ | STM32F030C6 | STM32F030... | Active | 0.542 | | LQFP48 | 32 kBytes | 4 kBytes | 39 | 48 MHz | 0.0 |
| ☆ | STM32F030C8 | STM32F030... | Active | 0.657 | | LQFP48 | 64 kBytes | 8 kBytes | 39 | 48 MHz | 0.0 |
| ☆ | STM32F030... | STM32F030... | Active | 1.0 | | LQFP48 | 256 kBy... | 32 kBytes | 37 | 48 MHz | 0.0 |
| ☆ | STM32F030F4 | STM32F030F... | Active | 0.385 | | TSSOP20 | 16 kBytes | 4 kBytes | 15 | 48 MHz | 0.0 |
| ☆ | STM32F030K6 | STM32F030... | Active | 0.471 | | LQFP32 | 32 kBytes | 4 kBytes | 25 | 48 MHz | 0.0 |
| ☆ | STM32F030R8 | STM32F030... | Active | 0.685 | ... 3... | LQFP64 | 64 kBytes | 8 kBytes | 55 | 48 MHz | 0.0 |
| ☆ | STM32F030... | STM32F030... | Active | 1.1 | | LQFP64 | 256 kBy... | 32 kBytes | 51 | 48 MHz | 0.0 |
| ☆ | STM32F031C4 | STM32F031... | Active | 0.97 | | LQFP48 | 16 kBytes | 4 kBytes | 39 | 48 MHz | 0.0 |
| ☆ | STM32F031C6 | STM32F031... | Active | 1.013 | | LQFP48 | 32 kBytes | 4 kBytes | 39 | 48 MHz | 0.0 |

Choose “Board Selector”

3 In “Board Selector” Put part number of the board you are using in the “Part Number Search”

For those of you who has, NUCLEO Board ( The white one) Put in “NUCLEO-F411RE”. Note that if you have a slightly different model, you must put in the correct number as written on the board. For example, if somehow, you have this NUCLEO-F303RE, you must choose that name



Part Number Search: NuCleo-F303RE

Price = 10.32

Oscillator Freq. = 0 (MHz)

| | | |
|---|---|---|
| Accelerometer | 0 | 0 |
| Analog I/O | 0 | 0 |
| Arduino Form Factor | 0 | 0 |
| Audio Line In | 0 | 0 |
| Audio Line Out | 0 | 0 |
| Battery | | |
| Button | 0 | 2 |
| CAN | 0 | 0 |
| Camera | | |
| Compass | | |
| Custom Form Factor | 0 | 0 |
| Digital I/O | 0 | 108 |
| Ethernet | | |
| Gyroscope | | |

NUCLEO-F303RE

STMicroelectronics NUCLEO-F303RE Board Support and Examples

ACTIVE Active
Product is in mass production

Unit Price (US$) : 10.32

Mounted device: STM32F303RETx

The STM32 Nucleo-64 boards provide an affordable and flexible way for users to try out new concepts and build prototypes by choosing from the various combinations of performance and power consumption features, provided by the STM32 microcontroller. For the compatible boards, the external SMPS significantly reduces power consumption in Run mode.
The Arduino™ Uno V3 connectivity support and the ST morpho headers allow the easy expansion of the functionality of the STM32 Nucleo open development platform with a wide choice of specialized shields.

Boards List: 1 item

| * | Overview | Part No | Type | Marketing Status | Unit Price (US$) | Mounted Device |
|---|---|---|---|---|---|---|
| ☆ | | NUCLEO-F303RE | Nucleo64 | Active | 10.32 | STM32F303RETx |

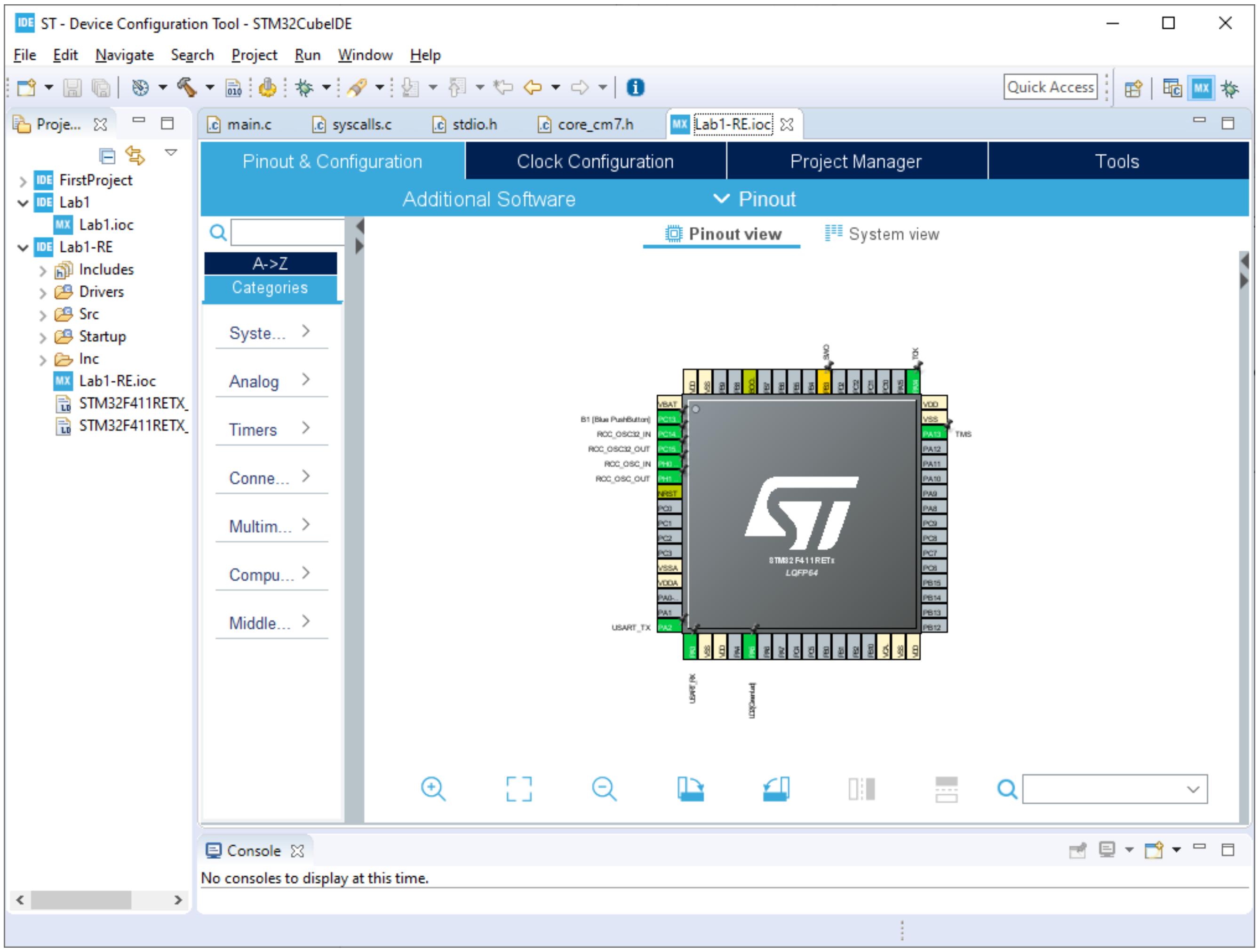
For those of you who do not have the board yet, and use the green board (STM32F4-Discovery) Put in "STM32F4DISCOVERY"

Hit Next and Choose a project name. Let's call it "Lab1"

When it is asking to "initialize all peripherals with their default mode? " choose yes. It may ask about perspective change. Please choose yes as well.

If you have not setup the software before, this process will take a while to download and install.

4. Now you will see the following page.



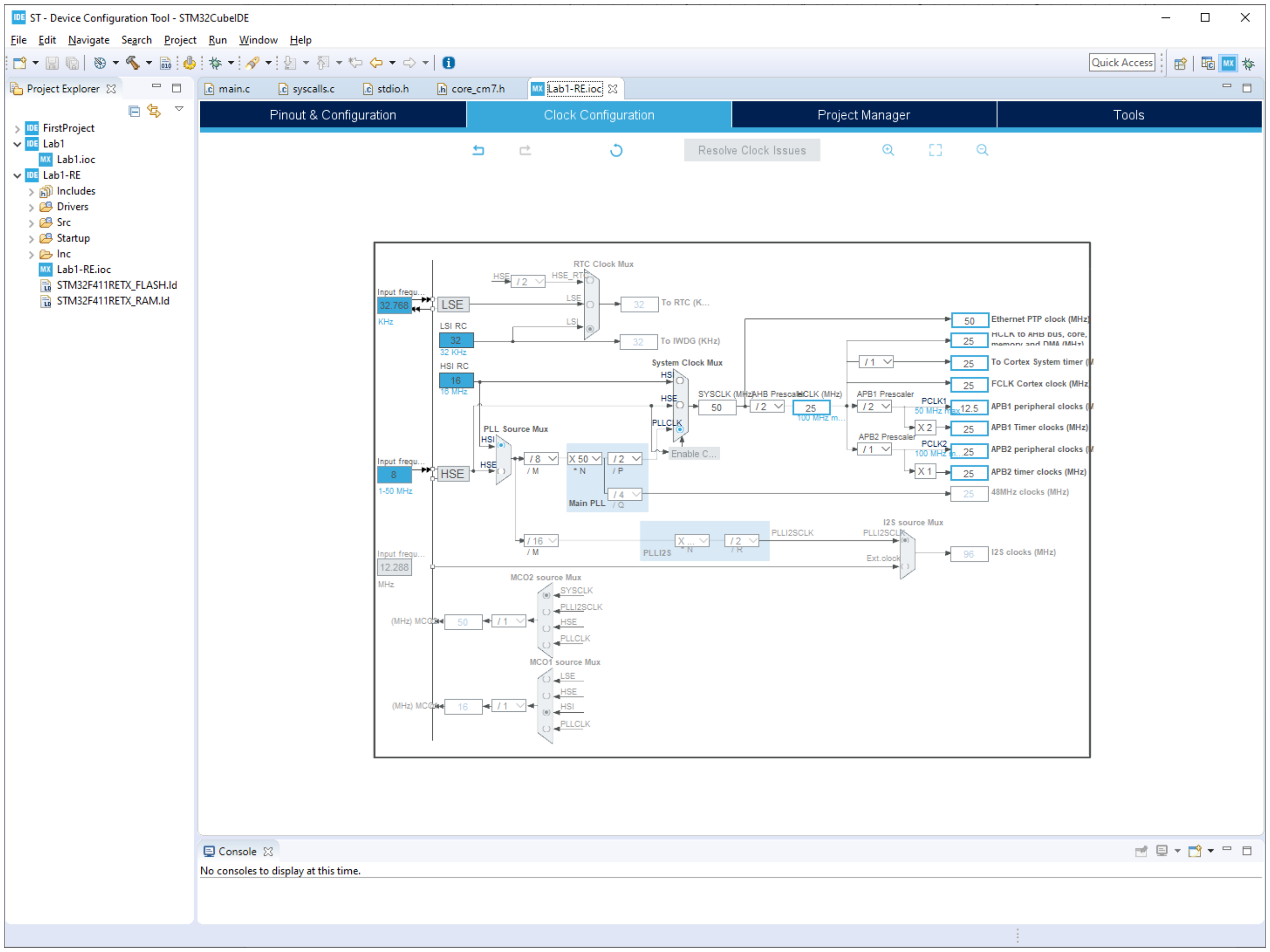
No consoles to display at this time.

4 On Pinout Tab, you can choose mode of the PINs you would like to set. For this Lab, we will be trying to set LED light to blink. If you have STM32F4-Discovery Board, LEDs indicators is connected to PINs PD12-PD15 has been selected (Green highlighted).

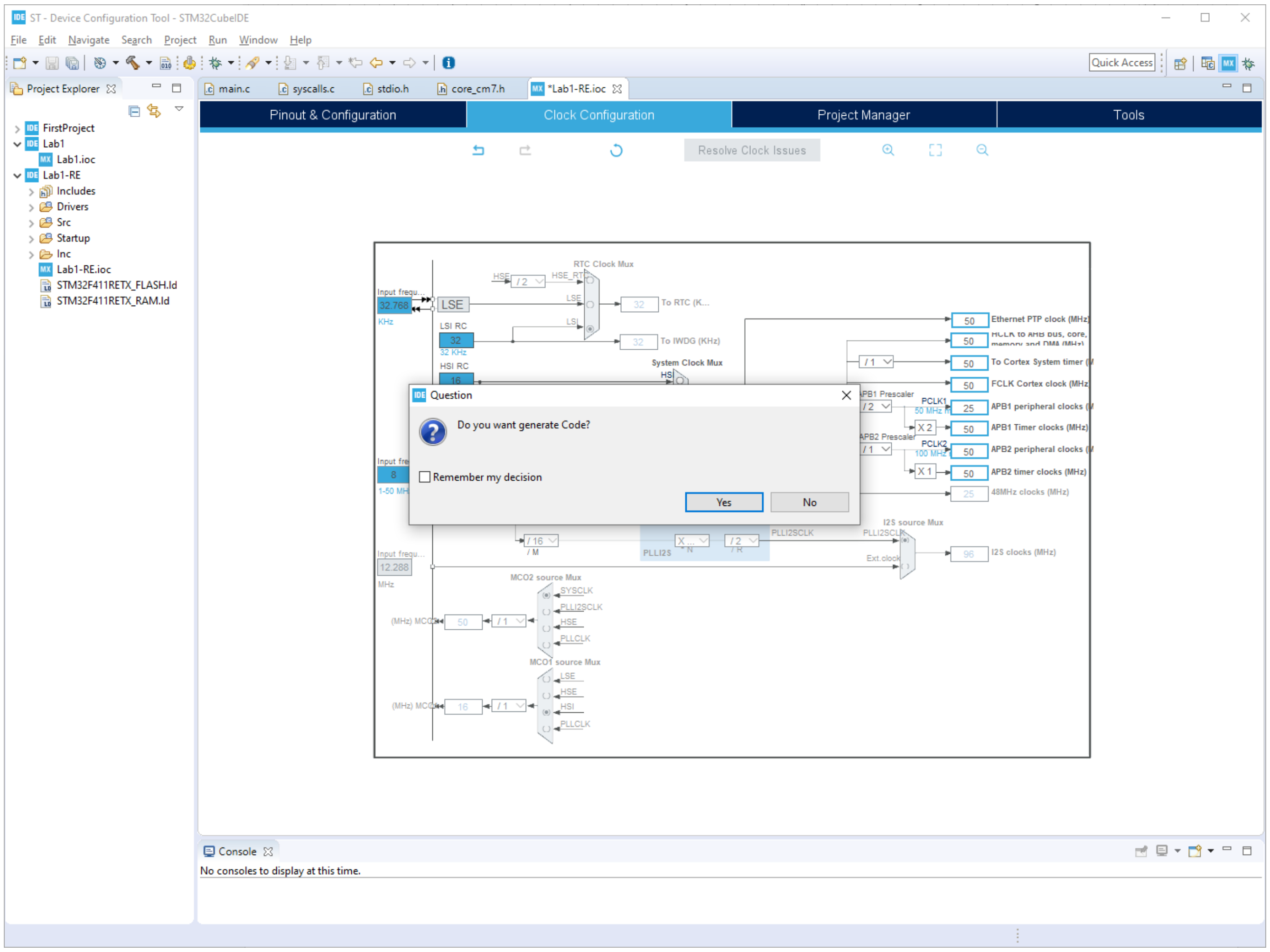
If you have Nucleo-Board, LED is connected to either PA5 or PB13. (Most likely PA5)

If you click at the name of the PIN, you can set the PIN to be different types. For the purpose of driven LED, it is a GPIO_Output. If we are to set it for input, then we can use GPIO_Input.

5 Microcontroller is typically not just one system, but many sub-systems combined together. Each of sub-systems may operate at a different clock speed. For your purpose, the main speed of the CPU is HCLK. Please try to set HCLK to 25



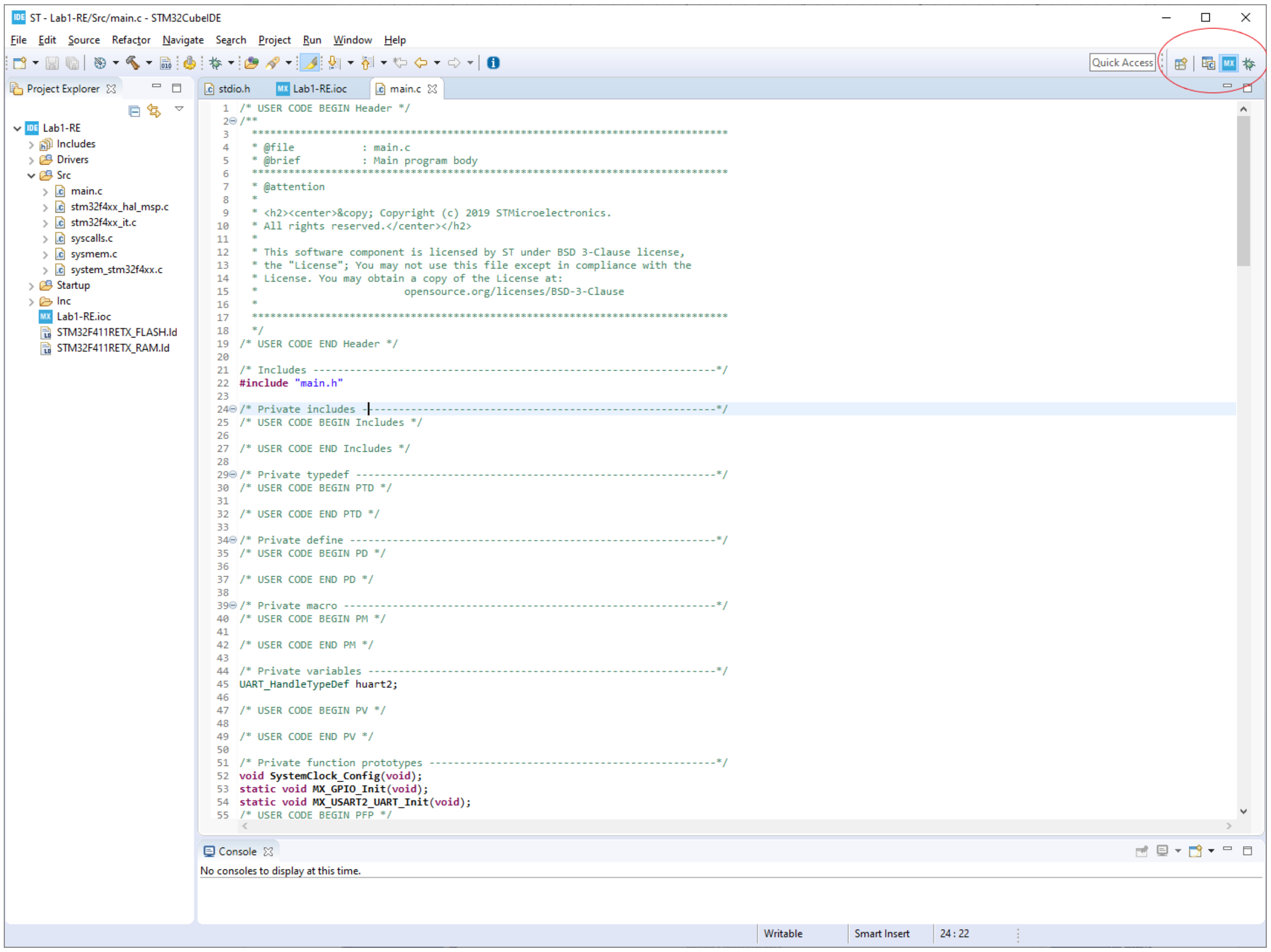
6 Click File -> Save, it will ask you the following question:



Choose Yes

7 Now, the program will generate new code.

In the left Windows pane, you can see a list of file in "Project Explorer." If you do not have "Project Explorer" You may try changing the project perspective by change using the icon on the top right corner. Expand the **project name-> Src**, then double click **main.c**



```c
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2019 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under BSD 3-Clause license,
  * the "License"; You may not use this file except in compliance with the
  * License. You may obtain a copy of the License at:
  *                        opensource.org/licenses/BSD-3-Clause
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.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 ---------------------------------------------------------*/
UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */

/* USER CODE END PV */

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

```
No consoles to display at this time.
```

This code is generated from the tool you were using in the previous section

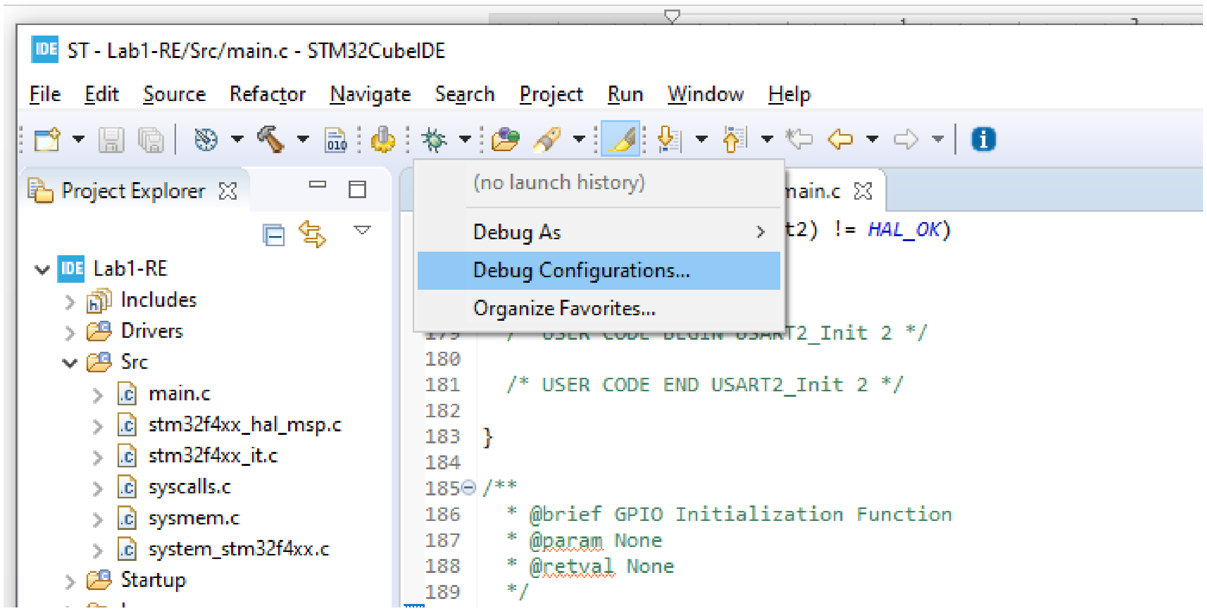
Scroll down and you will see that there are

```c
/* USER CODE BEGIN 1 */

/* USER CODE END 1 */
```

This tag is important for the code generator. If you want to be able to use the code generator to change any setting later on, all your code **must be** in between the comment of USER CODE BEGIN …… END

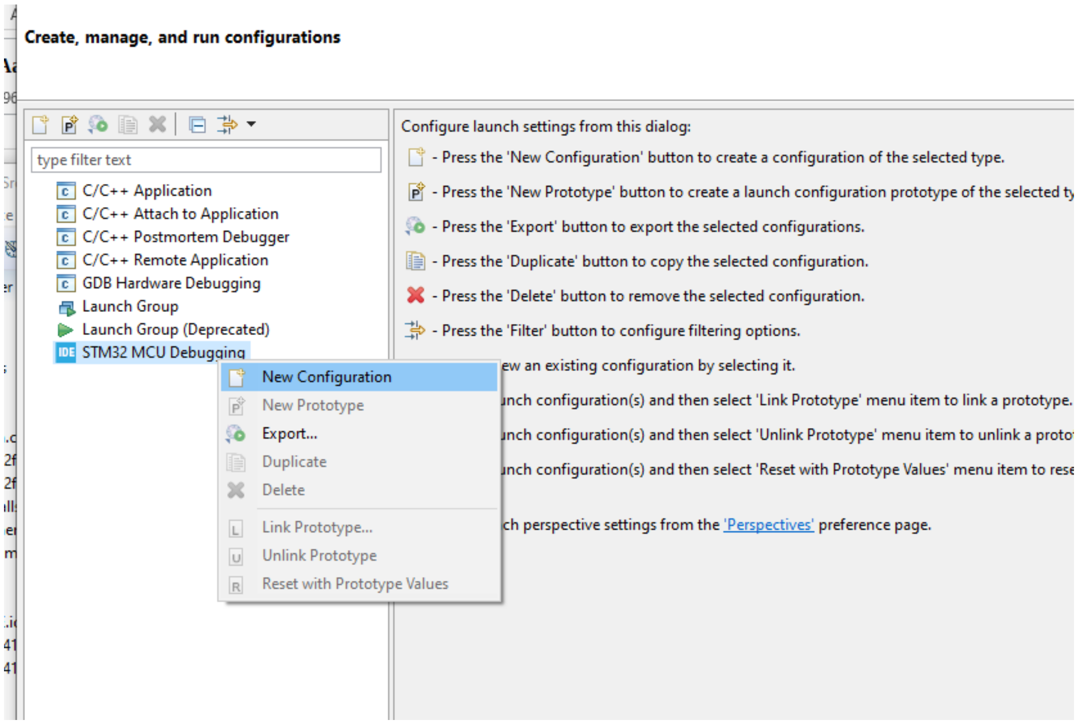
8 To compile the program, choose Project -> Build all

9 To upload the program to STM32 and start debugging,



```
t2) != HAL_OK)

  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
```

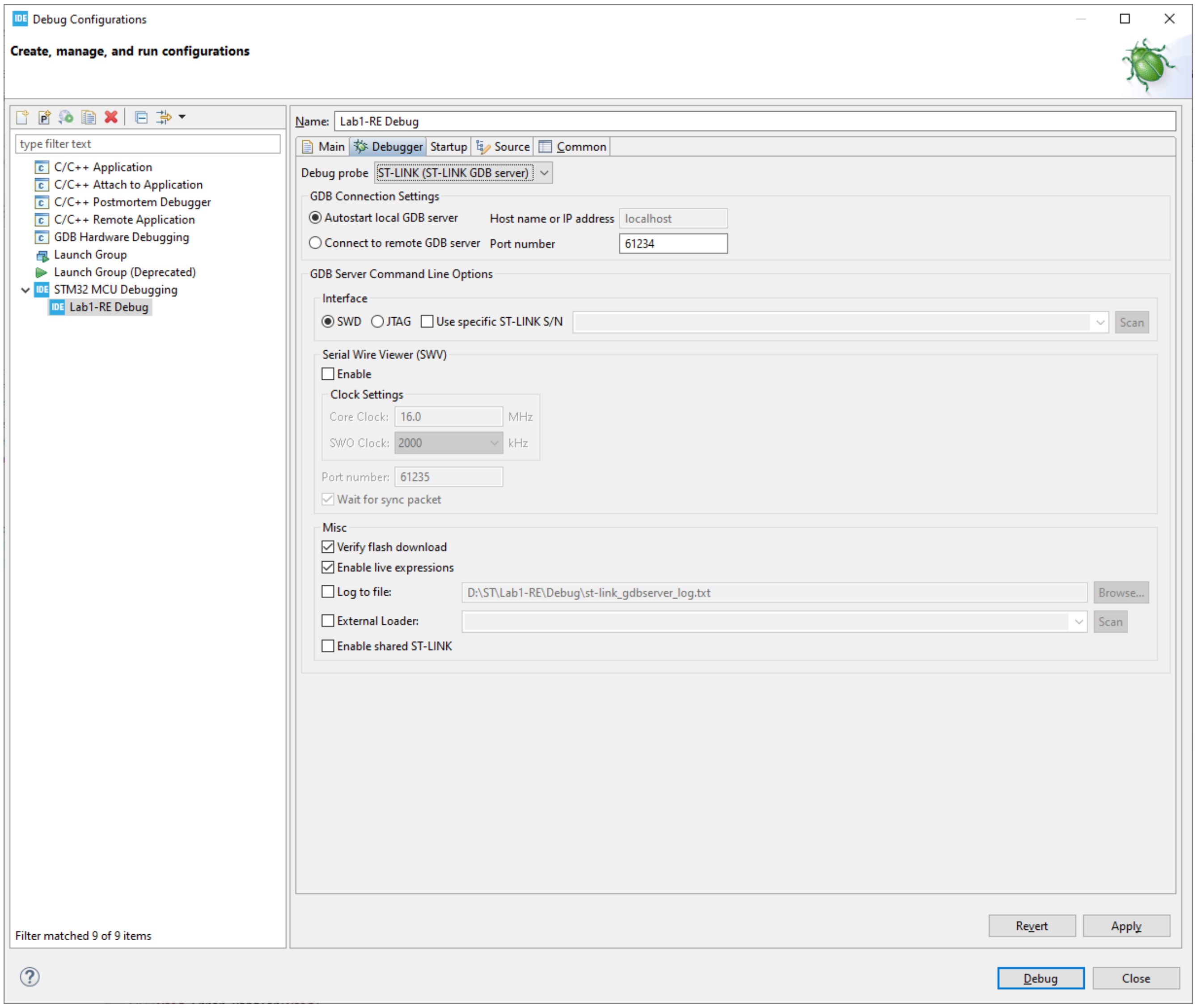
Then right click at "STM32 MCU Debugging"


Create, manage, and run configurations

Configure launch settings from this dialog:
- Press the 'New Configuration' button to create a configuration of the selected type.
- Press the 'New Prototype' button to create a launch configuration prototype of the selected ty
- Press the 'Export' button to export the selected configurations.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
ew an existing configuration by selecting it.
nch configuration(s) and then select 'Link Prototype' menu item to link a prototype.
nch configuration(s) and then select 'Unlink Prototype' menu item to unlink a proto
nch configuration(s) and then select 'Reset with Prototype Values' menu item to rese
ch perspective settings from the 'Perspectives' preference page.

In this page, choose "Search Project", then Hit OK.

Now, switch to "debugger plane"


Debug Configurations

Create, manage, and run configurations

Name: Lab1-RE Debug

Debug probe: ST-LINK (ST-LINK GDB server)

GDB Connection Settings
- (●) Autostart local GDB server — Host name or IP address: localhost
- ( ) Connect to remote GDB server — Port number: 61234

GDB Server Command Line Options

Interface: (●) SWD ( ) JTAG [ ] Use specific ST-LINK S/N

Serial Wire Viewer (SWV)
- [ ] Enable
- Clock Settings: Core Clock: 16.0 MHz; SWO Clock: 2000 kHz
- Port number: 61235
- [✓] Wait for sync packet

Misc
- [✓] Verify flash download
- [✓] Enable live expressions
- [ ] Log to file: D:\ST\Lab1-RE\Debug\st-link_gdbserver_log.txt
- [ ] External Loader:
- [ ] Enable shared ST-LINK

If you are using the board STM32F4Discovery from the lab, they are likely to be pre-install debugger as J-Link, if not, it would be ST-Link. You can change the debugger probe between ST-Link and J-Link here.

You can then hit “Debug”

## *Lab Exercises*

1 Create a new STM32 Project by following the Getting Started guide, and insert new main function code as below.

```
int main(void)
{

  /* USER CODE BEGIN 1 */
    uint32_t i,j;
  /* USER CODE END 1 */

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

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();

  /* USER CODE BEGIN 2 */
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
  /* USER CODE END WHILE */
            int j;
            GPIOA->ODR ^= 1<<5;
            for(j=0;j<2000000;j++);



  /* USER CODE BEGIN 3 */

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

2 Using debugging feature, set Breakpoint at “for loop”, then debug on STM32F4 Discovery Board.

2.1 From STM32F4XX Reference manual, what are these GPIO registers

2.1.1 GPIOD_MODER

2.1.2 GPIOD_OTYPER

2.1.3 GPIOD_OSPEEDR

2.1.4 GPOD_PUPDR

2.1.5 GPIOD_ODR

2.2 Using step/suspend/resume debugging feature on SW4STM32, what is the value of GPIOD_MODER and GPIOD_ODR from the start of debugging and breakpoint. What are the relations to LEDs' Discovery Board.

3 Create a new project with 4 times speed of System Clock using "Clock Configuration" on STM32CubeMX. What are the value of PLLP, PLLN and PLLM register, before and after set the new speed. (Look at RCC register on STM32F407 Reference manual)

4 Use a STM32 Cube embedded software libraries instead of direct register assignation and for loop delay (using functions from stm32f4xx_hal.c and stm32f4xx_hal_gpio.c in STM32F4xx_HAL_Driver) to create a same behavior as example code on 1.

# Laboratory 2: Simple digital I/O with GPIO and serial communication with USART

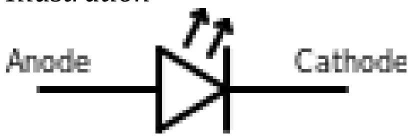
## *Objectives*

1. Understand the concept of Serial Communication
2. Understand the concept of data format
3. Understand General-purpose I/Os (GPIO) and its implementation as peripherals on STM32
4. Understand Universal synchronous asynchronous receiver transmitter (USART) and its implementation as peripherals on STM32
5. Understand the concept of Light Emitted Diode

## *LED*

LED is short for ***Light Emitting Diode***. It is a special kind of diodes (unipolar electronic devices) that will emit photon (light) where there is an electronic current flow in a correct direction (from anode to cathode).  Illustration and  Illustration show the symbol and the picture of an LED.

*Illustration*



*1: Electronic Symbol of LED*

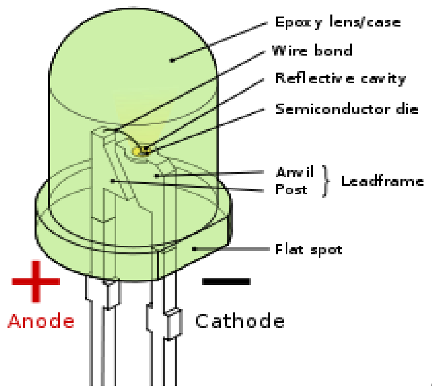


*Illustration 2: A picture of LED (a picture from wikipedia.org)*

Due to its characteristics, LED is an ideal choice for using as an output of a digital pin. Using LED as a digital output, we can easily observe the logic from the light. There are several ways to connect the LED to an output PIN. Here are two simple methods presented (see  Illustration). Depending on the electronic properties of the device, the quality may vary.  For most cases, LED1 should provide better quality (brighter).

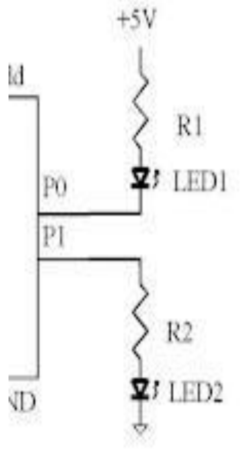


*Illustration 3: Two methods for interfacing LED*

In particular, the LED1 (connected to P0) will emit the light when the output of P0 is LOW. The LED 2 will emit the light when the output of P1 is HIGH. To determine the correct logic, a programmer must understand the schematic of the circuits.

## *Push-button switch*

Push-button switch is, perhaps, the most commonly used input for logic circuits. The idea is to toggle the voltages at the input pin. In Illustration, a method for connecting the switch to a pin is showed. With the pull-up resistor, the micro controller will see the HIGH logic when reading from the input pin. Once a button is pressed, the current will sink to the ground giving the LOW logic to the input pin.

The ease understanding, thinking of the voltage as a level of water. When there is no sink (hole), the water level would be high. However, when we open the sink (hole at the bottom), the water level would go down (low). The same idea can also be used to explain why a pull-up resistor is required. Without the pull-up resistor, it is difficult (if not impossible) to determine the correct level. The pull-up resistor make sure that we will always fill the water to the high level (if there is no sink to drain the water).

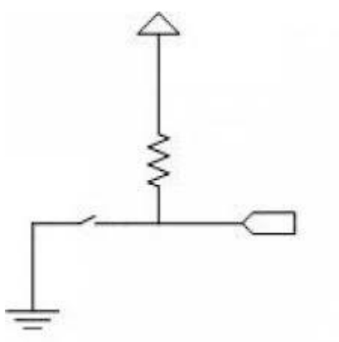


*Illustration 4: Symbol of push-button switch with pull-up resistor*

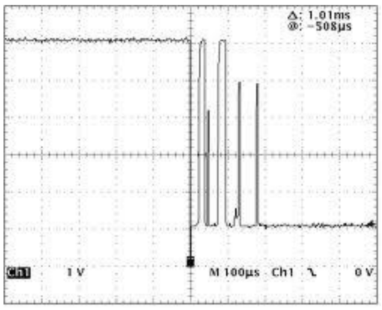
For some microcontroller (including our Avr/Arduino), the pull-up resistor is provided internally in the chip. This allows us to avoid connecting more resistor to the circuits.

## *Debounce*

Given that a switch is a mechanical device, a mechanical contact between the two surfaces would cause a bounce. (The same idea when dropping a ball to the floor. The ball may bounce few times before stopping at the floor.) The push-button switch also cause a bounce when it is being pressed. From an oscilloscope, the bounce is observed as Illustration. If a program does not carefully avoid the bounce, we may easily observe multiple clicks from only one physical click.

A way to avoid reading multiple clicks from a single click is called ***debounce***. Depending on the size and the mechanical properties of the switch, the bounce may last several milliseconds. There are several ways to avoid bounce. Some methods require extra hardware. However, the naïve software solution is to delay few milliseconds before reading the next input.

Illustration



5: bouncing switch

## *Serial Communication / U(S)ART*

Serial communication is a method for using a sequence of bits to transfer a byte of data. This is in contrast to parallel communication, where several bits are sent as a whole, on a link with several parallel channels as Illustration.

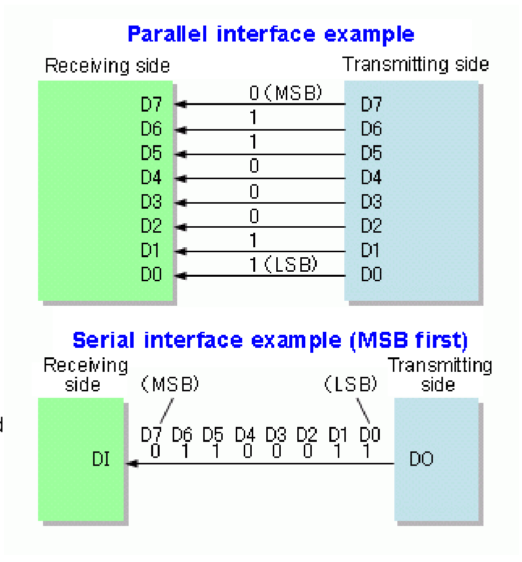


*Illustration 6 Parallel versus serial communication.*

For universal asynchronous receiver/transmitter (UART) – Universal Synchronous/Asynchronous Receiver/Transmitter (USART) if also supports synchronous operation, we use 8 bit to represent a byte of data. A byte is generally equal to a character. Thus, there are (theoretically) 256 possible characters in a byte. One standard that provides association between a value and a physical character is ASCII (see next section for more details).

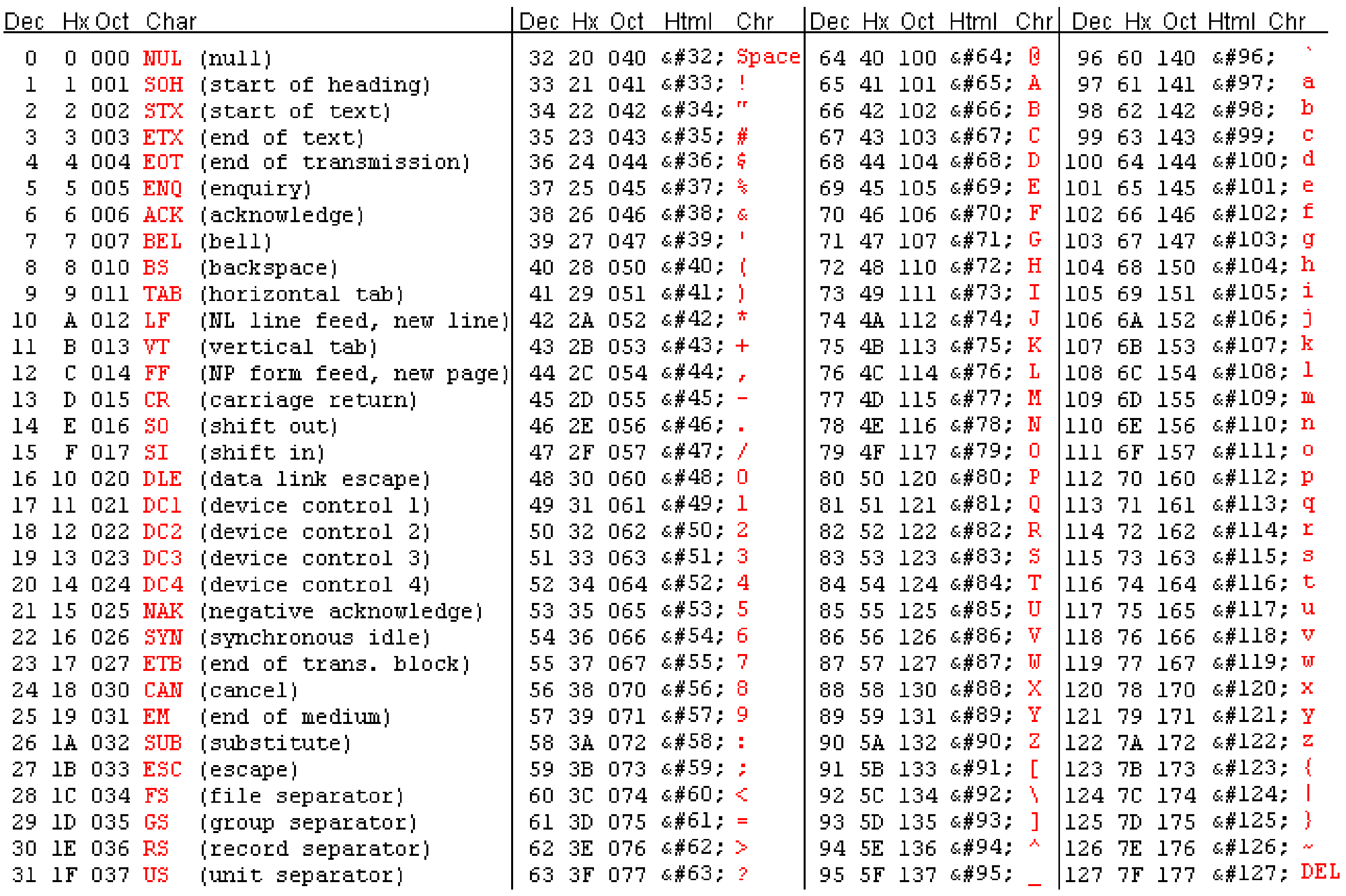
With a serial communication, a communication between two devices can be done with just two wires (one fore sending and one for receiving). This kind of communication is cheaper comparing to parallel communication where 8 wires are need to transmit a byte of data. Thus, to a byte of data from one device to another, we have to send (at least) 8 times (one for each bit). Given that no clock or control signal is involved in the communication, the key critical to the success of such communication is timing. Both devices has to be configured for the same speed of clock (specified as bit per second or bps) and the same protocol in order to observe the right data at the right time. This is the basic of network communication that we used today.

| **Bit number** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Start bit** | **5–9 data bits** | | | | | | | | | **Stop bit(s)** | |
| | Start | Data 0 | Data 1 | Data 2 | Data 3 | Data 4 | Data 5 | Data 6 | Data 7 | Data 8 | Stop | |

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter

## *ASCII*

ASCII is short for the American Standard Code for Information Interchange. It is a character-encoding scheme based on English alphabet, numbers and punctuations. It is nowadays used by several digital devices as a way to representing code for characters. Thus, when type a character 'A', the digital computer system simply record an ASCII code to the memory.



| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 | &#32; | Space | 64 | 40 | 100 | &#64; | @ | 96 | 60 | 140 | &#96; | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | &#33; | ! | 65 | 41 | 101 | &#65; | A | 97 | 61 | 141 | &#97; | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | &#34; | " | 66 | 42 | 102 | &#66; | B | 98 | 62 | 142 | &#98; | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | &#35; | # | 67 | 43 | 103 | &#67; | C | 99 | 63 | 143 | &#99; | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | &#36; | $ | 68 | 44 | 104 | &#68; | D | 100 | 64 | 144 | &#100; | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | &#37; | % | 69 | 45 | 105 | &#69; | E | 101 | 65 | 145 | &#101; | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | &#38; | & | 70 | 46 | 106 | &#70; | F | 102 | 66 | 146 | &#102; | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | &#39; | ' | 71 | 47 | 107 | &#71; | G | 103 | 67 | 147 | &#103; | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | &#40; | ( | 72 | 48 | 110 | &#72; | H | 104 | 68 | 150 | &#104; | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | &#41; | ) | 73 | 49 | 111 | &#73; | I | 105 | 69 | 151 | &#105; | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | &#42; | * | 74 | 4A | 112 | &#74; | J | 106 | 6A | 152 | &#106; | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | &#43; | + | 75 | 4B | 113 | &#75; | K | 107 | 6B | 153 | &#107; | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | &#44; | , | 76 | 4C | 114 | &#76; | L | 108 | 6C | 154 | &#108; | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | &#45; | - | 77 | 4D | 115 | &#77; | M | 109 | 6D | 155 | &#109; | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | &#46; | . | 78 | 4E | 116 | &#78; | N | 110 | 6E | 156 | &#110; | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | &#47; | / | 79 | 4F | 117 | &#79; | O | 111 | 6F | 157 | &#111; | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | &#48; | 0 | 80 | 50 | 120 | &#80; | P | 112 | 70 | 160 | &#112; | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | &#49; | 1 | 81 | 51 | 121 | &#81; | Q | 113 | 71 | 161 | &#113; | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | &#50; | 2 | 82 | 52 | 122 | &#82; | R | 114 | 72 | 162 | &#114; | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | &#51; | 3 | 83 | 53 | 123 | &#83; | S | 115 | 73 | 163 | &#115; | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | &#52; | 4 | 84 | 54 | 124 | &#84; | T | 116 | 74 | 164 | &#116; | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | &#53; | 5 | 85 | 55 | 125 | &#85; | U | 117 | 75 | 165 | &#117; | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | &#54; | 6 | 86 | 56 | 126 | &#86; | V | 118 | 76 | 166 | &#118; | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | &#55; | 7 | 87 | 57 | 127 | &#87; | W | 119 | 77 | 167 | &#119; | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | &#56; | 8 | 88 | 58 | 130 | &#88; | X | 120 | 78 | 170 | &#120; | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | &#57; | 9 | 89 | 59 | 131 | &#89; | Y | 121 | 79 | 171 | &#121; | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | &#58; | : | 90 | 5A | 132 | &#90; | Z | 122 | 7A | 172 | &#122; | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | &#59; | ; | 91 | 5B | 133 | &#91; | [ | 123 | 7B | 173 | &#123; | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | &#60; | < | 92 | 5C | 134 | &#92; | \ | 124 | 7C | 174 | &#124; | \| |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | &#61; | = | 93 | 5D | 135 | &#93; | ] | 125 | 7D | 175 | &#125; | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | &#62; | > | 94 | 5E | 136 | &#94; | ^ | 126 | 7E | 176 | &#126; | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | &#63; | ? | 95 | 5F | 137 | &#95; | _ | 127 | 7F | 177 | &#127; | DEL |

http://www.infocellar.com/binary/ascii-ebcdic.htm

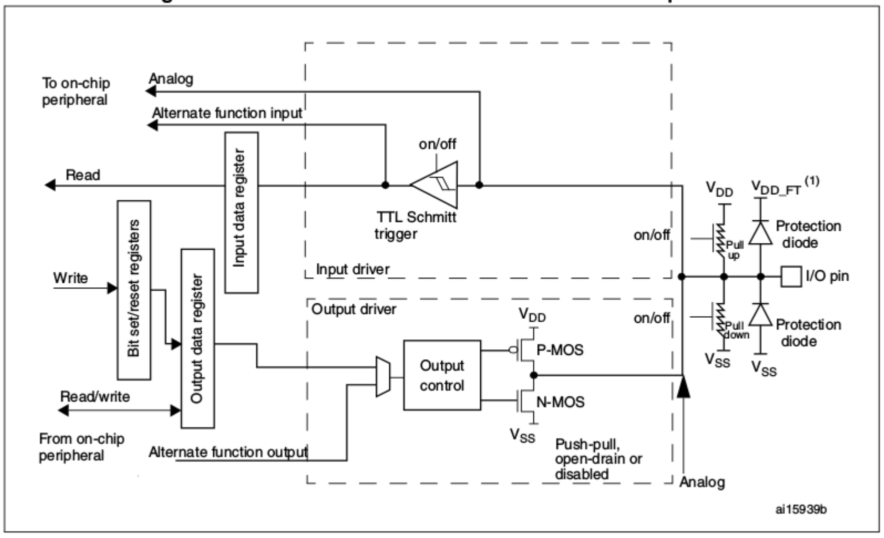
## *STM32's GPIO functional description*

Subject to the specific hardware characteristics of each I/O port listed in the datasheet, each port bit of the general-purpose I/O (GPIO) ports can be individually configured by software in several modes:

- Input floating
- Input pull-up
- Input-pull-down
- Analog
- Output open-drain with pull-up or pull-down capability
- Output push-pull with pull-up or pull-down capability
- Alternate function push-pull with pull-up or pull-down capability
- Alternate function open-drain with pull-up or pull-down capability

Each I/O port bit is freely programmable, however the I/O port registers have to be accessed as 32-bit words, half-words or bytes. The purpose of the GPIOx_BSRR register is to allow atomic read/modify accesses to any of the GPIO registers. In this way, there is no risk of an IRQ occurring between the read and the modify access.

**Figure 25. Basic structure of a five-volt tolerant I/O port bit**



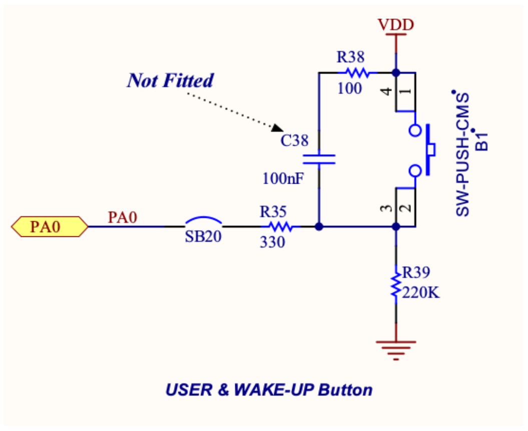
1. $V_{DD\_FT}$ is a potential specific to five-volt tolerant I/Os and different from $V_{DD}$.

Source : **Reference Manuals – RM0090**: STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM®-based 32-bit MCUs

## *NUCLEO-F411 Basic Input/Output Interface*

### LEDs

- User LD2: Green LED is a user LED connected to the I/O ***PA5*** of the STM32F411



### Pushbuttons

- B1 USER: User and Wake-Up buttons are connected to the I/O PC13 of the STM32F411

Source : **User Manuals – UM1742:** STM32 Nucleo-64 boards User Manual

## *NUCLEO-F411 UART Communication*

The USART2 interface available on PA2 and PA3 of the STM32 microcontroller. You do not need an external USB-Serial adapter in order to communicate between a computer and STM32

Source: **User Manuals – UM1742:** STM32 Nucleo-64 boards User Manual

## *Lab Exercises*

1 Create a new STM32 Project to blink (on/off) with a period of 0.2 sec. for an on board LED

2 Create a new STM32 Project to blink (on/off) with a period of 0.2 sec. for an external LED

3 Create a new STM32 Project to toggle an LED (on/off) with pushing USER push-button. (Debouching is required)

4 Create a new STM32 Project to echo back (transmit the receive data) the communication data from UART peripheral (USART2) interface using blocking mode APIs on STM32CUBE library (Look at stm32f4xx_hal_uart.c).

  For those of you who are using STM32F4Discovery, you will have to connect to a USB-Serial such as ET-MINI USB-TTL as mention above.

5 Create a new STM32 Project to toggle an LED status (on/off) with commands via serial console. (Type "on" or "off" then press Enter to on or off the LED

Optional Lab:

1. Pair up with your friend, and make one board as a master and another as a slave. The master send a command through UART to the slave board to toggle the LED when a USER push-button.

# Laboratory 3: Interrupt and Timer

## *Objectives*

1. Understand the concept of Interrupt

2. Understand the concept of Timer

## *Interrupt*

### Device Services: Polling Versus Interrupts

The CPU in a microcontroller or microprocessor is a sequential machine. The control unit sequentially fetches, decodes, and executes instructions from the program memory according to the stored program. This implies that a single CPU can only be executing one instruction at any given time. When it comes to a CPU servicing its peripherals, the program execution needs to be taken to the specific set of instructions that perform the task associated to servicing each device. Take for example the case of serving a keypad.

A keypad can be visualized at its most basic level as an array of switches, one per key, where software gives meaning to each key. The switches are organized in a way that each key depressed yields a different code. For every keystroke, the CPU needs to retrieve the associated key code. The action of retrieving the code of each depressed key and passing them to a program for its interpretation, is what we call servicing the keypad. Like the keypad in this example, the CPU might serve many other peripherals, like a display by passing it the characters or data to be displayed, or a communication channel by receiving or sending characters that make up a message, etc.

source: http://racegrade.com/keypad.html

When it comes to the CPU serving a device, one of two different approaches can be followed: service by polling or service by interrupts. Let's look at each approach with some detail.

**Service by Polling**

In service by polling, the CPU continuously interrogates or polls the status of the device to determine if service is needed. When the service conditions are identified, the device is served. This action can be exemplified with a hypothetical case of real life where you will act like a polling CPU:

Assume you are given the task of answering a phone to take messages. You don't know when calls will arrive, and you are given a phone that has no ringer, no vibrator, or any other means of knowing that a call has arrived. Your only choice for not missing a single call is by periodically picking-up the phone, placing it to your ear and asking "Hello! Anybody there?"

hoping someone will be in the other side of the line. This would be quite an annoying job, particularly if you had other things to do. However, if you don't want to miss a single call you'll have to put everything else aside and devote yourself to continuously perform the polling sequence: pick-up the phone, bring it to your ear, and hope someone is on the line. Since the line must be available for calls to enter (sorry, no call-waiting service), you have to hang-up and repeat the sequence over and over to catch every incoming call and taking the messages. What a waste of time! Well, that is polling.

### Service by Interrupts

When a peripheral is served by interrupts, it sends a signal to the CPU notifying of its need. This signal is called an interrupt request or IRQ. The CPU might be busy performing other tasks, or even in sleep mode if there were no other tasks to perform, and when the interrupt request arrives, if enabled, the CPU suspends the task it might be performing to execute the specific set of instructions needed to serve the device. This event is what we call an interrupt. The set of instructions executed to serve the device upon an IRQ form what is called the interrupt service routine or ISR.

We can bring this interrupting capability to our phone example above.

Let's assume that in this case your phone has a ringer. While expecting to receive incoming calls, now you can rely on the ringer to let you know that a call has arrived. In the mean time, while you wait for calls to arrive you are free to perform other tasks. You could even get a nap if there were nothing else to do. When a call arrives, the ringer sounds and you suspend whatever task you are doing to pick-up the phone, now with the certainty that a caller is in the other end of the line to take his or her message. The ring sound acts like an interrupt request to you. A much more efficient way to take the messages.

Interrupts can be used to serve different tasks. The following are just a few simple examples that illustrate the concept:

- A system that toggles an LED when a push-button is depressed. The push-button interface can be configured to trigger an interrupt request to the CPU when the push-button is depressed, having an associated ISR that executes the code that turns the LED on or off.
- A message arriving at a communication port can have the port interface configured to trigger an interrupt request to the CPU so that the ISR executes the program that receives, stores, and/or interprets the message.
- A voltage monitor in a battery operated system might be interfaced to trigger an interrupt when a low-voltage condition is detected. The ISR might be programmed to save the system state and shut it down or to warn the user about the situation.

Source: Introduction to Embedded Systems Using Microcontrollers and the MSP430 , Jiménez, Manuel, Palomera, Rogelio, Couvertier, Isidoro

## Nested vectored interrupt controller (NVIC)

### NVIC features

The nested vector interrupt controller NVIC includes the following features:

- 82 maskable interrupt channels for STM32F405xx/07xx and STM32F415xx/17xx, and up to 91 maskable interrupt channels for STM32F42xxx and STM32F43xxx (not including the 16 interrupt lines of Cortex ® -M4 with FPU)
- 16 programmable priority levels (4 bits of interrupt priority are used)
- low-latency exception and interrupt handling
- power management control
- implementation of system control registers

The NVIC and the processor core interface are closely coupled, which enables low latency interrupt processing and efficient processing of late arriving interrupts. All interrupts including the core exceptions are managed by the NVIC. For more information on exceptions and NVIC programming, refer to programming manual PM0214.

### External interrupt/event controller (EXTI)

The external interrupt/event controller consists of up to 23 edge detectors for generating event/interrupt requests. Each input line can be independently configured to select the type (interrupt or event) and the corresponding trigger event (rising or falling or both). Each line can also masked independently. A pending register maintains the status line of the interrupt requests.

## *Timer*

### Base Timer Structure

In its most basic form, a timer is a counter driven by a periodic clock signal. Its operation can be summarized as follows: Starting from an arbitrary value, typically zero, the counter is incremented every time the clock signal makes a transition. Each clock transition is called a "clock event". The counter keeps track of the number of clock events. Notice that only a single type of transition, either low-to-high or high-to-low (not both), will create an event. If the clock were a periodic signal of known frequency f , then the number of events k in the counter would indicate the time kT elapsed between event 0 and the current event , being T = 1/ f the period of the clock signal. The name "timer" stems from this characteristic.

To enhance the capabilities of a timer, further blocks are typically added to the basic counter. The conglomerate of the counter plus the additional blocks enhancing the structure form the architecture of a particular timer.

Figure 7.11 illustrates the basic structure of a timer unit. Its fundamental component include the following:

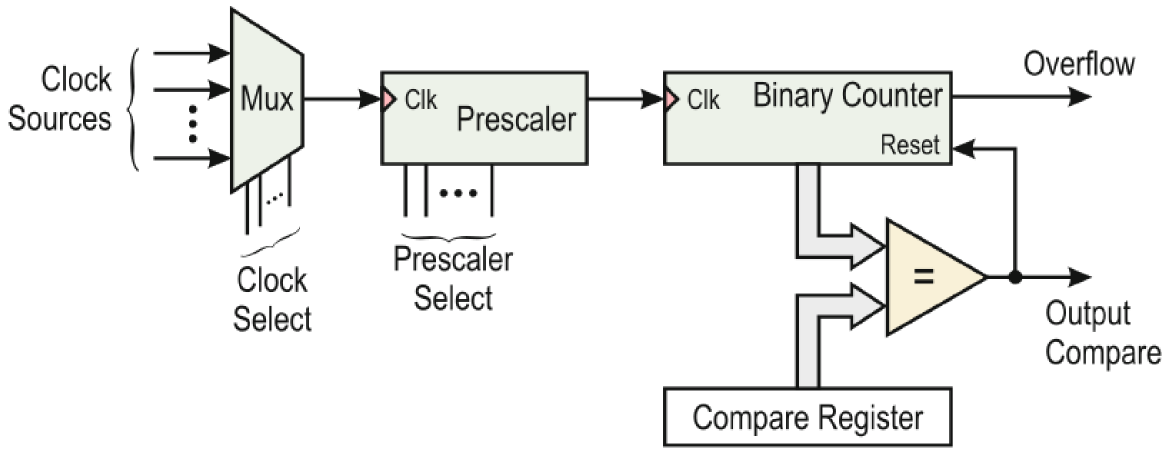


**Fig. 7.11** Components of a base timer

- Some form of a clock selector (Mux) allowing for choosing one from multiple clock sources.
- A prescaler, that allows for pre-dividing the input clock frequency reaching the counter.
- An n-bit binary counter, providing the basic counting ability.
- An n-bit compare register, that allows for limiting the maximum value the counter can reach.
- A hardware comparator that allows for knowing when the binary counter reaches the value stored in the compare register. Note how a match of these values resets the binary counter.

The STM32 devices are built-in with various types of timers, with the following features for each:

- General-purpose timers are used in any application for output compare (timing and delay generation), one-pulse mode, input capture (for external signal frequency measurement), sensor interface (encoder, hall sensor)...
- Advanced timers : these timers have the most features. In addition to general purpose functions, they include several features related to motor control and digital power conversion applications: three complementary signals with deadtime insertion, emergency shut-down input.
- One or two channel timers : used as general-purpose timers with a limited number of channels.
- One or two channel timers with complementary output : same as previous type, but having a deadtime generator on one channel. This allows having complementary signals with a time base independent from the advanced timers.
- Basic timers have no input/outputs and are used either as timebase timers or for triggering the DAC peripheral.
- Low-power timers are simpler than general purpose timers and their advantage is the ability to continue working in low-power modes and generate a wake-up event.
- High-resolution timers are specialized timer peripherals designed to drive power conversion in lighting and power source applications. It is however also usable in other fields that require very fine timing resolution. AN4885 and AN4449 are practical examples of high-resolution timer use.

## *Lab Exercises*

1 Create a new STM32 Project on System Workbench IDE to blink (on/off) red LED (LD5) with changeable period by pushing USER push-button from periods of 0.2 sec. ⟶ 1 sec. ⟶ 5 sec. and back to 0.2 sec. (using External interrupt/event controller (EXTI) is required)

2 Create a new STM32 Project on System Workbench IDE to generate clock signal from any output pins with 100 microsecond period and use an oscilloscope to monitor the signal.

3 Create a new STM32 Project on System Workbench IDE to blink (on/off) red LED (LD5) with 500 milliseconds period and green LED (LD4) with 490.5 milliseconds period and display the number of both red and green LED blinkings every 999.9 milliseconds on Serial terminal via UART. (using interrupt timer)

## Laboratory 4: Analog-to-digital converter (ADC) and Pulse-width modulation (PWM)

### *Objectives*

1. Understand the concept of Digital-to-analog converter (ADC)
2. Understand the concept of Pulse-width modulation (PWM)

### *Digital-to-analog converter (ADC)*

An analog (aka. Analog signal) is continuous signal. Unlink a digital signal which observed values are either HIGH or LOW, analog signal give an approximate level of value (e.g. 2.3 volts). In real life, everything is analog i.e. temperature (degree Celsius), brightness (Lux), sound (decibel), pressure (pascal), and speed (km/s)

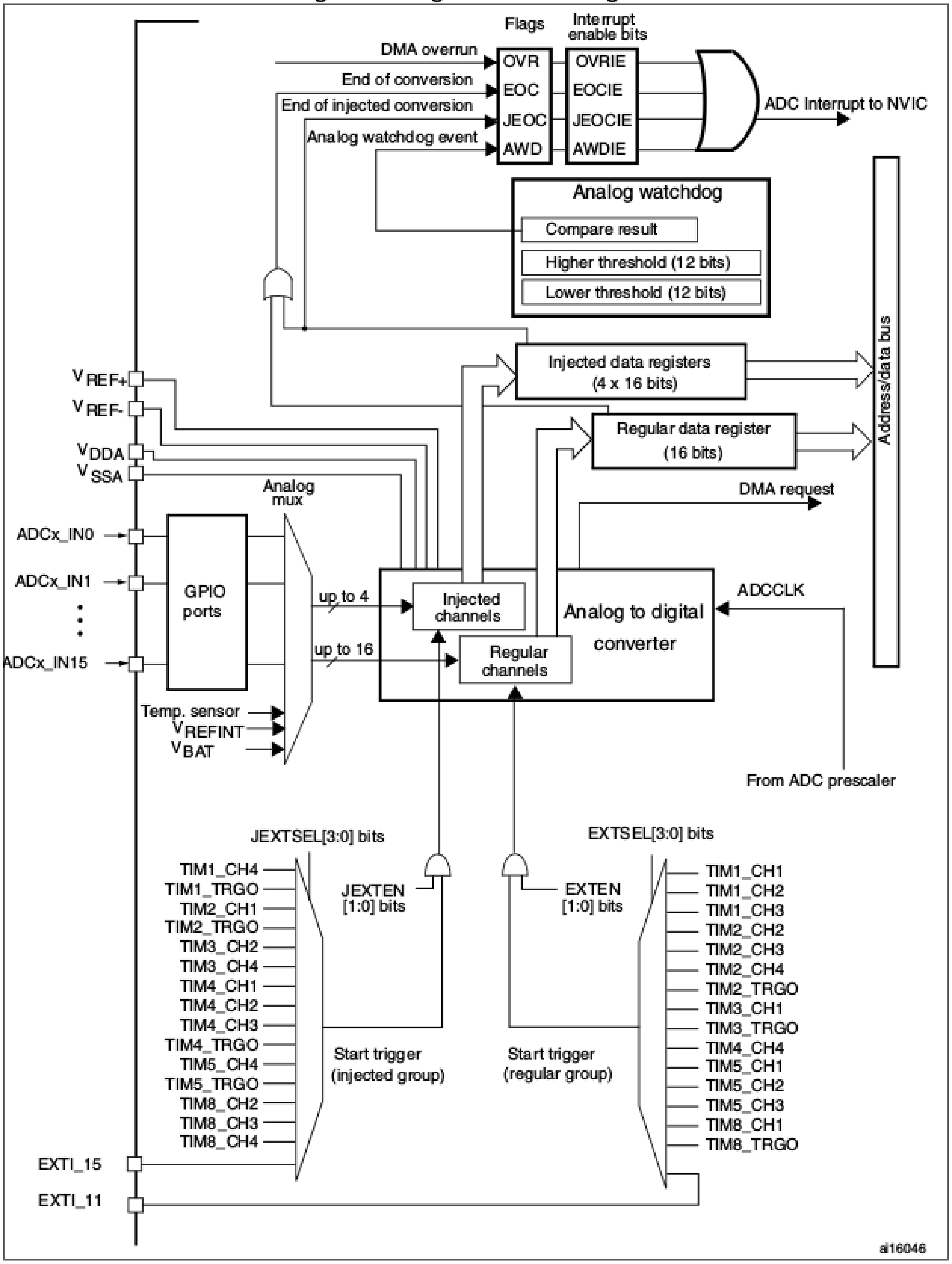
Since a computer is a digital device, any analog signal has be first converted to digital signal with a special device namely Analog-to-Digital converter (ADC). The resolution of the converter indicates the number of discrete values it can produce over the range of analog values. Thus, when we observed a value from ADC, the real voltage is a faction of value read multiplied by the reference voltage. For example, a 10-bit ADC can give a value between 0 and 1023. If a reference voltage is 3.3 volts, a 512 from ADC means 3.3*512/1023 volts.

#### STM32F4xx family's ADC

The 12-bit ADC is a successive approximation analog-to-digital converter. It has up to 19 multiplexed channels allowing it to measure signals from 16 external sources, two internal sources, and the $V_{BAT}$ channel. The A/D conversion of the channels can be performed in single, continuous, scan or discontinuous mode. The result of the ADC is stored into a left- or right-aligned 16-bit data register.

The analog watchdog feature allows the application to detect if the input voltage goes beyond the user-defined, higher or lower thresholds.

**Figure 44. Single ADC block diagram**



Source : **Reference Manuals – RM0090**: STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM®-based 32-bit MCUs

## *Photoresistor or light-dependent resistor (LDR)*

In this laboratory, we also use LDR (together with potentiometer) as an analog signal. "LDR is a light-controlled variable resistor. The resistance of a photoresistor decreases with increasing incident light intensity; in other words, it exhibits photoconductivity. A photoresistor can be applied in light-sensitive detector circuits, and light- and dark-activated switching circuits." definition taken from wikipedia.org

## *Pulse-width modulation (PWM)*

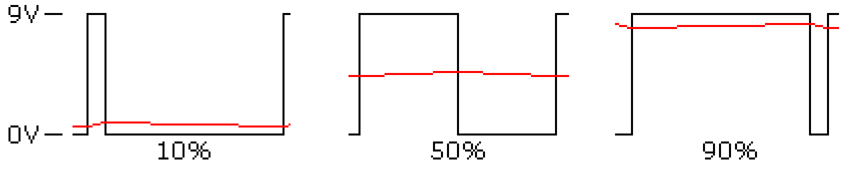
Pulse-Width Modulation allows microcontroller to control the duty cycle of a pulse. It can be used to controller the brightness of LED, speed of motor, or any analog

signal.

PWM can have many of the characteristics of an analog control system, in that the digital signal can be free wheeling. PWM does not have to capture data, although there are exceptions to this with higher end controllers.

One of the parameters of any square wave is duty cycle. Most square waves are 50%, this is the norm when discussing them, but they don't have to be symmetrical. The ON time can be varied completely between signal being off to being fully on, 0% to 100%, and all ranges between.

Shown below are examples of a 10%, 50%, and 90% duty cycle. While the frequency is the same for each, this is not a requirement.



*Examples of PWM Waveforms*

The reason PWM is popular is simple. Many loads, such as resistors, integrate the power into a number matching the percentage. Conversion into its analog equivalent value is straightforward. LEDs are very nonlinear in their response to current, give an LED half its rated current you you still get more than half the light the LED can produce. With PWM the light level produced by the LED is very linear. Motors, which will be covered later, are also very responsive to PWM.

Source: http://www.allaboutcircuits.com/textbook/semiconductors/chpt-11/pulse-width-modulation/

## *Lab Exercises*

1 Create a new STM32 Project on System Workbench IDE to DIM a red LED (LD5) by increasing the duty cycle from 0% duty cycle by 1% every 0.01 second. When the PWM reaches the 100% duty cycle, decrease the duty cycle by 1% every 0.01 second to 0% duty cycle. Repeat the following step forever. (use 100 microseconds period for PWM)

2 Connect LDR to STM32F4Discovery Board and create a new STM32 Project on System Workbench IDE to read the value from LDR and display to serial terminal via UART.

3 Create a new STM32 Project on System Workbench IDE to control a red LED (LD5) brightness with PWM from the value of LDR with this equation.

PWM Duty Cycle = ( 1 – (C / P) ) * 100 %

C = (max environment brightness) – (current environment brightness)

P = (max environment brightness) - (min environment brightness)

## Laboratory 5: Serial Communication Interface

### *Objectives*

1. Understand the concept of MCUs communication interface
2. Understand the concept of Serial Peripheral Interface Bus (SPI)
3. Understand the concept of Inter-Integrated Circuit Bus ($I^2C$)

### *Serial Peripheral Interface Bus (SPI)*

The Serial Peripheral Interface (SPI) is a synchronous serial bus standard with full-duplex capability introduced by Motorola to support communications between a master host processor and one or multiple slave peripheral devices.

SPI is one of the simplest synchronous communications protocols ever developed, as it only establishes a basic mechanism to relay packets between a dedicated master and one or more slaves, without specifying any data, session, or higher-level protocol. This simplicity translates into a protocol with little overhead, capable of achieving a high efficiency in the channel usage, particularly in point-to-point connections.

An SPI controller is developed around a single shift register that serves as both, receiver and transmitter, synchronized by the transmission clock. Figure illustrates the structure of an SPI master-slave pair implementing a point-to point channel. The CPU side of the interface connects to the data lines D0-D7 and the selection and control lines like any other interface, omitted in the figure for simplicity,

The channel side features four signals whose functions are:

- SCLK: Serial clock, sent by the master and synchronizing both master and slave.
- SDO: Serial data-out, the serial output stream from the device.
- SDI: Serial data-in, the serial input stream into the device.
- SS: Slave select, a selection line to enable the slave. The SS line is omitted in point-to-point interconnects, grounding the slave SS input.

A character sent by the CPU is stored in the interface data buffer and copied into the shift register. The master initiates the transfer by activating the slave select signal. In a point-to point, the slave select signal can be hardwired, allowing a 3-wire connection with the master.

Since the master and slave shift registers are tied together and synchronized by the same clock, both interfaces simultaneously transmit and receive. As bits are shifted out from the master they are also shifted into the slave and vice versa. A device doing a transmit-only transfer simply discards the received frame.

Since an SPI does not specify an address field, additional hardware is required for managing multiple slaves. The slave selection lines can be implemented with GPIO lines or an external decoder and user software must activate the corresponding slave.  Figure illustrates a single master, multi-slave SPI configuration.

The SPI clock signal can be derived from the system clock, from an external clock source, or from a timer channel, depending on the particular implementation. Interrupt-based servicing is possible in most MCUs. The specific details will depend on the manufacturer's design.

Like other serial formats, SPI transfers using the logic levels of the interface is limited to only a few inches. By adding appropriate drivers and receivers, SPI channels could be extended over longer distances, although it is mostly used for short intra-board distances. SPI has no error checking mechanism and is not defined as an standard.

Due to its simplicity, SPI has been adopted by numerous manufacturers of serial EEPROMs, real-time clock modules, data converters, LCDs, and other peripherals. Due to the absence of upper-level protocols and no standardization, each implementation can define its own particularities, and upper-level protocols are left to the application implementer

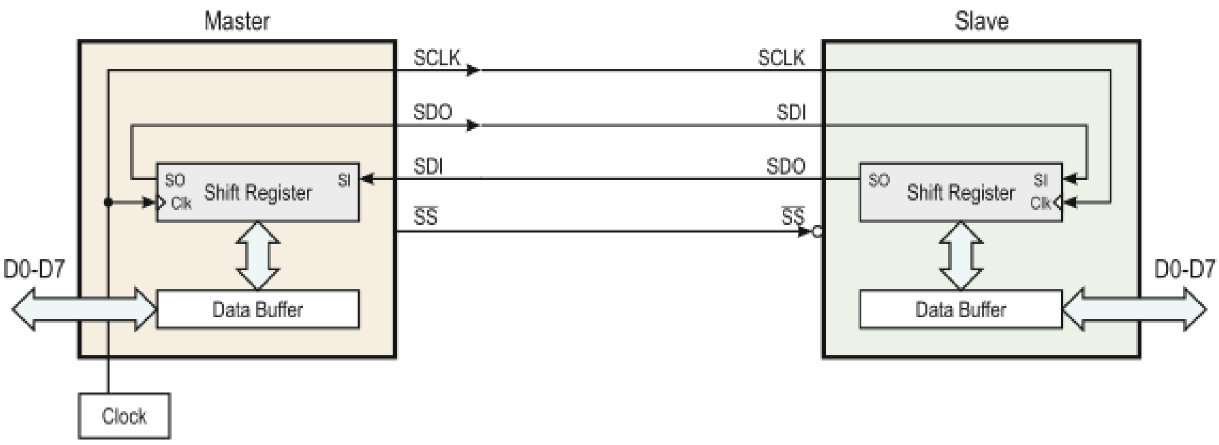


*Figure 1: SPI synchronous bus for a point-to-point connection*

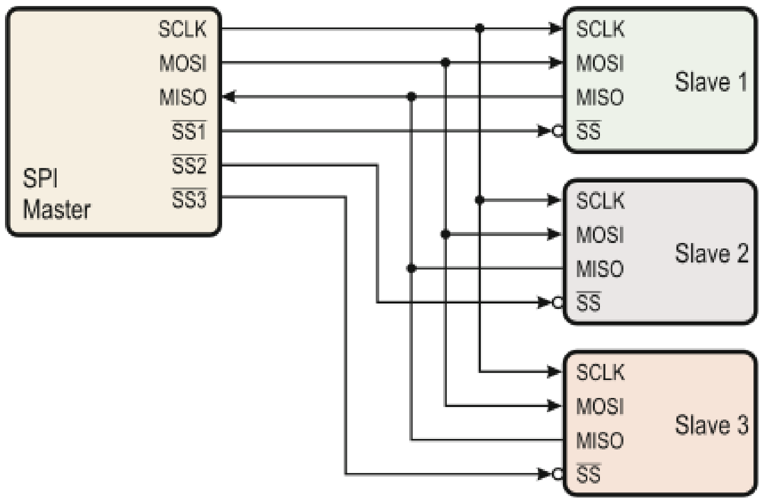


*Figure 2: SPI single-master, multi-slave connection*

## *STM32F407 SPI functional description*

The block diagram of the SPI is shown in  Figure.

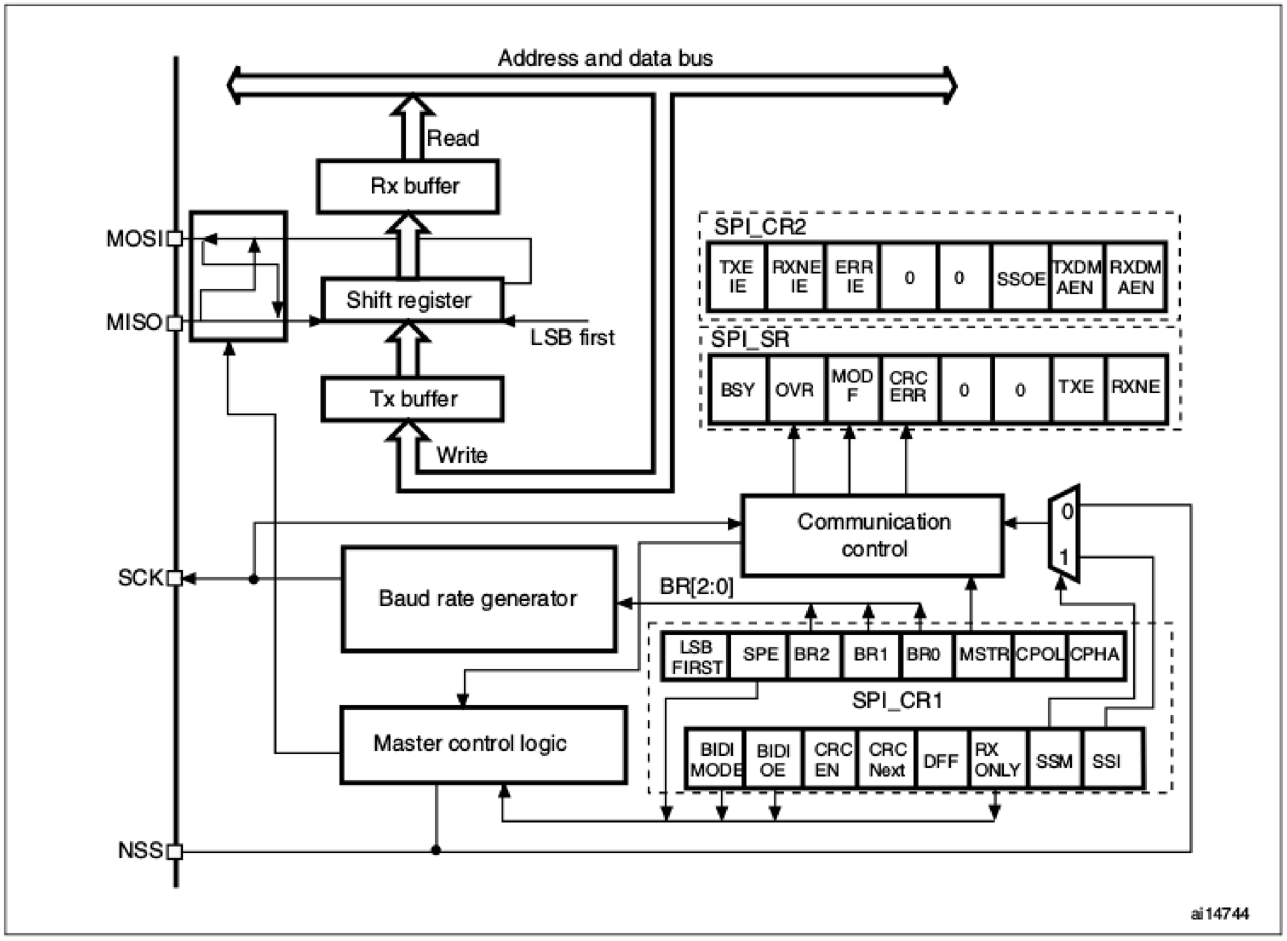
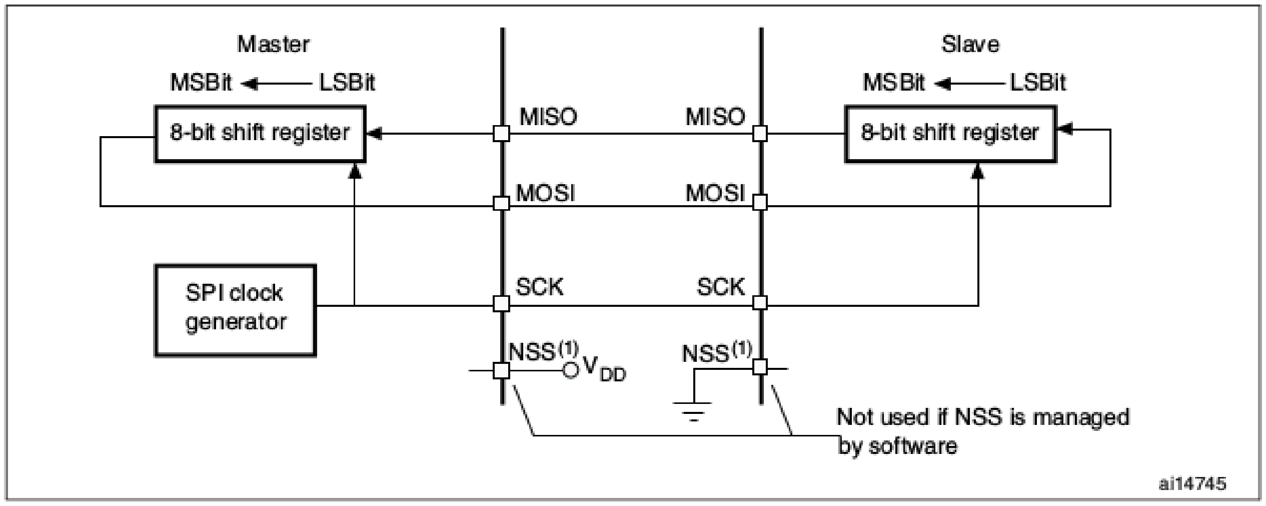


*Figure 3: STM32F4 SPI block diagram*

Usually, the SPI is connected to external devices through 4 pins:

- MISO: Master In / Slave Out data. This pin can be used to transmit data in slave mode and receive data in master mode.
- MOSI: Master Out / Slave In data. This pin can be used to transmit data in master mode and receive data in slave mode.
- SCK: Serial Clock output for SPI masters and input for SPI slaves.
- NSS: Slave select. This is an optional pin to select a slave device. This pin acts as a 'chip select' to let the SPI master communicate with slaves individually and to avoid contention on the data lines. Slave NSS inputs can be driven by standard IO ports on the master device. The NSS pin may also be used as an output if enabled (SSOE bit) and driven low if the SPI is in master configuration. In this manner, all NSS pins from devices connected to the Master NSS pin see a low level and become slaves when they are configured in NSS hardware mode. When configured in master mode with NSS configured as an input (MSTR=1 and SSOE=0) and if NSS is pulled low, the SPI enters the master mode fault state: the MSTR bit is automatically cleared and the device is configured in slave mode

A basic example of interconnections between a single master and a single slave is illustrated in Figure.



1. Here, the NSS pin is configured as an input.

*Figure 4: Single master/ single slave application*

The MOSI pins are connected together and the MISO pins are connected together. In this way data is transferred serially between master and slave (most significant bit first).

The communication is always initiated by the master. When the master device transmits data to a slave device via the MOSI pin, the slave device responds via the MISO pin. This implies full-duplex communication with both data out and data in synchronized with the same clock signal (which is provided by the master device via the SCK pin).

## *Inter-Integrated Circuit Bus ($I^2C$)*

The Inter-Integrated Circuit bus, or $I^2C$ is a synchronous serial protocol developed by Philips Semiconductor (now NXP Semiconductors) in the early 1980s to support board-level interconnection of IC modules and peripherals. The protocol uses two lines, SDA (Serial Data) and SCL (Serial Clock), (and ground) to establish a half-duplex, master/slave, multidrop bus capable of handling multiple masters and slaves. The serial clock line (SCL) synchronizes all bus transfers, while SDA carries the data being transferred.

$I^2C$ was designed to be an intra-system serial bus capable of accommodating all kind of peripherals found in embedded systems. These include MCUs, data converters (ADCs and DACs), display devices, memories, real-time clock calendars, GPIO modules, etc. A large number of IC peripheral manufacturers offer products compatible with $I^2C$.

Devices in an $I^2C$ bus are software addressable, with 7- or 10-bit address fields. Although the most common usage establishes a single master/multiple slave topology, the protocol allows for any device to be a master, as it incorporates collision detection and arbitration mechanisms necessary for a multi-master operation. Nominal maximum speeds can reach up to 5 Mbps, although in reality the limit is imposed by the total bus capacitance, with its maximum specified at 400 pF or 500 pF depending on the version. With input capacitances at 10 pF, plus that of cables, practical numbers call for a few dozen devices per bus. Bus extenders might be used to expand that number if the application requires doing so.

Being a bus for interconnecting ICs in a board, distances in $I^2C$ are expected to be short, in the order of a few inches. It might be possible to stretch its length to several feet, at the expense of reduced speed due to the effect of the increased capacitance.

The bidirectional multidrop capability of the SCL and SDA lines is achieved by driving them with open-collector or open-drain drivers. This calls for fitting bus lines pull-up resistors to complete their driver circuit. Figure shows an $I^2C$ bus topology featuring an MCU as master and several other devices as slaves.

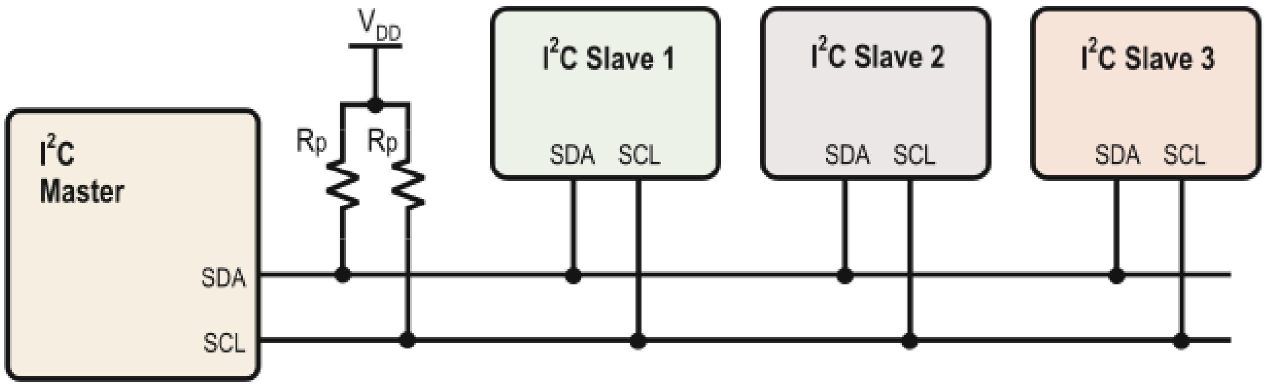


*Figure 5: Topology of an $I^2C$ bus connection*

### *STM32F4 I$^2$C functional description*

In addition to receiving and transmitting data, this interface converts it from serial to parallel format and vice versa. The interrupts are enabled or disabled by software. The interface is connected to the I$^2$C bus by a data pin (SDA) and by a clock pin (SCL). It can be connected with a standard (up to 100 kHz) or fast (up to 400 kHz) I$^2$C bus.

Mode selection

The interface can operate in one of the four following modes:

- Slave transmitter
- Slave receiver
- Master transmitter
- Master receiver

By default, it operates in slave mode. The interface automatically switches from slave to master, after it generates a START condition and from master to slave, if an arbitration loss or a Stop generation occurs, allowing multimaster capability.

Communication flow

In Master mode, the I$^2$C interface initiates a data transfer and generates the clock signal. A serial data transfer always begins with a start condition and ends with a stop condition. Both start and stop conditions are generated in master mode by software.

In Slave mode, the interface is capable of recognizing its own addresses (7 or 10-bit), and the General Call address. The General Call address detection may be enabled or disabled by software.

Data and addresses are transferred as 8-bit bytes, MSB first. The first byte(s) following the start condition contain the address (one in 7-bit mode, two in 10-bit mode). The address is always transmitted in Master mode.

A 9th clock pulse follows the 8 clock cycles of a byte transfer, during which the receiver must send an acknowledge bit to the transmitter. Refer to  Figure.

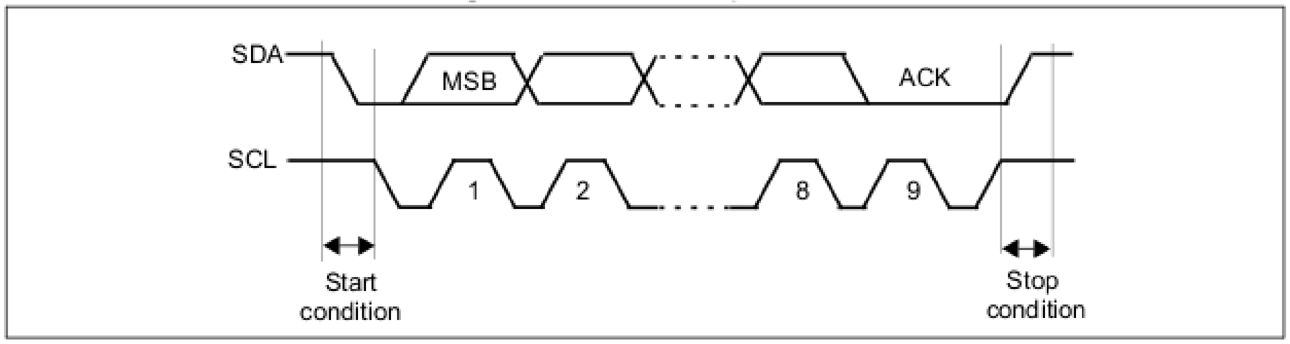


*Figure 6: I$^2$C bus protocol*

Acknowledge may be enabled or disabled by software. The I$^2$C interface addresses (dual addressing 7-bit/ 10-bit and/or general call address) can be selected by software.

The block diagram of the I$^2$C interface is shown in  Figure.

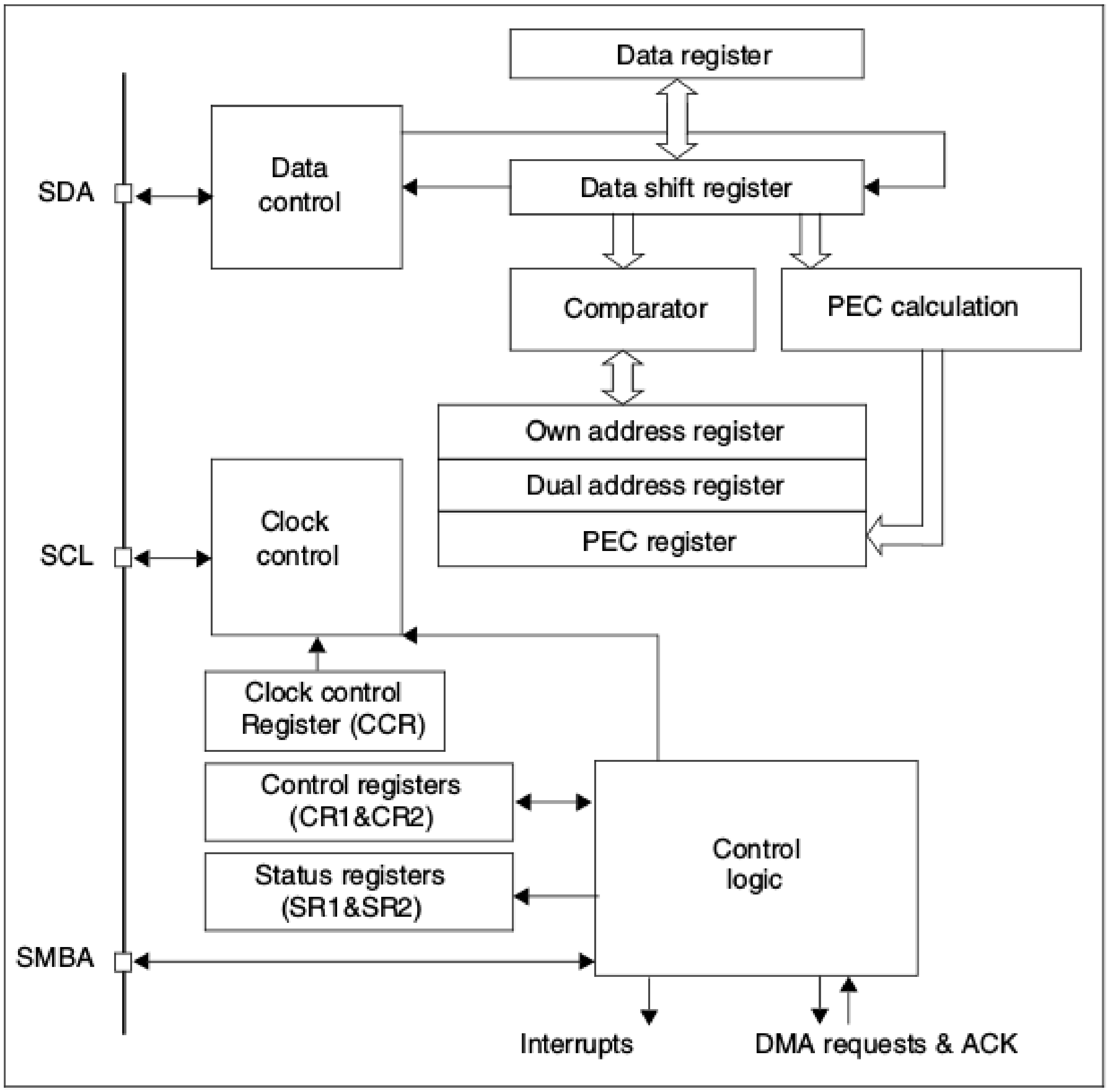
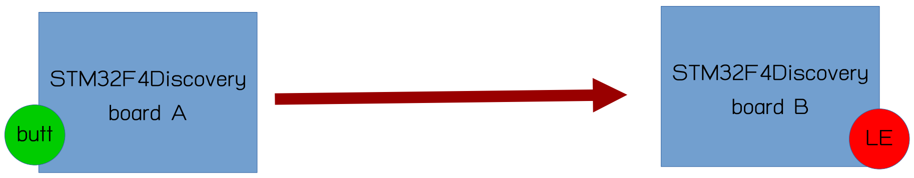


*Figure 7: I²C block diagram for STM32F40x/41x*

Source: Introduction to Embedded Systems Using Microcontrollers and the MSP430 , Jiménez, Manuel, Palomera, Rogelio, Couvertier, Isidoro

**Reference Manuals – RM0090**: STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM®-based 32-bit MCUs

## *Lab Exercises*



1. Create a new STM32 Project on System Workbench IDE to transfer a data between MCUs using UART. If User button on STM32F4Discovery board A is pressed, a red LED on STM32F4Discovery board B must be toggled.
2. Create a new STM32 Project on System Workbench IDE to transfer a data between MCUs using SPI. If User button on STM32F4Discovery board A is pressed, a red LED on STM32F4Discovery board B must be toggled.
3. Create a new STM32 Project on System Workbench IDE to transfer a data between MCUs using I$^2$C. If User button on STM32F4Discovery board A is pressed, a red LED on STM32F4Discovery board B must be toggled.