***Internet of Things Course***

**LAB 2**

**Digital Input/Output & GPIO**

**(Optional)**

**Issue 1.0**

Contents

[Lab overview 1](#_Toc36795931)

[1 Working with the GPIO Peripheral 3](#_Toc36795932)

[1.1 Configuration via registers 3](#_Toc36795933)

[1.2 GPIO Registers 4](#_Toc36795934)

[1.3 Register Addresses 5](#_Toc36795935)

[1.4 Defining the GPIO Structure 5](#_Toc36795936)

[2 Working with Mbed OS 6](#_Toc36795937)

[2.1 Mbed SDK API 6](#_Toc36795938)

[2.1.1 Driver APIs 6](#_Toc36795939)

[2.2 Hello World Example 7](#_Toc36795940)

[3 Application Code 7](#_Toc36795941)

## Lab overview

In this lab, we will learn how to configure and use the General Purpose Input/Output (GPIO) capability of a development board, at the register level without the use of an Application Programming Interface (API).

By the end of the lab you should understand how the configuration of GPIO peripherals is represented at a low level, as well as how to use this to implement a simple I/O functionality.

This lab requires the DISCO-L475VG-IOT01A board, as well as Mbed Studio or another suitable development environment.

**Hardware**

* **DISCO-L475VG-IOT01A**

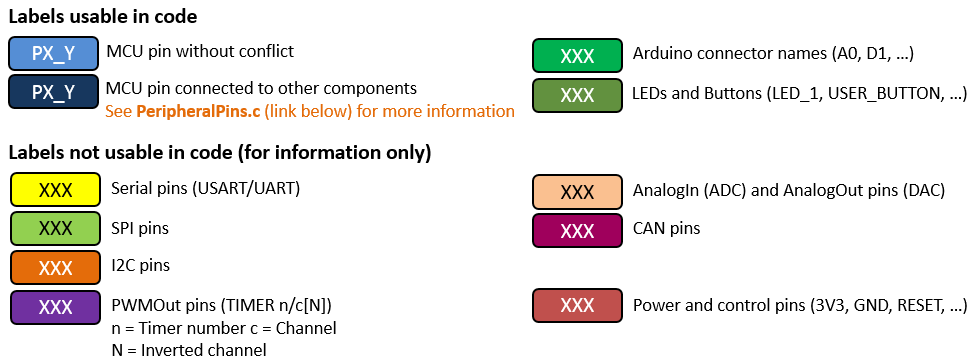


Figure 1: DISCO-L475VG-IOT01A

**Board Features**

* 64-Mbit Quad-SPI (Macronix) Flash memory
* Bluetooth® V4.1 module (SPBTLE-RF)
* Sub-GHz (868 or 915MHz) low-power-programmable RF module (SPSGRF-868 or SPSGRF-915)
* Wi-Fi® module Inventek ISM43362-M3G-L44 (802.11 b/g/n compliant)
* Dynamic NFC tag based on M24SR with its printed NFC antenna
* Two digital omnidirectional microphones (MP34DT01)
* Capacitive digital sensor for relative humidity and temperature (HTS221)
* High-performance 3-axis magnetometer (LIS3MDL)
* 3D accelerometer and 3D gyroscope (LSM6DSL)
* 260-1260hPa absolute digital output barometer (LPS22HB)
* Time-of-Flight and gesture-detection sensor (VL53L0X)
* Two push-buttons (user and reset)
* USB OTG FS with Micro-AB connector
* Expansion connectors:
  + Arduino™ Uno V3
  + PMOD
* Flexible power-supply options: ST LINK USB VBUS or external sources
* On-board ST-LINK/V2-1 debugger/programmer with USB re-enumeration capability: mass storage, virtual COM port, and debug port

**Board Pinout**



**Arduino Compatible Headers:**

A close up of a device

Description automatically generatedA picture containing implement, pencil

Description automatically generated

Figure 2: Arduino headers

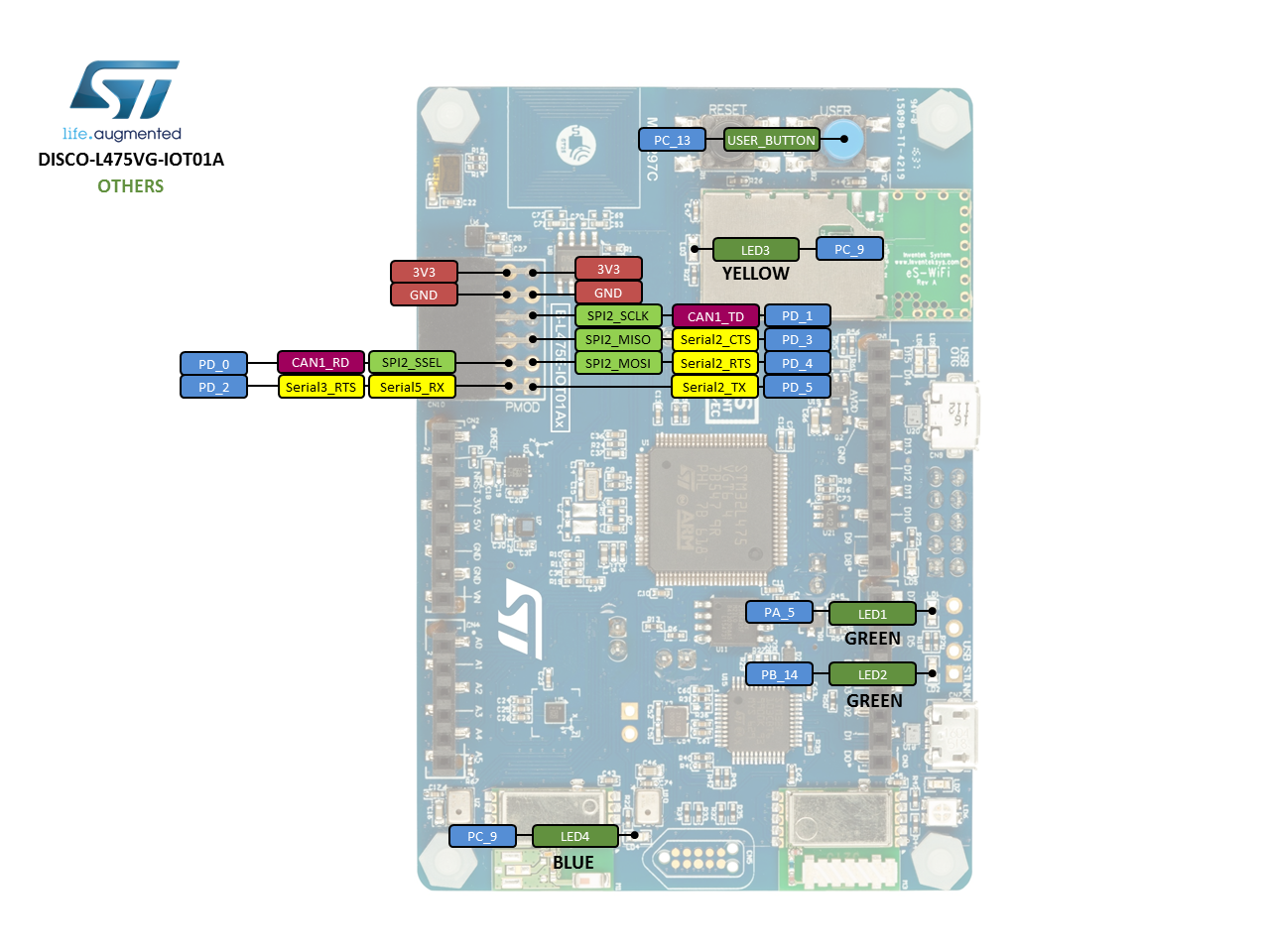


Figure 3: Board pinout

# Working with the GPIO Peripheral

## Configuration via registers

As the GPIO pins are, by name and nature, general purpose, to use them we must specify how we wish the board to treat them. This, as well as additional configuration (such as enabling by-default disabled clocks), is done via a specified set of registers, the values of which correspond to different configurations.

We can write to these registers directly in our program, by treating them as if they were numerical variables. We will show this below with an example where we write to PRI\_REG, which denotes a register REG associated with a peripheral PRI.

Bitwise operations: As with += and -=, X #= Y corresponds to X = X # Y, where # can be any of the bitwise logical operators on two variables, most notably AND and OR, denoted by & and | respectively. We can also use ~, bitwise NOT, and left and right shifts << and >>. From this we have the following examples of reading/writing to PRI\_REG:

PRI->REG = 0x04; //set bit[2] and clear other bits

PRI->REG = ~0x04; //clear bit[2] and set other bits

PRI->REG |= 0x05; //set bit[2] and bit[0] and keep others unchanged

PRI->REG &= ~0x05; //clear bit[2] and bit[0] and keep others unchanged

x = ((REG->DIR & (1 << 2))>> 2); //write the boolean value of bit[2] to variable x

## GPIO Registers

Standards used for inputs, outputs, and the associated registers vary between different families of boards, and reference manuals can usually be found through the manufacturers’ websites. For instance, the reference manual for the STM32L475VG family, can be found here: <https://www.st.com/content/ccc/resource/technical/document/reference_manual/02/35/09/0c/4f/f7/40/03/DM00083560.pdf/files/DM00083560.pdf/jcr:content/translations/en.DM00083560.pdf> Before being able to configure the GPIOs, you have to enable the clock through the corresponding port writing at the correct register of the Reset and Clock Control module (RCC).

* **RCC AHB2 peripheral clock enable register (RCC->AHB2ENR)**

Most of the modules are by default disabled to save power. Writing a 1 to this register will enable the corresponding clock. Writing a 0 will disable the clock.

Then the GPIO port has 10 32-bit registers to configure every pin.

* **GPIO port mode register (GPIOx\_MODER)**

This register configures the I/O direction mode. Writing into this register will set the corresponding pin as follows:

* 00: General purpose input mode.
* 01: General purpose output mode.
* 10: Alternate function mode.
* 11: Analog mode (reset state).
* **GPIO port output type register (GPIOx\_OTYPER)**

This register is used to select between an open drain or pull-up/pull-down resistor configuration for the outputs. Writing a 1 will select the open-drain mode.

* **GPIO port output speed register (GPIOx\_OSPEEDR)**

This register can be written to select four different speeds for the outputs. Lower speeds improve the power efficient but may not be suitable for some purposes. Write 00 for low speed, 01 for medium speed, 10 for fast speed, and 11 for high speed.

* **GPIO pull-up/pull-down register (GPIOx\_PUPDR)**

This register is used to choose between a pull-up or pull-down resistor, if you haven’t selected open-drain mode in register GPIOx\_OTYPER. Write 01 to select the pull-up resister or 10 to select the pull-down. For inputs, you can also select 00 for a non-pull-up/down resistor.

* **GPIO port input data register (GPIOx\_IDR)**

This register is read only and is used to read input values. These always reflect the logic level of the corresponding pin.

* **GPIO port output data register (GPIOx\_ODR)**

This register can be read and written. We can use it to read the inputs or to set the outputs to high or low. However, for stability, it is recommendable to read the inputs through register GPIOx\_IDR and set and clean the outputs using register GPIOx\_BSRR.

* **GPIO port bit set/reset register (GPIOx\_BSRR)**

This register has been separated into two parts. Bits 0 to 15 (the low bits) are set bits, and bits 16 to 31 are the reset bits.

* **GPIO port configuration lock register (GPIOx\_LCKR)**

This register is used to lock the configuration of the port bit.

* **GPIO alternate function low register (GPIOx\_AFRL) and high register (GPIOx\_AFRH)**

These two registers are used to select the different alternative functions that some pins have.

## Register Addresses

All the registers are one-to-one mapped to the 32-bit global memory space.

The base addresses for GPIO are listed below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Peripheral | GPIOA | GPIOB | GPIOC | GPIOD | GPIOE | GPIOH |
| Base Address | 0x48000000 | 0x48000400 | 0x48000800 | 0x48000C00 | 0x48001000 | 0x48001C00 |

Address offsets for the GPIO registers are listed below:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Register | MODER | OTYPER | OSPEEDR | PUPDR | IDR | ODR | BSRR | LCKR | AFR[2] |
| Offset | 0x00 | 0x04 | 0x08 | 0x0C | 0x10 | 0x14 | 0x18 | 0x1C | 0x20-0x24 |

## Defining the GPIO Structure

To define and use a GPIO peripheral, it is best to define the pins into a structure and map it to the memory. Most manufacturers provide the developer with a header file with the addresses and definitions for all the registers.

We can find the register definition of our microcontroller in the document **stm32f401xd.h.** This file has already been included in our copy of mbed-os, found in the project files.

typedef struct

{

\_\_IO uint32\_t MODER; /\*!< GPIO port mode register, Address offset: 0x00 \*/

\_\_IO uint32\_t OTYPER; /\*!< GPIO port output type register, Address offset: 0x04 \*/

\_\_IO uint32\_t OSPEEDR; /\*!< GPIO port output speed register, Address offset: 0x08 \*/

\_\_IO uint32\_t PUPDR; /\*!< GPIO port pull-up/pull-down register, Address offset: 0x0C \*/

\_\_IO uint32\_t IDR; /\*!< GPIO port input data register, Address offset: 0x10 \*/

\_\_IO uint32\_t ODR; /\*!< GPIO port output data register, Address offset: 0x14 \*/

\_\_IO uint16\_t BSRRL; /\*!< GPIO port bit set/reset low register, Address offset: 0x18 \*/

\_\_IO uint16\_t BSRRH;/\*!< GPIO port bit set/reset high register, Address offset: 0x1A \*/

\_\_IO uint32\_t LCKR; /\*!< GPIO port configuration lock register, Address offset: 0x1C \*/

\_\_IO uint32\_t AFR[2];/\*!< GPIO alternate function registers, Address offset: 0x20-0x24 \*/

} GPIO\_TypeDef;

You can modify the file or write your own definitions if you need. The file also defines masks. Masks are used to address a specific bit of a register. Below are the definitions of the masks to address the first bit of the register MODER, the second one, or both at the same time.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Bits definition for GPIO\_MODER register \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define GPIO\_MODER\_MODER0\_0 ((uint32\_t)0x00000001)

#define GPIO\_MODER\_MODER0\_1 ((uint32\_t)0x00000002)

#define GPIO\_MODER\_MODER0 ((uint32\_t)0x00000003)

Review the file and see how the chain of base addresses, offset addresses, plus masks give the correct absolute address for a specific bit of some register.

Finally, to clear a particular bit of a register in the peripheral, you can simply write:

GPIOA->MODER &= ~GPIO\_MODER\_MODER0\_1; //Clear bit 1 of register MODER

//of peripheral GPIO port A

To set a bit of a register in the peripheral, you can simply write:

GPIOA->MODER |= GPIO\_MODER\_MODER0\_0; //Set bit 0 of register MODER

//of peripheral GPIO port A

# Working with Mbed OS

The Mbed software development kit (SDK) provides the Mbed C/C++ software platform and tools for creating the microcontroller firmware that runs on smart devices. It consists of the core libraries that provide the microcontroller peripheral drivers, networking, RTOS, and runtime environment, build tools, and test and debug scripts. A components database provides driver libraries for components and services that can be connected to the microcontrollers to build a final product.

## Mbed SDK API

The Mbed SDK gives you an API-driven approach to microcontroller coding. We can code using meaningful abstract objects and API calls; therefore, there is no need to learn the microcontroller hardware details to start developing.

### Driver APIs

Driver APIs in Mbed OS include analog and digital inputs and outputs on development boards, as well as digital interfaces, which allow a board to interface with a computer or external devices. Some of the most commonly used driver APIs are listed below:

* **DigitalOut**: The DigitalOut interface is used to configure and control a digital output pin by setting the pin to logic level 0 or 1. It can be used on any pin with a blue label as shown in the pin diagram, and also with the on-board LEDs.
* **DigitalIn**: The DigitalIn interface is used to read the value of a digital input pin. The logic level is either 1 or 0. Any of the numbered Arm Mbed pins can be used as a DigitalIn.
* **AnalogIn:** The AnalogIn API is used to read external voltages applied to an analog input pin. The value is a floating point from 0.0 (VSS) to 1.0 (VCC). For example, if you have a 3.3V system and the applied voltage is 1.65V, then AnalogIn() reads 0.5 as the value.
* **InterruptIn:** TheInterruptIn API is used to trigger an event when the value of the digital input pin changes. We can trigger interrupts on the rising edge (change from 0 to 1) or falling edge (change from 1 to 0) of signals.

For all the Mbed C/C++ SDK APIs, see the [API List](https://os.mbed.com/docs/mbed-os/v5.15/apis/index.html).

## Hello World Example

Let’s start by looking at a basic “Hello World!” program that blinks the on-board LED

#include "mbed.h"

DigitalOut myled(LED1); //Digital pin LED1 is initialized as output

int main() {

    while(1) {

        myled = 1;

        thread\_sleep\_for(200); //delay in ms

        myled = 0;

        thread\_sleep\_for(200); //delay in ms

    }

}

# Application Code

In this lab, you need to complete two exercises:

Exercise 1: Low level LED Blinky

* Use a GPIO to read the status of the on-board user button;
* Use a GPIO to control the on-board LEDs;
* Complete the main function, so that while the user button is pressed, the LED is on.

Exercise 2: Blink LED1 and LED2 using Mbed OS

* Declare LED1 and LED2 as output;
* Use the user-button as interrupt to control the LEDs;
* Complete the main function, so that either of LED1/LED2 starts blinking each time when the user button is pressed at a desired frequency.