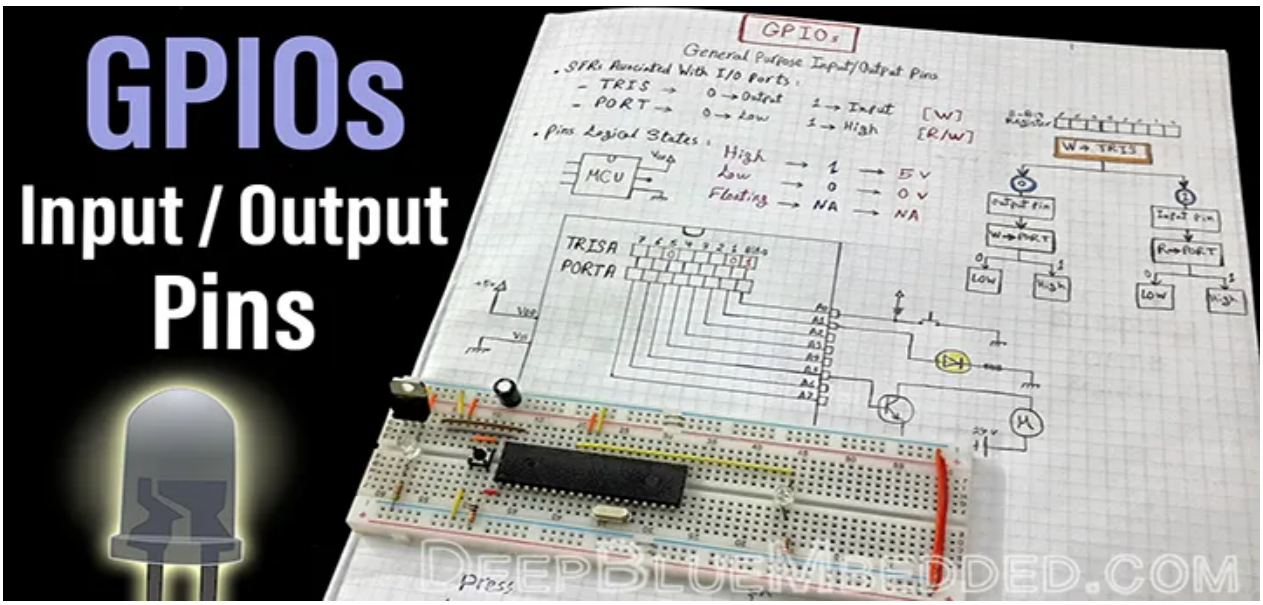


PIC Microcontrollers Input/Output IO Ports GPIOs Tutorial

by Khaled Magdy

Previous Tutorial	Tutorial 4	Next Tutorial
Input/Output IO Ports – GPIO Pins		

DeepBlue Menu

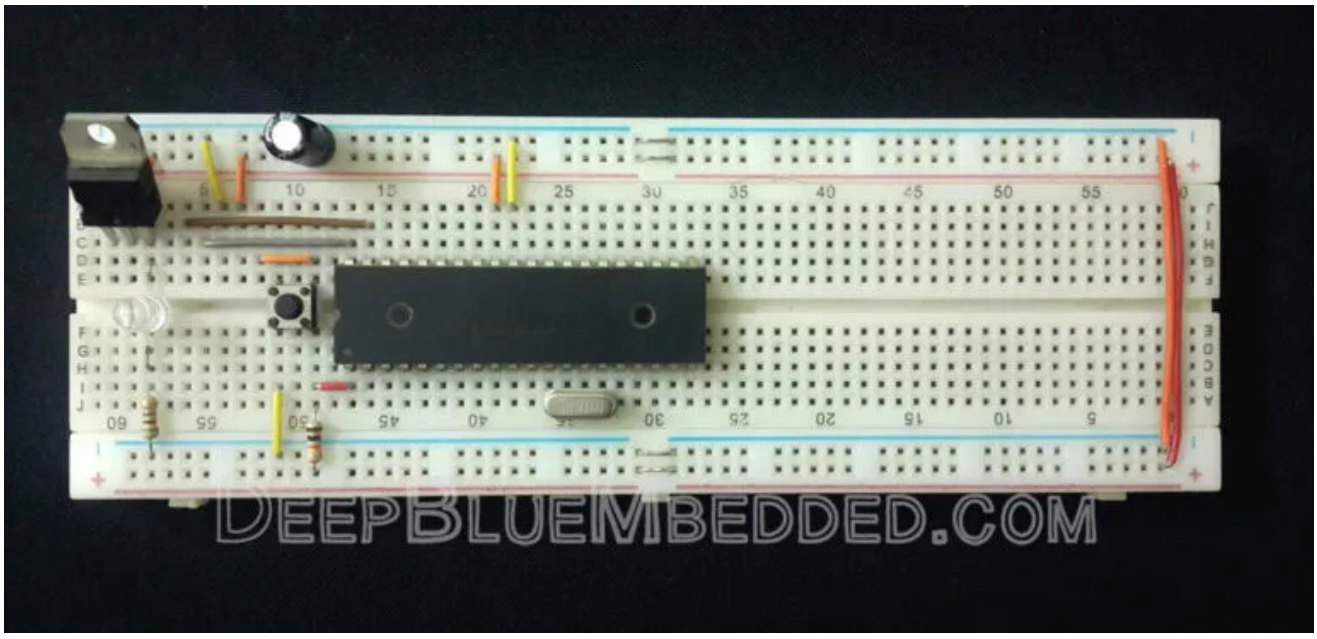
in this tutorial, we'll address the Input/Output Ports in the Microchip PIC microcontrollers. You'll learn much about the digital input/output pins **GPIO**s in microcontrollers, how they work and how to drive/interface them properly. We'll also flash some LEDs for testing out these concepts. With MPLAB x IDE, XC8 Compiler, and PICKit2 or 3. There is also a video guide by the end of this tutorial that you should check out.

Tutorial Contents

- 1 Required Components
- 2 What are GPIO pins?
- 3 How does a digital input/output port work?
  - 3.1 1. Configure i/o pins
  - 3.2 2. Read/Write digital i/o pins
- 4 Flashing some LEDs – LAB
  - 4.1 1. Coding
  - 4.2 2. Simulation
  - 4.3 3. Prototyping
- 5 Reading Digital Inputs – LAB
  - 5.1 1. Coding
  - 5.2 2. Simulation
  - 5.3 3. Prototyping
- 6 The power of #define directive
- 7 Checkout My PIC MCU Programming Video Tutorial
- 8 Share this:
- 9 Related

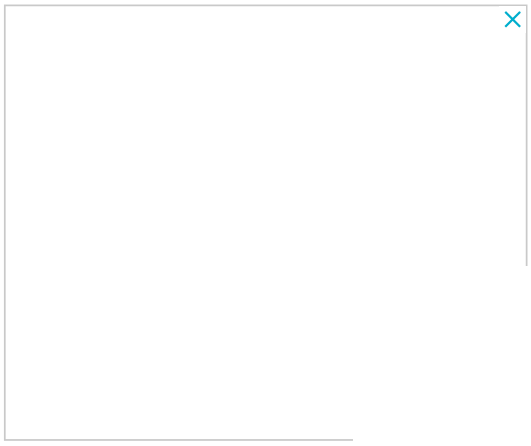
1	PIC16F877A	Add
1	BreadBoard	Add
3	LEDs	Add Add
2	Push Buttons	Add
1	Resistors Kit	Add Add
1	Capacitors Kit	Add Add
1	Jumper Wires Pack	Add Add
1	LM7805 Voltage Regulator (5v)	Add
1	Crystal Oscillator	Add
1	PICkit2 or 3 Programmer	Add
1	9v Battery or DC Power Supply	Add Add Add

The Prototyping Board Setup



GPIO stands for "**General Purpose Input/Output pins**". In fact, most of the pins in a typical microcontroller are GPIO pins except for some special pins. The special (non-GPIO) pins are usually the following ones:

- Power Supply pins: Vdd & Vss
- OSCillator pins: OSC1 & OSC2. Those couple of pins are used to provide the MCU with the oscillator clock input it needs.
- MCLR: the master clear reset pin. Used for restarting the MCU on purpose.



- V_{USB}: the external USB power source.

Except for the special function pins, all the other pins are GPIOs. As shown below from the PIC16F877A datasheet, the GPIO pins are clustered in PORTs.

As you may have noticed that the pins [2 to 7] are named [RA0 to RA5] and that's why this is called "**PORTA**". In our MCU there are 5 digital Input/output ports [**A, B, C, D, and E**]. It's common in 8-Bit microcontrollers to have I/O ports each of which has up to 8-pins. However, some I/O ports may have less than 8-pins as in our PIC16F877A for examples. As you can see

- PORTA: has 6 pins [2 to 7]
- PORTB: has 8 pins [33 to 40]
- PORTC: has 8 pins
- PORTD: has 8 pins
- PORTE: has 3 pins [8 to 10]

Note			
The pins of every single port may not be in consecutive order physically on the chip. Which is the case for PORTC & PORTD as you may have noticed			

Generally speaking, the I/O ports are operating similarly which mean that you only need to investigate one of them. Then the other ones will be using the exact same concept and working in a similar way. For this tutorial, we'll be using **PORTB** for our Demo. & Lab.

There are basically a couple of registers dedicated to each I/O port to control its operation.

- **TRISx** Register

- **PORTx** Register

The registers dedicated to PORTA pins are (TRISA & PORTA), and the registers dedicated to PORTB pins are (TRISB & PORTB), and so on.

Controlling or driving the I/O port is a pretty easy process. It involves only two consecutive steps.

1. Configure The Pin, determine the data direction, whether it's an **Input** or **Output**
2. Read / Write. If the pin is configured to be an input pin, then you'll be reading (polling) its state several times within your program. If the pin is configured to be an output pin, then you'll be driving its state (High / Low) within your program.

1. Configure i/o pins

Whenever you wish to use an I/O pin, you must first configure it to be an output or input pin. Here the function of TRISx register comes in, as the value we write to that register specifies the direction of the corresponding pins (input/output). The **x** here stands for any port [A, B, C, D or E].

- Writing **1 (High)** to a single bit of the TRISx register configures the pin corresponding to this bit to

- Writing **0 (Low)** to a single bit of the TRISx register configures the pin corresponding to this bit to be an **output** pin.

Which means that writing the following values to these bits will cause the following effects

Value	Bit	Effect
1	TRISB0	**RB0** pin becomes an input pin
0	TRISB1	**RB1** pin becomes an output pin
1	TRISC4	**RC4** pin becomes an input pin
0	TRISD3	**RD3** pin becomes an output pin
And so on…		

However, it’s possible to configure the 8-pin of any port all at once instead of bit-by-bit changes as follows

```
1 TRISB = 0x00;   // Configures the 8-pins of PORTB [RB0 to RB7] to be output pins
2 PORTB = 0xFF;   // Drives the 8-pins of PORTB [RB0 to RB7] to be High (logic 1)
```

Generally speaking, in C-Programming, writing to bits/registers could be done in several ways from which I'll discuss the most common 3 methods.

I- Register overwriting method

```
1 TRISB = 0x00; // Clears the low 3-Bits to be low , in order to make them output pin
```

or equivalently

```
1 TRISB = 0b00000000;
```

Pro Tip			

DeepBlue Menu

the low 3-Bits. Which we may not be interested in changing them. In the above **LOC** (line of code),

we're interested in affecting only 3-Bits of the TRISB register. However, doing "**Register overwriting**"

II- Bit-Masking method

```
1 TRISB &= (0b11111000); // Clears the low 3-Bits to be low
2                        // in order to make them output pins
```

pic.

Note
This method is a preferred one and considered to be a friendly way to set or clear specific bits within a register.

III- Bit-fields method

This method is the easiest and safest one of all. However, it's not a common practice for non-PIC dudes.

```
2 TRISBbits.TRISB1 = 0;
3 TRISBbits.TRISB2 = 0;
```

or (equivalently)

```
1 TRISB0 = 0;
2 TRISB1 = 0;
3 TRISB2 = 0;
```

In case you're curious. **How Bit-fields method works?**

" >**Click here to view answer**

Well, do you remember the step in which we wrote the name of our MCU **PIC16F877A** while creating our project?

This actually automatically includes for you (behind the scene) a pre-written header file called **pic16f877a.h** which has around 3 KLOC (3000 LOC)! I'll show you how to view it and illustrate a snippet of this code.

write the following LOC at the beginning of your code file

```
1 #include <pic16f877a.h>
```

then hover your mouse over the <**pic16f877a.h**> then right-click it & choose to navigate to the file definition. This will open up the whole file in a new tab. Here is a snippet of this file.

DeepBlue Menu

X

As you can see, they've created bit-fields of each register @ the same memory address in the RAM as in the datasheet. The fields' width is specified to be 1-Bit for each field, using this operator [**:1**] for each field. The fields' names are usually the same as in the datasheet.

Typically a bit-fields for a specific register follows the convention shown below.

Where X is the name of any specific register as found in the datasheet. Which means that accessing any bit is easily done by writing the following.

Xbits.

This will list down all the bits of this register (fields) for you to choose from. Try it yourself!

Notice: In fact, other than pic16f877a.h file, there are many other header files that contain register maps and bit-fields pre-defined and pre-written for all PIC microcontrollers within the XC8. Which is a free open-source code that you should definitely take advantage of.

2. Read/Write digital i/o pins

After configuring any i/o pin, you'll be able to read its state (if it's an input pin) or to change its logical state (if it's an output pin). Here the function of **PORTx** register comes in, as the value we write to that register specifies the logical state of the corresponding pins (High 1/ Low 0).

- Writing **1 (High)** to a single bit of the PORTx register sets the corresponding pin to be **High**.

- Writing **0 (Low)** to a single bit of the PORTx register sets the corresponding pin to be **Low**.

Which means that writing the following values to these bits will bring us the following results

Value	Bit	Effect
1	RB2	**RB2** pin is driven High (logic 1), but it must first be configured as an output pin
0	RB3	**RB3** pin is driven Low (logic 0), but it must first be configured as an output pin
1	RC5	**RC5** pin is driven High (logic 1), but it must first be configured as an output pin
0	RD7	**RD7** pin is driven Low (logic 0), but it must first be configured as an output pin
And so on…		

Reading-in the logical state of an input pin requires additional hardware connections in order to guarantee a stable system behavior. A typical pin can be driven to **High**, **Low** or left **Floating**. However, you shouldn't leave any i/o pin floating in the air as long as you're using it as an input for your system.

The floating state must be avoided under any circumstances, as it's not deterministic consistent input. It totally depends on how noisy the environment is. For this reason, input pins are generally pulled-down or pulled-up.

- **Input Pins Pull-Up**
  In this configuration, the pin is always pulled-up to High (logic 1) until an event occurs to drive it Low (to 0). Mostly all you need to have is a 10kohm resistor connected between the input pin and the Vdd (+5v) as shown below.

The logical state of the pin is always **1** until the button is pressed, then it's short-circuited with ground and becomes **0**. This is called **Negative Logic** input, as the action of the user (pressing button) transfers the digital pin state from High to **Low**.

- **Input Pins Pull-Down**

  In this configuration, the pin is always pulled-down to Low (logic 0) until an event occurs to drive it High (to 1). Mostly all you need to have is a 10kohm resistor connected between the input pin and the Vss (0v or Gd) as shown below.

The logical state of the pin is always **0** until the button is pressed, then it's short-circuited with Vdd (+5v) and becomes **1**. This is called **Positive Logic** input, as the action of the user (pressing button) transfers the digital pin state from Low to **High**.

Note			
Some pins have internal Pull-Up resistors built-in the microcontroller itself (e.g. PORTB pins). However, in most cases, you'll have to choose & create one of the 2 configurations above which are identical in function but reversed in logic handling mechanisms.			

Let's take a simple example for reading a digital input. Say we've got an i/o pin called **RD4** which is

```
2 {
3     // Do something ...
4 }
```

or equivalently

```
1 if ( RD4 == 1 )
2 {
3     // Do something ...
4 }
```

or equivalently

```
1 if ( RD4 )
2 {
3     // Do something ...
4 }
```

Take the time to get familiar with these concepts and play around to make sure that everything you understand is actually behaving as it should be.

Hint

Checking the input pins' states using control flow logic (e.g. if, else, switch, while, etc.) is called **Polling** and it's not the best choice for handling critical input signals to the system specifically those which require an instantaneous response from the system (Real-Time). This will be discussed later in The Interrupts Tutorial.

For now, we're done with reading digital inputs and let's move on to see how we can write to the digital output pins in our microcontroller.

Writing to the PORTx register will change the logical state of i/o pins (only if they are configured as output pins). The output pins' logical states 0, 1 corresponds to 0v, +5v respectively. Which means that a typical output pin could possibly be sourcing current to other devices or it could be sinking current instead.

- **Output Pins Current Sourcing**
  In this configuration, the output pin is used as a (+5v) source that delivers current to drive very small loads (e.g. LEDs, Transistors, Relays, Optocouplers, etc.). A typical pin in our microcontroller can source up to 25 mA for each i/o pin and this value varies from an MCU chip to another. This is the most commonly used configuration and it should be clear in the diagram below.

- **Output Pins Current Sinking**
  In this configuration, the output pin is used as a (0v or Ground) that sinks current from small loads (e.g. LEDs, Transistors, Relays, Optocouplers, etc.). A typical pin in our microcontroller can sink up to 25 mA for each i/o pin. This configuration is shown in the diagram below.

Check the datasheet for PIC16F877A, chapter 17.0 (Electrical Characteristics), page 173

Note			
You should never exceed the current limit for i/o pins at any cost. Otherwise, it'll cost you the entire chip itself or at least some pins if you're lucky enough. Loads like DC motors, Stepper motors, Large LEDs array, etc. all of them could be easily interface with a small transistor biasing circuit. You'll be safe and so do your MCU as well.			

Let's take a simple example for writing a digital output. Say we've got an i/o pin called **RC5** which is configured to be an output pin. This pin is connected to a 330 resistor with a small LED. We're willing to turn it on for 1 second and turn it back off again (Flash!). The C-Code for this is shown below.

```
1 RC5 = 1;            // Turn ON
2 __delay_ms(1000);   // Wait 1000ms = 1 second
3 RC5 = 0;            // Turn OFF
```

And now we're done with the demo part of this tutorial and let's move on to the practical LABs.

Lab Name	Flashing LEDs (Digital Output)
Lab Number	1
Lab Level	Beginner
Lab Objectives	Learn how to use GPIO pins for generating a digital output signal and use it for flashing some LEDs.

Note			
All the LABs in this course are performed in 3 successive steps: 1- **Coding** 2- **Simulation** 3- **Prototyping**			

1. Coding

Open the MPLAB IDE and create a new project name it “Flashing LEDs”. If you have some issues doing so, you can always refer to the previous tutorial using the link below.

Set the configuration bits to match the generic setting which we’ve stated earlier. And if you also find troubles creating this file, you can always refer to the previous tutorial using the link below.

Now, open the **main.c** file and let's start developing the firmware for our project.

Never forget to include the configurations header file within the main.c file of your project.

First of all, we've got three LEDs which will be hooked to 3 of our GPIO pins at PORTB. This means that we'll have to set the data direction for 3 of the PORTB pins to be output pins. Let's use the following pins for our LAB [**RB0, RB1, and RB2**].

Setting the direction (input or output) of these pins is achieved via the TRISB register as follows.

```
TRISB0 = 0;
TRISB1 = 0;
TRISB2 = 0;
```

After configuring the GPIO pins which we'll be using, it's time to move to our main loop. In which we'll flash the LEDs in order. Each of them will be ON for 50ms and OFF for another 50ms.

```
while(1)
{
// Flash LED1
    RB0 = 1;
    __delay_ms(50);
    RB0 = 0;
    __delay_ms(50);
// Flash LED2
    RB1 = 1;
    __delay_ms(50);
    RB1 = 0;
    __delay_ms(50);
// Flash LED3
    RB2 = 1;
    __delay_ms(50);
    RB2 = 0;
    __delay_ms(50);
}
```

Note			

Using the **__delay_ms()** macro requires that the pre-defined name **_XTAL_FREQ** to be holding your actual oscillator's frequency value. For our example, we're using a 4MHz Crystal Oscillator. Which means that you must include the following LOC at the beginning of your code.

```
#define _XTAL_FREQ     4000000
```

Cross your fingers. Hit the compile button! And we're done with the firmware.

2. Simulation

The schematic for this LAB is shown below. Just create a similar one in your simulator.

After connecting everything up in your simulator, you'll have to double-click the MCU chip and point to the firmware code **.hex** file. The path will be as follows

ProjectName.X **>** dist **>** default **>** production **>** FirmwareFile**.hex**

Make sure to simulate your project at the same oscillator frequency used in the code. Otherwise, the timing diagram will be messed up. Which means that the 50ms time delays in the code will NOT be a

Note

nightmare! There is so little chance of discovering that your Chip is insanely restarting all the time due to a dodgy MCLR pin or not being clocked at all due to OSC issues, so please, be careful.

3. Prototyping

You should have created the basic prototyping setup which we've prepared previously. If you've any troubles doing so, you can refer to that previous tutorial.

Connect the RB0, RB1 and RB2 pins to the 1st, 2nd and 3rd LEDs Cathode leads. Then hook 3 resistors 330 between the LEDs' anodes and the Ground line.

Plug the power supply et voila! you've accomplished your first (Hello World) embedded project. Congratulations!

The final running project in case you're curious

Reading Digital Inputs – LAB

Lab Name	Reading Digital Inputs (Digital Input)
Lab Number	2
Lab Level	Beginner
Lab Objectives	Learn how to use GPIO pins for reading a digital input signal and using it to control the flashing of an LED.

1. Coding

Open the MPLAB IDE and create a new project name it “Digital Input”. If you have some issues doing so, you can always refer to the previous tutorial using the link below.

DeepBlue Menu

troubles creating this file, you can always refer to the previous tutorial using the link below.

Now, open the **main.c** file and let's start developing the firmware for our project.

1. The first one will operate as a digital input pin and will be hooked to the push button to read its logical state (high/low).
2. The second pin will operate as a digital output pin and will be hooked to the LED in order to flash it twice per second.

Let's set the **RB0** pin to be an input pin and **RB1** to be an output. To configure these pins we'll write the following code.

```
1 TRISB0 = 1;    // Input Pin
2 TRISB1 = 0;    // Output Pin
```

After configuring the GPIO pins which we'll be using, it's time to move to our main loop. In which we'll read the state of the button hooked to pin **RB0**. And if it's pressed down, we'll flash the LED hooked to **RB1** pin.

```
1  while(1)
2  {
3     if(RB0 == 1)
4     {
5        // If the button is pressed, flash the LED twice/second.
6        RB1 = 1;
7        __delay_ms(250);
8        RB1 = 0;
9        __delay_ms(250);
10    }
11 }
```

Note

The bit-fields could be treated as if they are **bool**ean variables. Which means that the **RB0** checking (**polling**) could have been done with the following LOC without equality check (==1).

```
1 if (RB0)
2 {              ...              }
```

Cross your fingers. Hit the compile button! And we're done with the firmware.

2. Simulation

After connecting everything up in your simulator, you'll have to double-click the MCU chip and point to the firmware code **.hex** file, make sure to simulate your project at the same oscillator frequency used in the code. Otherwise, the timing diagram will be messed-up.

3. Prototyping

You should have created the basic prototyping setup which we've demonstrated previously. If you've any troubles doing so, you can refer to that previous tutorial.

Connect the **RB0** pin to the **button** which is connected in pull-down mode. Connect the **RB1** pin to the

The final running project in case you're curious.

The power of #define directive

When you're dealing with Input/Output post, whatever platform is being used, it becomes fundamentally much more efficient to use the pre-processor **#define** instead of writing directly to bits or registers. Let's consider the following LOC for example.

```
1 #define   RB0   LED1
```

What this directive actually does is telling the compiler before assembling code to replace every single occurrence of **LED1** in your code with **RB0.** Which means that you can switch the LED hooked to RB0 pin ON and OFF in the following manner.

```
1 LED1 = 1;   // ON
```

DeepBlue Menu

and the like. It's ultimately the best way to deal with i/o ports, and it's also substantial for creating libraries

that are easily portable. Let's take just one concrete example to see the real power of the so-

```
3     RC1 = 1;
4     __delay_ms(100);
5     RC1 = 0;
6     __delay_ms(100);
7     RC1 = 1;
8     __delay_ms(200);
9     RC1 = 0;
10    __delay_ms(200);
11    RC1 = 1;
12    __delay_ms(300);
13    RC1 = 0;
14    __delay_ms(300);
15 }
```

This code obviously toggling the logical state of the **RC1** pin for 100ms, 200ms, and 300ms. Well, let's suppose for whatever reason you're forced to leave the RC1 pin and move the (e.g. LED) and hook it to another i/o pin. Let's say you'll move it to **RD1** pin. Now, you'll go to the code and make some modifications at every occurrence for RC1 to replace it with RD1. Which is obviously an unnecessary additional overhead, and the final results will never be consistent nor efficient. However, let's do those modifications.

```
1  while(1)
2  {
3     RD1 = 1;          // First line needed modification
4     __delay_ms(100);
5     RD1 = 0;          // 2nd
6     __delay_ms(100);
7     RD1 = 1;          // 3rd
8     __delay_ms(200);
9     RD1 = 0;          // 4
10    __delay_ms(200);
11
12
13
14
15
      RD1 = 1;          // 5
      __delay_ms(300);
      RD1 = 0;          // 6
      __delay_ms(300);
   }
```

The decision of moving an i/o pin has resulted in roughly 6 lines that needed correction. It's not always the case of course and you may be in a situation in which this issue could leave you with a couple of hundreds LOC waiting to be corrected! Just consider instead if you're using the #define in the following manner.

```
1  #define LED1 RC1
2  ..
3  while(1)
4  {
```

DeepBlue Menu

```
8     __delay_ms(100);
9     LED1 = 1;
```

```
10      __delay_ms(200);
```

This code is typically the exact same equivalent of the first example. Which toggles the RC1 pin for varying time intervals. However, when a change has to be made for the i/o pin we are using, and let's say we're migrating the LED from RC1 to the RD1 pin. The ONLY change that must be made is just the first LOC as follows.

```
1  #define LED1 RD1
```

Pro Tip
Always pick relevant names for the #define pre-processor. Nothing is more painful than irrelevant naming for i/o, variables, functions or whatever. So please, pay attention and invest a couple of seconds of your time thinking about a relevant name that exactly describes your identifier.

Checkout My PIC MCU Programming Video Tutorial

Join Our +25,000 Newsletter Subscribers!

Stay Updated With All New Content Releases. You Also Get Occasional FREE Coupon Codes For Courses & Other Stuff!

Email Address

Subscribe

Embedded Systems, Embedded Tutorials, Microchip PIC

Embedded C, Embedded Systems Tutorials, Firmware, Learn Embedded Systems, Learn Microcontrollers Programming, Microchip PIC Tutorials, Microcontrollers, MPLAB, Tutorials, XC8

< Configuration Bits (Fuses) For Microcontrollers

> Interfacing 7-Segments Display With PIC Microcontrollers

Author

Khaled Magdy

I'm an embedded systems engineer with several years of experience in embedded software and hardware design. I work as an embedded SW engineer in the Automotive industry. However, I still do Hardware design and SW development for DSP, Control Systems, Robotics, Ai/ML, and other fields I'm passionate about. I love reading, writing, creating projects, and Technical training. A reader by day and a writer by night, it's my lifestyle. You can view my profile or follow me via contacts.

17 thoughts on “PIC Microcontrollers Input/Output IO Ports GPIOs Tutorial”

Sigma-Aldrich.
MISSION® esiRNA
Order Now

Lee
October 5, 2018 at 10:28 PM

I think there is a typo on section 2 (Read/Write digital i/o pins). In the table on passing 0 to RB3, it says that RB2 pin is driven low. Should that be RB3?

Reply

Khaled Magdy
October 5, 2018 at 9:09 PM

Yea, you’re right! It’s now fixed. Thanks ^^

Reply

I think the schematic is incorrect for lab 2. The 10k should be pulled down not up, as shown clearly in your video. I tried to implement the schematic and I got reverse logic. Press button = OFF, Not pressed = on.

Reply

Khaled Magdy
October 17, 2018 at 9:31 PM

That's right.. i misconnected that button as you've mentioned. It'll be edited right now. Thanks for your feedback ^^

Reply

Ahmad
May 6, 2019 at 1:09 AM

that was pretty simple and relieving

Reply

Adam Tamimi
June 26, 2019 at 4:58 PM

The preprocessor directive [#include "config.h"] has to be removed, otherwise the program does

————————————————————————————————————

1 error generated.
(908) exit status = 1
nbproject/Makefile-default.mk:106: recipe for target 'build/default/production/Firmware_Code.p1' failed
make[2]: Leaving directory 'C:/Users/otami/OneDrive/Desktop/Test_Project'
nbproject/Makefile-default.mk:90: recipe for target '.build-conf' failed
make[1]: Leaving directory 'C:/Users/otami/OneDrive/Desktop/Test_Project'
nbproject/Makefile-impl.mk:39: recipe for target '.build-impl' failed
make[2]: *** [build/default/production/Firmware_Code.p1] Error 1
make[1]: *** [.build-conf] Error 2
make: *** [.build-impl] Error 2

BUILD FAILED (exit value 2, total time: 658ms)
————————————————————————————————————

Reply

Khaled Magdy
July 3, 2019 at 1:16 AM

Try creating a new project and create the config.h header on your own. It should be generated and saved in the right directory which may resolve your issue

Reply

saideh
November 26, 2019 at 11:01 AM

how use hall effect of Arduino?

very good
please continue

mostafa saleh

Reply

AKshay
June 10, 2020 at 5:18 AM

Why don't you use LAT register as it is more reliable, because IF we set the TRIS bit as output and the same port bit to high, due to some noise or shorting the PORT value will change and cannot be referable. But LAT once a bit is set or cleared, it stays that way
Right?
Please clarify my curiosity…
Thanks

Reply

Mohamad
July 31, 2020 at 9:57 AM

Thank you Mr Khaled for your great tutorial
Unfortunately non the download links is working.

Reply

How did you connect the PIC3 to flash the chip?

Reply

Luca

November 8, 2020 at 12:51 PM

That's my thought exacly. I think he forgot to mention this fundamental aspect. Have a look at the pickit 3 user manual to understand how to connect it properly to the PIC.

Reply

Khaled Magdy

November 8, 2020 at 5:49 PM

Yea! I did really miss to mention this as it would vary from user to another. All pickit programmers do share the same 5 pinout in the same order. And should be easily connected to our prototyping board.
I did also the exact same thing step by step in the course's videos on youtube which is unfortunately not in English.

Reply

Mostafa

October 17, 2020 at 9:49 PM

shouldn't it be

Reply

Khaled Magdy
October 17, 2020 at 9:53 PM

Of course! Thanks for pointing this out. I should check this and edit it. However, I believe that it's written correctly in the downloadable project files. And it also needs to be checked. Thanks, Mostafa!

Reply

jorge
August 18, 2021 at 2:49 PM

The link for the "Prototyping Board Setup" is broken and I cannot figure out how to wire everything based on the picture shown (Wires on upper left cannot be seen and I don't know resistance, capacitance, and oscillation frequency values). Anyone have a link that works or a schematic? I couldn't find anything on the first tutorial either.

Reply

Leave a Reply

Learn PIC Basics

Introduction To Microcontrollers
Creating New MPLAB X Project
How To Set Config. Bits (Fuses)
GPIO Tutorial
Interrupts Tutorial
How To Write ISR Handlers
Timer Modules - Timer Mode
Timer VS Delay
Timer Preloading
External Interrupt Pins
CCP Modules (Capture-Compare-PWM)
PWM Tutorial

Sensors

- HC-SR04 Ultrasonic
- LM35 Temperature Sensor
- MPU6050 IMU

Motors & Actuators

- Unipolar Stepper Motors
- DC Motors + L293D
- Servo Motors - I
- Servo Motors - II
- Servo Motors - III

Display & HMI

- 7-Segment Display
- LCD 16x2 Alphanumeric
- I2C LCD 16x2
- Multi-Digit 7-Segments

Modules

- External EEPROM

Communication & Wireless & IoT

X

ezoic report this ad

Search The Blog

Search

Categories

Select Category

Subscribe To Our Newsletter To Get All New Updates

Email Address

Subscribe

DeepBlue Menu

Categories

- Electronics
  - Electronics Projects
  - Electronics Tutorials
- Embedded Systems
  - Articles
  - Embedded Projects
  - Embedded Tutorials
    - IoT

X

FPGA
Projects Ideas
Tech Reviews

ezoic report this ad

ezoic report this ad

ezoic report this ad

Categories

Select Category

Search The Blog

Search

Resources

STM32 ARM MCUs Programming Course

Embedded Systems - PIC Course

ABOUT DEEPBLUE

DeepBlueMbedded is an educational website where you can find technical content (Articles – Tutorials – Projects – etc..). Mainly on Embedded Systems & ECE Related topics. You'll find also downloadable resources like firmware code examples, schematics, hardware designs, and more.

It's been and will always be a free resource of information. So, please consider supporting this work if possible.

SHARE & SUPPORT

You can always show your support by sharing my articles and tutorials on social networks. It's the easiest way to support my work that costs nothing and would definitely be appreciated!

OR You Can Support Me Through This Link

SUPPORT

DeepBlueMbedded.com is a participant in the Amazon Services LLC Associates Program, eBay Partner Network EPN, affiliate advertising programs designed to provide a means for sites to earn advertising fees by advertising and linking to Amazon.com and eBay.com

You can also check my Full Disclaimer Page For More Information.

Copyright © 2022 DeepBlueMbedded.com . All Rights Reserved.