I/O PORT OPERATIONS IN MSP430

Contents

I/O Operations! Why and what?

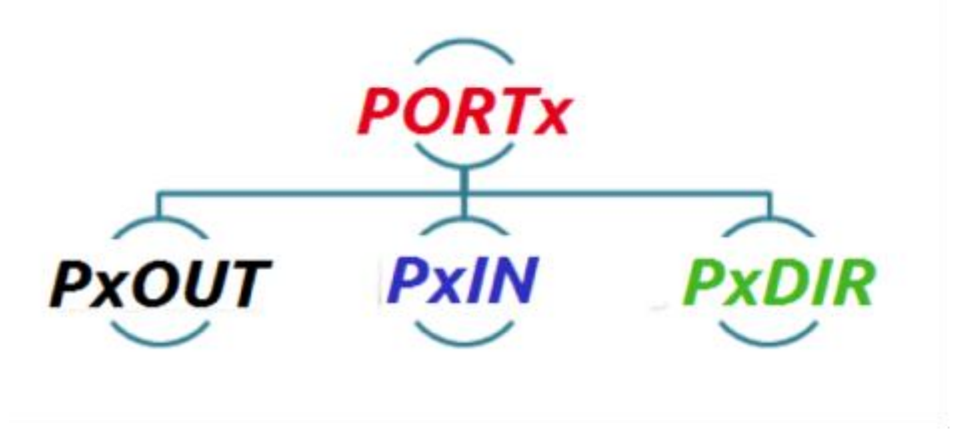
In my previous post, I discussed about the architecture of MSP430 microcontroller. It is a small computer-on-chip just like other microcontrollers. When, we think of a personal computer, we usually get an image of a desktop or laptop computer in our mind. These computers generally have keyboard and mouse as input devices and monitor or screen as output device. Through all these devices, a computer interacts with the outer world. Then, what does a microcontroller uses to interact with outer environment (or users)? The answer is *ports*. All the sensors, and other input-output devices are connected to microcontroller through a set of pins, called ports. If you have worked with AVR, you must be familiar with port names PORTA, PORTB, etc. But Texas Instruments use numeric listing instead of letters. The port name starts as P1, P2 and so on. There is no P0. Each port has eight pins. All pins of a port might not be available for user. Some pins are internally connected to the controller.

Memory Mapping

I mentioned in my previous post that, to access a byte from memory of a computer, there is always an *address*. It means that, to fetch any data or instruction from memory, CPU makes use of address of that data or instruction. Now, the question arises is- *How does the CPU of MSP430 access ports of the controller?* The answer is simple. CPU considers ports as memory registers. Each*port* is a byte in the memory, and each *pin* of the port represents a bit of that register. The registers assigned to the port are called peripheral registers. This is called *Memory Mapping of input-output*. These registers can be read, written and modified. Moreover, arithmetic operations can also be performed. But, how do we decide that whether a port is to used as input or output? For this purpose, there are three types of registers associated with each port.

1. PxIN
2. PxOUT
3. PxDIR

Here x is the port number (remember, P1, P2, etc?). For example, when x is 1, it becomes P1IN, P1OUT and P1DIR. *The keywords IN, OUT and DIR expands to input, output and direction respectively.*



Port Registers in MSP430

Setting P1DIR to one (1, HIGH) configures the pins of port 1 as *output* and clearing the P1DIR register to zero (0, LOW) makes the pins of port 1 as *input*which is also the default. This way, DIR gives direction of operation to the ports. It is similar to the DDRx register of AVR. Each pin or bit of the port can be configured individually. It is not necessary to make all pins of the port behave in same way. Now let us look at the example:

```
P2DIR = 0xB1;        // This is in hexadecimal form


P2DIR = 0b10110001; // Binary representation
```

The above two statements are equivalent to each other. Both can be used to make 0th, 4th, 5th and 7th pin of port 2 as output pin. You would have already noticed, we count from LSB (0th bit) to MSB (7th bit). This is a convention in logic design. Now, after setting the pins as output pin, we can assign values to them in following way.

```
P2OUT = 0x08;        // This is in hexadecimal form


P2OUT = 0b1000;      // Binary representation
```

This will set (to 1) the 4th pin of port 2. If we try to give an output to the pin or port which has not been assigned as output port using DIR, then the value we give is stored in PxOUT register and is passed to the port when it is assigned as output port. Please, keep it in mind that, in general digital logic language, setting a bit means giving logic 1 (or HIGH) to the bit and clearing a bit means giving logic 0 (or LOW) the bit.

C Programming Language

In my earlier post, I mentioned that MSP430 can be programmed using assembly language and C programming language. Most of us are familiar with C programming language for obvious reasons. Its simplicity, ease of understanding and programming, user defined types, flexibility and data structures make it much better than assembly language. Thus, we will be programming the MSP430 using C language only. This language is used by many other microcontrollers as well because it is efficient and easy to debug and supports the modern controller's architecture. Here, I am assuming that if you are aware of basic C programming language. If not, please stop here, read a little about C, and then come back! Now, let us look at some aspects of the C programming language which we will deal with in future posts. NOTE: This section does NOT teaches you how to program using C. You should be familiar with it by now. If not, take a break! This section teaches how to use C specific to embedded systems requirement.

Logical and Bitwise Operators

A lot of people easily get confused with these two kinds of operators. Let me explain you with examples one by one.

Logical Operators

Logical operator, as the name says logical, gives only one bit logical output. Consider the following example–

```
A = 0x11;

B = 0x00;

C = 0x01;

D = 0x00;
```

```
E = A && B;     // E = 0

F = A && C;     // F = 1

G = A || B;     // G = 1

H = A || C;     // H = 1

I = B || D;     // I = 0
```

Here, `&&` and `||` are the logical operators and are used to perform `logical AND`and `logical OR` operations respectively. For logical operators, result is `ZERO` if and only if–

- for logical AND, any one of the operands is exactly zero.
- for logical OR, both the operands are exactly zero.

For *all* other cases, the result is *always* `ONE`.

Bitwise Operators

Bitwise operator, as the name says, does bitwise operation. Look at the following examples.

```
A = 0x4B;      // A = 0b1001011

B = 0x19;      // B = 0b0011001

C = 0x2A;      // C = 0b0101010

D = A & B;     // D = 0x09 = 0b0001001

E = A | B;     // E = 0x6B = 0b1101011
```

Here, `&` and `|` are `bitwise AND` and `bitwise OR` operators respectively. The result is the bit by bit result of the `AND` and `OR` operations on the operands.

Accessing Individual Bits (Masking)

Byte is generally used as the smallest entity in microcontrollers. However, many times we need to test or modify individual bits of the controller. There are many ways by which a user can access the bits or the port. We always tend to use the optimized code for a program, thus it is always better to declare a BIT before starting the main code of program. You can declare it globally outside the `main()` as–

```
int BIT3 = 0b00001000;
```

Or else, a better way would be to define it as a macro (recommended) as follows–

```
#define BIT3 0b00001000
```

It's not necessary to have a name like BIT3, but it is always advised to name it something that can be related to easily. And make sure the name you choose is not a reserved keyword. This selection of individual bits is called ***masking***. Now, if we do bitwise AND operation `P13 = P1IN & BIT3`, it will make all pins of P1 zero except for bit 3 because `x & 0 = 0` and `x & 1 = x`. Similarly, `x | 0 = x` and `x | 1 = 1`. So, `P2OUT = P2OUT | BIT3` will set the bit 3 of P2, leaving the values of other bits unchanged. Thus, we can use the following way to read and to write a bit:

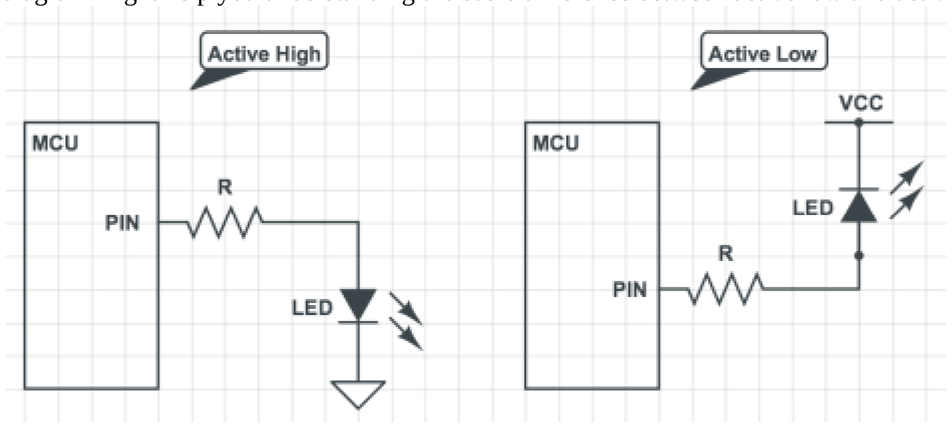
```
if( ( P1IN & BIT3 ) == 0 ) //  Reading the BIT3 of port 1, checking if it is logic low
{
    P2OUT = P2OUT | BIT3;  // Setting up BIT3 of P2. i.e. writing the individual bit
    // this can also be written as P2OUT |= BIT3
}

else
{
    P2OUT &= ~BIT3;        // Clearing the BIT3 of P2, if BIT3 of P1 is high.
}
```

If we observe the above code, it is nothing more than the operation of a switch which is connected to BIT3 of P1 and LED which is connected to BIT3 of P2.

Now, let us look at the above code once again line by line–

- The first statement uses `if` condition. If the condition is true i.e. if BIT3 of P1 is low, the statements within the curly braces `{}` will be executed. In other words, if a switch connected to P1.3 is closed i.e. P1.3 is grounded, it will make P2.3 HIGH. If an LED is connected to P2.3, it will NOT glow (assuming LED is connected in active low configuration). The LED will glow only if the PIN is logically LOW. The following diagram might help you understanding the basic difference between active low and active high pin.



Active High and Active Low

- On the contrary, if the condition of `if` statement is false, it will execute the `else` statement and make P2.3 logic LOW, and this time the LED connected to P2.3 will glow.

What if we want to toggle the output bit every time?? Then in that case, we can use following statement (why?).
Hint- `^` stands for XOR operation.

```
P2OUT = P2OUT ^ BIT3;


P2OUT ^= BIT3;   // Same as above
```

Data Types

While using C or C++ language, we use `int` or `char`. Similarly, for MSP430 (and other microcontrollers as well), a 16-bit microcontroller, there are few other types of commonly used data types. `char` and `int` can be used for declaration of counter variable. Their sizes are 1 byte and 2 bytes respectively. `int8_t` and`uint8_t` are used to declare signed and unsigned 8-bit integers respectively.`int16_t` and `uint16_t` are used to declare signed and unsigned 8-bit integers respectively. These `typedef` variables are also available in C, defined in`stdint.h`. Refer to this document for more information.

Why do we need special data types?

This is something which every rookie wonders about. The answer is necessity. When you are writing a program to be executed on your laptop/computer, you are least bothered about how much size it occupies. For instance, when you declare a variable as `int`, do you know how much space it occupies in memory? Is it 2 bytes or 4 bytes? What's the size of `unsigned short int` – 1 byte, 2 bytes or 4 bytes? How about `long int` – 4 bytes or 8 bytes? You don't know! Frankly speaking, it depends upon the compiler and your implementation. The C standardspecifies only minimum size of `int` data types. For instance, the C standard says that an `int` data type must have a minimum size of 16 bits (2 bytes). This means that, depending upon the compiler, it could be 2 bytes, 4 bytes or even 8 bytes! In embedded systems, you have to deal closely with the hardware. For example, you have to implement an 8-bit counter, you which, you need to count only up to 8 bits. Can you declare your variable as `unsigned short int` and depend upon your compiler to accurately store it as 8 bits? Or for example, your microcontroller has an ADC which converts analog input signals to digital with a precision of 10 bits. Could you declare your variable as `int` and wait for compiler to do the conversion for you? Heck NO!
You need fixed-width integer sizes, wherein you know exactly how much size your variable is gonna occupy! These are defined in stdint.h C header file. Whenever you use uint8_t data type, your variable can hold unsigned values of 8 bit (1 byte) sizes only, no matter which compiler compiles it! So if you define your variable as`int32_t` rather than `long int`, you know that its gonna occupy 4 bytes, no matter what, whereas in the latter case, it could have been either 4 bytes or 8 bytes depending upon the compiler.
As a kid when you would be in a candy store, you must have heard your mom saying to you "take only what you need". The same goes here. For example, you need a counter to count up to 200. What would you define it as? Would you define it as `int`, or `short int`, or `unsigned short int`, or `int8_t`, or `uint8_t`, or`uint16_t`, or what? Remember that `int` and `short int` are at least 16 bits wide. But you simply need to count up to 200, which is only 8 bits. This is where take-only-what-you-need concept chips in. If you can count perfectly using an 8-bit variable, why do you want to take a 16-bit one? It's a waste of space, and you should know that in embedded systems, memory is very limited. So ideally, one's choice would be `uint8_t` (why not `int8_t`?). But these days, compilers have also become smart. They know how to optimize your code and cut down the size of variables as and often required.

Source: http://maxembedded.wordpress.com/2013/12/26/io-port-operations-in-msp430/