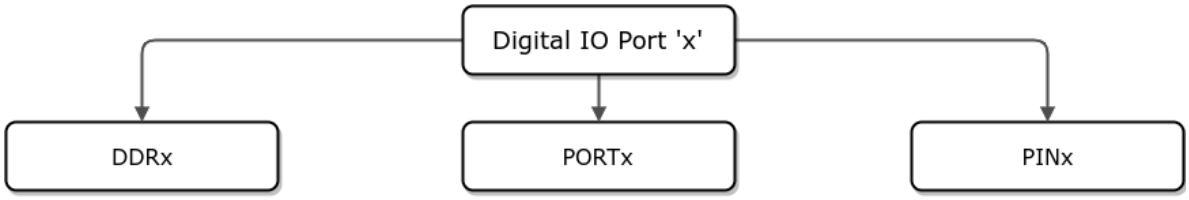
Some points on Atmega328 and AVR C Coding-

Each digital I/O port on Atmega328 is associated with 3 registers-



Since Atmega328 is an 8-bit microcontroller, these registers are 8-bit wide. 'x' can be A, B, C or D. Each bit of these registers corresponds to a specific pin on the microcontroller. Don't confuse the word 'Pin' with the register 'PINx'. When I say 'Pin', I mean the physical pin present on the microcontroller in which you insert wires. A port (in this case, A, B, C or D) is basically a collection of pins.

a) **DDRx:** Stands for "Data Direction Register." Before we can use the pin for any kind of interfacing, we have to tell the microcontroller whether it is an OUTPUT pin or an INPUT pin. This is the register to specify whether a pin should act as input or output-

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0

DDRB Register

Take an example of Data Direction Register of Port B (DDRB). Each bit in this register corresponds to a specific pin. *If a pin in DDRB is set as '1', it'll behave as an Output Pin whereas if it is set as '0', it'll behave as an Input Pin.*

Let's say that I've interfaced my LED to PB2 pin of Atmega328 which happens to be the digital pin 10 of the Arduino. Let's see how we can set this pin as output-

```
DDRB = B00000100;//equivalent to 0x04 in hex
```

So we see that PB2 is now ‘1’ which is what we wanted. But in the above statement, we are also changing every other pin to ‘0’. We are only concerned with PB2 for now and should only change its state. This is where the bitwise operators in C come in handy.

```
DDRB |= B00000100;
```

The bitwise OR operation is used for setting specific bits to ‘1’; PB2 in this case. Other bits won’t be affected.

Now let’s say I’ve connected a sensor or a simple pushbutton to PB0. It has to be configured as an input.

```
DDRB = B00000000; //0x00 in hex
```

This is again not the friendly way of coding. We just want to change PB0 to 0 (input) and the rest should remain unchanged.

```
DDRB &= B11111110; //0xFE in hex
```

Now, only PB0 will be affected. Instead of 0xFE, we can write ~0x01 which is the same thing but looks better.

```
DDRB &= ~0x01; //set PB0 as INPUT
```

b) **PORTx:** Now that we’ve configured our pin in DDRB, let’s now look at the PORTx register. If a pin is configured as Output, we can write a value to the pin by setting it either 0 or 1 in the PORTx register. If we write a ‘1’ for a specific pin in PORTx, that pin goes *HIGH (or 5V)*. Likewise, if I write ‘0’, that particular pin goes *LOW (or 0V)*.

In the previous example, we assumed an LED to be connected to PB2. Now we want to light it up, i.e, give 5V to PB2. So, in the PORTB register, set the bit for PB2 to ‘1’-

```
PORTB |= 0x04; //or B00000100 in binary
```

Now I want the LED to turn off. So I simply set PB2 to 0 in the PORTB register-

```
PORTB &= ~0x04;
```

So we've seen what PORTx register is used for if a pin is configured as output. But what if I've configured a pin as input. Will the PORTx register still be useful? After all, I can't write to an input port; I can only read from it! As it turns out, the answer is yes. *If you set a bit as '1' in PORTx register for a pin that is configured as input, you'll enable internal pullup on that pin. Writing a '0' will disable the internal pullup (which is the case by default).*

c) PINx: This register is used to read values from pins configured as input. In the example taken at the beginning, we had assumed a switch to be connected to PB0. Let's read the state of PB0 and store it in a variable-

```
unsigned long SW = PINB;
```

By doing this, we ended up reading not just PB0 but all other pins too. Since we are only concerned with PB0, make all others '0'.

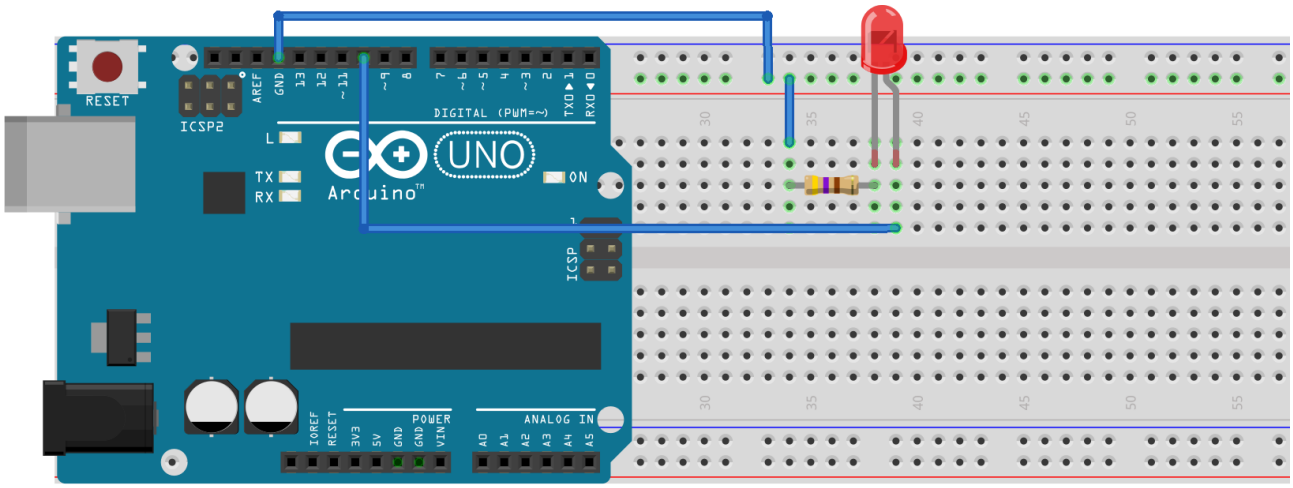
```
unsigned long SW = (PINB & 0x01);
```

The bitwise AND operation here will make others 0 and give us the state of PB0 only. Note that we are not assigning anything to PINB. We are just reading the states of all pins and neglecting those whose states are not required.

The result of `(PINB & 0x01)` can either be *0x01* (if PB0 is 1 or HIGH) or *0x00* (if PB0 is 0 or LOW). The *if-else* conditions should be written accordingly.

Interfacing an LED with a Microcontroller-

The LED is interfaced with PB2 (or digital pin 10 of the Arduino)-

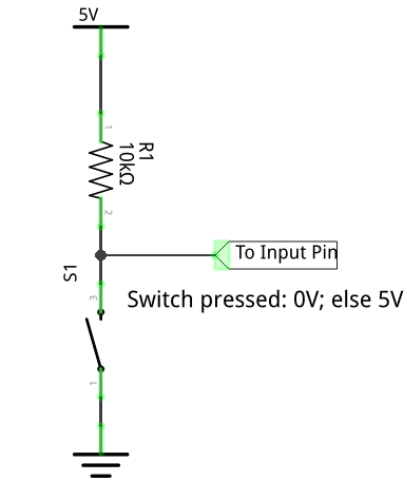


A current-limiting resistor (330 ohm – 470 ohm) has to be put in series or else the LED may overheat and burn.

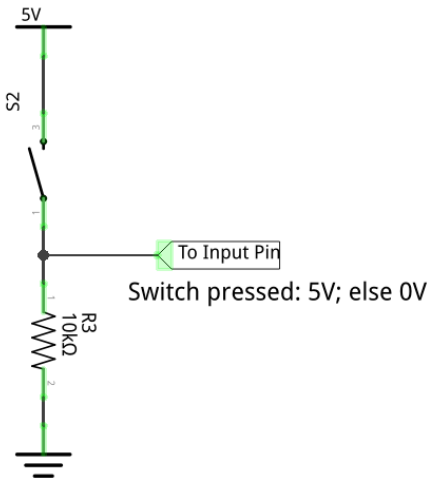
Interfacing a Pushbutton with a Microcontroller-

You have the liberty of putting either a physical resistor in series with the pushbutton (in pull-up or pull-down configuration) or omitting the physical resistor by using the internal pull-up resistor of the I/O port.

1. **External Pull-Up and Pull-Down-**



External Pull-Up

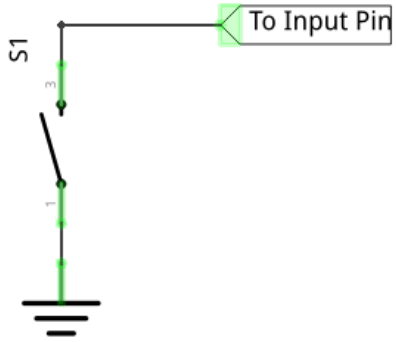


External Pull-Down

In this case, you're not utilizing the internal pullup configuration since you've put a physical 10k resistor. So after PB0 is set as input (through DDRB), you will not write anything to PORTB.

2. **Internal Pull-Up**-

The internal pull-up is same as external pull-up except the fact that the pull-up resistor is present inside the microcontroller itself along with a Vcc connection. You have to put one leg of the switch to ground and the other directly to the input port-

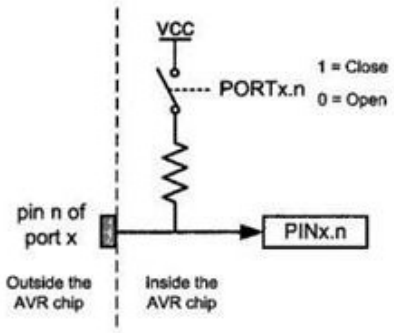


Connections for Switch in Internal Pullup

As specified earlier, to enable pullup on an Input pin (say PB0 in this case), we write '1' to the PORTx register-

```
DDRB &= ~0x01; //set PB0 as INPUT
PORTB |= 0x01; //Internal pullup enabled
```

This is what happens inside the AVR chip-



Pull-up Resistor