Table of contents

Source: Microchip Introduction(https://microchipdeveloper.com/8avr:ioports)

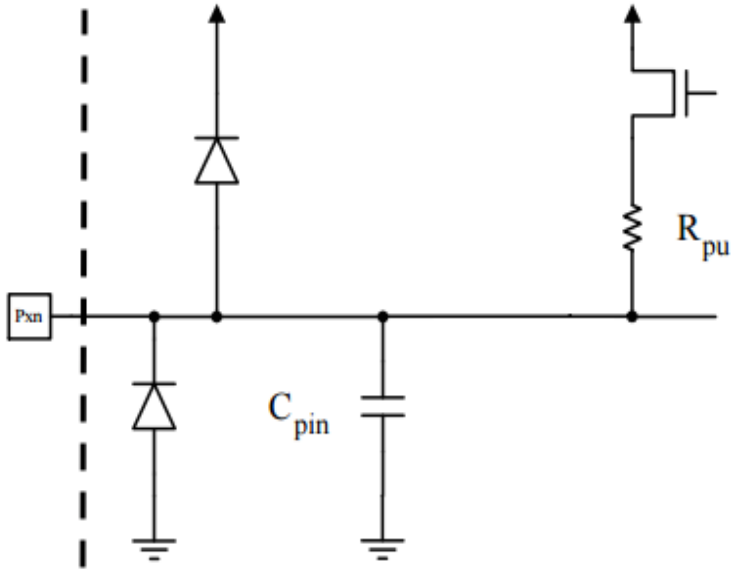
Digital Input/Output Ports on AVR

AVR® 8-bit microcontrollers control applications through their digital Input and Output (I/O) pins. These pins can monitor any voltage present as a high impedance input and supply or sink current as a high or low voltage digital output. These pins are usually organized in groups of eight and referred to as a port. The AVR uses the alphabet to name these ports, for example: PortA, PortB, etc. The pins of PortA are referred to as PA0 - PA7.

Overview

All AVR ports have true Read-Modify-Write functionality when used as general digital I/O ports. This means that the direction of one port pin can be changed without unintentionally changing the direction of any other. The same applies when changing drive value (if configured as an output) or enabling/disabling of pull-up resistors (if configured as an input). Each output buffer has symmetrical drive characteristics with both high sink and source capability.

The pin driver is robust enough to drive LED displays directly. All port pins have individually selectable pull-up resistors with a supply-voltage invariant resistance. All I/O pins have protection diodes to both VCC and Ground as indicated in the figure.

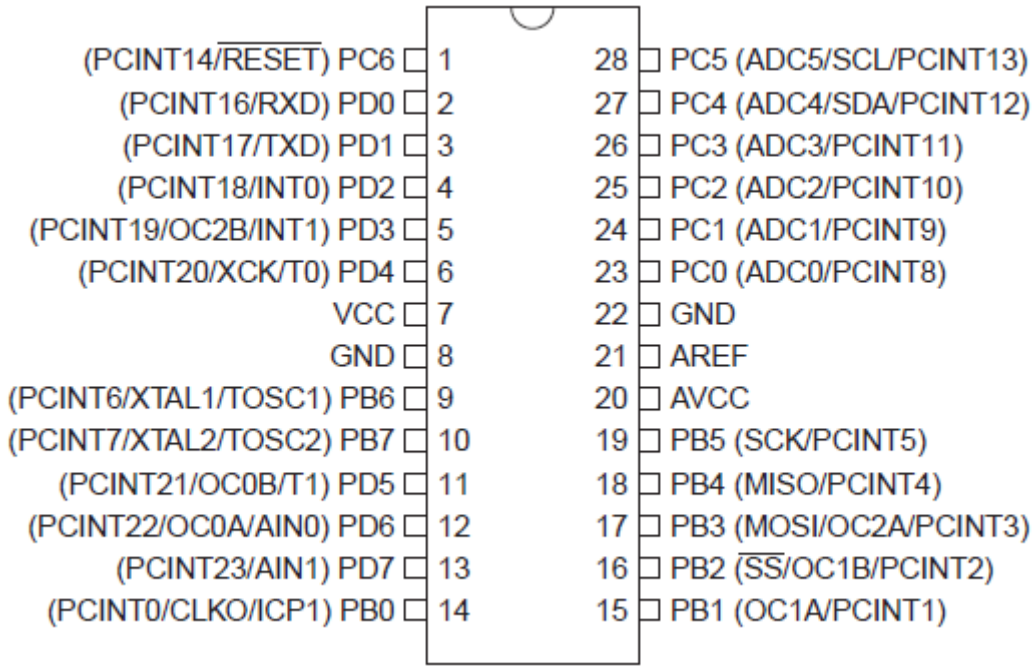


Current sinking and sourcing

- Each I/O pin can sink or source a maximum current of 40mA.
- Although each pin can sink or source 40mA current, it must be ensured that the current sourced or sinked from all the ports combined, should not exceed 200mA.

A LED requires roughly 16mA to light.

I/O Digital Pins (Registers)



ATMega 328 I/O Ports: PORTB (8 bits), PORTC (7 bits), PORTD (8 bits)

Each port consists of three registers:

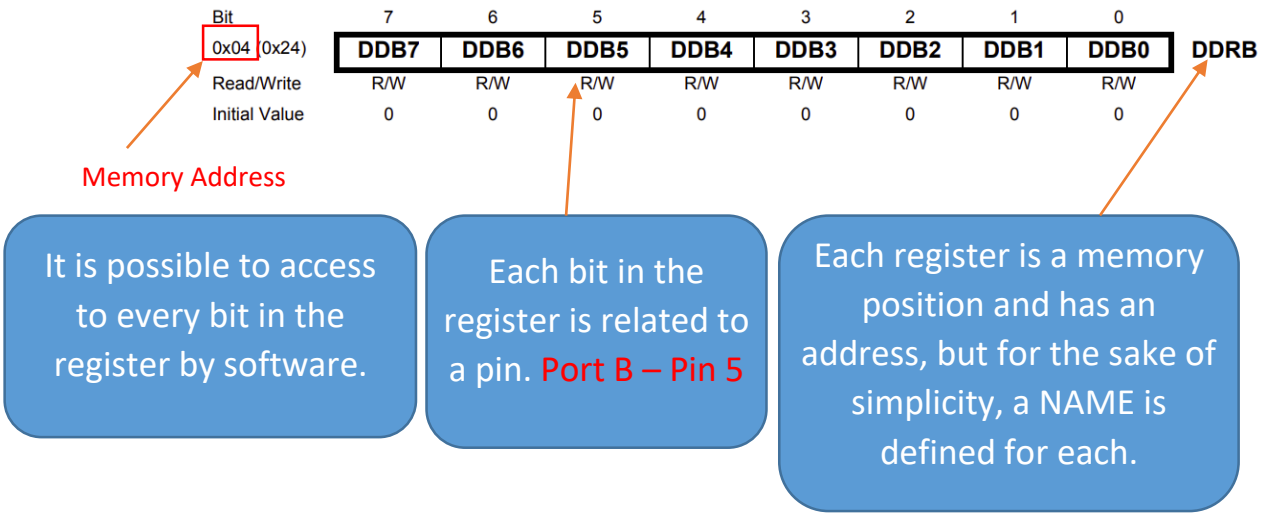
DDRx – Data Direction Register
PORTx – Pin Output Register
PINx – Pin Input Register

where x = Port Name (B, C or D)

Pin direction: DDRxn

The DDxn bits in the DDRx Register select the direction of this pin. If DDxn is written to '1', Pxn is configured as an output pin. If DDxn is written to '0', Pxn is configured as an input pin (default).

14.4.3 DDRB – The Port B Data Direction Register



DDRB = 0b00100000; // Pin 5 of port B is configured as "Output" (Binary format)

or

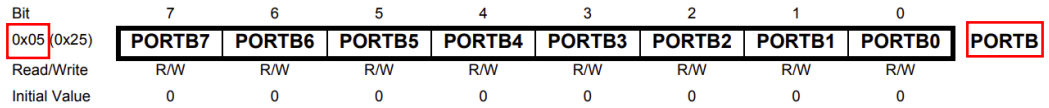
DDRB = 0x20; // Pin 5 of port B is configured as "Output" (Hexadecimal format)

Write pin: PORTxn

The PORTxn bits in the PORTx register have two functions. They can control the output state of a pin and the setup of an input pin.

If the pin was configured as an Output:
If a '1' is written to the bit when the pin is configured as an output pin, the port pin is driven high. If a '0' is written to the bit when the pin is configured as an output pin, the port pin is driven low.

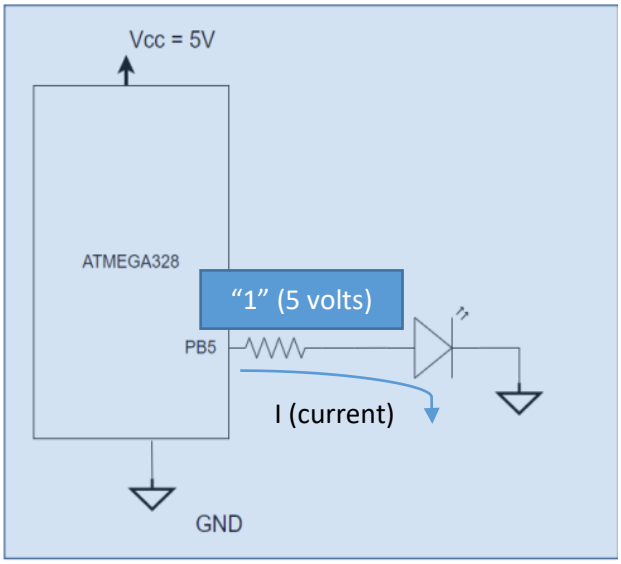
14.4.2 PORTB – The Port B Data Register

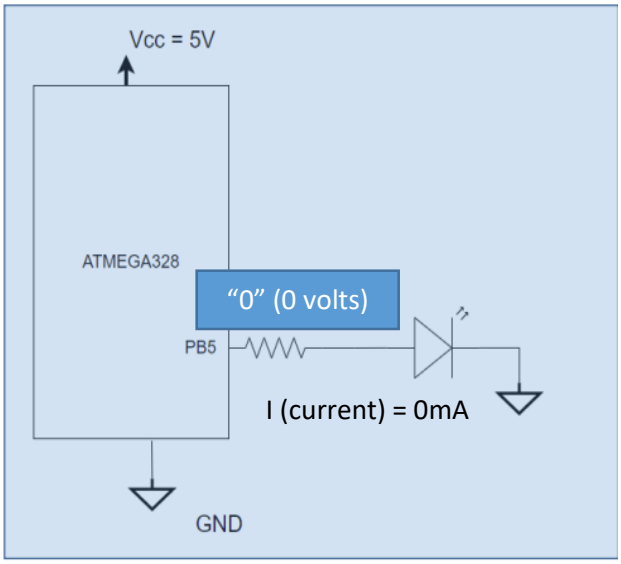


POTRB = 0b00100000; // Write '1' (5volts) in pin 5 of PORTB (Binary format)

or

PORTB = 0x20; // Write '1' (5volts) in pin 5 of PORTB (Hexadecimal format)





If the pin was configured as an Input:

If a '1' is written to the bit when the pin is configured as an input pin, the pull-up resistor is activated. If a '0' is written to the bit when the pin is configured as an input pin, the port pin is tri-stated.

Read pin: PINxn

The PINxn bits in the PINx register are used to read data from port pin. When the pin is configured as a digital input (in the DDRx register), and the pull-up is enabled (in the PORTx register) the bit will indicate the state of the signal at the pin (high or low).

14.4.4 PINB – The Port B Input Pins Address(1)

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

```
data = PINB; //read PORTB pins into variable data
```

Note: If a port is made an output, then reading the PINx register will give you data that has been written to the port pins.

As a Tri-State Input:

When the PORTx register disables the pull-up resistor the input will be tri-stated, leaving the pin left floating. When left in this state, even a small static charge present

on surrounding objects can change the logic state of the pin. If you try to read the corresponding bit in the pin register, its state cannot be predicted.

These registers determine the setup of the digital inputs and outputs. I/O pins can also be shared with internal peripherals. For example, the Analog to Digital (ADC) converter can be connected to the I/O pin instead of being a digital pin. In this case the I/O pin registers set it up as a tri-state high impedance input.

DRAWBACKS of this notation

By using "=" it is needed write values for all the bits

How to write a bit-mask

"BlinkLED_13.c" file

```
int main(void) {
/* ------------ SETUP ------------------  */
    //  1) Set pin 5 in PORTB as output pin
    DDRB |=(1<<DDB5);

/* ------------ LOOP ------------------
            For blinking LED
    ------------------------------------*/
    while (1) {
    // 1) LED OFF by clearing the pin PB5
        PORTB &=~(1<<PORTB5);
    // 2) Delay of TIME milli-seconds
        _delay_ms(BLINK_TIME);
    // 3) LED ON by setting the pin PB5
        PORTB |=(1<<PORTB5);
    // 4) Delay of TIME milli-seconds
        _delay_ms(BLINK_TIME);
    }
}
```

1.- Creating a bit-mask (Configuring the pin PB5 as output)

Left-shift 'n' bits being 'n' the position of the bit 3 in PORTD register

```
DDRB = (1<< DDB5) // Pin 5 in PORTB configured as output
```

"avr/io.h" includes the "iom328p.h" which defines the name of each bit of each register making it equal its position

p.e.: #define PORTB5 5

In this way a bit-mask is created in order to assert a bit in a register.

"00000001" is shifted '5' positions to left

DDRB = "00100000"

2.- OR (|) operation with bit-mask and a register (Configuring the pin PB5 as output)

```
DDRB |= (1<< DDB5);
```

It is the same that

```
DDRB = DDRB | "00100000"
```

Whatever the **bit5** is, the bit-mask with OR operation assures **bit5** is '1' in **DDRB,** remaining the values in the other bits.

3.- Complemented bit-mask (Writing '0' in Pin B5 to turn-off the LED)

```
(~(1<<PORTB5));
```

The bit-mask is created as usual

"00000001" is 5-bit left shifted to get "00100000"

And then the mask is complemented (~)

"11011111"

4.- AND (&) operation with a complemented bit-mask and a register. (Writing '0' in Pin B5 to turn-off the LED)

```
PORTB &= (~(1<<PORTB5));
```

It is the same that

```
PORTB = PORTB & 0b11011111
```

Whatever the bit5 value is, a bit-mask with AND operation assures bit5 is '0' in PORTB keeping the values in the remaining bits (pin).

5.- AND (&) operation with a bit-mask. (Writing '1' in Pin B5 to turn-on the LED)

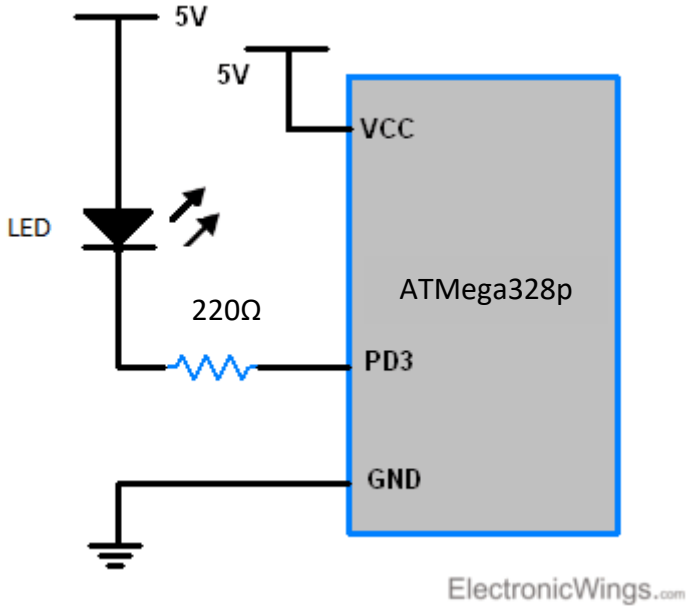
```
PORTB |= (1<<PORTB5);
```

It is the same that

```
PORTB = PORTB | 0b11011111
```

Exercise 1

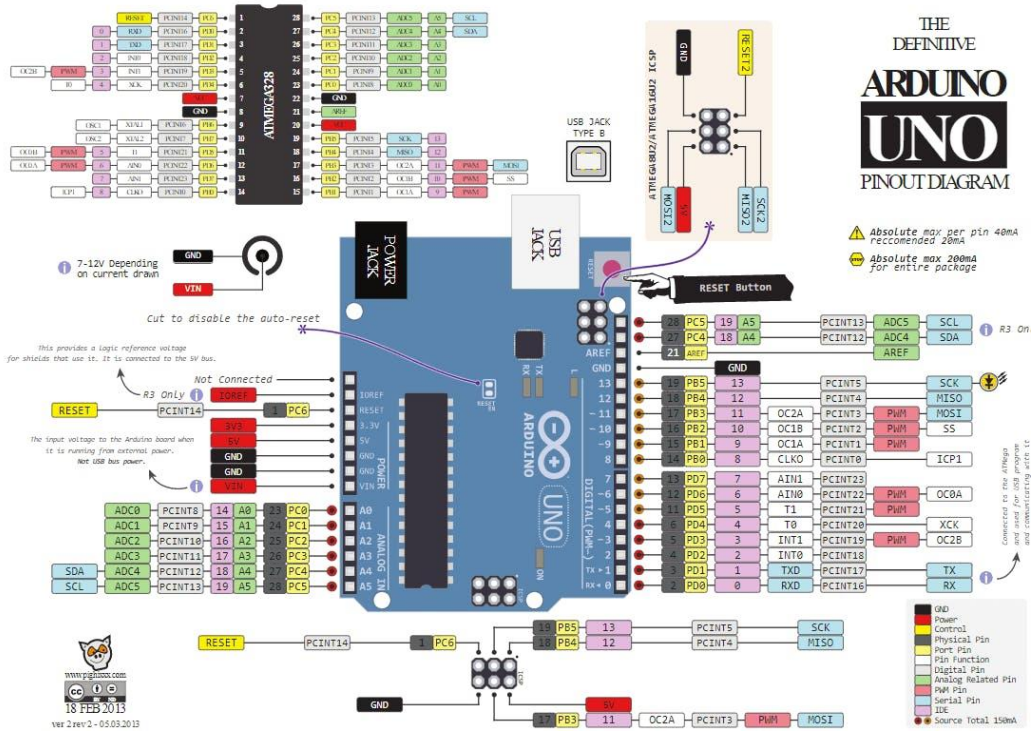
Let's write a code to program pin 3 of Port D as an output pin and use it to drive a LED. We will toggle the LED with some delay.



Circuit Diagram For Example 1

As shown in above figure, the LED is connected to pin 3 of Port D. The anode of the LED is connected to 5V.

(Note: Use the below picture to know the corresponding pin in Arduino board.)



Most of the LEDs require around 10mA current to glow properly. If current greater than maximum rated current of the LED flows through it, the LED will get permanently damaged. The maximum current rating for LEDs is usually around 20-30mA. Refer the datasheet of the LED you are using for this rating.

Absolute Maximum Ratings: (Ta=25℃) .

ITEMS	Symbol	Absolute Maximum Rating	Unit
Forward Current	I_F	20	mA
Peak Forward Current	I_{FP}	30	mA
Suggestion Using Current	I_{su}	16-18	mA
Reverse Voltage (VR=5V)	I_R	10	uA
Power Dissipation	P_D	105	mW
Operation Temperature	T_{OPR}	-40 ~ 85	℃
Storage Temperature	T_{STG}	-40 ~ 100	℃
Lead Soldering Temperature	T_{SOL}	Max. 260℃ for 3 Sec. Max. (3mm from the base of the expoxy bulb)	

Absolute Maximum Ratings: (Ta=25℃)

ITEMS	Symbol	Test condition	Min.	Typ.	Max.	Unit
Forward Voltage	V_F	I_F=20mA	1.8	---	2.2	V
Wavelenength (nm) or TC(k)	$\Delta\lambda$	I_F=20mA	620	---	625	nm
*Luminous intensity	I_V	I_F=20mA	150	---	200	mcd
50% Viewing Angle	$2\theta 1/2$	I_F=20mA	40	---	60	deg

In order to limit the current flowing through the LED, we connect a resistor in series with it.

Assuming that the LED has a 2V forward voltage drop, and that 16mA current is sufficient for the LED to glow properly, the value of resistance required can be calculated as follows:

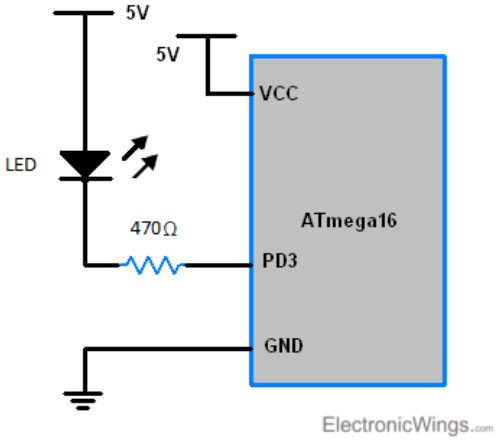
$$R^1 = \frac{VoltageSupply - ForwardVoltageDropLED}{CurrentFlowingThroughLED}$$

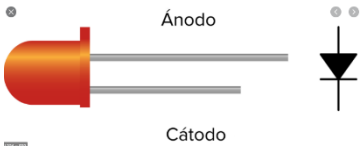
$$R = \frac{5V - 2V}{16mA}$$

$$R = 0{,}1875\ K\Omega = 187{,}5\ \Omega$$

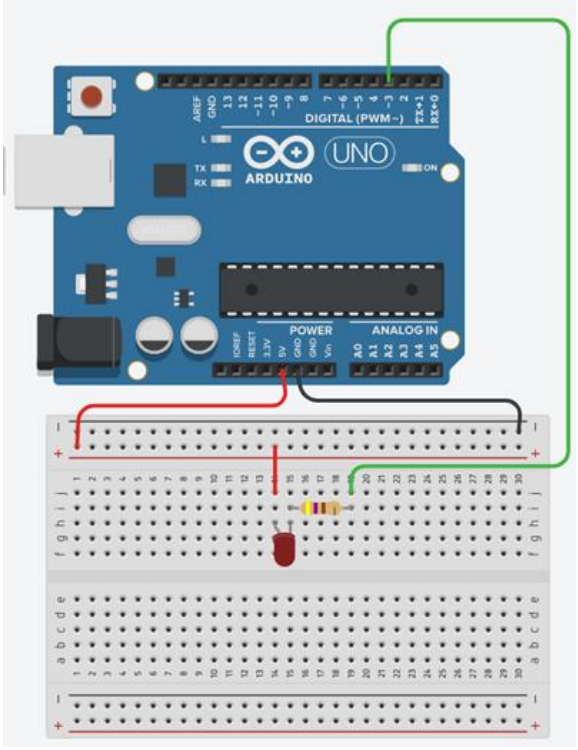
Therefore, the nearest standard value of resistor available is 220Ω. So, that value is used in the circuit. (Hook up the circuit by using a protoboard).

[1] Kirchoff Voltage Law and Ohm Law are applied in order to obtain the shown equation.





Source: Khan Academy



Exercise 2

Connect **four green LED** to the LSB four pins in the port D (PD3-PD0). Connect the LED to light them on when ATMEGA328 send a digital "1".

a) Write a code to send to the LED the values 0 to F in binary format (4 bits). *Make sure the remaining bits in port D are not modified.*

- *Analyse the problem and determine:*
  - *What are the parameters in this project? Type a name for them and create the "PARAMETERS.h" file.*
- *Create a "main.c" file based in a previous one. Try to add comments explaining each step in the algorithm.*

b) Modify the code to show a counter from 0 to 9 (binary).

*c) Change the LEDs to the ports PD5-PD2 and add a **red LED** in the pin PD7. This LED must always keep ON along the execution. Modify the code to get the same function than before.*

Logic levels

Source: (Sparkfun)

What is a Logic Level?

We often refer to the two states in a digital circuit to be ON or OFF. Represented in binary, an ON translates to a binary 1, and an OFF translates to a binary 0. There are several different technologies that have evolved over the past 30 years in electronics to define the various voltage levels.

Logic 0 or Logic 1

Digital electronics rely on binary logic to store, process, and transmit data or information. Binary Logic refers to one of two states -- ON or OFF. This is commonly translated as a binary 1 or binary 0. A binary 1 is also referred to as a HIGH signal and a binary 0 is referred to as a LOW signal.

The logic level describes the voltage level for each state. Manufacturers of chips generally define these in their spec sheets. The most common standard is TTL or Transistor-Transistor Logic.

TTL Logic Levels

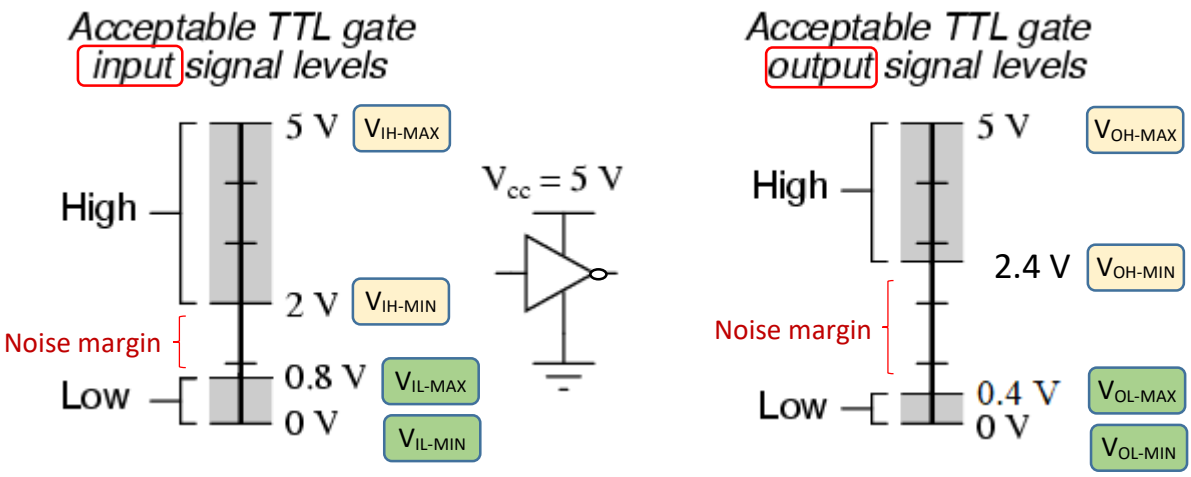
A majority of systems we use rely on either 3.3V or 5 V TTL Levels. TTL is an acronym for Transistor-Transistor Logic. It relies on circuits built from bipolar transistors to achieve switching and maintain logic states. Transistors are basically electrically controlled switches. For any logic family, there are a number of threshold voltage levels to know. Below is an example for standard 5V TTL levels:

V_{OH} -- Minimum OUTPUT Voltage level a TTL device will provide for a HIGH signal.

V_{IH} -- Minimum INPUT Voltage level to be considered a HIGH.

V_{OL} -- Maximum OUTPUT Voltage level a device will provide for a LOW signal.

V_{IL} -- Maximum INPUT Voltage level to still be considered a LOW.

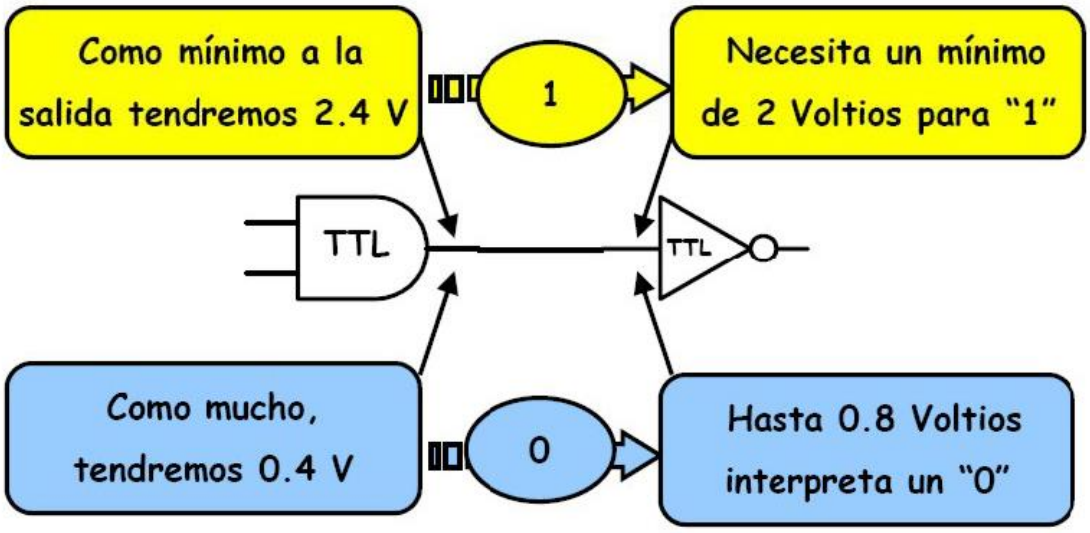


You will notice that the minimum output HIGH voltage (V_{OH}) is 2.4 V. Basically, this means that output voltage of the device driving HIGH will always be at least 2.4 V. The minimum input HIGH voltage (V_{IH}) is 2 V, or basically any voltage that is at least 2 V will be read in as a logic 1 (HIGH) to a TTL device.

You will also notice that there is cushion of 0.4 V between the output of one device and the input of another. This is sometimes referred to as **noise margin.**

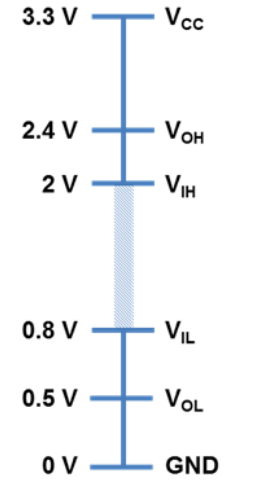
Likewise, the maximum output LOW voltage (V_{OL}) is 0.4 V. This means that a device trying to send out a logic 0 will always be below 0.4 V. The maximum input LOW voltage (V_{IL}) is 0.8 V. So, any input signal that is below 0.8 V will still be considered a logic 0 (LOW) when read into the device.

What happens if you have a voltage that is in between 0.8 V and 2 V? This range of voltages is undefined and results in an invalid state, often referred to as floating. If an output pin on your device is "floating" in this range, there is no certainty with what the signal will result in. It may bounce arbitrarily between HIGH and LOW.



3.3 V CMOS Logic Levels

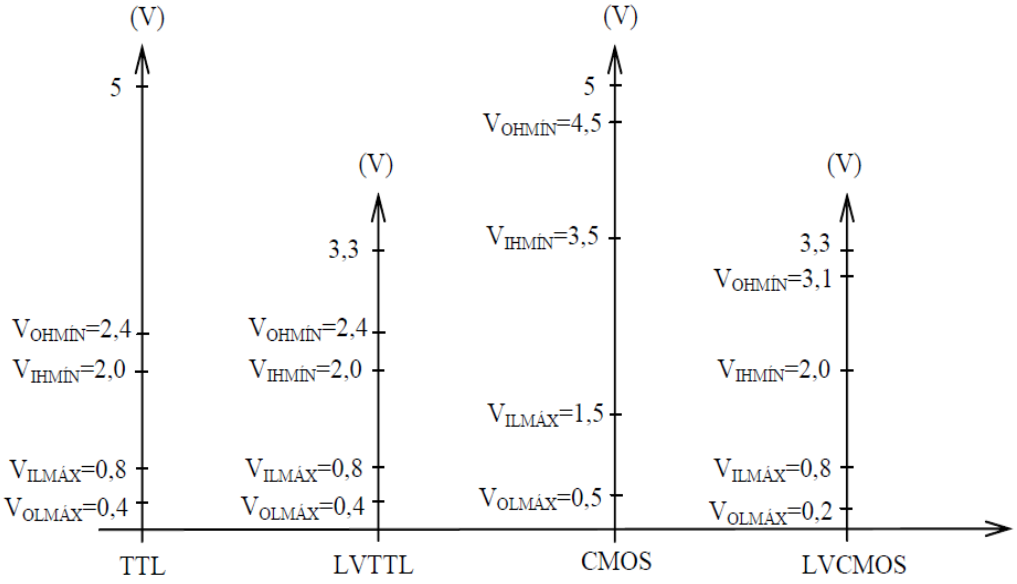
As technology has advanced, we have created devices that require lower power consumption and run off a lower base voltage (V_{cc} = 3.3 V instead of 5 V). The fabrication technique is also a bit different for 3.3 V devices that allows a smaller footprint and lower overall system costs.



3.3 V Logic Families
(based on 74LVT04 Hex Inverter)

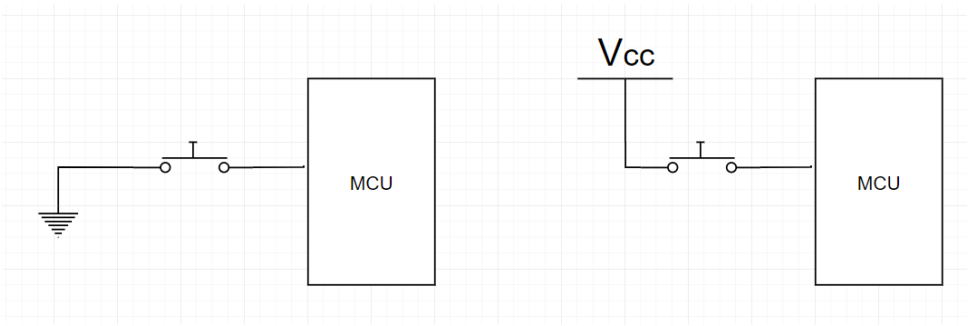
In order to ensure general compatibility, you will notice that most of the voltage levels are almost all the same as 5 V devices. A 3.3 V device can interface with a 5V device without any additional components. For example, a logic 1 (HIGH) from a 3.3 V device will be at least 2.4 V. This will still be interpreted as a logic 1 (HIGH) to a 5V system because it is above the V_{IH} of 2 V.

A word of caution, however, is when going the other direction and interfacing from a 5 V to a 3.3 V device to ensure that the 3.3 V device is 5 V tolerant. The specification you are interested in is the *maximum* input voltage. On certain 3.3 V devices, any voltages above 3.6 V will cause permanent damage to the chip. You can use a simple voltage divider (like a 1KΩ and a 2KΩ) to knock down 5 V signals to 3.3 V levels or use one of our logic level shifters.

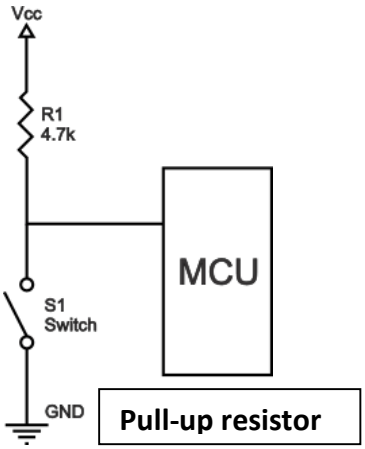


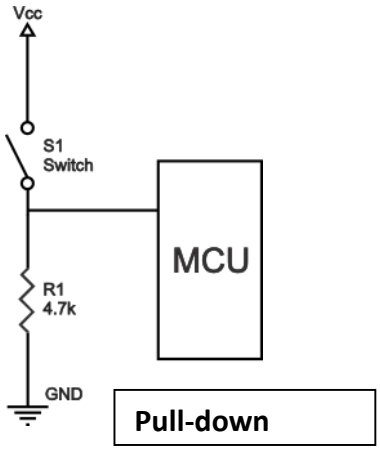
What is a Pull-up/Pull-down Resistor?

In electronic logic circuits, a pull-up resistor or pull-down resistor is a resistor used to ensure a known state for a signal. It is typically used in combination with components such as switches, push-buttons and so on. When the switch is closed, it creates a direct connection to ground or VCC, but when the switch is open, the rest of the circuit would be left floating (i.e., it would have an indeterminate voltage).



For a switch that connects to ground, a pull-up resistor ensures a well-defined voltage (i.e. VCC, or logical high) And, for a switch that connects to VCC, a pull-down resistor ensures a well-defined ground voltage (i.e. logical low) when the switch is open.





ATmega328p has internal pull-up resistors. We can activate it to make pin status HIGH when the pushbutton is not pressed.

14.2.1 Configuring the Pin

Each port pin consists of three register bits: DDxn, PORTxn, and PINxn. As shown in "Register Description" on page 100, the DDxn bits are accessed at the DDRx I/O address, the PORTxn bits at the PORTx I/O address, and the PINxn bits at the PINx I/O address.

The DDxn bit in the DDRx Register selects the direction of this pin. If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configured as an input pin.

If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated. To switch the pull-up resistor off, PORTxn has to be written logic zero or the pin has to be configured as an output pin. The port pins are tri-stated when reset condition becomes active, even if no clocks are running.

If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero).

Source: ATMEGA320 Datasheet (Microchip)

OPTIONAL CONTENT

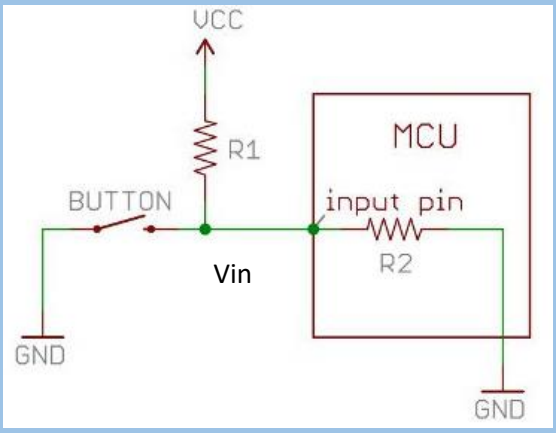
A resistor with low resistance is often called a "strong" pull-up or pull-down. When the circuit is open, it will pull the output high or low very quickly but will draw more current.

A resistor with relatively high resistance is called a "weak" pull-up or pull-down. When the circuit is open, it will pull the output high or low more slowly, but will draw less current.

OPTIONAL CONTENT

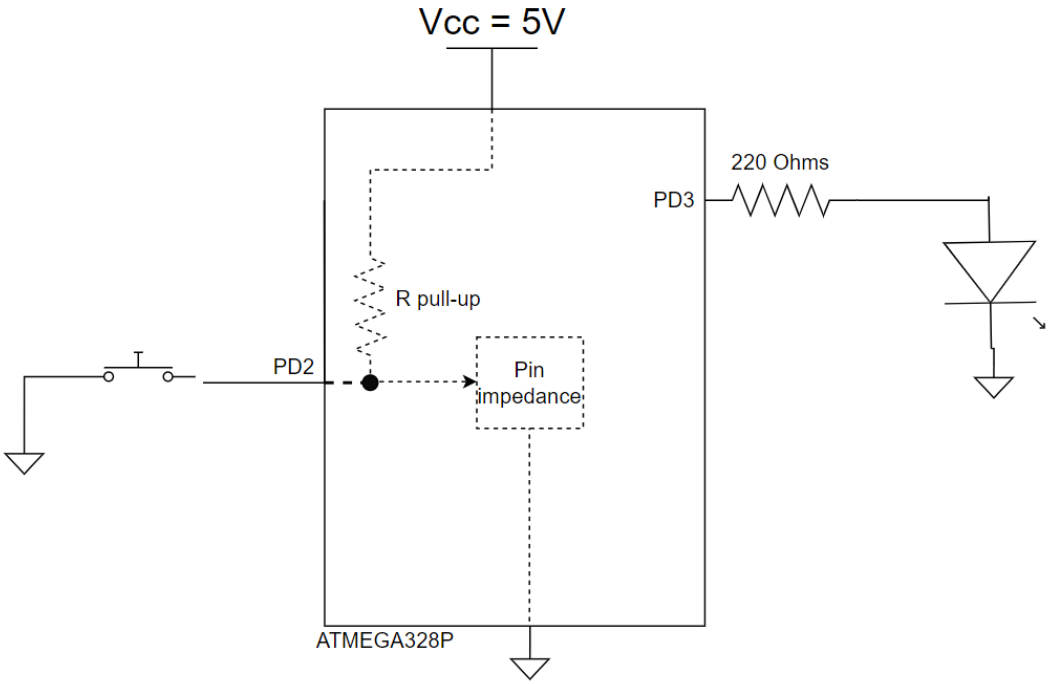
The general rule is to use a pull-up resistor (R1) that is an order of magnitude (1/10th) less than the input impedance (R2) of the input pin. An input pin on a microcontroller has an impedance that can vary from 100k-1MΩ. Impedance is "a way of saying resistance" and is represented by R2 in the picture below.

So, when the button is not pressed, the pull-up resistor R1 and input pin impedance R2 divides the voltage. But Vin voltage is high enough for the input pin to read a high state.



Exercise 3

Let's write a code to read status of pin as input. We will indicate this status on LED.



Schematic for Example 2

In this example, we are using a pushbutton as an input. Pushbutton is connected to pin PD2 of the microcontroller. We are using LED as an indicator of the status of pin PD2. LED is connected to PD3.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void){

/* ------------ SETUP ---------------------------------------*/
// 1. Set LED pin as output. All the I/O pins are inputs by default.
      DDRD  |= (1<<DDD3);
//-------------------------------------------------------------
// 2. Enable the Pull-up resistor for the input PUSHBUTTON
      PORTD |= (1<<PD2); // Explained in datasheet or page 5
```

```
/* ------------ LOOP ----------------------------------------
Read the state of the pushbutton to show it onto a LED.
Keep in mind that the read value when the PUSHBUTTON
is released is '1' due to the pull-up resistor.
-------------------------------------------------------------*/
while (1) {
//-------------------------------------------------------------
// 1. Read the pushbutton (read the whole PIND register)
// 2. If PUSHBUTTON is pressed, then PIND2='0'
        if (! (PIND & (1 << PIND2))) { // (1<<PIND2) <-> "0000 0100"

// ---------------------------------------------------------------------
//        PIND     |    MASK     |
//---------------------------------------------------------------------
// If (XXXX X1XX & 0000 0100 = 0000 0100) → Result is not Zero because
//                     PIND2='1' (Pushbutton is released, not pressed)
//
// If (XXXX X0XX & 0000 0100 = 0000 0000) → Result is Zero because
//                     PIND2='0' (Pushbutton pressed)

//---------------------------------------------------------------------
//    Not Zero is TRUE and Zero is false for the If structure
//---------------------------------------------------------------------
// 3. Switch ON the LED
              PORTD |= (1<<PD3);
//---------------------------------------------------------------------
        }
        else {
//---------------------------------------------------------------------
// 4. else switch OFF the LED
              PORTD &= (~(1<<PD3));
//---------------------------------------------------------------------
        }
   }
}
```

Disable Pull-Ups Over-ride

The PUD Pull-up Disable bit in the MCUCR register can over-ride the DDRx and PORTx pull-up settings.

Name: MCUCR
Offset: 0x55
Reset: 0x00
Property: When addressing as I/O Register: address offset is 0x35

Bit	7	6	5	4	3	2	1	0
		BODS	BODSE	PUD			IVSEL	IVCE
Access		R/W	R/W	R/W			R/W	R/W
Reset		0	0	0			0	0

When this bit is written to one, the pull-ups in the I/O ports are disabled even if the **DDxn** and **PORTxn** Registers are configured to enable the pull-ups **({DDxn, PORTxn} = 0b01)**.

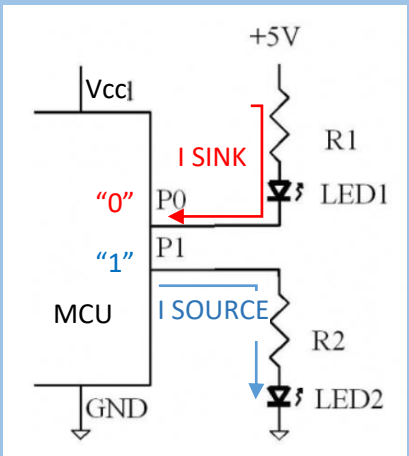
Unconnected Pins

If some pins are unused, we recommend that you ensure that these pins have a defined level, even though most of the digital inputs are disabled in the deep sleep modes. Floating inputs should be avoided to reduce current consumption in all other modes where the digital inputs are enabled (Reset, Active Mode and Idle Mode).

The simplest method to ensure a defined level of an unused pin is to enable the internal pull-up. In this case, the pull-up will be disabled during reset. If low power consumption during reset is important, we recommend you use an external pull-up or pull-down. Connecting unused pins directly to VCC or GND is NOT recommended, since this may cause excessive currents if the pin is accidentally configured as an output.

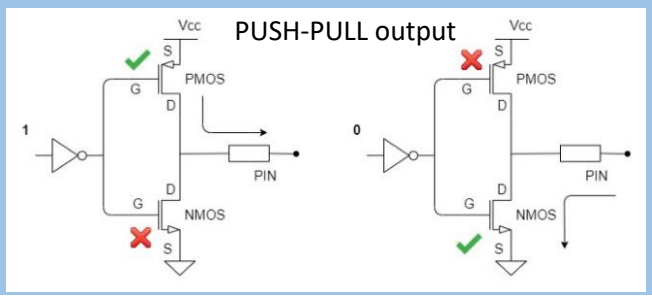
OPTIONAL CONTENT: Sinking or sourcing?

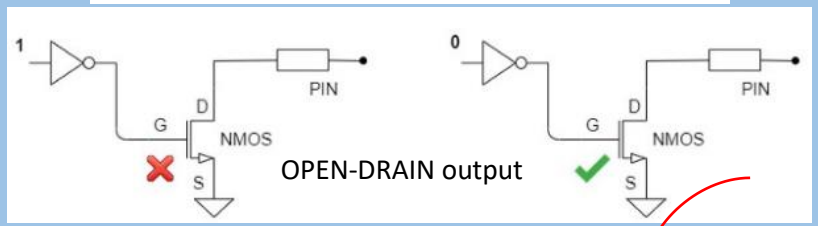
Which one should you use? To turn on LED1 you need to set the P0 register bit to a "0". To turn on LED2, you set P1 to a logical "1". To many developers, setting a pin to a "1" or high seems more natural for turning something on, so you might want to use the microcontroller pin as a source. One case you might need to use one driving method over another is **when the source and sink currents are not the same**, and only one method will be able to handle the current needed.

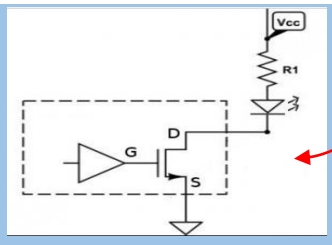


Source: https://www.w9xt.com/page_microdesign_pt4_drive_led.html
https://suketushah.medium.com/gpio-as-fast-as-possible-f78cbb9db50e

If the pin you selected is an open collector/drain type, you will need to **sink** the current. Open collector/drain outputs do not have an internal transistor to "Vcc", so the pin can't supply current. Some microcontrollers have a few I/O pins that are open collector/drain.





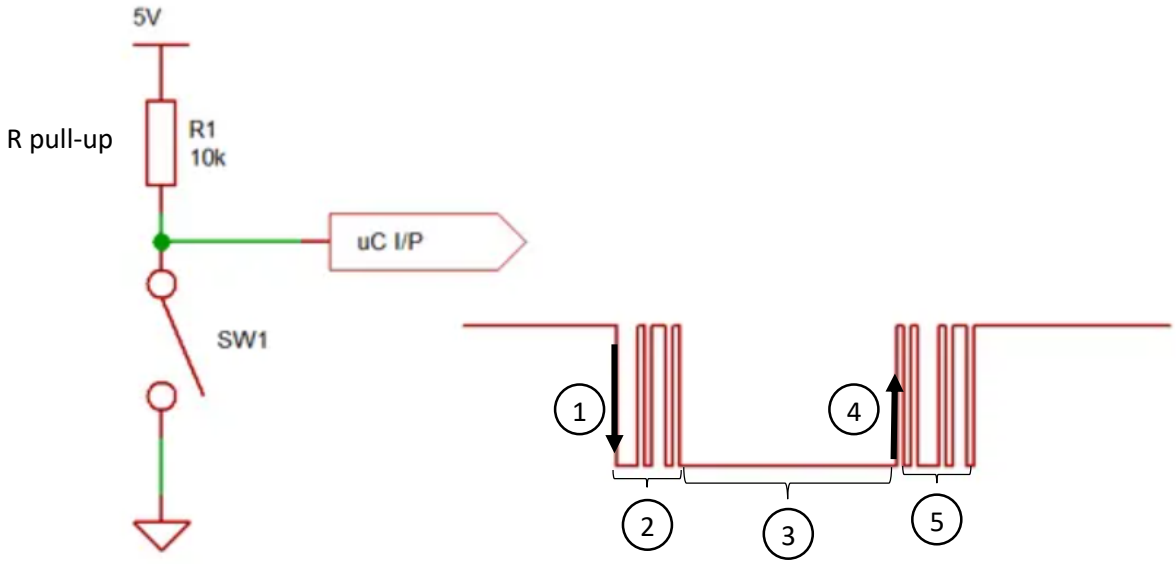


Exercise 4

Write a new code to light 2 out of 4 LED connected to the port D (the odd ones for example). While a pushbutton is not pressed, that pattern must be kept. But when the pushbutton is pressed the four LED must blink.

Exercise 5

Write a new code to switch ON the LED the first time the pushbutton is pressed, and to switch OFF the LED the second time is pressed.



1. Anyone has pressed the pushbutton. (Falling-edge)
2. Bouncing time (approx. 10ms is enough).
3. Does push button keep pressed?
   If so, caution! The task (toggle) must be done when the pushbutton is pressed, '0', but be aware of reading '0' many times.
4. The push button has been released.
5. It is recommended to skip this new period of bouncing.

Exercise 6

Making a clear, readable and easily portable code.

PORTS.h

- *Create a new project and add the main file and the header file "PARAMETERS" from the P02_Exercise_01 (BLINK_LED).*
- *Create a new header file called "PORTS.h". In this file add some macro definitions to rename the ports with a generic name instead that of the ATMEGA328.*

```
#ifndef PORTS_H_
#define PORTS_H_

/**************************************************************************
                    PORTS DEFINITION
**************************************************************************/
///MCU Port where the LED is connected. Mode Configuration register.
#define GPIO01_MODE     DDRB
///Pin of port where the LED is connected. Configured as output.
#define LED_MODE        DDB5
///MCU port where the LED is connected.
#define GPIO01          PORTB
///Pin where the LED is connected.
#define LED             PORTB5
```

- *In the main file, include the "PORTS.h" and replace the name of the REGISTER by the corresponding macro definition. Now, the code can be easily adapted for a new microcontroller (with different names for the registers).*

- *In the file "PORTS.h" add a function named "GPIO_init" declared with the qualifiers "static" and "inline". This function is for encapsulating the instructions to configure the mode of the ports (next page).*

- *In the main file, SETUP section, replace the instruction to configure the pin by the function GPIO_init();*

```
#ifndef PORTS_H_
#define PORTS_H_

/*****************************************************************
                    PORTS DEFINITION
*****************************************************************/
///MCU Port where the LED is connected. Mode Configuration register.
#define GPIO01_MODE     DDRB
///Pin of port where the LED is connected. Configured as output.
#define LED_MODE        DDB5
///MCU port where the LED is connected.
#define GPIO01          PORTB
///Pin where the LED is connected.
#define LED             PORTB5

/*****************************************************************
    PRIVATE FUNCTIONS (STATIC INLINE)   BarrGroup recommendation
*****************************************************************/
static inline void GPIO_init()
{ // Function related to PORTS
   ///-------------------------------------------------------------------------
   /// STATIC: invisible outside of the file
   /// INLINE: is not called, instead is copied into the file which is calling it
   ///-------------------------------------------------------------------------
   ///  1.- Set output pins
    GPIO01_MODE |=(1<<LED_MODE);
   ///----------------------------------------
}
#endif /* PORTS_H_ */
```

MACROS.h

It is recommended to define macros for common actions with the microcontroller registers to enhance clarity.

Create a new header file called "MACROS.h", add and complete the following macros in "MACROS.h" header file:

```
#define SETBIT(ADDRESS,BIT)        ???
#define CLEARBIT(ADDRESS,BIT)      ???
#define TOGGLEBIT(ADDRESS,BIT)     ???
```

Add a new main file replacing the port and pin names by the new names and use the macros for light on and light off the LED.

Exercise 7

*Create a new project and add from the **P02_Exercise_05** the following files:*

- *the main file (the non-blocking version)*
- *the header file "PARAMETERS"*

*From the **P02_Exercise_06** add:*

- *the header file "PORTS"*
- *the header "MACROS.h"*

a) Add to the file PORTS.h all the definitions related to the PUSH BUTTON. Assign the following names:

```
#define GPIO01_OUT          PORTB
#define GPIO01_IN           PINB
```

Write new macros in "MACROS.h" to define the following actions and names:

- `READ_PIN(ADDRESS,BIT)` `?? → For reading a PIN in a register`
- `BUTTON_PRESSED` `?? → To define a pressed button is '0' (false)`

Modify the code to add the new definitions. Try the design!

b) Add a copy of the C file to the project. Declare a Boolean variable to warn the pushbutton has been pressed (`buttonPressed`*), the <stdbool.h> header file is required.*

Replace the action TOGGLEBIT by the assignation of the value BUTTON_PRESSED to the new variable. Add the instructions to toggle the LED according to the state of the pushbutton.

Try the design!

c) Put all the code related to the process for debouncing into a function ("GPIO_readButton"), the function should return a boolean value. Place the function before the main function. Call the function in main to know if the pushbutton is pressed and then toggle the LED.

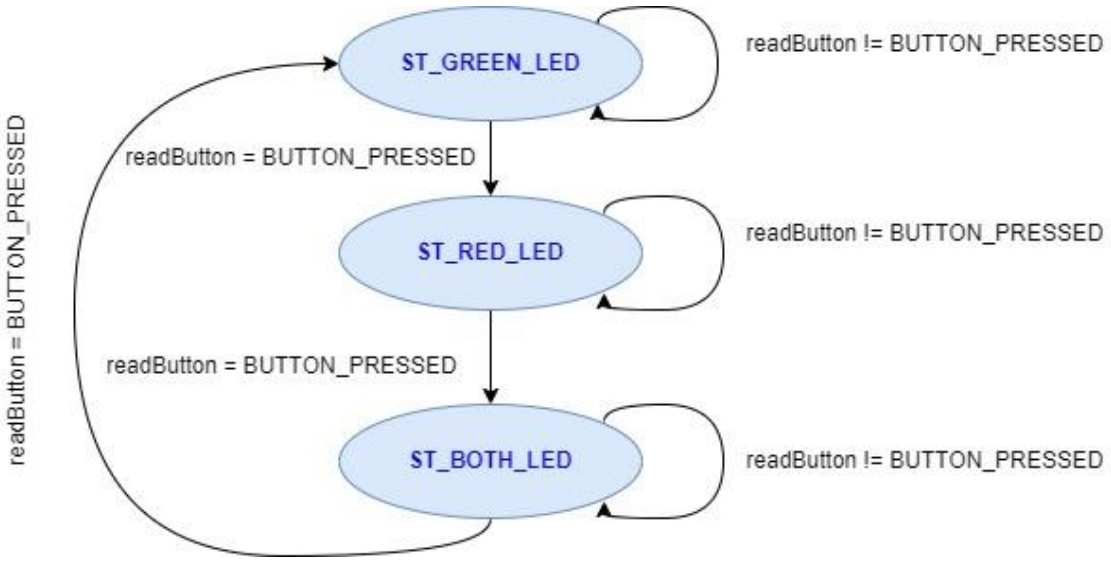
Try the design!

Exercise 8

Write a code, by using the FSM style and switch-case structure, to design a system such as:

- *the first time a pushbutton is pressed a LED_GREEN is turn ON.*
- *the second time, a LED_RED is turn ON.*
- *the third time the pushbutton is pressed both LED are turned off.*

Start drawing the states chart.



- *Add the function "`GPIO_readButton`" to check the input event.*
- *Add a red or green LED to the protoboard and add the required names to the PORTS.h file.*

a) VERSION 00:

Describe the main function by means of a FSM (4 steps: event, switch transitions, switch outputs, nextState).

- *Write down the bytes used by the ROM and the RAM memories.*

b) VERSION 01:

Describe the main function by means of a FSM (2 steps: event, switch transitions and outputs, nextState).

- *Write down the bytes used by the ROM and the RAM memories.*