Electronics tutorials

The purpose of this project is to learn electronics and microcontrollers with “playing” with Arduino microcontrollers. The project is divided in two parts divided in several chapters. The first chapter describes what is an Arduino, so in other words what is a microcontroller in general, what is it made for and of, and what is the difference with a computer. It also presents the wide range of Arduino and their applications. The second chapter is to be familiar with the board by doing the tutorials of Sunfounder RFID kit. It will cover all the basic knowledge we need to know to start doing great application of the Arduino. It covers switches, LED, several sensors, PWM, two types of motors (Stepper and servo), the Serial Communication with the computer, LCD1602 and so on. In the third chapter, we will work on communication buses, both wired and wireless communication, such as the most important serial buses like UART, I²C and SPI. This chapter covers Bluetooth and radio frequency communication as well. Then, in the fourth chapter, we deal with sensors and their circuit. I decided to use the most well-known sensors such as US HC-SR04 but also some interesting sensors, like the radar sensor that uses the Doppler effect. We will see some application of these sensors, how they are printed in board and how they work, and perhaps go inside their library to see how the are coded. Fifth chapter teaches the use of graphic display. Then we will go in more advanced chapter in the further chapters, like using OOP to structure our code, making GUI to improve the use of Arduino with humans in the sixth chapter. The finale chapter is the longer perhaps, it will be based on the book “The AVR microcontroller and Embedded systems using Assembly and C” of Naimi, Sephr. It teaches how the microcontroller works with low code (assembly), how it is build and afterwards we will be an AVR master.

The second part is more about pure electronics. Most of the basics will have been covered, this part is more to work on special circuit like the famous Timer NE555, switching power supply, transistors and so on. Most part are taken from the Art of Electronics. All the sources will be claimed.

Both parts are totally independent, you can read the second one first for instance?

This is a summary of what we will do during the whole project.

1. Arduino
2. What is an Arduino?
   1. History and features
   2. Arduino Family
   3. The UNO, MEGA and NANO
3. 30 quick Arduino lessons
   1. Controlling LED by button
   2. Controlling LED by PWM
   3. Interactive LED flowing lights
   4. RGB LED
   5. LED display (dot-matrix,
   6. Photoresistor
   7. Tilt switch
   8. Button switch
   9. Buzzer
   10. Answer machine
   11. Controlling voice by light
   12. LCD 1602
   13. Voltmeter
   14. Password lock
   15. Flame sensor
   16. Voice sensor
   17. Temperature sensor
   18. Water level sensor
   19. Humidity sensor
   20. 74HC595
   21. Relay module
   22. Stepper motor
   23. Servo motor
   24. Joystick
   25. RTC
   26. Serial monitor
4. Communication
   1. Wired
      1. UART
      2. I²C
      3. SPI
      4. CAN – MCP2515 chip
      5. Ethernet
   2. Wireless
      1. Bluetooth – HC-05 chip
      2. Radio 2.5G NRF24L01+PA+LNA SMA
      3. RFID RC-522
      4. WIFI
5. Sensors and they board
   1. Ultra-Sonic sensor HC-SR04
   2. Infrared sensors HC-SR501
   3. Laser sensor VL53L0X
   4. Microwave sensor CDM324 induction radar
   5. Speed encoder HC-02K and PID controller
   6. Accelerometer & gyroscope GY-521 MPU-6050
   7. GPS module NEO-6M-GPS6MV2
6. Graphics/Information display
   1. Classic liquid crystal display (LCD 2\*16)
   2. OLED 128\*64
   3. Tactile Display
7. Storing data
   1. Type of memory
      1. Flash
      2. EEPROM
   2. SD card
   3. Useful example: Data logger
8. Arduino and C++ Oriented-Object Programming (OOP)
   1. Basics C++ OOP
   2. OOP with Arduino
   3. Qt Application (more generally, GUI application)
   4. Examples (Protocole & Parseur, CAN, …)
9. Deeper look inside the Arduino: the AVR 8-bits microcontroller
   1. AVR Assembly book
10. Electronics
11. Analog
12. Digital

Part ONE : The Arduino

# What is an Arduino?

## History and features

## Arduino Family

## The UNO, MEGA and NANO

# 30 quick Arduino lessons

Let’s start doing interesting stuff with the Arduino Uno, or other board you have. I recommend the UNO because it is chip, most of the shield are compatible but a MEGA or NANO is still working for these small exercises. In my case I use a kit available on Internet including a board and many components such as sensors, relay, button, resistor, etc. I will follow the manual of the kit by adding some advice and tricks I found by doing them.

## Controlling LED with a switch button

### Introduction

First, it mays be great to see how look the Arduino code and the Arduino IDE.

How looks the Arduino code :

// #include

// Variable definition

void setup(){

// Code that will run just once, good to initialize port for example, serial communication, etc..

}

void loop(){

// Code that runs eveytime, it is a while(1) loop

}

The first Arduino example is really simple, it consists in bling an LED with a press button. This will be the first example with using the I/O ports (Input/Output ports). The Arduino UNO consists of several ports that can be both INPUT/OUTPUT.

### Components

Arduino Un

### Principle

### Schematic

### Code

You can choose a port can either be an INPUT or an OUTPUT with the function pinMode(#pin, MODE) .

Then, for the OUTPUT pins, to give them a value, we call the the function digitalWrite(#pin, VALUE);

In this case, VALUE can either take HIGH or LOW value. HIGH means the pin sends all the voltage it can, i.e. 5 volts and LOW means no voltage, i.e. closed.

In the Arduino UNO, there is an LED connected to the pin 13 and blinks according to the voltage across the pin.

SO we can first write the setup code and the variables. It is better to define the variable of port to constant or define. Define are quite better since they don’t use memory. In this case I chose const int to show you a method, we will have time to see all the possibilities across the project.

// Variables

const int port\_button = 12;

const int port\_LED = 13;

void setup() {

// INPUT definition

pinMode(port\_button, INPUT);

// OUTPUT definition

pinMode(port\_LED, OUTPUT);

}

Now, we need to blink the LED when we press the button. To do that, we will use a condition IF(condition). The IF(condition) executes the block is the condition is validated. Here, the condition is “button is pressed” which means electrically (or informatically) “there is a voltage across the pin” which now means in code “port\_button == HIGH”. To know the value of a port, the function digitalRead(#port) is necessary. It reads the the port and returns its value, HIGH(1) or LOW(0) for a digital port (non PWM). In the other hand, if it is not pressed, we lower the pin:

void loop() {

if(digitalRead(port\_boutton) == HIGH) // Read the value and compare it to HIGH

{

digitalWrite(port\_LED, HIGH);

}

else

{

digitalWrite(port\_LED, LOW);

}

}

## Controlling LED by PWM

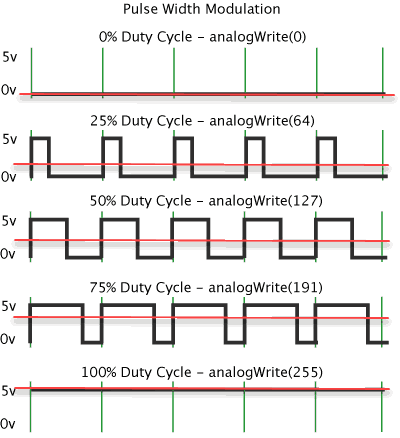
### Introduction

We can change the luminance of the LED through programming.

### Components

### Principle

PWM means Pulse-Width Modulation and is widely applied in motor speed regulation, light intensity control, power supply and signal ouput. PWM is a digital coding method for analog signal levels? Since a computer cannot output an analog voltage but a digital one between 0V and 5V (in the Arduino case), we modulate the duty cycle of square waves to create a new voltage that is simply the mean voltage.



### Schematic

### Code

We will learn two new functions, analogWrite(#port, value) that writes an “analog” value, a PWM value to be exact. From Arduino.cc:

“Can be used to light a LED at varying brightnesses or drive a motor at various speeds. After a call to analogWrite(), the pin will generate a steady square wave of the specified duty cycle until the next call to analogWrite() (or a call to digitalRead() or digitalWrite()) on the same pin. The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz.

On most Arduino boards (those with the ATmega168 or ATmega328P), this function works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 - 13 and 44 - 46. Older Arduino boards with an ATmega8 only support analogWrite() on pins 9, 10, and 11.  
The Arduino DUE supports analogWrite() on pins 2 through 13, plus pins DAC0 and DAC1. Unlike the PWM pins, DAC0 and DAC1 are Digital to Analog converters, and act as true analog outputs.  
You do not need to call pinMode() to set the pin as an output before calling analogWrite().  
The analogWrite function has nothing to do with the analog pins or the analogRead function.”

And delay(ms) that block the program during the number of ms.

We use a new loop as weel, the FOR loop. It is useful when you know the number of iteration we need to do.

## Interactive LED Flowing Lights

### Introduction

In this lesson we introduce the potentiometer to change the time interval. This lesson is not really complicated but it is quite tricky. I will show you two ways to program it. The first one will be the most intuitive in my opinion but has weakness, the second one is more “cerebral”.

### Components

### Schematic

### Code

We add a new function, analogRead(#pin) reads the value from the specified analog pin. The Arduino board contains a 6 channel (8 channels on the Mini and Nano, 16 on the Mega), 10-bit analog to digital converter. This means that it will map input voltages between 0 and 5 volts into integer values between 0 and 1023. This yields a resolution between readings of: 5 volts / 1024 units or, .0049 volts (4.9 mV) per unit. The input range and resolution can be changed using analogReference().

I use an array to store the pin number of each LED. It is smart to store big number of data in arrays, it avoids to declares each variable.

It takes about 100 microseconds (0.0001 s) to read an analog input, so the maximum reading rate is about 10,000 times a second.

The first code is the most intuitive in my opinion. It consists in blinking the LED in one way through a for loop, then in the way back in another for loop. In the loop, we read the value of the pot to actualize it. Otherwise, if I read the value outside the loop, I must wait the end of the block to read it again. So, if the loop takes 1s to run, the value of the pot is actualized every second that is too long. Therefore, I read it inside the loop.

The second method uses a function that is called in every loop of “void loop”. The principle is simple, I switch a LED ON while I am switching OFF the previous one.

# Communication

Embedded electronics is all about interlinking circuits to create a symbiotic system. In order for those individual circuits to swap their information, they must share a common communication protocol. Hundreds of communication protocol have been defined to achieve this data exchange, and, in general, each can be separated into one of two categories: parallel or serial.

The asynchronous serial protocol has a number of built-int rules – mechanism that help ensure robust and error-free data trasnfers. These mechanisms, which we get for eschewing the external clock signal, are:

* Data bits
* Synchronous bits
* Parity bits
* Baud rate

The protocol is highly configurable. The critical part is making sure that both devices on a serial bus are configured to use the exact same protocols.

The Baud rate specifies how fast data is sent over a serial line. It is usually expressed in units of bit-per-seconds. If we invert the baud rate, we can find how long it takes to transmit a single bit. Baud rates can be just about any value within reason. The only requirement is that both devices operate at the same rate. One of the more common baud rates, especially for simple stuff where speed isn’t critical, is 9600 bps. Other “standard” baud are 1200, 2400, 4800, 19200, 38400, 57600, and 115200.

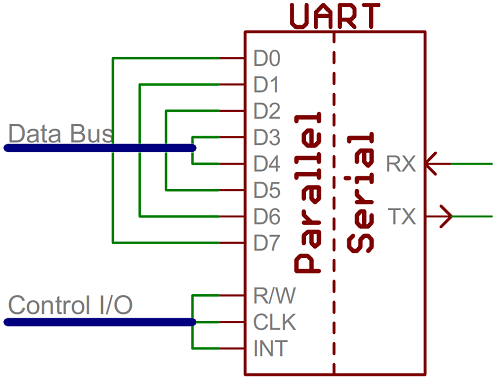
The higher a baud rate goes, the faster data is sent/received, but there are limits to how fast data can be transferred. You usually won’t see speeds exceeding 115200 - that’s fast for most microcontrollers. Get too high, and you’ll begin to see errors on the receiving end, as clocks and sampling periods just can’t keep up

## UART

Introduction

The final piece to this puzzle is finding something to both create the serial packets and control those physical hardware lines. Enter the UART.

A Universal Asynchronous Receiver/Transmitter is a block of circuitry responsible for implementing serial communication. Essentially, it acts as an intermediary between parallel and serial interfaces.



It does exist as a stand-alone IC, but as in the Arduino it is commonly found inside microcontrollers.

## SPI

Introduction

Serial Peripheral Interface (SPI) is an interface bus commonly used to send data between mircocontrollers and small peripherals such as shift registers, sensors and SD cards. It uses separate clock and data lines, along with a select line to choose the device you wish to talk to.

It is a synchronous serial(see difference between synchronous and asynchronous serial). SPI works in a slightly different manner. It’s a “synchronous” data bus, which means that it uses separate lines for data and a clock that keeps both sides in perfect sync. The clock is an oscillating signals that tells the receiver exactly when to sample the bits on the data line.

<https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi>

## I²C

### Introduction

I²C states for Inter-Integrated Circuit, it I a protocol intended to allow multiple “slave” digital integrated circuits (“chips”) to communicate with one or more “master” chips. Like the SPI, it is only intended for short distance communication within a single device? Like Asynchronous Serial Interfaces (RS-232 or UART), it only requires two wires to exchange information.

Difference with Serial ports

Because serial ports are asynchronous (no clock data is transmitted), devices using them must agree ahead of time on a data rate? The two devices must also have clocks that are close to the same rate, and will remain so-excessive differences between clock rates on either end will cause garbled data.

Cf <https://learn.sparkfun.com/tutorials/i2c#why-use-i2c>

## CAN

### Introduction

The Controller Area Network is a bus standard for vehicle designed to allow microcontrollers and devices to communicate each other in applications without a host computer. It is a message-based protocol. The first chip produced by Intel and Phillips came on the market in 1987. The Mercedes-Benz W140 was the first production vehicle to feature a CAN-based wiring system.

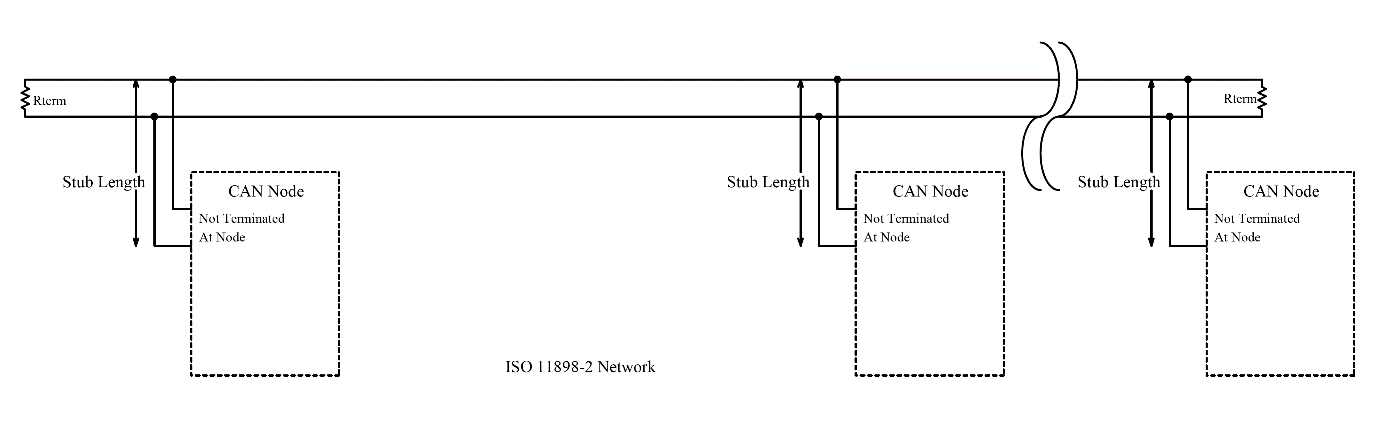
Some features of the CAN bus:

* Broadcast type of bus, it means that all nodes can “hear” all transmissions. There is no way to send a message to just a specific node; all nodes will invariably pick up all traffic. The CAN hardware, however, provides local filtering so that each node may react only on the interesting messages.
* Tus bus uses Non-Return to Zero (NRZ) with it-stiffing: if just one node is driving the bus to a logical 0, then the whole bus is in that state regardless of the number of nodes.
* Has an error handling.
* Up to 8 bytes data in a frame
* Maximum rate at 40 meters: 1Mbaud/s
* Maximum rate at 6 km: 10kBaud/s
* Many nodes per bus: practical limit is approximately 100 nodes due to transceiver
* Data fields

### Protocol & Architecture

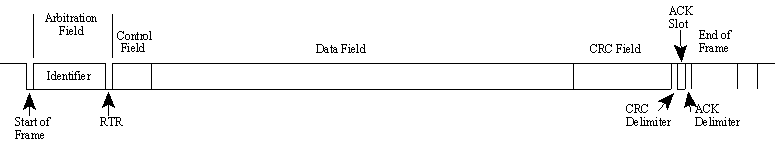
CAN is a multi-master serial bus standard for connecting Electronic Control Units [ECUs] also known as nodes. Two or more nodes are required on the CAN network to communicate. The complexity of the node can range from a simple I/O device up to an embedded computer with a CAN interface and sophisticated software. The node may also be a gateway allowing a standard computer to communicate over a USB or Ethernet port to the devices on a CAN network.

All nodes are connected to each other through a two-wire bus. The wires are a twisted pair with a 120 Ω (nominal) characteristic impedance.



ISO 11898-2, also called high speed CAN, uses a linear bus terminated at each end with 120 Ω resistors.

The CAN message contains an identifier and data. The identifier can be either standard (11bits) or extended (29 bits). The CAN ID is sometimes referred to as a PGN (Group). I won’t explain the differences between standard and extended ID, you can easily find it on the internet. In our case we will only use standard ID, easier to use for our needs. A typical frame looks like that:



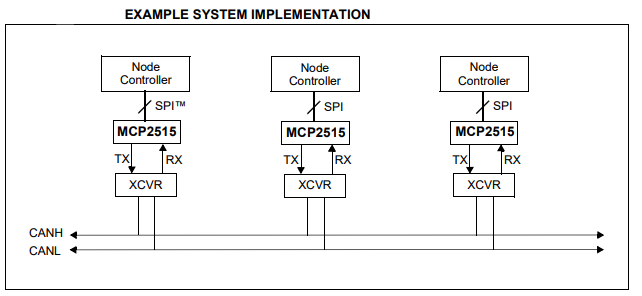
What is interesting us is the identifier and the data field.

The chips we use are built the same way, they have two main components: one chip to create the physical interface of the bus (TJA1050 or MCP2551) and one chip to handle the CAN protocol, the MCP2515.

According to the datasheet, he MCP2551 is a high-speed CAN, fault-tolerant device that serves as the interface between a CAN protocol controller and the physical bus. The MCP2551 device provides differential transmit and receive capability for the CAN protocol controller and is fully compatible with the ISO-11898 standard, including 24V requirements. It will operate at speeds of up to 1 Mb/s. Typically, each node in a CAN system must have a device to convert the digital signals generated by a CAN controller to signals suitable for transmission over the bus cabling (differential output). It also provides a buffer between the CAN controller and the high-voltage spikes that can be generated on the CAN bus by outside sources (EMI, ESD, electrical transients, etc.). The TJA1050 chip has the same feature, so I assume using both has the same result.

The MCP25115 is a stand-alone CAN controller. It is capable of transmittinf and receiving both standard and extended data and remote frames. It has two acceptance masks and six acceptance filters tat are used to filter out unwanted messages, thereby reducing the host MCUs overhead? The MCP2515 interfaces with microcontrollers (MCUs) via an industry standard Serial Peripheral Interface (SPI).

The figure below shows a simple example of system implementation.

(source: microchip)

Arduino

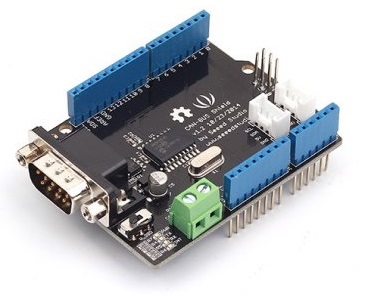
Board connected to Arduino

The boards we use have all the components needed to work directly on the CAN bus. I use two of them: one with the MCP2515+MCP2551 and an other one with MCP2515+TJA1050. It doesn’t matter whether a board has the MCP2551 or the TJA1050, but the MCP2515 is what we need to have in common to use the library. Indeed, it is the MCP2515 that communicates with the Arduino through the SPI bus. So, people has written a library to easy the use of the chip, and this library is only written for the MCP2515. We will talk about the library later.

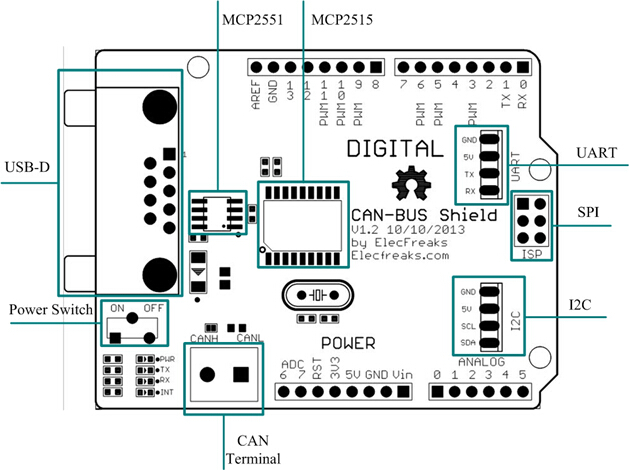
This module lets you talk on a CAN bus using SPI to connect it to something like an Arduino which doesn't have its own built-in CAN circuitry. All the CAN logic and protocol handling is done automatically.

The MCP2515 provides buffering, filtering, interrupts and addressing so you don't need to worry about programming that all yourself.

CAAN-bus-shield v1.2 for Arduino UNO

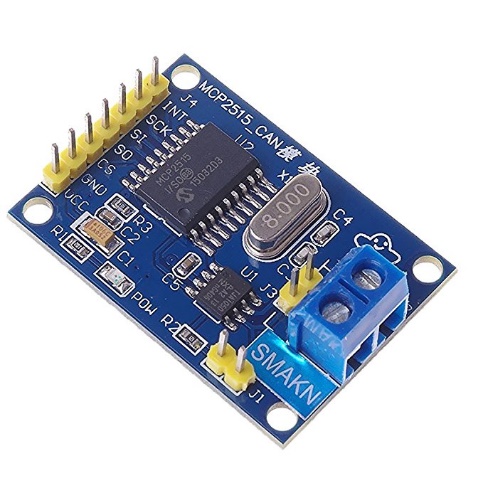


Wiki: <http://www.elecfreaks.com/wiki/index.php?title=CAN-BUS_Shield>



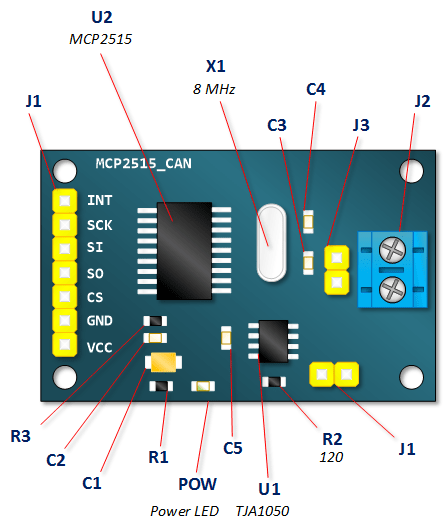
I use the v1.2 version of the board. The CS is the pin 10 and not 9 as the datasheet. There is another mistake. The SPI interface of the MCP2515 is not connected to the pin D13(SCK), D12(MISO) and D11(MOSI). However, it is well connected to the ICSP as in the Arduino board like in the datasheet available in the wiki page.

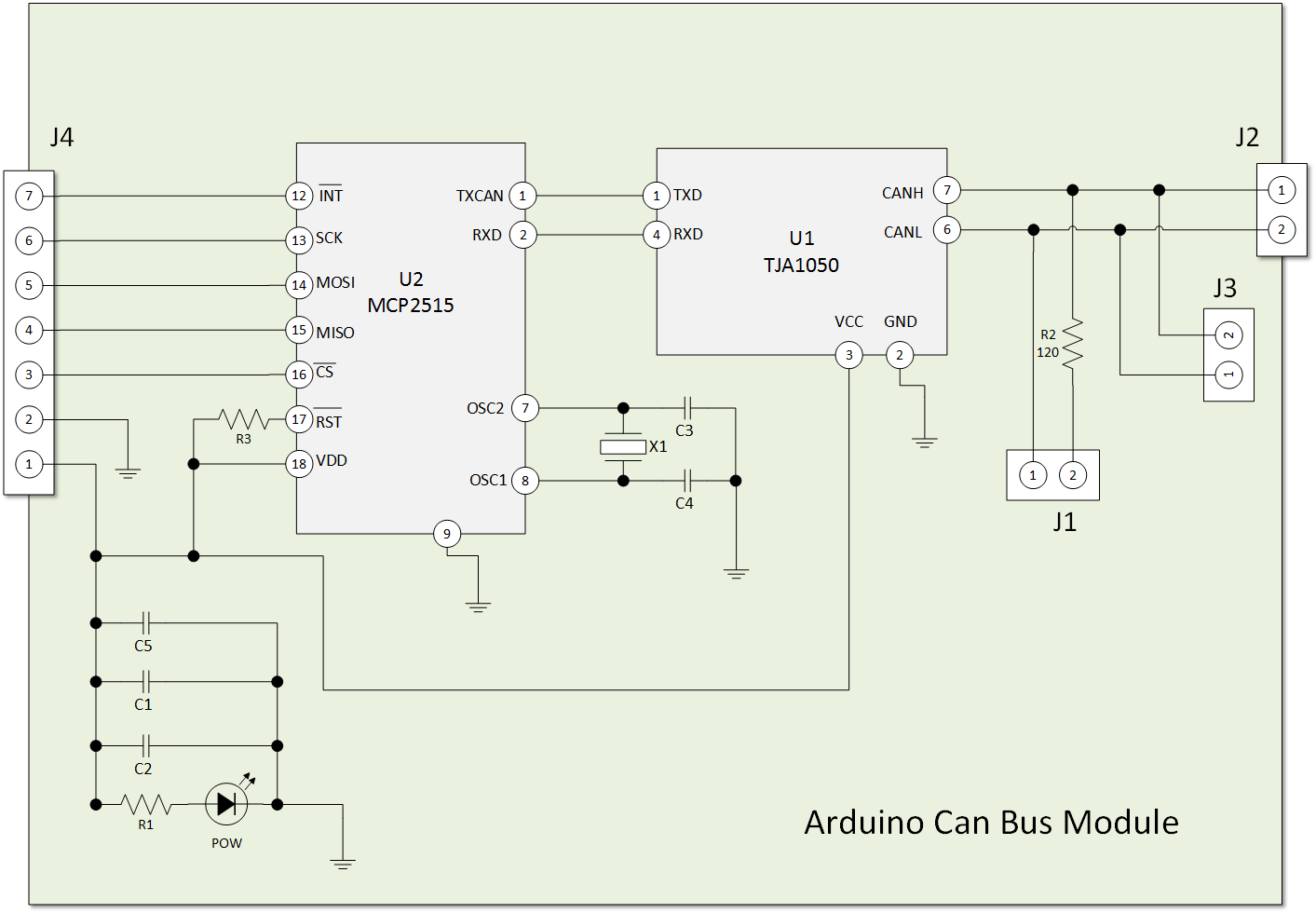
The second module is smaller and better for embedded system.

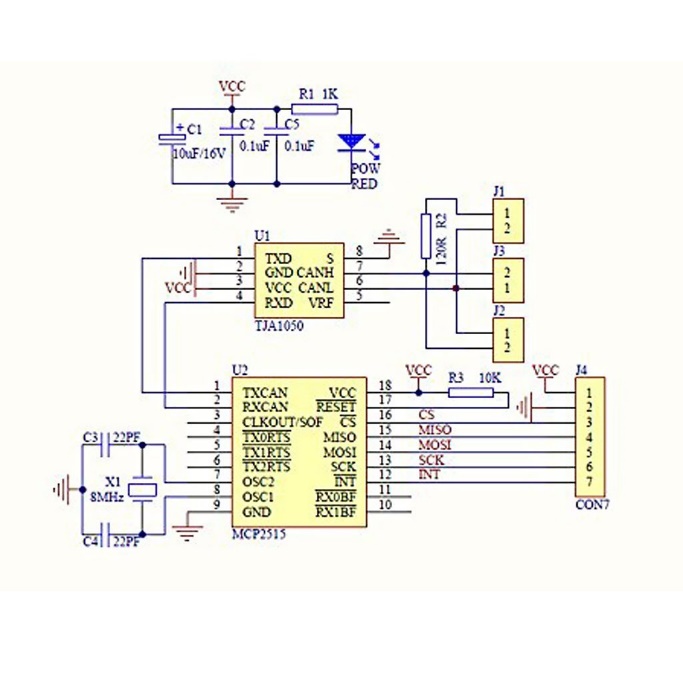


This module contains a 120 ohm resistor that can be connecting by bridging the including pin-header jumper near the CAN\_H and CAN\_L lines.

Here is the schematic of the second board:

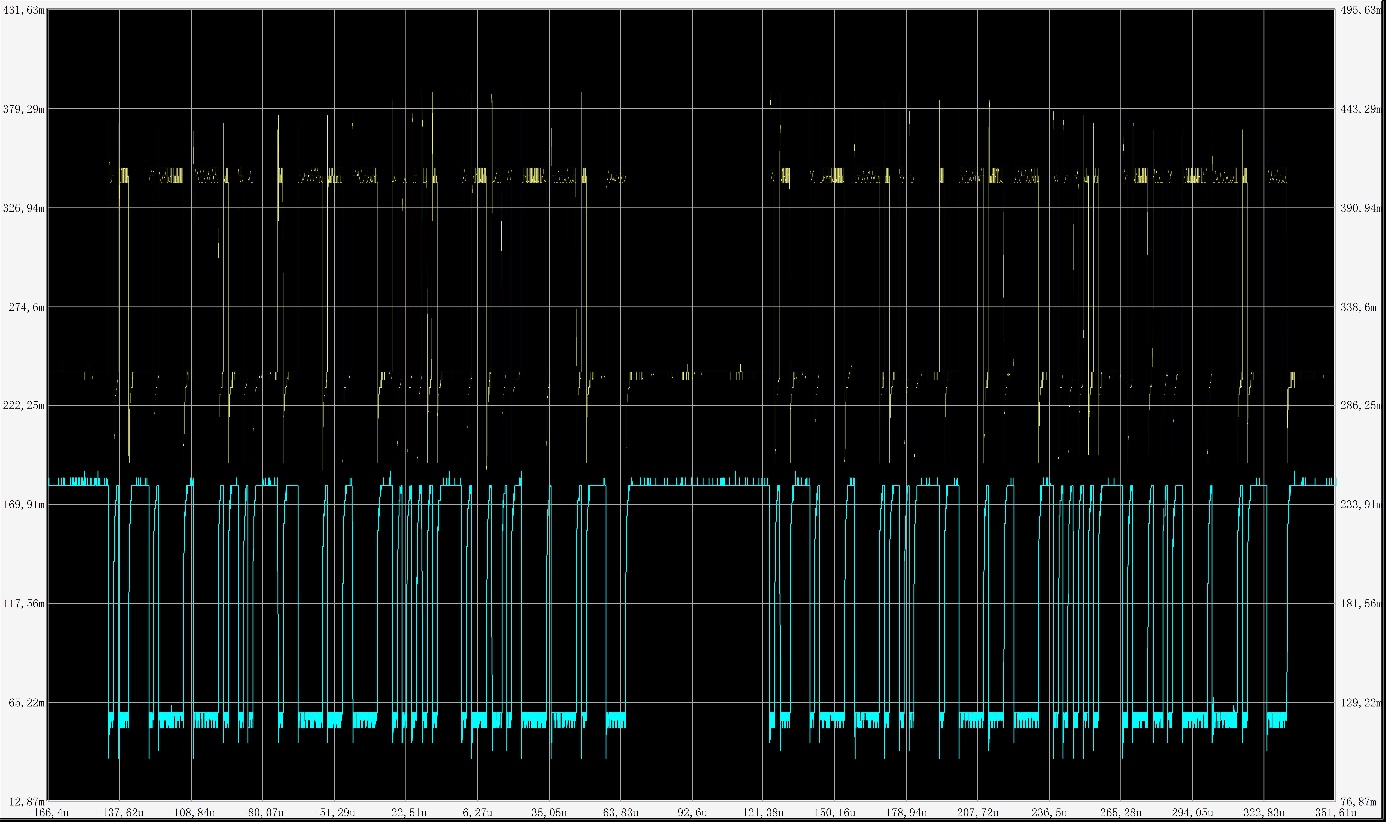






Each module has it own oscillator [explain why].

I use this library because it supports almost all the frequency, due to the fact I used a 10Mhz crystal during a project.



* 1. Ethernet

# Sensors module and their board

This part deals with several sensors that are either well known such as the first we deal with, the HC-SR04, or rare like the doppler sensor CDM324. In the first time, we introduce the physics of the sensor: how does it work and where it can be find. Then we look closer to the chip mounted on the board, it specification and the circuit to show the amplification circuit, filtering etc. Then we see the library used for it and explain how it works. Finally the last part is more pragmatic, we see an or several examples of the module and the library.

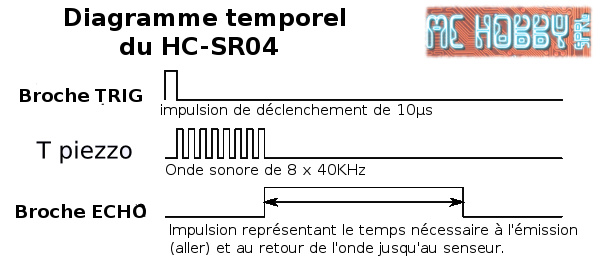
## Ultra-sonic sensor HC-SR04

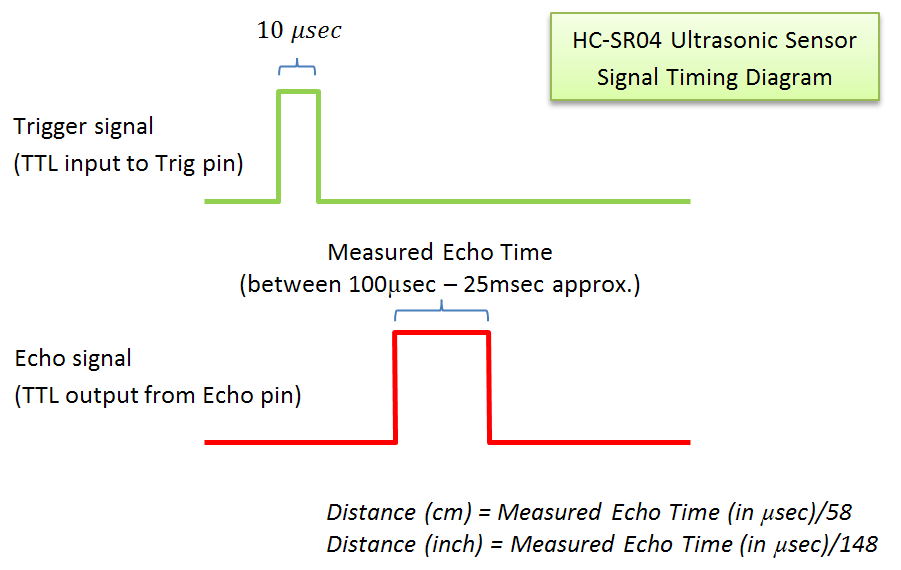
### Introduction

The ultrasonic waves are widely used in many fields like healthcare whit the echography, in the industry as quality checking, etc. It is based on emitting a sound wave at high frequency (40kHz for example) that is inaudible [see range of audible frequencies]. The time it takes to go back and forth is proportional to the distance between each objects.

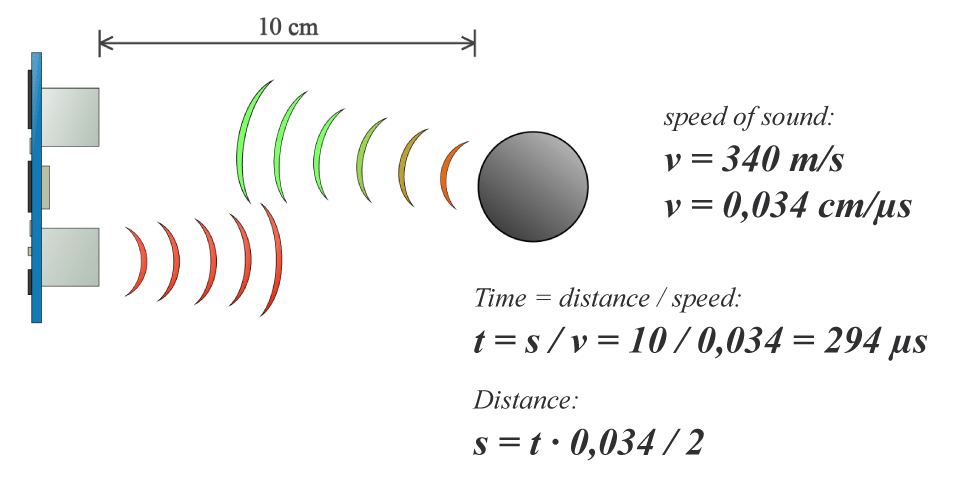
To compute the length, the principle is based on the speed of sound in the air. As we know, the mean distance can be computed as followed: . There is a transmitter that send the wave and a receiver that catches the sending wave. The time elapsed between the send and receive is equal to two times the distance (see figure below).

The HC-SR04 is the most common US module used with the Arduino. It is really cheap, around 5€ per sensor. It emits an ultrasound wave at 40kHz frequency which travels through the air. In order to generate the ultrasound you need to set the Trig on a High State for 10 µs. That will send out an 8-cycle sonic burst which will travel at the speed sound and it will be received in the Echo pin. The Echo pin will output the time in microseconds the sound wave traveled.





For example, if the object is 10 cm away from the sensor, and the speed of the sound is 340 m/s or 0.034 cm/µs the sound wave will need to travel about 294 u seconds. But what you will get from the Echo pin will be double that number because the sound wave needs to travel forward and bounce backward. So in order to get the distance in cm we need to multiply the received travel time value from the echo pin by 0.034 and divide it by 2.



### The chip

The module has 4 pins as shown below:



|  |  |  |
| --- | --- | --- |
| Pin Number | Pin Name | Description |
| 1 | Vcc | It powers the sensor with +5V |
| 2 | Trigger | It is an INPUT pin. This pin has to be kept high for 10 us to initialize measurements by sending US waves |
| 3 | Echo | Echo pin is an OUTPUT pin. This pin goes hi  0gh for a period if time which will be equal to the time taken for the US wave to return back to the sensor. |
| 4 | Ground | Connected to the ground |

HC-SR04 feature

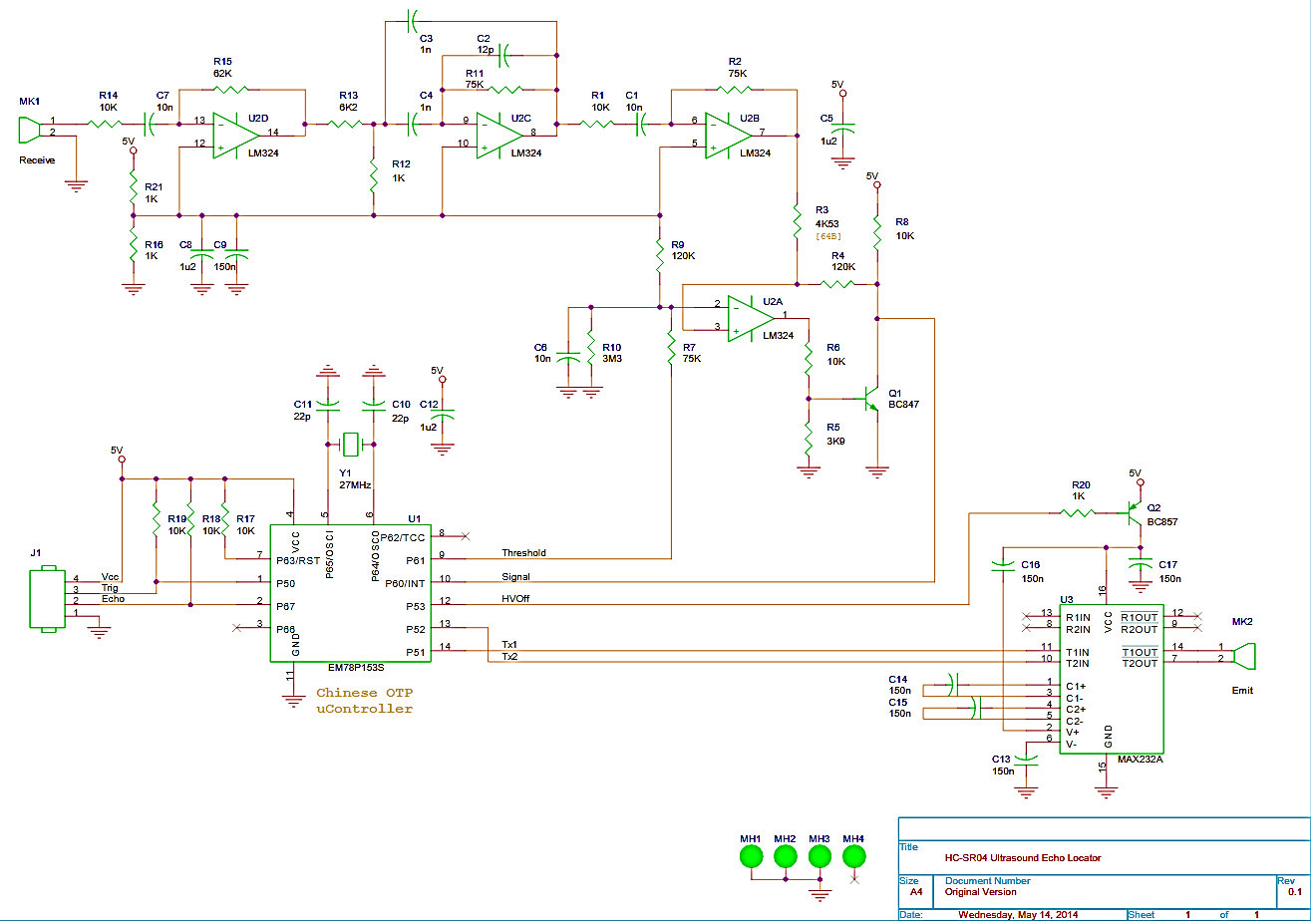
* Operating voltage: +5V
* Theoretical Measuring Distance: 2cm to 450cm
* Practical Measuring Distance: 2cm to 80cm
* Accuracy: 3mm
* Measuring angle covered: <15°
* Operating Current: <15mA
* Operating Frequency: 40Hz

Explanation of the circuit.

The circuit has two transducers, one for emission and one for reception.

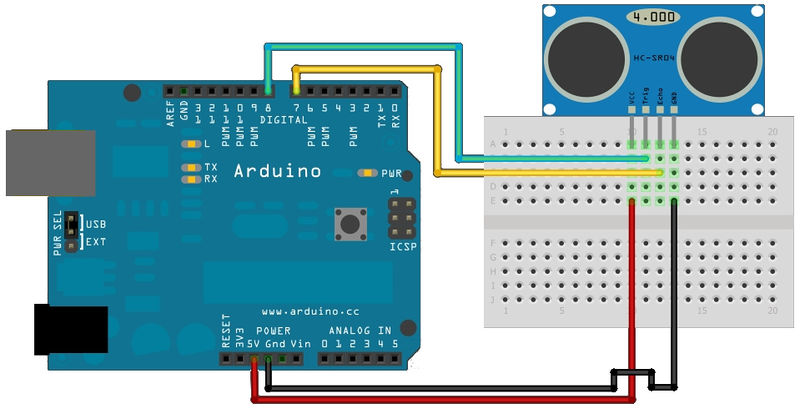
To transmit the ultrasonic pulses, a relatively high voltage is needed. A MAX232 is cleverly used to produce +/-10V (which are the normal USART voltage) from 5V. The transducer is connected between two outputs, so it is in fact powered at 20V. Power is only applied to this circuit through Q2 some time before and during the pulse emission because the internal switching charge-pump is noisy. When the circuit switches to receive mode, the MAX232 power is cut off.

The receive and emit circuit are controlled but an EM78P153S Chinese microcontroller running at 27 MHz. The receiver side uses LM324 which contains 4 OPAMPs. U2D is just a times 6 amplifier. U2C is a multiple feedback (1st order) pass band which is followed by another times 8 amplifier (U2B). The last OPAMP (U2A) is used together with Q1 as a hysteresis comparator.



Source : <https://uglyduck.ath.cx/ep/archive/2014/01/Making_a_better_HC_SR04_Echo_Locator.html>

How to connect it to the Arduino?



The code

New function : pulseIn(pin, state) returns the time in microseconds of the pin at the state.

<https://www.arduino.cc/reference/en/language/functions/advanced-io/pulsein/>

First we need to define the pins of the trigger and the echo. In my case, I chose respectively pin 8 et 7 on the Arduino Board. Then you need a long variable “duration” for the travel time that you will get from the sensor and an integer for the distance.

In the setup loop, we define the pinTrig as an OUTPUT and the pinEcho as an INPUT and also start the serial communication for showing the results on the serial monitor.

In the void loop, we set the pinTrig on HIGH state for 10 us. To do this, the easiest way is to set the pinTrig HIGH with the function digitalWrite, then launch a delayMicroseconds and set the pin to LOW. We will see that gives us some time error, like an interrupt of the timer1. But for a start, let’s do this way.

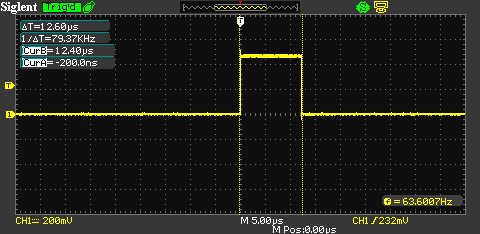
After setting the trigger LOW, we read the pulse of the ECHO pin. The function pulseIn isn’t that efficient, but in our case, it is fast enough.

As I described earlier, to compute the distance value we just have to multiplicate by 0.034 and divided by 2 so multiplicate by 0.017.

What is wrong with this code ?

As you can see, the length of the trigger pulse is longer than 10 us, it is mainly because of the digitalWrite that takes a little bit longer. To reduce the delay, we work with the register.

For instance, to write HIGH to the pin 8, we need to know to which register the pin depends on.



Then if you used an oscilloscope to see the pulse, you may have noticed that the pulse is sometimes wider. This is due to an interrupt that appears with the timer0. To avoid it, we need to disconnect interrupt with this timer. However, using this method will make the function pulseIn useless because it is based on the timer0 interrupt.

#include "wiring\_private.h"

#include "pins\_arduino.h"

*/\* Measures the length (in microseconds) of a pulse on the pin; state is HIGH*

*\* or LOW, the type of pulse to measure. Works on pulses from 2-3 microseconds*

*\* to 3 minutes in length, but must be called at least a few dozen microseconds*

*\* before the start of the pulse. \*/*

**unsigned** **long** **pulseIn**(**uint8\_t** pin, **uint8\_t** state, **unsigned** **long** timeout)

{

*// cache the port and bit of the pin in order to speed up the*

*// pulse width measuring loop and achieve finer resolution. calling*

*// digitalRead() instead yields much coarser resolution.*

**uint8\_t** bit = digitalPinToBitMask(pin);

**uint8\_t** port = digitalPinToPort(pin);

**uint8\_t** stateMask = (state ? bit : 0);

**unsigned** **long** width = 0; *// keep initialization out of time critical area*

*// convert the timeout from microseconds to a number of times through*

*// the initial loop; it takes 16 clock cycles per iteration.*

**unsigned** **long** numloops = 0;

**unsigned** **long** maxloops = microsecondsToClockCycles(timeout) / 16;

*// wait for any previous pulse to end*

**while** ((\*portInputRegister(port) & bit) == stateMask)

**if** (numloops++ == maxloops)

**return** 0;

*// wait for the pulse to start*

**while** ((\*portInputRegister(port) & bit) != stateMask)

**if** (numloops++ == maxloops)

**return** 0;

*// wait for the pulse to stop*

**while** ((\*portInputRegister(port) & bit) == stateMask) {

**if** (numloops++ == maxloops)

**return** 0;

width++;

}

*// convert the reading to microseconds. The loop has been determined*

*// to be 20 clock cycles long and have about 16 clocks between the edge*

*// and the start of the loop. There will be some error introduced by*

*// the interrupt handlers.*

**return** clockCyclesToMicroseconds(width \* 21 + 16);

}

We can replace the pulseIn with an interruption routine. The interruption is called when the edge changes. When it is rising, it means that the INPUT is HIGH so we save the time. Then when it is falling, we compute the time difference.

<https://os.mbed.com/components/HC-SR04/>

<http://www.ezdenki.com/ultrasonic.php>

**Improved code using AVR Assembly language**

This is the first code written in Assembly we have seen so far. As I said earlier, writing in C & Assembly makes the code less readable but highly efficient. This code comes is available here.

Code 2.0 : <http://extremeelectronics.co.in/avr-tutorials/interfacing-ultrasonic-rangefinder-with-avr-mcus-%E2%80%93-avr-tutorial/>

The code below is written for an AVR microcontroller. The steps required to read distance are:

1. Microcontroller make the i/o line output. (by using the[DDRx Register](http://extremeelectronics.co.in/avr-tutorials/part-v-digital-io-in-avrs/) in AVR or [TRISx Register](http://extremeelectronics.co.in/microchip-pic-tutorials/general-purpose-digital-io-with-pic-microcontrollers/) in PIC)
2. The i/o line is made low (this may be the default state of i/o pin)
3. Wait for 10uS
4. Make the i/o line high.
5. Wait for 15uS
6. Make the i/o line low
7. Wait for 20uS
8. Now make it input (by using the[DDRx Register](http://extremeelectronics.co.in/avr-tutorials/part-v-digital-io-in-avrs/) in AVR or [TRISx Register](http://extremeelectronics.co.in/microchip-pic-tutorials/general-purpose-digital-io-with-pic-microcontrollers/) in PIC)
9. Module will keep it low. Wait till it is low, as soon as it becomes high start the [**timer**](http://extremeelectronics.co.in/avr-tutorials/avr-timers-an-introduction/).
10. After that wait till it is high, as soon as it becomes low copy the [**timer**](http://extremeelectronics.co.in/avr-tutorials/avr-timers-an-introduction/) value and stop the timer.
11. Finally we have the time required for the wave to go hit the obstacle and come back to the module.