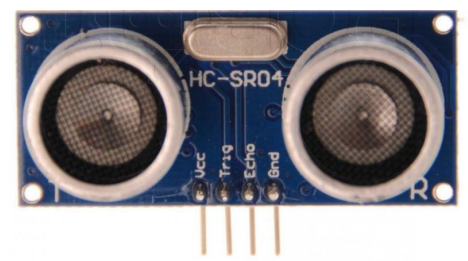
**Ultrasonic distance sensor**

**J. Lemaire**

**(Pierrefeu Mai 2021)**

# Introduction

Here, I consider a distance sensor, the ultrasonic sonar **HC-SR04**[[1]](#footnote-2) :

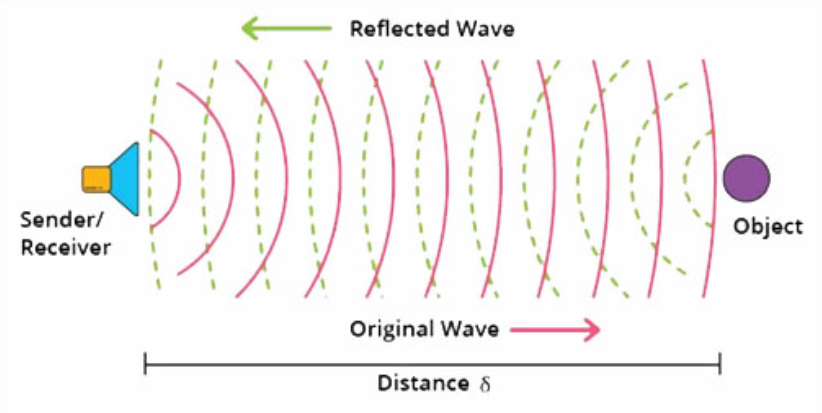


I tried to understand how it measures distances and to improve the classicaly proposed C++ code, using **Pin Change Interrupts** of an Arduino Uno.

# The HC-SR04 distance sensor

## Characteristics

It’s a simple and low cost analog sonar which measures a distance δ from the sensor to an object, on evaluating the time τ taken by an ultrasonic wave (the « ping ») to moves in the air :

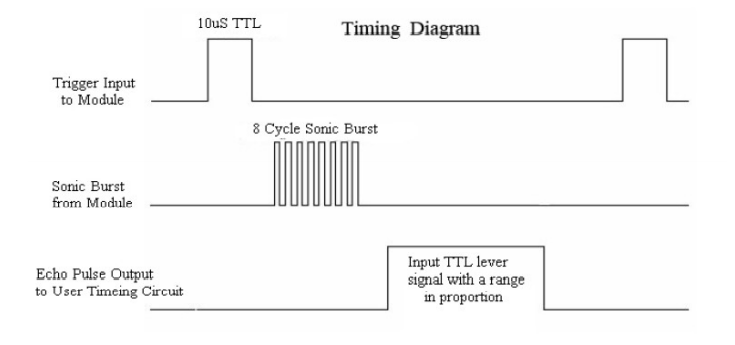
[[2]](#footnote-3)

At a temperature of 20° Celcius and a pressure of 1024hPa, the wave speed in the air beeing approximatively 340m/s = 0.034cm/µs, the distance can be obtained by the formula :

where, (cm) and (µs).

The HC-SR04 can be powered with 5V DC and has 2 pins[[3]](#footnote-4) with 5V TTL logic :

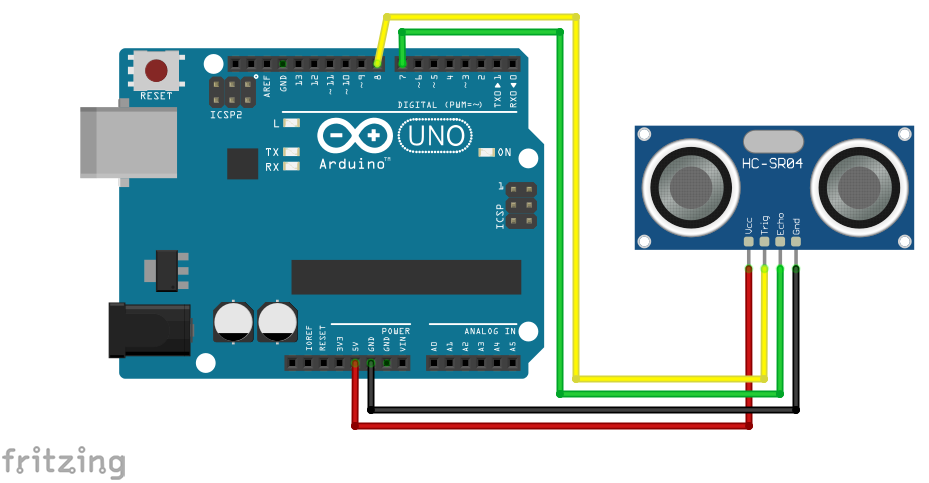
* **Trigger** : to send a « ping »[[4]](#footnote-5) when HIGH level during 10 µs
* **Echo** : it is at a HIGH level during the µs



The distance need to be in the interval and the precision is near 0.3 cm[[5]](#footnote-6).

## Simple Arduino sketch to measure the distance

We consider such a sensor, connected on the pin 8 (trigger) and the pin 7 (Echo) of an Arduino Uno[[6]](#footnote-7) :



To evaluate the distance between the sensor and an object, we just have to measure the duration of the HIGH level state of the Echo pin, after a « ping » and the Arduino **pulseIn**[[7]](#footnote-8) function can do it very simply.

The following sketch calls this function periodically, using the **ElapsedMillis**[[8]](#footnote-9) library to manage simply the period :

// TestSonar1.ino

// Test the ultrasonic sensor HC-SR04 using pulseIn

#include <elapsedMillis.h>

#define TRIGGER\_PIN 8

#define ECHO\_PIN 7

elapsedMillis timer**;**

unsigned long tau**,** tauNew**;**

unsigned long duration**;**

void setup**()**

**{**

Serial**.**begin**(**9600**);**

delay**(**100**);**

pinMode**(**TRIGGER\_PIN**,** OUTPUT**);**

pinMode**(**ECHO\_PIN**,** INPUT**);**

digitalWrite**(**TRIGGER\_PIN**,** LOW**);**

**}**

void loop**()**

**{**

**if** **(**timer **>** 500**)**

**{**

// Reset

timer **=** 0**;**

// Start processing

duration **=** micros**();**

// Ping

digitalWrite**(**TRIGGER\_PIN**,** HIGH**);**

digitalWrite**(**TRIGGER\_PIN**,** LOW**);**

// Wave time measure

tauNew **=** pulseIn**(**ECHO\_PIN**,** HIGH**,** 40000**);**

**if** **(**tauNew **>=** 50 **&&** tauNew **<=** 25000**)** tau **=** tauNew**;**

// End processing

duration **=** micros**()-**duration**;**

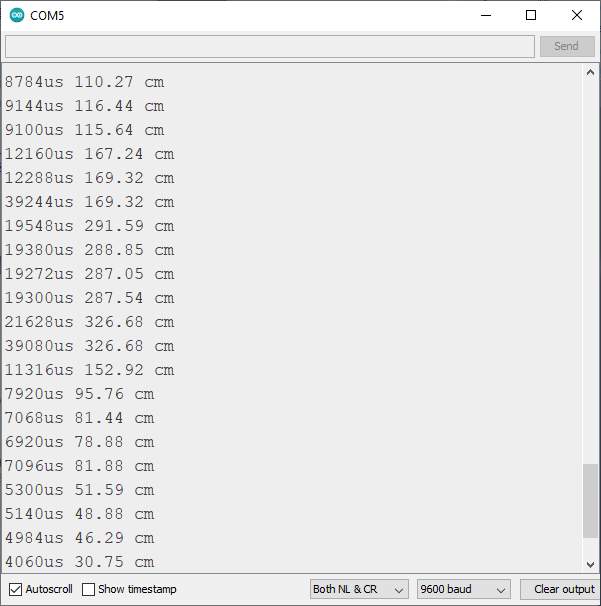
// Display

Serial**.**print**(**duration**);** Serial**.**print**(**"us "**);** Serial**.**print**(**tau**/**59.**);** Serial**.**println**(**" cm"**);**

**}**

**}**

The results are correct, but this sketch is blocking :



Mean values for the observed blocking times[[9]](#footnote-10) :

|  |  |
| --- | --- |
| Distance (cm) | Measure time (ms) |
| 25 | 4 |
| 70 | 6 |
| 125 | 9 |
| 250 | 16 |

So an interrupted non blocking would be a better approach and fortunately there is a very simple solution which doesn’t interfere with the other processes.

## The Pin Change Interrupt approach

The **ATMega328P** of Arduino Uno offers several types of interruptions[[10]](#footnote-11), to measure the HIGH level duration of the echo pin, in a non blocking way :

* The Timer interrupts ;
* The External Interrupts ;
* The Pin Change Interrupts.

The **timer** objects (timer0, timer1 and timer2 for an Arduino Nano) are typically used in an interrupted mode, overflow for example, where an interrupt routine can be called when the count register of the timer overflows (256 for a 8 bits timer or 65536 for a 16 bit timer). But this approach isn’t well suited here because all the timers are already used :

* **timer0** by **millis()** and **micros() ;**
* **timer1** by the **Servo** library ;
* **timer2** by the **tune** function (sounds).

The **External interrupts** are very simple to implement using the **attachInterrupt[[11]](#footnote-12)** function. But they are limited to pin **2** or **3** on an Arduino Uno, and because these pins are already reserved in Otto by his leg servos, we prefer the last option.

The **Pin Change Interrupts[[12]](#footnote-13)** (**PCI** to simplify),look like the previous ones but practically for all the pins. Unfortunately, they are less simple to implement, essentially because one can define only one **ISR**[[13]](#footnote-14) for all the pins of the same **port**[[14]](#footnote-15) :

|  |  |  |
| --- | --- | --- |
| **Pin#** | **Port#** | **PCINT#** |
| 8 | 0 | 0 |
| 9 | 1 |
| 10 | 2 |
| 11 | 3 |
| 12 | 4 |
| 13 | 5 |
| 14 (A0) | 1 | 8 |
| 15 (A1) | 9 |
| 16 (A2) | 10 |
| 17 (A3) | 11 |
| 18 (A4) | 12 |
| 19 (A5) | 13 |
| 0 | 2 | 16 |
| 1 | 17 |
| 2 | 18 |
| 3 | 19 |
| 4 | 20 |
| 5 | 21 |
| 6 | 22 |
| 7 | 23 |

More precisely[[15]](#footnote-16) :

* **ISR(PCINT0\_vect)** for **Port0** (pins 8 to 13) ;
* **ISR(PCINT1\_vect)** for **Port1** (pins 14 to 19) ;
* **ISR(PCINT2\_vect)** for **Port2** (pins 0 to 7).

Furthermore, contrary to the external interrupts, the **PCI** cannot make a difference between the **FALLING** or **RISING** pin changes. But it’s not a problem : just need to test the pin level in the ISR !

A **PCI** activation is similar to the activation an External interrupt :

* Uses the bits 0-2 of the **PCICR[[16]](#footnote-17)** register to enable the ports ;
* Uses the **PCMSK0**, **PCMSK1** or **PMSK2** registers[[17]](#footnote-18) to select the pins, using the previous correspondance between **pin#** and **PCINT#**.

For instance, to use **PCI** on pins **6** and **7**, which all belong to **Port2** and correspond respectively to **PCINT22** and **PCINT23**, just need to use :

bitSet**(**PCICR**,** PCIE2**);**

bitSet**(**PCMSK2**,** PCINT22**);**

bitSet**(**PCMSK2**,** PCINT23**);**

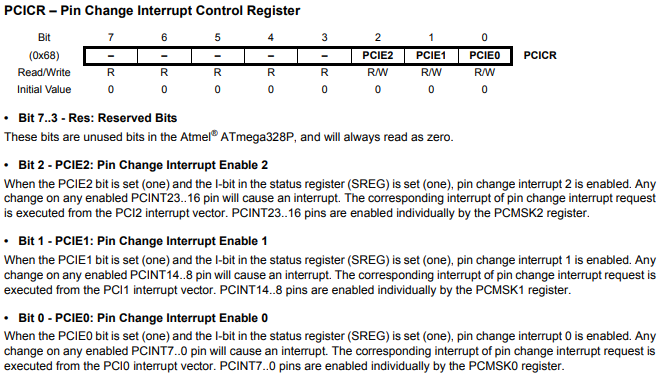
**...**

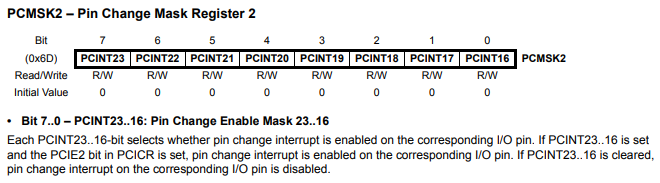
ISR**(**PCINT2\_vect**)**

**{**

…

**}**

****



***Remarks***

* Be carrefull on doing this to not destroy a previous implementation of such **ISR**s ! For instance, when the Bluetooth module of Otto is connected to pins 11 (TX) and 12 (RX), it uses the **SerialSoftware** library which activates PCI on these pins belonging to Port0 and then implements **ISR(PCINT0\_vect)**. So avoid to use other pins of that port !
* Don’t forget that all the interrupts are suspended inside an **ISR** function[[18]](#footnote-19), and especially the **timer0** interrupts used in **millis()** and **micros()**. So begin to call these functions if they are requested and have after a treatment as short as possible.
* In an **ISR(PCINTx\_vect)** function, the only way to detect what is the pin which caused the interrupt consists in maintaining the levels of the active pins declared in **PCMSKx** and testing which has changed.
* An **ISR** is a static function, therefore it cannot use non-static members of a class. A solution consists in using an instance of that class[[19]](#footnote-20).

Having understood these crucial points, you can then use the **PCI** without any problems.

There exist a library[[20]](#footnote-21) who does the job for a lot of microcontrollers (of which the Arduino series is part) and implements an equivalent of the **attachInterrupt** function :

attachPinChangeInterrupt(<no PCINT>, <callbackFunction> , <mode>)

But its code is a bit complex and not easy to understand.

So I prefer to implement here a much more simple code, but restricted to an ATMega328P and only one pin of the port2.

// TestSonar2.ino

// Test the ultrasonic sensor HC-SR04 using PCI

#include <elapsedMillis.h>

#define TRIGGER\_PIN 8

#define ECHO\_PIN 7

elapsedMillis timer**;**

volatile unsigned long tau**,** tauNew**;**

unsigned long duration**;**

bool toUpdate**;**

ISR**(**PCINT2\_vect**)**

**{**

unsigned long t **=** micros**();**

**if** **(**digitalRead**(**ECHO\_PIN**))** tauNew **=** t**;**

**else**

**{**

tauNew **=** t **-** tauNew**;**

**if** **(**tauNew **>=** 50 **&&** tauNew **<=** 25000**)**

**{**

tau **=** tauNew**;**

// Stop PCI for Port2

bitClear**(**PCICR**,** PCIE2**);**

**}**

**}**

**}**

void setup**()**

**{**

Serial**.**begin**(**9600**);**

delay**(**100**);**

pinMode**(**TRIGGER\_PIN**,** OUTPUT**);**

pinMode**(**ECHO\_PIN**,** INPUT**);**

digitalWrite**(**TRIGGER\_PIN**,** LOW**);**

// Enable PCI for ECHO\_PIN

bitSet**(**PCMSK2**,** ECHO\_PIN**);**

toUpdate **=** **true;**

**}**

void loop**()**

**{**

**if** **(**timer **>** 480 **&&** toUpdate **)**

**{**

// Start processing

duration **=** micros**();**

toUpdate **=** **false;**

// Ping

digitalWrite**(**TRIGGER\_PIN**,** HIGH**);**

digitalWrite**(**TRIGGER\_PIN**,** LOW**);**

// Enable PCI for Port2

bitSet**(**PCICR**,** PCIE2**);**

// End processing

duration **=** micros**()-**duration**;**

**}**

**else** **if** **(**timer **>** 500**)**

**{**

// Reset

timer **=** 0**;**

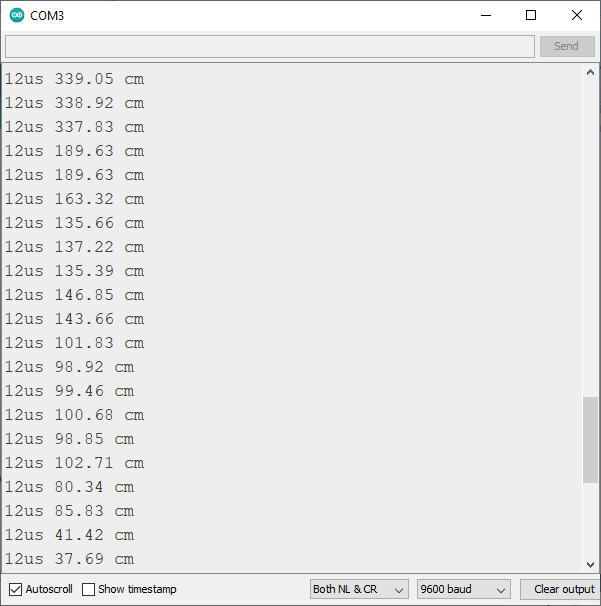
toUpdate **=** **true;**

// Display

Serial**.**print**(**duration**);** Serial**.**print**(**"us "**);** Serial**.**print**(**tau**/**59.**);** Serial**.**println**(**" cm"**);**

**}**

**}**



Hence, a much less blocking approach !

***Remarks***

* Observe that waiting 10µs to send a pin doesn’t seem necessary in both approaches.
* As in the simple approach, we used a temporary variable, **tauNew** to store the measured durations, and a simple interval test to discard the incorrect values. It appears sufficient here ; but there exist much more sophisticated filters, such as the median filters[[21]](#footnote-22).
* In the PCI approach, to simplify, we opted to update the distance measure 20ms before the displays. It’s more efficient to update this distance independently but periodically : we do it in the class implementation which follows.

# Class version

// Sonar.h

#pragma once

#include "Arduino.h"

#include <elapsedMillis.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class Sonar

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class to manage an HC-SR04 sonar with periodic measures of the distance using

PCI (Pin Changed Interrupts) on the echo pin (need to be in [0, 7]).

\*/

extern "C" void PCINT2\_vect**();**

class Sonar

**{**

public**:**

Sonar**(**unsigned pinTrigger**,** unsigned pinEcho**,** unsigned int period**);** // Constructor (ms) for period

bool updateDistance**();**

double getDistance**();** // (cm)

private**:**

unsigned \_pinTrigger**;**

unsigned \_pinEcho**;**

unsigned \_period**;**

volatile unsigned long \_tauNew**,** \_tau**;**

elapsedMillis \_timer**;**

void funcISR**();**

friend void PCINT2\_vect**();** // To use private funcISR() in the ISR

**};**

// Sonar.cpp

#include "Sonar.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ISR

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Sonar**\*** pSonar**;** // As static

ISR**(**PCINT2\_vect**)**

**{**

pSonar**->**funcISR**();**

**}**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class Sonar

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Sonar**::**Sonar**(**unsigned pinTrigger**,** unsigned pinEcho**,** unsigned int period**)**

**{**

pSonar **=** **this;**

\_pinTrigger **=** pinTrigger**;**

\_pinEcho **=** pinEcho**;**

\_period **=** period**;**

pinMode**(**\_pinTrigger**,** OUTPUT**);**

digitalWrite**(**\_pinTrigger**,** LOW**);**

pinMode**(**\_pinEcho**,** INPUT**);**

bitSet**(**PCMSK2**,** \_pinEcho**);** // Enable PCI for \_pinEcho

**}**

bool Sonar**::**updateDistance**()**

**{**

**if** **(**\_timer **>** \_period**)**

**{**

\_timer **=** 0**;**

// Ping

digitalWrite**(**\_pinTrigger**,** HIGH**);**

digitalWrite**(**\_pinTrigger**,** LOW**);**

// Enable PCI for Port2

bitSet**(**PCICR**,** PCIE2**);**

// return

**return** **true;**

**}**

**else** **return** **false;**

**}**

double Sonar**::**getDistance**()**

**{**

**return** **(**\_tau**/**58.82**);**

**}**

void Sonar**::**funcISR**()**

**{**

unsigned long t **=** micros**();**

**if** **(**digitalRead**(**\_pinEcho**))** \_tauNew **=** t**;**

**else**

**{**

\_tauNew **=** t **-** \_tauNew**;**

**if** **(**\_tauNew **>=** 50 **&&** \_tauNew **<=** 25000**)**

**{**

\_tau **=** \_tauNew**;**

bitClear**(**PCICR**,** PCIE2**);** // Stop PCI for Port2

**}**

**}**

**}**

// TestSonar3.ino

// Test the ultrasonic sensor HC-SR04 using PCI and a Sonar class

#include "Sonar.h"

#include <elapsedMillis.h>

Sonar sonar**(**8**,** 7**,** 20**);**

elapsedMillis timer**;**

void setup**()**

**{**

Serial**.**begin**(**9600**);**

delay**(**100**);**

**}**

void loop**()**

**{**

sonar**.**updateDistance**();**

**if** **(**timer **>** 500**)**

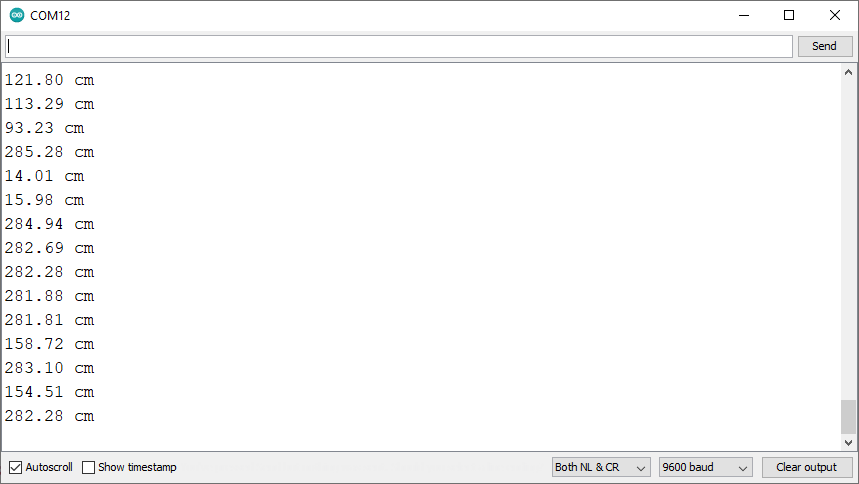
**{**

timer **=** 0**;**

Serial**.**print**(**sonar**.**getDistance**());** Serial**.**println**(**" cm"**);**

**}**

**}**



1. <https://s-o.webnode.cz/_files/200001547-3ef073fea3/hcsr04.pdf> for instance. [↑](#footnote-ref-2)
2. On the HC-SR04, the Sender and the Receiver are separated : the « eyes » of Otto ! [↑](#footnote-ref-3)
3. On the HY-SR05, the Trigger and Echo functions can be managed by only one pin. [↑](#footnote-ref-4)
4. 40 kHz and 8 cycles, therefore above the frequency that an human ear can perceive (20 kHz generally). [↑](#footnote-ref-5)
5. For a perfect reflecting surface ! [↑](#footnote-ref-6)
6. Equivalent to a Nano. [↑](#footnote-ref-7)
7. <https://www.arduino.cc/reference/en/language/functions/advanced-io/pulsein/> [↑](#footnote-ref-8)
8. <https://www.arduino.cc/reference/en/libraries/elapsedmillis/> [↑](#footnote-ref-9)
9. They are identical with the famous **NewPing** library from Tim Eckel with an Arduino Uno : <https://bitbucket.org/teckel12/arduino-new-ping/wiki/Home>. But this library can use timer interrupts for other microcontrollers, which is much more efficient. [↑](#footnote-ref-10)
10. <http://gammon.com.au/interrupts> [↑](#footnote-ref-11)
11. <https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/> [↑](#footnote-ref-12)
12. [ATmega328P](http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf) (datasheet) p49-57. [↑](#footnote-ref-13)
13. Interrupt Service Routine. [↑](#footnote-ref-14)
14. **Port0**, **Port1** and **Port2** are called **PortB**, **PortC** and **PortD** also. [↑](#footnote-ref-15)
15. Very regrettably, In the ATMega328P datasheet, the same notation is used to represent both :

    a port number interrupt, used in ISR(PCINTx\_vect) : PCINT0, PCINT1 and PCINT2

    a pin number interrupt : PCINT0, …, PCINT23

    Only the context can help you to make the difference ! [↑](#footnote-ref-16)
16. [ATmega328P](http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf) p 56. [↑](#footnote-ref-17)
17. [ATmega328P](http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf) p 57. [↑](#footnote-ref-18)
18. <http://gammon.com.au/interrupts> [↑](#footnote-ref-19)
19. <https://stackoverflow.com/questions/41443720/how-to-create-an-isr-in-an-arduino-class> [↑](#footnote-ref-20)
20. <https://github.com/NicoHood/PinChangeInterrupt> [↑](#footnote-ref-21)
21. <https://bitbucket.org/teckel12/arduino-new-ping/wiki/Home> for instance. [↑](#footnote-ref-22)