UNIT 13: THE I2C BUS

AIMS

Any controller with communication capabilities has access to the outside world. It can exchange information with other controllers or electronic devices like PCs or tablets, with peripherals like printers, scanners, digital pads, mice and also with other devices such as sensors, integrated circuits and actuators.

There are different forms and protocols for exchanging information. You can use standard serial communication as you have already done on a number of occasions; and then there's the 1-wire protocol you used in the previous unit. Another alternative is the I^2C protocol that we're going to study in this unit.

As examples of the practical applications of the I^2C bus you'll be working with two new peripherals: an ultrasonic range finder and a real-time clock and calendar. You can put these two on your list right now; they'll give you a lot more scope in future projects and applications.

THEORY SECTION

- INTRODUCTION
- THE I^2C PROTOCOL
  - The I2C Bus
  - Features
  - I2C Terminology
- THE WIRE LIBRARY
- GETTING SOME IDEAS STRAIGHT
- THE SRF02 ULTRASONIC RANGE FINDER
  - Operational Principles
  - Features and Connections of the SRF02
  - Internal Registers
  - Commands
  - Examples of Information Frames

- THE DS1307 REAL-TIME CLOCK AND CALENDAR
  - Features
  - Enclosure and Pin-out
  - Internal Registers
  - Examples of Information Frames
  - The DS1307 Library

PRACTICE SECTION

- EXAMPLE 1: A Version of Firmware
- EXAMPLE 2: Distances
  - A Brief Digression: Operations between Bits
  - Applications of Example 2
- EXAMPLE 3: More Distances
- EXAMPLE 4: Collision Avoidance System
- EXAMPLE 5: Speed Detector
- EXAMPLE 6: Area Metre
- EXAMPLE 7: Real-Time Clock
- EXAMPLE 8: Clock and Calendar Part 1
- EXAMPLE 9: Clock and Calendar Part 2
- EXAMPLE 10: Billboard
- EXAMPLE 11: Data Logger

PRACTICE MATERIALS

-Lap top or desk top computer

-Arduino IDE work environment; this should include the supplementary material already installed and configured.

-Arduino UNO controller board

-A USB cable

TABLE OF CONTENTS

THEORY SECTION

1. INTRODUCTION

Don't think of a controller as an isolated device. It's not: every time you've written or read binary values on a range of peripherals using the input and output pins the controller's communicated with the outside world. Just the same, all current controllers, including Arduino, have electronic circuits specifically designed for bit by bit serial communication with the outside world. They use very few pins and cables to transfer the information.

You've already used one of these circuits. Think back to the examples on serial communication that transferred information from the PC using the Arduino IDE or Integrated Development Environment serial monitor. This circuit is commonly known as "USART", (Universal Synchronous Asynchronous Receiver Transmitter).

This type of communication is designed for transmitting information between controllers (like the Arduino) and electronic devices such as modems, printers, computers, CNC numerical control machines, digital pads, other Arduinos and lots more.

Another type of circuit for serial communication is known as "SSP" or Synchronous Serial Port. It's mainly used for transferring information between controllers and other auxiliary integrated circuits within the same system: various sorts of memories, ADC and DAC converters, clocks, timers, sensors and others.

One of the most common protocols used for this kind of communication is known as "I^2C" (Inter Integrated Circuit). This is the one you're going to study and use in this unit although you did "cut your teeth" with another serial protocol in the unit 8: the 1-wire.

2. THE I2C PROTOCOL

The I2C protocol was developed by Philips in the nineties to connect up integrated circuits within single devices or household appliances such as radio receivers, television sets, video players and recorders, DVDs, hi-fi systems and others.

Imagine a controller that manages all the functions of a television set. It relies on other special integrated circuits that take care of tuning in the channels, storing them in the memory, decoding the remote control signals, setting the sleep timer and adjusting the colour, brightness, contrast, volume, tones and other details. That's all very well but how do we connect them all up so that they exchange information? Well, one way of doing it is by using the I2C protocol.

These days it's universal. You'll find dozens of different appliances on the market that do all sorts of different things but they've all got something in common: they use the same protocol and the same electric signals to communicate with each other.

You'll find lots of information about this protocol and its features on the net. It's worthwhile paying a visit to this website:

http://www.nxp.com/products/interface_and_connectivity/i2c/

This is the semiconductor division of the Dutch company Philips and you'll be able to see the enormous range of appliances available.

A. THE I2C BUS

In computer architecture, a "bus" (a contraction of the Latin omnibus) is a communication system that transfers data between components inside a computer or between computers. It consists of one or a number of pins or electric signals that are shared by one or more devices to exchange information. I2C uses only two signals or pins as you can see from the Figure 1 .

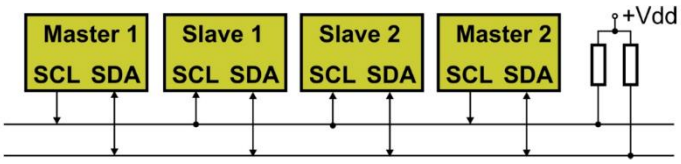


Figure 1

NAME	DESCRIPTION
SCL	Serial clock line. It's always output when used for the master or host and input for the slave.
SDA	Serial data line. This is bidirectional. The data can travel from the master to the slave or the other way around.

All I2C devices have SCL and SDA signals and they've got "open collector" or "open-drain" pins. All you need to know is that when there's no data transfer the pull-up resistors ensure the signals stay on level "1".

The master, also known as the host, is the controller itself. In this case it'll be the Arduino UNO It's responsible for generating the clock signal, initiating the communication with the slaves, indicating the address of the slave it wishes to communicate with and terminating the communication.

B. FEATURES

The following is a summary of the most important features of the I2C protocol or bus:

- It uses just two signals to transfer data: SCL and SDA.
- They're both open collector signals so they have to be connected to +V using two separate pull-up resistors.
- Byte wise data transfer. 8 bits is the minimum for each word transferred.
- "Multi-master" replication system. We won't be working with it but a single bus (SCL, SDA) may include several master or host controllers as well as several slaves.
- All slave devices have an address assigned them during their manufacture; it differentiates them from other slaves on the bus.

- A single bus cannot accommodate two slave devices with the same address.
- The master or host uses this address to select the slave device it wants to communicate with.

C. I2C TERMINOLOGY

I2C protocol makes frequent use of a number of terms or expressions that you should familiarize yourself with.

Bit transfer

I²C protocol uses synchronous communication. This means that every bit that's transferred on the SDA data pin has to be accompanied by a clock pulse on the SCL pin. And what's more, the SDA bit is only valid when the SCL signal is at level "1".

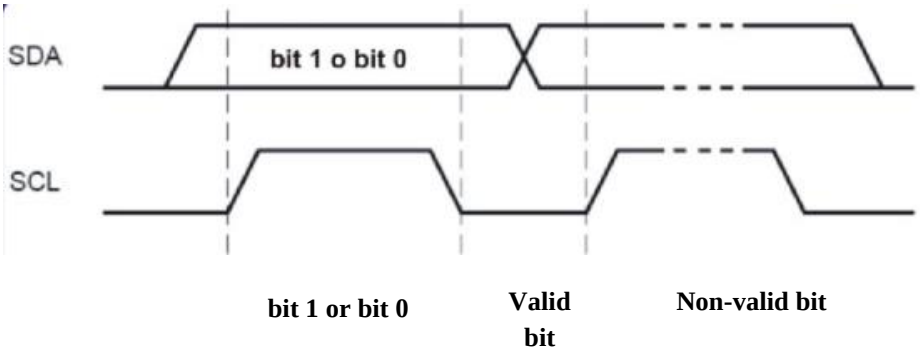


Figure 2

Start/Stop Condition

The host initiates all transfers by sending a sequence or Start (S) condition and terminates transfers with a Stop (P) condition. They occur just once. Have a look at the Figure 3

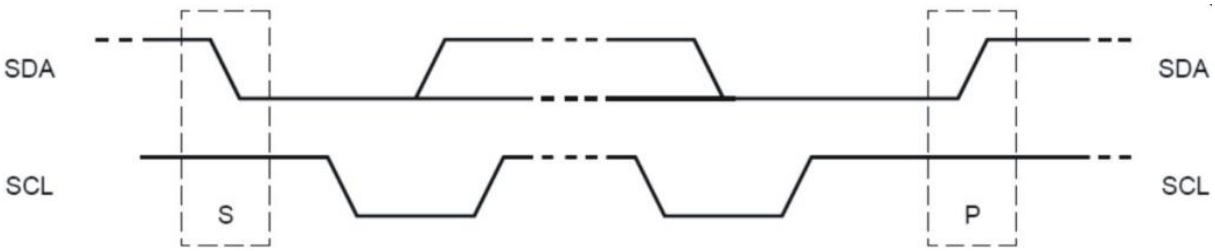


Figure 3

The Start condition (S) is issued when the SDA data signal goes to level “0” and the SCL signal is on level “1” (falling edge). This is a sign that something is about to be transmitted on the bus and as a result all the slave devices “hooked up” or connected to the bus “listen” to the information that follows. The bus is said to be “occupied” after the Start condition is issued.

The Stop condition is issued when SCL Clock signal goes to “1” and causes a rising edge on the SDA signal. The bus is then said to be “vacant” until the next Start condition is issued.

The Acknowledge Bit

We’ve already mentioned that data on the I2C bus is transferred in 8-bit packets (bytes) and that each byte starts with the most significant bit (MSB). There is No limitation on the number of bytes however each one must be followed by an Acknowledge bit or ACK/NACK as shown in the figure below. This bit signals whether the device is ready to proceed with the next byte.

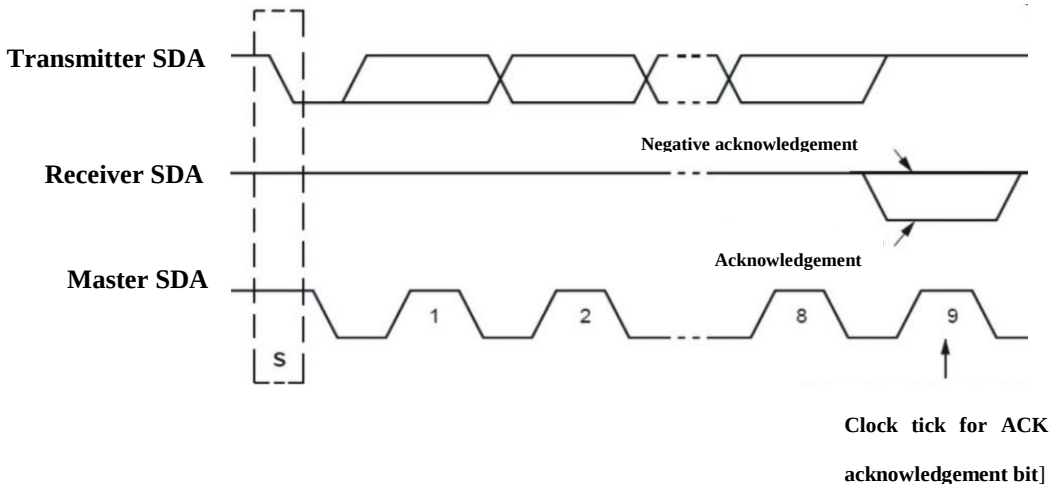


Figure 4

For all data bits including the Acknowledge bit, the master must generate clock ticks. If the slave device does not acknowledge transfer this means that there is no more data or the device is not ready for the transfer yet. The master device must either generate a Stop or Repeated Start condition.

This Acknowledgment bit, which is the ninth of the transmission, is followed by a clock tick as indeed all others are; the pulse is generated by the master or host.

But hang on a minute…who generates the ACK/NACK bit? The answer is “whoever just received a bit”. If the master sends a byte to the slave then it’s be the slave that sends the ACK/NACK bit to the master. If, on the other hand, it’s the slave that sends a byte to the master, then it’s the master that sends the ACK/NACK bit to the slave.

If the ACK/NACK bit is on level “0” this means that the byte has been received (ACK). If the recipient’s occupied, if it’s unable to receive a byte or wasn’t expecting one, the bit is a level “1” (NACK).

ACK is an abbreviation of Acknowledgement. And NACK, as you might imagine, is an abbreviation of Negative Acknowledgement. In short, this bit is a kind of “acknowledgement of receipt”.

It's like when you send a letter by certified mail. Any device, whether it be a master or a slave, that's just received a byte, must acknowledge its receipt with an ACK ("0") bit or a NACK ("1") bit.

The I2C Transaction Frame

All I2C transfers consist of a frame of one or more bytes. Have a close look at the Figure 5.

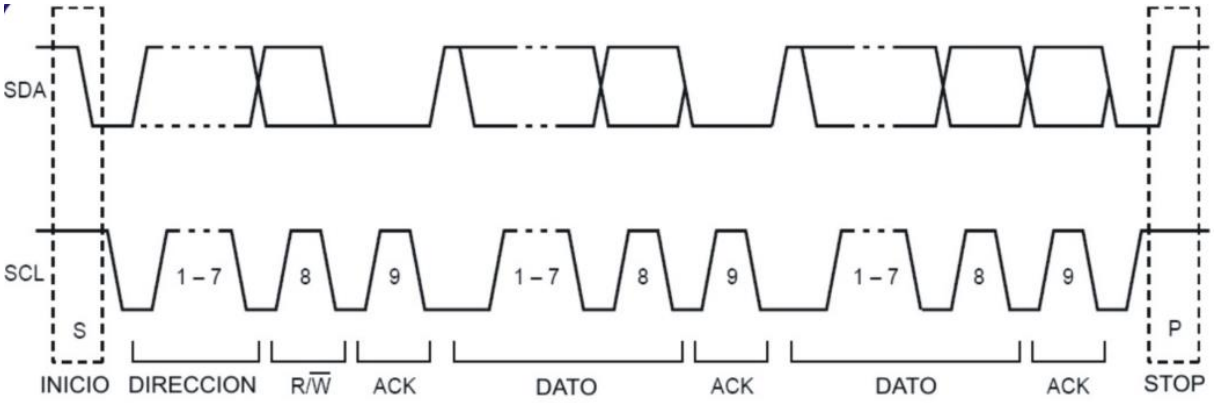


Figure 5

After the Start (S) condition the master or the slave transmits the address of the slave it wants to communicate with. This address consists of seven bits plus an eighth one that says whether it wants to write (R/W=0) to or read (R/W=1) from the slave.

Remember that as many as 128 addresses (2^7) may be represented with a seven bit address. You can literally "hook up" dozens of slave devices to a single bus using just two pins on the master controller (SDA and SCL). All the slave devices on the same bus must have different addresses though.

NOTE: The latest versions of the I2C protocol can support 10 bit addresses; this allows them to select up to around 1023 different devices.

Only the slave with the address indicated by the host can answer; the others keep "sleeping". The slave's address is decided by its manufacturer. It may be fixed or variable and you'll find it on the data sheet of the model you're going to use.

Once the master has transmitted the address to the slave and the slave has responded by sending the master an ACK or a NACK bit, the master begins to transmit the bytes to execute the operation and the slave to receive them. The transaction format finishes with the Stop (P) condition.

In this information frame (Figure 6) the master transmits n bytes towards a slave in write mode (R/W=0). The slave responds to each byte it receives from the master or slave with an ACK/NACK acknowledgement bit.

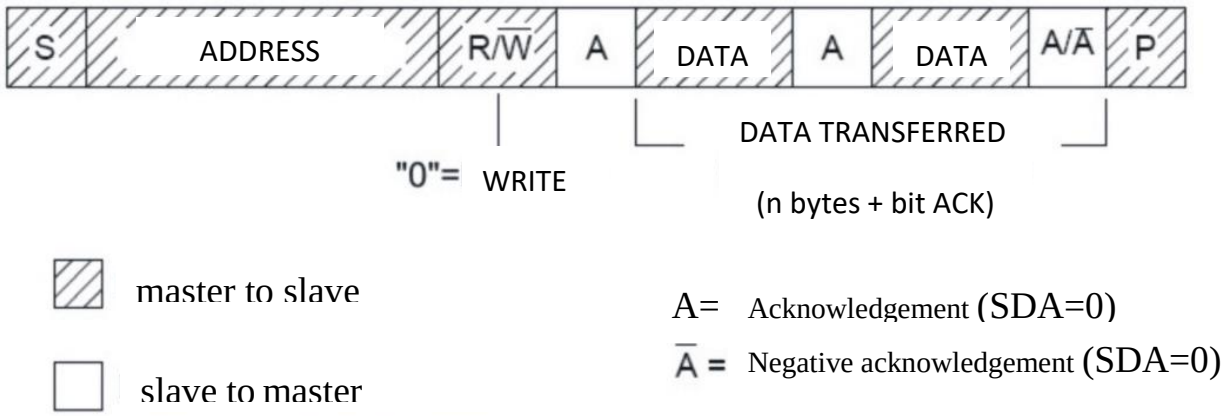


Figure 6

Another frame follows(Figure 7): the master receives data from the slave device which is in read mode. (R/W=1). On this occasion it's the slave that transmits data to the master which responds with an ACK/NACK bit.

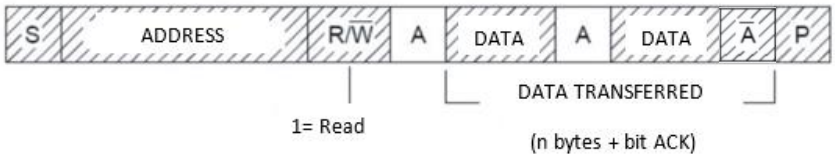


Figure 7

The Figure 8 is just for interest's sake: it shows the SCL (in yellow) and SDA (in green) signals for an I2C frame as seen on a measuring instrument like an oscilloscope or a logic analyser. Study it carefully because I'll be asking some questions about it later.

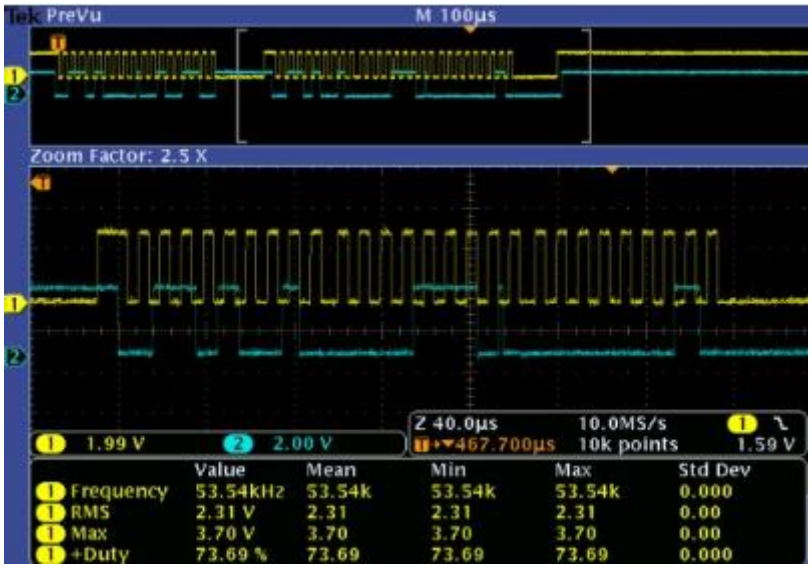


Figure 8

3. THE WIRE LIBRARY

Now that you're a bit familiar with the Terminology employed in I2C protocol it's time you started to use it a bit. And to help you out you've got a library at your disposal.

This one is called the "Wire" library and it was installed together with the Arduino Integrated Development Environment (IDE). The functions contained in this library manage the electronic circuits or hardware integrated in the Arduino controller to implement what we called "SSP" or Synchronous Serial.

Now, all you need is a little bit of patience while we examine each one of these functions.

- **The Wire.begin() Function**

This function initiates the wire library and joins the I2C bus as a master or slave. On the Arduino UNO board pins A4 and A5 become SDA and SCL signals respectively and you mustn't connect any other sort of device to them. This function should normally be called only once. It usually forms part of setup().

Syntax:

Wire.begin(address);

address: This is an optional seven bit integer (between 0 and 128). If it isn't indicated it is assumed that the device, in this case the Arduino UNO, will function as the master or host. If, on the other hand, it is indicated, the number represents the address. In this case, it is assumed that the Arduino UNO will function as a slave.

Examples:

```
Wire.begin();          //Initiates the I²C bus and the Arduino functions as a host or master
Wire.begin(18);        //Initiates the I²C bus and the Arduino functions as a slave. The address
                       //assigned is 18 (0x12).
```

- **The Wire.beginTransmission() Function:**

This function generates the Start (S) sequence and sends the slave's address in write mode (R/W=0). It subsequently queues bytes for transmission with the **write()** function and transmits them by calling **endTransmission().**

Syntax:

Wire.beginTransmission(address);

address: a seven-bit integer (between 0 and 128) which represents the address of the device the slave wants to talk to.

Example:

```
Wire.begin();                  //Initiates the bus in Master mode.

Wire.beginTransmission(18);    //Initiates the transmission of data to Slave No. 18 (0x12)
```

- **The Wire.write() Function:**

This function writes data from a slave device in response to a request from a master, or queues bytes for transmission from a master to slave device (in-between calls to **beginTransmission()** and **endTransmission()**).

Syntax:

Wire.write(value);

Wire.write(string);

Wire.write(data,length);

value: a value to send as a single byte or eight bits

string: a chain of characters comprising several bytes

data: an array of data to be sent as bytes

length: the number of bytes to transmit

Example:

```
Wire.begin();                    //Initiates the bus in master mode.

Wire.beginTransmission(18);      //Initiates data transmission to slave No 18 (0x12)

Wire.write("Hello");             //Transmits the "hello" string to the slave
```

- **The Wire.endTransmission() Function**

This function ends a transmission to a slave device that was begun by beginTransmission() and transmits the bytes that were queued by write(). In other words, it generates the Stop condition (P).

Syntax:

Wire.endTransmission(mode);

mode: may be TRUE or FALSE. If true, **endTransmission()** sends a stop message after transmission, releasing the I^2C bus. If false, **endTransmission()** sends a restart message (S) after transmission. This message is required by some I^2C devices and is optional. The bus will not be released, which prevents another master device from transmitting between messages. This allows one master device to send multiple transmissions while in control. *The default value is true.*

Returns:

The function returns the following error codes for assessment:

- 0: success
- 1: data too long to fit in transmit buffer
- 2: received NACK on transmit of address
- 3: received NACK on transmit of data
- 4: other error

Example:

```
Wire.begin();                  //Initiates the bus in master mode.
Wire.beginTransmission(18);    //Initiates data transmission to slave No 18 (0x12)
Wire.write("Hello");           //Transmits the "hello" string to the slave
Wire.endTransmission();        //Stop (S) Condition. End of transmission
```

- **The Wire.requestFrom() Function:**

This function is used by the master to request bytes from a slave device. The bytes may then be retrieved with the **Wire.available()** and **Wire.read()** functions.

Syntax:

Wire.requestFrom(address, quantity, mode)

address: this is a seven-bit integer (between 0 and 128) and represents the address of the device which bytes will be requested from.

quantity: the number of bytes to be received.

mode: This may be TRUE or FALSE. TRUE will send a stop message (P) after the request and reception of all the bytes, releasing the bus. FALSE will send a constant restart (S) after the request thus keeping the connection active. This message is required by some I^2C devices and is optional. The default value is true.

- **The Wire.available() Function:**

This function returns the number of bytes available for retrieval with read(). This should be called on a master device after a call to **requestFrom()** or on a slave inside the onReceive() handler. These bytes may be subsequently read using the **Wire.read()** function.

Syntax:

Wire.available();

- **The Wire.read() Function**

This function reads a byte that was transmitted from a slave device to a master after a call to **requestFrom()** or was transmitted from a master to a slave.

Syntax:

Wire.read();

- **The Wire.onReceive() Function:**

This function registers another function to be called each time a slave device receives a transmission from a master. This situation occurs when the slave receives its own address in write mode (R/W=0).

Syntax:

Wire.onReceive(function);

function: registers the function to be called when a slave device receives a transmission from a master. This function usually reads the bytes that the master writes to it.

Example:

```
Wire.onReceive(handler);        //Executes the "handler" function when the master
                                //indicates that it wants to transmit bytes to the slave.
                                //"handler" function. It should take a single integer
                                //parameter (read the number of bytes transmitted from
                                //the master). It gives no return.

void handler(int number bytes)

{

…

…

}
```

- **The Wire.onRequest() Function**

This function registers another function to be called when a master requests data from this slave device. This happens when the slave has received its own address in read mode (R/W=1).

Syntax:

Wire.onRequest();

handler: This is the name of the function to be called every time the master requests data from it. This function usually transmits the bytes requested.

Example:

```
Wire.onRequest(handler);      //Executes the handler function when the master
                              //requests data from the slave
                              //"Handler" function. Returns nothing. It confines itself to transmitting
                              //the data requested by the master

void myHandler()

{

…

…

}
```

4. GETTING SOME IDEAS STRAIGHT

As I've already commented, the "Wire" library comes with the Arduino Integrated Development Environment (IDE) included. It also contains a number of examples on usage and these are automatically installed at the same time as the library. They're quite simple and I recommend having a close look at them before you start the practice area. They'll give you an idea of how you should use the functions related to the I2C protocol.

Remember that the Arduino can function as an I2C master or host device, or, on the other hand as an I^2C slave device. When it functions as a master or host its task is to control all the I^2C slave devices on the bus. When it functions as a slave it's controlled by a master which could be another Arduino or any other controller.

To have a look at the examples all you have to do is select the **File Examples** and **Wire** options as indicated in the Figure 9. I think the following are the four most interesting examples:

- ✓ **master_writer**: configures the Arduino as an I^2C master device for transmitting data
- ✓ **slave_receiver**: configures the Arduino as a slave data receiver.
- ✓ **master_reader**: configures the Arduino as a master data receiver.
- ✓ **slave_sender**: configures the Arduino as a slave data transmitter.

These examples cover the four possible situations shown in the Figure 9.

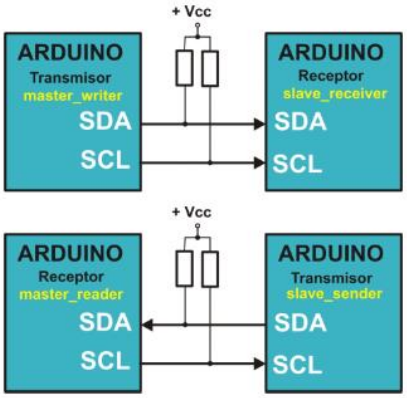


Figure 9

Imagine you've got two Arduinos and you want them to communicate with each other using I2C protocol. In the first examples you get one Arduino to function as an I2C master device that transmits data and the other as an I²C slave device that receives data.

In the other two examples you can get one Arduino to function as an I2C master device that receives data and the other as an I²C slave device that transmits data.

5. THE SRF02 ULTRASONIC RANGE FINDER

Now that you've learnt a bit about the theory of the protocol used on the I2C bus, it'd be a good idea see a device that uses it. So that's why we're going to have a look at the SRF02 ultrasonic range finder manufactured by the English firm Devantech Ltd.

A. OPERATIONAL PRINCIPLES

The Figure 10 shows the spectrum of frequencies audible to the human ear.

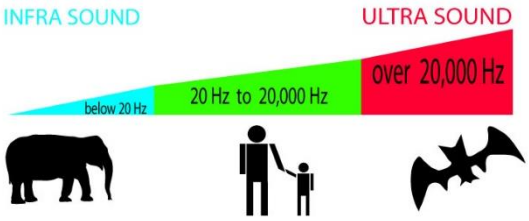


Figure 10

Infrasound is on the left of the spectrum; it occurs at below 20 Hz; we can't perceive it. In the middle of the spectrum between about 20 Hz and 20 KHz are the so called audible sounds.

From 20 KHz onwards we enter the range of ultrasound. These signals or sound waves lie beyond the threshold of human hearing; in other words, we are unable to perceive them.

I'm sure you're aware that some mammals like dolphins or bats use ultrasound to navigate. This phenomenon is known as "echolocation" and it works in much the same way as radar. These animals emit ultrasonic signals that bounce off the objects around them; they "listen" to the rebound and their brains create an image of their surroundings.

An ultrasonic range finder works on the same principles. A capsule emits an ultrasonic signal. This bounces off an object (a wall, for example) and creates an echo that goes back to the range finder. It then measures the time transpired between the signal and the echo.

Ultrasonic waves move at the speed of sound: 343 m/s through the air at sea level at a temperature of 20ºC and relative humidity of 50%. This speed may change in other conditions or media such as water, wood or concrete. That is to say that 343 m/s means that sound travels one metre in about 3 mS (1 / 343). We could draw up a table that shows the time an ultrasonic signal takes to cover certain distances.

DISTANCE	TIME	DESCRIPCTION
1 cm	0.00002915 " = 0.02915 mS = 29.15 µS	(1 / 343) / 100
1 m	0.002915 " = 2.915 mS = 2915 µS	1 / 343
1 Km	2.915 " = 2915 mS = 2915000 µS	(1 / 343) * 1000

However, you should bear in mind that we're talking about measuring the time that the ECHO takes to come back. In other words, the sonic signal has to make a return journey. So according to this logic, if you emit a signal and you receive the echo 10 mS later, what will have been the total distance covered by the signal? **10 mS / 2,915 mS = 3,43 m.**

And how far away is the object really? Well given that the signal made a journey back and forth the distance is calculated according to the following equation: **3.43 / 2 = 1.71 m.**

Now you're probably asking yourself how we generate ultrasound. It's actually quite simple: capsules or "transducers" transform an electric signal into a movement that causes the air to vibrate. It's much the same as what you do with your vocal chords when you speak or what the diaphragm of a loudspeaker does when it reproduces a sound. In this case, we use piezoelectric capsules and they vibrate at a frequency of over 20 KHz. We're going to use capsules similar to the ones in the figure on the right for our experiment; they vibrate at 40 KHz.

B. FEATURES AND CONNECTIONS OF THE SRF02

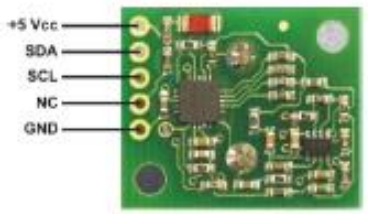
These are the most important features of the SRF02 ultrasonic rangefinder:

- ✓ Range: from 16 cm to 6 m (in theory)
- ✓ Power: + 5 V a 4 mA
- ✓ Ultrasonic frequency: 40 MHz
- ✓ Size: 24 mm x 20 mm x 17 mm
- ✓ Analogue Gain: Automatic 64 step gain control

- ✓ Connection Modes: 1 - Standard I²C Bus 2 - Serial Bus (connects up to 16 devices to any UART serial port)
- ✓ Full Automatic Tuning: No calibration, just power up and go
- ✓ Units: Range reported in µS, mm or inches.
- ✓ The default shipped address of the SRF02 is 224 (0xE0). It can be changed by the user to any of 16 addresses: E0, E2, E4, E6, E8, EA, EC, EE, F0, F2, F4, F6, F8, FA, FC or FE, therefore up to 16 different addresses can be used.

The SRF02 functions as an I2C slave device and includes its own controller responsible for making measurements and calibrations and then transmitting them to the host controller, our Arduino

These are its electrical connections:



PIN	NAME	DESCRIPTION
1	+5v Vcc	Voltage 5 V
2	SDA	I2C bus data line
3	SCL	I2C time line
4	Mode	Left unconnected
5	GND	Ground

The Figure 11 shows a life size image of the ultrasonic range finder, a simplified representation of it and the electrical symbol that we'll be using for it in the diagrams to come.

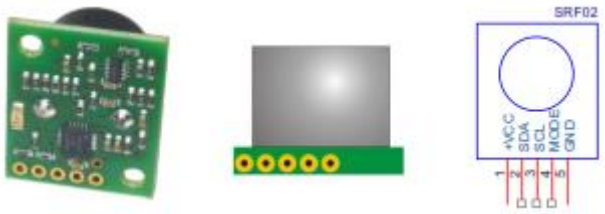


Figure 11

C. INTERNAL REGISTERS

Now that you're going to be using the SRF02 ultrasonic rangefinder you should be aware that there are a large number of I2C devices that are controlled in a similar way. You'll be able to apply what you learn here to these other devices. As usual you'll have to refer to the manufacturer's specifications. There you'll find the registers that are available, what you can write in them, what you can read etc.

For practical purposes the SRF02 functions like a memory with different positions or internal registers. The ones you can read provide you with information on the measurements taken. There are

others you can write in. The device behaves one way or the other depending on the value you assign it. A summary of the internal registers of the SRF02 appear in the table below.

When the device reads register number 0 it returns the internal version of the firmware that controls it. Don't forget that the SRF02 has its own controller.

LOCATION	READ	WRITE
0	INTERNAL VERSION OF I2C DEVICE FIRMWARE	COMMAND REGISTER
1	UNUSED (READS 0X80)	N/A
2	RANGE HIGH BYTE	N/A
3	RANGE LOW BYTE	N/A
4	AUTOTUNE MINIMUM - HIGH BYTE	N/A
5	AUTOTUNE MINIMUM – LOW BYTE	N/A

Locations 2 and 3 are the sixteen bit unsigned result from the latest ranging - high byte first. The meaning of this value depends on the command used and gives the range in inches or cm or the flight time in µS.

Locations 4 and 5 don't have any specific function. The SRF02 performs a number of tuning cycles to determine the minimum distance it can measure; this process is automatic and has No impact on scan time. This minimum distance can vary according to working conditions and is recorded in the locations; you may well see different distances from one moment to the next.

D. COMMANDS

When you write something in the No 0 location you're sending a command to the SRF02. Here's a list of the available commands:

Command		Action
Decimal	Hex	
80	0x50	REAL RANGING MODE - RESULT IN INCHES
81	0x51	REAL RANGING MODE - RESULT IN CENTIMETRES
82	0x52	REAL RANGING MODE - RESULT IN MICRO-SECONDS
86	0x56	FAKE RANGING MODE - RESULT IN INCHES
87	0x57	FAKE RANGING MODE - RESULT IN CENTIMETRES
88	0x58	FAKE RANGING MODE - RESULT IN MICRO-SECONDS
92	0x5C	TRANSMIT AN 8 CYCLE 40KHZ BURST - NO RANGING TAKES PLACE
96	0x60	FORCE AUTOTUNE RESTART - SAME AS POWER-UP. YOU CAN IGNORE THIS COMMAND.
160	0xA0	1ST IN SEQUENCE TO CHANGE I2C ADDRESS
165	0xA5	3RD IN SEQUENCE TO CHANGE I2C ADDRESS
170	0xAA	2ND IN SEQUENCE TO CHANGE I2C ADDRESS

Real Ranging

Any one measures the distance of an object from the SRF02. The device emits an 8 cycle ultrasonic burst at 40 KHz. It then waits to receive an echo - if there is one, that is. Work out the time the echo takes to come back and, based on that, work out the distance the way I told you to. These three commands return a result in inches, centimetres or microseconds.

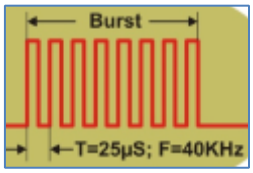


Figure 12

TAKE NOTE: The SRF02 takes some 66 mS to perform a complete measurement. In other words, when you execute any one of these commands, you'll have to wait this long to be able to read the results on register 2 and 3.

Fake Ranging

The SRF02 does not emit any kind of ultrasonic burst and cannot therefore measure the distance between the range finder and another object. These commands are designed to detect and measure bursts emitted by other SRF02 devices. Imagine an environment in which there are several SR02 sensors. These three commands also return a result in inches, centimetres or microseconds.

Burst

Bursts don't perform any measurement. A burst is an eight cycle ultrasonic signal emitted at 40 KHz. It's used as a warning signal or a synchroniser in an environment where there are several SRF02 sensors.

Restart

Each time you send this command to the SRF02 it's like you're powering up again; it performs the initial tasks of adjustment and calibration.

Changing the SRF02 Address

Just like any I²C device, the SRF02 has a default shipped address which is assigned during manufacture: 224 (0xE0). Nevertheless, this can be changed to any one of the sixteen following addresses:

D.	224	226	228	230	232	234	236	238	240	242	244	246	248	250	252	254
H.	E0	E2	E4	E6	E8	EA	EC	EE	F0	F2	F4	F6	F8	FA	FC	FE

Let's say you want to assign the address 240 (0xF0). You'd have to transmit the following sequence: 160, 165, 170, 240 (0xA0, 0xA5, 0xAA, 0xF0).

E. EXAMPLES OF INFORMATION FRAMES

Just a couple of examples will give you a good idea of what the data frames circulating through the I²C bus are like when you perform certain operations with the SRF02 ultrasonic range finder and what functions from the "Wire" library create them.

Reading the internal version of the SRF02 firmware

You're already aware that the SRF02 has its own controller with its own program or "firmware" recorded in it. It manages the device's internal operation and takes care of communication with the master or the host which in this case would be your UNO Arduino. This example shows the data frame that's created when the master, your Arduino, requests to see the firmware. Have a look at the Figure 13.

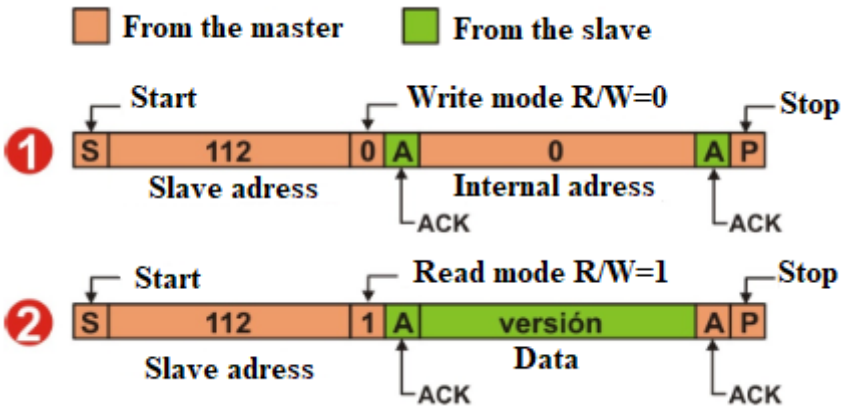


Figure 13

- The Arduino, the master in this instance, establishes communication with the SRF02 by sending the start sequence, **S**, its address and the write mode: "**Wire.beginTransmission(112);**". Next it writes the internal address of the SRF02 it's going to read - 0 in this case; the address also contains the "**Wire.write(byte(0));**" version of the firmware. It finishes the frame by sending the stop sequence, **S**, "**Wire.endTransmission();**".

- 2. The Arduino establishes communication with the SRF02 by sending the start sequence, **S**, its address and the read mode and the number of bytes to be received (1): "**Wire.requestFrom(112,1);**". Using "**Wire.available()**" it waits to receive a byte from the SRF02; this is read by "**Wire.read();**".

Measuring distance

Using this formula (Figure 14) the Master orders a reading of the distance in centimetres between the SRF02 and a given object.

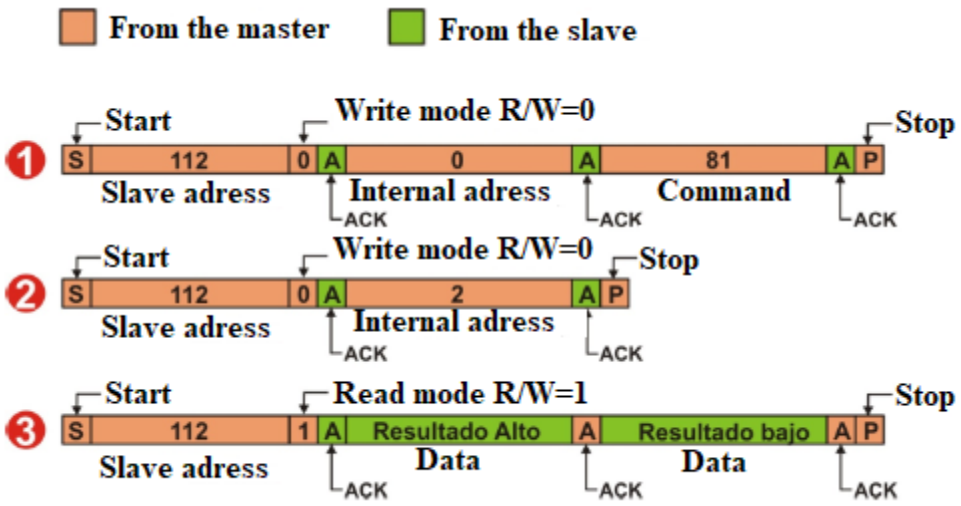


Figure 14

- 1. The Arduino, the master in this instance, establishes communication with the SRF02 by sending the start sequence, S, its address and the write mode: "**Wire.beginTransmission(112);**". Next it writes the internal address of the SRF02 it's going to read - 0 in this case; the address contains the command register "**Wire.write(byte(0));**". Using **"Wire.write(byte(81));"** it sends the command to make a measurement in centimetres. It completes the procedure by sending the Stop sequence, **S**, **"Wire.endTransmission();"**.

- 2. The Arduino, or Master in this instance, establishes communication with the SRF02 by sending the start sequence, S, its address and the write mode, "Wire.requestFrom(112);". Then it writes the internal address of the SRF02 it's going to read, 2 in this case; this contains the result of the measurement **"Wire.write(byte(2));".** It concludes the procedure by sending the stop sequence, S, **"Wire.endTransmission();".**

- 3. The Arduino, or Master in this instance, establishes communication with the SRF02 by sending the start sequence, S, its address and the read mode and the number of bytes to be received (2), **"Wire.requestFrom(112,2);".** Using "Wire.available()" it waits to receive the two bytes from the SRF02 and uses two **"Wire.read();"** functions to read them.

NOTE: The default address for the SRF02 Slave is 224 (0xE0). As I2C protocol demands that addresses have to comprise seven bits and not eight, the value 224 should be 112 (224/2).

6. THE DS1307 REAL-TIME CLOCK AND CALENDAR

Here's another device you can "seat" on the I2C bus next to the SRF02 ultrasonic range finder. This one is an integrated circuit called the DS1307; it contains a real-time clock and calendar. The clock/calendar provides seconds, minutes, hours, day, date, month, and year information. You'll find the datasheet amongst the supplementary material provided by the manufacturer.

You can use an external battery to power it so that it always shows the time even when the main system fails.

It's also got a 56 byte RAM memory where you can store all kinds of information. This may be volatile or not if you power it, as I have just suggested, with an external battery.

So when you get to the practice area you'll have an I2C bus controlled by the UNO Arduino at your disposal; this acts as the master or host of two slaves, the SRF02 and the DS1307:

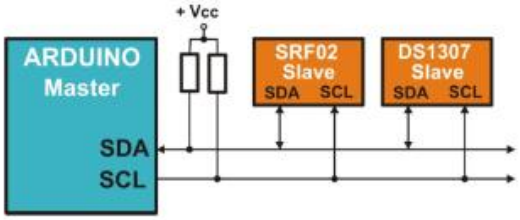


Figure 15

A. THE FEATURES OF THE DS1307

This is an integrated circuit made by the Dallas Semiconductors company acquired by Maxim Integrated Products (www.maximintegrated.com) in the year 2001.

The following are its most important features:

- It's a real-time clock and provides seconds, minutes, hours, the day of the week, the day of the month, the month and the year.
- It includes corrections for leap years to the year 2099.
- It contains 56 bytes of NV SRAM or Non volatile RAM memory powered by an external Non volatile battery.
- I2C interface with the host or master.
- Programmable square-wave output signal.
- The DS1307 has a built-in power-sense circuit that detects power failures and automatically switches to the backup supply.
- Low power operation extends battery backup run time and consumes less than 500nA in battery backup mode with oscillator; the timekeeping operation continues and the RAM memory data is conserved while the part operates from the backup supply.
- 8-Pin DIP and 8-Pin SO minimizes required space.

B. ENCLOSURE AND PIN-OUT

You'll be using a DS1307 in the practice area with an eight-pin DIP format as shown in the Figure 16

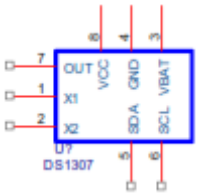


Figure 16

The symbol represents this appliance and you'll be using it in the circuit diagrams in the practice area. Below is a table with a description of each pin:

Nº	NAME	DESCRIPTION
1	X1	X1 is the input to the oscillator and is connected to an external quartz crystal 32.768kHz oscillator.
2	X2	X2 is the output to the oscillator and is connected to an external quartz crystal 32.768kHz oscillator.
3	VBAT	Backup supply input for any standard 3V lithium cell or other energy source.
4	GND	The earth for the main power supply
5	SDA	Serial Data Input/Output. SDA is the data input/output for the I2 C serial interface.
6	SCL	Serial Clock Input. SCL is the clock input for the I2C interface.
7	SQWE/OUT	Square wave/output driver.
8	VCC	Primary +5V power supply.

C. INTERNAL REGISTERS

The DS1307, just like almost all I2C devices, contains a series of internal registers that can be read or written. As we're dealing with a real-time clock and calendar, it's logical to assume that the data that you enter in certain registers is for adjusting the time and date. In exactly the same way, the data that you find in some of the registers tells the time and date.

The DS1307 also has a total of fifty-six additional general purpose registers. Just think of them as a small RAM memory where you can store some more bytes of data.

Both the real-time clock and calendar data and the data you store in the RAM memory continue to be updated even if the main power supply fails. In case of a power failure, connect up a standard 3V lithium cell or other energy source between the VBAT and GND pins. Have a look at the basic diagram of the Figure 17.

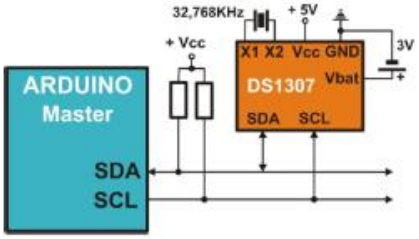


Figure 17

This table shows all the internal registers of the DS1307; you can read or write data on all of them. The first eight are designed for the real-time clock and calendar tasks and represent BCD binary values. The remaining fifty-six correspond to the other positions in the general purpose RAM memory.

Address	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Function	Range
0	CH	10 seconds				Seconds			Seconds	00-59
1	0	10 minutes				Minutes			Minutes	00-59
2	0	12	PM/AM	10 hours	Hours				Hours	1-12
		24	10 hours							00-23
3	0	0	0	0	0	DAY			Day	01-07
4	0	0	10 Date		Date				Date	01-31
5	0	0	0	10 month	Month				Month	01-12
6	10 Year								Year	00-99
7	OUT	0	0	SQWE	0	0	RS1	RS0	Control	-
8-63	56 registers for data								RAM	0-255

NOTE: The day of the week in register No. 3 is represented by three bits between 1 and 7. If we assume that the 1 corresponds to Mondays, 2 will be Tuesdays, 3 Wednesdays and so on.

KEY:

- ✓ **CH**: Bit 7 of register No 0 is the clock halt (CH) bit. When this bit is set to 1, the oscillator is disabled and consumption reduced. When cleared to 0, the oscillator is enabled. It may be useful if you're only going to use the DS1307's RAM memory.
- ✓ **12/24**: Bit 6 of the hours register is defined as the 12-hour or 24-hour mode-select bit. The DS1307 can be run in either 12-hour or 24-hour mode. When high, the 12-hour mode is selected. In the 12-hour mode, bit 5 is the AM/PM bit with logic high being PM. In the 24-hour mode, bit 5 is the second 10-hour bit (20 to 23 hours). The hours value must be re-entered whenever the 12/24-hour mode bit is changed. Bit 5 and bit 4 together represent the tens of hours.
- ✓ **OUT**: Bit 7 of the output control (OUT). This bit controls the output level of the SQW/OUT pin when the square-wave output is disabled. If SQWE = 0, the logic level on the SQW/OUT pin is 1 if OUT = 1 and is 0 if OUT = 0. On initial application of power to the device, this bit is typically set to a 0.

- ✓ **Square-Wave Enable or SQWE**: Bit 4 of the No. 7 control register. This bit, when set to logic 1, enables the oscillator output. The frequency of the square-wave output depends upon the value of the RS0 and RS1 bits. With the square-wave output set to 1Hz, the clock registers update on the falling edge of the square-wave. On initial application of power to the device, this bit is typically set to a 0.
- ✓ **RS1:RS0**: Bits 1 and 0 of the No. 7 control register, (Rate Select (RS[1:0]). These bits control the frequency of the square-wave output when the square-wave output has been enabled. On initial application of power to the device, these bits are typically set to a 1.

The table below may help you to understand the relationship between the OUT, SQWE, RS1 and RS0 bits of the control register and the OUT/SQWE output pin of the device.

OUT/SQWE PIN	SQWE Bit	OUT Bit	RS1 Bit	RS0 Bit
1 Hz	1	X	0	0
4096 Hz	1	X	0	1
8192 Hz	1	X	1	0
32768 Hz	1	X	1	1
0	0	0	X	X
1	0	1	X	X

This table shows the ten primary digits from 0 to 9 represented in BCD binary code:

DIGIT	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

EXAMPLES

Enter the BCD binary value for registers of the DS1307 in different cases. You can also enter their equivalent decimal and hexadecimal values with the aid of a calculator.

Case 1: the time is 00:45:18…

Register number	BCD Binary	Dec.	Hex.

Case 2: the time is 21:35:23…

Register number	BCD Binary	Dec.	Hex.

Case 3: the time is 18:05:53…

Register number	BCD Binary	Dec.	Hex.

Case 4: Bearing in mind that Monday is the day 1 of the week, we're up to Thursday 12 March 2015…

Register number	BCD Binary	Dec.	Hex.

Case 5: it's Wednesday 16 December 2015 and the time 16:52:14…

Register number	BCD Binary	Dec.	Hex.

Case 6: you want the SQW/OUT pin to go to logic level "1".

Register number	BCD Binary	Dec.	Hex.

Case 7: you want to send an 8192 Hz square-wave signal on the SQW/OUT pin.

Register number	BCD Binary	Dec.	Hex.

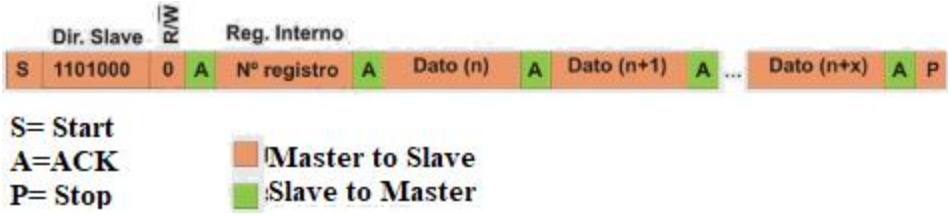
ACCURACY:

The accuracy of the clock depends on the accuracy of the crystal and the accuracy of the match between the capacitive load of the oscillator circuit and the capacitive load for which the crystal was trimmed. Crystal frequency drift caused by temperature shifts may create error. Interference or external circuit “noise” coupled into the oscillator circuit may cause the clock to gain time.

D. EXAMPLES OF INFORMATION FRAMES

To make things a little clearer, you’ll be seeing a series of examples in which the master (Arduino) accesses the DS1307 real-time clock and calendar to read or write something on one of its internal registers (including the ones that correspond to the RAM memory).

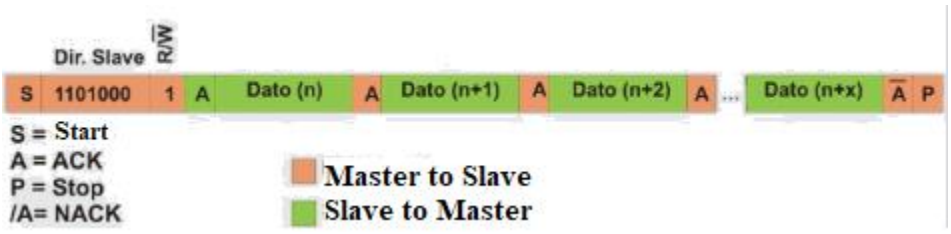
The master writes on the DS1307



After **S** start sequence, transmit the address of the DS1307 (104 = 0x68) in write (R/W=0): **“Wire.beginTransmission(104);”**. Next indicate the address of the internal register: **“Wire.write(byte(nº reg));”**. Based on this address, use the following functions to transmit the data: **“Wire.write(byte(Data_n));”**, **“Wire.write(byte(Data_n+x));”**... Finalize the sequence by sending the **S** stop condition using: **“Wire.endTransmission();”**.

The master reads from the current register of the DS1307

After the S start sequence transmit the address of the DS1307 (104 = 0x68) in read mode en (R/W=1) and the number of bytes to be read: **“Wire.requestFrom(104,n);”.** Using **“Wire.available();”** wait until the bytes start to come in and then read them using **“Wire.read();”**



The bytes received come from the last internal register of the DS1307 that was selected.

The master reads from a predetermined internal register of the DS1307

After the **S** start sequence, send the address of the DS1307 (104 = 0x68) in write mode (R/W=0): **"Wire.beginTransmission(104);"**. Indicate the address of the internal register you're going to read: **"Wire.write(byte(reg nº));"**.

After another **S** Start sequence, send the address of the DS1307 (104 = 0x68) in read mode (R/W=1) and the number of bytes to be read: **"Wire.requestFrom(104,n);"**. Using **"Wire.available();"** wait until the bytes you're going to read with **"Wire.read();"** start coming in.

E. THE DS1307 LIBRARY

- **The setRegister() Function**

This function writes any value on any of the internal registers of the DS1307. Bear in mind that the first eight registers (registers 0-7) are associated with the time functions and the other fifty-six are general purpose RAM registers.

Syntax:

setRegister(n, value);

n: the number of the internal register of the DS1307 to be written (between 0 and 63).

value: this is the value to be saved in the selected register (between 0 and 255)

- **The setBCDtoRegister() Function:**

This function writes a BCD value on any one of the internal registers of the DS1307.

Syntax:

setBCDtoRegister (n, value);

n: the number of the internal register of the DS1307 to be written (between 0 and 63).

value: the BCD value to be saved in the selected register (between 0 and 99).

- **The getRegister() Function:**

This function returns an eight bit or one byte value that corresponds to the current contents of the register indicated.

Syntax:

getRegister(n);

n: the number of the internal register of the DS1307 to be read (between 0 and 63).

- **The resume() Function**

This function enables the real-time clock and calendar. Bit 7 in register No 0, CH, is set at level "0".

Syntax:

resume();

- **The standby() Function**

This function disenables the real-time clock and calendar. Bit 7 in register No 0, CH, is set at level "1". The consumption of the DS1307 drops considerably.

Syntax:

standby();

- **The getDate() Function**

This function returns the contents of the first seven registers of the DS1307. These contain all the information relevant to the real-time clock and calendar: the seconds, minutes, hours, the days of the week, the days of the month and the months and the years.

Syntax:

getDate(buffer);

buffer: This is the seven position byte array that carries the date and time contained in the DS1307:

buffer[0] = seconds (0-59)	**buffer[1] = minutes (0-59)**
buffer[2] = hours (1-12 o 0-23)	**buffer[3] = day of the week (1-7)**
buffer[4] = day of the month (1-31)	**buffer[5] = month (1-12)**
buffer[6] = year (0-99)	

- **The setSeconds() Function**

This function adjusts the seconds on the DS1307 real-time clock and calendar.

Syntax:

setSeconds(v);

v: the new value for the seconds between 0 and 59.

- **The setMinutes() Function**

This function adjusts the minutes on the DS1307 real-time clock and calendar.

Syntax:

setMinutes(v);

v: the new value for the minutes between 0 and 59.

- **The setHours() Function**

This function adjusts the hours on the DS1307 real-time clock and calendar.

Syntax:

setHours(v);

v: the new value for the hours between 1 and 12 or 0 and 23.

- **The setDow() Function**

This function adjusts the day of the week on the DS1307 real-time clock and calendar.

Syntax:

setDow(v);

v: the new value for the day of the week between 1 and 7.

- **The setData() Function**

This function adjusts the day of the month on the DS1307 real-time clock and calendar.

Syntax:

setData(v);

v: the new value for the day of the month between 1 and 31.

- **The setMonth() Function**

  This function adjusts the month on the DS1307 real-time clock and calendar.

Syntax:

setSMonth(v);

v: the new value for the month between 1 and 12.

- **The setYear() Function**

  This function adjusts the year on the DS1307 real-time clock and calendar.

Syntax:

setYear(v);

v: the new value for the day of the week between 0 and 99.

PRACTICE SECTION

Now it's time to create your own I2C bus and you'll be connecting with the SRF02 ultrasonic range finder as well as the DS1307 real-time clock and calendar. The exercises will give you the chance to experiment with both appliances and, what's even more important, use the functions in the "Wire" library to communicate with them.

As we have already said on a number of occasions, these exercises don't pretend to be any more than that: exercises. You can, and should, study, change and improve them. I'm sure they'll provide you with new ideas for your own projects and applications.

7. EXAMPLE 1: A VERSION OF FIRMWARE

This is the simplest one. The version of firmware that controls the SRF02 ultra sonic range finder is displayed on the LCD screen. You're going to start off by assembling the circuit according to the Figure 18

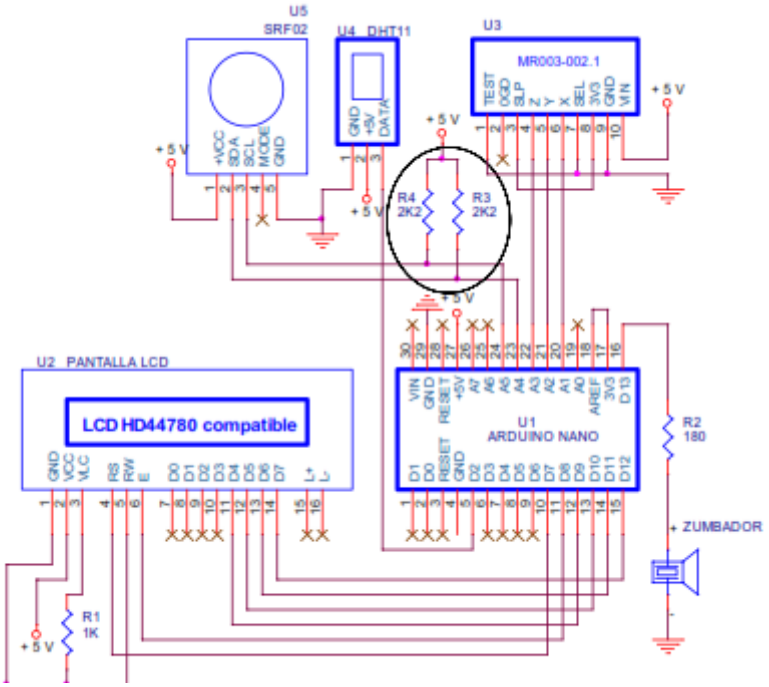


Figure 18

The A5 and A4 analogue inputs also behave like SCL and SCD signals respectively on the Arduino UNO I2C bus. They're connected to the R3 and R4 pull-up resistors and the SCL and SDA pins on the SRF02 ultra sonic range finder. Make sure you complete all the connections required.

The example (Figure 19) reads the internal register No 0 of the DRF02 and it returns the version of the firmware. The important thing about the program is the way it uses some of the functions from the "Wire" library. They're in the main loop() function.

```
void loop()
{
// STEP 1
  Wire.beginTransmission(112);
  Wire.write(byte(0));
  Wire.endTransmission();

// STEP 2
  Wire.requestFrom(112,1);

// STEP 3
  if(Wire.available())
    ver = Wire.read();

  lcd.print(ver);
  while(1);
}
```

Figure 19

Step 1: Select the SRF02 appliance and its internal register No. 0

Wire.beginTransmission(112); generates the S start sequence and sends the 112 address which selects the SRF02 device in write mode (R/W=0). Wire.write(byte(0)); selects the internal register No. 0, which contains the version of the firmware. Last of all, Wire.endTransmission(); generates the P Stop sequence.

Step 2: Reading the slave's No. 0 register

Wire.requestFrom(112,1); generates the S start sequence and sends the 112 address which selects the SRF02 ultrasonic range finder in read mode (R/W=1). On this occasion the aim is to read a byte.

Step 3: Receiving a byte from the SRF02Slave

Wire.available(); acknowledges having received a byte. **Wire.read();** performs the reading and stores the byte in the "see" variable. Next, it displays it on the LCD screen and **White(1);** terminates the program.

IMPORTANT: The SRF02 slave's default address is 224 (0xE0 in hex. or 11100000 in binary). According to I2C protocol the address of a slave comprises 7 bits, so the previous address would be 1110000 = 0x70 = 112. In other words, we divide the original 224 address by two and it becomes 112.

When you upload exercise, you'll see that it isn't particularly spectacular. All that happens is that a number that corresponds to the internal version of the SRF02 firmware will appear on the LCD screen.

8. EXAMPLE 2: DISTANCES

This is a really easy practical exercise. The screen will display the distance in centimetres between the range finder and an object.

The program is divided into five steps; have a close look at them:

Step 1: Communication with the SRF02 Slave is enabled by selecting the command register No. 0 and writing the value 81. This command initiates a new measurement of the distance in centimetres.

Step 2: The waiting cycle. The SRF02 requires a minimum of 65 mS to complete the measurement.

Step 3: Select the first register and save the result of the measurement. The most significant byte is stored in register No. 2 and the least significant one in No. 3.

Step 4: The reading of the two bytes in registers No. 2 and No. 3 expresses the result of the measurement.

Step 5: Both bytes are read and then combined to form a 16 bit integer. The LCD screen displays the result.

Download the program and test it by placing an object at different distances from the range finder. Measure the distances with a tape measure, a ruler or something similar and compare the results with the ones displayed on the LCD screen. Remember that the SRF02 has a minimum range of 16 cm and a maximum one of 600 cm (6 m).

A. A BRIEF DIGRESSION: OPERATIONS BETWEEN BITS

There's been no need to perform logical operations between bits in any of the variables occurring in the exercises you've been doing throughout. It's time to have a look at them now. As well as performing these operations in this exercise, I'm sure you'll be able to use them in future projects and applications and even to improve some of the previous and present exercises.

You're already aware that a variable is a RAM memory reservoir where you can store data. These variables may be bytes (8 bits), integers (16 bits) or long (32 bits). So now you're going to perform logical operations with the bits from any one of these three types of variables.

AND (&) Logic Function

The AND logic returns "TRUE" when the two bits also return "TRUE".

```
byte Data1 = B10110110;          //Data1 = 10110110 = 0xB6 = 182
byte Data2 = B10101011;          //Data2 = 10101011 = 0xAB = 171
byte Data3 = Data1 & Data2;      //Data3 = 10100010 = 0xA2 = 162
```

OR (|) Logical Function

The OR logic operation returns “TRUE” when either of the bits is “TRUE”.

```
byte Data1 = B10110110;          //Data1 = 10110110 = 0xB6 = 182
byte Data2 = B10101011;          //Data2 = 10101011 = 0xAB = 171
byte Data3 = Data1 | Data2;      //Data3 = 10111111 = 0xBF = 191
```

Exclusive OR or XOR (^) Logical Function

The XOR logic operation (which stands for "Exclusive OR") returns “TRUE” when the bits differ (i.e. one is true, the other is false).

```
byte Data1 = B10110110;          //Data1 = 10110110 = 0xB6 = 182
byte Data2 = B10101011;          //Data2 = 10101011 = 0xAB = 171
byte Data3 = Data1 ^ Data2;      //Data3 = 00011101 = 0x1D = 29
```

NOT (~) Negation Function

The NOT logic operation returns “TRUE” if its input is FALSE, and false if its input is true.

```
byte Data1 = B10110110;          //Data1 = 10110110 = 0xB6 = 182
byte Data3 = ~ Dato1;            //Data3 = 01001001 = 0x49 = 73
```

Movement to the Left (<<)

The bits contained in the variable move the number of positions indicated to the left.

```
byte Data1 = B10110110;          //Data1 = 10110110 = 0xB6 = 182
byte Data2 = B10101001;          //Data2 = 10101001 = 0xA9 = 169
byte Data3 = Data1 << 3;         //Data3 = 10110000 = 0xB0 = 176
byte Data4 = Data2 << 4;         //Data4 = 10010000 = 0x90 = 144
```

Movement to the Right (>>)

The bits contained in the variable move the number of positions indicated to the right.

```
byte Data1 = B10110110;          //Data1 = 10110110 = 0xB6 = 182
byte Data2 = B10101001;          //Data2 = 10101001 = 0xA9 = 169
byte Data3 = Dato1 >> 3;         //Data3 = 00010110 = 0x16 = 22
byte Data4 = Data2 >> 4;         //Data4 = 00001010 = 0x0A = 10
```

B. APPLICATIONS OF EXAMPLE

Let's have a look at how to apply some of these operations between bits in exercise you're working on now.

1.- When the **distance = Wire.read(); function** is executed the 8 most significant bits from the H7:H0 measurement performed by the SRF02 are loaded onto the 16 bit "distance" variable:

"distance" variable															
..	**..**	**..**	**..**	**..**	**..**	**..**	**..**	**H7**	**H6**	**H5**	**H4**	**H3**	**H2**	**H1**	**H0**

2.- When the **distance = distance << 8;** function is executed the contents of the variable move eight places to the left:

"distance" variable															
H7	**H6**	**H5**	**H4**	**H3**	**H2**	**H1**	**H0**	**..**	**..**	**..**	**..**	**..**	**..**	**..**	**..**

3.- Last of all, the **distance |= Wire.read();** function is executed. The OR function couples the contents of the "distance" variable with the result of reading the 8 least significant bits (L7:L0) of the measurement performed by the SRF02 ultrasonic range finder:

"distance" variable															
H7	**H6**	**H5**	**H4**	**H3**	**H2**	**H1**	**H0**	**L7**	**L6**	**L5**	**L4**	**L3**	**L2**	**L1**	**L0**

The two bytes created by the SRF02 after performing the distance measurement are stored in the "distance" variable. In other words, the two bytes issued by the internal registers No. 2 and No. 3 of the SRF02 become 16 bit integers. Nice little trick, isn't it? As I said before, I'm sure this idea of doing operations between bits will be really useful.

9. EXAMPLE 3: MORE DISTANCES

Here's another exercise on measuring distances. This time the SRF02 will provide results in centimeters, inches and microseconds and they'll be displayed on the LCD screen.

A new function, **measure(),** based on the previous example, has been created. It generates the sequence necessary for the SRF02 to perform a new measurement:

- ✓ **medir(cm);** //Performs the measurement in centimeters
- ✓ **medir(in);** //Performs the measurement in inches
- ✓ **medir(uS);** //Performs the measurement in microseconds

In all three cases the function returns the result in the "distance" variable; this result is then shown on the LCD screen.

One of the problems with this example is viewing the results on the screen. Each measurement may have a different number of digits or characters. The result of the measurement in centimetres may lie between 16 and 600, the one in inches between 6 and 236 and the one in microseconds between 930 and approximately 34900. If a measurement has fewer digits than the previous one, some of the old ones stay on the screen and this leads to an erroneous reading.

Imagine you obtain the following series of measurements: Cm= 550, In= 216 and uS= 32065. When the LCD screen displays these results they'll look like this:

C	m	=		5	5	0			I	n	=		2	1	6
				u	S	=		3	2	0	6	5			

Now suppose you get the following results in the next series of measurements:

Cm= 20, In= 8 and uS= 1166. If you don't "clean" the LCD screen, it'll look something like this:

C	m	=		2	0	0			I	n	=		8	1	6
				u	S	=		1	1	6	6	5			

What's going on? Well, the "leftovers" from the previous measure stay on the screen: in the Cm the 0, in the In the 16 and in the uS the 5. Have a close look:

C	m	=		2	0	0			I	n	=		8	1	6
				u	S	=		1	1	6	6	5			

What's the answer? This problem has already appeared in other exercises. We solved it by writing blank spaces (" ") on the screen or by wiping the screen completely clean using lcd.clear() or lcd.home(), and rewriting the results.

You're going to use a more "sophisticated" solution for this exercise. Have a close look at how we use the lcd.print() function. You'll Notice that we use it like this: n = lcd.print(distance);. The "n" variable returns the number of characters that have just been printed.

And what's the use of that? It tells you how many characters have been written in each case. What for? So you know how many blank spaces you have to print to clean the leftovers off the LCD screen.

Have close look at this extract from a program:

```
measure(in);                    //Measurement in inches

lcd.setCursor(13,0);

n=lcd.print(distance);          //display on the LCD

for(n; n<3; n++)

lcd.print(" ");                 //Complete with blank spaces
```

The distance is measured in inches: medir(in);. You position the cursor on the screen where you want the result to be displayed: lcd.setCursor(13,0);. The result is displayed: n=lcd.print(distance);. The "n" variable contains the number of characters displayed. The measurement in inches may be between 6 and 236, or, in other words, from one to three characters many be displayed.

The for(n; n<3; n++) prints as many the blank spaces (" ") as required to clean the screen of leftovers. If you display the number 193 (three characters) there's no need to print a blank space. If you display an 8 the next time (one character), then you'll have send two blank spaces to "wipe off" the 9 and the 3 from the previous display.

Do you like the trick? Well Now you know another "technique" that could be useful in the future.

10. EXAMPLE 4: COLLISION AVOIDANCE SYSTEM

Here's a good practical exercise. The idea is to detect the distance between the SRF02 ultrasonic range finder and an object or obstacle. As the object gets closer and closer to the SRF02 and passes certain minimum distances the piezo-electric buzzer connected to the D13 output pin starts to emit a warning signal at different frequencies.

You must have seen something similar in some cars - maybe there's one in your own vehicle; it's called an intelligent parking assist system.

Vehicles such as those in the illustration on the right have a number of ultrasonic sensors similar to the one we've just been talking about; they're fitted along the front and rear bumper bars. They give off sound waves which bounce off potential obstacles that may be in range of the vehicle. A warning signal, similar to the one in the exercise, advises the driver that there's an obstacle in range so he can take the necessary evasive action and avoid a collision.

Other more sophisticated systems display a visual representation of the situation and the distance of the obstacles on a dashboard screen. There are even ones that have sensors fitted on the sides of the vehicle that also assist with parking; they detect spaces or gaps, distances and the vehicle itself if necessary etc.

The program in the exercise isn't very complicated. Using **#define** the program establishes a number of minimum values and when the distance between the obstacle and the SRF02 is less than one of them, the program reacts and generates an appropriate warning signal.

Upload the program and experiment with it. You can try changing the minimum distances.

11. EXAMPLE 5: SPEED DETECTOR

This exercise is purely experimental and while it's not easy to check its accuracy it can give you a basic idea of what a speed detector is like. It works in much the same way as a speed radar gun or lidar although these devices obviously use other technologies like lasers; they're also far more accurate and have a significantly greater range. You can call this example a basic "homemade radar."

Let's see what the general idea is - have a look at the Figure 20:

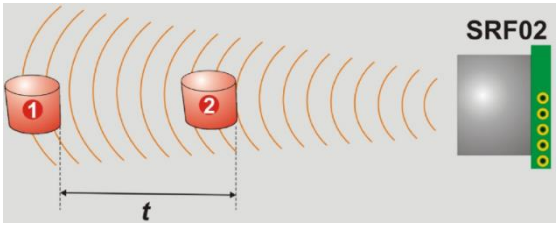


Figure 20

The SRF02 takes two different measurements: the distance of object 1 and the distance of object 2. The difference between the two measurements represents the distance or space travelled by

the object. If you know the period of time, t, between the two measurements, you'll be able to tell what speed the object was travelling at (speed = space / time).

In the example the interval between the first and second measurements is 250 mS (0.25"). We get this time from the delay(179); function and the time the dis_2=medir(cm); function takes to make the second measurement which is about 71 mS. So this is the interval between the two measurements: 179 mS + 71 mS = 250 mS.

We multiply the difference between the first and the second measurement by four. This difference represents the distance in centimetres travelled by the object in 0.25 seconds. When we multiply it by four we get the distance travelled by the object in a second (cm/s) or, in other words, the speed.

If the object exceeds a predetermined speed limit the alarm system goes off: the piezo-electric buzzer sounds and the speed is displayed on the LCD screen for two seconds. Download the program and do the trials as shown in the video included in the supplementary material.

12. EXAMPLE 6: AREA METRE

Some tradesmen such as painters, carpenters, bricklayers, decorators and others often use this instrument to help them to work out the size of an area they have to work on.

These metres are able to use ultrasound or laser light to measure distances. In this case, of course we'll be using the SRF02 ultrasonic range finder.

If you place a metre at one end of a wall and make the measurement with the device pointing at the opposite end you get the length of one of the sides. Next, you place the metre on the floor and make a second measurement with the device pointing at the ceiling; now you get the length of the other side. If you multiply the two distances you'll get the surface area of the wall. You're going to do something similar with this example.

Here's the circuit diagram (Figure 21). All we've done is connect a pushbutton to the Arduino UNO D3 pin which will function as an input. Each time it's pressed, it makes a new measurement.

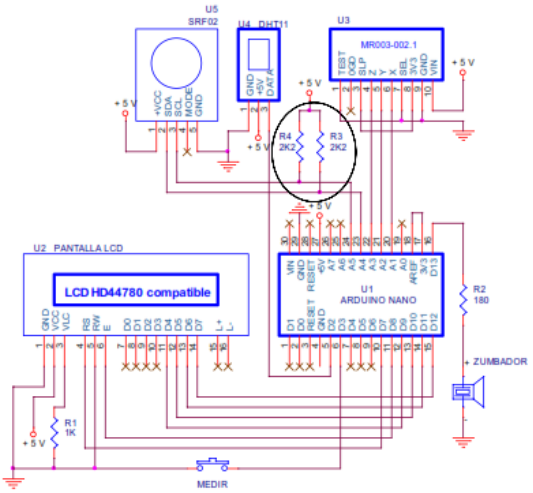


Figure 21

The program's very simple. When you press the pushbutton connected to the D3 pin, the program makes the first measurement (dis_1): one of the sides of the surface. When the pushbutton is pressed a second time (dis_2) it makes the other measurement: the other side of the surface. The program calculates the area by multiplying the two measurements - dis_1 * dis_2 - and displays the result in square metres.

It uses the same **measure()** function you used in previous examples. There's a new function too: **pulse_D3()**. It waits until it receives the 1-0-1 sequence from the D3 input pin and then generates a 100 mS sound signal on the D13 output, the one the piezo-electric buzzer's connected to.

13. EXAMPLE 7: REAL-TIME CLOCK

Here's the first exercise on using the real time clock and calendar in the I2C DS1307 device. This exercise is very simple. All you're going to do is read the first seven internal registers to get the hours, minutes and seconds and then display them in real time.

Here's the circuit diagram (Figure 22); have a close look at it.

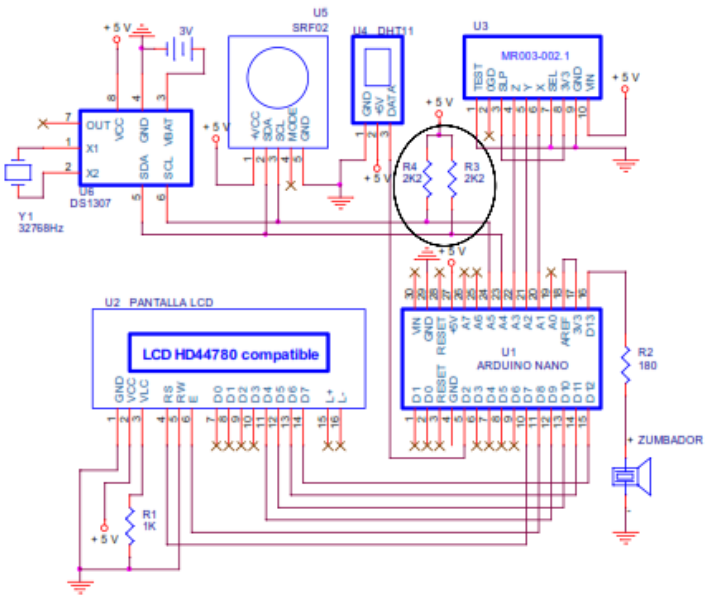


Figure 22

See how the same pins A4 and A5 (SDA and SCL) are connected to both the SRF02 ultrasonic range finder that you've been working with recently and the DS1307 device. This is called a "bus", and in this case it's a I2C bus: you've connected two or more devices to the same two pins.

There's a 3 V battery between pins 3 (VBAT) and 4 (GND) of the DS1307. This enables the clock to continue functioning and keep good time even when the main 5 V power is off. It also preserves the data you've saved in the fifty-six additional registers of the RAM memory.

Finally, we connect the quartz crystal between pins 1 (X1) and 2 (X2); this enables us to measure the time accurately. We'd be using a standard 32,768 KHz crystal in this case.

Remember that the white dot on the DS1307 represents the No. 1 pin. They're numbered consecutively in an anticlockwise direction.

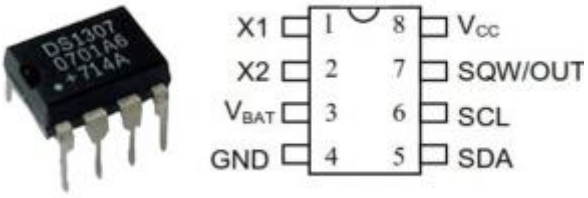


Figure 23

The program uses the getDate(buffer); function - the one that's in the "ds1307.h" library. It reads the current value of the first seven internal registers of the DS1307. They contain all the information on the time and the date and they store it in "buffer[0]" – "buffer[6]".

To be precise, "buffer[2]", "buffer[1]" and "buffer[0]" contain the hours, minutes and seconds respectively; they then displays them on the LCD screen.

When you record the program, you'll see the time that doesn't necessarily correspond to the real one (e.g. 00:00:00). Remember we still haven't adjusted the real-time clock and calendar or set it to the right time. Try turning off the main power supply. After a few minutes have gone by connect it up again and you'll see that the clock has kept working. This is because of the external battery. If you feel like it, do the experiment again but this time disconnect the battery from the circuit.

14. EXAMPLE 8: CLOCK AND CALENDAR PART 1

This exercise is just like a continuation of the previous one: the date and the time are displayed on the LCD screen. Two new functions have been created to make this possible; you'll be able to use them in future projects too: visuDate() and visuTime().

visuDate() displays the current day and date on the first line of the LCD screen using the "ddd MMM dd yyyy" format where ddd are the days of the week from Sunday (SUN) to Saturday (SAT), MMM are the names of the months from January (JAN) to December (DEC), dd is the date from 1 to 31 and yyy is the year from 2000 to 2099.

visuTime() displays the current time on the second line of the LCD screen using the "hh:mm:ss" format where hh represents the hours from 00 to 23, mm the minutes from 00 to 59 and ss the seconds from 00 to 59.

The getDate() function obtains the data from the first seven internal registers of the DS1307 and stores it in "buffer[0] … buffer[6]" in the Arduino RAM memory. It corresponds to the time and date of the clock and calendar.

Have a look at how the program encodes the number of the day of the week from 1 to 7 and stores it in buffer[3] using the following abbreviations: SUN, MON, TUE, WED, THU, FRI and SAT. It encodes the number of the month from 1 to 12 in the same way and stores it in buffer[5]: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV and DEC.

Two arrays of character chains or “strings” have been created especially for this purpose: dayWeek[7] [4] and nameMonth[12] [4]. The first one comprises seven four-character positions:

Array dayWeek[7] [4]							
Position	**0**	**1**	**2**	**3**	**4**	**5**	**6**
string	Sun	Mon	Tue	Wed	Thu	Fri	Sat

The second array comprises twelve four-characters positions:

Array nameMonth[12] [4]												
Position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
string	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC

When **getDate()** is executed, the buffer[3] loads a number between 1 and 7 that corresponds to a day of the week. It subtracts 1 from this number so the result lies between 1 and 6. It’s used to look for the corresponding chain in the **dayWeek[]** array and display it on the LCD screen:

```
lcd.print(dayWeek [buffer[3]-1]);                    //Displays the day of the week
```

For example, if buffer[3] = 2, the chain displayed on the LCD screen will be: “MON”.

A similar procedure is used to display the name of the month. Have a look at it.

Upload the program and try it out. Remember that you still haven’t set the clock or adjusted the calendar. So it doesn’t really matter what time or date it shows.

15. EXAMPLE 9: CLOCK AND CALENDAR PART 2

At last we're there. This example uses the previous visuDate() and visuTime() functions to run a real-time clock and calendar; you can also adjust any of the following fields: day of the week, day of the month, month, year, hours and minutes.

That's what the "Sel" pushbutton does; it's connected to the D3 input pin and selects the field requiring adjustment. Another button, "Adj", adjusts the value in the selected field.

The circuit diagram appear below (Figure 24). Take a close look at them:

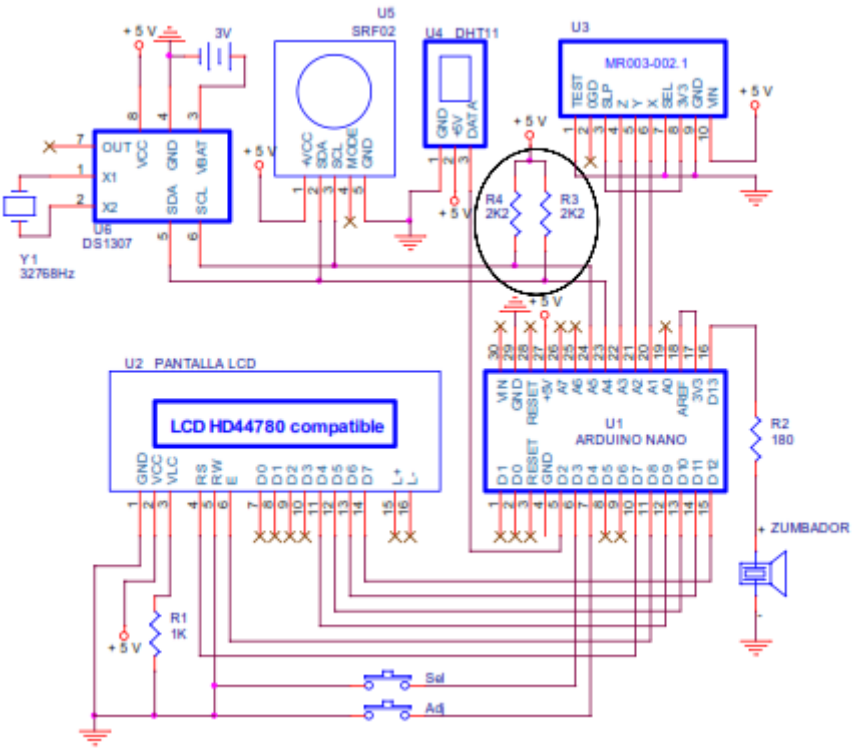


Figure 24

This exercise may look quite long but it's actually really straightforward. Remember that what we try to do is present programs that are simple and easy to understand; this may sometimes mean that some of them don't work quite as well as they could and may be improved in lots of different ways.

All this program does is read the contents of the DS1307 clock and calendar and display the current date and time on the LCD screen until someone presses the "Sel" pushbutton connected to the D3 pin.

When you press the "Sel" pushbutton the program goes into a sequence to adjust the following fields: the day of the week, the day of the month, the month, the year, the hours and the minutes. It starts off with the day of the week. Have a close look at the following sequence of instructions; it's quite similar to the ones used to adjust any one of the other fields.

```
//Adjust day of the week from 1 to 7 (Sun-Sat)
while(digitalRead(sel))          //As long as sel remains disenabled...
{
lcd.setCursor(1,0);              //Positions the cursor in the first field to be adjusted
lcd.blink();                     //Displays the cursor
if(!digitalRead(adj))            //If the adjustment (D4) is enabled...
{
delay(500);                      //Pauses 0.5"
buffer[3]++;                     //Changes the day of the week
if(buffer[3] > 7)                //If it's greater than 7 ...
buffer[3] = 1;                   //Starting from 1 (Sun)
DS1307.setDow(buffer[3]);        //Adjusts the day of the week in the DS1307
visuFecha();                     //Displays the changes on the LCD screen
}
}
```

As long as the "Sel" pushbutton continues to remain disenabled, the cursor is positioned appropriately and displayed. If the program detects that the "Adj" pushbutton connected to D4 is enabled, the day of the week in "buffer[3]", goes up by one unit; it may vary from 1 to 7. The new value is adjusted in the DS1307 using the **DS1307.setDow(buffer[3]);** function and is displayed with **visuDate();.** Every five seconds the program checks the status of the "*Adj*" pushbutton.

Adjustments to fields finish each time the "*Sel*" pushbutton is pressed. The program then goes on to adjust the next field; the sequence of instructions is very similar. The minutes is the final field and when the program's finished with it the adjustment sequence terminates. The cursor disappears and the cycle starts again.

Now you know how to set your clock and calendar to the right time and date. Load the program and give it a go.

16. EXAMPLE 10: BILLBOARD

Now let's make things even more interesting. You're going to get into the DS1307 real-time clock and calendar to get the date and the time out by using the I2C protocol. You're also going to communicate with the DHT11 sensor from the previous unit to the get the room temperature and the relative humidity by using the 1-wire protocol.

Or to put it another way, you're going to display the date and the time and then the temperature and the relative humidity alternately on the LCD screen. You're going to create the sort of billboard you've probably seen in the streets and avenues of the town where you live.

The example displays the date and the time for five seconds and then the temperature and the relative humidity for another five seconds. Load it and give it a go.

17. EXAMPLE 11: DATA LOGGER

This example goes along the lines of the previous one; it may well help you to create a data recording system or “data logger”. This is the last program in unit 7 and it sends a record of the date, time, humidity and temperature at regular intervals using serial communication.

Imagine that a PC receives the data and then processes it: it stores it, displays it, prints it out, creates records etc.

You use the **t** constant to indicate the gap in tenths of a second between each record; the example uses 50 (5 seconds). When the program is loaded it opens the serial monitor which receives the data records each time they come in.

REFERENCES

BOOKS

[1]. **EXPLORING ARDUINO**, Jeremy Blum, Ed.Willey
[2]. Practical ARDUINO, Jonathan Oxer & Hugh Blemings, Ed.Technology in action
[3]. Programing Arduino, Next Steps, Simon Monk
[4]. Sensores y actuadores, Aplicaciones con ARDUINO, Leonel G.Corona Ramirez, Griselda S. Abarca Jiménez, Jesús Mares Carreño, Ed.Patria
[5]. ARDUINO: Curso práctico de formación, Oscar Torrente Artero, Ed.RC libros
[6]. 30 ARDUINO PROJECTS FOR THE EVIL GENIUS, Simon Monk, Ed. TAB
[7]. Beginning C for ARDUINO, Jack Purdum, Ph.D., Ed.Technology in action
[8]. ARDUINO programing notebook, Brian W.Evans

WEB SITES

[1]. https://www.arduino.cc/
[2]. https://www.prometec.net/
[3]. http://blog.bricogeek.com/
[4]. https://aprendiendoarduino.wordpress.com/
[5]. https://www.sparkfun.com