SCIENTIFIC ARDUINO PROGRAMMING

Arduino programming for scientists

A free addendum to "Scientific Programming"

GIOVANNI ORGANTINI

Sapienza Università di Roma & INFN–Sez. di Roma

March 4, 2016

Contents

SCIENTIFIC ARDUINO PROGRAMMING – VERS. MARCH 4, 2016

This work was partly supported by Farnell Element14 who kindly provided some of the parts described in the text.

Contacts:

Prof. Giovanni Organtini
Sapienza Università di Roma
Dip.to di Fisica
P.le Aldo Moro, 2
00185 ROMA (Italy)

Tel: +39 06 4991 4329 Fax: +39 06 4453 829
e–mail: giovanni.organtini@uniroma1.it

Technical note

This book has been written using LaTeX, an open source, high–quality typesetting application. Schematics have been realised using FRITZING, an open source application to draw electronics schemas.

Arduino itself is an open source initiative. We strongly support this kind of projects, not only because their open nature makes them inexpensive, but mainly because the open standard guarantee a very high quality of the tools (they can be contributed by thousands of developers) and mostly because they provide a formidable tool for learning.

We invite you too to support the open initiatives you use: you can do that in a variety of ways. You can, for example, donate money to the developers, to recognise their effort, or you may prefer to buy products from those who make them available to anyone as open source projects. That is the case of Arduino, for example. You can certainly find Arduino clones that are cheaper than those you can get from the Arduino official store, or from other official stores of legal Arduino–compatible boards, but saving few dollars does not help much, in fact. On the other hand you create a damage to those who are trying to change the standard business model based on hiding information and patents. As you can read on the Arduino website, you can download and use their reference designs and "you are free to use and adapt [those designs] for your own needs without asking permission or paying a fee". We believe this is really a revolutionary paradigm with respect to the commonly adopted one, that is suitable to increase dramatically both knowledge and welfare all around the world.

Chapter zero
Introduction

This paper is an introduction to Arduino programming for students who learned C on "Scientific Programming" by L.M. Barone, E. Marinari, G. Organtini and F. Ricci–Tersenghi [1], edited by World Scientific (or its italian counterpart "Programmazione Scientifica" edited by Pearson). "Scientific Programming" is an innovative textbook on computer programming thought for science students, who does not care about writing `Hello World!` on a screen, but are interested in using a computer as a tool to do science.

Chapter 0 of "Scientific Programming" is about how computing and programming is important for a scientist. Chapter 1 is about information representation (there is no need for a Chapter uno here, since information is represented exactly in the same way on Arduino boards).

One of the authors of "Scientific Programming" (Giovanni Organtini), started to teach Arduino programming in his course about computing and programming for physicists, being the Arduino board a great resource for physicists and engineers. As well as in the case of C++ he wrote these notes in the style of [1], keeping the same chapter structure and a very similar style. In this booklet, chapters have the same numbering of [1], except the numbering is given in italian, as a tribute to the italian origin of the Arduino board, invented by Massimo Banzi and his colleagues in Ivrea (TO), Italy[1]. Each chapter uses notions that can be learned in the corresponding chapter in [1]. In fact, Arduino can be programmed in C++ language, but at a very basic level it can be thought as a dialect of the C language, sharing mostly the same syntax. You can learn some basics of C++ reading another free addendum to "Scientific Programming", by the same author, available on the Scientific Programming website. However, you can obtain good results just knowing some C language: the chosen language for "Scientific Programming". For these reasons, some chapter number is missing: the corresponding content is exactly the same of [1].

Students wishing to learn some Arduino programming without owning a copy of [1] shall not be scared: those students should not have too much difficulties in learning how to write programs for Arduino if they are able to write C programs.

[1] The name Arduino comes from their haunt in Ivrea: a pub called after the name of an ancient king of Ivrea whose name was Arduino.

zero.1 What is Arduino?

Arduino is an inexpensive, commercially available electronic board with a microcontroller and some I/O capabilities. It exists in various versions, that share the same, simple programming language. The huge success of Arduino, with respect to other microcontroller boards, was due to the fact that both hardware and software were released as Open Source projects: you can read, study and even expand its capabilities both in terms of software as well as in terms of hardware. All the information are shared under the Creative Commons Attribution-ShareAlike 3.0 License.

You can use Arduino for many different purposes: from teaching to home automation[1], from scientific purposes to commercially available devices, as well as to have fun (you can be surprised about the many ways in which people use Arduino). Thanks to its very simple interface to I/O ports you can control many different devices, both digital and analogical. For example, you can measure voltages using analog inputs or drive a DC motor using a digital output port. You can as well switch on and off an LED or a relay using digital output ports and transmit/receive data to/from more complex devices such as GSM boards. The job of (at least a large part of) physicists is to measure something: Arduino is then a very useful tool both to control measuring apparata or as a device to take measurements by itself (for many purposes it can be accurate enough to replace professional, and expensive, instruments).

The design of Arduino boards is such that its form factor is (almost) independent on the Arduino version. The first Arduino boards used a microcontroller whose chip took a somewhat large space; nowadays the same chip is available in a much smaller form factor, however the size and the shape of the Arduino board is still the same (and in fact there is plenty of free space on it). That choice has one big advantage: third party manufacturers can easily design, produce and sell boards that extend the functionalities of any Arduino, and users can easily connect them to it. In fact, those board, called **shields** have a set of pins that just plug into the corresponding pins on the Arduino board and no specific electrical connection is needed to make them work. You can buy, for few bucks, boards designed to provide Internet or GSM connectivity, GPS capabilities, stepper motor control and much more.

Using Arduino, anyone with a very basic knowledge of some elementary electronics is able to build complex electronic devices effortlessly: the complexities of the electronics are translated into software, hence even people not used to work with analog and digital devices such as diodes, transistors, operational amplifiers, integrated circuits, logic ports, etc., can realise interesting projects. Tutorials can easily be found on the Internet for various tasks: given the Open Source nature of the project, people are encouraged to share their projects with others so anyone can benefit of other's experience.

[1] The author, for example, built a device able to switch on and off home appliances remotely just placing a phone call.

zero.2 What this booklet is intended for

Scientific Programming aims to fast, precise, efficient and complex computation. All of these characteristics are addressed in our previous publications [1] [2].

This booklet is intended as an introduction to Arduino programming from the point of view of the microcontroller programming. It does not include details about hardware if not strictly needed. It is not even a collection of Arduino projects: those provided are just meant as examples.

zero.3 How to use this booklet

As stated in the introduction of the chapter, this booklet cannot be considered complete and consistent by itself. It must be used at least in conjunction with another book teaching scientific programming in C. The best, of course, is [1], since this booklet is modeled upon it and is using the same chapter structure.

The way in which this booklet can be used depends wether or not you already knows C or not. If you don't, study the C language using [1]. As soon as you study a given chapter, look for the corresponding chapter in this publication, read it and make the proposed exercises.

If you already know C, proceeds with this book, after refreshing your mind having a look to the corresponding chapters on [1], to recall the numerical techniques or language details involved.

Please also consider that this is an experiment. Any comment concerning the content of the present publication will be greatly appreciated. You can communicate with us via e–mail writing to giovanni.organtini@uniroma1.it.

zero.4 Supporting this work

You can support the development of this manual by making a donation on PayPal using the e–mail address giovanni.organtini@uniroma1.it (but not if you are a student of mine, at least until you have successfully passed your exams).

In particular, details about projects specific for physicists will be added in future, upon reception of a reasonable amount of donations. We are, in fact, using donations, to buy new devices to be tested. Once tested, programming details will be added to this book.

Donations are intended to support the development of this publication, then we consider donations to be fair in the range 1–5 euros (5 euros already being a big donation), not more.

If you are a teacher you can be interested in using Arduino in your school to make physics experiments in lab. Designing scientific experiments to be done with Arduino is the main topic of this manual. Donations will make more and more affordable experiments

to appear on this publication. Your pupils may want to learn about Arduino anyway. You can support this work collecting donations among your students and making a unique payment. Consider the possibility to print and *sell* the printed manual to your student as a tool for raising some funding for your lab: you can do that, according to the license, provided that you are a non–commercial entity.

All donors will be included in a mailing list and notified as soon as a new, improved version of the manual is made public. Of course, there is no need to donate more than once: if you were included in the mailing list you stay there forever, unless you ask to be removed.

If you make a donation, we only collect your e–mail address. The only purpose is to notify you when a new version of this document will be made public. We will not share any information about you with others. You are free to ask to be removed from our list at any time, just by sending us an e–mail at giovanni.organtini@uniroma1.it.

Chapter due

How Arduino works

Chapter 2 of "Scientific Programming" is about computer architecture and programming languages. In this chapter we provide a brief description of the Arduino board, from the point of view of its architecture as a computer. You need to understand how a computer works and what is intended for terms like machine language, high–level programming language, compiler, etc.. Arduino is in fact a computer much like the one described on Chapter 2 of [1]. Before reading this chapter, it is advisable to review the content of that chapter of [1], if you can.

Whenever we mention an Arduino board, we always refer to the Arduino UNO: the most widely used version of it. There are other versions of Arduino boards that differ for the size of the memory, the microcontroller used on board, the number and the type of the I/O ports and other capabilities, or the form factor. However, all of them share the same first principles.

due.1 Arduino basic architecture

The core of the Arduino board is a microcontroller chip known as the ATmega328. The Atmega328 is in fact an 8 bit computer just as the one described in the first chapters of [1]: once switched on, its CPU loads a byte from a predefined memory location and interpret it as a statement. What follows is interpreted according to the content of such a byte. Contrary to the computers to which you are familiar with, the ATmega328 does not run any **operating system**: the usage of the resources is completely under the control of the programmer. You cannot rely on the operating system to prevent wrong memory usage, overflows, underflows and any other error. Moreover, that CPU can only run one task at the same time (you may remember that this is true for all the CPU's, however the operating system distribute the operation time to various tasks in such a way you have the impression that many programs run at the same time on your computer).

A fresh Arduino has an empty memory, hence the first byte loaded by the CPU correspond to the statement `nop`: **no operation**. Before using Arduino you must then load its memory with an **executable** program, i.e. a sequence of bits, the first of which

is interpreted as a statement and executed. If the statement needs parameters to be executed, they are taken from the following bytes in the memory. Once the execution of a statement has been completed, the CPU loads the successive byte in the memory and interprets it as well as a statement. If you switch off your Arduino, the memory is not lost. The sequence of bytes loaded into it are kept in a non–volatile memory, such that when you switch it on again, the program starts again from its beginning.

The timing for CPU operation is provided by a 16 MHz clock, while the power can be provided through a dedicated power jack as well as through its USB interface (see below). For operation, Arduino requires an input voltage between 7 and 12 V (voltage regulation is provided on board, so you just need an inexpensive power supply for that). On board, both a 5 V and a 3.3 V regulated output are provided to the user, too, from which you can drive a maximum current of 50 mA.

Arduino memory is of three types: a **flash memory**, where the program is stored, of 32 kB; a static random access memory (SRAM) of 2 kB, where the CPU stores and manipulates the variables used in the program; and an erasable read only memory (EEPROM) of 1 kB where the programmer can store data that must survive the switch off (as the flash memory, where the program is stored). Compared to modern computers, who often bear as much as few GB, a total of 35 kB seems ridiculous, but in fact it is enough for most purposes. Because of the lack of an operating system, the memory usage is under your own responsibility: if you run out of memory or you try to access a non existing memory location, your program may behave strangely and it is very difficult to debug it. You must always keep the number of variables under control in your program.

The ATmega328 CPU is connected to 14 I/O digital pins (numbered from 0 to 13), 6 analog inputs and a USB port. A digital pin is an electrical connection that can have two logical states: 1 and 0, or true and false or, as in the Arduino jargon, `LOW` and `HIGH`. When put to `LOW`, the corresponding pin is at the ground potential: if you measure the voltage between the ground and the pin you get zero. If the pin is set to `HIGH` the voltage between the pin and the ground is 5 V.

Pins 0 and 1 are used for serial transmission and reception: on those line the Arduino board can communicate with shields on it using a serial protocol. Serial protocols are communication protocols in which each bit is transmitted/received one after the other. Pins 2 and 3 can also be used as **interrupts**. An interrupt is an electrical signal that interrupts the current CPU program upon the occurrence of a given event. Interrupts exists also on computer's CPU's[1]. Once an interrupt is detected, the CPU saves its state in the memory and abandon the execution of the program, jumping to execute the **interrupt handler**: a short and fast piece of software needed to serve the interrupt. Upon completion, the CPU resumes the status it had before serving the interrupt and restart the execution of the program.

Pins 3, 5, 6, 9, 10 and 11 are PWM (Pulse Width Modulation) pins (labelled with a

[1]For example, on computers running Windows, pressing the `Ctrl-Alt-Del` key combination, generates an interrupt (a software interrupt) and all tasks are suspended until the interrupt handler (the task manager) is finished.

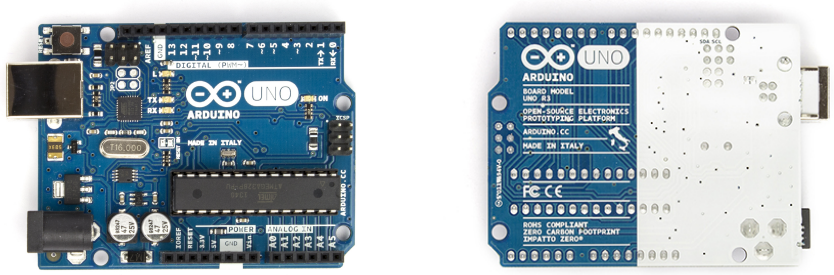

Figure due.1 Arduino UNO as it appears on the front and back side of the board. Note the map of Italy on the back side.

~) and provide also some analog capabilities (see Chapter sette). Correspondingly, their memory counterparts contains values between 0 and 255.

Pin 13 is also connected to an LED on board. When the pin is `LOW` the LED is off, while if the pin is `HIGH` the LED is on.

Besides standard use as digital I/O ports, pins 10, 11, 12 and 13 provide a mean to communicate with external peripherals.

Analog inputs are labelled `A0` through `A5`: each of them provides a resolution of 10 bits, i.e. they convert any voltage from 0 to 5 V to a number between 0 and 1023 that can be accessed in the memory.

All the systems are mounted on a board whose size is 60.6×53.4 mm^2 and weights as low as 25 g (Fig. due.1).

The board also carry a USB A/B connector through which you can connect it to a computer for communications. The USB connection also provides power to Arduino when connected to a computer, so that you don't need an external power supply.

due.2 Program development

A program for an Arduino is, as any other program for a CPU, a sequence of bits in **machine language**. In order to make the life of a programmer easier, the Arduino team has provided a high–level programming language, a compiler and a communication tool to deploy the machine code on the Arduino memory.

All those tools are included in an IDE (Integrated Development Environment) freely available on the Arduino website for download: identify the version needed for your preferred operating system and install it. It appears, as many computer applications, as a tool with a menu and some windows. One of those windows is used to edit the program, called a **sketch** in Arduino jargon. Sketches are written in C++, but you can think them as C programs, since the basic syntax is exactly the same.

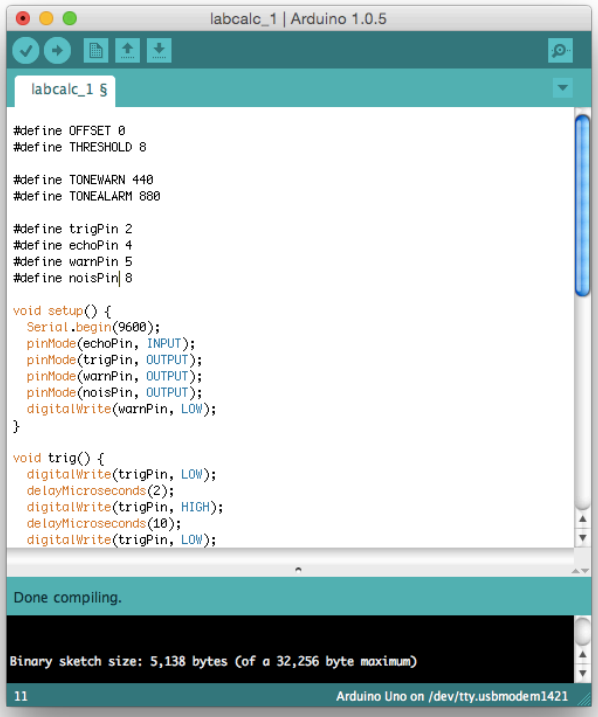

Figure due.2 The Arduino IDE appears as a window in which you can type the text of the program, called **sketch**.

You can compile your sketch within the Arduino IDE (Fig. due.2) clicking on the **verify** button at the top left corner of the window: the compilation process translates each C++ statement in the sketch into one or more machine language statements for the Atmega328 processor. Once compiled, the executable sketch can be transferred into the Arduino memory through the USB cable, upon clicking on the **upload** button . You may need to select the appropriate port from the menu, if there are more than one available. A transfer always trigger the invocation of the compiler, first. The sketch execution starts as soon as the transfer finish.

More capabilities are added to the basic language by means of external libraries provided by the Arduino team or third parties. Libraries can be included in the executable code, acting on the appropriate menu item. If needed, adding a library, automatically

adds lines on the sketch to inform the compiler about the syntax of the new statements provided by the library.

due.3 Using Arduino on Linux

Few versions of Linux (such as Ubuntu) are designed for not very experienced end users. Other versions, such as Scientific Linux, however, are not: they are supposed to be used by people expert enough to understand some internals of the operating system. If you are reading this book, you probably are a scientist (or at least you aim to become a scientist) and most probably you are using a version of Linux not so *user friendly*[1]. In this case you may miss some needed component.

In order to use the Arduino IDE on Linux, you need to make sure you have the **Java Development Kit** or **JDK**. Only after having it installed you can run the `arduino` script included in the bundle. Search for `jdk` or `openjdk` in the packages available for your distribution. For example, in systems using `yum` for package management:

```
yum search yum
```

You will get something like:

```
[organtin@pc1 ~]$ yum search jdk
Loaded plugins: refresh-packagekit, security
================================================ N/S Matched: jdk =
java-1.6.0-openjdk.x86_64 : OpenJDK Runtime Environment
java-1.6.0-openjdk-demo.x86_64 : OpenJDK Demos
java-1.6.0-openjdk-devel.x86_64 : OpenJDK Development Environment
java-1.6.0-openjdk-javadoc.x86_64 : OpenJDK API Documentation
...
```

Then, to install:

```
yum install java-1.6.0-openjdk
```

When you run the IDE, it may happen that many error messages appear on the terminal, complaining you have no permission to create *locks*. That's why normally Linux prevent standard users to write on places where the system stores important information for its working. You can either run the IDE as `root`, or better you can, as `root`, give yourself the rights to write in the proper place. The Arduino IDE needs write permission on `/var/lock`. To give them you can use the system command `chmod` in a shell running in a terminal:

```
chmod o+rwx /var/lock
```

[1] In *italics*, since there is nothing friendly in hiding details to users: this is what is usually intended for *user friendly*.

It means *change file mode* for file `/var/lock` such that others (not in the group of the super user) (`o`) should be given (`+`) permissions to read (`r`), write (`w`) and execute (`x`) that directory (executing a directory makes it possible to `cd` to it).

On most Linux distributions USB devices are created as soon as you plug the cable into a USB slot. Connecting an Arduino to your computer using the USB cable may then result in the creation of a device called `/dev/ttyACM0` or so, or `/dev/ttyUSB0` or so. These devices are needed for the communication between Arduino and the computer and must be properly set. In the IDE menu you can see the available **ports** in the `Tools/Port` menu item. The port in use must have the right permissions, too. If you can't connect to any port, you may alter permissions (still as `root`) with

```
chmod o+rw /dev/ttyACM0
```

There is no need to set the `x` permission in this case. If the device is deleted when you disconnect your Arduino, each time you reconnect it you must set permissions again. A possible workaround, in this case, is to include the user under which you run the Arduino IDE, in the same **group** of the device (usually `dialout`). You can check what are the owner and the group to which the device belongs using the Linux terminal with the command

```
ls -l /dev/ttyACM0
```

You should see something like

```
crw-rw----  1 root  dialout   31,   2 Jan  5 14:10 /dev/ttyACM0
```

that means that `/dev/ttyACM0` is a **character device** (`c`) belonging to user `root` of the `dialout` group. The owner and any other user in the same group have read and write permissions. Others have no rights (the last group of three dashes: `---`). If you want to be able to read and write from/to this device, you must belong to the `dialout` group. You can add a user `username` to another group using the system command

```
usermod -a -G dialout username
```

This way, each time you plug your Arduino to your computer, the device is created on the fly and you can seamlessly use it, belonging to the group authorised to read/write from/to it.

Chapter tre

Arduino basic programming

In Chapter 3 of "Scientific Programming" we teach how to write a very basic program able to make something interesting for a scientist. In this chapter we illustrate how to write a very basic sketch for Arduino using its IDE. In this chapter no usage of I/O ports is made, but for the USB connection. The syntax of the Arduino language is exactly the same of the C–language syntax, hence we are not going to discuss it. We concentrate on the particular aspects of the Arduino language, in which you can use all the concepts you learnt about C programming: variables, operators, statements, types, constants, etc..

tre.1 The first Arduino sketch

The Arduino language does not need a **starting point** as in C–language, where you are forced to define the `main` program. Indeed, when the program starts, it loads what is called an **object** in **Object Oriented Programming** OOP [2] into the memory. Objects in OOP belong to **classes**. For each object of the same class a **state** is defined as a set of **attributes** or **members** that can be thought as variables, being represented as a collection of data of various types in the memory. The state of an object can be manipulated by **methods**: a collection of statements intended to perform a given operation to alter or provide the state of the object. If you are not familiar with OOP, you can think to the state as a collection of variables and to the methods as a collection of functions.

Here we assume you are familiar with the C language learnt on [1], so we adopt a slightly incorrect language to describe how a sketch is organised, that however is consistent with the notions given in the corresponding chapter on [1] and works quite well, even if not formally correct.

An Arduino sketch is self contained in one file in which, contrary to the C–language, you need to define at least two *blocks* of statements: one called `setup()` and another called `loop()`. Variables to be shared between the two blocks must be defined outside them, as if they were **global** variables[1].

[1] In fact they are not: they are the members of the object loaded into the memory at start. As such,

As soon as the program starts, the statements collected within the `setup()` block are executed: they are intended to initialise the content of the variables at start as well as to configure the Arduino ports behaviour. Once the execution of the `setup()` block is finished, Arduino starts executing statements in the `loop()` block. After execution those statements are executed again forever (hence the name **loop**).

```
1 void setup() {
2     ...
3 }
4
5 void loop() {
6     ...
7 }
```

Listing tre.1 An Arduino basic sketch.

Both the `setup()` and `loop()` blocks are defined as `void` blocks, i.e. they do not return anything (see Listing tre.1). You can use standard preprocessor directives such as `#define`, `#ifdef`, `#ifndef`, `#endif`, etc. In particular, we strongly encourage you to define constants as preprocessor symbols (not as variables, since they eat the SRAM memory).

tre.2 I/O with Arduino

Arduino does not have a port to connect to a screen, nor to a keyboard: they are not needed on this type of devices. I/O pins illustrated in Chapter due are meant to provide some input and output capabilities to some electronic devices: in practice you can only read and write voltages to that ports. There is no I/O like the one you are used to in computer programming, however in some cases it is useful to read some message on screen (e.g. during the debugging phase).

In order to make Arduino display some text message or to provide some input from the keyboard you can connect the board to a computer via the USB cable. Messages are displayed on a dedicated window called the **serial monitor**. Such an interface is not intended for complex functions, and has a very basic behaviour, even more basic than terminals. You can activate a serial monitor by selecting the corresponding item in the main menu of the IDE. At start, the serial monitor may behave strangely, showing (apparently) random characters. Those characters are those remained in the serial buffer flushed to the monitor when connected.

Text I/O capabilities are provided through the serial communication using the USB cable, hence there is no `printf` statement in the language. Instead, there is an object called `Serial` providing methods to read and write to the serial line. You can think to these methods as statements in C language, whose name contains a dot.

they are not at all *global*, but **encapsulated** within the object.

If you want to print a text message on the serial monitor window, you can use the `Serial.print()` statement, whose syntax is

```
Serial.print(<message>};
```

where `<message>` is a variable or a constant. The way in which the content of the message is displayed depends on its type. For example, writing

```
int i = 67;
Serial.print("the value of i is ");
Serial.print(i);
Serial.print("\n");
```

makes the text `the value of i is 67` appear on the serial monitor window. The first `Serial.print` statement[1] contains a string constant as a parameter (specified by the characters `"` surrounding the text) and it is written as such. The second one contains an integer variables, whose content is read from memory and represented as a standard integer number on the screen. The last statement adds a newline character (note that it can be written even as `Serial.print('\n')`, being `\n` a single character) after `67`. You can print a text message ending with a newline also using the `Serial.println` statement that automatically adds a newline character at the end of the message as in

```
int i = 67;
Serial.print("the value of i is ");
Serial.println(i);
```

In order to configure the speed of the serial communication, you need to setup the communication parameter before starting using the channel. To this purpose, use the `Serial.begin(9600);` statement, where 9 600 is the communication speed in **bauds** (a unit of speed in telecommunications). Such a speed can be any number among a range documented on the Arduino website, depending on your hardware. Usually 9 600 works well with any relatively modern computer.

Laboratory tre.1 *I/O with Arduino*

Write a sketch with few variables of various type and write its value on the serial monitor using the various forms of the `Serial.x` statements. Try putting the statements within the `setup()` block and within the `loop()` block and observe what happens in the two cases. Note that `Serial.print` tries to get the type of variable to represent it in the right way. For example, if `ch` is a character variable, try to print `c` and `c+1`: the results are different. Why? How would you write the character that comes after a given one in the ASCII table?

[1] Remember we are adopting an incorrect language here: `Serial.print` is not a statement, but a method of the `Serial` object.

The `Serial.read()` statement returns the first byte available in the input buffer. Its usage is quite more complex with respect to `scanf` and is not described here (you must be familiar with the selection structure, first). On the other hand, its usage is not so frequent in Arduino programming and is not so important as on computers.

tre.3 Showing data

Despite Arduino does not have a screen, you can connect some display to it. There are few options for this: either you use an LCD display, or a TFT shield, i.e. a shield with a Thin Film Transistor LCD. Plain LCD displays can show up to four lines of text or so, while with a TFT shield you can display data with high resolution (typical resolutions are 160×128 pixels and 240×320 pixels). They exist both in black and white and in colour versions and few models have also touch features.

TFT and LCD programming is not discussed in this version of this document: they will be available when we will reach the amount of donations needed to buy the corresponding hardware. See Chapter zero about how to donate.

Chapter quattro

Program execution control

Chapter 4 of "Scientific Programming" is devoted to the illustration of the structures used to control the program execution flow (logic management). Logic management on Arduino is exactly the same as for C. There is no exception with respect to the C language structures. This chapter, then, is devoted to applications specific to Arduino.

quattro.1 The selection structure

The selection structure is useful to test whether there are characters waiting to be read in a serial communication or not. Before using the I/O statements on the serial monitor you must be sure that the connection is up and running. Remember that there is no operating system on Arduino taking care of the resources usage, hence you can successfully compile and run a sketch trying to use serial communication, but it can behave in an unpredictable way if the communication channel does not exist and is configured properly.

The statement `Serial.available()` returns `true` if there are characters waiting to be read on the serial line, `false` otherwise. In order to test if characters are ready to be read from the line you can use something like

```
Serial.begin(9600);
...
if (Serial.available()) {
   char c = Serial.read();
   Serial.print(c);
}
...
```

With `Serial.read()` you can only read one character: there is no equivalent of `scanf(str, "%s")` statement in the Arduino language.

quattro.2 The iteration structure

The iteration structure, too, can be very useful in Arduino programming. Besides the usages to which you are familiar with, a quite frequent usage of the iteration structure can be found in Arduino sketches and, in particular, in the `loop()` block. As stated in Chapter tre, the statements in the `loop()` block are repeatedly called in an infinite loop. If you need the sketch to stop completely its execution, you need to setup an infinite empty loop:

```
while (1) {
  // do nothing
}
```

Another usage is to wait for characters on the serial line and build strings from those characters, like in

```
char ch = NULL;
char str[255] = {0};
int i = 0;
while (ch != '\n') {
  if (Serial.available()) {
    ch = Serial.read();
    str[i++] = ch;
  }
}
```

In the code above, we define an array of characters whose elements are set to `NULL`. Then, we start reading characters from the serial line, as soon as they become available (`Serial.available()`). Once read using `Serial.read()`, characters are added to the string `str`, assigning the corresponding element in the array. The index of the next element (`i`) is updated and the loop is continued until the last read character is a newline.

Laboratory quattro.1 *A better control of the serial monitor*

Modify the sketch you wrote in Chapter tre to include control of serial communication availability. Each time you deploy a new sketch to Arduino, the serial monitor is closed, being the communication channel used to transmit the executable code to the Arduino flash memory. Before writing to the serial channel, check that someone is connected to it. To do so, you can write a loop in which you control the availability of characters to be read on the serial line: leave the loop, if so. Opening the serial monitor, the sketch should be blocked, waiting for characters to appear on the serial line. Send at least one character to it (just pressing the enter button is enough), then either discard the character or show it, and exit from the loop, continuing the execution of the sketch.

Laboratory quattro.2 *Controlling the execution of the* `loop()`

Define a variable whose value is set to zero at the beginning. The variable must be incremented at each execution stage (every time the `loop()` function is called. Show the value of the variable in the serial monitor. Use a `for` loop to change the value of another variable whose value is set to zero at the beginning of each `loop()` invocation and is incremented at each step for a number at your choice. Write both the number of times the `loop()` function is called and the value of that variable.
Use an iteration structure to change the way in which the `loop()` block is executed. In particular try avoiding it to be called more than one, two or ten times.

Chapter sei

Saving data

Pointers are illustrated in Chapter 6 of "Scientific Programming". Besides other uses, pointers are used to make data persistent in files. If you are using Arduino to take data in a physics lab or for any other application requiring data logging, you need some persistent storage, i.e. a way to, at least, write data into a file to be retrieved offline for analysis.

There are several ways to achieve the task: for example, one can transmit these data via Internet to some server who takes care of getting the data over the communication protocol and store them on disk; if you can write a program working as a client over a serial communication channel, you can send data over this channel using the `Serial.xxx` family of statements described in Chapters tre and quattro; another way consists of writing data directly from Arduino to some external storage attached to it.

sei.1 Using Serial communications

Arduino uses serial communication to exchange data with a computer both when deploying the executable program into its memory, as well as when using the **serial monitor** tool. Of course, one can exploit this ability to write a program on a computer that sends/reads characters over the serial line to exchange data with Arduino.

In principle serial communication is very easy: on the Arduino side just use the `Serial.xxx` functions, while the computer sees the serial channel as a file from which you can read characters using binary read/write functions (see Chapter 6 of [1]). However, using a serial line requires some initial configuration that may sound difficult to beginners. Despite this textbook is focused on Arduino programming, we include here few information about how to write a program on a computer to be used in conjunction with Arduino to establish a serial connection.

To fully understand the program, one need to know about bitwise operators, illustrated in Chapter 14 of "Scientific Programming" and about `struct`, the topic of Chapter 9. However, some *copy and paste programming* is acceptable even for a scientist, if at least the first principles are understood.

A serial communication implies to connect to the serial port: this is done using the `open` statement in C as in

```
int fd = open(serialport, O_RDWR | O_NONBLOCK );
```

where `serial port` is a string containing the filename corresponding to the port (on Linux, it is something like `/dev/ttyACM0` or `/dev/ttyUSB0`, while on MAC OS X is something like `/dev/tty.usbmodem1421`). The `O_RDWR` and `O_NONBLOCK` constants are defined in `fcntl.h`, as `open`. The `|` character is a bitwise OR operator: it essentially perform the sum of the two values represented by the constants. Browse the `fcntl.h` file on your system to get their values. On our system, we found

```
#define O_RDWR      0x0002
#define O_NONBLOCK  0x0004
```

The `0x` characters preceding the values indicate that the values are expressed as hexadecimal numbers (being bot less than 16 they coincide with decimal ones). The result of `O_RDWR | O_NONBLOCK` operation is 6 (`0x0006` in hexadecimal), since the bitwise OR operator returns the sum of the operands expressed in binary, hence `0010 + 0100 = 0110`.

`O_RDWR` is needed to open the file (the port) in both read and write mode, while O_NONBLOCK tells the system that the open statement should return after connection without waiting for data becoming available on the port.

Serial ports can be configured to run at different speed (e.g. 9 600 baud). The speed is contained in a `struct` and can be set for input and output as

```
struct termios toptions;
...
cfsetispeed(&toptions, 9600);
cfsetospeed(&toptions, 9600);
```

where `options` is a `struct` called `termios` (the operator `&` returns the address of the `struct`) and 9 600 is the chosen speed. As shown on Chapter 14 of "Scientific Programming", a `struct` is a sort of **complex** variable. You can think about it as a composite variable whose components are ordinary variables or other `struct` variables. Each single component can be addressed as the name of the `struct` and the name of the component separated by a dot `.`, as in `options.c_ispeed`, an unsigned long integer containing the port speed expressed in baud. Both the functions and the definition of the `struct` are in `termios.h`.

Once called, the functions above fill the `struct` with standard values, that can be modified acting on each `struct` component. There are many options to configure, but their full description goes beyond the scope of this book. Often, the default configuration of the port works well and most of the type you just need to disable the **canonical mode** flag. In canonical mode, devices provide data in form of strings terminated by a newline character (i.e., line by line). If you want to read single characters from the line you must disable the canonical mode with

```
toptions.c_lflag &= ~ICANON;
```

`c_lflag` is the component of `termios` defined as an unsigned long integer. It represents a binary coded set of flags (bits that can be either 1 or 0). `ICANON` is a constant, while the `~` character is a bitwise NOT operator. The `&=` operator combines a bitwise AND with an assignment operator. Indeed, what happens here is that the CPU reads the current content of `toptions.c_lflag` and perform a bitwise AND operation between this value and `ICANON`; the result of this operation is in turn assigned to `toptions.c_lflag`. In practice, `toptions.c_lflag` is set (its value is 1) if canonical mode is active, 0 otherwise.

According to the description given above, to put back the port in canonical mode, it is enough to be sure that the corresponding bit is 1, then

```
toptions.c_lflag |= ICANON;
```

that performs a bitwise OR operation between the current value of `toptions.c_lflag` and `ICANON`. Once `toptions.c_lflag` has been assigned properly, its value should be transferred to the port with

```
tcsetattr(fd, TCSANOW, &toptions);
```

Here, `fd` is the file descriptor returned by the `open` statement, `TCSANOW` a flag that instruct the port to change its properties immediately and `&toptions` is the address of the struct containing the actual configuration of the port.

```
1 #include <stdio.h>
2 #include <fcntl.h>    // needed for open
3 #include <termios.h>  // termios definition
4 #include <unistd.h>   // needed for read
5
6 int main() {
7   struct termios toptions;
8   int fd;
9
10  fd = open("/dev/tty.usbmodem1421", O_RDWR | O_NONBLOCK );
11
12  cfsetispeed(&toptions, 9600);
13  cfsetospeed(&toptions, 9600);
14
15  toptions.c_lflag &= ~ICANON;
16  tcsetattr(fd, TCSANOW, &toptions);
17
18  char ch;
19  char buf[255] = {0};
20  while (1) {
21    int i=0;
22    do {
23      int n = read(fd, &ch, 1);
24      if (n > 0) {
25        buf[i++] = ch;
```

```
26        }
27      } while( ch != '\r' && i < 255);
28
29      buf[i] = 0;  // null terminate the string
30      printf("%s", buf);
31    }
32 }
```

Listing sei.1 A program to read single characters from the serial line. The program prints each line received from Arduino on the screen.

You are now ready to read what Arduino writes on the port according to your configuration. A complete program to read data from Arduino is shown in Listing sei.1. The program read the serial line character by character. A line ends with a carriage return \r. Listing sei.2 provides an example on how to use canonical mode, instead.

```
1 #include <stdio.h>
2 #include <fcntl.h>    // needed for open
3 #include <termios.h>  // termios definition
4 #include <unistd.h>   // needed for read
5
6 int main() {
7   struct termios toptions;
8   int fd;
9
10  fd = open("/dev/tty.usbmodem1421", O_RDWR | O_NONBLOCK );
11
12  cfsetispeed(&toptions, 9600);
13  cfsetospeed(&toptions, 9600);
14
15  toptions.c_lflag |= ICANON;
16  tcsetattr(fd, TCSANOW, &toptions);
17
18  char buf[255] = {0};
19  while (1) {
20    int i=0;
21    int n = read(fd, buf, 255);
22    if (n > 0) {
23      printf("%s", buf);
24    }
25  }
26 }
```

Listing sei.2 A program to read text lines from the serial line. The program prints each line received from Arduino on the screen.

Choosing between canonical and non canonical mode depends on your application. In our

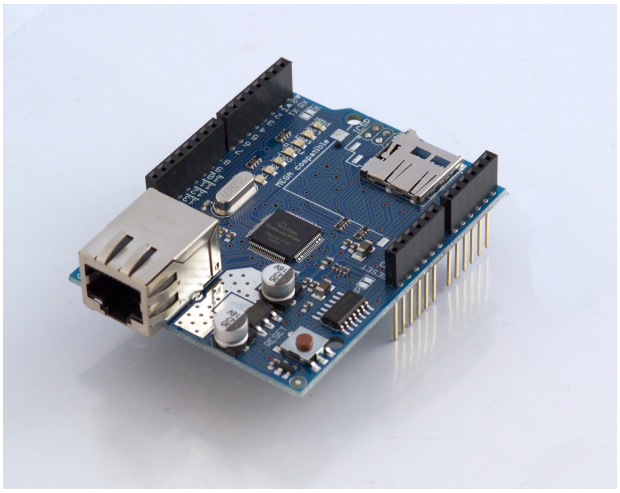

Figure sei.1 The Arduino Ethernet shield is a board that plugs on top of an Arduino board. This shield has an RJ45 connector to connect it to a plug via an RJ45 cord and a SD card slot. There exists shields with WiFi capabilities.

examples they are equivalent, since we assume that Arduino is writing text lines ended with a newline to the serial line.

Data read from the serial line can then be saved on your computer running the programs described above, for offline analysis. Using files on computers is the topic of Chapter 6 of "Scientific Programming".

sei.2 Connecting to the Internet

When collecting data, you may want to send these data over the Internet to be visualised and stored for future use. Data visualisation and storage is then demanded to a server connected to the Internet, receiving data from an Internet connected Arduino. Hence all the processing needed to render, visualise and store the data on disk is taken by the server and requires some standard programming skill (not necessarily in C: in most case you may want to use Java, Perl, Python or something similar). This side of the problem is out of the scope of this publication and is not covered here.

Of course your Arduino must be able to collect these data and send them over the Internet. For this you need what is called an Ethernet shield: an Arduino shield able to communicate with other devices using the Internet protocols.

Ethernet shields exist in a number of flavours: some of them have an RJ45 connector to plug them to the Internet by means of a cable, some other have WiFi capabilities. Usually they carry an SD slot on board to host a SD card, too.

There exist also Arduino boards with Internet capabilities on board, such as the Arduino YUN.

sei.2.1 Configuring the Ethernet shield

In order for any device to connect to the Internet, the device must acquire a unique identity on the network: the **IP address**. The IP address (IP stands for Internet Protocol) of a device is a unique identifier composed, in the IPv4 version of the standard, the most commonly used, of four bytes, each of which can have a value between 0 and 255. Few of them are reserved internationally and, in particular, subnetworks whose first two bytes are 192 and 168 are non–routable, i.e. packets sent over such a network cannot go beyond an Internet switch. In other words they can only reach those devices on the same physical network. Thats why devices in a home network usually have IP addresses like 192.168.x.y. The IP address of a device can be statically or dynamically assigned to it. In the first case, the device administrator tells the device its IP address: in this case the address is going to remains the same with time forever. A device not having a static IP address can ask for an IP address to any computer on the same network running as a **DHCP server** (Dynamic Host Configuration Protocol). Depending on the configuration of the server, available IP addresses can be assigned to the device randomly or based on the device identity.

Every physical device, in fact, brings a unique MAC (Media Access Control) address: a set of six bytes, usually expressed in the hexadecimal notation, as 5e:a4:18:f0:8a:f6. The MAC address is broadcasted over the network so that a DHCP server can choose if the request must be ignored, served using a randomly chosen address or with a fixed address.

Having an IP address in not enough for a device to communicate with other devices: data packets must reach a special device, called the **gateway**, that knows how to send data to the recipient. The gateway, too, must be assigned an IP address and its address must be known to the devices aiming to communicate with others.

IP addresses can be associated to strings composed of a **host name** and a **domain name**. All devices on the same network shares the same domain name, while host names are unique. A **DNS** (Domain Name System) is a device able to associate the IP address of any other device to its host name.

The last piece of information needed to setup a network device is the subnet mask. This is a rather technical element, but we can think of it as a way to identify the range of addresses a device can reach directly through a gateway. It is usually in the form of an IP address (but it is not, it is a mask) and often equal to 255.255.255.0, meaning that all the devices on the same network share the first three bytes of their IP address.

We now have all the ingredients to connect our Arduino to the network: first of all plug the Ethernet shield to the Arduino board (see Figure sei.2), then connect the shield to your router using an RJ45 cable. In order to configure the device you must know the MAC address of your shield (you can find it printed on the box or you can just invent it, provided it is unique in your network). Consider the excerpt of Arduino in Listing sei.3

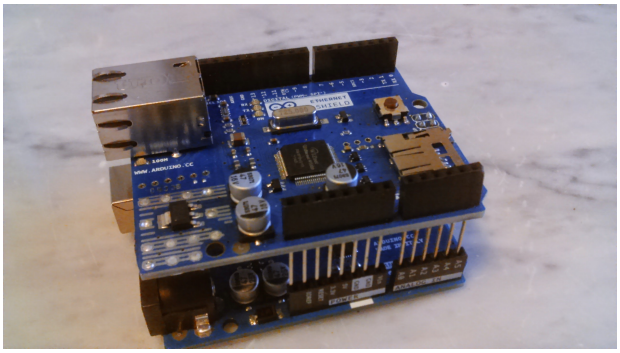

Figure sei.2 An Arduino board with an Ethernet shield mounted on it.

```
1 #include <Ethernet.h>
2 #include <SPI.h>
3
4 byte macAddr[] = {0x5e, 0xa4, 0x18, 0xf0, 0x8a, 0xf6};
5 IPAddress arduinoIP(192, 168, 1, 67);
6 IPAddress dnsIP(192, 168, 1, 254);
7 IPAddress gatewayIP(192, 168, 1, 254);
8 IPAddress subnetIP(255, 255, 255, 0);
9
10 void setup() {
11   Ethernet.begin(mac, arduinoIP, dnsIP, gatewayIP, subnetIP);
12 }
```

Listing sei.3 Ethernet shield configuration.

Including `Ethernet.h` and `SPI.h` is mandatory: the files contain the definition of the classes used in the sketch. The MAC address is defined as an array of bytes, each of which is represented as a pair of hexadecimal digits (thanks to the `0x` preceding each number). The IP addresses of the shield, the DNS and the gateway is given as an object of class `IPAddress`, as well as the subnet mask. The object constructor takes four arguments that represent the four bytes of the address. Our Arduino will acquire the IP address 192.168.1.67, in a network whose gateways address is 192.168.1.254; the gateway works also as the DNS in this case, while the subnetwork is restricted to those devices having an IP address like 192.168.1.x.

The `Ethernet.begin(mac, arduinoIP, dnsIP, gatewayIP, subnetIP)` call does the job: it configures the Ethernet shield as above (and, of course, it does that in the setup() method).

In many tutorials you can easily find a much simpler configuration, that reads as shown in Listing sei.4.

```
#include <Ethernet.h>
#include <SPI.h>

byte mac[] = {0x5e, 0xa4, 0x18, 0xf0, 0x8a, 0xf6};

void setup() {
 Ethernet.begin(mac);
}
```

Listing sei.4 Simple Ethernet shield configuration.

In this case the Ethernet shield acquires a dynamic IP address from a DHCP server on the network. In fact the `begin()` method of the Ethernet class exists in many variants (it is said to be **polymorphic**). To many novices the last sketch may appear much more convenient: its simpler and shorter and does not require the knowledge of too many parameters. However, the length of the source code has mostly nothing to do with the size of the sketch in the Arduino memory.

This happens because what is stored in the Arduino memory is not the sketch as you can see here, but the sketch in machine language. Microprocessors work using electrical signals representing data and instructions [1]. Because an electrical device can be easily found in two states (e.g. on/off), information (data and instructions) is represented as binary strings. A program for a microprocessor is then a long sequence of bits 0 and 1, not a flow of characters. The characters you write in the editor are translated into corresponding sequences of bits by the compiler (automatically invoked before uploading the sketch or when you click on the **Verify** button of the Arduino IDE). It is this long sequence of bits that is uploaded on the Arduino memory, not your sketch.

It happens that, in order for the shield to ask for an IP address to a DHCP server, the number of operations to perform is much larger with respect to those needed to assign manually all the parameters. As a result, the compiled program in the two cases is very different in size: the first sketch takes 2 634 bytes in memory, once added an empty `loop()` method; the latter takes 10 424 bytes! Its about a factor 4 more space!

The memory space of an Arduino is precious, since it is not so large: as a result you may prefer the apparently longer sketch of the first example to the second one.

sei.2.2 Using the Ethernet shield to collect data

Suppose you want to perform the following calorimetry experiment: take a resistor and wrap it in a waterproof material; then connect its leads to a voltage generator and make current flow through it. If R is the resistance of the device and V the voltage across its leads, the Ohms Law states that the current flowing is $I = V/R$. The resistor dissipates heat, because of the Joules effect, as $W = RI^2$, where W is the amount of energy per unit time. If you plunge the resistor into water, the energy released by the resistor causes the heating of the water and you expect that the temperature of the water raises linearly with time (see also Section otto.3 about temperature measurements).

You can perform this experiment using an LM35 connected to an Arduino to measure the water temperature versus time (the sensor leads must be made waterproof, of course, e.g. using some heat-shrink tubing). An Ethernet shield can then be used to send data to a computer.

Lets start looking at the Arduino sketch, shown in Listing sei.5.

```
#include <Ethernet.h>
#include <SPI.h>

#define PORT 5000
#define LM35PIN A0

byte mac[] = {0x5e, 0xa4, 0x18, 0xf0, 0x8a, 0xf6};
IPAddress arduinoIP(192, 168, 1, 67);
IPAddress dnsIP(192, 168, 1, 1);
IPAddress gatewayIP(192, 168, 1, 1);
IPAddress subnetIP(255, 255, 255, 0);

EthernetServer server(PORT);
boolean notYetConnected;

void setup() {
  Ethernet.begin(mac, arduinoIP, dnsIP, gatewayIP, subnetIP);
  notYetConnected = true;
}

void loop() {
    int i = 0;
    EthernetClient client = server.available();
    if (client) {
      if (notYetConnected) {
        client.println("Welcome!");
        notYetConnected = false;
      }
      if (client.available()) {
        unsigned long now = millis();
        int lm35 = analogRead(LM35PIN);
        now += millis();
        double T = 5000.*lm35/10240.;
        server.print(0.5*now);
        server.print(" ");
        server.println(T);
      }
    }
}
```

Listing sei.5 Arduino sketch to collect temperature data and send them over the Internet.

The first include directives are needed to use the Ethernet shield. Then we define two symbols: `PORT` is used to send data over the Internet, `LM35PIN` represents the Arduino pin to which the LM35 sensor is connected (`A0` in the example).

Besides the addresses used to configure the Ethernet shield, as described in the previous section, a number of data members are defined: in particular, the `EthernetServer` object called `server` is instantiated (i.e. created), listening on port `PORT`. This creates an object in the Arduino memory that connects to the Internet and waits for signals on the given port, represented as an integer (5 000 in the example).

The `setup()` method just initialise variables and configure the Ethernet shield. The most interesting part is in the `loop()` method. Here we instantiate an object called `client` belonging to the class `EthernetClient`. Such an object is returned by the `server` object that continuously polls the port to which is connected. If no client is connected, the server returns `NULL`. Then, as soon as client is found to be not `NULL`, and available for communication, we can send and receive data to/from it.

Before sending data we must get them: first of all we obtain the current time as the number of milliseconds elapsed since the beginning of the execution of the sketch (we dont care about the absolute time of the event). This time is returned by the function `millis()` and is represented as an unsigned long integer, i.e. a binary code of 32 bits. With 32 bits, the highest number that can be represented is $2^{32} - 1 = 4\,294\,967\,295$. Dividing this number by 86 400 (the number of seconds in a day) and by 1 000 we get about 50: this is the number of days during which the `millis()` function can work without reaching the overflow condition. In other words, there is plenty of time to perform our experiment.

Then we get the reading from the LM35 sensor using `analogRead` and measure the time again. Averaging the last measured time with the one previously measured provides a better estimate of the time of reading. Note that, in between, we just read raw data, in such a way we minimise the time spent in data acquisition and obtain the time with as much precision as possible. Computing the temperature in degrees is made after getting the time: the analog pin reading is a 10 bits binary number: its highest value (1 024) corresponds to an input of 5 V, i.e. 5 000 mV. The actual temperature, in Celsius, can be obtained reading the output voltage of the LM35 in mV divided by 10.

To transmit data to a remote client, its enough to call the `print`[1] method of the `server` object. We then print the time reading, a blank and the actual temperature in Celsius. Without any delay in the `loop()`, the sketch will read temperatures at a rate of one measurement every few milliseconds (quite fast, indeed). Of course, for an experiment like this, there is no need to obtain data with such a high rate, but there are cases in which data rate must be high.

In order to collect those data on a computer you need an **Internet client** that connects to the Arduino port 5 000, writes some data on that port to announce it (*knock, knock*) and waits for data. An example of such a program in C language is shown in Listing sei.6.

[1] or its `println` variant.

```
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>

#define PORT 5000
#define ADDRESS "192.168.1.67"

int main() {
  int len;
  int i;

  /* create socket */
  int sock = socket(AF_INET , SOCK_STREAM , 0);
  if (sock <= 0) {
    printf("Can't create socket. Error");
    return -1;
  }

  struct sockaddr_in s;
  server.sin_addr.s_addr = inet_addr(ADDRESS);
  server.sin_family = AF_INET;
  server.sin_port = htons(PORT);

  /* connect */
  if (connect(sock , (struct sockaddr *)&s , sizeof(s)) < 0) {
    printf("can't connect to the server. Error");
    return -1;
  }

  printf("CONNECTED: hit return to start DAQ\n");

  char message[255];
  scanf("%s", message);
  send(sock, message, strlen(message), 0);

  /* read */
  while (1) {
    unsigned char c;
    recv(sock, &c, sizeof(unsigned char), 0);
    printf("%c", c);
  }

  return 0;
}
```

Listing sei.6 A basic Internet client for communicating with an Arduino.

Briefly, we first create a so-called `socket` to make a connection between the **client** (running on a computer) and the **server** (on the Arduino). A socket works like a `FILE` in C language and is represented by an integer. It is create by the `socket()` function to which we must pass three arguments: the so–called **communication domain**, selecting the protocol family to be used for communication (`AF_INET`, a symbol defined in `sys/socket.h` that in turn is included in `arpa/inet.h`, selects the most commonly used IP protocol); the **socket type** (`SOCK_STREAM` is a *full–duplex* byte stream socket, i.e. a socket through which bytes can *pass* is both ways) and the specific **protocol** in the selected family (only protocol 0 exists).

We must then connect the socket to the same port to which the server is listening at: this is done with the `connect()` function, whose parameters are

- the integer representing the socket;
- a structure of type `socked` describing the socket type and protocol family, properly formatted using functions like `inet_addr` and `tons`;
- the size, in bytes, of the socket structure (it may change for different types of sockets).

Once connected, with `scanf` we just read a string (just a newline is enough) from the keyboard and send it to the server. This way the server answer and data acquisition can start. Data sent from the server are read with the `recv` function (one character at a time, in the example). The `recv` function takes, as parameters, the socket from which data are expected, the address of the variable on which data have to be stored in memory, the size of the latter and a flag (a combination of bits used to tell the function how to behave in special cases: this is usually set to zero).

In the above example the reading loop lasts forever (`while (1)`) and just print the received characters on screen. You can, of course, write data on a file until some event happens (e.g. key pressed, maximum number of data received, etc.).

sei.3 Using an SD card

Another possibility to store data persistently is to log them on an SD card. You can then retrieve data offline when data taking is over. Data can be stored in text files on an SD card. There are several Arduino shields with an SD card reader on board; you may also find cheap SD card readers to be connected to your Arduino board.

Upon loading headers `SPI.h` and `SD.h`, to have access to the SD card you need to initialise it with

```
int result = SD.begin()
```

If `result` is true, files can be created as

```
File f = SD.open("filename.txt", FILE_WRITE);
```

where `filename.txt` is the file name. You can then write characters on file `f` using

```
f.println(line);
```

or its `print` variant without the newline character at the end. Close the file using

```
f.close();
```

Chapter sette

Arduino specific functions

Functions is the topic of Chapter 7 of "Scientific Programming". Arduino functions work exactly as in C language. In fact, they are not exactly functions, but rather methods of the main class. However, they resembles and behave exactly as a function in C and we treat them as such. As in C, functions have access to all global variables (i.e. variables defined out of their scope, but not within the scope of another function), while variables declared within their body are local to them.

The `setup()` and `loop()` blocks seen above are in fact functions admitting zero parameters. The first is called at the beginning of the execution cycle, while the second is repeatedly called as long as Arduino remains on.

Most of the specific usages of an Arduino is performed through dedicated functions provided by the IDE itself. In this chapter we illustrate few of them.

sette.1 Setting up pins

In order to tell Arduino how to treat a digital pin, it is mandatory to set each of them as an **input pin** or an **output pin**. The latter is a pin to which you can assign a value and, correspondingly, you get a voltage on the pin. An input pin is a pin to which you connect an electrical signal, whose value can be read in the sketch.

The behaviour of a pin depends on its **mode** that can be of two types: `INPUT` and `OUTPUT` with obvious meaning. To set a pin as an input pin you can use (usually in the `setup()` function),

```
pinMode(pin, INPUT);
```

where `pin` is an integer constant or variable whose value ranges from 0 to 13, indicating the **address** of the pin. Pins correspond to connectors on the Arduino board numbered accordingly. You can then connect any electrical signal whose amplitude is within 5 V to pin number `pin`. A call to the reading function (see below) allows you to measure such value.

Conversely, to set a pin as an output one, you use

Figure sette.1 To tell which lead of the LED is the cathode look through its body: the thickest lead is the cathode (on the right in this picture). The picture has been taken from Wikipedia.

```
pinMode(pin, OUTPUT);
```

You can then set the state of this pin via software.

sette.2 Writing and reading digital pins

All pins can be used as digital pins, admitting only two values: `HIGH` and `LOW`. Some pin is available as analog pins (pins 3, 5, 6, 9, 10 and 11). When used as digital pins, output pins provide a signal of either 0 or 5 V, according to their state. To set a digital pin state you can use

```
digitalWrite(pin, HIGH);
```

or

```
digitalWrite(pin, LOW);
```

In the latter case you can find 0 V on the corresponding connector on the Arduino board, while in the opposite case, you can find 5 V. With a digital pin, then, you can power on and off any device requiring a voltage of 5 V (or less, if you can divide the voltage by an appropriate circuit). For example, you can switch on and off an LED. Just connect the anode of the LED to the pin and its cathode to the ground (see Fig. sette.1). Upon execution of the statement `digitalWrite(pin, HIGH);` the LED switches on brightly. The brightness of the LED depends on the current flowing through it. Depending on the color, LED's require a current of 15–20 mA to produce light. The current I flowing through a passive circuit element like the LED is given by the Ohm's Law:

$$I = \frac{V}{R} \qquad \text{(sette.1)}$$

black	brown	red	orange	yellow	green	blue	purple	grey	white
0	1	2	3	4	5	6	7	8	9

Table sette.1 Correspondence between resistors values and band colors.

where V is the voltage across its leads and R its **resistance**. The LED resistance depend on its **bias**, i.e. the sign and the magnitude of the voltage drop across its leads. In our application the LED is said to be **forward biased** and its resistance is almost negligible. The current flowing through it depends, then, on the power supply capabilities. Since Arduino pins can draw up to 50 mA, this is the current flowing through the LED. It is advisable to reduce it, in order to prevent damage of the LED and heating too much the board. To do so, you can limit the current adding a **resistor** in series with the LED. In order to compute the right value for the resistor you can still use the Ohm's law, taking into account that the voltage across the LED drops of about 2-3 V, depending on the color (2 for red, 3 for blue). From the Ohm's law

$$R = \frac{V - V_{drop}}{I} \,. \tag{sette.2}$$

To pilot a red LED for which $V_{drop} \simeq 2$ V using Arduino for which $V = 5$ V using a current of 15 mA you need a resistor whose resistance is

$$R = \frac{5 - 2}{15 \times 10^{-3}} = 200\,\Omega \,. \tag{sette.3}$$

You can then use a 220 Ω commercially available resistor. The value of a resistor is impressed on its body as a set of three coloured bands. To each color corresponds a value, as in Table sette.1. Indicating with n_i the corresponding value of the color of the band i, its value is

$$(n_1 \times 10 + n_2) \times 10^{n_3} \,. \tag{sette.4}$$

A fourth band indicates the precision of the value: it is usually silver (10 %) or gold (5 %). For example, a resistor with three bands coloured as brown, black, red, is a resistor for which $n_1 = 1$, $n_2 = 0$ and $n_3 = 2$, hence $R = (1 \times 10 + 1) \times 10^2 = 100\,\Omega$.

The circuit is shown in Fig. sette.2. In order to simplify connections you can use a **breadboard**: a board with a set of holes electrically connected along given lines.

Figure sette.3 shows the same circuit using a breadboard (the white board on the right). The black cable connects the `GND` pin of the Arduino board to the − line (coloured in blue) on the breadboard: all the holes aligned close to the − line are electrically connected to each other, so they have the same electrical potential. The cathode of the LED is then connected to the same potential, while its anode in inserted into the `a5` hole in the breadboard. In this breadboard all the holes with the same number are electrically connected to each other: in this way, a lead of the resistor (the one inserted in `b5`) is

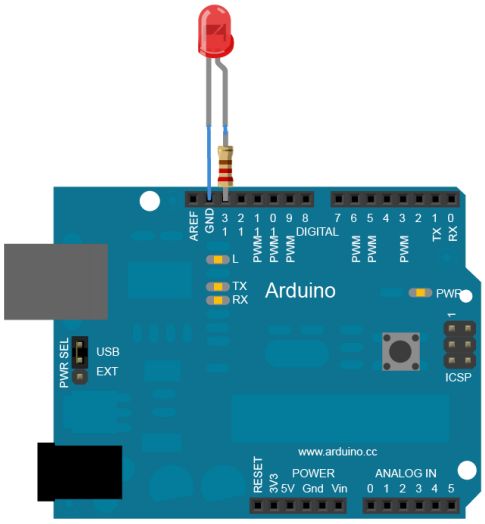

Figure sette.2 An Arduino board with a red LED connected to pin 13 through a 220 Ω resistor.

connected to the LED and the other to the green cable, connected to the pin 13 of the Arduino board.

Reading the value of a digital pin can be done using the `digitalRead` function, returning the pin value, as

```
int val = digitalPin(pin);
```

This function is useful, e.g., when you must tell the status of a push button or any other device having two states. The device must provide 0 or 5 V depending on its status. There is no need to draw lot of current to the input pin, in this case, then you may want to connect the push button to the Arduino pin by means of a large resistance (e.g. 1–10 kΩ).

Sometimes you want to wait for some external input to continue running your program. The external input can be configured as a logical values, as for a pushbutton. Instead of writing a piece of code with an iteration structure testing the value of a digital pin, you can use the `pulseIn` function that waits for a signal to become `HIGH` or `LOW` depending on its second parameter, as in

```
unsigned int val = pulseIn(pin, HIGH);
```

If the pin is `LOW` the execution stops here and continue as soon as the pin becomes `HIGH`, then `LOW` again, unless a timeout is specified as a third argument. The timeout is given in microseconds. The value returned by the `pulseIn` function is the duration of the signal

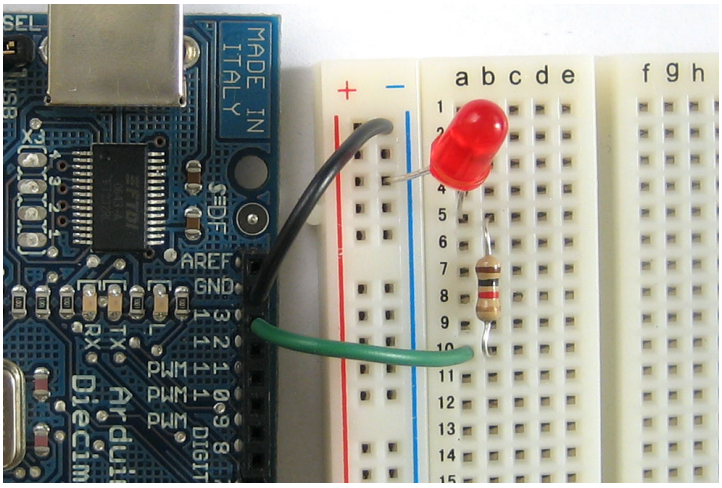

Figure sette.3 An Arduino board with a red LED connected to pin 13 through a 100 Ω resistor.

in microseconds. The minimal duration that can be detected is of 3 μs. The maximum is 3 minutes.

Laboratory sette.1 *Using an LED*

Connect a LED to your Arduino and write a sketch to switch on the LED at the beginning of the program. If successful, write a program that switch on and off the LED alternatively and forever, with a pattern chosen such that the LED keeps on for T_1 s and off for T_2 s. Using a pushbutton try to suspend the pattern when the button is pressed. If so, play a tone with a fixed frequency using a speaker. Find the wavelength of the seven basic tones on the web and play a scale in conjunction with the LED switching on and off.

Using a digital PWM pin you can as well produce a square wave, repeatedly calling `digitalWrite` with `LOW` and `HIGH` values in sequence, with the proper spacing in time. A train of square waves can be produced with the `tone` function that can be used to drive a speaker to produce sounds of given frequency ν. Its syntax is

```
tone(pin, frequency, duration);
```

The value of `duration` is given in ms, while the frequency in Hz. Both must be `unsigned int`. To produce a tone corresponding to the A tone at 440 Hz for 2.2 ms then use

```
tone(pin, 440, 2200);
```

in your sketch and connect a speaker to pin `pin` and to the `GND` one.

sette.3 timing

Arduino has some timing capability thanks to its clock. The `delay()` function suspend the execution of the program for the given amount of time, expressed in ms, like in

```
delay(1500);
```

that causes a pause of 1.5 s in sketch execution. Delays can be expressed in microseconds using the `delayMicroseconds()` function. Functions `micros()` and `millis()` return, respectively, the number of microseconds and the number of milliseconds elapsed since the Arduino started executing the sketch. They reset back to zero on overflow.

sette.4 Analog pins

If one of the analog pins is connected to an external voltage source whose values is between 0 and 5 V, you can read its values as a digit using the ability of Arduino to perform Analog to Digital Conversion (ADC). To read the voltage between `GND` and pin `pin` use

```
int val = analogRead(pin);
```

The result is an integer whose value ranges from 0 to 1023, proportional to the voltage. You can derive the voltage as

$$V = 5\frac{\mathrm{v}al}{1023}\,. \tag{sette.5}$$

If a PWM pin is defined as an output pin you can write a number between 0 and 255 in it using `analogWrite(pin, value)`. The pin, then, emits a square wave whose duty cycle depends on that value: if `value` is 0 the square wave is always off (i.e. zero amplitude), while if it is 255 it is always on (at the maximum amplitude of 5 V). Any value in between makes the pin emit a square wave with a proportional duty cycle. For example, using 128 as value, you get a square wave with 50 % duty cycle, i.e. a wave that half of the time is off and half is on. The duration of each pulse depends on the board and on the pins. The duty cycle is computed based on the clock frequency of 490 Hz or 980 Hz, depending on the board, provided by an internal clock. For a clock frequency of 490 Hz, each pulse has a maximum width of

$$\tau = \frac{1}{\nu} = \frac{1}{490} \simeq 2\,\mathrm{ms}\,. \tag{sette.6}$$

A duty cycle of 50 % means that the width of a positive 5 V pulse is 1 ms, followed by a 1 ms with at $V = 0$ V. The square wave is repeated until another call to `analogWrite()`

is done.

Laboratory sette.2 *Check the status of a battery*

Standard 1.5 V batteries can be used as long as they can provide enough current to the load. Alkaline AA or AAA batteries are designed to have a **capacity** of the order 2 Ah (Ampère–hour). The capacity is a measure of the amount of energy stored in the battery. A capacity of 2 Ah means that the battery can provide a current of 2 A for one hour, or a current of 1 A for two hours and so on. When batteries are exhausted, the voltage across their poles decreases with respect to its nominal value of 1.5 V, however, the battery can still be used if the voltage between the poles is larger than 1.3 V or so. The battery must be thrown (dispose it according to the rules in your country) if the voltage is as low as 1.2 V. Build a circuit in which the battery to be checked is connected to a resistor such that the current flowing through it is of the order of 100–200 mA and use Arduino to measure the voltage between the poles. If the battery is ok, switch on a green LED, if $1.3 \leqslant V \leqslant 1.2$, switch on a yellow LED, while if the voltage is below 1.2 V make a red LED blink.

Chapter otto

Measuring with Arduino

This chapter is devoted to Arduino applications, as Chapter 8 of [1] is devoted to computing applications. Of course, we are interested in applications of some interest for a physicist and, in general, for a scientist or an engineer. We do not consider applications like performing music[1], building robots, doing something funny with lights and so on.

Arduino pins are intended to measure or provide voltages between 0 and 5 V. If you can read or provide some voltage, you can do mostly any measurements, providing you are able to transform the measurement of some quantity into a voltage.

Fortunately enough, it is plenty of devices that do that seamlessly. You have just to understand how they work and write the appropriate sketch. Being a scientist, you do not want just to operate those devices: you need to understand their working principle and their limits in order to prevent false measurements or keep the measurement error under control.

In this chapter we briefly describe few types of sensors, without entering into the details about how to use them to take real measurements. These topics will be covered in the future, depending on the amount of donations received. We are going to use that money to buy samples of sensors, actuators and components to improve the content of this book. See Chapter zero about how to donate.

otto.1 Voltages

Arduino can easily measure voltages from 0 to 5 V with analog pins. The

```
analogRead(pin);
```

method (function) returns an integer value between 0 and 1023 proportional to the input voltage with respect to the ground. As outlined in Section sette.4, in order to get the value of the voltage in Volt, one must multiply the reading by a conversion factor given by $C = 5/1023 \simeq 0.00489$.

[1] There are people playing music with old floppy drives: moving their motors at different speed, they produce different tones that they combine to obtain incredible results.

In writing code one must consider that the value returned by `analogRead()` is an integer and a line as

```
int value = analogRead(A5)*5/1023;
```

leads to surprising results for people not used to write computer programs. The result of `analogRead()` is an integer whose value is between 0 and 1023. In fact the actual value can be even larger due to the fact that the Arduino ADC has 10 bits, while the integer is made of 32 bits. In particular, the 11^{th} bit can be set in case the ADC is saturated. When the integer returned by this function is multiplied by 5 we get another integer that is, in turn, divided by 1023. The latter being an integer, too, the result of this operation will be an integer. Hence, if `value`$\times 5 < 1023$, the result is zero. If `value` is greater than $1023/5 = 204.6$ the result is not zero, but it can be much different from what expected, since it will be just the integer part of the ratio.

The correct way to achieve the result is to force the CPU to represent the conversion factor as a floating point number. To do that it is enough to write either 5 or 1023 as such, just adding a decimal period after them. For example:

```
int value = analogRead(A5)*5/1023.;
```

is correct because the integer `value`$\times 5$ is now divided by the floating point constant `1023.` (note the period).

This kind of measurement can be useful for all those transducers that provides a voltage as their output or for electric measurements. Using this technique one can easily measure, e.g., the time constant of an RC circuit. An RC circuit is a series of a capacitor with capacitance C and a resistor with resistance R.

Connecting the circuit leads to a battery with voltage V_0 one can write the equation of the circuit as

$$V_0 = RI + \frac{Q}{C}\,, \tag{otto.1}$$

where $I = \frac{dQ}{dt}$ is the current flowing in the circuit. The equation can be rewritten as

$$\frac{dQ}{dt} = \frac{V_0}{R} - \frac{Q}{RC}\,. \tag{otto.2}$$

Setting

$$y = \frac{V_0}{R} - \frac{Q}{RC}\,, \tag{otto.3}$$

its derivative with respect to time t is

$$\frac{dy}{dt} = -\frac{1}{RC}\frac{dQ}{dt} = -\frac{y}{RC}\,. \tag{otto.4}$$

Let's integrate it:

$$\frac{dy}{y} = -\frac{dt}{RC}\,, \tag{otto.5}$$

and

$$\log \frac{y(t)}{y(0)} = -\frac{r}{RC}\,. \tag{otto.6}$$

Taking the exponential of both members we have

$$y(t) = y(0)\exp\left(-\frac{t}{RC}\right) \tag{otto.7}$$

and substituting the expressions for y

$$\frac{V_0}{R} - \frac{Q(t)}{RC} = \left(\frac{V_0}{R} - \frac{Q(0)}{RC}\right)\exp\left(-\frac{t}{RC}\right) \tag{otto.8}$$

where we made the dependency of q from time explicit. At the beginning the charge in the capacitor is null, then $Q(0) = 0$. On the other hand, at any time

$$\frac{Q(t)}{C} = V_C(t) \tag{otto.9}$$

where $V_C(t)$ is the voltage across the capacitor leads. We can then write

$$V_0 - V_C(t) = V_0\exp\left(-\frac{t}{RC}\right) \tag{otto.10}$$

or

$$V_C(t) = V_0\left(1 - \exp\left(-\frac{t}{RC}\right)\right)\,. \tag{otto.11}$$

The product $RC = \tau$ is called the **time constant** of the circuit. An RC circuit (Fig. otto.1) can be realised connecting the 5V pin of Arduino to a resistor. The latter is connected to a capacitor, whose free lead is then connected to the ground pin (GND) of Arduino. As soon as the circuit is closed, the capacitor starts charging and this can be observed connecting one of the analog pins to the lead of the capacitor connected to the resistor. Consider the following code in Listing otto.1.

```
1 float lastReading;
2 unsigned long time;
3
4 void setup() {
5   Serial.begin(9600);
6   lastReading = analogRead(A5)*5/1024.;
```

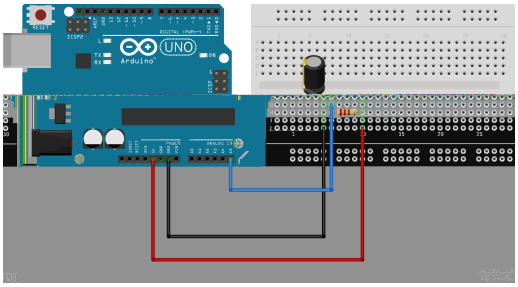

Figure otto.1 An RC circuit connected to Arduino.

```
}

void loop() {
  while (fabs(analogRead(A5)*5/1024. - lastReading) < 0.1) {
    time = micros();
  }
  for (int i = 0; i < 480; i++) {
    unsigned long now = micros();
    Serial.print(now - time);
    Serial.print(" ");
    float currentReading = analogRead(A5)*5/1024.;
    Serial.println(currentReading);
  }
  while (1) {
    // do nothing
  }
}
```

Listing otto.1 Following the charge of a capacitor.

The first empty loop is needed to start measuring as soon as the voltage across the capacitor changes enough. The value of `lastReading` is obtained as `analogRead(A5)*5/1024.` in the `setup()` method. In this loop we always measure the current time as the number of microseconds elapsed since the beginning of the sketch, returned by the `micros()` function.In this way, as soon as the program abandon this loop the variable `time` contains the start time of the measurements.

With this technique you can keep the circuit ready, not connected to the 5V pin, while uploading the sketch and starting the Serial monitor. As soon as you connect the circuit to the 5V pin of Arduino the capacitor starts charging and the voltage across it changes enough to start the second loop. Such a loop is executed 480 times and consists in just reading the voltage and sending its value to the Serial monitor. Together with the voltage, we measure the elapsed time and write its value, with respect to the time at the

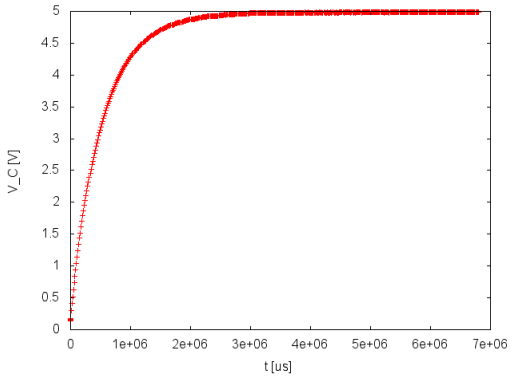

Figure otto.2 The voltage across the capacitor of an RC circuit versus time.

start of the measurements, to the Serial monitor, too.

When the measurements are over, the program stops thanks to the last `while` loop. Using a resistor R =10 kΩ and a capacitor $C = 47\,\mu$F, we obtained the result shown in Fig. otto.2.

The shape is consistent with that expected in these cases. Of course you can do a similar experiment discharging the capacitor. It is enough to restart the program after disconnecting the circuit from the 5V pin. The capacitor remains charged until its short circuited on the resistor. In this case the voltage V_C changes and the system takes 480 measurements.

Fitting the data you can easily obtain the time constant and verify that is consistent with what expected. Note that the actual resistance of a standard resistor is within 10 % of its nominal one, while for capacitors the tolerance can be as high as 40 %. With the experiment above one gets that

$$\tau = 0.534 \pm 0.011\,\mathrm{s}\,. \tag{otto.12}$$

The resistor has been measured with a multimeter obtaining $R = 9.94 \pm 0.01\,\mathrm{k\Omega}$. From these data we obtain for C the value $C = 53.7 \pm 1.1\,\mu$F, that differs from the nominal value of less than 15 %, well within the nominal tolerance.

Laboratory otto.1 *Capacitors in parallel.*

Collect up to five capacitors and realise as many RC circuits you can using from one to five capacitors in parallel. Measure the capacitance of each combination using the technique outlined in this section measuring the voltage across the system of capacitors. According to the theory the capacitance C_{eq} of the combination of N capacitors, each with

capacitance C_i, is

$$C_{eq} = \sum_{i=1}^{N} C_i \,. \tag{otto.13}$$

Verify the result comparing the difference between the expected and the measured values. Using N capacitors with the same nominal value you can even fit the distribution of C_{eq} versus N. What do you expect for this?

---

otto.2 Distances

Distance is a very basic kind of measurements that any scientist do at least at the beginning of his/her career while being a student. Transforming a distance into a voltage is not so straightforward, however you can easily find interesting devices on the market that, moreover, are so cheap that you can certainly afford them.

Most of these devices use sound waves. In order to prevent false measurements and to not hurt people in the neighbourhoods, they use ultrasonic sound waves, i.e. sound with very short wavelength (or very high frequency).

Sound waves travel in the air at constant speed of about $c \simeq 340$ m/s. The ultrasonic sensors are composed of a speaker and a microphone for ultrasonic waves: the speaker produces a train of waves that is reflected back from any obstacle in front of it (provided is large enough). The reflected sound is detected by the microphone after a time delay t that can be estimated as

$$t = 2\frac{d}{c} \tag{otto.14}$$

where d is the distance between the microphone and the obstacle. The factor 2 in front of the ratio in the right hand side of the equation is there because the pulses have to travel from the speaker to the obstacle and back to the microphone.

The sensor is equipped with an electronic circuit that measures the time and produces a single pulse whose duration is proportional to that time. You can then trigger the device with a digital pin and read the duration of the measurement pulse with the `pulseIn()` function. The value returned is proportional to t and can be used to get the distance inverting the above equation.

There are two kinds of sensors on the market: a type having four pins (`GND` and `VCC` to be connected to the Arduino `GND` and 5 V pins, the trigger pin and the return pulse pin) and a type having three pins (besides `GND` and `VCC` pins, there is one single pin for both input and output). The details on the operation is given in their data sheet.

It is important to note that, in order for the measurement to be accurate, the obstacle should be large enough to intercept the wave and reflect it back only once, and be not too close nor not too far from the sensor. Also, the reflecting surface should be perpendicular to the wave direction. The speed of sound in air depends on the temperature and

Figure otto.3 : The HC-SR04 sensor is an ultrasonic model composed by a speaker and a microphone. It can be used to measure distances.

the humidity of air, as well as on the presence of the wind. You should monitor these quantities in order to make a very accurate measurement and you should not rely on the calibration given by the supplier: it is advisable that you calibrate your device by yourself, measuring its response as a function of few distances (at least five) and taking a straight line fit to the data.

Consider, for example, the HC-SR04 module having four pins (Fig. otto.3): the pin labelled `GND` should be connected to the Arduino ground, the pin labelled as `Vcc` to the 5 V pin, while the `trig` and the `echo` pins should be connected, respectively, to a digital pin and to a PWM pin. Defining the pins as in

```
#define trigPin 2
#define echoPin 4
```

you must define the `trigPin` as an output pin and the `echoPin` as an input pin. When a pulse of duration of at least 10 μs reaches the trigger pin, the device emits a set of ultrasonic pulses. The microphone on board detects the same pulses as reflected from a surface in front of the speaker and provides, on the echo pin, a signal whose duration is proportional to the delay of the echo signal with respect to the emitted one. A function like the following is then going to start the emission of ultrasonic pulses:

```
void trig() {
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
}
```

You can call it in the `loop` block. The function puts the trigger pin to level 0 at the

beginning, then put it at level 1 for about 10 μs (`delayMicrosencod(10)`), then it put the pin back to 0. Once this is done, an ultrasonic signal leaves the speaker on board and, if any, reaches a surface in front of it, being reflected back. You must then wait for a signal on the echo pin and measure its width. To do that you can use the function

```
duration = pulseIn(echoPin, HIGH);
```

Such a function, assuming `echo Pin` is `LOW`, waits until the pin given as its first argument becomes `HIGH` (its second argument), then waits until it comes back to `LOW` and returns an integer proportional to the duration of the pulse that, in turn, is proportional to the delay between the ultrasonic pulse emitted by the device and the one detected. The delay is given in μs, hence, to estimate the distance of the obstacle you can compute

$$d \simeq \frac{ct}{2}\,, \tag{otto.15}$$

where t is the reading returned in `duration` and $c = 340\,\mathrm{m/s} = 340 \times 10^{-4}$ cm/μs the average speed of sound (this is an average value that, as said above, depends on many variables like humidity, temperature, wind, etc.).

The module is said to be capable to measure distances ranging from $d_{min} = 2$ cm ro $d_{max} = 4$ m. Given the average sound speed, the expected values of the time needed to detect the echo ranges from

$$t_{min} = 2\frac{d_{min}}{c} \simeq 2\frac{2 \times 10^{-2}}{340} \simeq 118 \times 10^{-6}\,\mathrm{s} = 118\,\mu\mathrm{s} \tag{otto.16}$$

to

$$t_{max} = 2\frac{d_{max}}{c} \simeq 2\frac{4}{340} \simeq 118 \times 10^{-4}\,\mathrm{s} = 11\,800\,\mu\mathrm{s}\,. \tag{otto.17}$$

You should take into account that the ultrasonic signal emitted by the speaker is quite large and, in order to correctly measure large distances, there must be almost no obstacles in a wide enough space around the line connecting the module with the object to be measured.

Laboratory otto.2 *Estimating the precision of an ultrasonic sensor*

In order to estimate the precision of the ultrasonic sensor you can do the following: given a distance d, measure the time returned by the ultrasonic sensor to detect the echo from that distance. Repeat the measurement many times, e.g. 10 000. Make an histogram of the measurements increasing an array of counters `count[N]`. The value of `N` should be kept low because of the limited memory, so you may want to subtract a **pedestal** from the measurement. For example, assuming $d \simeq 2$ cm the reading from the sensor is expected to be around 120. You can expect the value read from the sensor to be in the range $110 - 130$, hence `N` is 20 and you can store in `count[0]` the number of times you got 110, in `count[1]` the number of times you got 111 and so on. Once collected all the

data, show the content of the array in the serial monitor and stop the program execution (using, e.g. an infinite loop). Transforming back to μs the indexes of the arrays adding the pedestal to them, compute the average time $\langle t \rangle$ and its standard deviation as the square root of the variance σ^2 defined as $\sigma^2 = \langle t^2 \rangle - \langle t \rangle^2$. Remember that the average value of a variable x is given by

$$\langle x \rangle = \frac{\sum_i n_i x_i}{\sum_i n_i}$$

where x_i is a value obtained n_i times making $N = \sum_i n_i$ measurements. Consider to introduce a small delay (1 ms) after each measurement to avoid multiple echoes.

In order to make an accurate measurement, it is much better to calibrate the sensor reading the output values t_i for obstacles accurately positioned at fixed and known distances d_i (at least for five different positions). Then, fit the data of d_i versus $x_i = 1/t_i$ with a straight line $y = \alpha x + \beta$ (see Chapter 10 of [1]). Once α and β are known for your application (they may depend on environmental conditions of course) you can use them to measure, e.g., the position of a car as a function of time (using the timing capabilities of Arduino).

The precision of the position measurement depends on the precision with which you are able to measure the time. In our tests we measured an average value $\langle t \rangle \simeq 150$ and a variance $\sigma_t^2 \simeq 35$, then $t = 150 \pm 6$, with a precision of

$$\frac{\sigma_t}{t} = \frac{6}{150} = 4\,\%. \tag{otto.18}$$

Given the precision on time, you can derive the precision on distance σ_d using **uncertainty propagation techniques** [3], i.e. from $d = \frac{1}{2}ct$, assuming the uncertainty on c to be negligible,

$$\frac{\sigma_d}{d} = \frac{\sigma_t}{t}\,. \tag{otto.19}$$

Laboratory otto.3 *Measure the gravitational acceleration*

Once an ultrasonic module is working and precisely calibrated, measure the time needed for a car to fall along an inclined plane. The ultrasonic module should be places at the beginning of the track, and precisely oriented such that the sound is reflected by the car. Take the time elapsed since the start of the Arduino sketch and the distance measured with the ultrasonic model in as much steps as possible. Fit the data with the appropriate model, taken from any physics textbook (take into account that what you measure is the path length of the car, while on physics textbooks the equations of motion usually show the coordinates of the body in a reference frame with an horizontal and a vertical axis). The equations of the motion depend on the gravitational constant g: measure it from the collected data. You may need to consider the fact that friction is not always negligible. In order to minimise the effects of friction without using professional tools,

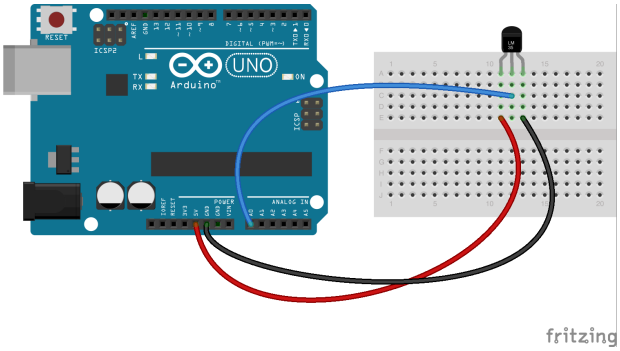

Figure otto.4 Using an LM35 temperature sensor with Arduino.

you can use a plastic ball that rolls along a raceway, as those used by electricians. The ball touches the raceway just on two points and friction effects are small. In this case you must take into account that the ball is rolling and should be considered as a rigid body. The equations of motion depend on the moment of inertia of the ball.

---

otto.3 Temperature

The measurement of a temperature can be done using a thermocouple: device using the so called **Seebeck effect** consisting in the production of an electromotive force by a pair of metals at different temperatures. Thermocouples produces a voltage proportional to the temperature of the probe. You can easily measure such a voltage using an Arduino. Thermocouples are not easy to calibrate, however they can be used on a very wide interval of temperatures, are very hard and can be used in a variety of difficult environmental situations (e.g. they can be immersed in liquids without special precautions) and does not dissipate heat into the sample.

There are integrated devices on the market that can perform more accurate measurements, much easily. The LM35 sensor, for example, is a device with three pins: two for powering it (`VCC` and `GND`) and one to be read as an analog input providing a very precise measurement of the temperature of the sensor body. With respect to thermocouples, a big advantage is that it does not need to be calibrated (though a good practice is to always check the calibration measuring few reference temperatures such as those of ice and boiling water). On the other hand, care must be taken in handling it (you cannot immerse it in water, for example, without protection) and it produces a bit of heat, being powered, thus increasing the systematic error of your measurements.

Using an LM35 is straightforward: just connect the `VCC` pin to the 5 V pin on Arduino and the `GND` pin to the corresponding pin on the board, as in Fig. otto.4. Then connect

the signal pin (the middle one) of the LM35 to any analog pin on Arduino. Read the temperature as

```
int reading = analogRead(lm35pin);
int temperature = C * reading;
```

where `C` is a constant that in principle can be computed taking into account that the LM35 provides a voltage output of +10 mV/°C. If the temperature of the LM35 is T °C, the voltage output is then $T/10^2$ V (i.e. a temperature of 100°C gives 1 V). Measuring it with an Arduino analog pin, whose resolution is 10 bits, the temperature is then

$$T = \frac{5}{1023} \times 10^2 \times r \tag{otto.20}$$

where r is the value read on the analog pin, hence `C = 500/1023`$\simeq 0.4888$. With this method, the minimum temperature you can measure is of course 0°C, while the maximum is, in principle, the one corresponding to a voltage of 5 V, i.e. 500° C. The LM35, however, is certified to work between −55°C and +150°C, then the maximum output voltage would be 1.5 V. It is not very useful to improve the dynamic range of the device changing the reference voltage from 5 V to a lower value, since the precision of the LM35 is typically 0.75°C, corresponding to an uncertainty on the output voltage of $\delta V = 7.5$ mV, while the resolution of the Arduino ADC is $5/1023 \simeq 5$ mV.

However, it can be useful to know how to improve the resolution of the ADC if you change the reference voltage used by the analog pins. By default the reference voltage is of 5 V, but the `analogReference()` function allows you to change it to other values. Possible values are

- `DEAFULT`: in this case the pin reads 1 023 when the input voltage is 5 V;
- `INTERNAL`: the reference voltage is 1.1 V so, if the reading of the ADC is r, the voltage is $1.1/1023 \times r$;
- `EXTERNAL`: the reference voltage is the one provided on the `AREF` pin that, in any case, cannot be larger than 5 V.

The values above are valid for Arduino UNO boards. For other boards, please check the documentation. Then, if you want to use the `INTERNAL` reference, add the following statement in the `setup()`:

```
analogReference(INTERNAL);
```

Then, when you read r on the analog pin, you get the value in V as

$$V = \frac{1.1}{1023} \times r \simeq 1.075 \times 10^{-3} r\,. \tag{otto.21}$$

Much more useful is to be able to read negative temperatures. In this case you need to provide a different grounding to the LM35. This can be achieved using silicon diodes, as the voltage drop across them is known to be 0.7 V. If the `VCC` pin of the LM35 is

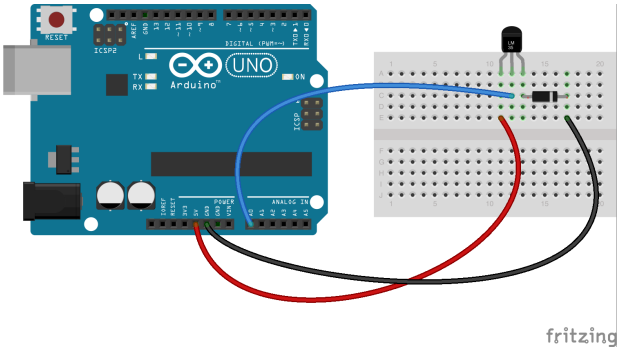

Figure otto.5 Using an LM35 temperature sensor with a forward biased silicon diode, to read negative temperatures.

connected to the 5 V line and its `GND` pin to a (properly biased[1], see Fig. otto.5) silicon diode that, in turn, in connected to the Arduino ground, the LM35 supply voltage is $5 - 0.7 = 4.3$ V (it works between 4 and 20 V), still enough to make the LM35 to work. Reading the output pin of the LM35 with Arduino still gets +10 mV/°C, with respect to the LM35 ground pin that now is at 0.7 V. Then, the temperature is given by

$$T = \left(\frac{5}{1023} \times r - 0.7\right) \times 10^2 \,. \tag{otto.22}$$

When $r = 0$ you now have that $T = -70$°C. For better precision you may want to read the voltage V_d on the LM35 ground pin, to be subtracted to the output pin reading r as in

$$T = \frac{5}{1023} (r - V_d) \times 10^2 \,. \tag{otto.23}$$

otto.4 Light

Light can be detected in a number of ways: there are integrated sensors providing accurate measurements of light intensity as well as light wavelength; the simplest light measuring device is, however, a photodiode. It provides a current whose intensity is proportional to the light intensity detected. Making this current flow through a resistor makes it possible to convert it into a voltage that you can read using Arduino.

RGB sensors can be bought for as low as ten dollars: they provide the relative intensity of red, green and blue light in a beam. More complex devices are needed to split white light into components of different wavelengths. You can build your own just using a

[1] Silicon diodes have a mark on one side to tell the user how to bias them. The mark position and type depends on the diode type and is usually documented in their data sheet.

simple diffraction grating (a Compact Disc is a very powerful grating, in fact, provided you remove its label) and measuring the intensity of the light as a function of the angle to which the light is diffracted. Using Arduino you can easily build a device that moves a sensor along a line via a stepper motor and measure the intensity of the light at different places.

There are also IR sensors on the market, able to detect **infrared** light. There are IR LED's, too.

otto.5 Magnetic field

Magnetic field can be measured using **Hall probes**. A Hall probe provides a voltage as output proportional to the magnetic field along the direction of its side. There are devices specifically tuned to be used in conjunction with Arduino to which you just supply some power, that gives a voltage proportional to the magnetic field intensity.

Another possibility is to use **magnetoresistive probes**. They exploit the magnetoresistive effect, consisting in the variation of the electrical resistivity of some material when immersed in a magnetic field. Many **digital compasses** uses this kind of sensors.

An example of such a sensor is the Digilent PmodCMPS digital compass: a device able to measure the intensity of a magnetic field in the range ± 8 G with a resolution of up to 2 mG over three mutually perpendicular axis.

The device (shown in Fig. otto.6) integrates a digital compass integrated circuit (IC) made by Honeywell, with a circuit that allows the user to read the three ADC's value using I2C, one of those many serial protocols to exchange data. A serial protocol is a way to send and receive data between a CPU and a peripheral device using just one single line (an electrical wire) called SDA (Serial DAta). Binary digits travel along this line one after the other in both directions. The communication being serial, it is important that each bit is transferred along the line at regular time intervals. A second line, called SCL (Serial CLock) is then employed to provide a clock signal allowing both transmitter and receiver to schedule the transmission/reception of signals properly. In order to use the device, then, you just need to feed it with 5 V, and connect the two I2C lines to an I2C enabled device such as an Arduino. The device (in the picture) comes with four pins: two of them must be connected to the ground (GND) and 5 V, respectively, while the other two pins are marked as SDA and SCL for communication.

You can connect these four pins to the corresponding pins on an Arduino board. The I2C pins location on Arduino depend on the board flavour. On the most popular Arduino UNO board they correspond to A4 and A5 pins. In this section, however, we are going to show how to use the digital compass using an Arduino Leonardo board, where the I2C pins are located on digital pins 2 and 3. See the Arduino - Wire web page for details about other Arduino flavours. The reason for using the Leonardo board is that it comes with built-in USB communication capabilities that makes it appear as an USB keyboard or mouse to a computer, hence we are going to use this feature to show the magnetic field vector on the screen.

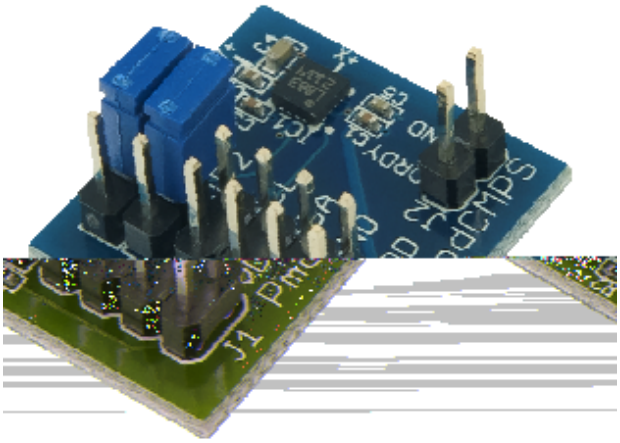

Figure otto.6 The PmodCMPS module: a digital compass to measure magnetic fields.

First of all, connect the four pins of the digital compass to the corresponding pins on Arduino, as shown in Figure otto.7. The need for the fifth (red) cable connecting the 3.3 V pin of the Arduino with its `A0` pin is explained below.

The `Wire` library is intended to let the programmer communicate using the I2C bus. It includes the `Wire.h` header file, that must be loaded into the sketch via the `#include <Wire.h>` directive. In the `setup()` method, initialise the library using the following code

```
void setup() {
  Wire.begin();

  Wire.beginTransmission(ADDRESS);
  Wire.write(0x02); //select mode register
  Wire.write(0x00); //continuous measurement mode
  Wire.endTransmission();
}
```

where `ADDRESS` has been defined as a constant with `#define ADDRESS 0x1E`.

Here `0x1E` is the address of the device given in hexadecimal digits, as defined by the producer (see the data sheet). The `beginTransmission()` method tells the Arduino to send data to the I2C device whose address is specified in parenthesis. Data travel from Arduino to the device until the communication is closed with `endTransmission()`. We use the `write()` method to send data over the line. In this case data represents commands or instructions given to the device.

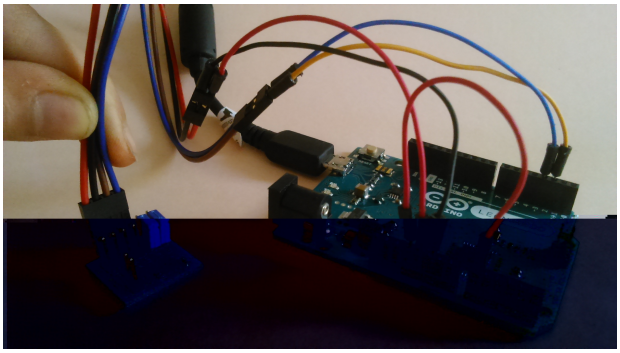

Figure otto.7 Connecting a digital compass to an Arduino Leonardo board.

I2C devices can be controlled via a number of registers, i.e. memory locations on board. According to the documentation, register `0x02` is the **mode register**, that can be used to configure the working mode of the device. We select the `0x00` mode, corresponding to continuous measurement. Other modes are `0x01` and `0x02` that, respectively, set the output of the device to be biased such that the current provided on the output pins is either positive or negative. Those modes can be useful when connecting the device to other circuits: some of them may be able to detect only positive or negative currents. Using Arduino these modes are useless.

Now the device starts measuring the magnetic field continuously and we can read the values in the `loop()`. At its beginning, it reads as follows:

```
void loop() {
  int x,y,z;
  Wire.beginTransmission(ADDRESS);
  Wire.write(0x03); //select register 3
  Wire.endTransmission();
  ...
```

Looking at the data sheet, one can see that registers from 3 to 8 contain the data output. Each axis reading is represented by two bytes: there are then three pairs of bytes, the first byte containing the most significant bits of the data and the second the least significant ones. Writing on register 3 one tells the I2C to send the data contained in that register to the Arduino. To receive data the `requestFrom()` method is exploited. It has two parameters: the address of the device and the number of registers whose content must be transmitted. The `read()` method, then, obtains the data when the device is ready to transmit it:

```
  //Read data from each axis, 2 registers per axis
  Wire.requestFrom(ADDRESS, 6);
  if(Wire.available()) {
```

```
    x = Wire.read() * 256;
    x += Wire.read();
    z = Wire.read() * 256;
    z += Wire.read();
    y = Wire.read() * 256;
    y += Wire.read();
}
```

The first pair to be read is the one containing the value of the magnetic field measured along the x–axis of the device. As said, it is composed of two bytes of which the first is the most significant. The value of this component of the field is then the one read from the first register multiplied by 256, to which we add the value of the second. The second pair contains the value of the field measured along the z–axis, while the last two bytes contain the value of the field measured along the y–axis.

Once obtained the components, the intensity of the magnetic field can be computed using Pythagoras' theorem, as the square root of the sum of the squares of x, y and z components.

In order to graphically represent the vector in space on the screen of a computer, one can exploit the capability of the **Leonardo** board to simulate a keyboard. Data collected by our device can be sent to the computer as if they were typed on a keyboard by a human. It is enough to format those data in such a way that they appear as such, then we must *convert* data into characters representing them (remember that numbers in the memory of a computer are represented using binary digit in the positional system, not as a string of characters).

Data come in the form of 12–bits integer numbers. For the purpose of illustrating another useful function we are going to scale them by a factor of 1000, then convert them into strings. We do that with

```
char xstr[6];
char ystr[6];
char zstr[6];
dtostrf(0.001*x, 5, 2, xstr);
dtostrf(0.001*y, 5, 2, ystr);
dtostrf(0.001*z, 5, 2, zstr);
```

the `dtostrf()` function takes four arguments: a floating point number, the maximum number of digits to be use to represent it, the number of non–integer digits to be shown and the address of a long enough array of characters to store the result. We then take each value, multiply it by 0.001, and store its character representation in an array of characters with exactly two non–integer digits. As an example, if the reading from the x-axis is $B_x = 23572$, multiplying it by 0.001 gives $B_x = 23.572$ that, expressed with exactly two non–integer digits is truncated to $B_x = 23.57$. If $B_x = 12000$, its character representation is going to be $B_x = 12.00$. The numbers will be then represented with five characters of which two are used for the non–integer part, one for the decimal period and the remaining two for the integer part. We then need arrays made of 6 characters:

one of them, in fact, has to be used as the terminator character of the string (the `NULL` character, marking the end of the string).

In order to show the vector representation on the screen we use GeoGebra: an open source mathematics software freely available for download. With `GeoGebra` one can draw vectors on the screen in a variety of ways, of which one is the following: open the 3D graphics view, then define the origin as a point whose coordinates are $(0, 0, 0)$ typing in the input box

```
O=(0,0,0)
```

Then, define a second point initially with the same coordinates

```
A=(0,0,0)
```

Now you can define a vector typing

```
vector(O, A)
```

You should see nothing on screen, since the vector length is zero, but if you change the coordinates of `A` you can see an arrow starting from the origin of the reference frame and terminating on the point located by `A`.

With this method we can now use the Leonardo board as the input device for `GeoGebra`: simply tell Leonardo to send characters to the computer on which `GeoGebra` is running, containing the commands to define the coordinates of point `A`:

```
sprintf(cmd, "A=(%s,%s,%s)\n", xstr, ystr, zstr);
int l = strlen(cmd);
int ok = analogRead(A0);
if (ok > 400) {
  for (int i = 0; i < l; i++) {
    Keyboard.write(cmd[i]);
  }
}
```

The first `sprintf()` statement formats a `cmd` string, defined as a long enough array of characters, such that the resulting string will be `A=(`x,y,z) where x, y and z are the content of the `xstr`, `ystr` and `zstr` obtained above, i.e. the readings of the compass. The second argument of `sprintf()` tells it how to interpret what follows: normal characters are inserted in the string as such, while characters preceded by a `%` sign are interpreted as **descriptors**. The `%s` descriptor tells `sprintf()` to interpret the corresponding parameter as a string. In plain C, the `%`n.m`f` descriptor can be used to format floating point values, where n and m are, respectively, the size of the field and the number of non–integer digits to show (we should have used `%5.2f`). However such a descriptor is not available on Arduino, that's why we were forced to use `dtostrf()`.

The `\n` characters at the end of the string represent the *newline* character.

We can then transmit each character forming the string to the computer as if they come from a keyboard with the `for` loop, that takes each character in the string and transmit it over the USB cable. If the pointer of the mouse has been previously located in the

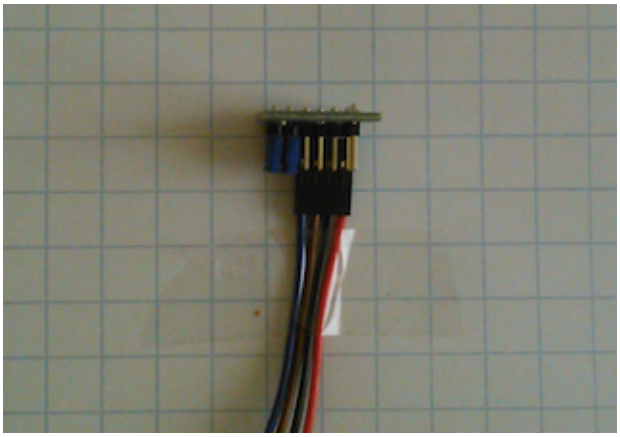

Figure otto.8 The digital compass can be firmly placed on a piece of paper just using a strip of scotch.

`GeoGebra` input box, those characters are entered as if they were pressed on a keyboard and the position of the `A` vector changes continuously if we change the orientation of the digital compass device. Changing `A` makes the vector to change, as can be seen from the movie at the following address: http://www.element14.com/community/videos/16560.

We are now able to read the output value of a digital compass, and it is time to make some more quantitative measurement. The sensor we are using in this example can measure fields up to ± 8 G. The reading on each channel is a 16–bit number, whose 5 most significant bits are used as the sign: the reason for using 5 bits and not just 1 is that the ADC on the sensor has only 12-bits. With 11 bits we can get numbers from 0 up to $2^{11} - 1 = 2047$. That means that a reading of $+2047$ corresponds to $+8$ G, at maximum gain, while a reading of -2047 corresponds to -8 G. Playing with the sensor one can see that sometimes also gets $+4096$ or -4096. That is an artefact of the ADC that, if saturated, returns zero and the reading appears as a 12-bit integer number.

Firmly placing the sensor in a given position allows us to measure the field produced by a magnet with a relatively good precision. In Figure otto.8 is shown how to fix the sensor in an effective and simple way: just use some scotch to fix it on a piece of paper.

With the sensor in the position shown in the picture, the x–axis is perpendicular to the paper, the y–axis points toward left and the z–axis points to the top of the figure.

If we want to make an absolute measurement, the readings made on each channel must be converted to the proper units (G) before being used and to do that we need to know the calibration constants, i.e. the numbers C such that the value B of the field in a given direction can be obtained as $B = C \times R$ where R is the reading. Those constants, having the units of G, depend on the gain of the device that can be adjusted using the sensor's configuration register. The default value of the gain is such that the maximum

value of the reading (2047) is attained when the device senses a magnetic field of 1.3 G (see pag. 13 of the data sheet).

In order to get the value of the magnetic field in a given direction returning R as value, we then must multiply R by 1.3/2047.

Suppose we want to measure the magnetic field produced by a small neodymium magnet, such as those that can be bought on Internet: what we are going to measure, in fact, with our sensor, is the sum of the earth magnetic field and the magnetic field produced by the magnet. If we want to know the latter, we need to subtract the effect of the first. To do that we can measure such a field keeping the magnet far from the sensor. We can then repeat the measurement of the earth magnetic field several times and average them for better precision:

```
float b[3] = {0.};
...
for (int i = 0; i < N; i++) {
  for (int j = 0; j < 3; j++) {
    b[j] += B(j);
  }
}
for (int j = 0; j < 3; j++) {
  b[j] /= N;
}
```

In the above listing, `b` is a three dimensional array aiming to contain the three components of the earth magnetic field, `N` is the number of measurements to perform before averaging and `B()` a method (function) that returns the i-th component of the magnetic field as read by the sensor, defined as follows:

```
int B(int i) {
  int b[3];

  Wire.beginTransmission(ADDRESS);
  Wire.write(0x03); //select register 3, X MSB register
  Wire.endTransmission();

 //Read data from each axis, 2 registers per axis
  Wire.requestFrom(ADDRESS, 6);
  if (Wire.available()) {
    b[0] = Wire.read() * 256;
    b[0] += Wire.read();
    b[2] = Wire.read() * 256;
    b[2] += Wire.read();
    b[1] = Wire.read() * 256;
    b[1] += Wire.read();
  }
  return b[i];
```

```
}
```

In fact the code can be made much more efficient: this way it always reads both the three components, then returns one of them. Since performance is not an issue we can keep the code as it is: **keep it simple** is always a good rule when no performance nor memory are an issue.

In our experiment we measured $(-284, -543, -17)$ as the three components of the earth magnetic field. In order to check that these values are correct we compute their values in G using calibration constants. Multiplying each of them by 1.3/2047 we obtain $(-0.180, -0.345, -0.011)$. The magnitude of such a vector is given by the Pythagoras' theorem as the square root of the sum of the squares of the components, i.e. $\sqrt{0.180^2 + 0.345^2 + 0.011^2} = 0.39$, not far from the expected value of about 0.5 G (in fact the earth's magnetic field ranges from about 0.3 to 0.6 G, depending on the location). Note, also, that we expect the field aligned almost along the earth's surface and in fact the size of the field in the horizontal plane is

$$|B_h| = \sqrt{B_y^2 + B_z^2} = \sqrt{0.345^2 + 0.011^2} \simeq 0.345\,\mathrm{G}$$

to be compared with $|B_v| = |B_x| = 0.180$ G.

To measure the magnetic field, that depends on coordinates, one can put the magnet in a given place and make the reading. Subtracting the earth magnetic field gives the magnet field that can be written directly on the piece of paper used to make the measurements.

Fig. otto.9 shows how to proceed: the metal ring seen in the picture close to the sensor is a small neodymium magnet. The position of the magnet with respect to the sensor can be obtained from the graph paper. For each position we read the three values of the field as provided by the sensor and write them down on the paper in the position occupied by the magnet.

In this way we can obtain a complete map of the field, measuring the values at different positions. Remember that the values provided by our sketch are now the readings in excess with respect to the earths magnetic field. We can get the magnetic field of the magnet after subtracting from each component the corresponding component of the earth's magnetic field measured as explained above.

Taking the values on the top $(57, 1, -19)$ we can see that the magnetic field provided by that small magnet has almost no component along y (left), so it points below and toward the reader (the z-axis is perpendicular to the figure and point towards it).

The values of the magnetic field components in G are $(0.036, 0.000, -0.012)$ G (just multiply the readings by 1.3/2048). The magnitude of the vector is then 0.038 G.

Moving the magnet closer its magnetic field increases. For example, when the magnet is in a different position, closer to the sensor, the readings are $(833, 703, -1254)$ corresponding to $(0.529, 0.446, 0.796)$ G, and the strength of the field is 1.05 G.

This kind of magnets provides a field that decreases strongly with the distance, as you can see from the values measured by our sensor.

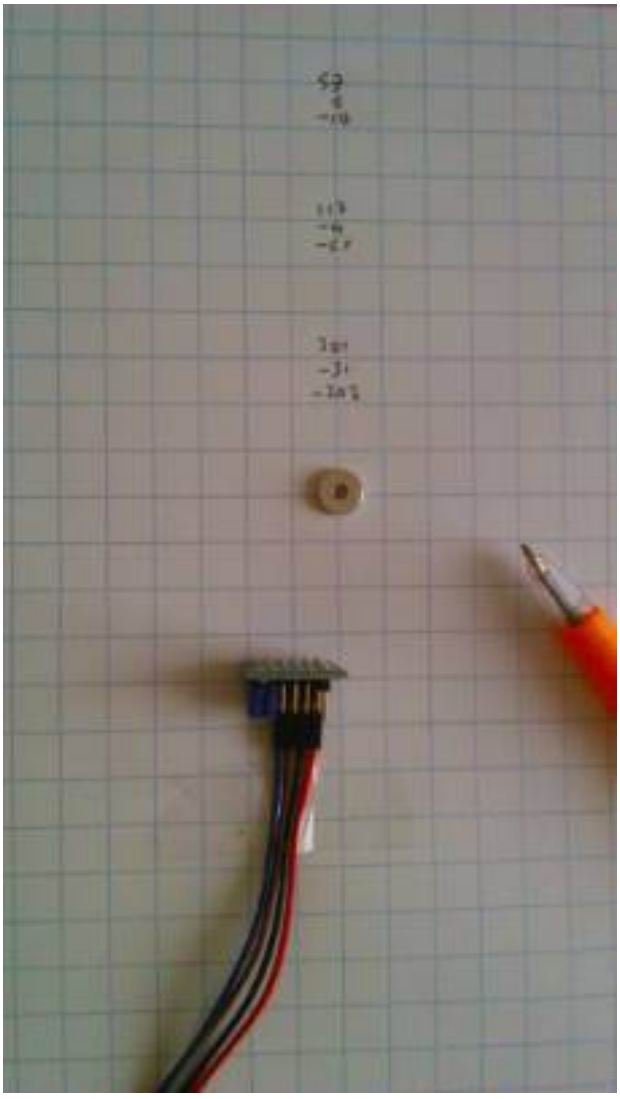

Figure otto.9 Measuring the magnetic field of a neodymium magnet.

Using a similar technique you can measure the field generated by any magnet, either by moving the magnet or the sensor in space. You can then compute a complete map of the field or of the field strength.

otto.6 Acceleration

An **accelerometer** is a device able to detect its acceleration and provides an electrical signal proportional to it. In most modern smartphone there are accelerometers used to detect when the device is rotated from the landscape to portrait direction. The same devices are commercially available for few dollars and can be easily connected to Arduino. Most of them comes in the form of a IMU (Inertial Measurement Unit) carrying three accelerometers and as much gyroscopes, each aligned perpendicular to the others, that can provide accelerations in the three directions. Using a IMU you can measure the acceleration along three mutually perpendicular directions and the Euler angles with respect to the device reference frame.

Sometimes IMU devices are equipped with magnetometers, too, to measure te magnetic field components. In the market you can find IMU devices for as low as 30 dollars or so. Depending on their sensitivity and performance, IMU devices can cost much more (up to 200 dollars). A triple axis accelerometers, instead, can cost as low as about 15 dollars.

Bibliography

[1] L.M. Barone, E. Marinari, G. Organtini, F. Ricci–Tersenghi, "Scientific Programming", World Scientific. An italian version exists, published by Pearson ed. with the title "Programmazione Scientifica".

[2] Giovanni Organtini, "Scientific Programming++", a free addendum to "Scientific Programming", available on http://www.scientificprogramming.org.

[3] Louis Lyons, "A practical guide to data analysis for physical science students", Cambridge University Press.