Using Arduino Boards in Measurements for dummies

M. Parvis

..Quid fiet? Tu occupatus es, vita festinat: mors interim aderit, cui, volis nolis, vacandum est

Seneca

..and then what will happen? You are always busy, you have time for nothing, but the life runs, and suddenly your time will arrive, for which either willing or not willing you will have to find time...

Ver 1.7 - August 2016

Contents

Chapter 1

Legalese

The software here described has been developed with freely available tools such as

- Visual Studio express edition `http://www.microsoft.com`,
- Mono/Xamarin `http://monodevelop.com/download`,
- Arduino environment `http://arduino.cc`
- Java and NetBeans `http://java.sun.com`,
- Scilab `http://www.scilab.org`,
- LibreOffice `http://www.libreoffice.org`

This document written in Latex under OSX and converted into `pdf` by using the latex distribution MaxTex available at `http://tug.org/mactex/`.

Chapter 2

Forewords

2.1 STOP (at least for a moment) and READ THIS

This is a rather long tutorial you can easily get lost in. If you are in hurry read at least these few lines before digging into the material:

- Arduino is simply a small board which contains an 8-bit Atmel microcontroller and the minimal circuits for supplying it and connecting it to a PC via serial communication in order to download the code into the microcontroller. A simple working environment can be downloaded that enable the communication with the PC. While this is very handy and you can have the board up and working in seconds, the overall performance is really minimal in comparison with any PC and DAQ board you can imagine and this has at least a couple of consequences.
- Firstly, if you are familiar with high level languages (e.g. Labview© Matlab© C#, Java...) and Integrated Development Environments (e.g. Visual Studio, NetBeans,...) be prepared to get back to the iron age: Arduino itself and the tools for working with it are light years back the actual state of the art.
- Secondly, if you are familiar with the possibilities and power of actual PC be prepared to get back to the copper age: the memory you are dealing with can be measured in terms of kilobytes not giga or megabytes. The microcontroller does not contain a floating point unit. While the floating point operations can be in principle provided in software, the real capability is hampered by the memory limitations.

- In addition to the above limitations, the C++ implementation on the Arduino compiler is not complete, so special care must be taken in the code development. As a rule of thumb, the general advice is to use only integer math to avoid problems.

2.2 Content and philosophy

This is not a course on programming microcontrolles. If you wish to learn tips & tricks on how to get every drop of power from a microcontroller you are not in the right place.

This is a primer designed to allow students of courses of master degree regarding Automatic Measurement Systems at Politecnico di Torino to get acquainted with a basic use of Arduino boards for data acquisition.

An Arduino based custom automatic measurement system can basically be arranged in two ways:

- By developing a specific hardware interface (e.g. an LCD panel) to be coupled to the Arduino board to obtain a complete acquisition system (i.e. a device which gives the user a result he can directly see and interpret).
- By connecting the Arduino board to a PC using the PC as the man-machine interface. This requires writing not only the program which runs on the Arduino board, but also a program which runs on the host PC and is able to display the results. This way the Arduino board is used as a DAQ card.

This document ONLY deals with the second approach.

This tutorial comprises a set of Arduino programs and a skeleton of PC program, which can be customized to arrange a complete acquisition system. Here a list of the provided sketches with a brief description. Remember that, due to the way the Arduino software is designed, each program must lay inside a directory with the same name.

- AnalogReadSerial.ino is taken from Arduino examples and is used as the first program to discuss the Arduino basics
- SerialWrite.ino shows the effect of the internal Arduino buffering when sending data to the PC
- SerialRead1.ino shows how to read data from a PC by using a blocking approach based on `Serial.readBytesUntil`
- SerialRead2.ino shows how to read data from a PC by using a non-blocking approach based on a char reading
- SerialRead3.ino shows how to read data from a PC by using an interrupt based procedure
- MeasureOnDemand.ino shows how to control an Arduino board so that it takes a measurement only when requested
- MultiMeasureOnDemand1.ino shows how instruct the Arduino board to take several measurements sending them back to the PC without storing them inside the Arduino board. This example uses the non-blocking approach discussed in SerialRead2.ino

- MultiMeasureOnDemand2.ino behaves like MultiMeasureOnDemand1.ino, but employs the blocking approach described in SerialRead2.ino
- MultiMeasureOnDemand3.ino behaves like MultiMeasureOnDemand1.ino, but employs the interrupt approach described in SerialRead3.ino
- MultiChannelMeasure.ino shows how to take measurement from several channels (multiplexing) and how to manage the ADC internal reference
- MultiMeasureBinary.ino shows how to transmit data in binary mode to speedup the transmission. This example employs an IEEE488.2 style coding for fixed length binary blocks
- MultiMeasureBuffered.ino shows how to allocate a local buffer where measurement can be stored at high speed without relying on the serial transmission speed.
- TimedAcquisition.ino shows how to use a timer to achieve a specific sampling rate not available otherwise.
- MeasureADCInterrupt.ino shows how to use a timer to fire the ADC and be interrupted ar conversion end.
- ManualRead.ino shows how to directly access the ADC registers to have it measuring channels not accessible via `analogRead`, such as the internal temperature sensor.
- DigitalInputPullup.ino shows how to configure, read and write digital channels.
- DigitalOutput.ino shows how to use register access to speed-up the digital output.
- PWM.ino shows how to use `analogWrite` to get an analog output.
- ArduinoCalUse.ino shows how to use the `EEPROM` class to store and read values form the Arduino non-volatile memory.

If you wish to learn more on Arduino boards and on ATMega microcontrollers there are a lot of resources on the internet you can browse for free.

If you wish to learn more on C# there are a lot of resources on the internet you can browse for free. Simply google *C# tutorial*.

Chapter 3

Getting started: my first Arduino program

BEWARE
This tutorial is designed to be suitable for Arduino IDE starting from version 1.0.1. New IDE releases may contain additions and code facilities which may help you speeding up the coding. One specific example is the `Serial` classes which may contain the possibility of formatting a message sent to the serial line by using the standard (e.g. `%i... %s... %f`) sintax used in c programs by using a specific `Serial.printf` method. You may check if your environment supports this method, otherwise you can use the `Serial.print` and `Serial.println` methods as reported in this tutorial. More details in chapter 4

3.1 Arduino program structure

Arduino is provided with a simple form of operating system which allows you to load and run programs. An Arduino program is always composed by two basic procedures `setup` and `loop`.

The `setup` procedure is executed at startup, i.e. after an Arduino reset. The Arduino boards can be reset in hardware, by pushing the button which is present on the board and are also reset automatically when the PC starts the connection via the `USB` connection. In the setup procedure you may insert all the initialization code (e.g. memory allocation, flag initialization,...) which is required to run your program.

```
void setup() {
 // this runs after reset
 //put here all the init code
}
```

After the `Setup` procedure finishes, the Arduino operating system starts running the `loop` procedure. Each time the procedure ends, the operating system restarts it. You can think of the procedure is encapsulated within a `while(true){}` loop, so there is no reason to insert loops inside the procedure itself.

```
void loop() {
// The loop routine runs over
  //and over again forever:
}
```

Of course, in addition to these procedure you can add other procedure you may wish to implement.

3.2 A simple read example

This simple program reads the analog voltage on pin 0 and sends back the raw ADC result to the serial line. This example is taken from the Arduino examples

The setup routine in this case only contains the initialization of the serial line to a speed of 9600 bits per second.

```
void setup() {
// initialize serial communication
  Serial.begin(9600);
}
```

Once the line is open, the main loop routine reads an analog value, sends it to the serial line, wait for one second and then cycles again

```
void loop() {
  // read the input on analog pin 0:
  int sensorValue = analogRead(A0);
  // print out the read value:
  Serial.println(sensorValue);
  delay(1000); // pause 1s and repeat
}
```

Some remarks about this simple example:

- As you can see, the program only uses integer maths: code 0 represents zero volts, while code 1023 represents the maximum voltage, in this case 5 V. Later we will see we can also use floats, though with some limitation
- This program uses a `C++` object of type `Serial`, which manages the serial line transmission. Since this kind of transmission is the only we can use with most Arduino boards, we will discuss it deeply in the following section.
- This example employs a free-run approach, i.e. data are transmitted without any synchronization to the PC and therefore we cannot precisely tag the measurement in time. In general this is not the behavior we want from an instrument, we will discuss this point later.

Chapter 4

Arduino data conversion and printing (Serial.print, sprintf, scanf,....)

This chapter deals with the common problem of data conversion from and to strings.

Even though Arduino environment employs the conventional `C++` sintax, which turns out in using `sprintf` for converting numeric data to strings and in `sscanf` for decoding and converting strings to numeric values, not all basic facilities of `C` language are implemented and specifically the float management can be buggy and with limitations.

4.1 Printing float variables (encoding to strings)

Printing (via serial line as Arduino does not have a display...) operation results in arduino can be done ins everl ways. In following chapters you will see how to manage the serial line mainly using the object `Serial` in order to send and receive data from a PC; in this section we will discuss several limitations connected to the Arduino environments.

The `C` way of converting a float to a string is to use the `sprintf` coomand, i.e.

```
char buf[20];
sprintf(buf,"Float: %f", buf,aFloat);
Serial.print("Direct sprint convertion gives ");
Serial.println(buf);
```

Unfortunately in most arduino environments a buggy implementation of the float management leads to useless results, with a question mark instead of the value. The previous example usually gives:

```
Direct sprint convertion gives Float: ?
```

If the problem is limited to printing a result, there is a simple way to overcome this issue by using the `Serial.print` method provided by the Arduino enviroment, i.e. by calling `Serial.print` once for each variable to print

```
float aFloat=1.2345678; //float to print
Serial.print("Arduino serial print ");
Serial.println(aFloat);
```

```
  Serial.print("Arduino serial print with more digits ");
  Serial.println(aFloat,6);
```

This option, though not elegant, provides the correct result and let you to specify the number of digits for the float conversion;

```
Arduino serial print 1.23
Arduino serial print with more digits 1.234568
```

Another possibility is to use `dtostrf` to convert the float to a string and then `sprintf` to combine the different field in one string

```
  char buf[20];     //space for dstostrf
  int w=3;
  int p=6;
  dtostrf(aFloat,w,p,buf); // emulate %w.pf field with p decimals
  char total[40];   //space for the combined string
  sprintf(total,"dtostrf converion gives  %s", buf);
  Serial.println(total);
  //use total with more fields
  int anInt=123;
  int anInt2=9923;
  sprintf(total,"Mixed conversion float init int %s %02d %i", buf,anInt,anInt2);
  Serial.println(total);
```

4.2 Decoding strings

Decoding strings can be obtained by using the `sscanf` command by using the common field specifier of `C`, i.e. `%f, %X.Yf,....` even though this NOT ALWAYS works!. As an example, the following code:

```
  char aString[]={"1.23456"};
  sscanf(aString,"%f",aFloat);
  Serial.println(aFloat,4);
```

usually produces:

```
1.2346
```

i.e. the expected result with four decimal digits.

In the case the above solution fails and the result is `0.00000`, another possibility is to employ the `atoi` and `atof` instructions:

```
  char aString[]={"1.23456"};
  aFloat=atof(aString);
  Serial.println(aFloat,8);
```

produces:

```
1.23456788
```

Chapter 5

Serial communication

This section deals with the problem of sending and receiving data to/from Arduino boards via serial line. The problems connected with the serial line management are mainly related to the slowness of the serial line with respect to the microprocessor speed: opcode execution takes nanoseconds to microseconds, depending on the microprocessor type, while sending a single character on the serial line may take up to 10 ms (e.g. at a baud rate of 9600). This great speed difference usually suggests to avoid using simple 'atomic' operations (i.e. operations that are executed in block without interruptions such as the procedures that read a complete string blocking the program until the reading is complete). Using an atomic operation is not totally forbidden and often you can find examples of this approach due to its easiness, but results is a tremendous waste of microprocessor power, since the microprocessor itself stops doing anything during the data read.

5.1 Serial line speed

Transmitting data on a serial line means transmitting one bit at a time by using a timing which is defined by the serial line baud rate. The baud rate is the number of bits which can be transmitted in one second. Baud rates have historically defined values. Commonly used values are 2400, 4800, 9600, 19200. Higher values may be used and nowadays very high speeds can be achieved depending on the hardware. However you have to test each speed above 19200 to see if it is operational before relying on it. Among the higher speed which are often operational we can cite 38400, 57600 and 115200 (bits per second). Higher speeds can be employed, though with some limitations, more details on this point later.

The baud rate has a great impact on the serial line use. According to the ASCII coding, a character may be composed by 5 to 8 bits, with a great preference nowadays for 8 bits, which allows one to transmit easily not only characters, but also raw bytes. In addition to the 8 bits, the serial line (which is inherently asynchronous) needs to transmit at least two sync bits so that each character usually[1] requires 10 bits. This means that at a baud rate of 9600, each character needs about 10 ms to be transmitted (i.e. ten times 1 ms) and about 100 μs at a baud rate of 115200. These values are orders of magnitude greater than the cycle time of every microprocessor, which can vary in the range of nanoseconds to microseconds, therefore special attention must be paid when using the serial line.

[1]The actual number of bits may range between 7 and 12 depending on the configuration of the serial line, however nowadays most serial lines employ 10 bits.

5.2 Sending data to the PC: SerialWrite.ino

Sending data to the PC in an Arduino environment is in principle very simple and is obtained through `Serial.print` and its brothers (`Serial.println`, and `Serial.write` for not formatted, binary send).

Since `Serial.print` accepts different types of parameters (strings, integer, floats,..) and provides the required conversions, no other work is in principle required. The introductory example of ReadAnalogSerial shows this.

However a deeper analysis of the `Serial` object shows some tricky points you have to be aware of.

Since the data transmission is slow, the `Serial` object implements an internal buffer of 64 bytes. This means that each time a `Serial.print` method is called, the system writes the characters into the buffer (if there is space in it) and then returns, leaving to the Arduino kernel the burden to actually sending the characters. While this background sending may slightly slow downs the processor apparent speed, the net result is that we can think of having transmitted the data much faster than in reality. The problem is that, if the buffer gets full, the `Serial.print` method has to wait until it can put the data into the buffer, so the timing abruptly slow down.

The following code fragment may be used to show what happens sending many data to the serial line.

```
//SerialWrite: sending data to PC

void setup() {
  // initialize comm:
  Serial.begin(9600);
  for (int cycle=0;cycle<5;cycle++){
   unsigned long before=micros();
   Serial.print ("1234567890 sent in ");
   unsigned long after=micros();
   Serial.print(after-before);
   Serial.println(" us");
  }
}


void loop() {
 ;
}
```

The program takes the time before and after sending a block of data and repeats this operation five times. The result of this operation is:

```
1234567890 sent in 188 us
1234567890 sent in 176 us
1234567890 sent in 7568 us
1234567890 sent in 19760 us
1234567890 sent in 19760 us
```

The program starts sending 19 character (i.e. `1234567890 sent in`). These characters are inserted into the 64 byte buffer, so that the operation takes less than 200 μs, i.e. less than the time required to deliver a single character, which is about 1 ms at a baud rate of 9600.

The second time other 19 characters are pushed to the buffer which now contains the original 19 characters plus `188 us` plus the newline (which is actually composed of two characters i.e. `\r\n`), i.e. a total of 46 characters. All the characters can be contained inside the buffer, thus the operation takes again less than 200 μs. After the second line is printed, the buffer contains 54 characters (i.e. the original 46 plus `176 us` plus a newline). The third time the programs tries queueing other data, since no character has been actually sent, the buffer has space only for 10 character. The `Serial.print` therefore has to wait until at least 9 characters are sent, i.e. a little less than 9 ms since part of the first character has been sent. The subsequent cycles find the buffer full, therefore the time required to deliver all the 19 characters is of about 19 ms, plus the microprocessor connected time.

The `Serial` object contains a method to be sure the buffer has been emptied, i.e flushed. By using this method one is sure to leave an empty buffer and is sure to continue only after the message has been sent (with the exception of the last character, which is being sent just after the buffer is emptied).

```
for (int cycle=0;cycle<5;cycle++){
   unsigned long before=micros();
   Serial.print ("1234567890 sent in ");
   Serial.flush();
   unsigned long after=micros();
   Serial.print(after-before);
   Serial.println(" us");
   Serial.flush();
 }
```

The result of this code is

```
1234567890 sent in 17716 us
1234567890 sent in 19760 us
1234567890 sent in 19760 us
1234567890 sent in 19760 us
1234567890 sent in 19760 us
```

As you can see this time the requested time is about 19 ms plus the microprocessor time, with the exception of the first line which is slightly faster since the first character can be immediately delivered to serial line hardware.

If we want to avoid this problem, we can add a 1 ms delay after the last flush

```
void setup() {
  // initialize comm:
  Serial.begin(9600);

  for (int cycle=0;cycle<5;cycle++){
   unsigned long before=micros();
   Serial.print ("1234567890 sent in ");
   Serial.flush();
   unsigned long after=micros();
   Serial.print(after-before);
   Serial.println(" us");
   Serial.flush();
   delay(1000);
 }
```

to insure the character is sent before recycling and this leads to an equal time for all cycles.

```
1234567890 sent in 17716 us
1234567890 sent in 17712 us
1234567890 sent in 17716 us
1234567890 sent in 17712 us
1234567890 sent in 17716 us
```

5.3 Receiving data from the PC

5.3.1 Do it the simple way with blocking read: SerialRead1.ino

The first example employs blocking read obtained by using `Serial.readBytesUntil`. By using this procedure you let the system take care of all the burden of receiving end queueing the characters until the terminator (in this case newline) arrives.

The first part of the program is dedicated to the initialization and to the allocation of the buffer which is used to retain the characters. Nothing special here except that we set the serial line timeout (`Serial.setTimeout(200)`) to a value that allows the PC to send at least 20 characters at 9600 bits per second.

```
//SerialRead1: blocking read

const int BUFFER_SIZE=10;
//allocate space for chars+1
char commandBuffer[BUFFER_SIZE+1];
char welcome[]={"Send me a terminated line!"};

//use functions when we call them
//more than once in the code
void clearBuffer(){
    for (int i=0;i<BUFFER_SIZE+1;i++)
      commandBuffer[i]=(char)0;
    bufferPos=0;
}
void setup() {
  //clean the input buffer
  clearBuffer();
  // initialize comm:
  Serial.begin(9600);
  //print out a greeting message
  Serial.println(welcome);
  Serial.setTimeout(200);
}
```

Data is received in the main loop. We use `Serial.available()` to trigger the read operation, i.e. the loop runs normally until a character is received. Then the `Serial.readBytesUntil` waits for the characters until a newline is received, `BUFFER_SIZE` are received or the timeout set before is elapsed. During this receiving time the microprocessor is practically locked down.

```
void loop() {
  //if nothing in the line loop...
  if (Serial.available()){
    // got at least one char
    //wait for message
    //THIS BLOCKS!!!!!
    int n=Serial.readBytesUntil('\n', commandBuffer,
     BUFFER_SIZE);
    switch(n){
     case BUFFER_SIZE: //too long
       Serial.println("Command too long!");
       clearBuffer();
       //beware other character might arrive
       //that are not skipped
       break;
     default:
       Serial.print("You sent me: ");
       Serial.println(commandBuffer);
       clearBuffer();
    }//of switch
  }//of available
}//of loop
```

5.3.2 Reading char by char to avoid blocking: SerialRead2.ino

If the blocked time is not tolerable, the solution is to manually manage the reading from the serial line avoiding the use of
`Serial.readBytesUntil`.

In this case when the `Serial.available()` detects a character, we read it with `Serial.read()`. The reading is not blocking since we know the character is available. Then the program checks to see if the character is the terminator and if the case it queues it into the buffer. Of course we have to pay attention to the number of characters to avoid a buffer overflow

```
//SerialRead2: char mode in loop.

const int BUFFER_SIZE=10;
//allocate space for chars+1
```

```
char commandBuffer[BUFFER_SIZE+1];
//we need to track the number of characters
int bufferPos=0; //position in the buffer
......
......

void loop() {
  if (Serial.available()){
    // read the character
    char inChar = Serial.read();
    if (inChar=='\n'){
      Serial.print("You sent me: ");
      Serial.println(commandBuffer);
      clearBuffer();
    }else{
      //not a \n
      //if possible queue the chars
      if (bufferPos<BUFFER_SIZE-1){
        commandBuffer[bufferPos++]=inChar;
      }else{
        Serial.print("Message too long!\n");
        //clear our buffer
        clearBuffer();
        //beware: other characters may arrive
        //and can refill the buffer!
      }
    }
  }//of available
}//of loop
```

5.3.3 Using interrupts: SerialRead3.ino

The previous example performs better than the blocking one, but still waste a lot of power continuously polling the serial buffer to see if a character is arrived. All this burden can be avoided by switching to an interrupt mode approach. This way, an interrupt routine is linked to the serial line and is automatically awaken when a character arrives. In the remaining time the microprocessor is free to do other things. Linking a routine to the serial line for Arduino is extremely easy: if a routine is present that is named `serialEvent()` the compiler/linker automatically link it to the serial line. Such routine can implement all the code we already discussed in previous example

```
// SerialRead3: via interrupt
// using serialEvent

..... as in SerialRead2

//if a procedure 'serialEvent' is present
//it is linked to the serial line interrupt
void serialEvent(){
  char inChar = Serial.read();
  if (inChar=='\n'){
    Serial.print("You sent me: ");
    Serial.println(commandBuffer);
    clearBuffer();
  }else{
    //not a \n
    //if possible queue the chars
    if (bufferPos<BUFFER_SIZE-1){
      commandBuffer[bufferPos++]=inChar;
    }else{
      Serial.print("Message too long!\n");
      //clear our buffer
      clearBuffer();
      //beware: other characters may arrive
      //and can refill the buffer!
    }
  }
}

//now the loop is completely free !
void loop() {
  ;
}//of loop
```

Chapter 6

Basic use of the ADC

In section ReadAnalogSerial we saw a very basic use of the Analog to Digital Converter contained in the AT microprocessor which is the basis of the Arduino board. In this chapter we will see some more complex applications which address sync measurement and measurements of more than one sample.

6.1 Taking a measurement on request: MeasureOnDemand.ino

If we want to control the Arduino board making it starting a measurement on demand, we need to implement some form of serial reading to receive commands from the PC. The simplest approach is to use a single character software protocol. In this case, if the character is an uppercase 'M', Arduino takes a measurement and sends it back via serial line. Any other character is rejected. Note the clause on character new-line: if the PC (as it often does) sends the newline after a character, the Arduino program does not complain.

```
// These constants won't change.
char welcome[]=
   {"Measuring on Demand\n"};

void setup() {
  Serial.begin(9600);
  //print out a greeting message
  Serial.println(welcome);
}


void loop() {
  if (Serial.available()){  // wait for input
    char inChar = Serial.read(); // get it
    if (inChar=='M'){
      int sensorValue = analogRead(A0);
      //convert to float using nominal values
      float result=sensorValue*5.0/1023.0;
      // print the results to the serial monitor:
      Serial.print("Measured value is: ");
      Serial.print(result);
      Serial.println(" V");
     else if (inChar=='\n'){
       ; //avoid messages on newline
     }else{
       Serial.println("Command not recognized");
       Serial.println(welcome);
     }//of char
  }//of serial.available
}//of loop
```

Some remarks before going on:

- As we discussed in the section devoted to the serial line, this approach is not the only you can use to read the serial line. If you need to perform other concurrent operations a different approach might be used (e.g. interrupt).
- The `Serial` class we are using for sending back the results is only a minimal implementation. As an example, without specific precautions it prints `float` only with two decimal digit. This is far less than the resolution we have. To solve this problem you can either use the form `Serial.print(floatValue, numberOfDigits` or try multiplying the result for 1000 (i.e. computing the measured value in millivolts).
- For Arduino environment `float` and `double` are exactly the same (even though both keywords are recognized by the compiler).

6.2 Multiple measurements: MultiMeasureOnDemand.ino

This example uses Arduino for measuring and arbitrary number of samples. The samples are not stored inside Arduino, so that there are no memory problems, but the measurement timing is slow, since we loose the time required to send data over the serial line. More details on this subject in next chapter.

Since we have to deal with messages more than one character long, we need a buffer for queueing the characters. This implementation does not use the `Serial.readBytesUntil` to avoid blocking.

```
// We need to buffer the chars.
const int BUFFER_SIZE=10;
//allocate space for chars+1
char commandBuffer[BUFFER_SIZE+1];
int bufferPos=0; //position in the buffer
```

The buffer is managed in the loop function. Of course, since we cannot rely on bound checks provided by the compiler, we have to do everything by ourselves.

The code is rather straightforward: the loop function waits for a character coming from the serial line. Then it checks if it is a terminator (in our case a newline); if it is not, the character is queued to the buffer, otherwise the buffer is decoded and possibly the command executed.

```
void loop() {
  if (Serial.available()){ // wait for input
    char inChar = Serial.read(); // get it
    if (inChar=='\n'){
    //the line is complete decode it
      if (commandBuffer[0]=='M'){
       //the command is in the form M xxxx
       //we need xxxx so skip the first char
       //WARNING no checks!
       int n=atoi(commandBuffer+1);
       Serial.print("Measuring ");
       Serial.print(n);
       Serial.println(" values");
       //save the time
       unsigned long startTime=micros();
       for (int i=0;i<n;i++){
         //save the time
         unsigned long runningTime=micros();
         //compute the difference with previous meas
         unsigned long delta=runningTime-startTime;
         //update the start time for next cycle
         startTime=runningTime;
         int sensorValue = analogRead(A0);
         Serial.print("Elapsed ");
         Serial.print(delta);

         Serial.print(" Code: ");
         Serial.println(sensorValue);
       }
       //remember to clear the buffer
       clearBuffer();
     }else{
       badCommandMessage();
     }//of 'M'
    }else{
      //not a \n
      //if possible queue the chars
      if (bufferPos<BUFFER_SIZE-1){
        commandBuffer[bufferPos++]=inChar;
      }else{
        //command too long
        badCommandMessage();
      }
    }
  }//of available
}//of loop
```

Some remarks:

- We do not know how many characters the PC is sending so we have to check every time the position in the buffer to avoid a catastrophic buffer overflow.
- The only recognized command starts with uppercase 'M'. The decoding is trivial and employs `atoi` function. Please note that in most cases other more powerful keywords such as `sscanf`, even though recognized, give unpredictable results due to the minimal implementation, so it is better to avoid using them.
- The buffer must be cleared before using it and after any (either successful or not successful) received command. Since we need to perform this operation in different location of the code, it is better to put the cleaning code into a function. See below.

```
//use functions when we call them
//more than once in the code
void clearBuffer(){
    for (int i=0;i<BUFFER_SIZE+1;i++)
      commandBuffer[i]=(char)0;
    bufferPos=0;
}
void badCommandMessage(){
  Serial.print("Command not recognized ");
  Serial.println(commandBuffer);
  Serial.println(welcome);
  clearBuffer();
}
```

When you run the program you get a apparently surprising result:

```
This is MultiMeasuring on Demand
Available commands are:
M xx: measure xx values
Measuring 10 values
Elapsed 4 Code: 254
Elapsed 500 Code: 253
Elapsed 616 Code: 252
Elapsed 22528 Code: 251
Elapsed 26000 Code: 261
Elapsed 26000 Code: 262
Elapsed 26000 Code: 251
Elapsed 26000 Code: 256
Elapsed 26000 Code: 263
Elapsed 26000 Code: 253
```

As discussed in the serial line section, Arduino takes something like $4\mu s$ to enter the loop a start the first measurement. Then it takes about $600\mu s$ to perform the first two measurements since the results are queued into the serial line buffer, but later it takes about $26ms$ for each measurement since it has to send about 25 characters (e. g. `Elapsed 22528 Code: 251\n`) and this takes about $25ms$.

6.3 Measuring more than one channel and changing ADC reference: MultiChannelMeasure.ino

Measuring more than one channel simply requires changing the channel number in `analogRead`. In this case, instead of using the mnemonic Arduino provides (i.e. `A0... A5`, we can use a simple integer ranging zero to five;

```
int sensorValue = analogRead(n);
```

Many AT microprocessors have more than 6 channels (e.g. channels for internal temperature sensors), but these channels cannot be directly accessed with this function. We will see how to access them in the advanced section.

The ADC can work with different references. The actual availability of references must be checked depending on your Arduino board, however at least three values should be always present, which are named

- `DEFAULT` corresponding to the supply voltage (i.e. either about 5 V or 3.3 V depending on the board)
- `INTERNAL` corresponding to an internal reference of 1.1 V
- `EXTERNAL` corresponding to a pin where you can provide a voltage. Be aware that using this feature requires special care, since the microprocessor can be easily damaged if improper programming commands are issued. Refer to the manual for more details.

Changing the reference is easy and involves only calling a function.

```
analogReference(DEFAULT);
analogReference(INTERNAL);
analogReference(EXTERNAL);
```

BEWARE!!! the reference is changed by the ATMega hardware in a way that insures it does not happen during a measurement. When using the Arduino software this turns out in a change which may happen only after a measurement is taken. In addition, when changing from default to internal reference, the ATMega internal circuitry needs some time to settle. To avoid problems you are strongly advised to take a dummy measurement and to wait before going on with the actual measurements, i.e.

```
......
//switch to internal ref
analogReference(INTERNAL);
{//take a measurement
 int  sensorValue = analogRead(A0);
 //wait some time
 delay(10);
 //now we are ready
 .........
 //back to def
analogReference(IDEFAULT);
```

```
{//take a measurement
 int  sensorValue = analogRead(A0);
 //wait some time
 delay(10);
 //now we are ready
```

BEWARE!!! taking measurements by using the power supply as the reference means adding an uncertainty which is connected to the power supply value. Such value may change with time and with the actual PC used for supplying the Arduino board.

Chapter 7

Advanced use of ADC: high speed sampling

7.1 Serial line bottleneck

As we saw in section MultipleMeasurementOnDemand, the time required to send data over the serial line is often the largest time in a measurement operation. In that example it took about 20 ms to send each measurement resulting in an equivalent sampling rate of about 50 Hz. This extremely low rate is the result of two combined effects: the slow serial line speed and the number of characters sent over it; both these parameters can obviously optimized to increase the speed. The described example used a baud rate of 9600, which results in about 1 ms per character.

7.1.1 Increasing the baud rate

The serial line speed can be increased up to by changing the `Serial.begin` command. The maximum speed which can be set depends on a multiplicity of factors, however a value of 115200 is usually always supported by most PC, and this reduces the time required to send a character to about 90 μs. The 115200 value is also the maximum value supported by the Arduino serial monitor, but writing a custom PC program usually allows one to go to higher values such as 230400 and 460800 bit per second. While these higher values can reduce the transmission time down to 25 μs, be advised that you may encounter stability problems using them. Before using these values check they feasibility on your specific configuration.

7.1.2 Optimizing the transmitted code: MultiMeasureBinary.ino

Since the ADC has 10 bits, i.e. figures up to 1023, the number of characters required to send one measurements can be reduced to four. This way the total time required to send on measurement can be reduced to about 350 μs, thus increasing serial line throughput to about 2800 measurements per second. Such a throughput can further be increased is the ASCII coding is replaced by a binary coding. The 10 bits provided by the ADC require 2 bytes and this turns out in a transmission time to about 175 μs at a baud rate of 115200, i.e. to a throughput of about 5700 measurements per second, which can be increased to about 22000 measurements per second at a baud rate of 460800. Of course this choice requires a receiving systems (usually the PC connected to the Arduino board) to be able to manage and decode binary data.

Sending data in a binary fashion requires defining a protocol to handle the data transmission since it is not possible to rely on a special character acting as a marker to highlight the transmission end. Among the different possibilities, one can employ the solution defined in the IEEE488.2 protocol which is rather flexible and efficient. The protocol allows for arbitrary length transmission prepending the data with an header that allow the receiver to decode the data. The header starts with character `#` followed by one ASCII digit. Such digit tells the receiver the number of subsequent ASCII digits that actually contain the binary message length. After the header the binary block is transmitted and at the end a newline character is used to close the message. Below some examples of messages

```
#19..9bytes..\n
#3100.... 100 bytes ....\n
#512345.... 12345 bytes....\n
```

Sending this way requires generating the header and actually sending the binary data. The header creation is simple

```
.... send n measurements
  //we will send back
  // n*2 bytes
  //make a string containing 2*n
  String s=String(2*n);
  //get the string length
  int sl=s.length();
  //output the header IEEE488.2 style
  Serial.print("#");
  Serial.print(sl);
  Serial.print(s);
  Serial.flush();
```

Since the `Serial.write` command is not able to send anything but a byte or a block of bytes, we have to represent the `int` which contains the measurement result in forms of a byte buffer. The operation can be done by passing the address of the variable and specifying a length of 2 bytes

```
  int sensorValue = analogRead(A0);
  //Serial.write(n) sends only one byte
  //regardless of the type. We need
  //to pass it as a buffer of bytes
  //this can be done by passing the
  //address of int
  Serial.write((byte*)&sensorValue, 2);
```

7.2 Direct register manipulation

Many of the techniques described in the following sections require direct manipulation of the AtMega registers either to obtain specific behaviors or to get rid of the overhead connected to the Arduino high level functions (e.g. `analogRead`).

The Arduino environment provides mnemonics for almost all registers (e.g. `ADCSRA`) and bits within registers (e.g. `ADEN`). When using bit mnemonics keep in mind that they refer to the number of the bit not to its binary value. (e.g. `ADEN`, the bit that enables the ADC is the seventh bit so it is defined as 7). The mnemonics can be directly used by employing the following syntax, which is based on the `read-modify-write` approach.

- To set a specific bit: `ADCSRA |= (1 << ADEN)`. This equals to a logical or with one shifted to the correct position
- To reset a specific bit: `ADCSRA &= ~(1 << ADEN)`. This equals to a logical and with the complement of one shifted to the correct position.

In addition, some registers are 16 bits wide, but the AtMega328 has an 8 bit bus. The 16 bit registers therefore have to be read/write according to specific sequence.

7.3 Speeding up the ADC converter

The Arduino boards contain an ATmega microcontroller which is equipped with a 10 bit successive approximation ADC whose clock is obtained by scaling the 16 MHz clock. The ADC conversion requires 13 clocks cycles (i.e. 10 clocks cycles for the ten bits plus 3 clocks cycles for starting and loading data). The master clock frequency can be divided in powers of 2 in the range of 2 to 128.

The actual ADC clock frequency therefore can be theoretically changed in the range of 125 kHz to 8 MHz corresponding to conversion intervals in the range of 1.6 μs to 104 μs, however the ATMega datasheet warn that the ADC accuracy is retained only for clocks in the range of 50 kHz to 200 kHz. The only value inside this range is 125 kHz, i.e. the value selected by default by the Arduino board. By using this value the theoretical sampling rate is slightly below 10 kHz. As discussed in previous section, the maximum throughput achievable by the serial line at the safe baud rate 115200 is of about 5700 measurements per second therefore, as anticipated, the continuous measurement speed in this case is limited by the serial line bottleneck.

Anyway, if you wish to increase the ADC speed you may manually change the scaling factor by directly manipulating the ADC register `ADCSRA` that manages not only the scaling, but also many other ADC features (enable, start conversion,... refer to the ATMega datasheet for further details) so, when accessing it you must be sure not to alter the bits not specifically connected to the scaling. The scaling bits are the lower three (`ADPS0, ADPS1 ADPS2`). The scaling is 2^N where N is the number set in the register, therefore scaling up to 128 can be achieved. There is an exception to this rule: the scaling corresponding to the code zero instead of 1 is actually 2. The following function shows how to set the scaling.

```
//scaler=requested scaling 2-128
//abnormal values result in
//scaling=128
//the function returns the
//conversion time in us
float setADCScaler(int scaler){
  float clock=16; //MHz
  ADCSRA &=0xF8; //clear lower 3 bits
  switch(scaler){
    case 2:
      ADCSRA |=1;
      return 13*2/clock;
      break;
    case 4:
      ADCSRA |=2;
      return 13*4/clock;
      break;
    case 8:
      ADCSRA |=3;
      return 13*8/clock;
      break;
    case 16:
      ADCSRA |=4;
      return 13*16/clock;
      break;
    case 32:
      ADCSRA |=5;
      return 13*32/clock;
      break;
    case 64:
      ADCSRA |=6;
      return 13*64/clock;
      break;
    case 128:
      ADCSRA |=7;
      return 13*128/clock;
      break;
    default:
      ADCSRA |=7;
      return 13*128/clock;
  }
}
```

Be warned that using shorter conversion times may result in severe accuracy reduction: check before use!

In addition, be warned that the first conversion may take 25 clocks, so a good approach is to take a dummy measurement after reset and before using the ADC in a normal way.

7.4 Buffered measurements: MultiMeasureBuffered.ino

As seen in previous section, the Arduino sampling speed limit is determined by the serial line, so if one stores the measurements inside the local memory, it is possible to achieve a faster sampling. The problem in this case is represented by the limited RAM which is available on the Arduino board. Since no additional ram is installed, one can take advantage only of the internal ram, which depends on the microprocessor. The maximum value, which correspond to `ATMega328P` is `2 kbytes`, but be warned that this ram is used also by the operating system to allocate the system stack and any static variable you declare (including the string used for your messages!!!).

Considering as an example a space of $1.5 kbytes$, one can store a maximum of 750 samples, each of 2 bytes.

The amount of free ram can be determined by looking at the kernel locations which manage the heap, i.e. `__heap_start` and `__brkval`. The following short function allocates a local variable (i.e. `v`) and returns the difference between the address of this variable and the end of the available size

```
int freeRam () {
  extern int __heap_start, *__brkval;
  int v;
  if (__brkval==0)
   return (int) &v - (int) &__heap_start;
  else
   return (int) &v - (int) __brkval;

}
```

Once the free ram is determined, you can allocate space for your data vector using the conventional `malloc`, but remember to leave enough space for the on line allocations otherwise your program may stop without advices without recovering possibilities (i.e. you'll have to reset the micro!)

```
  values=(int*)malloc(nMax*sizeof(int));
```

A few comments:

- In principle any space allocated with `malloc` should be released with `free`, but in Arduino this often does not work due to a poor memory management. In general it is better to avoid allocating/deallocating memory since you easily leak memory.
- There is the possibility to store only 8 bits for conversion thus reducing the ADC resolution, in this case you can double the number of samples.

Of course the advantage of a local buffering becomes evident when is it is coupled with a higher ADC speed as explained in previous section. The example `MultiMeasureBuffered2` shows an example of this use. Again, be warned that this solution may severely affect the ADC accuracy.

7.5 Loop based acquisitions

All tricks discussed in previous sections allow one to increase the acquisition speed, however the exact sampling rate cannot be easily determined in advance, since it depends on the microprocessor operations.

The following example highlights the problem. The small program acquires in sequence a series of measurements and the time each cycle is repeated. Then, after the acquisition is over, it sends back all the measurements.

```
startTime=micros();
for (i=0;i<n;i++){
  values[i]= analogRead(A0);
  elapsed[i]=micros()-startTime;
}
endTime=micros();
Serial.print ("Total time: ");
Serial.println(endTime-startTime);
//send back the measurments
//and either the interval
//or the elapsed time
for (i=0;i<n-1;i++){
  //Serial.print(elapsed[i]);
  Serial.print(elapsed[i+1]-elapsed[i]);
  Serial.print(",");
  Serial.println(values[i]);
}
```

Below an example of acquired data. The ADC input is floating therefore the values are meaningless, however the times are interesting. The ADC in this case is left with the default scaling i.e. 128 and this results in an acquisition time of about 104 μs. The interval between the acquisitions is instead 120 ± 8 μs, i.e. we have an average overhead of about 16 μs and a jitter of about 8 μs.

```
Total time: 1292
120,456
120,455
120,454
120,454
120,453
120,453
128,452
112,452
120,451
```

The total time for 10 measurements in this example is 1292 μs i.e. 92 μs more than the acquisition time due to the loop overhead. Unfortunately, due to the way the microprocessor works, it is really difficult to preview the actual sampling rate so that the only safe way is to monitor the time, but this greatly reduce the available space for the samples.

In addition, by using this 'loop based' approach it is extremely difficult to modify the time interval between the acquisitions to match a specific desired value. The alternative is to use an 'interrupt based' solution, i.e. to use one of the hardware counters which are available inside the ATMega microcontrollers to fire a measurement at fixed time intervals as explained in the following chapter.

Chapter 8

Timers and Interrupts for Timed acquisitions

BEWARE
This chapter describes how to use the Arduino timers by directly manipulating the microcontroller registers. This option is always available regardless of the Arduino IDE level you are using. Since the Arduino environment is based on on a set of c++ classes, there is a possibility you can find in your environment specific classes devoted to timer management (such as `SimpleTimer`, `IntervalTimer`, etc..).
Using these classes may be easier than manipulating the registers, but your code this way may be not portable to a different development environment.
Unless you know what you are doing avoid using these classes whenever possible.

A way to obtain a fixed pacing for the acquisitions is to use some forms of hardware counters to measure the elapsing time, firing the ADC at fixed intervals.

AtMega processors have at least three timers named `Timer0`, which has a length of 8 bit, `Timer1`,which has a length of 16 bit, and `Timer2`, which has a length of 8 bit.

`Timer0` is used by the Arduino kernel to keep track of the time (i.e. for functions like `micros()` and `millis()`) so it is better not to use it. The other two timers can be used (at least if you do not use specific Arduino libraries). For our examples we will use `Timer1` and `Timer2`.

The approach based on timers and interrupts has advantages and disadvantages which can be summarized as

- pros
  - the interrupt based solution has a minimal processor loading, especially if the sampling rate is low
  - the time interval can be fixed since all the work to define the interval is done in hardware. The only perturbation in the time interval and only in some configurations can be due to the microcode being executed when the interrupt fires, so, also in this case, the jitter should be limited to few microsends, regardless of the interval
- cons

  - using interrupts is difficult, since it involves registering a routine to be fired by the hardware.
  - the interrupt is inherently an asynchronous event, which can happen at any time with respect to the other operation performed by the microprocessor (remember, the processor continuously runs the loop code)
  - the interrupt routine involves some overhead, which may become important if the sampling rate is high.

Neglecting the possibility of using external hardware to pace the ADC, there are basically four ways to have measurements at fixed intervals using timers and interrupts:

- Using a timer in match mode: this approach is very flexible and accurate and permits sampling frequencies of up to 64 kHz (provided that special precaution are taken), since the pacing is completely managed in hardware.
- Using a timer in CTC mode: this approach also is very flexible and accurate and permits sampling frequencies of up to 64 kHz (provided that special precaution are taken), since the pacing is completely managed in hardware. In addition it has the advantage of triggering the ADC in hardware so the processor is used at its best.
- Using a timer in overflow mode: this approach is potentially less accurate then the match mode and can be used only with a limited number of prescalers and lead to a coarse selection of sampling frequencies. Unless one has special needs, there is no reason to prefer this approach instead of the previous one.
- Using the ADC in autotrigger mode: this approach is limited to few sampling rates so there is no reason to use it unless one has special needs.

8.1 Full hardware timing: timer in match mode

A timer can be programmed to count up starting from zero until a specific value is reached. When this happens the timer can fire an interrupt and resets. This way an interrupt can be fired at fixed intervals without manual interventions. If this technique is coupled to the 16 bit timer, it gives a fine control over the interval and allows pacing from few microseconds to up to about 4 s. The following examples therefore refer to the use of `Timer1`, which is the 16 bit timer. Two examples are provided that make use of the simpler `analogRead` (i.e. `MeasureMatchInterrupt.ino`) and that make use of a direct register access (i.e. `MeasureMatchInterruptDirect.ino`)

Using the full hardware timing approach requires:

- Programming the prescaler to select the timer clock by accessing `TCCR1B` register: the timer can be paced starting from the 16 MHz clock divided by a value which can reach 1024.
- Loading the correct code into the match register (e.g. `OCR1A`): the code is the pace interval in clocks
- Registering an interrupt routine to serve the match interrupt (e.g. writing a routine named `ISR(TIMER1_COMPA_vect)`

- Activating the match mode by accessing `TCCR1B` register.
- Activating the interrupt by accessing the `TIMSK1` register

The prescaler is programmed by manipulating the three lower bits of register `TCCR1B`

```
void setTimer1PreScaler(int scaler){
  TCCR1B &= 0xF8; //clear the three lower bits
  switch(scaler){
    case 1:
      TCCR1B |= (1 << CS10);
      break;
    case 8:
      TCCR1B |= (1<<CS11);
      break;
    case 64:
      TCCR1B |= (1 << CS11) | (1 << CS10);
      break;
    case 256:
      TCCR1B |= (1 << CS12) ;
      break;
    case 1024:
      TCCR1B |= (1 << CS12) | (1 << CS10);
      break;
    default:
      TCCR1B |= (1 << CS12) ;
    }
}
```

The code and the match register can be easily computed and loaded by means of a function like:

```
float InitTimer1(float samplingRate){
  //this function returns the actual time interval
  //the clock is 16MHz i.e. 62.5 ns
  //the timer can count up to 65535
  // we have these scalers
  int scalers[]={1,8,64,256,1024};
  //look for the required scaler
  int scalerVal;
  for (scalerVal=0;scalerVal<5;scalerVal++){
        float minFreq=1/(62.5e-9*scalers[scalerVal]*65530);
        if (samplingRate>minFreq)
        break;
  }
  float interval=1/samplingRate;
  //now compute the matching code according to the scaler
  unsigned int code=(unsigned int)
               (interval/(62.5e-9*scalers[scalerVal]));
  //set the scaler
  TCCR1A = 0;
  TCCR1B = 0;
  setTimer1PreScaler(scalers[scalerVal]);
  //load the match code
  OCR1A = code;
  //set the timer in compare mode
  TCCR1B |= (1 << WGM12);   // CTC mode
  //now return the actual freq
  return 1.0/(code*62.5e-9*scalers[scalerVal]);
}
```

Once everything is prepared the interrupt can be enabled/disabled by accessing `TIMSK1` register:

```
void fireTimer1Match(){
 TCNT1  = 0; //clear timer
 //since the timer was already running
 //it reached the match value
 //and an interrupt is pending
 //enabling the mask fires it
 //unless we clear it
 TIFR1 = (1<<OCF1A) ; //clear match flag
 TIMSK1 = (1 << OCIE1A);  // enable only
                         //timer compare interrupt
}
void disableTimer1Int(){
  TIMSK1=0; //disable interrupt
}
```

Eventually the interrupt routine has to take care of the data acquisition and storage. Registering the interrupt routine is done simply naming it as `ISR` with parameter the interrupt vector entry to be substituted. The routine contents depends on the actual requirements, the following fragment as an example reads a certain number of samples and then gives up signaling the acquisition end via a flag.

```
ISR(TIMER1_COMPA_vect)
{
    samples[cnt++] = analogRead(A0);
    if (cnt==NSAMPLES){
      disableTimer1Int();
      done=true;
    }
}
```

Be warned that the way the interrupt routine is written plays an important role in the minimum interval time, i.e. in defining the maximum sampling frequency: if the routine takes more time than the timer interval, the behavior can become not predictable, so care must be exercised to avoid this condition. Specifically remember that the ISR calling requires saving/restoring some registers and this requires $5 - 7$ μs; every operation, even the simple end check, requires time, usually $3 - 4$ μs; the ADC reading via `analogRead` requires waiting the ADC to take the measurement (e.g. about 104 μs at the default scaler) plus the operations connected to the function calling (about 8 μs). The above routine therefore requires about 120 μs to be executed and this limits the maximum reliable sampling rate to about 8 kHz.

The theoretical maximum rate, limited by the ISR overhead and by the end check and without taking any measurements is of about $80 - 100$ kHz so that a careful ISR routine redesign allows one to speed up the measurements.

This rate can be increased by avoiding the use of the `analogRead`, by increasing the ADC clock (at the expense of its accuracy) and by controlling the ADC so that it takes the measurement during the interval between timer interrupts.

Avoiding the use of `analogRead` means manually setting the voltage ref and channel (by wirting `ADMUX`), enabling the ADC and triggering it each time we want a measurement (by writing `ADCSRA`)

```
  int channel=0;
  // Set ADC reference to AVCC and channel
  ADMUX = (1 << REFS0)  | channel;
  ADCSRA |= (1 << ADEN); //enable ADC
  ADCSRA != (1 << ADSC); //trigger ADC
```

Once the the ADC has been triggered, it takes 13 clock to complete the measurement. If necessary one can check the end conversion by looking for a low value at bit 6 (i.e. `ADSC`) of `ADCSRA`. In the following code fragment the check is not performed to save time, since the timing is set by the timer.

```
ISR(TIMER1_COMPA_vect)
{ //read data and trig the ADC for next meas
  samples[cnt++] = ADCL | (ADCH << 8);
  ADCSRA |= (1 << ADSC); //trigger ADC
  if (cnt==NSAMPLES){
    disableTimer1Int();
    done=true;
  }

}
```

By using this approach one can speed up the pacing to about 64 kHz, i.e. to about 15 μs. In this case the ADC clock must be set to 16 (or less) to have a conversion time shorter than 15 μs (in case of prescaler equals to 16, the conversion time is of about 13 μs

8.2 Full hardware timing with CTC mode

Timer 1 can be programmed to pace the acquisition and to automatically start the ADC conversion so that the program is interrupted when the conversion is actually complete.

This behavior can be obtained by using the timer CTC mode and by setting two thresholds. In CTC mode the timer counts on until it reaches the A threshold, then it resets to zero and restarts counting. It is possible to have an interrupt generated each time the A threshold is reached and it is also possible to se another threshold, which is referred to as B threshold which is capable of generating interrupts. If B is set lower than A an interrupt can be generated each timer cycle. This double threshold is designed to allow Arduino di easily manage PWM signal, but is quire useful also for using the ADC. Threshold B (and only threshold B for timer 1) can be routed to the ADC and can starts its conversion. This way the ADC is paced at the intervals set by threshold A, in a way similar to the previous example. If the interrupt generated by the ADC conversion completed event is enabled, it is therefore possible to arrange a very simple acquisition system which leaves the Arduino processor nearly free. A working example is provided (`MeasureADCInterrupt.ino`) that shows the capabilities of the CTC approach.

The set-up requires setting the timer an the ADC as shown in the this code fragment. Note that this fragment does not enable any interrupt. Note also that timer value and prescaler can be can be adjusted to obtain a specific sampling frequency as described in previous section.

```
  TCCR1A = 0x00;     // No outputs on compare math, no PWM mode
  TCCR1B = 1 | (1 << WGM12);
   // Enable CTC mode up to OCR1A no prescaling
  TCCR1C = 0x00;    // Nothing to set into C register

  TCNT1 = 0;            // Clear Counter
  OCR1A = 320;        // Set Threshold A
   THIS IS the sampling time
   (f= 16e6/N) 50kHz with 320, i.e. 20us
  OCR1B = 10;          // Set Threshold B less than A

  //to reach a high sampling rate we need to lower the
  //ADC prescaler
  ADCSRA &= 0xF8; //clear the lower 3 bits
  ADCSRA !=2; //select prescaler to 4 i.e. 4MHz i.e. 3us
  analogRead(A0);     // Dummy Read to make
   Arduino setting the AD
  ADCSRB = 5;       // Select Timer 1 Match B as trigger
                          //do not enable any isr
```

When the acquisition has to be started, the interrupts have to enabled

```
TCNT1 = 0;   // Clear Counter
TIFR1 = (1 << OCIE1A) | (1 << OCIE1B);
// Clear Interrupt flag Match A and B
TIMSK1 = (1 << OCIE1B); // | (1 << OCIE1B);
// Enable Interrupt flag Match A and B
ADCSRA |= (1 << ADATE) | (1 << ADIE);    ;
// now Enable ADC Auto trigger
```

After the last line the system starts acquiring and, when an acquisition is complete, an interrupt is generated. Note that you MUST provide interrupt handlers for A and B thresholds even though you do not use them.

```
ISR(ADC_vect)
{
   .....
   samples[cnt++]=ADC;
   if (cnt==NSAMPLES){
      TIMSK1 &=0xF8; //disable all interrupts
      ADCSRA &=  ~(1 << ADIE); //disable ADC interrupt
      done=true;
   }
}

ISR(TIMER1_COMPA_vect)
  {
      //NEVER HERE !
      Serial.println(" - Interrupt A");
  }

 ISR(TIMER1_COMPB_vect)
  {
      //CANNOT PRINT ANYTHING HERE is the speed is high
      //Serial.println(" - Interrupt B");
  }
```

Please note that in general you must provide a way to stop the acquisition and this must be done inside the interrupt routine.

8.3 ADC Autotrigger: FreeRunADC.ino

BEWARE
In usual conditions using the ADC Autotrigger capability gives you NO ADVANTAGES at the price of a STRONGLY REDUCED FLEXIBILITY
Unless you know what you are doing AVOID using the autrigger solution and prefer one of the full hardware timer solution described in sections 8.1 and 8.2

Another way to have the ADC measuring at fixed intervals is to use the ADC auto-trigger facility. This approach has a limited flexibility, since the sampling rate is simply set by ADC conversion time, i.e. 13 time the ADC clock. Since the ADC clock is set by the prescaler you have only few choices, with minimum speed at slightly less than 10 kHz.

The auto-trigger mode can be set by programming the `ADCSRA` register.

```
void setADCAutoTrig(){
  int channel=0;
  // Set ADC reference to AVCC
  ADMUX = (1 << REFS0)  | channel;
  //ADMUX = (1 << REFS1) | (1 << REFS0)  | channel;
  //set the working mode
  //get the actual prescaler
  int prescaler=ADCSRA &0xf;
  Serial.print("Current prescaler is: ");
  Serial.println(prescaler);
  Serial.flush();
  //start ADC in normal mode
  ADCSRA  = (1 << ADEN) | (1 << ADSC)  | prescaler;
  //wait meas to finish
  delay(10);
  //set auto trigger
  ADCSRA =
     (1 << ADEN) | (1 << ADATE) | (1 << ADIE) | prescaler;

}
```

Once the ADC is in autotrigger mode when it is started
`( ADCSRA |= (1 << ADSC); )`
an interrupt is generated each time the ADC finishes the measurement. The interrupt can be trapped as usual:

```
ISR(ADC_vect) //ADC interrupt
{
  values[cnt++]= ADCL | (ADCH << 8);
  if (cnt==NSAMPLES){
    t2=micros();
    ADCSRA &= ~(1 << ADEN);    //Stop ADC
    done=true;
  }
}
```

8.4 Hardware timing via overflow

Using the timer in overflow mode involves loading the timer with a code and letting it counting up until it overflows. A specific interrupt routine can be fired by this event; it it reloads the counter with the initial code, it is possible to have the timer running continuously. As explained below, this operation may generate time losses, so, if possible, the approach based on the match register is preferred. You may decide to use this approach if you do not want to use the 16-bit `Timer1`.

The following code shows an example of this behavior by using `Timer2`. is an example of selection. The code increases the prescaler until the requested interval can be obtained, then it computes the code to be preloaded into the timer. The function sets two global variables (`scalerVal` and `tcnt2`) which are used by the function which actually set up the timer.

```
float InitTimer2(int muSeconds) {
  TCCR2B = 0x00; // Disable Timer2
  //we can count up to 255
  int scalers[]={1,1,8,32,64,128,256,1024};
  scalerVal=0;
  //find a scaler suitable for
  //reaching the required interval
  //BEWARE: the scan starts with scaler 64
  //to avoid loosing codes
  for (scalerVal=4;scalerVal<7;scalerVal++){
    float maxTime=254.0*scalers[scalerVal]/16;
    if (maxTime>muSeconds)
      break;
  }
  float timeInterval=scalers[scalerVal]/16.0;
  //compute the number of clocks
  //required to reach the interval
  value int n=muSeconds/timeInterval;
  //we count up
  tcnt2=256-n;
  //compute the real interval
  float realInterval=n*timeInterval;
  return realInterval;
}
```

Once the values are computed the timer can be started writing the values into the registers

```
void fireTimer2(){
    TCCR2B = 0x00;      // Disable Timer2
    TCNT2  = tcnt2;      // Preload timer
    TIFR2  = 0x03;        // Clear Timer Overflow Flag
    TIMSK2 = 0x01;      // Enable Overflow Interrupt
    TCCR2A = 0x00;     // Set normal count mode
    TCCR2B = scalerVal;// Load Prescaler
}
```

The interrupt routine must reload the timer if you want to have it regenerating the interrupt after the same time.

```
ISR(TIMER2_OVF_vect) {
  //Reload the timer
  TCNT2 = tcnt2;
  samples[cnt++]= analogRead(A0);
  if (cnt==NSAMPLES){
      done=true;
      TCCR2B = 0x00;
   }
}
```

The problem of this routine is that the timer is restarted by the line `TCNT2 = tcnt2;`, but this line get executed only after the routine calling overhead, i.e. after $2-3$ μs. If this time is longer than the timer clock interval one clock is missed and therefore the interval is no longer correct. This means that faster safe prescaler is 64, which corresponds to 4 μs. With this scaler the frequency selection is rather coarse: faster available values are 62.5 kHz, 50 kHz, 41.66 kHz.

Chapter 9

Accessing special channels: ManualRead.ino

The ATMega microcontrollers such as the ATMega328 are provided with an input MUX with 16 channels. Not all channels are physically connected, but, in addition to the six (or seven) analog input channels there is a channel connected to a temperature sensor, a channel connected to ground for check and a channel connected to the internal 1.1 V reference. All these channels cannot be accessed directly by using the `analogRead` function, which is limited to the first six channels. If one wish to access the other channels, has to write a function that directly access the ADC register `ADMUX` and the other register to start the ADC and get the results;.

The `ADMUX` register has the bits organized in the form `RRxxMMMM` where bits `RR` control the reference, `xx` control the data format and must not be changed, and `MMMM` control the MUX selecting one of 16 channels. Channels 0 to 7 are regular input channels (not always connected to a pin, depending on the selected microcontroller package), channel 8 is connected to the temperature sensor, channel 14 is connected to the 1.1 V reference and channel 15 is connected to ground.

```
//clear bits of mux and ref
//do not touch other bits
 ADMUX  &= 0x30;
 //set ref without affecting
 //other bits
 if (useInternalRef)
  ADMUX |= 0xC0;
 else
  ADMUX |= 0x40;
 //set channel without
 //affecting other bits
 ADMUX |= channel;
```

After the channel selection, the ADC must be managed as explained in previous chapter, i.e. it must be enabled, started and the program must wait for the acquistion to end. Eventually the result must be read. All the operations are controlled via `ADCSRA` register.

```
 //enable ADC writing bit 7
 ADCSRA |= 0x80;
```

```
delay(20); // wait for voltages to become stable.
//start ADC writing bit 6
ADCSRA |= 0x40;
//Wait for bit 6 to return to zero
//signalling EOC
while (ADCSRA & 0x40);
// Reading register "ADCW" takes
//care of how to read ADCL and ADCH.
int r= ADCW;
```

Please observe that this code keeps the microcontroller stuck waiting for the ADC to finish (as `analogRead` does!). If this cannot be tolerated (the ADC may take 200 μs to complete the conversion!), an approach based on interrupts can be used.

The code received by the temperature sensor must eventually be interpreted and converted into a temperature. Refer to the ATMega data sheet for calibration constants. If nominal values are used the conversion is

```
//m received code
//compute the voltage
float v=m*1100.0/1024.0;
//use the constants
//provided by the manual
float zero=290.44;
float scale=0.942;
float t = (v - zero) *scale;
```

Chapter 10

Using digital channels

10.1 Using digitalWrite & digitalRead: the easy slow way

The ATMega microcontrollers such as the ATMega328 are provided with an several digital channels. Accessing these channels within the Arduino environment either as input or as outputs is quite simple and involves setting the channel direction (IN-OUT) with `pinMode` followed by a read with `digitalRead` (for in channels) and `digitalWrite` (for out channels):

```
boolean x=false;
//Arduino have a led connected
//to a specific channel.
//The channel is usually 13
//But sometimes 9
int LED_CHANNEL=9;

void setup(){
  //start serial connection
  Serial.begin(9600);
  //configure pin2 as an input and
  //enable the internal pull-up resistor
  pinMode(2, INPUT_PULLUP);
  //configure pin LED_CHANNEL as output
  pinMode(LED_CHANNEL, OUTPUT);

}

void loop(){
  //read the input channel
  int sensorVal = digitalRead(2);
  Serial.print("Channel 2 is ");
  Serial.print(sensorVal);
  //now toggle the led
  if (x){
    digitalWrite(9, HIGH);
    Serial.println("   LED is high");
  }
  else{
    digitalWrite(9, LOW);
    Serial.println("   LED is low");
  }
   x=!x;
  //wait 1 s
  delay(1000);
}
```

10.2 Digital IO via port manipulation: the fast way

Using `digitalRead` and `digitalWrite` allows one to get an easy access to the channels, but is accomplished by a slow performance. Actually `digitalRead` and `digitalWrite` perform several operations to configure the channels even though such operation could be avoided. If a fast access is required, a direct port manipulation can be performed to avoid these overhead at the expense of a reduced simplicity. Example `DigitalOutput.ino` shows the two procedures.

Arduino Uno has three digital ports which are named B, C, and D. Port D maps pins 0 to 7, Port B maps pins 8 to 13, Port C maps pins 0 to 5. Each port can be accessed via three registers:

- DDRx (read/write) is used to select the port direction
- PORTx (read/write) is used to write digital outputs
- PINx (read only) is used to read digital values

A bit in a port is identified by the corresponding bit in the register, i.e. line 13 (the led line on Arduino Uno) corresponds to bit 5 of port B which can be accessed with code 32 or with (1¡¡5).

Thus in the normal way channel 13 can be accessed as:

```
int OUT_CHANNEL=13;
pinMode(OUT_CHANNEL, OUTPUT);
digitalWrite(OUT_CHANNEL, HIGH);
```

while the fast approach is:

```
int OUT_CHANNEL=13;
byte OUT_SELECTOR=(1<<(OUT_CHANNEL-8));
DDRB |= OUT_SELECTOR; //this changes only the correct bit
PORTB=OUT_SELECTOR;
```

The advantage in terms of execution time is remarkable: the convential approach takes about 4 μs to execute to digital write, while the fast approach takes about 62 ns.

BEWARE: the example `DigitalOutput.ino` employs `for` cycles to show the results: in the case of the fast approach the `for` overhead is much higher than the simple port write!

Chapter 11

Analog Output (via PWM)

Arduino Uno (i.e. the ATMega328) does not have hardware Digital to Analog converters, therefore the board does not have analog outputs. Nevertheless, the ATMega328 has hardware circuits capable of generating Pulse Width Modulated (PWM) signals that can be used to emulate analog outputs. Using the PWM can be performed either by accessing the timer registers or by using `analogWrite(pin,value)`. This function permits to generate a PWM square wave signal on a specific pin, whose mean value is related to `value` according to the equation

$$OutV = PS \cdot \frac{value}{255} \tag{11.1}$$

where PS is the power supply value (i.e. nominally 5 V).

When using `analogWrite` remember that:

- The output pins are only 3, 5, 6, 9, 10, and 11.
- The square wave has a frequency of about 490 Hz except for pins 5 and 6 where it is 960 Hz. The reason of these values lies in the way `analogWrite` is written and in the way the ATMega328 timers are used for timing the programs (i.e. implementing as an example `millis()`).
- The output is a square wave thus it must be filtered to get a DC value. This implies the presence of noise on the output and some limitations in the output changing speed. Remember that the lesser the noise (i.e. the stronger the filter) the slower the output change speed
- The square wave is generated by the digital outputs thus the `ON` value is related to the actual power supply, which often comes from the USB and therefore has a value of about 5 V $\pm$ 10%. Be sure to take this uncertainty into account when designing arduino.
- Once the PWM is started, it remains working until you reuse the pin in other ways and does not consume processing power since everything is in hardware.

Accessing the timer registers allows one to change the PWM frequency usually increasing it. This permits to have outputs that with the same level of noise can change much more rapidly. Use the solution is you need this peculiarity. BEWARE: altering the timers may results in abnormal behaviours so be careful!

Chapter 12

Arduino EEPROM

Arduino Uno (i.e. ATMega328) contains a non-volatile memory space of 1024 bytes composed by an EEPROM, which can be used to store long terms data such as calibration values.

Accessing the EEPROM is easy as the Arduino environment provides specific code for doing that. The code is accessed through the EEPROM class by importing `EEPROM.h` and by using the `.put` and `.get` methods. Both mehods get two parameters, an address and a variable which is read/write for a number of bytes equals to its size.

Your Arduino should have been calibrated and should contain calibration data related to its internal analog reference plus other values.

BEWARE: you are strongly advised NOT to write into the EEPROM as it contains the calibration values.

The data stored into the EEPROM can be read by using a specific structure defined as follows

```
struct CalConstants {
  int year;
  int month;
  int day;
  float vRef;
  float tRef;
  int vCode;
  int tCode;
  int gndCode;
  double xtal;
};
```

The meaning are:

- three field for the calibration date (year, month, day)
- two fields for the calibration constants i.e. the voltage reference and the temperature at which the calibration has been performed
- three values corresponding to three ADC codes

    - `vCode` obtained measuring the reference voltage by using the external VRef (i.e. the power supply when the Arduino was calibrated)
    - `tCode` obtained measuring the temperature sensor by using the external VRef
    - `gndCode` obtained measuring the ground voltage by using the external VRef (usually it is always zero)

- An additional code designed to store the actual clock, not used for now

The following code is an example of how the calibration constants can be read (and possibly write)

```
#include <EEPROM.h>

struct CalConstants {
  int year;
  int month;
  int day;
  float vRef;
  float tRef;
  int vCode;
  int tCode;
  int gndCode;
  double xtal;
};

CalConstants calibration;



void getCalConstants() {
  //read ref voltage
  EEPROM.get(0, calibration);
}

//DO NOT USE THIS FUNCTION !!!!
void putCalConstants() {
  //read ref voltage
  EEPROM.put(0, calibration);
}
```

Chapter 13

Arduino math performance

The ATMega microcontrollers such as the ATMega328 do not embed a floating point unit, thus all the floating point operations must be performed in software. The controllers have a hardware multiplier which is able to perform multiplications of integers by using 2 clocks; this permits simple multiply operations to be performed quite fast, but this feature is limited to 16 bits. Longer operations require longer times.

Assessing the real performance is quite difficult, since the compilers usually perform several optimizations to speed up the code; however a rough estimation can be performed by using codes of the type reported at the end of this chapter. By running this code in a 1.0.1 environment the result is summarized below.

10k integer add	13 ms
10k long add	22 ms
10k float add	95 ms
10k integer mult	18 ms
10k integer div	159 ms
10k long mult	50 ms
10k long div	410 ms
10k float mult	99 ms
10k float div	320 ms
10k float sin	1.04 s

Table 13.1: Time required to perform different types of operations on ATMega 328

```
// Code to measure the timing of different operations
int roundV=0;
unsigned long t1=0,t2=0;
//mark variables as volatile to force
//the compiler to make all computations
volatile int cI,sI,sAuxI;
volatile long cL,sL,sAuxL;
volatile float cF,sF,sAuxF;
void setup() {
  Serial.begin(9600);
  Serial.println("Computing timing\n\n\n\n");
  //be sure the micro has nothing to do before stating
  Serial.flush();
  //be sure the last character is emitted
  delay(10);
  }


void loop() {
  unsigned long delta=0;
  int a=0,b=0,c=0;

  //print out a grreting message
  if (roundV<1){
    delay(10);
    roundV++;
    unsigned long i, j, k;
    cI=1;
    sI=2;
    t1 = micros();
    for (i=0; i<10000L; i++){
      cI+=sI;
    }
    t2 = micros();
    Serial.print("10k int add ");
    Serial.print(t2-t1);
    Serial.print(" us k is ");
    Serial.println(cI);
    Serial.flush();
    delay(10);

    cL=1;
    sL=2;
    t1 = micros();
    for (i=0; i<10000L; i++){
      cL+=sL;
    }
    t2 = micros();
    Serial.print("10k long add ");
    Serial.print(t2-t1);
    Serial.print(" us k is ");
    Serial.println(cL);
    Serial.flush();
    delay(10);


    cF=1;
    sF=2;
    t1 = micros();
```

```
for (i=0; i<10000L; i++){
  cF+=sF;
}
t2 = micros();
Serial.print("10k float add ");
Serial.print(t2-t1);
Serial.print(" us k is ");
Serial.println(cF);
Serial.flush();
delay(10);


sI=3;
sAuxI=123;
t1 = micros();
for (i=0; i<10000L; i++){
  cI=sAuxI*sI;
}
t2 = micros();
Serial.print("10k int mult ");
Serial.print(t2-t1);
Serial.print(" us k is ");
Serial.println(cI);
Serial.flush();
delay(10);


sI=3;
sAuxI=12345;
t1 = micros();
for (i=0; i<10000L; i++){
  cI=sAuxI/sI;
}
t2 = micros();
Serial.print("10k int div ");
Serial.print(t2-t1);
Serial.print(" us k is ");
Serial.println(cI);
Serial.flush();
delay(10);


sL=323L;
sAuxL=123L;
t1 = micros();
for (i=0; i<10000L; i++){
  cL=sAuxL*sL;
}
```

```
t2 = micros();
Serial.print("10k long mult ");
Serial.print(t2-t1);
Serial.print(" us k is ");
Serial.println(cL);
Serial.flush();
delay(10);


sL=33L;
sAuxL=1234567L;
t1 = micros();
for (i=0; i<10000L; i++){
  cL=sAuxL/sL;
}
t2 = micros();
Serial.print("10k long div ");
Serial.print(t2-t1);
Serial.print(" us k is ");
Serial.println(cL);
Serial.flush();
delay(10);


sF=323.345;
sAuxF=123678.234;
t1 = micros();
for (i=0; i<10000L; i++){
  cF=sAuxF*sF;
}
t2 = micros();
Serial.print("10k float mult ");
Serial.print(t2-t1);
Serial.print(" us k is ");
Serial.println(cF);
Serial.flush();
delay(10);


sF=33.2345;
sAuxF=12345.678;
t1 = micros();
for (i=0; i<10000L; i++){
  cF=sAuxF/sF;
}
t2 = micros();
Serial.print("10k float div ");
Serial.print(t2-t1);
```

```
    Serial.print(" us k is ");
    Serial.println(cF);
    Serial.flush();
    delay(10);


    sF=33.2345;
    sAuxF=0.912345;
    t1 = micros();
    for (i=0; i<10000L; i++){
      cF=sin(sAuxF);
    }
    t2 = micros();
    Serial.print("10k float sin ");
    Serial.print(t2-t1);
    Serial.print(" us k is ");
    Serial.println(cF);
    Serial.flush();
    delay(10);
  }
}//of loop
```

Chapter 14

GPL Licence

14.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether free or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

14.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

14.3 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

one line to give the program's name and a brief idea of what it does. Copyright (C)

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.