Chapter

LabView and Connections with Third-Party Hardware

Giuseppe Porzio

Abstract

Data acquisition is a function that plays a fundamental role in the automatic supervision and system control, it combine the system (software and hardware) to the process to be controlled (real world). The field of application starts from research to automation, from industry to home automation, in practice everything that in some way must be performed without human supervision. Data acquisition systems are mainly used to measure physical phenomena such as: temperature, voltage, current, distance and pressure, shock and vibration, and displacement, RPM, angle and discrete events, weight. In order to measure it we need a DAQ, Data AcQuisition System, in this chapter we propose to use a cheap open source hardware: Arduino.

Keywords: Arduino, cheapest hardware, wiring code, LabView code, producer/consumer, SCADA system

1. Introduction

Data acquisition is a function that has a role of fundamental importance in the functions of automatic supervision and control because it relates the system (software and hardware architecture) with the process to be controlled (real world). The field of application ranges from research to automation, from industry to home automation, basically everything that in some way must be performed without human supervision.

Data acquisition systems are primarily used to measure physical phenomena such as: temperature, voltage, current, strain and pressure, shock and vibration, distance and displacement, RPM, angle and discrete events, and weight.

When the engineer is interested in controlling a physical process (light intensity, sound analysis, mass measure, position check, velocity, PID control, etc.) his first problem is to acquire the right information coming from one or more sensors, in some cases we talk about sensor strings or distributed sensors.

The goal is to acquire data that are consistent over time and that correctly describe the shaping of the physical process. All this allows both the correct processing of data and a fast action on the control system through its actuators (motors, LEDs, speakers, etc.).

"Data acquisition" means data exchange in both directions: from the process to the system and vice versa. In all control systems the "heart" of the process is the data acquisition that plays a main role but at the same time it must be accompanied by a simple and intuitive user interface, the HMI-Human Machine Interface. Data acquisition systems are generally referred to by the acronym DAQ (Data AcQuisition).

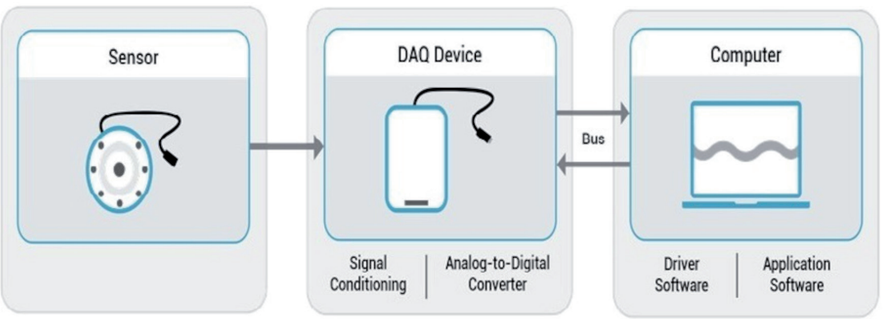

Figure 1.
(Acquisition chain) [1].

Figure 1 shows the electronic chain to acquire an analog signal. The sensor is the device sensitive to the physical feature, the analog-to-digital conversion system, and the computer on which the SW architecture for managing the information is developed. Both feedback and actuators are missing in this figure as they are not the subject of this chapter.

This chapter is designed to be a guide for beginners, programming amateurs and students who wish to approach the world of automation with LabView using low-cost third-party DAQs such as Arduino.

Arduino is a "machine" capable of working in Stand Alone, it can perform simple industrial control tasks.

In a SCADA (Supervisory Control And Data Acquisition) system there is a Master and many Slaves. The Master device carries out the configuration, supervision and control of the slaves. The slave, a local device very close to the process, is equipped with a processor and a system of ports to interface with the sensors and actuators. In this chapter we will write some code to have LabView in the role of Master and Arduino in the role of Slave.

2. Sensor, filtering and multiplexer

We speak about Data Acquisition process, DAQ, when we refer to the process of making measurements of physical phenomena with a PC (tablet, smartphone, workstation, etc). The signals, to be processed, are converted from the analog domain to the digital domain. Only after the digital acquisition we can process the data acquired (recording, visualization, analysis). For this purpose, an A/D (Analog to Digital) subsystem is used to convert the signal.

We report, below, some theoretical hints of the components visible in **Figure 2**.

At the sensor output, the electronic chain includes a "signal conditioning circuit", a multiplexer, the sampling circuit and finally the A/D converter.

The measurement of a physical phenomenon, such as temperature, sound level, vibration of motion oscillatory, or wind speed, begins with a sensor. A sensor is a device that converts the physical phenomenon into a measurable electrical signal.

For example, an elevator gets to the floor through the installation of positioning sensors; a washing machine is equipped with a sensor that measures the rpm of the motor or the water level in the drum; a twilight light; a TV remote control. The classic mercury thermometer is also a type of sensor that is used to measure temperature. In this case, however, the measure is expressed directly on a graduated scale readable by man and not by the machine: we speak in this case of **human readable** type sensor.

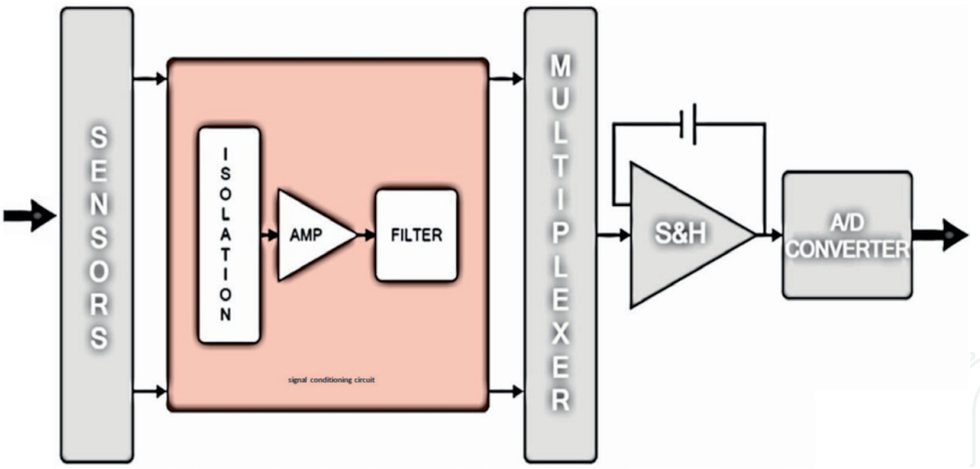

Figure 2.
Detail of the complete acquisition scheme.

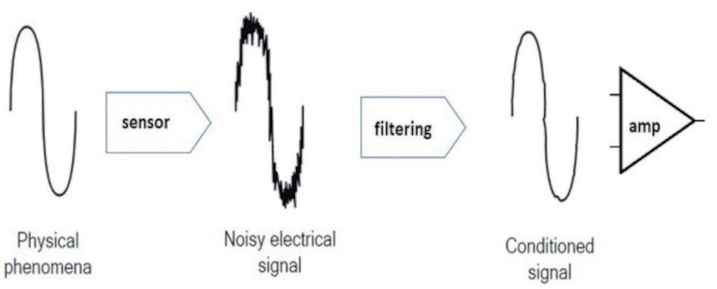

Figure 3.
Signal conditioning, filtering and amplification.

The sensors can produce several kind of electrical outputs such as voltage, current, resistance, or other electrical characteristics modulated from physical phenomenon. When the signal coming from the sensor or from the transmission line is noisy or the ground reference is not at 0 volts (as it should) is preferable to use an isolation system.

In "signal conditioning circuit" we propose a section with electrical isolation that allows the separation of the signal from other electrical sources. This aspect is also essential for the measurement of signals with very small amplitude in which external electrical potentials can affect the quality of the signal considerably, providing incorrect results.

2.1 Signal conditioner

The **signal conditioner** circuits are designed to process the analog signal from the sensors and prepare it to be digitally sampled. The conditioning circuit must linearize the sensor output, eliminate electrical interference that adds to the signal (so-called "noise", as shown in **Figure 3**), and amplify the small signal (mV, μV) to a nominal level, to be easily digitized.

2.2 Multiplexing

Multiplexing, on the other hand, is that part of the circuit that allows us to expand the inputs of our DAQ, thus using a single conversion line on multiple input channels **Figure 4**.

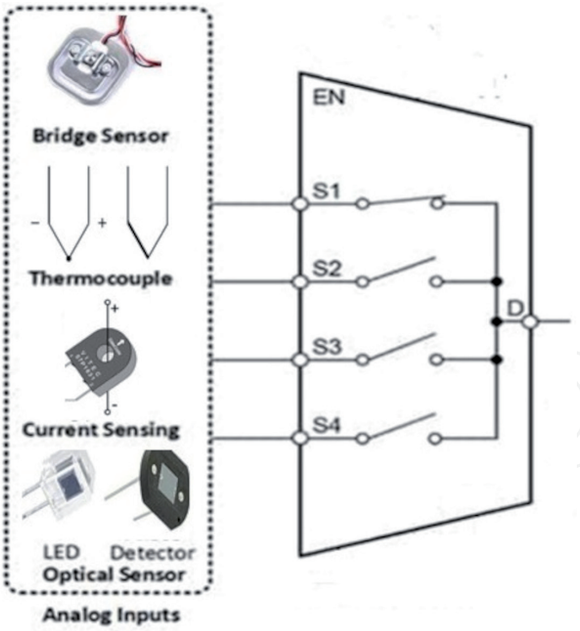

Figure 4.
Example of a 4-input multiplexer.

The multiplexer (commonly called MUX) is a selector of data lines (analog or digital) able to select different input signals: once selected the channel, the corresponding signal is collected and sent on the output line. There are some particularly performing and expensive devices that do not use the MUX but they have a complete acquisition chain for each input.

3. Sampling and coding

3.1 Sample and hold

The S&H system is the circuit part that performs the sampling of the signal (sampling phase). Sampling, in signal theory, is a technique that consists in converting a continuous signal in time into a discrete signal, evaluating its amplitude at regular time intervals. Therefore, considering that the physical quantity attributed to the physical phenomenon varies continuously over time without any interruption, it is necessary to decide with which time interval to interrogate the sensor in order to have meaningful data for our measurement.

From the definition of the sampling interval (T_c) for the scan we derive the sampling rate:

$$f_c = \frac{1}{T_c} \tag{1}$$

The effect of the circuit in **Figure 5** is to store the analog value taken at a given time (sample phase) and keep it constant for as long as it takes the converter to perform the conversion (hold phase).

But how fast should the sampling rate be? Clearly it depends on the phenomenon we are observing. See two examples below:

1. We want to monitor the temperature of a room to stabilize it at a value of T_{set} ± error. Considering the inertia of the room and the radiators it makes sense to acquire the temperature every second i.e. f_c = 1 Hz.

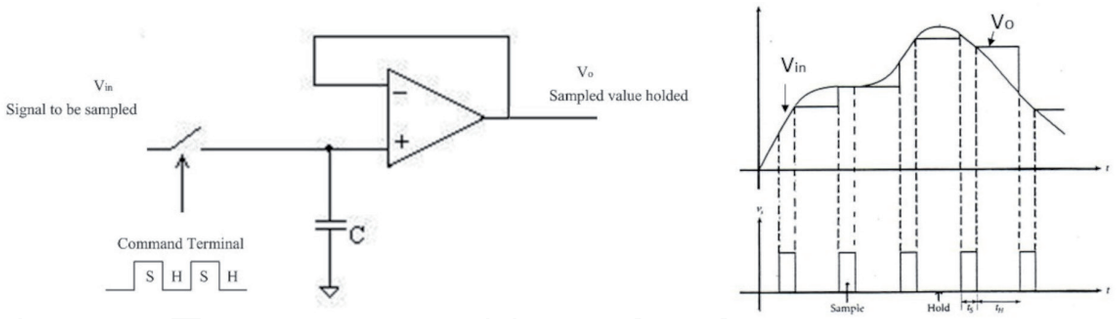

Figure 5.
S&H circuit and example.

Question:

2. How high has to be the sampling rate if we would like to create automatic braking for anti-collision car system? Assume that max velocity, for small/medium sized car, is 180 km/h.

Answer:

$$180 km/h \overset{:3,6}{\Rightarrow} 50 m/s \tag{2}$$

Let us assume that the control system reacts in such a time that the car still travels at maximum for 10 cm (**response time**).

So, if we make some calculations, the time between one reading and the next one of the vision sensor must be less 2 ms. These involves

$$f_c > 500 Hz \tag{3}$$

The proposed cases are at the antipodes: while in the first one we do not have any criticality, in the second one there is a big responsibility due to the need to manage the stop of the car before the impact.

In the real world, according to the mathematician J. Fourier, an analogue signal can be represented by linear combination of sinusoidal functions (called **harmonics**).

The first harmonic, called fundamental, has the same frequency as the input signal, while the following harmonics will have a frequency multiple of the fundamental.

The transition from the analogue to the digital domain, therefore discrete, leads us to acquire one of these harmonics, of course the first one, therefore a suitable sampling frequency will be the key to a good acquisition of the analogue signal, preserving its main characteristic, its frequency.

3.2 Sampling theorem (or Nyquist-Shannon theorem)

In order to have a correct sampling (without loss of information) we must to choose a correct frequency of sample rate. Supposing that the frequency signal (first harmonic) is f_{max} then Nyquist-Shannon Theorem says that the minimum sampling frequency f_s, that preserve the frequency information of a original signal, should be double of the fmax.

$$f_s \geq 2 f_{\max} \tag{4}$$

If, in addition to the frequency, we would like to storage also the shape of the signal we need:

$$f_s \geq 5 f_{\max} \tag{5}$$

The sampling rate is normally expressed in Sample Rate and the unit of measure is number of samples per second [#S/s].

To understand better how the theorem works in the **Figure 6** we report a sequence of acquiring with several sampling rate. The software used is developed for university student's lectures [2].

Figure 6 shows how the sampling frequency acts. We start with a 440 Hz source signal (resonance frequency of a conventional tuning fork) which is visible in the first waveform graph of the sequence. In the following sequences the following sampling frequencies were used: 440 Hz, 600 Hz, 880 Hz and 2200 Hz.

In the first and second cases the fs is not adequate, in fact we have an under-sampling. In the third case we have a result that preserves the frequency of the input signal. Finally, in the last case, we have reconstructed quite faithfully the profile of the original signal.

3.3 Coding

The last sequence in the DAQ chain (**Figure 2**) consists of the operations performed by the A/D converter: quantization and encoding. First we need to introduce the concept of signal dynamics. The dynamics of the signal indicates the maximum excursion of the signal and, therefore, also the maximum and minimum values it can reach, the range of V_{in} (also defined as the Full Scale value):

$$V_{FS} = V_{\max} - V_{\min} \tag{6}$$

(we have assumed a voltage signal)

The input signal, being continuous in time, can by definition take on an infinity of values. As well as the sampler has discretized the signal in time (X axis) we now need another circuit which discretizes the values of the physical quantity which represents the information (Y axis). So the technique is to approximate the value acquired in the sampling phase to a discrete value. The number of discrete values available for these approximations is given by a very simple calculation. If we choose *n* bit to make a digital conversion then the number of discrete value is 2^n.

At this point we have to define the unit of quantization that we call quantum or quantization step, that is the smallest approximation interval that we use to compare the sampled signal to discretize it.

$$Q[V] = \frac{V_{FS}}{2^n} \tag{7}$$

Q is called quantization step. It is possible to assert, at this point, that a higher bit number and a smaller V_{FS} interval implies the greater number of intervals available. This means that the size of the interval will tend to be an extremely small value with increasingly accurate measure.

The simplest coding (commonly used for unipolar signals, i.e. always positive ones), natural binary code (straight binary), consists in making each quantization interval correspond to a progressive binary number, starting from 0 (corresponding to the lowest level) up to 2^n-1.

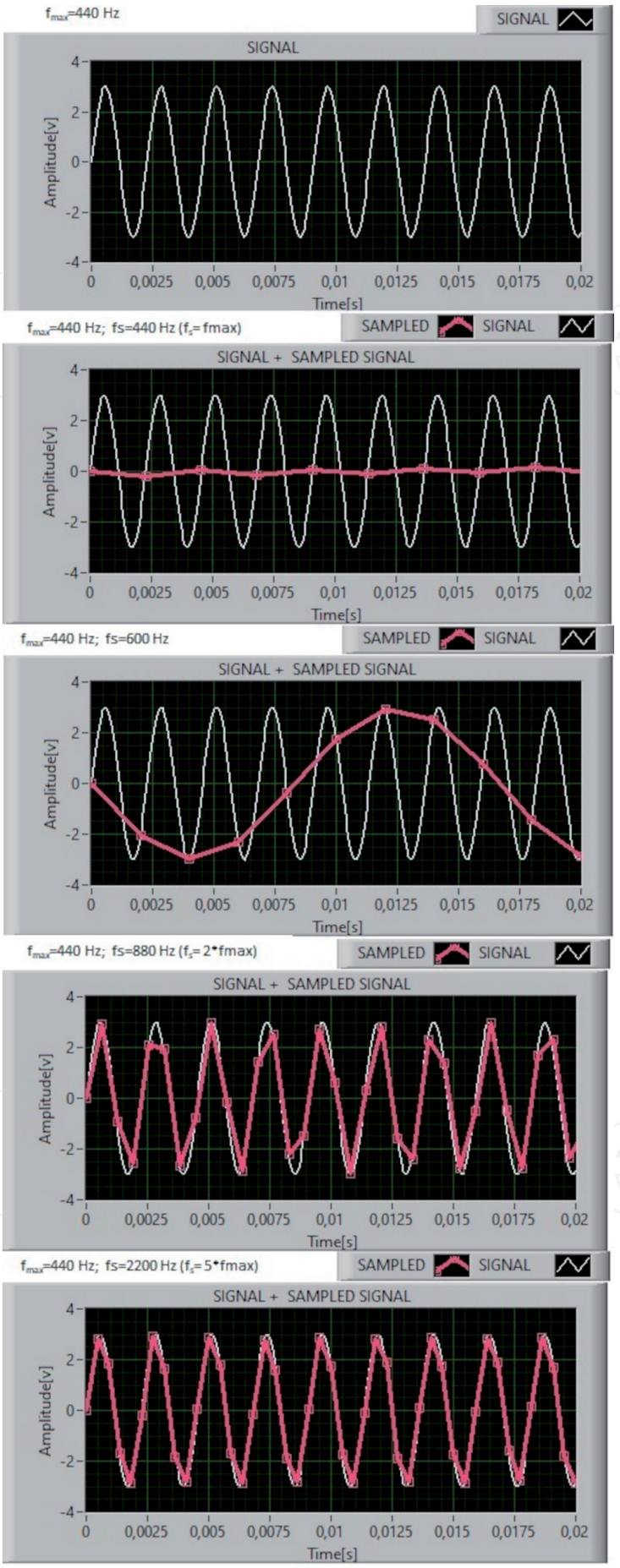

Figure 6.
Signal sampled with different f_s.

In **Figure 7** we show what we have said, on the X-axis we put the intervals between V_{max} and V_{min} and beside them the bit combinations. The first level consists of all bit to zero, so the word 000…00 corresponds to V_{min} while the last level is given by the word with all ones 111….11 i.e. V_{max}.

A different number of resolution bits clearly produces different quantization ranges, some data is shown in **Figure 8**.

Clearly the measurement of Q is affected by error and corresponds precisely to Q/2 and is defined as quantization error.

Recapitulate, in order to perform a correct measurement through a DAQ system, the following points must be satisfied:

1. Prefer a sensor with a linear response and that the maximum and minimum values are compatible with the dynamics of the DAQ;

2. Choose an appropriate sampling rate;

3. Choose an appropriate resolution;

The premises made so far are useful to better understand the code written for the Master unit and the slave unit. In this chapter we propose an cheap and open source prototyping board for which we will write some code to transform it into a DAQ. The proposed board is Arduino UNO rev.3. In the next paragraph, the Arduino technology will be presented [3].

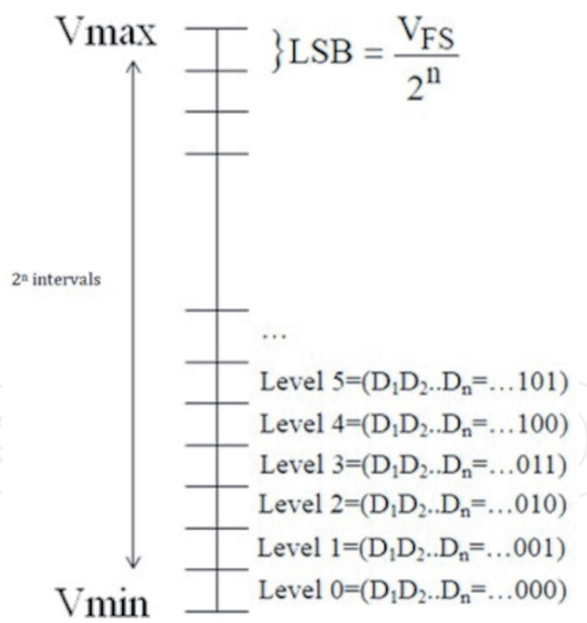

Figure 7.
Quantization and coding.

	12 bit	16 bit	18 bit	24 bit
#level	4.096	65.536	262.144	16.777.216
input range ± 10 V	4,88 mV	305 µV	76,4 µV	1,192 µV

Figure 8.
Resolution example.

DOI: http://dx.doi.org/10.5772/intechopen.96056

4. Arduino UNO rev. 3

Arduino Uno (**Figure 9**) is a microcontroller board (Italian open source project) based on the ATmega328P (resolution @10 bit; input range 0÷5 V). It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz like internal clock (sample rate = ~10 kS/s), a USB high speed connection, a power jack 9 Volt input, an ICSP header, reset button and several states LED like Tx/Rx serial communication.

It contains all interfaces needed to support the microcontroller and its functionality; You can use prototype board with your Uno without worrying about doing something wrong, worst case you can replace chip with a new one and start over again. The Uno board is the first USB Arduino boards, today are available several models of it: with wifi o ethernet, compact or large model, wearable, etc.

Wiring is an open-source programming framework for microcontrollers C/C++ based.

The developer, under conditions of classical use, writes code for Arduino in order to have a "machine" that works in Stand Alone, in **Figure 10** is shown his working scheme, the code runs on Arduino, through the code reads the sensors and produces actions on the physical world. In the next paragraph will be discussed the code to transform Arduino from Master to Slave.

The new role of Arduino will be to be used in LabView environment as a real data acquisition system (**Figure 11**).

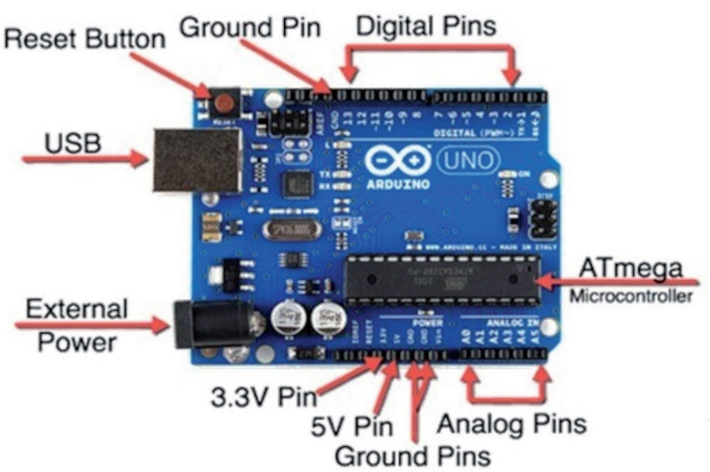

Figure 9.
Arduino UNO rev.3.

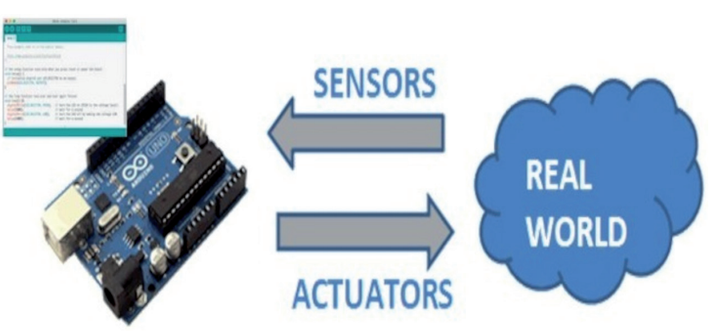

Figure 10.
Arduino-stand alone mode.

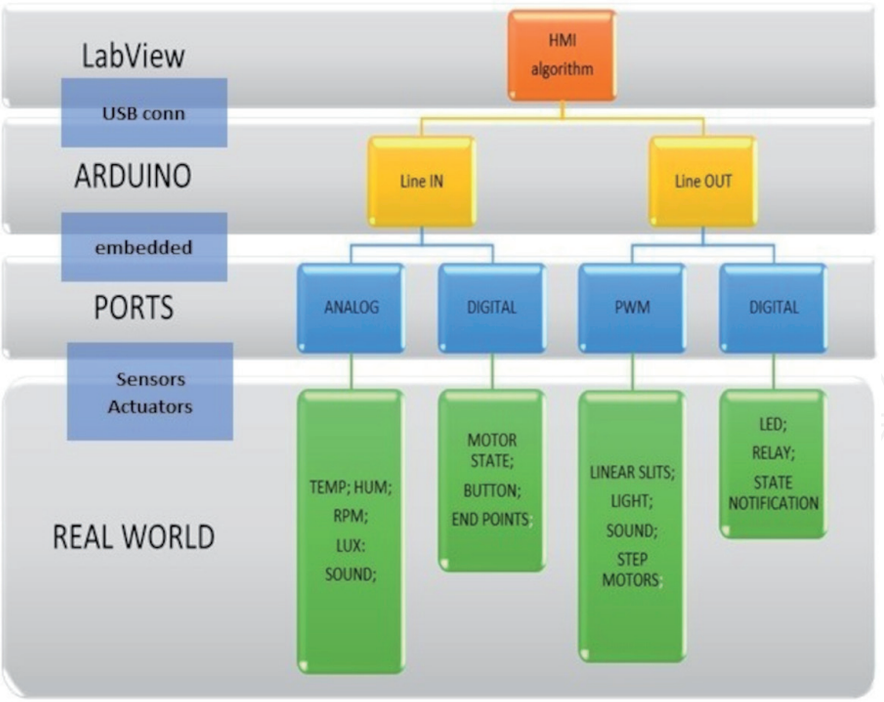

Figure 11.
Control hierarchy with LabView-Arduino.

4.1 From master to slave

Among of programmer "sketch" is the name that Arduino's programmer uses for a program. It's of code written in like C, compiled and, then, uploaded on the board. After it is possible to run on an Arduino board the code. There are two distinct functions available in Arduino sketch: *setup()* and *loop()*.

The *setup()* is called once time only at beginning when the sketch goes in run. It's a correct place to make setup tasks like setting pin modes or initializing libraries.

The *loop()* function is a infinite loop and is heart of most sketches. You need to include your algorithm and functions in it.

Normally in the setup() section there is the sequence of instructions to configure all the Arduino peripherals and features that will be used in the project such as: Analog input, PWM, i2c. In loop(), instead, is written all the control algorithm that will be characterized by an infinite loop.

In this paragraph we propose the development of a code from a different perspective, Arduino will be used as a DAQ system. So inside the setup() there will be a pre-cycle in which the Arduino waits for the USB connection to LabView and waits for the ASCII character sequence to configure the Arduino ports as desired.

The ASCII code, we call op-code from now, to send for configuration are printable characters, so you can always test the Arduino code from any serial terminal or using the serial monitor of the IDE.

For example, to configure the Analog Input channel zero (A0) just send the code "a". Arduino will remain in the setup() section until the master sends the character "z" on the serial which will end the setup cycle to execute the code in the loop().

The code proposes a scenario in which analog inputs A0÷A5, DIO pin2 and pin4 and a PWM channel on pn3 are configurable. Clearly it is possible to extend the "offer" by adding other input or output lines. The complete management of a sensor through Arduino libraries could also be included.

Regarding the sampling time Ts it is possible to define through the ASCII codes A,B,C,D a time delay equal respectively to 100 msec, 10 msec, 1 msec, 500 μsec. If it is omitted the acquisition time is 1000 msec.

```
/* SerialEvent occurs whenever a new data comes in the hardware serial RX. This routine
is run between each time loop() runs, so using delay inside loop can delay response.
Multiple bytes of data may be available. */
void serialEvent() {
  // get the new messagio from master
  String input_from_LV = Serial.readString();
  if (input_from_LV == "R\n"){
   Serial.println("Reset, you have to wait...");
   Reset();
  }
  if (input_from_LV == "D0_ON\n"){
   Serial.println("DIGITAL ZERO_ON");
   digitalWrite(2, HIGH);
  }
  if (input_from_LV == "D0_OFF\n"){
   Serial.println("DIGITAL ZERO_OFF");
   digitalWrite(2, LOW);
  }
  if (input_from_LV == "D1_ON\n"){
   Serial.println("DIGITAL UNO_ON");
   digitalWrite(4, HIGH);
  }
  if (input_from_LV == "D1_OFF\n"){
   Serial.println("DIGITAL UNO_OFF");
   digitalWrite(4, LOW);
  }
  if ((input_from_LV.substring(0,3)) == "PWM"){
   val=input_from_LV.substring(3,6).toFloat();
   analogWrite(3, val);
  }
  Serial.println(input_from_LV);
}
```

```
// blinking function- Called during the channel configuration
void blinking() {
 for (int i=0; i<2;i++){
  // turn the LED on (HIGH is the voltage level)
  digitalWrite(LED_BUILTIN, HIGH);
   // wait for 150 msecond
  delay(150);
 // turn the LED off by making the voltage LOW
  digitalWrite(LED_BUILTIN, LOW);
  delay(150);
}
}
```

Figure 12.
Code- SerialEvent() and blinking().

The code developed in the loop() section collects data from the previously configured input line ports, maps them to the following format #A0#A1#A2#A3#A4#A5$D0$D1 and sends the message continuously to the USB

port. The message will contain as many strings as there are lines configured. In the syntax #Ai (i = 0...5) the value of Ai corresponds to the decimal decoding of the combination of the 10 bits, so there will be 2^n combinations. At value 0 will correspond 0 (zero) Volt and at value 1023 will correspond 5 Volt.

WE suggest to the reader to test own system velocity before to set 500 μsec of sample rate. Usually, for my experience, it is very rare to follow with a LabView (not real time) Loop code that velocity. In case the system is not fast enough, one way of not losing data could be the following: change the Arduino's code to collect the msg (measured value) in a vector of 100 elements and send it to LabView each 50 msec. You can choose different size of vector but you have avoid to saturate the Arduino memory.

The op-code (operation code) we have written does not belong to any standard communication protocol. We have invented a sequence of simple ASCIII strings to be sent over serial. So the Master will have at his disposal a set of instructions, which can be extended by the reader, to change the status of a digital output: D0_ON\n, D0_OFF\n, D1_ON\n, D1_OFF\n.

In order to avoid a slowdown loop() for sensors reading, due at continuous polling on the receipt of messages from the Master, an event-driven solution has been considered.

The reception on the serial line of a request from the Master is triggered by the event generated by the chip that manages the USB communication. When a byte arrives on RX an event is generated and triggered by a software procedure. When this occurs the Master message will be read (**Figure 12**).

In the end we can send a message to set a Analog output by pin3 in PWM mode.

The **Pulse Width Modulation** [4], or PWM, is a powerful technique to control analogic circuits (applied to a load) using a digital signal. It is a type of digital modulation, in particular we speech of pulse width modulation which allows to obtain a variable average voltage depending on the ratio between the duration of the high pulse and the entire period (duty cycle).

In electronics it is used to change the voltage, and therefore the power, on a generic load. For example, to change the speed of a direct current electric motor, to vary the brightness of light bulbs, especially LEDs. A useful duty cycle of 0% indicates a pulse of zero duration, in practice no signal (V_{out} = 0 volts), while a value of 100% indicates that the pulse ends when the next one begins (V_{out} = V_{cc}). To use

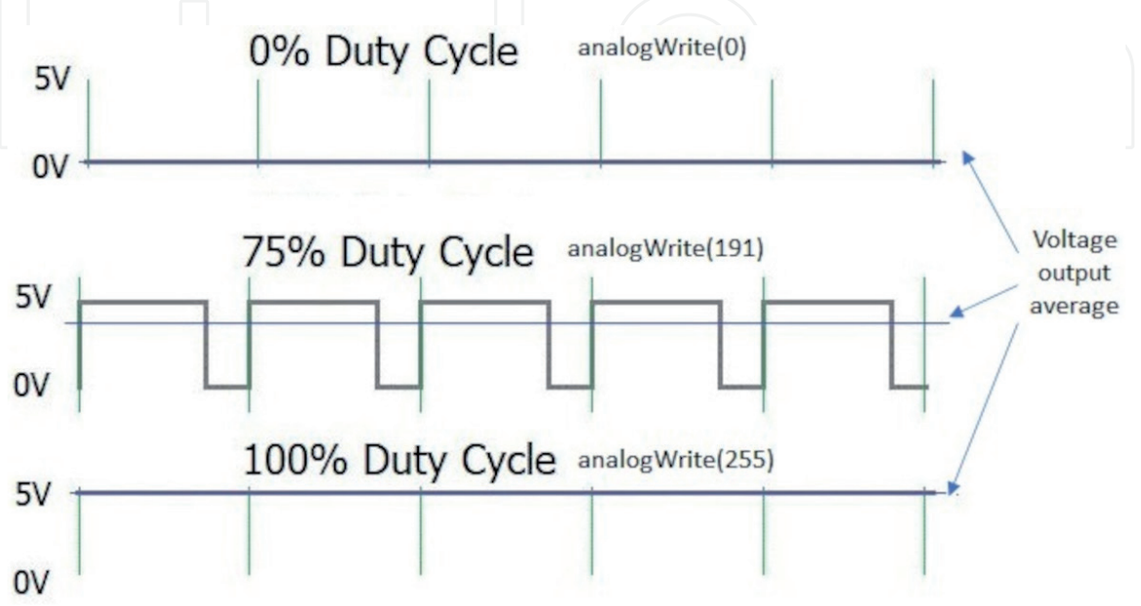

Figure 13.
PWM example.

this technique with Arduino is very simple, with the analogWrite (PIN, VALUE) function it is possible to modulate the work cycle. The PIN corresponds at PWM pins and VALUE is scale from 0 to 255. For example analogWrite (pin, 255) corresponds to a 100% duty cycle and analogWrite (191) is a 75% duty cycle (**Figure 13**).

4.2 Arduino code

In the following boxes (**Figure 15**) we'll show some code that you can use to create a communication Master–Slave from LabView and Arduino [5].

In **Figure 14** it is possible to understand the functionally of declarations reading the comments.

In **Figure 15** is possible to verify the **setup()** code, inside it there are the comments to understand it. The code in **Figure 15** has been conceived to be very static and the expansion is very simple for the novice programmer. The serial speed has been set at $2 * 10^6$ bit/s in order to have the maximum communication speed between Arduino and the Master.

The code written in the "WHILE LOOP" could be redesigned to treat the Arduino channels dynamically. We want to say that configuration strings (like **pinMode(4, OUTPUT)**) can be sent directly from the MASTER unit.

```
// This Code is written frm G. Porzio for chapter of
// "LabVIEW - History, Applications, Current Trends and Perspectives",
// ISBN 978-1-83968-841-6.
//---------------------------------------------------------------------
// Variable Declarations
int opCode = 0;         // incoming serial byte
int Ts=1000;            // Sampling time [msec]; inizialize at 1 second
String output;          // msg to send at serial port with analog value
// each bit corresponds a sensor, the value at 1 means the sensor is present the LSB bit = Analog Sensor in A0
byte Sensors = 0;
// each bit corresponds a Digital line, the value at 1 means that the line is configurated as input.
byte digital_line_OUT=0;
// each bit corresponds a Digital line, the value at 1 means that the line is configurated as output.
byte digital_line_IN=0;
// each bit corresponds a PWD line, the value at 1 means that the line is.
byte Analog_write_PWM_LB=0;
String input_from_LV=""; // String contains the op-code
int val;
void(* Reset)(void) = 0;  // is a code for software RESET
//---------------------------------------------------------------------
```

Figure 14.
CODE-variable declaration.

```
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  // We use it to blink when Arduino recevices the string configuration from LaBView
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, LOW);   // we turn it OFF

  // start serial port at 2Mbit/sec and wait for port to open:
  Serial.begin(2000000);
  Serial.flush();
  while (!Serial) {
    ; // wait for serial port to connect.
  }
  // we send a msg to LabView
  Serial.println("connection estabilized");

  delay(500); //  wait 500 msecond
  /* the following code is used to configure, in this example,
  the analog input A0-A5, the digial I/O (pin 2 and pin 4),
  and PWM line (pin 3) of course it is possible to expand
  it as want as you wan.
  We use in the follow "SWICTH CASE" statement The ASCII code:
  1)a,b,c,d,e,f to configure A0-A5;
  2)g,h to Configure the pin 2 and pin4 like digital in;
  3)i,l to Configure the pin 2 and pin4 like digital out;
  4)m to configure the pin 3 like digital out and
  we'll use in PWM mode;
  5)A,B,C,D to set the Ts (Sample rate) at 100 msec,
  10m sec, 1 m sec, 500 usec;
  6) z to finish the setup().
    */
   while (opCode!=122){ // exit with statement "z"
    // get incoming byte:
     opCode = Serial.read(); //reads the Bytes sequence on USB
     switch (opCode){
       case 97:  // "a"
       Serial.println("channel A configurated");
       bitSet(Sensors,0);
       blinking();
       break;
       case 98:  // "b"
       Serial.println("channel B configurated");
       bitSet(Sensors,1);
       blinking();
       break;
       case 99:  // "c"
       Serial.println("channel C configurated");
       bitSet(Sensors,2);
       blinking();
       break;
       case 100: // "d"
       Serial.println("channel D configurated");
       bitSet(Sensors,3);
       blinking();
       break;
       case 101: // "e"
       Serial.println("channel E configurated");
       bitSet(Sensors,4);
       blinking();
       break;
       case 102: // "f"
       Serial.println("channel F configurated");
       bitSet(Sensors,5);
       blinking();
       break;
```

```
case 103: // "g"
       Serial.println("Digital 2 is input");
       pinMode(2, INPUT_PULLUP);
       bitSet(digital_line_IN,0);
       blinking();
       break;
       case 104: // "h"
       Serial.println("Digital 4 is input");
       pinMode(4, INPUT_PULLUP);
       bitSet(digital_line_IN,1);
       blinking();
       break;
       case 105: // "i"
       Serial.println("Digital 2 is output");
       pinMode(2, OUTPUT);
       bitSet(digital_line_OUT,0);
       blinking();
       break;
       case 106: // "j"
       Serial.println("Digital 4 is output");
       pinMode(4, OUTPUT);
       bitSet(digital_line_OUT,1);
       blinking();
       break;
       case 107: // "k"
       Serial.println("Digital 3 in PWD mode");
       pinMode(3, OUTPUT);
       bitSet(Analog_write_PWM_LB,0);
       blinking();
       break;
       case 65:  // "A"
       Serial.println("set Ts at 100 msec");
       Ts=100;
       blinking();
       break;
       case 66:  // "B"
       Serial.println("set Ts at 10 msec");
       Ts=10;
       blinking();
       break;
       case 67:  // "C"
       Serial.println("set Ts at 1 msec");
       Ts=1;
       blinking();
       break;
       case 68:  // "D"
       Serial.println("set Ts at 500 ucsec");
       Ts=0,5;
       blinking();
       break;
       case -1:  // null
       case 10:  // "\n"
       case 122: // "z"
       break;
       default:
       Serial.println("channel error");
       break;
     }
```

Figure 15.
Code-slave mode setup().

```
Serial.flush();
  while (input_from_LV.substring(0,5)) != "EXIT"){
   String input_from_LV = Serial.readString();
  // Master send, for example, pinMode 5 OUTPUT, we have to parse the phrase
   if (input_from_LV.substring(0,7)) == "pinMode"){
  pinMode(input_from_LV.substring(8,9), input_from_LV.substring(10,16))
  // put your code here

   blinking();
   }
```

Figure 16.
Example code for dynamic configuration.

```
/* In the loop() code we use the bitRead to Know if the channel
 *  is configurated  in the setup() statement
 */
void loop() {
output= String ("");
// Serial.println(Sensors);
 if (bitRead(Sensors, 0)==1){
    output+="#";
    output+= String (analogRead(A0));
 }
 if (bitRead(Sensors, 1)==1){
    output+="#";
    output+= String (analogRead(A1));
 }
  if (bitRead(Sensors, 2)==1){
    output+="#";
    output+= String (analogRead(A2));
 }
 if (bitRead(Sensors, 3)==1){
    output+="#";
    output+= String (analogRead(A3));
 }
 if (bitRead(Sensors, 4)==1){
    output+="#";
    output+= String (analogRead(A4));
 }
 if (bitRead(Sensors, 5)==1){
    output+="#";
    output+= String (analogRead(A5));
 }
 if (bitRead(digital_line_IN, 0)==1){
    output+="&";
    output+= String (digitalRead(2));
 }
 if (bitRead(digital_line_IN, 1)==1){
    output+="&";
    output+= String (digitalRead(4));
 }
Serial.println(output);

delay(Ts);

}
```

The "delay()" behaviour is like the sample rate

Figure 17.
Loop ( ) code.

Figure 16 shows a small piece of code that is a good starting point for completing the dynamic channel configuration.

At this point we show the code about the blinking procedure and the Serial events procedure, respectively both in **Figure 12**.

In the end we report the loop() code, **Figure 17**. The code is very simple, the final message is made-up by concatenating the message in each "if" statement.

5. LabView architecture

In this section we show you the architecture that we use to run LabView code in Mater mode. We have chosen the *Producer/Consumer Architecture* [6].

The Producer/Consumer design pattern (**Figure 18**) is based on the Master/Slave pattern, and is geared towards enhanced data sharing between multiple loops running at different rates. The Producer/Consumer pattern is commonly used when acquiring multiple sets of data to be processed in order. Suppose you want to write an application that accepts data while processing them in the order they were received. Because queuing up (producing) this data is much faster than the actual processing (consuming), the Producer/Consumer design pattern is best suited for this application. In our project we can set a high sample rate (up to f_s = 10 kHz) so in this can we can occur in a data loss case. With Producer/Consumer is sure that we are implementing a data lossless LabView architecture. But we have considerated also an architecture *Event-Driven* to catch the write instance from LabView vs. Arduino only if asked from the operator.

In **Figure 19** we show the front panel developed in LabView [7].

5.1 Front panel

On the left side of front panel are present a several controls to configure the DAQ (Arduino in Slave mode) in according with previous paragraphs. Instead on the right side we found a control to set the PWM value (analog output) and the digital output state. We use Waveform chart like oscilloscope to view the six signal.

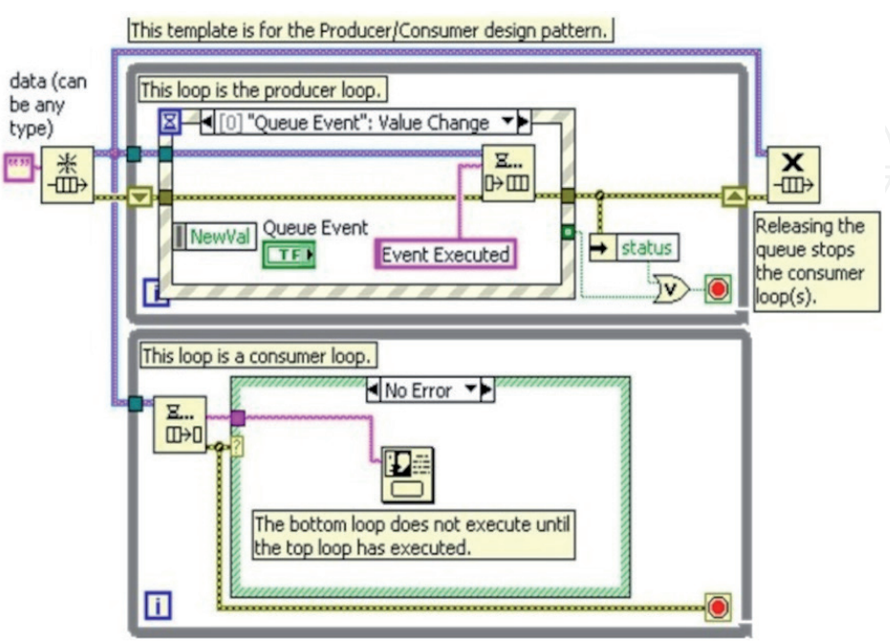

Figure 18.
Event structure in producer/consumer design pattern.

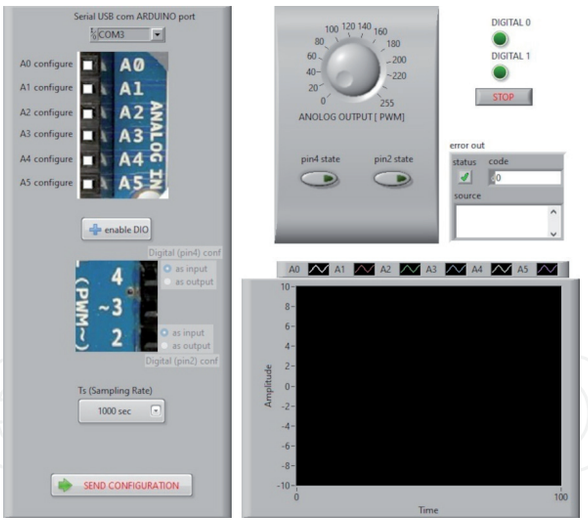

Figure 19.
LabView front panel.

After defined the configuration you have to send a message at the serial VISA communication by the pressing of "SEND CONFIGURATION" button.

After that the cycle Producer/Consumer starts and the sensor reading is shown on the Waveform Chart.

5.2 Block diagram

Inside the LabView Code (block diagram) there are three nodes. The first one composed by "While Loop" (**Figure 20a**) that waiting for user's hardware configuration. In this Loop we create a Boolean array with all hardware instance, at the end of the configuration the user pushes the button and send the array to subvi "open and configure.vi" (second node). It makes a rights sequence of op-code, open the Serial Port communication (in this case **com3**) and send it at Arduino (**Figure 20b**). During this phase you can observe the blinking LED on Arduino board, this means that the configuration message has correctly reached Arduino and it is processing the op-code.

From **Figure 20b** is possible to verify that the serial port velocity is 2Mbps.

In this way the communication between Master and Slave does not make interference with acquiring. In fact one character, in ASCII encoding (1 byte), from Arduino to LabView is sent in 4 μsec. If we would configure all analog inputs (6) and all digital inputs (14) the maximum number of characters would be = 6 prefixes (#) + 6*4 (digits of value among 0÷1023) + 14 prefixes (&) + 14 digital states = 58 bytes.

Maximum time to transmit the entire message is 58 Byte * 4 μsec = 232 μsec. This time is half of the minimum sampling time set in the code, that is 500 μsec. You could also reach 100 μsec of sampling rate that corresponds to 10 kHz of sampling frequency, in this case you have to merge the bits of the digital input, so it is possible to save 26 bytes but it is not enough. We have to modify the syntax of sending analog input values to reach at least 80 μsec of transmission time. This modification to the Arduino code we leave to the reader as an exercise.

In last one node, **Figure 21**, we can see the Producer Loop and the Consumer Loop. Both are connected by the queue, in queue process we read the Arduino's

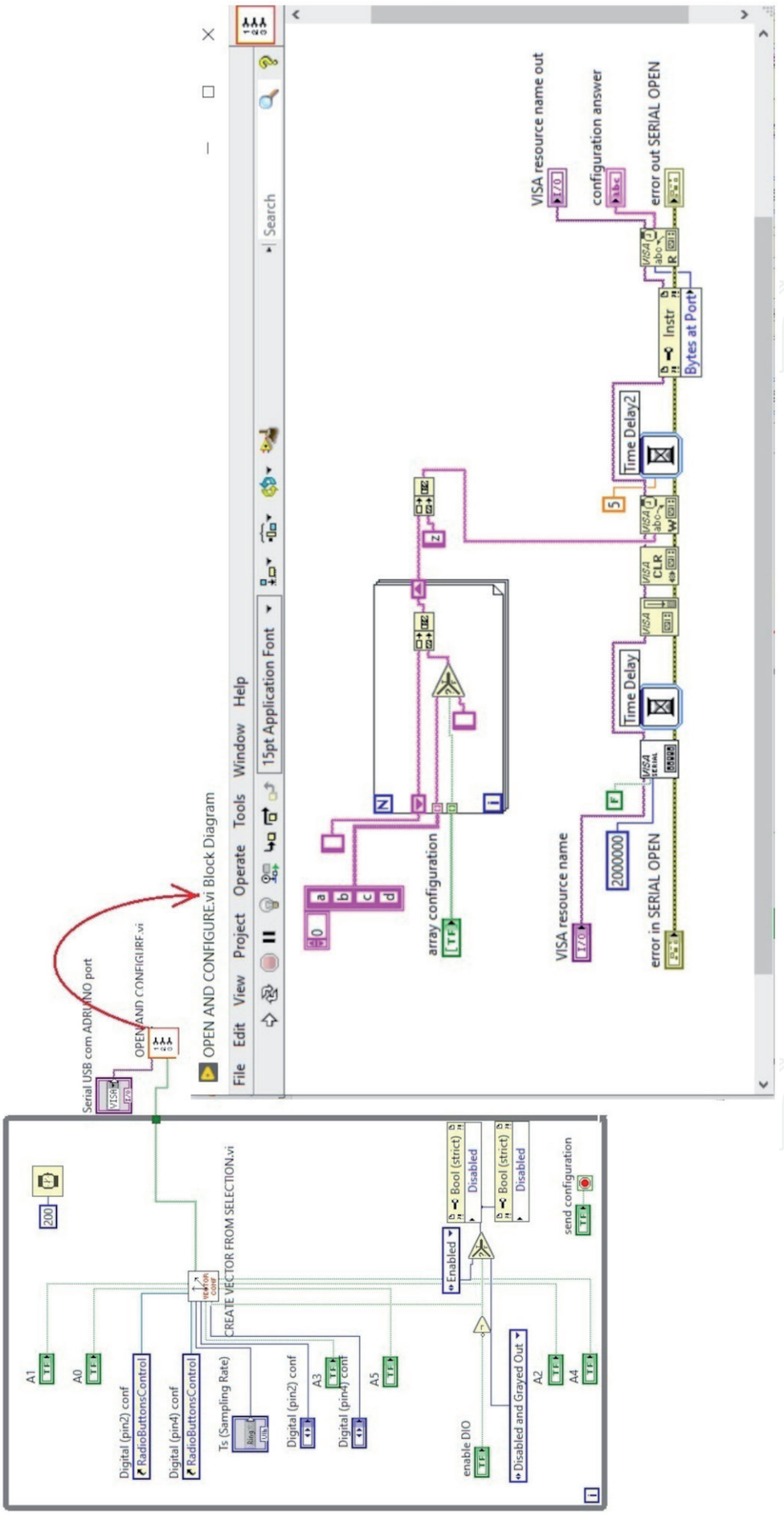

Figure 20.
First node and second node in block diagram (a and b).

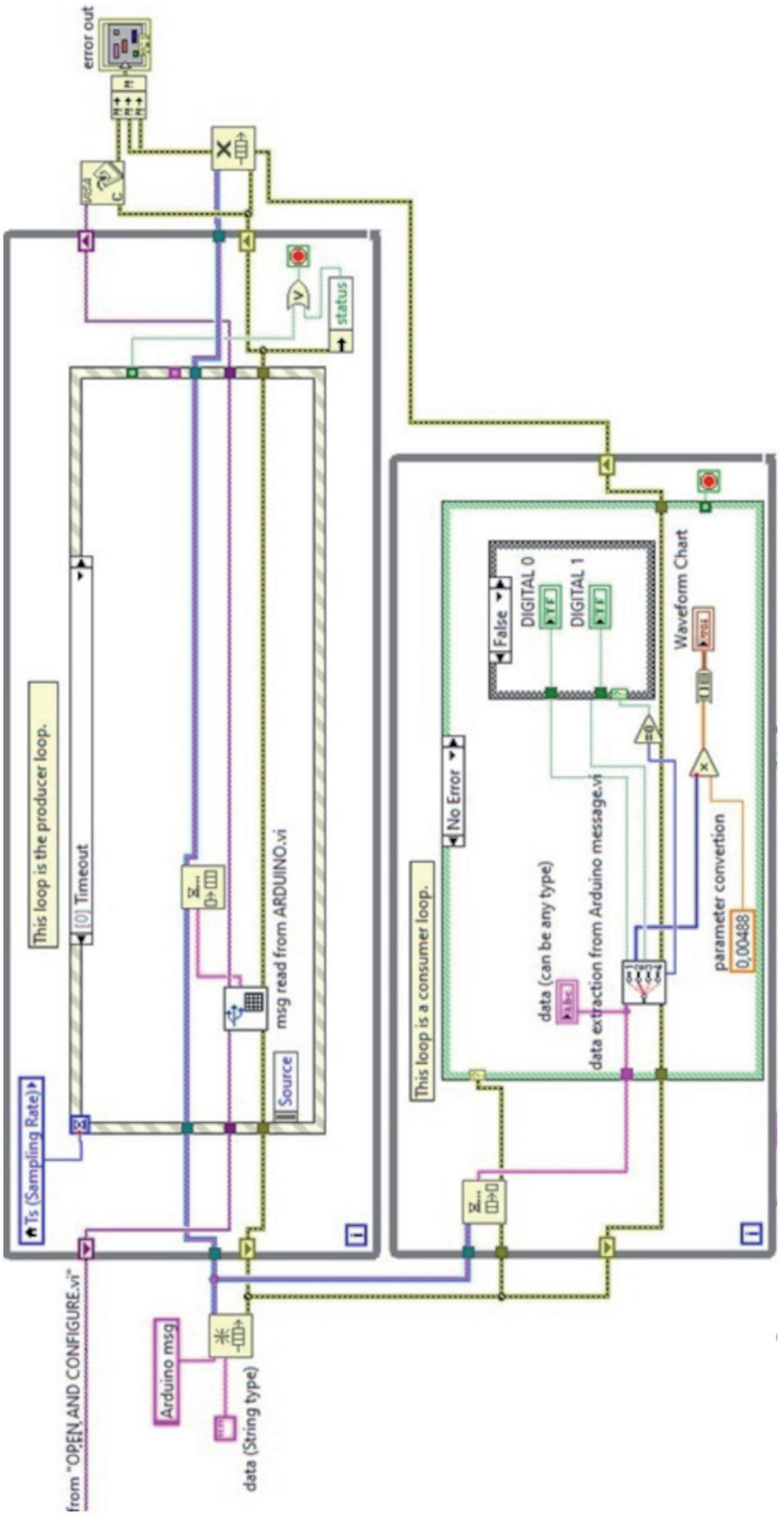

Figure 21.
Producer/consumer event-driven LabView CODE.

message at maximum frequency and by consumer loop we process the data. The Event-Driven statement is configurated with the following **events.**

5.3 Timeout

The timeout terminal of "Event Structure" is connected, of course, at local variable"Ts (Sampling Rate)" in according with sample rate configurated in Arduino in node 1. In this "case" we read with "msg read from ARDUINO.vi" the Arduino's message from serial (**Figure 22**). It is very simple code. The data are available on serial port (hardware) and the code read it using a **Bytes at Port** function.

5.4 Analog output [PWM]

in this case we send a message to Arduino by serial port, remember that Arduino reads the message with a SerialEvent() function. Here we make a message with a

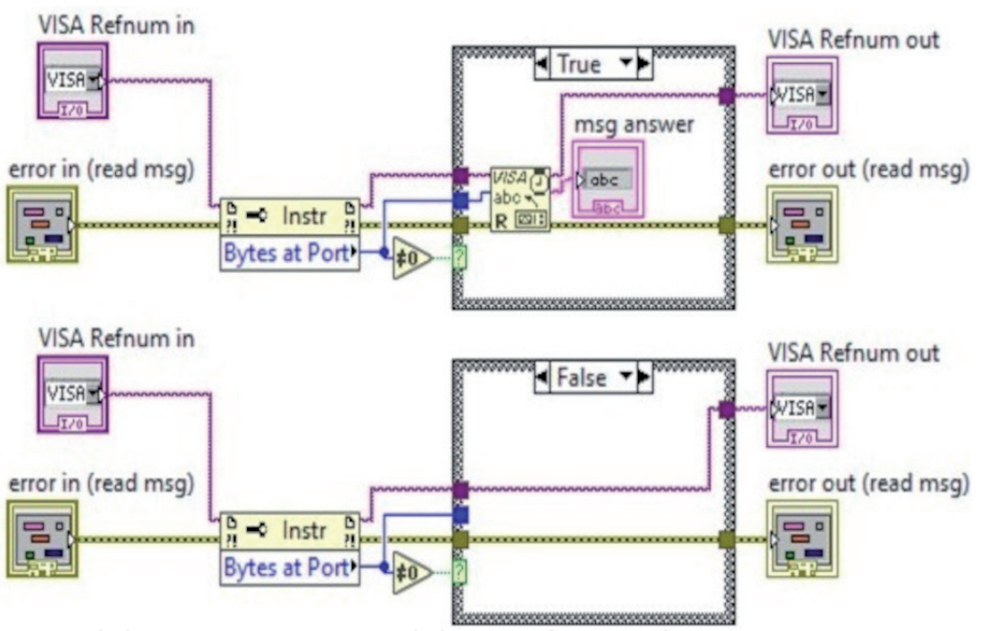

Figure 22.
Block diagram of msg read from ARDUINO.Vi.

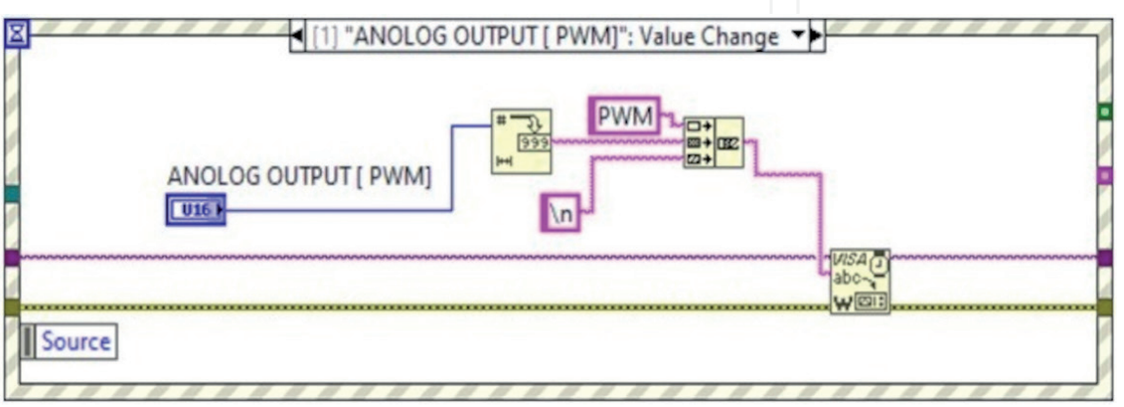

Figure 23.
Analog output [PWM] code.

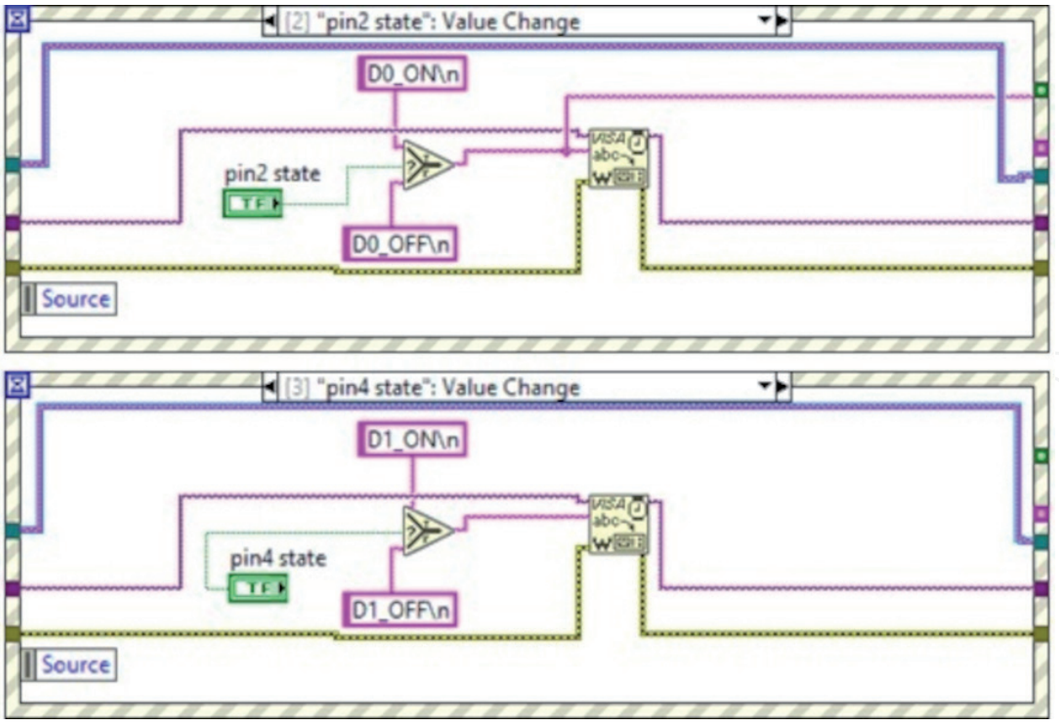

Figure 24.
Pin2 & pin4 event.

word PWM followed with "ANALOG OUTPUT [PWM]" control knob converted in ASCII code (**Figure 23**).

5.5 Pin2 state and Pin4 state

In this "case" (**Figure 24**) we build the message to send Arduino by serial port to change the digital pin state, remember that Arduino reads the message with a SerialEvent() function **Figure 12**.

Now we go back at **Figure 21** where we have to talk about the Consumer Loop. Through the enqueue function we read the data from the head of the queue with the FIFO method (first in first out). If we have not error the data read are processed with the subvi "data extraction from Arduino message.vi".

In **Figure 25** there is the screen code. The code scan the message, check if present special ID char (#) or (&) and collect the data by **indexing** it on the loop edge. With **Conditional indexing** we choose where collect the data: Analog Array or Digital Array.

The subvi "data extraction from Arduino message.vi" returns the status of the digital inputs and the numerical values of the analogue inputs, if configured.

To convert the integer values reads from analog ports we need to perform a simple conversion. According to what we have studied in the previous paragraphs having a 10 bit ADC and a dynamic of 5 volts we obtain:

$$\frac{5[V]}{2^{10}} = 0{,}00488[V] \tag{8}$$

At this point, in the consumer loop, before displaying the analogue signals on the Waveform chart we multiply the output by the value 0.00488.

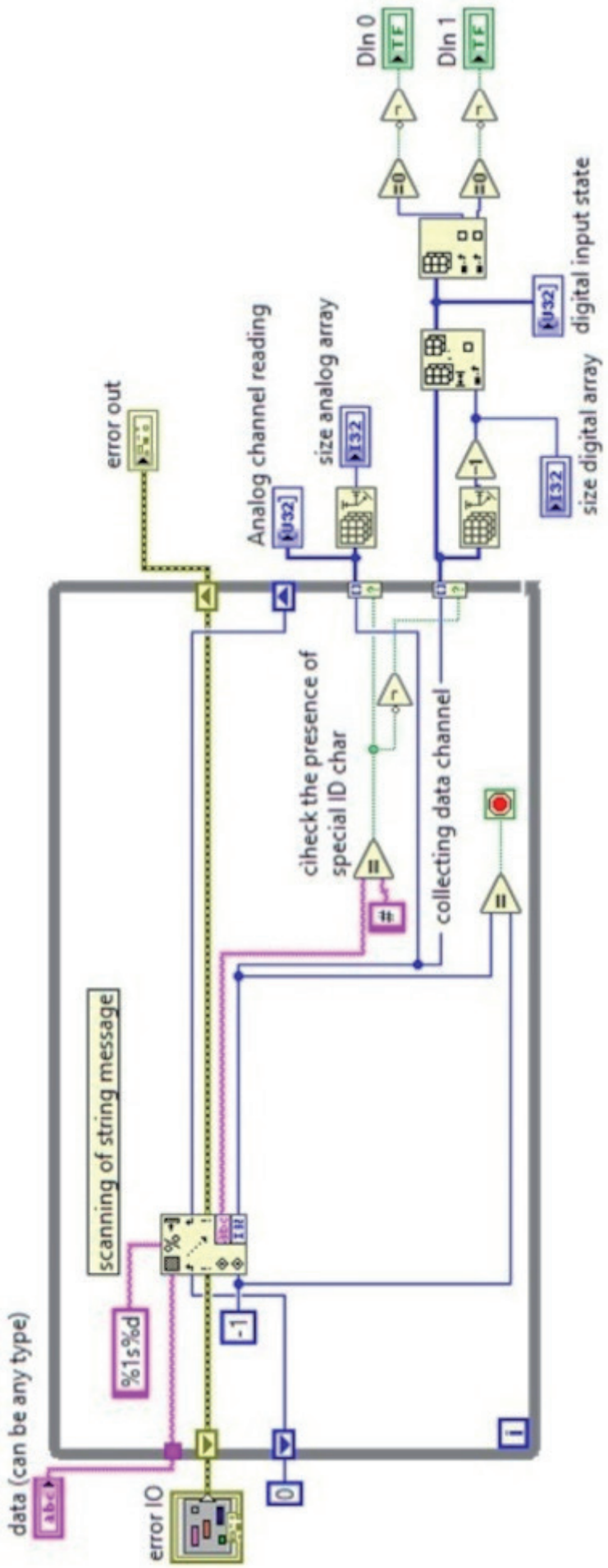

Figure 25.
Data extraction from Arduino message.vi.

6. Conclusions

In this chapter we have seen one of the many ways of how LabView can be used with third parties hardware. The idea is to have an inexpensive tool not for industrial use but for High School applications where it is possible with a few euros to set up a laboratory for the analysis of an RC/RLC circuit, voltage divider, diode/transistor characterization. With a cheap sensors, connected at Analogue inputs, you can prepare laboratory experiments such as the pendulum oscillation, spring characterization, measurements of angles in uniform angular motion, etc.

In the end you could organize LabView CORE I and CORE II training courses in e-learning where the DAQ board is very cheap and easily purchased on the web from the students.

Conflict of interest

The authors declare no conflict of interest.

Thanks

Dedicated to My wife and my daughters for encouraging and supporting me.

I would like to thank, my friend, the Director of the Department of Mathematics and Physics at my University, Prof. Lucio Gialanella, for supporting my initiative and for his precious advice.

Author details

Giuseppe Porzio
University of Study Of Campania "Luigi Vanvitelli", Caserta, Italy

*Address all correspondence to: giuseppe.porzio@unicampania.it

References

[1] LabVIEW™ Core I PN 326292A-01

[2] Giuseppe Porzio is the author of LabView SW for didactic experience

[3] Alan GS. Introduction to Arduino: a piece of cake. ISBN: 1463698348

[4] Jian S. Dynamics and Control of Switched Electronic Systems Pulse-Width Modulation. pp. 25-61. ASIN: B00A9YGCWC

[5] Available from: https://github.com/gporziog/LabView-and-connections-with-third-party-hardware/tree/master/Ardunio_Like_DAQ

[6] LabVIEW™ Core II PN 326293A-01

[7] Available from: https://github.com/gporziog/LabView-and-connections-with-third-party-hardware/tree/master/LabView%20Master%20CODE