Digital sound processing using arduino and MATLAB

Conference Paper · November 2012
DOI: 10.1109/SAI.2015.7237295

3 authors:

Sergio Silva
Universidade de Trás-os-Montes e Alto Douro
11 PUBLICATIONS **22** CITATIONS
SEE PROFILE

Salviano Soares
Universidade de Trás-os-Montes e Alto Douro
80 PUBLICATIONS **259** CITATIONS
SEE PROFILE

Antonio Valente
Universidade de Trás-os-Montes e Alto Douro
78 PUBLICATIONS **401** CITATIONS
SEE PROFILE

Digital Sound Processing using Arduino and MATLAB

Sérgio Silva[1,3]
[1]School of Sciences and Technology
Engineering Department
UTAD, Vila Real
sergio.s.silva@inescporto.pt

António Valente [1, 3]
[3]INESC TEC
INESC Technology and Science
(formerly INESC Porto, UTAD pole)
Porto, Portugal

Salviano Soares[1,2]
[2]IEETA
UA, Aveiro, Portugal

Sylvain T. Marcelino
IPLeiria/ESTG, Portugal
stam@co.it.pt

***Abstract*—Over the last decade, impelled by the huge open source software community support, the low cost Arduino platform presents itself as an alternative for digital sound processing. Although Arduino is generally used for small applications for the artistic and maker community, its built-in Analog to digital converter can be used for sound capturing, processing and reproduction. Equipped with a powerful AVR 8 bit RISC microcontroller, the Arduino, can achieve up to 200kHz with a 10 bit resolution according to the Atmel ATmega328P datasheet that is the AVR core that we are going to focus on this article.**

Realizing the hardware potential, software suppliers like Matworks or National instruments, have included the Arduino packages on the software accessories of MATLAB and LABView.

This work presents some of the sound capabilities and specific limitations of the Arduino platform, enfacing its connection and installation with MATLAB software. A series of examples of the Arduino interface with MATLAB are detail and shown in order to facilitate users initiation of MATLAB and Arduino Digital Sound Processing enhancing education fostering.

Keywords—Digital Sound Processing; MATLAB; Arduino; ADC/DAC; Sampling; Vocoder; FM Synthesis of Instrument Sounds

I. INTRODUCTION

The Atmel ATmega328P is the core of our Arduino and, as all the AVR cores, is equipped with a 10 bits analog to digital converter (ADC). An analog signal, as often call, is a continue steam of analog values and can be read or sampled a certain number of times per second, which is referred to as the sampling frequency (f_s). Figure 1 shows an analog signal with a D.C. component around 2.5Volts.

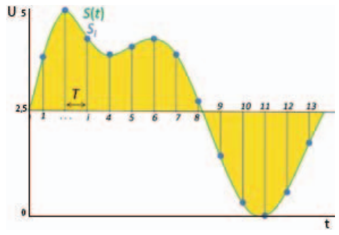


Fig. 1. Analog Signal

To deal with sound signals center around 0 Volts (V) ranging [-5; 5] (V) and to be able to read then with Arduino we need to do some level adjustment with a circuit like the one showed on figure 2.

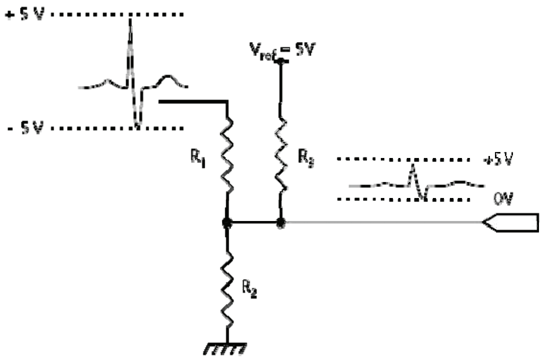


Fig. 2. Simple Level Shifting circuit

To get the right input, usually R_3 and R_2 have the same value around 10 kΩ and R_1 is often replaced by a decoupling capacitor of 10 μF (Coupling capacitors are used to block D.C. and pass A.C. that represents the music signal e.g).

There are different types of ADC architectures but most 8 bits AVR use successive approximation ADC, while 32 bits AVR uses comparative ADC. Figure 3 is a simplified diagram of the 8 bit AVR ADC.

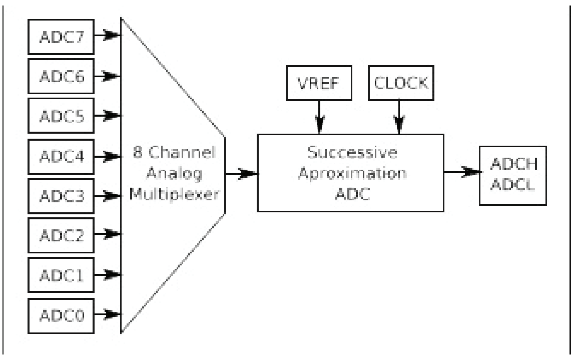


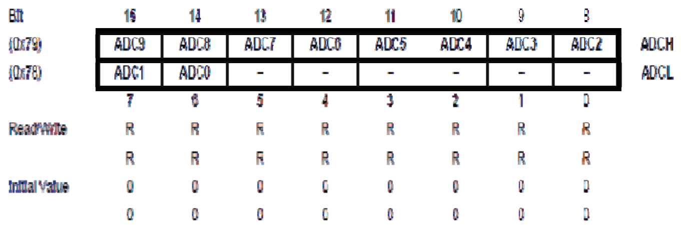
Fig. 3. ADC Block diagram [1]

The AVR only has a 10 bit ADC and uses an 8 channel analog multiplexer to sample each of the 8 analog. The ADC circuit takes samples between [13; 250] µs and AVR has a dedicated clock that ensures the independency of conversion from other microcontroller parts. The conversion mechanism can be triggered either on demand or automatically. The ADC conversion result is stored in two registers the ADC high and low. The result is obtained by the formula:

$$\left(\frac{V_{IN}}{V_{REF}}\right) * 1023$$

If V_{IN} is 2.5 Volts the converted value will be 512 and if $V_{IN} = V_{REF}$ than the result is 1023.

The Arduino uses a simple command to start a single conversion the "*analogRead(pin number)*". If nothing else is performed the conversion is complete. Figure 4 shows both ADC Data Register, in this particular case the ADLAR bit is set and the result is left adjust (the ADLAR bit is bit number 5 of the ADMUX – ADC Multiplexer Selection Register).



Bit	15	14	13	12	11	10	9	8	
(0x79)	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
(0x78)	ADC1	ADC0	-	-	-	-	-	-	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

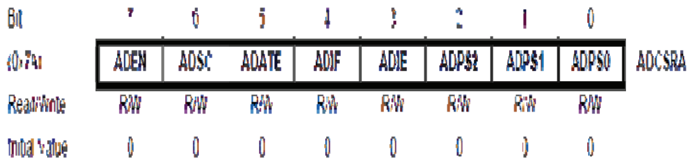
Fig. 4. ADC Data Register, ADCH and ADCL[1]

Remember that the ADC stores 10 bits the ADC0 … ADC9 so if we read only ADCH we ignore the first 2 bits.

It is also possible to use ADCW which is not found in Atmel datasheet in order to get the ADC result. To use it just assign it to an unsigned integer:

unsigned int adc_value = ADCW.

Clock frequencies between 50 kHz and 200 kHz should be use to get the higher resolution. Higher frequencies above the 200 kHz will produce less bits resolution. The ADC prescaler, is set by the ADC Control and Status Register A (ADCSRA), as shown in Figure 5.



Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 5. ADCSRA - ADC Control and Status Register A[1]

Combining the 3 ADPS bits sets the relation from AVR clock frequency and the ADC clock is display in Table 1.

Suppose we have system clock with frequency 16 MHz (16000000 Hz) and set division factor to 128 (Arduino default), then ADC clock frequency is:

$$\frac{16000000}{128} = 125000Hz = 125kHz \qquad (2)$$

TABLE I. ADCSRA DIVISION FACTORS

Division Factors	ADCSRA		
	ADPS2	ADPS1	ADPS0
2	0	0	0
2	0	0	1
4	0	1	0
8	0	1	1
16	1	0	0
32	1	0	1
64	1	1	0
128	1	1	1

So in order to set the ADC Prescaler to 64 we should write the following setup instructions:

bitSet(ADCSRA,ADPS2) ;

bitSet(ADCSRA,ADPS1) ;

bitClear(ADCSRA,ADPS0) .

The total conversion takes 14.5 clock cycles. AVR clock frequency is 16 MHz if we set the ADC prescaler to 2, the conversion period can be obtain using the formula:

$$T_{conv} = 14.5 * \frac{2}{16} = 1.8125\mu s \qquad (3)$$

Table 2 presents the ADC conversion period times and frequencies for all possible combinations of the prescaler the convertion frequency has calculated using the formula fconv=1/ Tconv.

TABLE II. ADC CONVERSION TIMES AND FREQUENCIES

ADC prescaler	Tconv (µs)	$T_m$conv (µs)	fconv (kHz)
2	1,8125	5,6000	178,571
4	3,6250	7,1900	139,082
8	7,2500	10,5800	94,518
16	14,5000	17,0600	58,617
32	29,0000	30,0600	33,266
64	58,0000	56,1200	17,819
128	116,0000	112,0700	8,923

The differences between measure and calculated times can be explain by the use of the *micros()* function that has a resolution of about 4 µs [2]. As stated by André Bianchi is not entirely true because on our implementation the results show that this difference only applies to the first 3 measurements and all other presents smaller differences.

With the ADC prescaler set to 16 and a clock speed of 1 MHz we achieve a total of 58,617 samples per second (f_s~59kHz) without compromising ADC resolution.

In fact the ADC accuracy also depends on the ADC clock. The recommended maximum ADC clock frequency is limited by the internal DAC in the circuitry conversion. For optimum performance, the ADC clock should not exceed 200 kHz, however, frequencies up to 1 MHz do not reduce the ADC resolution [3].

II. ADC IMPLEMENTATION

In order to perform the ADC measurements the following code was used:

```
unsigned long start;
unsigned long stop;
unsigned long ADC_value[100];
unsigned int j;
void setup() {
  Serial.begin(9600);  // Begin Serial port
  pinMode(1, INPUT);
  ADCSRA &= ~(1 << ADPS2) | (1 << ADPS1) | (1 <<
ADPS0); // ADC settings
  ADCSRA |= 1 << ADPS2;  // set 1 MHz  frequency
 }
void loop() {
        start = micros();  // starts measurements
        for(j=0;j<100;j++) {
          ADC_value[j] = analogRead(0);
        }
        stop= micros();
        Serial.println((stop - start));
        for(j=0;j<100;j++) {
          Serial.println(ADC_value [j]);
        }
        delay(5000);
}
```

The operation *ADCSRA* |= (*1* << *ADPS2*); sets bit ADPS2 from the ADC Control and Status Register A and is equivalent to the instruction bitSet*(ADCSRA,ADPS2);*

After processing we can use Pulse-Width Modulation (PWM) available in pins 3, 5, 6, 9, 10 and 11 of the AVR to convert it back to analog using an analog filtering stage to filter and smooth the sound wave.

III. Digital Signal Processing

One of the main concerns in terms of sound processing is, of course, the amount of time available in computation of output samples because they must be ready to be consumed by the playback hardware avoiding glitches and other unwanted artifacts [2]. Nevertheless there are ways that involve delay (buffering techniques) to compute all the samples in time.

For our work, we study two different approaches for sound processing. The first one involves a more compact and simple processing where all computation is done by the Arduino and on a second approach the samples are sent to the laptop where some MATLAB programs do all the computation before send it back to playback on the Arduino.

So let's first analyze how can the Arduino playback the sounds and what kind of hardware do we need.

To generate high quality sound, from the output signal, one needs to add a few but important components: for example if we need a 8 bit resolution (8 bit video game music e.g)[4], we just need to use one PWM pin attach to a resistor and a capacitor in a low pass filter configuration as shown on figure 6 (Cut-off frequency of 22.104kHz when R=1.8kΩ and a C=4nF capacitor).

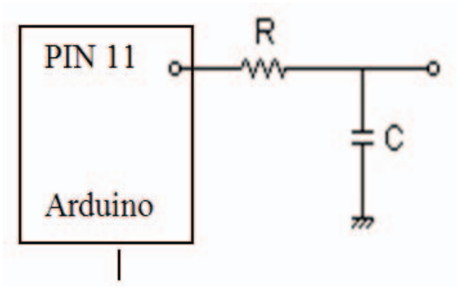


Fig. 6. Low Pass Filter

In order to calculate the PWM frequency we should explain what PWM is and how it does works on AVR.

PWM depends on the microcontroller timer. This is a simple internal clock that counts up to some number, and then goes back down to zero. PWM is generated by these timers, by having an external pin go *high* when the timer hits zero, and then go *low* at some other number, which we can vary. In this manner, it's possible to have an external pin stay high for a specific amount of time, without having to manually toggle it with the code.

A PWM waveform is generated from a counter by counting clock ticks, a register and a comparator [5]. The counter's purpose is to create the duty cycle resolution. One complete cycle of the counter is one period of the PWM. The counter's size (and thus the duty cycle resolution) is 8 bits, which equals to 256 values. So the 16MHz clock gets divided by 256 — because a cycle of the counter takes 256 ticks— and may give a maximum frequency of 62.5KHz. In order to get lower frequencies, we can divide your clock by another factor, known as the first clock division. This is also called pre-scaling, because it precedes the counter. Table 3 summarizes the overflow interrupt frequency for all possible values of prescaler.

TABLE III. Overflow Interrupt Frequency for Prescaler Values

PWM prescaler	f_{incr} (KHz)	$f_{overflow}$ (Hz)
1	16000	62500
8	2000	7812
32	500	1953
64	250	976
128	125	488
256	62,5	244
1024	15,625	60

The register and the comparator are used to set and create the duty cycle output. The register sets for how long during each period the output is high. For example for a 25% duty cycle, we would set the register to 256*0.25-1=63. The comparator compares the register's value with the current counter value and gives *high* in the output if the value of the former is equal or lowers from the value of the latter and *low* otherwise.

For each counter we actually get 2 registers and 2 comparators, so we can get 2 PWM waveforms with different duty cycles but with the same frequency. Whenever is possible loosing precision, a faster PWM frequencies than 62,500 kHz on AVR is available.

Let's take some time to analyze the trade of between PWM frequency and noise floor.

The smallest sound that can be heard is correlated to the noise floor. This is the low level "hiss" heard in the background of most signals. For sound applications, there must be undetectable by the ear but, this is often not achievable with a single PWM generator. To get a lower noise floor, is necessary to use more bits, and the exact amount is set by the equation:

$$SNR_{(dB)} = (Bit\ Depth) * 6.02_{(dB)} + 1.76_{dB}$$

Two options are available to get more bits: lower the PWM frequency, or increase the number of PWM in use. Due to Nyquist theory, the PWM frequency must be at least twice the highest frequency of interest [6]. Furthermore, if the PWM frequency is in the audible range (less than 22 kHz) we will need to filter it heavily to not hear a high pitched squeal behind the sounds. This sets a hard floor for how many bits you can achieve with your Arduino without having to add a ton of extra circuitry.

Also AVR have two different kinds of PWM: the Fast PWM (Single slope) versus Phase Correct PWM (Dual Slope). With Fast PWM, the counter will increase to TOP, and then reset to zero, whereas Phase Correct PWM will reach TOP, and then count backwards to zero, where it will count up again. Phase Correct takes twice as long to complete a cycle, so it will only go half as fast for any given bit depth but is much higher fidelity [4].

So let's build some code to summarize all of what has been said until now.

```
const int PWM1 =11;     //pin for the PWM output
const int PWM2 =3;      //pin for the PWM output
int val=0;                   //variable used to store the value
void setup()
{
  cli(); //disable interrupts while registers are configured
  // Setup ADC to work  with division factor of 16  and a
clock speed of 1 MHz
  // we achieve a total of 58,617 samples per second 58 kHz
  bitSet(ADCSRA,ADPS2) ;
  bitClear(ADCSRA,ADPS1) ;
  bitClear(ADCSRA,ADPS0) ;
  DIDR0 = 0x01;
  // Analog output configuration
  bitSet(TCCR2A, WGM20);  //Puts bit 0 of the TCCR2A
register (Timer/Counter
                              //Control   Register   A),   named
WGM20 to 1
  bitClear(TCCR2A, WGM21); //Puts bit 1 of the TCCR2A
register (Timer/Counter
                              //  Control   Register   A),   named
WGM21 to 0
  bitClear(TCCR2A, WGM22); //Puts bit 2 From TCCR2A
register (Timer/Counter
                              //  Control   Register   A),   named
WGM22 to 1
  /*This sets Timer2 to PWM, Phase Correct mode
   Table of Timer/Counter Mode of Operation
   WGM22  WGM21  WGM20    Timer/Counter  Mode  of
Operation
   0    0    0          Normal
   0    0    1          PWM, Phase Correct
   0    1    0          CTC  - Clear Timer on Compare Match
(CTC) Mode
   0    1    1          Fast PWM
   1    0    0          Reserved
   1    0    1          PWM, Phase Correct
   1    1    0          Reserved
   1    1    1          Fast PWM
   */
  bitSet(TCCR2B, CS20);      //Sets to 1 bit CS20 (bit 0) of
TCCR2B register
                              // (Timer/Counter Control Register
B)
  bitClear(TCCR2B, CS21);    //Sets to 0 bit CS21 (bit 1) of
TCCR2B register
                              //      (Timer/Counter       Control
Register B)
  bitClear(TCCR2B, CS22);    //Sets to 0 bit CS22 (bit 2) of
TCCR2B register
                              //(Timer/Counter Control Register
B)
  sei(); //enable interrupts now that registers have been set
  pinMode (PWM1,OUTPUT);
  pinMode (PWM2,OUTPUT);
  Serial.begin(57600);
}
void loop()
{
  val=analogRead(0);                //read value of sound in at
analogRead on pin 0
  Serial.println(val);
  analogWrite(PWM1,map(val,400,600,0,255));
  analogWrite(PWM2,map(val,400,600,0,255));
}
```

In the code we set the ADC to work with division factor of 16 and a clock speed of 1 MHz that gives 58.617 samples per second and we set the PWM for phase correct at 31,250 kHz with 2 pins with 2 resistors for hardware mixing of an upper and lower value: Figure 7 shows the circuitry hardware setup.

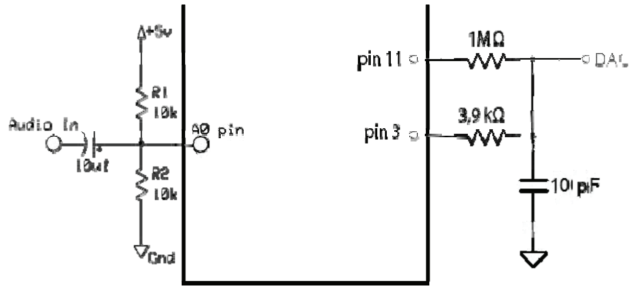


Fig. 7. Circuitry hardware Setup [4]

IV. MATLAB INTERFACE WITH ARDUINO

For installing the Arduino software package in MATLAB, just type the command *supportPackageInstaller* this starts the Support Package Installer and select the web installation. Finally choose the Arduino package and it should have installed your Arduino package support.

In order to check if everything it's ok we are going first to develop a data logger. In MATLAB, there are several functions that are related to Serial communication. First, we need to target a Serial COM Port. Replace COM31' with whatever port your Arduino is connected to and set the communication speed.

```
s = serial('COM31');
```

```
s.BaudRate=115200;
```

Next, we need to open the Serial port.

```
fopen(s);
```

Then, to read from it and store to var data, we write:

```
dat = fscanf(s);
```

```
dado = str2double(dat);
```

The first reads the data and the second converts it from char to double.

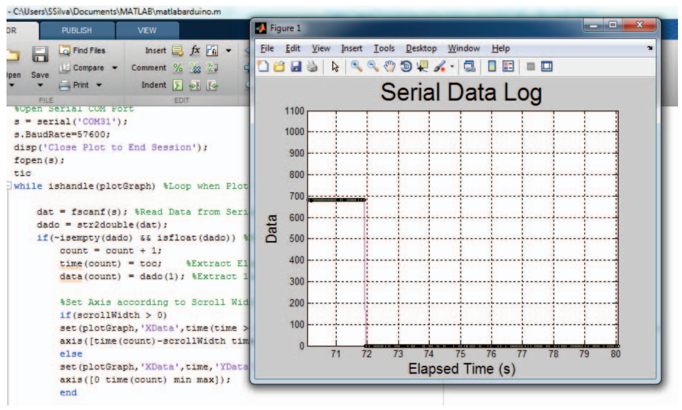
After finishing with the Serial COM port, it is very important remember to close it else other applications cannot access it.

```
fclose(s);
```

So let's put all together and check if our ADC is sampling well.

```
% **CLOSE PLOT TO END SESSION
clear
clc
%User Defined Properties
plotTitle = 'Serial Data Log';  % plot title
xLabel = 'Elapsed Time (s)';    % x-axis label
yLabel = 'Data';                % y-axis label
plotGrid = 'on';                % 'off' to turn off grid
min = 0;                        % set y-min
max = 1100;                     % set y-max
scrollWidth = 10;               % display period in plot, plot entire data log if <=
0
delay = .0001;                  % make sure sample faster than resolution
%Define Function Variables
time = 0;
data = 0;
count = 0;
%Set up Plot
plotGraph = plot(time,data,'-mo',...
            'LineWidth',1,...
            'MarkerEdgeColor','k',...
            'MarkerFaceColor',[.49 1 .63],...
            'MarkerSize',2);
title(plotTitle,'FontSize',25);
xlabel(xLabel,'FontSize',15);
ylabel(yLabel,'FontSize',15);
axis([0 10 min max]);
grid(plotGrid);
%Open Serial COM Port
s = serial('COM31');
s.BaudRate=115200;                          %define baud rate
disp('Close Plot to End Session');
fopen(s);
tic
while ishandle(plotGraph) %Loop when Plot is Active

    dat = fscanf(s); %Read Data from Serial as Float
    dado = str2double(dat);
    if(~isempty(dado) && isfloat(dado)) %Make sure Data Type is Correct
        count = count + 1;
        time(count) = toc;    %Extract Elapsed Time
        data(count) = dado(1); %Extract 1st Data Element

        %Set Axis according to Scroll Width
        if(scrollWidth > 0)
        set(plotGraph,'XData',time(time > time(count)-
scrollWidth),'YData',data(time > time(count)-scrollWidth));
        axis([time(count)-scrollWidth time(count) min max]);
        else
        set(plotGraph,'XData',time,'YData',data);
        axis([0 time(count) min max]);
        end

        %Allow MATLAB to Update Plot
        pause(delay);
    end
end

%Close Serial COM Port and Delete useless Variables
fclose(s);
clear count dat delay max min plotGraph plotGrid plotTitle s ...
        scrollWidth serialPort xLabel yLabel;
disp('Session Terminated...');
```

If there are now news you should see on MATLAB a graph appearing like the one on Figure 8 (on second 72 we disconnect A0 from 3.3 volts and connected it to GND).



```
%Open Serial COM Port
s = serial('COM31');
s.BaudRate=57600;
disp('Close Plot to End Session');
fopen(s);
tic
while ishandle(plotGraph) %Loop when Plot

    dat = fscanf(s); %Read Data from Seri
    dado = str2double(dat);
    if(~isempty(dado) && isfloat(dado)) %
        count = count + 1;
        time(count) = toc;    %Extract El
        data(count) = dado(1); %Extract 1

        %Set Axis according to Scroll Wid
        if(scrollWidth > 0)
        set(plotGraph,'XData',time(time >
        axis([time(count)-scrollWidth tim
        else
        set(plotGraph,'XData',time,'YData
        axis([0 time(count) min max]);
        end
```

Fig. 8. Serial Data Log In MATLAB

We will notice that if everything it's okay then when connecting A0 port from the Arduino to 5 or 3.3 or even 0 Volts the values will appear on logger and should have precise values. If we change the *scrollWidth* parameter from 10 to 0 the graph will display the entire log history instead of just a section. After closing the plot, the data log is available by accessing the data variable in the workspace.

V. OTHER MATLAB SOUND APPLICATIONS

A. Vocoder Application

A Vocoder is by definition a sound effect that can make a human voice sound synthetic [7,8] so, one of the purposes is to replace the carrier sound with another carrier from a different source. A main goal is obtained: it changes how the original sound sounds but keeps the original message.The next example shows a small sample of our Vocoder application.

To create the vocoder a Matlab function Start Vocoder App is call as can be seen on the block diagram that illustrates the operation of the vocoder. After executing the start function a wave file is call and transformed into several parameters namely x-sampled data, f-sampling rate and b-number of bits. The first two are sent to the vocoder function where we simulate a transmission channel, with transmission and reception, after the signal is recover, from the parameters sent and, after reconstruction, the signal is playback.

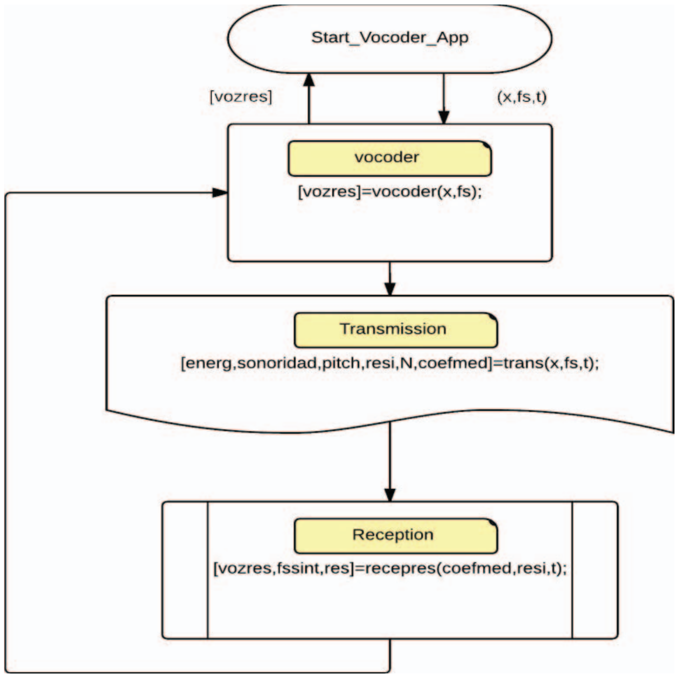

Start_Vocoder_App
[vozres]
(x,fs,t)
vocoder
[vozres]=vocoder(x,fs);
Transmission
[energ,sonoridad,pitch,resi,N,coefmed]=trans(x,fs,t);
Reception
[vozres,fssint,res]=recepres(coefmed,resi,t);

Fig. 9. Block diagram of Vocoder Application

After playing the reconstruct signal is possible to execute some measurements to evaluate the quality of the reconstruct signal. Figure 10 shows the original signal versus the synthesized and the error.

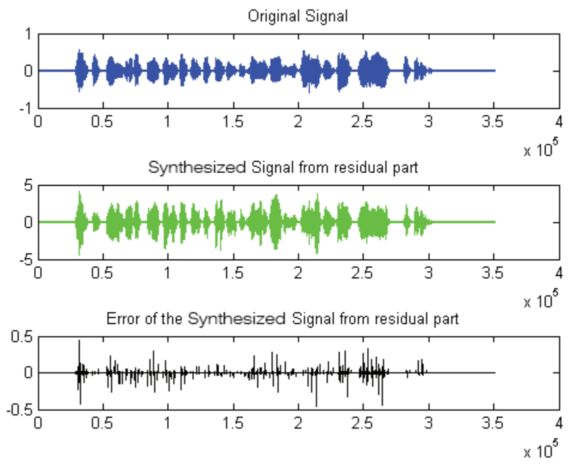


Fig. 10. Original versus synthesized and error signal

From the figure analyses it's clear that the error from the synthesized signal from the residual part is very small, figure 11 show the overlap of the 3 signals showing this fact.

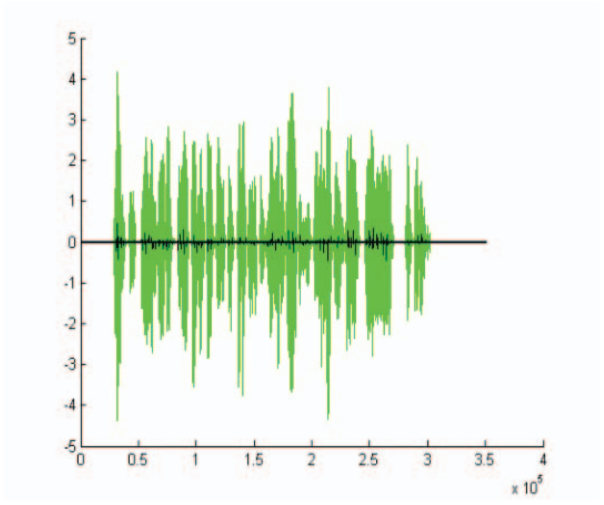


Fig. 11. Overlapped of original, synthesized and error signals

The Matlab the code for signal reconstruction is:

```
Function [vozres,fssint,res]=recepres(coefmed,resi,t);
N=length(coefmed);
l=length(resi);
vozres=[];
c=[];
residu=[];
res=[];

 for i=1:N
     %%ASSIGNED EACH LINE OF COEFICIENTS(H) TO A VECTOR
    for j=1:11
   c(j)=coefmed(i,j);
    end
    %%ASSIGNED EACH LINE OF RESIDUAL(H) TO A VECTOR
    for s=1:l
   residu(s)=resi(i,s);
    end
    %%REVERSE FILTER FROM CODIFICATION
   res=[res residu];
   v=filter(1,c,residu);
   vozres =[vozres v];
 end
  fssint=(l*1000)/t;
  wavwrite(vozres,fssint,'vozres.wav');
```

At the end of the code the play frequency is calculated and the residual reconstructed signal wav is created.

B. FM Synthesis of Instrument Sounds

Frequency modulation can be used to make interesting sounds that mimic musical instruments, such as bells, woodwinds or drums.

Our Bell application allows the user to control the sound of bells by touching different parts from the control system. To show the potential of the Arduino-Matlab connection the control system is attached to the Arduino and consists into 12 wires connected to the Arduino allowing music performance in one octave Figure 12.

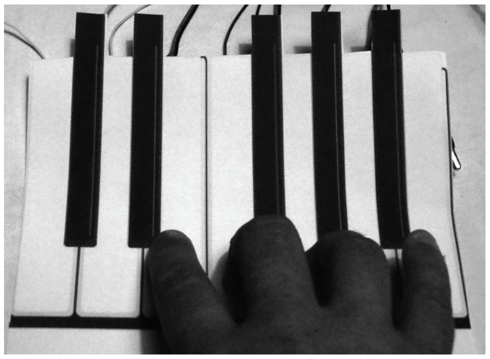


Fig. 12. Piano Paper Octave with wires to arduino

The implementation code from the matlab size is:

```
dur=6;
tau=2;
Io=10;
fc=110;
fm=220;
Fa=8000;
samples=[0:1/Fa:dur];
tempoexecucao=300; % 5 minutes 300 seconds
%Open Serial COM Port
s = serial('COM36');
s.BaudRate=9600;
disp('0 to end session or wait 5 minutes');
fopen(s);
s.ReadAsyncMode = 'continuous';
t=0;
tic
dat=4;
while t<tempoexecucao
   if (s.BytesAvailable>0)
   dat = fscanf(s); %Read Data from Serial as Float
   temp = str2double(dat);
   if(temp>0&&temp<8)
      tau=temp;
   end
   if(temp>=8 && temp<20)
      Io=temp+2;
   End
   t=toc;
   if(temp==0)
      t=301;
      break
   end
for i=1:size(samples)
   A=exp(0-samples(i)/tau);
   I=Io*exp(-samples(i)/tau);
   onda(i)=A*cos(2*pi*fc*samples(i)+I*cos(2*pi*fm*samples(i)-
              pi/2)-pi/2);
end
   sound(onda,Fa);
   end
end
```

It's also possible a different operation mode where by touching the different wires the user controls different parameters creating different sounds. The general equation for an FM sound synthesizer is [9].

$$x(t) = A(t)\cos(2\pi f_c t + I(t)\cos(2\pi f_m t + \varphi_m) + \varphi_c)$$

Where $A(t)$ is a function of time called envelope, f_c is the carrier frequency, f_m is the modulation frequency, and technically $I(t)$ is called the modulation index envelope.

TABLE IV. DIFFERENT CASE VALUES FOR THE BELL SOUND[9]

Case	f_c Hz	f_m Hz	$I_{(0)}$	T sec	T_{dur} sec	F_s
1	110	220	10	2	6	11,025
2	220	440	5	2	6	11,025
3	110	220	10	12	3	11,025
4	110	220	10	0.3	3	11,025
5	250	350	5	2	5	11,025
6	250	350	3	1	5	11,025

We will demonstrate this application and others during the conference.

VI. CONCLUSION

This paper present an approach to Digital Sound Processing using the Arduino platform and MATLAB, with some examples we show that the Arduino can be use together with Matlab to achieve the connection between physical and computation worlds. The AVR's registry analysis gave us the insight to how we should control our Arduino in order to achieve the best performance. An example of MATLAB and Arduino shows how we can proceed with better sound analysis since the choose Arduino limitations. One should here remember that there is also the Arduino Due that is around 15 times faster than the Arduino Uno used and has 2 DAC ports besides the normal PWM ones. Furthermore it has a 12 bits ADC instead of the current 10 bits of the Uno, unfortunately it doubles the Arduino Uno platform price. Nevertheless this price is still much less than the traditional DSP platforms used in Sound processing.

VII. FUTURE WORK

The Arduino platform can be use together with Matlab for real time sound processing, but some cares should be taking into account when using it, because the sampling process introduces some noise.

The use of Interrupt routines for the ADC sampling/PWM synthesis process and compare the performance improvements relatives to our approach is also a viable path.

ACKNOWLEDGMENT

This work is financed by the ERDF − European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT − Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project ref. FCOMP – 01 – 0124 – FEDER − 022701, and in the context of the projects PEst-OE/EEI/UI0127/2014 and Incentivo/EEI/UI0127/2014.

REFERENCES

[1] "Atmel ATmega48A/48PA/88A/88PA/168A/328/328P datasheet," http://www.atmel.com/devices/ATMEGA328P.aspx?tab=documents, [Online; accessed 1-Oct-2014].

[2] A André Jucovsky Bianchi - Real time digital audio processing using Arduino http://www.ime.usp.br/~ajb/artigos/article-smc2013-ajb-mqz.pdf [Online: accessed 1-Oct-2014]

[3] "AVR120: Characterization and Calibration of the ADC on an AVR – Application Note, 2006." http://www.atmel.com/images/doc2559.pdf, [Online; accessed 12-Oct-2014]

[4] Dual PWM Circuits: Online Article from Open Music Labs http://www.openmusiclabs.com/learning/digital/pwm-dac/dual-pwm-circuits/, [Online; accessed 1-Oct-2014].

[5] M. Nawrath, "Arduino realtime sound processing," http://interface.khm.de/index.php/lab/experiments/arduino-realtime-sound-processing/, [Online; accessed 12-Jun-2014]

[6] Alan. V. Oppenheim, Ronald. W. Schafer, "Digital Signal Processing". Prentice Hall, 1975.

[7] L. R. Rabiner, Ronald. W. Schafer, "Digital Processing of Speech Signals". Prentice Hall, 1978.

[8] Achim Settelmeier – Online Article - What is a Vocoder, http://www.sirlab.de/linux/descr_vocoder.html [Online: accessed 5-Oct-2014]

[9] James. H. McClellan, Ronald. W. Schafer, Mark A. Yoder. "DSP FIRST A Multimedia Approach ". Prentice Hall, 1998.