Digital keyboard

Ognjen Glamocanin, EE15/2013, *UNS, FTN, KEL, Novi Sad*

***Abstract*— Sound, as a mechanical wave, by nature an analog signal, can be driven from electrical loudspeakers with analog electric signals. A digital keyboard is a mixed digital and analog system which creates and drives those signals. The main problem of this paper is the process of storing the samples of an arbitrary shaped signal into a memory of a microcontroller, and loading them at a wanted frequency creating a digital signal, converting that digital signal to an analog signal using a DAC, and amplifying and driving that signal on loudspeakers – in other words, making a simple digital keyboard.**

***Index Terms*— Analog signal, AT89S51, Audio Amplifier, Coding, DAC, DDS, Digital Keyboard, Digital Signal, DSP, Sampling, Sample**

I. INTRODUCTION

Digital (sound) signal processing (DSP) is a rapidly growing field of research. It is used to create and alternate sound signals in many different ways. With the rapid development of digital electronics, specialized DSP microprocessors were created, with a goal to process digital signals faster and better. Those microprocessors are now the basis of every quality digital keyboard on market. This paper, however, has a goal to present a design of a simple digital keyboard using a textbook (non DSP) microcontroller AT89S51. The writer is aware that there are more optimal solutions for this kind of problem, but the accent in this paper is on elaborating the concept of the creation of arbitrary signal shapes using microcontrollers and driving those signals on a speaker. This concept is relatively old, and is classified as a basic concept in DSP, but by writing this paper, the author wanted to learn more about that topic, as a stepping stone to the field of digital audio signal processing. In the earlier mentioned concept a major question arises: How can we create a sound signal, of an arbitrary shape, which is analog by nature, using a microcontroller? Using the theory of sampling, the memory of the microcontroller and peripheral electronic circuits (DAC, audio amplifiers, etc.) this question will be answered in this paper.

The II chapter is about the basic principal of creating a signal of an arbitrary shape using memory loading. The III chapter is about defining an optimal signal shape for an audio signal. The IV chapter is about signal sampling. The V chapter is about signal quantization and coding. The VI chapter is about musical note frequencies, and in the VII chapter the microcontroller code is defined. The last three chapters are related to the electronic interface of the microcontroller: VIII is about the AT89S51 interface circuit, IX about interfacing the DAC with the microcontroller and the X about the audio amplifier interface. Appendix A contains the MATLAB sampling code, appendix B contains the C quantizer and coder code, and finally, appendix C contains the microcontroller code.

II. MEMORY LOADING METHOD

The general idea of this project is to create a digital keyboard based on memory loading of the AT89S51 at a specific frequency. A sampled and coded signal of an arbitrary shape is stored in the ROM memory of the microcontroller. Using a specific method for measuring time on a microcontroller, samples are loaded and driven on the pins with a wanted frequency. The digital signal is then converted to its analog representative using a digital to analog converter (DAC). The analog signal is then amplified and driven on a loudspeaker using an audio amplifier. The block diagram of the system that implements the memory loading method is shown on Fig. 1.

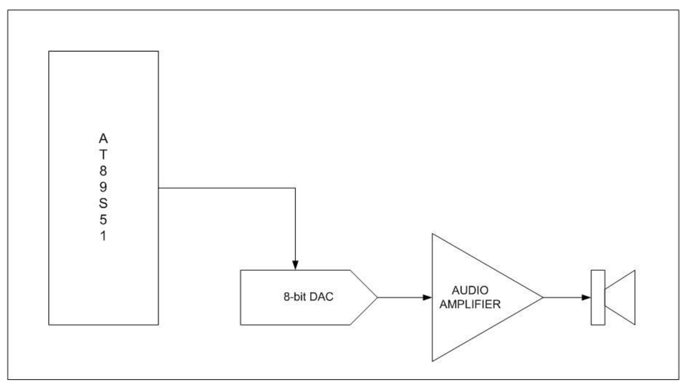

Figure 1. Memory Load System

III. SIGNAL SHAPE

As the goal of this paper is to create a digital sound keyboard, the signals stored in the memory of the microcontroller should be sound signals. More accurately, they should have a sine basis of a specific frequency - $A_0\sin(2\pi f_o t)$. As a pure sine signal with the frequency of the wanted musical note sounds more like a beep than a music note, adding harmonics is necessary to create a more pleasant sound. By adding harmonics, the sound signal changes in shape and in sound. Every next harmonic has an amplitude that is equal to the amplitude of the base harmonic divided by the current harmonics frequency multiplying factor N –

$(A_0/N)*\sin(2\pi N f_0 t)$. Using the program package MATLAB, it is easy to see and hear the resulting signal. With every following harmonic, the resulting shape resembles more and more to a saw shaped signal, and the sound becomes more pleasant to the ear. A sine signal, a signal with even and odd harmonics and a signal with just odd harmonics are shown on fig. 2.

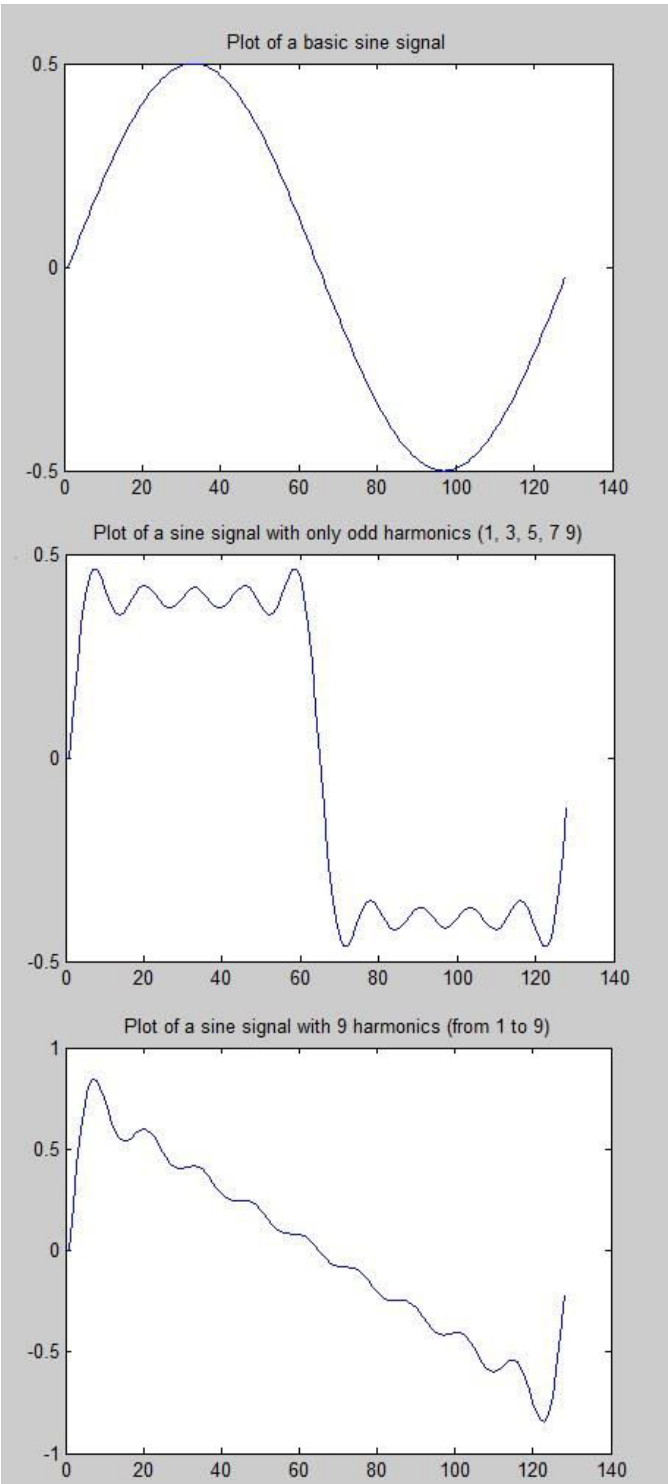

Figure 2. Signals With and Without Harmonics

IV. Signal Sampling

Signals mentioned in the previous chapter are analog (continuous). In order to store a signal into a memory of a microcontroller, the signal must be converted to a digital signal. The first step in that conversion is signal sampling. The signal has to be sampled in discrete intervals of time. The result of this operation is a discrete signal. In order to be properly reconstructed, the sampling frequency has to fulfill the Nyquist sampling theorem: the sampling frequency must be at least two times higher than the highest frequency in the signals spectrum. In the case of 9 harmonics, the highest frequency is the one of the ninth harmonic ($9*f_0$). If we decide for 128 samples per period, the sampling frequency is $128*f_0$. As $128*f_0 > 2*f_0$, the sampling theorem is fulfilled. Number 128 is chosen because it is a good compromise between memory usage and a number of samples needed to properly restore a sound signal into its analog form.

The program package MATLAB can also sample a signal, and store the samples into a text file. The code that does that functionality can be found in appendix A.

V. Signal Samples Quantization and Coding

The result of sampling is a discrete signal. It has a continuous number of values for its amplitude, but it is discrete in time. In order to get a purely digital signal, the amplitude has to be rounded and coded. There are many digital codes used, each with its complexity, advantages and flaws. The main parameter for choice of the code of the samples is the code that the DAC uses, so that the microcontroller can send data directly to the DAC, without using a code convertor.

In this case, the code used by the selected DAC is the 8 bit shifted binary code, where the maximum of the absolute value of the signal is coded by 0xFE, zero is coded by 0x7F and the negative of the maximum of the signals absolute value is coded by 0x00. The value of 0xFF is not used. Because 0 is not coded by 0x00, the code is called shifted binary code. The amplitude difference between two levels is

$$\Delta A = \frac{2*\max(A_{positive}, |A_{negative}|)}{2^n-2},$$

where n is the number of bits used, $A_{positive}$ the positive amplitude of the signal and $A_{negative}$ the negative amplitude of the signal. The 2^n-2 comes from the fact that the 0xFF level is not used (-1), and the fact that ΔA is the space between levels, not the level itself (between three points there are two line segments). Knowing ΔA and A = -max($A_{positive}$, |$A_{negative}$|), the values of voltage levels can be calculated by adding ΔA, multiplied by a number, to A. The binary value of the multiplication number is the binary coded level. For example, A+0* ΔA is coded by 0x00, A+1* ΔA is coded by 0x01, etc. The discrete signal has values that are different from the values used by the code. The process of rounding those values to the values used by the code is called quantization, and the process of the assignment of those values to a binary code is called coding. If a value is between two levels, it must be rounded to a closer value. The decision levels are in the middle of two levels used by the code. If a value is higher than the decision level, the value is rounded to and coded by the higher number; otherwise, it is rounded to and coded by the lower number.

The quantization and coding of the signal can be done by hand, or it can be done by writing a program that will do it automatically. In this case, the author has written a program in C language that will reduce the work needed to quantize and code the signal. The C code can be found in appendix B.

VI. MUSICAL NOTE FREQUENCIES

As mentioned before, each musical note is theoretically a sine signal with a specific frequency. The list of the musical notes of interest in this paper and their frequencies is shown in Table I.

TABLE I
MUSICAL NOTE FREQUENCIES

Note	Frequency (Hz)
C_4	261.63
$C^{\#}_4/D^b_4$	277.18
D_4	293.66
$D^{\#}_4/E^b_4$	311.13
E_4	329.63
F_4	349.23
$F^{\#}_4/G^b_4$	369.99
G_4	392.00
$G^{\#}_4/A^b_4$	415.30
A_4	440.00
$A^{\#}_4/B^b_4$	466.16
B_4	493.88
C_5	523.25

VII. MICROCONTROLLER CODE

The code from appendix B provides the digital samples of the wanted sound signal. The samples must be manually stored into the memory of the microcontroller and loaded to the pins at a precise frequency. As previously mentioned, the microcontroller chosen by the author is AT89S51, because it is a textbook microcontroller learned in a lot of university programs. The S51 series were chosen because of the availability and price reasons.

The variable that contains the digital samples of the signal is a 128 bit long unsigned char array. Unsigned char type was chosen because the samples are the same length as an unsigned char variable - 8 bit long. Because the AT89S51 has 128 bytes of RAM, the variable cannot be stored into the RAM memory, and it must be stored into the ROM memory, using the Keil uvision3 keyword *code*. Storing the variable into the ROM memory makes it a constant.

The note determination is simple: 13 push buttons represent 13 musical notes (12+C_5). When a button is pushed, the program enters in a while loop where samples are loaded with a specific frequency, and exits when the button is released. This means that only one note can be played at a time.

The frequency of the samples that are loaded and driven to the pins from the constant is crucial; a minimal difference between the wanted and the actual frequency can result in a completely different musical note.

The microcontroller AT89S51 has two embedded counter/timers, and they can be used for timing purposes. However, because of the time needed for the microcontroller to execute the ISR, an error in timing can occur, so the timers are not used when precise timing is needed.

For precise timing purposes, knowledge in assembly language programming is needed. To be more precise, the knowledge in the number of machine cycles needed to execute an instruction. A machine cycle in 8051 series lasts 12 clock cycles. A clock cycle is determined by the crystal connected to the pins XTAL1 and XTAL2. The number of clock cycles needed to achieve a specific frequency of samples is the machine cycle frequency (clock divided by 12) divided by the wanted frequency. The specific values for each note are shown in Table II.

TABLE II
MACHINE CYCLES COUNT

Note	Frequency	Sample Frequency	Machine Cycle Count (12MHz)	Machine Cycle Count (18.432MHz)
C_4	261.63	33488,64	30	46
$C^{\#}_4/D^b_4$	277.18	35479,04	28	43
D_4	293.66	37588,48	27	41
$D^{\#}_4/E^b_4$	311.13	39824,64	25	39
E_4	329.63	42192,64	24	36
F_4	349.23	44701,44	22	34
$F^{\#}_4/G^b_4$	369.99	47358,72	21	32
G_4	392.00	50176	20	31
$G^{\#}_4/A^b_4$	415.30	53158,4	29	29
A_4	440.00	56320	18	27
$A^{\#}_4/B^b_4$	466.16	59668,48	17	26
B_4	493.88	63216,64	16	24
C_5	523.25	66976	15	23

As can be seen in Table II, when using a 12MHz crystal, most of the notes differ in only one machine cycle. This cannot be achieved by a timer. The 18.432MHz crystal provides a bigger difference in the number of machine cycles between different frequencies, so that is the reason why the author chose that crystal.

Writing the complete program in assembly language is not necessary, because the environment (Keil uvision3) generates the assembly language code itself during the compilation process. The while loop that loads the samples and drives them is simple, and can be easily written in C code:

```
…
while(!P0_0){ //C note!
    if (i>127)
        i=0;
    else {
        //DELAY, so that else can last the same as if
    }
    P1 = signal[i++];
    //DELAY
}
```

The Keil compiler compiles and creates the assembly language code shown in Table III.

TABLE III
ASM CODE

Label	ASM Code	Machine Cycles
C: 0114	JB P0_0(0x80), C: 00D2	2
	MOV A, i(0x08)	1
	SETB C	1
	SUBB A, #0x7F	1
	JC C: 00E2	2
IF TRUE 1	CLR A	1
IF TRUE 2	MOV i(0x08), A	1
IF TRUE 3	SJMP C:0127	2
ELSE 1	NOP	1
ELSE 2	NOP	1
ELSE 3	NOP	1
ELSE 4	NOP	1
C: 0127	MOV R7, i(0x08)	2
	INC i(0x08)	1
	MOV A, R7	1
	MOV DPTR, #signal(0x008F)	2
	MOVC A, @A+DPTR	2
	MOV P1(0x90), A	2
	;DELAY IN ASM	???
	SJMP C:0114	2

In order to make the specific loop last the same number of machine cycles whether the IF statement is true or false, NOP instructions are added in the ELSE block. The total number of machine cycles per iteration is 23 (Either the IF TRUE block or the ELSE block is executed). In order to achieve specific frequencies, new machine cycles that do not corrupt the flow of the program must be added to each iteration. The perfect example of this is the NOP instruction, which lasts one machine cycle. In addition, a new loop is needed in order to prevent writing a large number of written NOP instructions. The loop which results in a smallest number of assembly instructions is a properly used do while loop:

```
unsigned char p = 11;
do{

    _nop_ ();
    _nop_ ();

while(--p);
```

This loop is compiled into the assembly language code shown in Table IV.

TABLE IV
LOOP ASM CODE

Label	ASM Code	Machine Cycles
	MOV p(0x09), #0x0B	2
C: 0138	NOP	1
	NOP	1
	DJNZ p(0x09), C: 138	2

The assembly language code for the preparation for the loop takes 2 machine cycles to execute. An iteration of the loop lasts 4 machine cycles. The whole code then, without the delay loop, lasts 25 cycles.

Using the information provided in Tables III and IV, we can calculate the initial value of the variable p for each musical note. Not all numbers from the last column in Table II are divisible by 3, so sometimes additional NOP instructions are needed before the loop. In addition, in some cases the note loop lasts more machine cycles than needed, so there are negative values of additional NOP instructions, and the loop must be shortened. The shortening can be done by eliminating the preparation for the delay loop (p = initial_p_value) and the delay loop, as they are not needed. Calculated initial values for p and additional NOP numbers are shown in Table V.

TABLE V
INITIAL P VALUES AND ADDITIONAL NOP NUMBERS

Note	Initial p value	Additional NOP number
C_4	5	1
$C^{\#}_4/D^b_4$	4	2
D_4	4	0
$D^{\#}_4/E^b_4$	3	2
E_4	2	3
F_4	2	1
$F^{\#}_4/G^b_4$	1	3
G_4	1	2
$G^{\#}_4/A^b_4$	1	0
A_4	0	2
$A^{\#}_4/B^b_4$	0	1
B_4	0	-1
C_5	0	-2

The C code for the CA51 compiler that implements the wanted functionality can be found in appendix C.

VIII. AT89S51 CIRCUIT

A basic AT89S51 circuit consists of a power-on reset circuit, a crystal oscillator circuit, power supplies and the EA pin connected to Vcc. The AT89S51 circuit for this project consists of 14 additional switches (push buttons), which have switch debouncing circuits (a simple RC circuit), and a port connected to a DAC. A schematic of the AT89S51 circuit used in this project is shown on Fig.3.

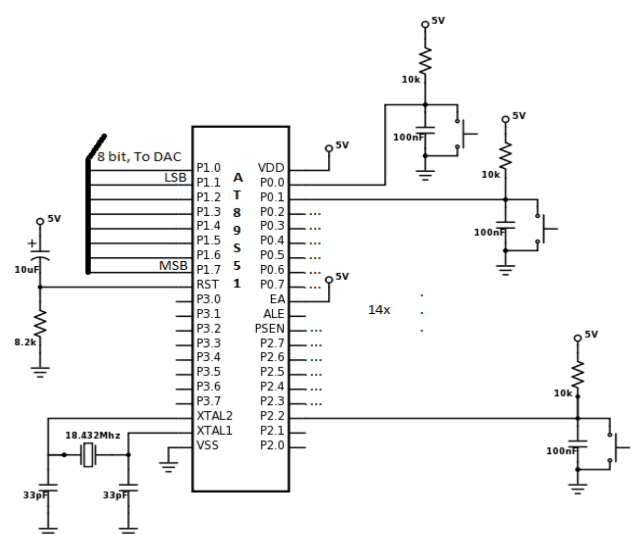

Figure 3. AT89S51 Circuit

IX. Interfacing AT89S51 with DAC

As previously mentioned, if the binary code of the samples stored in the memory of the microcontroller is the same as the code used by the DAC, the interface between the microcontroller and the DAC consists of two connected ports – the output port of the microcontroller is connected to the input port of the DAC.

The chosen DAC is the DAC 0800. It was chosen because of its simplicity and availability on the market. It is an 8 bit parallel DAC which can use a number of binary codes, including the shifted binary code. The DAC interface circuit, as well as the interface circuit with the AT89S51 microcontroller is shown on Fig. 4.

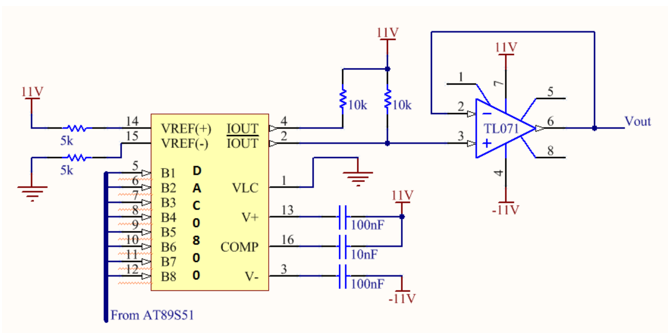

Figure 4. DAC 0800 Interface Circuit

X. Amplifying and Driving the Signal on Speakers

The signal at the output of the DAC interface circuit is buffered and has relatively high amplitude, and can be driven directly to an 8 Ohm speaker. However, to achieve better quality and loudness, an audio amplifier is desirable. Such an amplifier is the LM386 audio amplifier, and it is used in this project. At the output of the DAC interface circuit, and before the input of the audio amplifier is a basic RC circuit, used as a low pass analog filter, to prevent high frequency noise to pass to the input of the amplifier. The audio amplifier with its interface, which includes a loudspeaker on which the audio signal is driven, is shown in Fig 5.

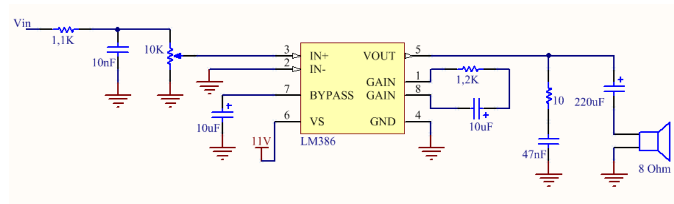

Figure 5. Audio Amplifier Circuit

References

[1] I. S. Stojanovic, *Osnovi telekomunikacija,* 2nd ed. Beograd, Gradjevinska knjiga, 1973.

[2] Rastislav Struharik, *Diskretni sistemi: racunarske vezbe u MATLAB-u,* Novi Sad, Fakultet tehnickih nauka, Novi Sad, 2014.

[3] www.elektronika.ftn.uns.ac.rs, 'Razvoj Softvera za Digitalne Mikrokontrolere', 2016. [Online] Available: www.elektronika.ftn.uns.ac.rs/index.php?option=com_content&task=category&sectionid=4&id=184&Itemid=199 [Accessed: 20.09.2016].

[4] www.keil.com, '8051 Instruction Set Manual', 2016. [Online] Available:www.keil.com/support/man/docs/is51_overview.htm [Accessed: 20.09.2016].

[5] Atmel, *8-bit Microcontroller with 4K Bytes In-System Programmable Flash,* AT89S51 datasheet, 2008.

[6] Texas Instruments, *DAC0800/DAC0802 8-Bit Digital-to-Analog Converters,* DAC0800 datasheet, 1999 [Revised Feb. 2013].

[7] Texas Instruments, *TL07xx Low-Noise JFET-Input Operational Amplifiers,* TL071 datasheet, Sept. 1978 [Revised June 2015].

[8] Texas Instruments, *LM386 Low Voltage Audio Power Amplifier*, LM386 datasheet, Aug. 2000.

APPENDIX A – MATLAB SIGNAL SAMPLING CODE

```
clear all;
clc;
%%%%%%%%%%%%%%%%%%%%RZSDMK%%%%%%%%%%%%%%%%%%%%%%%

%DEFINING THE SIGNAL FOR ROM MEMORY

%Amplitude
A=0.5;
%Frequency, only for Matlab
fo=440;
fs=128*fo;
%number of signal samples that will be written into ROM, it has to satisfy
%the sampling theorem, in this case fs=128*fo, and the maximal frequency in
%the spectrum is 9*fo, hence fs>2*9*fo, so the sampling theorem is
%satisfied
n = 0:127;

x = A*sin(2*pi*(fo/fs)*n)+A/2*sin(2*pi*2*(fo/fs)*n)+A/3*sin(2*pi*3*(fo/fs)*n)+A/4*sin(2*pi*4*(fo/fs)*n)+...
  A/5*sin(2*pi*5*(fo/fs)*n)+A/6*sin(2*pi*6*(fo/fs)*n)+A/7*sin(2*pi*7*(fo/fs)*n)+A/8*sin(2*pi*8*(fo/fs)*n)+...
  A/9*sin(2*pi*9*(fo/fs)*n);

figure;
plot(x);

figure;
stem (0:127, x); %0:15

a= max(x);

save('signal.txt', 'x', '-ASCII', '-append');

```

APPENDIX B – QUANTIZER AND CODER C CODE

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define SAMPLE_NUMBER 128 //16

struct quantization_info{

    int level_number;
    double *level;
    double level_amplitude;
    double decision_amplitude;
    double max_amplitude;
    int bit_number;

};

struct quantized_signal{

    double samples_dec[SAMPLE_NUMBER];
    int samples_hex[SAMPLE_NUMBER];

};

//FUNCTION THAT CREATES BASIC INFO ABOUT THE QUANTIZATION PROCESS
void quant_info_calculator(struct quantization_info *quant_info, double max_amplitude, int bit_number) {

    quant_info->level_number = pow(2, bit_number)-1; //because of the simetry aroud zero
    quant_info->level = (double *)malloc((quant_info->level_number)*sizeof(double));
    //the whole amplitude range divided by the number of levels minus one (because it is the distance between levels)
    quant_info->level_amplitude = (max_amplitude-(-max_amplitude))/((quant_info->level_number) - 1);
    quant_info->decision_amplitude = (quant_info->level_amplitude)/2;
    quant_info->max_amplitude = max_amplitude;
    quant_info->bit_number = bit_number;

};

//FUNCTION THAT CALCULATES THE QUANTIZATION LEVELS
void level_calculator (struct quantization_info *quant_info) {

    int i=0;
    FILE* levels_file;
    printf("level amplitude = %.18lf\n", (quant_info->level_amplitude));
    quant_info->level[0]=-(quant_info->max_amplitude);
    for (i=1; i<(quant_info->level_number); i++) {

        quant_info->level[i] = quant_info->level[i-1] + quant_info->level_amplitude;

    }

    levels_file = fopen("levels.txt", "w");
    if(levels_file==NULL) {

        printf("Error during the opening of file!\n");

    }
    for (i=0; i<quant_info->level_number; i++) {

        fprintf(levels_file, "level[%d]=%.18lf\n", i, quant_info->level[i]);
        fprintf(levels_file, "level[%d] in hex: %x\n", i, i);

    }

    fclose(levels_file);

}
```

```
//FUNCTION THAT QUANTIZES AND CODES THE SAMPLES (CODE - MOVED BINARY CODE)
void quantization_and_coding (struct quantized_signal *signal_q, double *signal, struct quantization_info *quant_info)

    int i, j;

    for(i=0; i<SAMPLE_NUMBER; i++)
        for(j=0; j<quant_info->level_number; j++) {

            if(signal[i] <= (quant_info->level[j] + quant_info->decision_amplitude)){

                signal_q->samples_dec[i] = quant_info->level[j];
                signal_q->samples_hex[i] = j;
                break;

            }

        }

};

//FUNCTION THAT LOADS THE SIGNAL SAMPLES FROM A FILE
void signal_loader (double * signal) {

    FILE *signal_file;
    int i;

    signal_file = fopen("signal.txt", "r");

    for(i = 0; i < SAMPLE_NUMBER; i++){

        fscanf(signal_file, "%lf", &signal[i]);
        //printf("Signal[%d]=%.15f\n", i, signal[i]);

        if(i>=SAMPLE_NUMBER) {
            printf("Too much signal samples!\n");
            break;

        }
    }

    fclose(signal_file);

}

//FUNCTION THAT PRINTS QUANTIZED SAMPLES INTO A FILE
void quantized_signal_printer (struct quantized_signal *signal_q) {

    FILE* quantized_signal_file;

    quantized_signal_file = fopen("quantized_signal.txt", "w");

    if(quantized_signal_file==NULL) {

        printf("Error during the opening of file!\n");

    }

    int i;

```

```
    for (i=0; i<SAMPLE_NUMBER; i++) {

        fprintf(quantized_signal_file, "quantized_sample[%d]=%.18lf\n", i, signal_q->samples_dec[i]);
        fprintf(quantized_signal_file, "quantized_sample[%d] in hex: %x\n", i, signal_q->samples_hex[i]);

    }

    fclose(quantized_signal_file);


}

int main()
{
    struct quantization_info *q_info = malloc(sizeof(struct quantization_info));
    quant_info_calculator(q_info, 0.842982750824474, 8);
    level_calculator(q_info);

    double signal[SAMPLE_NUMBER];
    signal_loader(signal);

    struct quantized_signal *signal_quant = malloc(sizeof(struct quantized_signal));
    quantization_and_coding(signal_quant, signal, q_info);

    quantized_signal_printer(signal_quant);

    free(q_info);
    free(signal_quant);

    return 0;
}

```

APPENDIX C – AT89S51 C CODE

```
#include <REGx52.h>
#include <intrins.h> //FOR NOP

#define C      P0_0
#define CIS    P0_1
#define D      P0_2
#define DIS    P0_3
#define E      P0_4
#define F      P0_5
#define FIS    P0_6
#define G      P0_7
#define GIS    P2_7
#define A      P2_6
#define AIS    P2_5
#define B      P2_4
#define H      P2_3
#define C2     P2_2
#define output P1

unsigned char i=0;
unsigned char p=0;
code unsigned char signal[128] = {0x7f, 0xa0, 0xbe, 0xd8, 0xec, 0xf8, 0xfe, 0xfd, 0xf8, 0xf0, 0xe6, 0xde,
0xd6, 0xd2, 0xd0, 0xd1, 0xd3, 0xd6, 0xd8, 0xd9, 0xd8, 0xd6, 0xd2, 0xcd, 0xc8, 0xc3, 0xc0, 0xbd, 0xbc,
0xbc, 0xbd, 0xbe, 0xbe, 0xbd, 0xbc, 0xb9, 0xb5, 0xb1, 0xad, 0xaa, 0xa7, 0xa5, 0xa4, 0xa4, 0xa4,
0xa4, 0xa3, 0xa1, 0x9e, 0x9a, 0x96, 0x93, 0x90, 0x8d, 0x8c, 0x8c, 0x8b, 0x8b, 0x8b, 0x8a, 0x88, 0x86,
0x83, 0x7f, 0x7b, 0x78, 0x76, 0x74, 0x73, 0x73, 0x73, 0x72, 0x72, 0x71, 0x6e, 0x6b, 0x68, 0x64, 0x60,
0x5d, 0x5b, 0x5a, 0x5a, 0x5a, 0x5a, 0x5a, 0x59, 0x57, 0x54, 0x51, 0x4d, 0x49, 0x45, 0x42, 0x41, 0x40,
0x40, 0x41, 0x42, 0x42, 0x41, 0x3e, 0x3b, 0x36, 0x31, 0x2c, 0x28, 0x26, 0x25, 0x26, 0x28, 0x2b, 0x2d,
0x2e, 0x2c, 0x28, 0x20, 0x18, 0x0e, 0x06, 0x01, 0x00, 0x06, 0x12, 0x26, 0x40, 0x5e};

void Initialization(void) {

    P0=0xFF; // so that loading from this port could be possible
    P2=0xFF; // so that loading from this port could be possible

}

void main(void) {

    Initialization();

    while (1) {

        i=0;

        while (!C) {//asserted low!!!
                //The while loop lasts for 25 machine cycles, and for the C note and
                //fosc=18.432MHz => time before two samples has to be 46 machine cycles
                //       the NOP loop has to last 21 machine cycle


                if(i>127)
                    i=0;
                else {
                    _nop_();
```

```
                _nop_();
                _nop_();
                _nop_();
            }
            output = signal[i++];

            //NOP, MOV - ONE MACHINE CYCLE
            //DJNZ - TWO MACHINE CYCLES
            //46=25+21
            //21=1+4*5

            p = 5;

            _nop_ ();

            do {

                _nop_ ();
                _nop_ ();

            }while(--p);

    }


    while (!CIS) {//asserted low!!!
            //The while loop lasts for 25 machine cycles, and for the CIS note and
            //fosc=18.432MHz => time before two samples has to be 43 machine cycles
            //      the NOP loop has to last 18 machine cycles


            if(i>127)
                i=0;
            else {
                _nop_();
                _nop_();
                _nop_();
                _nop_();
            }
            output = signal[i++];

            //NOP, MOV - ONE MACHINE CYCLE
            //DJNZ - TWO MACHINE CYCLES
            //43=25+18
            //18=2+4*4

            p = 4;

            _nop_ ();
            _nop_ ();

            do {

                _nop_ ();
                _nop_ ();

            }while(--p);                                          : the CIS note and
                                                                  ? 43 machine cycles
    }

```

```
        while (!D) {//asserted low!!!
                //The while loop lasts for 25 machine cycles, and for the D note and
                //fosc=18.432MHz => time before two samples has to be 41 machine cycles
                //        the NOP loop has to last 16 machine cycles


                if(i>127)
                    i=0;
                else {
                    _nop_();
                    _nop_();
                    _nop_();
                    _nop_();
                }
                output = signal[i++];

                //NOP, MOV - ONE MACHINE CYCLE
                //DJNZ - TWO MACHINE CYCLES
                //41=25+16
                //16=4*4

                p = 4;

                do {

                    _nop_ ();
                    _nop_ ();

                }while(--p);

        }

        while (!DIS) {//asserted low!!!
                //The while loop lasts for 25 machine cycles, and for the DIS note and
                //fosc=18.432MHz => time before two samples has to be 39 machine cycles
                //        the NOP loop has to last 14 machine cycles


                if(i>127)
                    i=0;
                else {
                    _nop_();
                    _nop_();
                    _nop_();
                    _nop_();
                }
                output = signal[i++];

                //NOP, MOV - ONE MACHINE CYCLE
                //DJNZ - TWO MACHINE CYCLES
                //39=25+14
                //14=2+3*4

                p = 3;

                _nop_ ();
                _nop_ ();

                do {

```

```
                    _nop_ ();
                    _nop_ ();

                }while(--p);

        }

        while (!E) {//asserted low!!!
                //The while loop lasts for 25 machine cycles, and for the E note and
                //fosc=18.432MHz => time before two samples has to be 36 machine cycles
                //      the NOP loop has to last 11 machine cycles


                if(i>127)
                    i=0;
                else {
                    _nop_();
                    _nop_();
                    _nop_();
                    _nop_();
                }
                output = signal[i++];

                //NOP, MOV - ONE MACHINE CYCLE
                //DJNZ - TWO MACHINE CYCLES
                //36=25+11
                //11=3+2*4

                p = 2;

                _nop_ ();
                _nop_ ();
                _nop_ ();

                do {

                    _nop_ ();
                    _nop_ ();

                }while(--p);

        }

        while (!F) {//asserted low!!!
                //The while loop lasts for 25 machine cycles, and for the F note and
                //fosc=18.432MHz => time before two samples has to be 34 machine cycles
                //      the NOP loop has to last 9 machine cycles


                if(i>127)
                    i=0;
                else {
                    _nop_();
                    _nop_();
                    _nop_();
                    _nop_();
                }
                output = signal[i++];

                //NOP, MOV - ONE MACHINE CYCLE
```

```
            //DJNZ - TWO MACHINE CYCLES
            //34=25+9
            //9=1+2*4

            p = 2;

            _nop_ ();

            do {

                _nop_ ();
                _nop_ ();

            }while(--p);

        }

        while (!FIS) {//asserted low!!!
            //The while loop lasts for 25 machine cycles, and for the G note and
            //fosc=18.432MHz => time before two samples has to be 32 machine cycles
            //       the NOP loop has to last 7 machine cycles


            if(i>127)
                i=0;
            else {
                _nop_();
                _nop_();
                _nop_();
                _nop_();
            }
            output = signal[i++];

            //NOP, MOV - ONE MACHINE CYCLE
            //DJNZ - TWO MACHINE CYCLES
            //32=25+7
            //7=3+1*4

            p = 1;

            _nop_ ();
            _nop_ ();
            _nop_ ();

            do {

                _nop_ ();
                _nop_ ();

            }while(--p);

        }

        while (!G) {//asserted low!!!
            //The while loop lasts for 25 machine cycles, and for the G note and
            //fosc=18.432MHz => time before two samples has to be 31 machine cycles
            //       the NOP loop has to last 6 machine cycles


            if(i>127)
```

```
                i=0;
            else {
                _nop_();
                _nop_();
                _nop_();
                _nop_();
            }
            output = signal[i++];

            //NOP, MOV - ONE MACHINE CYCLE
            //DJNZ - TWO MACHINE CYCLES
            //31=25+6
            //6=2+1*4

            p = 1;

            _nop_ ();
            _nop_ ();

            do {

                _nop_ ();
                _nop_ ();

            }while(--p);

        }

        while (!GIS) {//asserted low!!!
            //The while loop lasts for 25 machine cycles, and for the GIS note and
            //fosc=18.432MHz => time before two samples has to be 29 machine cycles
            //      the NOP loop has to last 4 machine cycles


            if(i>127)
                i=0;
            else {
                _nop_();
                _nop_();
                _nop_();
                _nop_();
            }
            output = signal[i++];

            //NOP, MOV - ONE MACHINE CYCLE
            //DJNZ - TWO MACHINE CYCLES
            //29=25+4
            //4=0+1*4

            p = 1;

            do {

                _nop_ ();
                _nop_ ();

            }while(--p);

        }

```

```
        while (!A) {//asserted low!!!
                //The while loop lasts for 25 machine cycles, and for the A note and
                //fosc=18.432MHz => time before two samples has to be 27 machine cycles
                //        the NOP loop has to last 2 machine cycles


                if(i>127)
                    i=0;
                else {
                    _nop_();
                    _nop_();
                    _nop_();
                    _nop_();
                }
                output = signal[i++];

                //NOP, MOV - ONE MACHINE CYCLE
                //DJNZ - TWO MACHINE CYCLES
                //27=23+2+2
                //2+2=1+1+1+1

                //two nops instead of p=0;
                _nop_ ();
                _nop_ ();

                _nop_ ();
                _nop_ ();

                //no do while loop

        }

        while (!AIS) {//asserted low!!!
                //The while loop lasts for 25 machine cycles, and for the AIS note and
                //fosc=18.432MHz => time before two samples has to be 26 machine cycles
                //        the NOP loop has to last 1 machine cycles


                if(i>127)
                    i=0;
                else {
                    _nop_();
                    _nop_();
                    _nop_();
                    _nop_();
                }
                output = signal[i++];

                //NOP, MOV - ONE MACHINE CYCLE
                //DJNZ - TWO MACHINE CYCLES
                //26=23+2+1
                //2+1=1+1+1

                //two nops instead of p=0;
                _nop_ ();
                _nop_ ();

                _nop_ ();

                //no do while loop
```

```

        }

        while (!AIS) {//asserted low!!!
                //The while loop lasts for 25 machine cycles, and for the B note and
                //fosc=18.432MHz => time before two samples has to be 24 machine cycles
                //       the NOP loop has to last -1 machine cycles


                if(i>127)
                    i=0;
                else {
                    _nop_();
                    _nop_();
                    _nop_();
                    _nop_();
                }
                output = signal[i++];

                //NOP, MOV - ONE MACHINE CYCLE
                //DJNZ - TWO MACHINE CYCLES
                //24=23+1
                //1=1

                //one nop instead of p=0; -> 2-1=1
                _nop_ ();

                //no do while loop

        }

        while (!C2) {//asserted low!!!
                //The while loop lasts for 25 machine cycles, and for the C2 note and
                //fosc=18.432MHz => time before two samples has to be 23 machine cycles
                //       the NOP loop has to last -2 machine cycles


                if(i>127)
                    i=0;
                else {
                    _nop_();
                    _nop_();
                    _nop_();
                    _nop_();
                }
                output = signal[i++];

                //NOP, MOV - ONE MACHINE CYCLE
                //DJNZ - TWO MACHINE CYCLES
                //23=23+0

                //no nops instead of p=0; 2-2=0

                //no do while loop

        }

    };

}
```