

SIMPLIFIED ADPCM DESIGN AND SIMULATION FOR PORTABLE SPEECH  
RECORDING DEVICES

Bhagirathsinh Aniruddhsinh Jadeja  
B.E., Saurashtra University, India, 2007

PROJECT

Submitted in partial satisfaction of  
the requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL AND ELECTRONIC ENGINEERING

at

CALIFORNIA STATE UNIVERSITY, SACRAMENTO

SPRING  
2011

SIMPLIFIED ADPCM DESIGN AND SIMULATION FOR PORTABLE SPEECH  
RECORDING DEVICES

A

Project

by

Bhagirathsinh Aniruddhsinh Jadeja

Approved by:

\_\_\_\_\_, Committee Chair  
Preetham Kumar, Ph. D.

\_\_\_\_\_  
Date

Student: Bhagirathsinh Aniruddhsinh Jadeja

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the Project.

\_\_\_\_\_, Department Chair  
Suresh Vadhva, Ph. D.

\_\_\_\_\_  
Date

Department of Electrical and Electronic Engineering

Abstract  
of  
**SIMPLIFIED ADPCM DESIGN AND SIMULATION FOR PORTABLE SPEECH  
RECORDING DEVICES**

Adaptive Delta Pulse Code Modulation is a well known encoding scheme used for speech processing. This project focuses on the simplification in the technique so that the hardware complexity can be reduced for the portable speech recording devices. The reduce hardware complexity also provides added advantages like low power consumption and portability.

The simulation of the suggested simplification in the algorithm is done using the MATLAB tool. This project focuses on .wav format audio file and encoding and decoding using ADPCM technique is done. This project provides practical comparison by comparing the actual audio file with decoded audio file and their respective graphical representations.

The hardware implementation of speech recording and playback is done using Atmel STK500 kit and ATMEGA32 microcontroller. The ADC and PWM schemes are used to interface analog signals to the core digital hardware. To conclude, this project tries to provide insight of algorithm and hardware that can be used for the portable speech recording playback devices.

\_\_\_\_\_, Committee Chair  
Preetham Kumar, Ph. D.

\_\_\_\_\_  
Date

## ACKNOWLEDGEMENTS

This project was successfully completed due to guidance and blessings of many persons. I shall start with Dr Preetham Kumar, Professor and Graduate Coordinator of Electrical and Electronics Engineering department in College of Engineering and Computer Science. He was of great help to make this project possible. If his constant guidance and motivation was not there, accomplishment of this project would not have been possible.

I shall also mention my project-mate and good friend Bhavin Gandhi, with whom I spent restless hours in lab to make the hardware work. I shall also mention my roommate Romin Shah, with whom I have had number of fruitful discussion that contributed towards success of this project.

And last but not the least, I will like to acknowledge the emotional support and blessings of my parents that helped me remain strong and focused towards achievement of my goals.

## TABLE OF CONTENTS

Acknowledgements.....	v
List of Tables .....	viii
List of Figures .....	ix
Chapter	
1. INTRODUCTION .....	1
1.1 Goal of the Project .....	1
1.2 Pulse Code Modulation.....	2
1.3 Differential Pulse Code Modulation .....	3
1.4 Atmel STK500 Kit.....	4
2. ADAPTIVE DELTA PULSE CODE MODULATION .....	5
2.1 ADPCM Encoder .....	6
2.2 ADPCM Decoder.....	7
2.3 Step Size Calculation .....	8
3. ADPCM SIMULATION USING MATLAB AND C LANGUAGE .....	11
3.1 ADPCM Encoding Algorithm .....	12
3.2 ADPCM Decoding Algorithm .....	13
3.3 Simulation Results .....	14
3.3.1 Raw File.....	14
3.3.2 File Encoded with ADPCM Algorithm .....	16
3.3.3 File Decoded with ADPCM Algorithm .....	16
3.3.4 Comparison of RAW File and Decoded File .....	18
3.4 C Language Implementation.....	19
4. HARDWARE DESIGN.....	20
4.1 Pre-amplification .....	20
4.2 Programming with STK500.....	22
4.2.1 Initialization.....	22
4.2.2 Recording.....	24
4.2.3 Playback.....	27
4.2.4 Erase .....	28
4.4 PWM Filter .....	29

5. CONCLUSION AND FUTURE WORK .....	31
Appendix A: MATLAB Codes .....	32
A.1 MATLAB Commands .....	32
A.2 Encoder .....	32
A.3 Decoder .....	36
Appendix B: C Language Codes.....	39
B.1 Encoder .....	39
B.2 Decoder .....	45
References .....	50

## LIST OF TABLES

Table 2-1: M (L (n) values) .....	9
Table 2-2: Calculated Step Size .....	9



## LIST OF FIGURES

Figure 1-1: PCM Encoder .....	2
Figure 1-2: PCM Decoder.....	2
Figure 1-3: DPCM Encoder .....	3
Figure 2-1: ADPCM Encoding Algorithm.....	6
Figure 2-2: ADPCM Decoding Algorithm .....	8
Figure 3-1: Raw .wav File .....	15
Figure 3-2: Zoomed Raw File.....	15
Figure 3-3: ADPCM Encoded File .....	16
Figure 3-4: Decoded File .....	17
Figure 3-5: Zoomed Decoded File .....	17
Figure 3-6: Raw Samples v/s Decoded Samples .....	18
Figure 4-1: Microphone Pre-amplifier Circuit.....	21
Figure 4-2: Initialization Algorithm (part 1).....	23
Figure 4-3: Initialization Algorithm (part 2).....	24
Figure 4-4: Recording Algorithm (part 1).....	25
Figure 4-5: Recording Algorithm (part 2).....	26
Figure 4-6: Playback Algorithm .....	27
Figure 4-7: Erase Algorithm .....	28
Figure 4-8: Analog Filter Circuit .....	29

## Chapter 1

### INTRODUCTION

#### 1.1 Goal of the Project

Data compression along with analog to digital conversion is an important factor in data communication and data storage. Digital format provides conventional benefits like lower noise and computer friendliness, while compression saves bandwidth for the same SNR (Signal to Noise Ratio). The Adaptive Delta Pulse Code Modulation is a technique that gives above mentioned advantages along with Excellent Speech Quality, 4:1 or 8:1 compression ratios. The ADPCM is simple to implement when compared to LPC or MP3 coding scheme and that also makes ADPCM circuit size and power effective. ADPCM is ideal for portable speech recording devices where speech quality and hardware complexity are major issues.

In this project, I will demonstrate the MATLAB simulation of simplified ADPCM encoding and decoding algorithms. I will also demonstrate the C language implementation of ADPCM technique to demonstrate lower bit errors that may appear, while using the technique. In the end I will demonstrate the speech recording and playback on the using ATMEGA32 microcontroller that is mounted on Atmel STK500 kit. I have used AVR32 Studio 2.1 to generate the object file to be loaded on the ATMEGA32.

Before we go in the details of ADPCM and the simplification that I have made to achieve a compact ADPCM encoder and decoder, let's look at the PCM, DPCM and Atmel STK500 Kit in brief.

## 1.2 Pulse Code Modulation

In 1937, Alec Reeves invented a method to convert analog signals of sinusoidal nature in to digital pulse stream. The analog signal is first sampled at a frequency greater than Nyquist frequency. The samples are than assigned a discrete value by Quantization Unit. Finally those discrete samples are coded in to RZ (Return to Zero) or NRZ (Non-Return to Zero) binary stream by Binary Encoder. Following are the block level representations of ADPCM encoder and decoder.

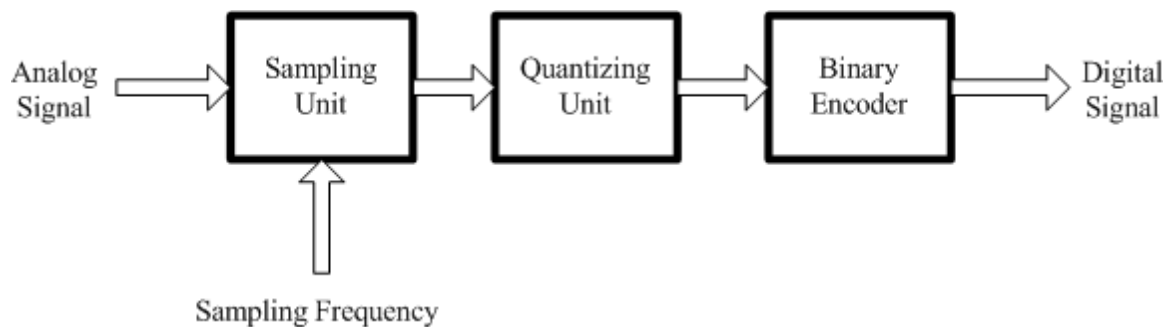


Figure 1-1: PCM Encoder

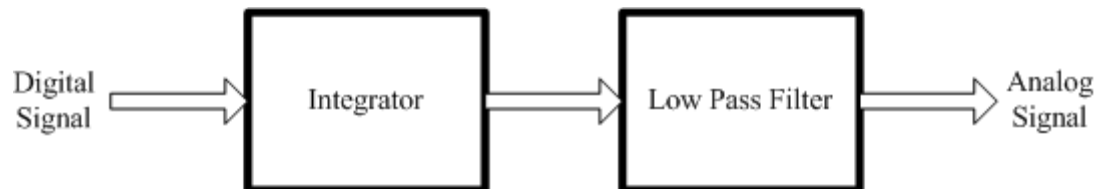


Figure 1-2: PCM Decoder

The quantization error and significantly large bandwidth are two major drawbacks of PCM systems. The former can be moderated by using smaller step size, which will eventually increase the bandwidth of the system. The later can be improved by using more complex algorithms like DPCM and ADPCM.

### 1.3 Differential Pulse Code Modulation

Alternatively, we can take difference between two consecutive samples and transmit the binary code for the difference to save some bandwidth. The saved bandwidth comes with added complexity in the hardware. Following is the Block level representation of ADPCM encoder.

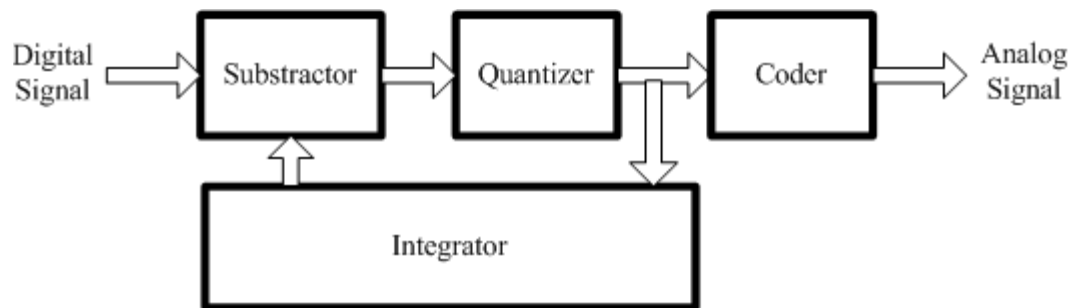


Figure 1-3: DPCM Encoder

The ADPCM decoder will look like the PCM decoder; accept for the decoder block that will provide signal to integrator. The bandwidth can be further saved by adjusting the step-size as explained the following chapters on ADPCM. Now let's look at the hardware components of the project.

## 1.4 Atmel STK500 Kit

The STK500 Kit comes with the default AT90S8515-8PC microcontroller that is the heart of the kit. The microcontroller stores the code that eventually makes the kit work the way the programmer wants it to. The kit requires an external 10-15V DC power supply. The kit can be configured for different clock frequencies and power. The kit can be connected to a computer through a serial RS232 port. Since the port has almost become absolute in all the modern day computer motherboards, I have used a USB to RS232 connector for the interface between the kit and computer. The kit also comes with added components like input switches and LEDs for better control and observation capabilities.

Applications for the STK500 kit can be developed and debugged in an Integrated Development Environment named AVR Studio. I have used the AVR Studio (Version 2.1) for the development of the code that controls recording and playback on the kit. I have used Pulse Width Modulation scheme to convert analog signal to digital signal. It will be explained in depth in the coming chapters about the hardware design.

## Chapter 2

### ADAPTIVE DELTA PULSE CODE MODULATION

ADPCM takes advantage of constantly changing quantization step size to further reduce the required bandwidth. The compression ratios up to 8:1 are possible with this algorithm with excellent speech quality and high audio quality. We will use a simplified version of algorithm with compression ratio of 2:1 for our project. When compared to DPCM with adequate speech and average audio quality and PCM that has very high audio and speech quality, ADPCM comes with advantage of reduced bandwidth, although the saved bandwidth comes with the added hardware complexity. In the modern era the bandwidth is becoming very critical issue and at the same time semiconductor circuits are getting denser and cheap, ADPCM becomes an obvious choice particularly for the application where audio quality can be compromised. Other comparable algorithms that can be used for voice or audio compression are PCM, DPCM, LPC and MP3. ADPCM is used widely throughout the telecommunication industry. VoIP also takes advantage of this technique.

It works by calculating the difference between two consecutive samples in standard pulse code modulation (PCM) and codes the error of the 'predicted' next sample increment (from the previous sample increment) to the true sample increment. Binary code is normally used to encode the prediction error. The ADPCM can be used along with TDM (Time Division Multiplexing) to take advantage of the saved bandwidth. The following section explains ADPCM encoding and decoding block diagram.

## 2.1 ADPCM Encoder

ADPCM compression starts with comparing present sample to the estimated value of the previous sample. The difference is then presented to encoder logic along with the step size for the present sample, estimated from the encoded data of the previous sample. Estimation of the previous sample is done by the decoder using the encoded data of the previous sample.

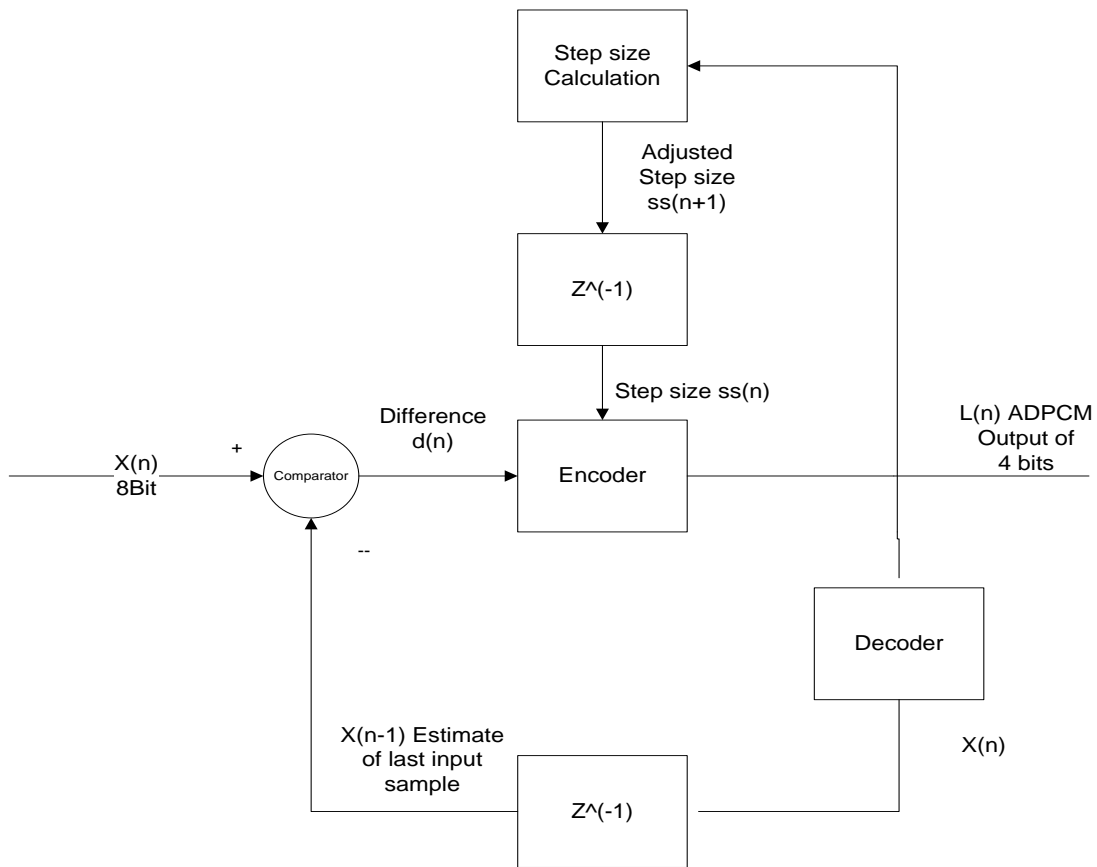


Figure 2-1: ADPCM Encoding Algorithm

8 bit sample  $X(n)$  is compared with the estimated value of the previous sample  $X(n-1)$ . Difference  $d(n)$  along with step size  $SS(n)$  is presented to the encoder logic. Encoder outputs 8bit data with only 4 LSB bits to be used for storing.

Step size  $SS(n)$  for  $X(n)$  is calculated from the encoded code for the previous sample  $X(n-1)$ . Similarly, estimated value of the previous sample is also calculated by decoding the encoded code of the previous sample. Full decoder is also present in the ADPCM encoder. This ensures that decoder and encoder are in synchronization without need to send any extra bits.

## 2.2 ADPCM Decoder

Decoder does reverse of what encoder does. Encoder encodes the difference between linear present sample and estimated previous sample by using estimated step size for the present sample obtained from previous encoded sample. Decoder decodes the encoded data into a difference present sample and estimated previous sample by using step size for the present sample obtained from previous sample exactly in the same way encoder uses it. Decoded value of previous sample is added in that difference of present sample and estimated previous sample to obtained decoded 8bit value of the present sample.



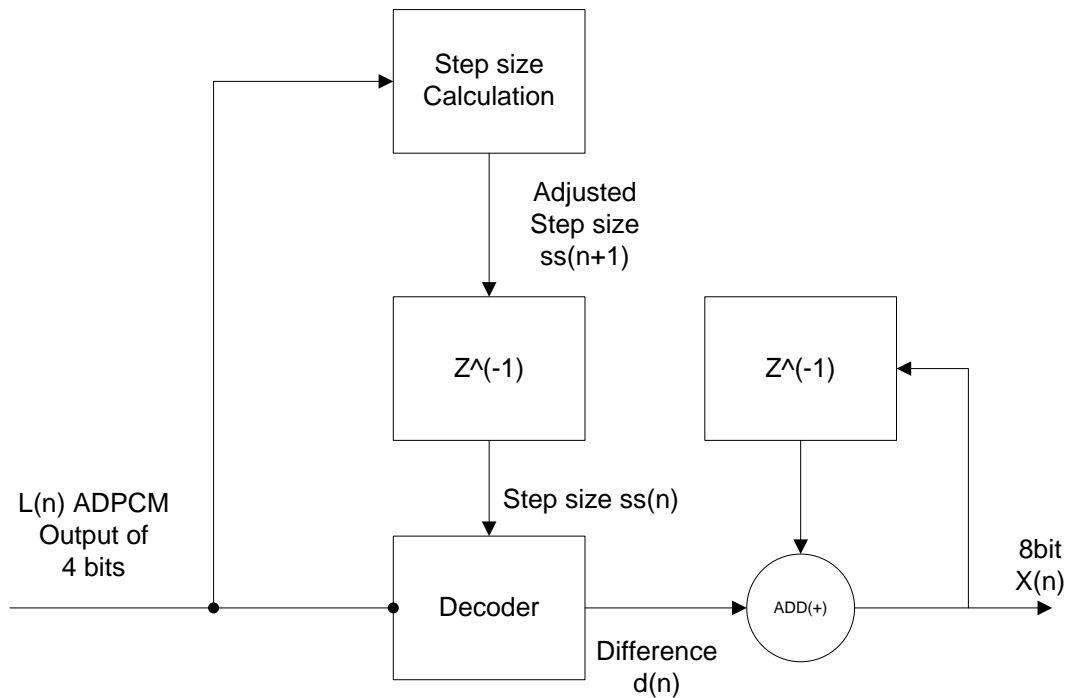


Figure 2-2: ADPCM Decoding Algorithm

As shown in the figure encoded ADPCM present sample  $L(n)$  is presented to decoder along with step size  $ss(n)$  which is calculated by using previous ADPCM sample. Decoder decodes the difference between present sample  $X(n)$  and previous sample  $X(n-1)$ . Decoder output is then added to the previous sample to obtain the 8 bit value of present sample  $X(n)$ .

### 2.3 Step Size Calculation

In encoding as well as decoding, the algorithm for ADPCM adjusts the step size of quantizer for the next sample based on the most recent ADPCM value. The following equation is used for the calculation of the step size for the next sample

$$stepsize_{(n+1)} = stepsize_n * 1.1 * M(L(n))$$

L(n)	Value	M (L(n))
1111	0111	+8
1110	0110	+6
1101	0101	+4
1100	0100	+2
1011	0011	-1
1010	0010	-1
1001	0001	-1
1000	0000	-1

Table 2-1 M (L (n) values)

Step Number	Step Size	Step Number	Step Size
1	7	18	31
2	8	19	34
3	8	20	37
4	9	21	41
5	10	22	44
6	11	23	48
7	12	24	53
8	13	25	58
9	14	26	63
10	15	27	69
11	17	28	75
12	18	29	82
13	20	30	89
14	22	31	98
15	24	32	107
16	26	33	116
17	29	34	127

Table 2-2 Calculated Step Size

The implementation of equation this can be done efficiently using lookup table of two stages. In first stage the index value of L (n) is determined using table and then it is added to the previous

value. Using that new index value, new step size is determined using table. Index value of less than 3 will result in decrease in step size while index value of greater than 3 will result in increase in step size.

This method for quantizing an adaptive step size can only be used for voice signal. When the ADPCM algorithm is reset, the step size  $SS(n)$  is set to the minimum value (7) and the estimated waveform value  $X$  is set to zero (half scale).

## Chapter 3

### ADPCM SIMULATION USING MATLAB AND C LANGUAGE

MATLAB is a software tool that can be used to simulate the behavior of ADPCM encoding and decoding process. The self sufficient functions for matrix operations and flexible plotting function made it an ideal tool for this project. I will also give you insight of the C language implantation in the later parts of this chapter.

For simulation and verification of ADPCM process, we will firstly record an audio file with sampling rate of 8 KHz and 8 bits per sample. We will then plot the graph of the audio signal with respect to time. Then the encoding function will take the audio file in form of matrices and will create an encoded file. That encoded file will be saved in a variable in MATLAB. We will use that encoded file as an input to a decode function that will eventually give us the decoded matrices. Hence we can compare the two matrices to see the effectiveness of the algorithm. We can also show the error in the decoded file when compared to original file and plot it.

But first let us look at the details of ADPCM encoding and decoding functions used for this project. Please note that, there can be number of different approaches to achieve the same goal of ADPCM encoding and decoding. This project is solely based on the approach explained in the following sections. The relevant MATLAB codes are provided in Appendix A.

### 3.1 ADPCM Encoding Algorithm

ADPCM algorithm used for the project can be explained by following steps [1].

- 1) Take an 8 bit sample
- 2) Reset values of predicted sample and quantizer step size index
- 3) Use the look-up table (Table 2-1 and Table 2-2) to find new step size
- 4) Calculate the difference between actual sample and predicted sample
- 5) Quantize the difference into ADPCM code, including sign bit
- 6) Inverse Quantize the ADPCM code into predicted difference using the step size calculated in step 3
- 7) Add old predicted sample and predicted difference to calculate the predicted sample value
- 8) Moderate the new predicted sample for the range of 127 to -128
- 9) Find the new quantizer step size index by adding the previous index and table look up based on ADPCM code
- 10) Moderate the new quantization step size index for the range of 0 to 34
- 11) Save the new predicted sample and step size index for the next iteration on next set of data
- 12) Return the 4-bit ADPCM code and go to step 1 till the end of new data samples

The process of encoding starts with accepting the 8-bit audio sample. The encoder then restores the predicted value of new sample and the new step size from the previous sample. After that it compares the actual sample value and its predicted sample value of the previous sample

and obtained the difference between two. Then it finds the absolute value of the difference and set the sign bit of encoded sample accordingly and encodes the difference using the step size. This encoded code is stored for retrieving the data after decoding. The encoded code is of four bit only. Three bits are used for the amplitude and the fourth bit as a sign bit. Now, encoder decodes the encoded code for the prediction of this sample for comparing it with the new sample for obtaining difference. The process of calculating the step size for the next sample while making sure it does not overflow follows after that. In the end the encoder function returns the 8-bit code with a 4-bit ADPCM value and upper 4 bits as 0000.

### 3.2 ADPCM Decoding Algorithm

The ADPCM decoding algorithm can be explained by following steps [1].

- 1) Take and 8 bit sample
- 2) Reset values of predicted sample and quantizer step size index
- 3) Use the look-up table (Table 2-1 and Table 2-2) to find new step size
- 4) Inverse Quantize the ADPCM code in to predicted difference using the step size calculated in step 3
- 5) Add old predicted sample and predicted difference to calculate the predicted sample value
- 6) Moderate the new predicted sample for the range of 127 to -128
- 7) Find the new quantizer step size index by adding the previous index and table look up based on ADPCM code
- 8) Moderate the new quantization step size index for the range of 0 to 34

- 9) Save the new predicted sample and step size index for the next iteration on next set of data
- 10) Return the 8-bit signed number and go to step 1 till the end of new data samples

### 3.3 Simulation Results

In this section, we will see the graphs that will show the raw file, encoded file and decoded file. We can compare the raw file to decoded file to estimate the error graphically. I have also plotted the raw file values v/s decoded file values to see an approximation of error. I have also attached a CD that has the raw .wav file and decoded .wav file to see the actual difference in audio quality.

#### 3.3.1 Raw File

The raw file is of wav extension. It has the sampling rate of 8 MHz, bit depth of 8 bit and it is mono. The following figures show the sample values against sample numbers. The second figure is zoomed version of the first one.

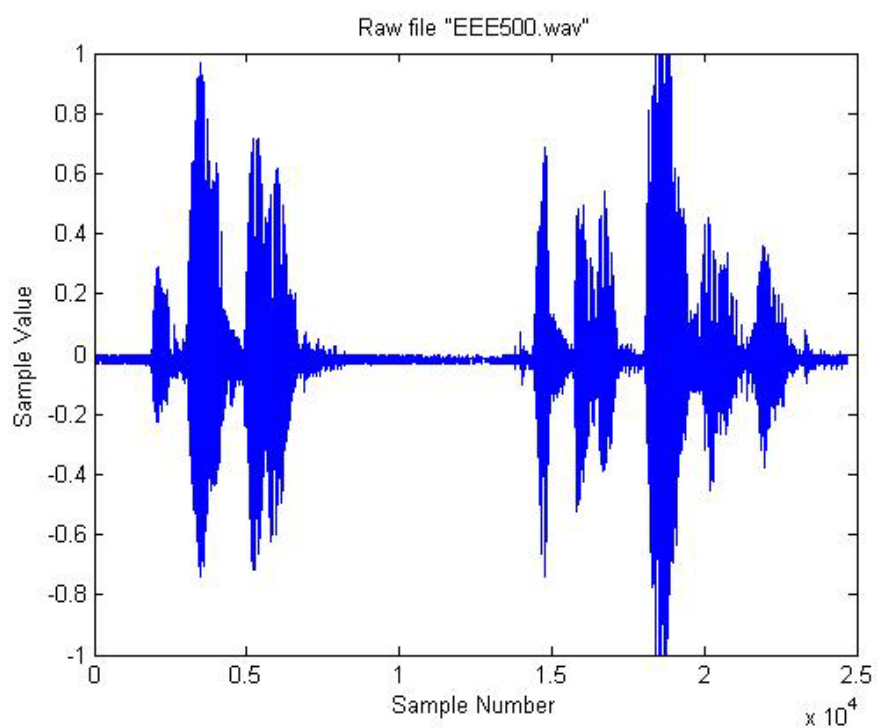


Figure 3-1: Raw .wav File

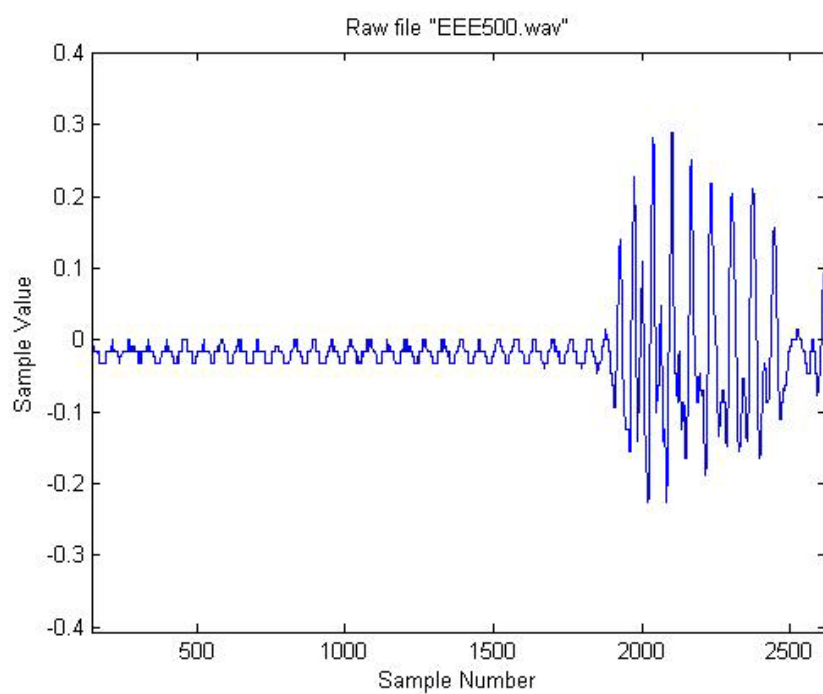


Figure 3-2: Zoomed Raw File



### 3.3.2 File Encoded with ADPCM Algorithm

The ADPCM algorithm discussed earlier can be used to encode the file to save bandwidth. The following figure will show the encoded file. Please note that there are only  $2^4 = 16$  possible levels. Hence only 4 bits are required to store or send the voice information.

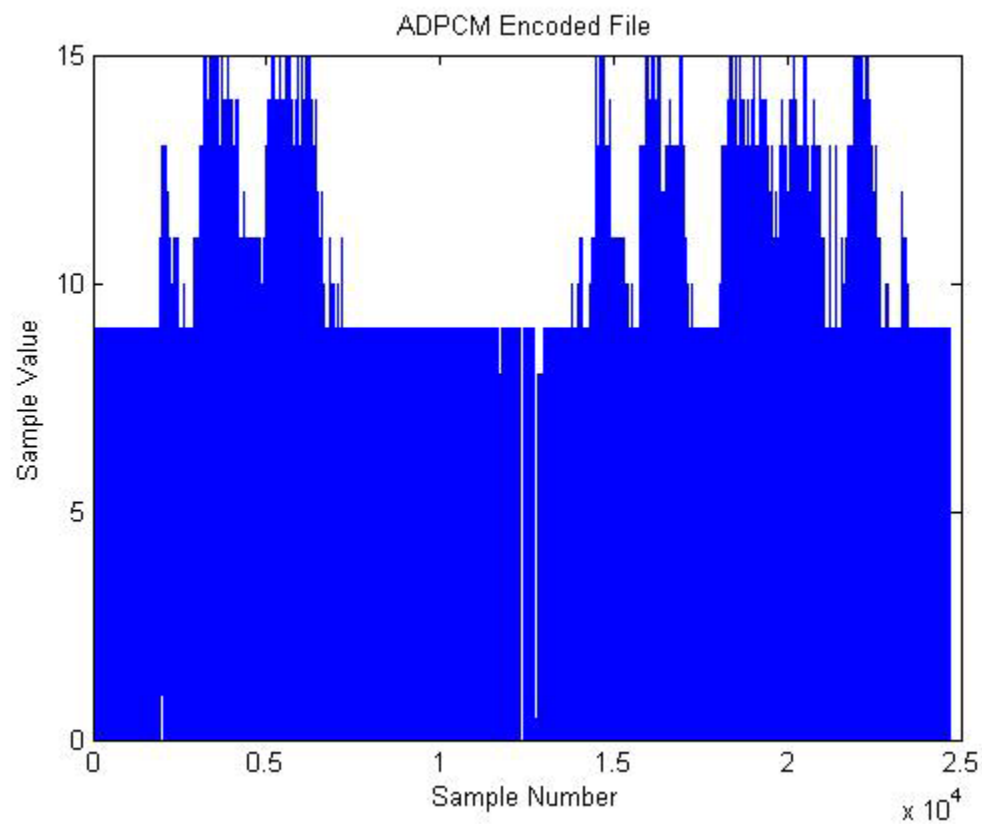


Figure 3-3: ADPCM Encoded File

### 3.3.3 File Decoded using ADPCM Algorithm

The following figures will show the decoded sample values.

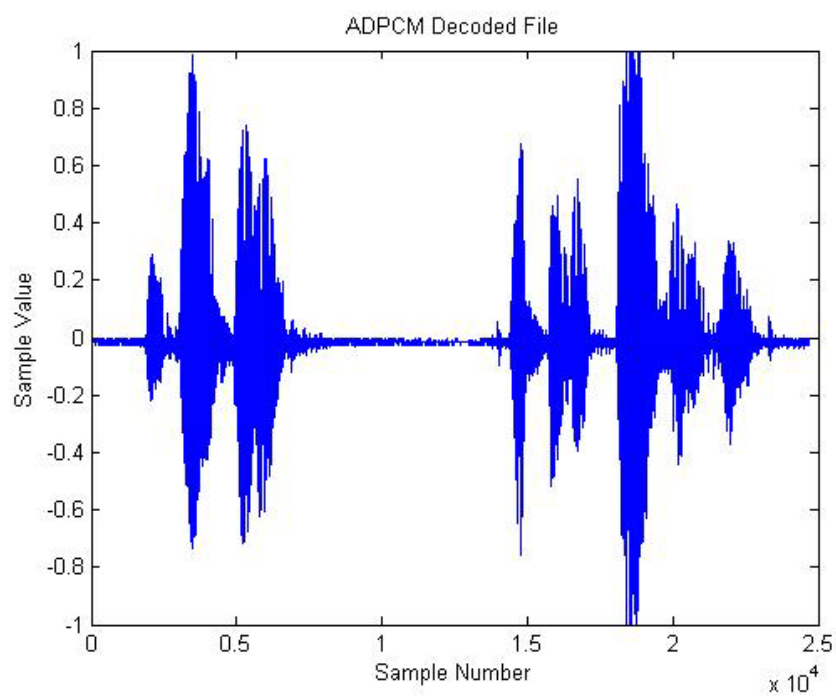


Figure 3-4: Decoded File

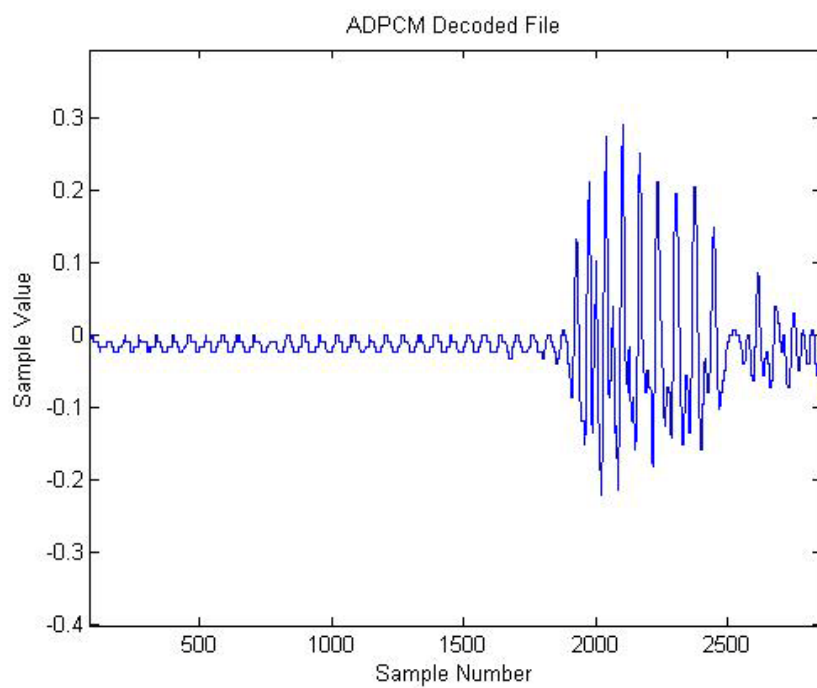


Figure 3-5: Zoomed Decoded File

### 3.3.4 Comparison of Raw File and Decoded File

The following graph is plotted for decoded values versus the raw values of the sample. Ideally it should be a straight line with angle of 45 degrees. But here due to some error, we can observe slight deviation from the ideal line.

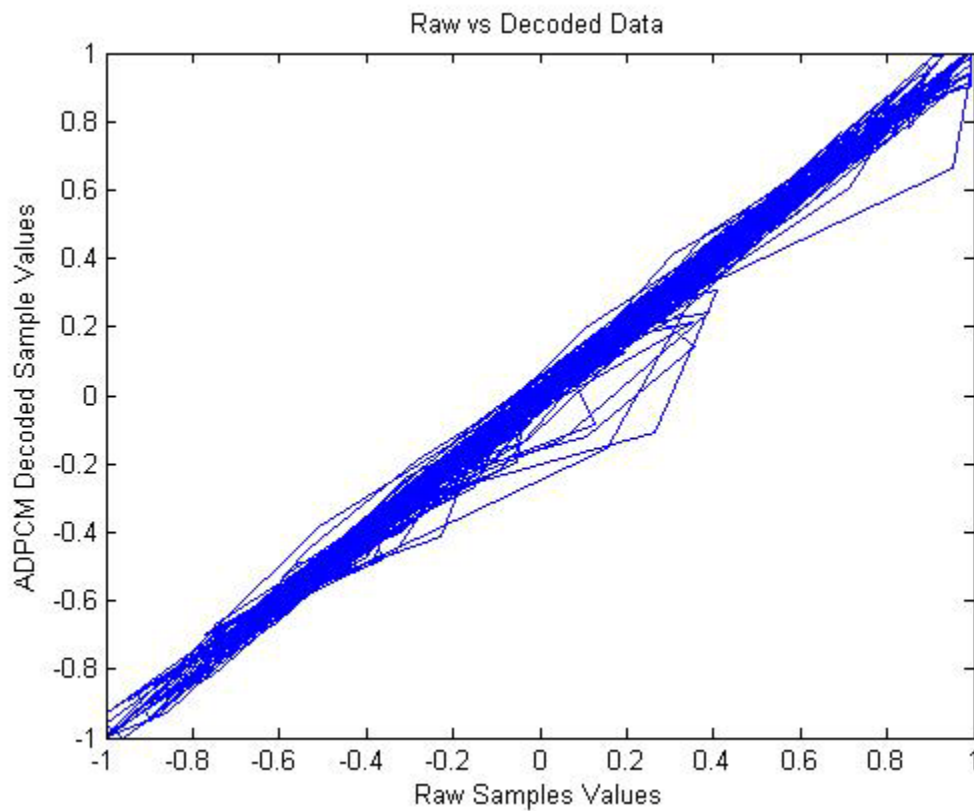


Figure 3-6: Raw Samples v/s Decoded Samples

### 3.4 C Language Implementation

Once you implement any algorithm in MATLAB, it is logically very easy to replicate in programming languages like C. I have used the same algorithmic steps to replicate the encoding and decoding functions. The experiment was done on the test file, but the results were not that encouraging. The main reason behind it was that the txt file can be read as group of ASCII character in C language. The difference in a bit can be interpreted as an entirely different character. While with the speech the case is not the same. The LAB errors are not much of significance in speech as the analog filters will smoothen it and minor different in sound level is not detectable by human ear. Thus the c language implementation was not of much help to the project, but none the less it can be used as a platform for further development in the same area.

To conclude, we have achieved 2:1 compression ration with low SNR and very compact hard ware. It can give added benefits like hardware portability and low power design. Both the added advantages are of extreme importance in speech recording playing devices. There is some error in decoded samples, but it is very much in tolerance range for speech signal. You can tell it by listening to the audio file in the attached CD. The C language codes are provided as an Appendix B.

## Chapter 4

### HARDWARE DESIGN

We will discuss the interfacing of microphone and stereo system with the STK 500 kit in this section. We will also see how the sound can be recorded played and erased from the chip. I have used ADC, PWM and flash read write interfaces for recording any playing the sound signal.

#### 1.1 Pre-amplification

An op-amp IC LM358 is used in non-inverting mode to amplify the voice signal that is provided by the microphone. The amplified signal is subsequently given to the ADC. During the design of pre amplifier circuit, getting the rail to rail output from the operational amplifier was very challenging. We solved that problem by changing the bias voltages at the inverting pin of the operational amplifier. The pre amplifier circuit is shown in the figure 4-1. I have used the PSPICE tool to plot the diagram and run the simulation.

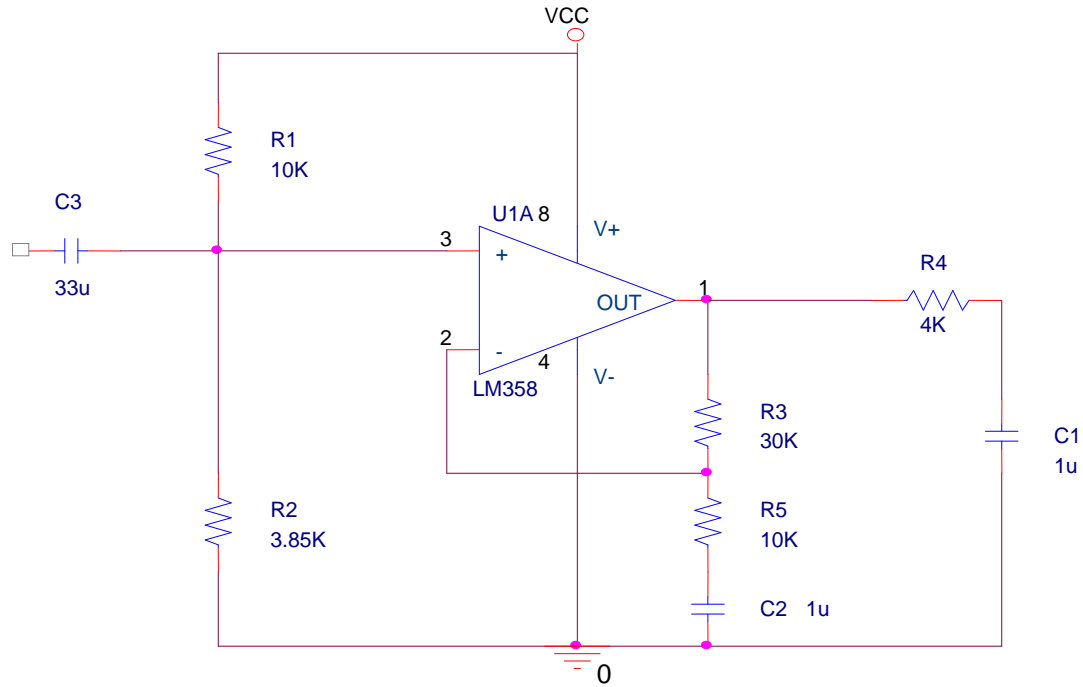


Figure 4-1: Microphone Pre-amplifier Circuit [2]

The gain of the voltage divider circuit can be approximated by following equation.

$$V_{out} = R_2 * \frac{v_{in}}{R_1 + R_2}$$

The value of the  $R_2$  was chosen to be 3.85K after monitoring the amplifier output for the microphone for the optimum value. Low pass filter was used to filter out the noise from the pre-amplified signal. Formula for RC filter cutoff frequency is as per following equation.

$$f_c = \frac{1}{2 * \pi * R * C}$$

R and C are the resistance and the capacitance of the Butterworth filter respectively.

Gain of the operational amplifier mainly depends upon feedback resistor ( $R_f$ ) and input resistor ( $R_i$ ). The approximate equation for the gain can be given as follows.

$$Gain = 1 + \frac{R_f}{R_i}$$

In our design  $R_f$  was 30K and  $R_i$  was 10K, hence the gain was approximately equal to 4.

## 4.2 Programming with STK500

There are mainly three operations that we are performing in this experiment, namely recording, playback and erasing. There is also a program that initializes the kit. We will look at each of the programming modules one by one.

### 4.2.1 Initialization

During the first run, all the logics supported by ATMEGA32 Microcontroller are initialized and switches are scanned for the other operations. The algorithm can be described by figure 4-2 and 4-3.

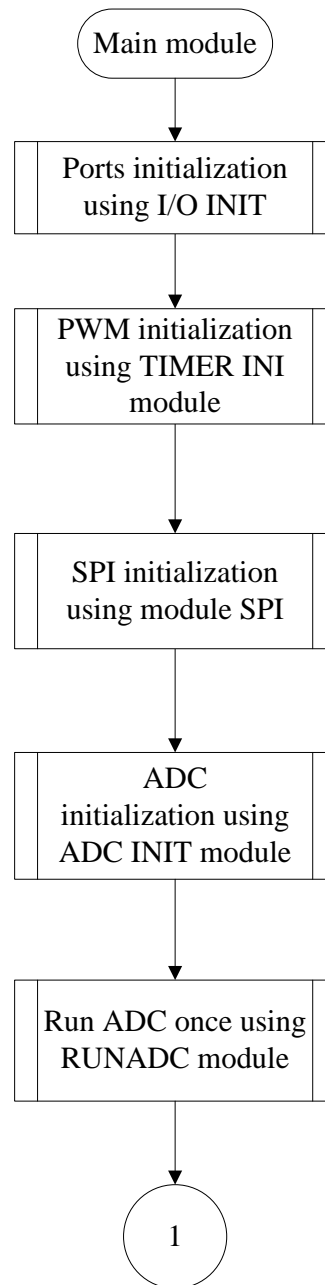


Figure 4-2: Initialization Algorithm (Part 1) [3]



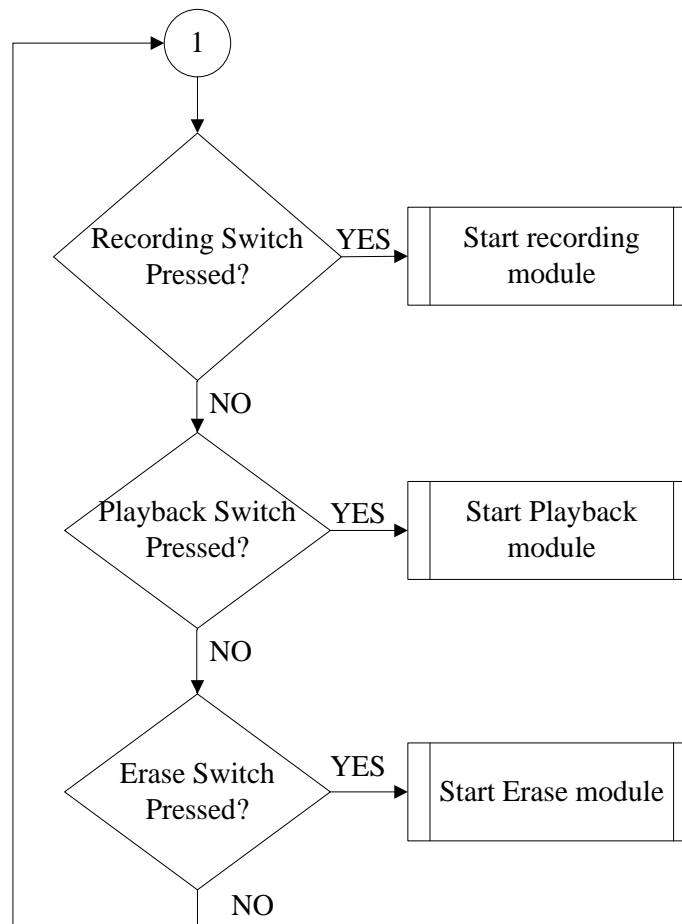


Figure 4-3: Initialization Algorithm (Part 2) [3]

#### 4.2.2 Recording

The recording function has two main parts as follows.

- a. To sample the audio data using ADC
- b. To store the sampled data in to the flash memory

One critical aspect of recording is to determine the sampling frequency that will eventually determine the number of samples for the given audio signal.

Figure 4-4 and 4-5 explains the recording process by algorithmic charts.

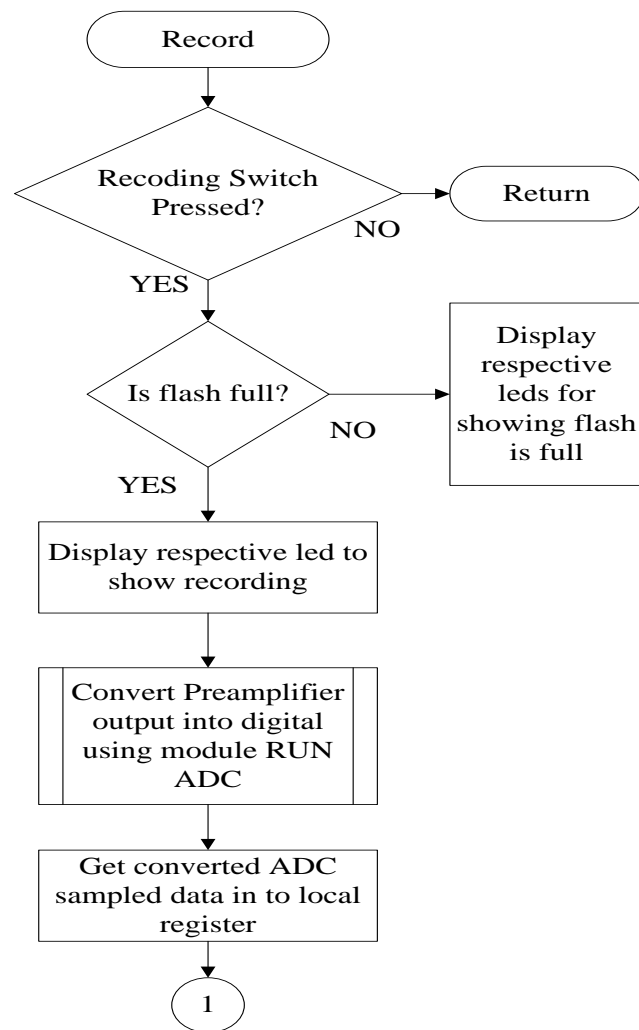


Figure 4-4: Recording Algorithm (Part 1) [3]

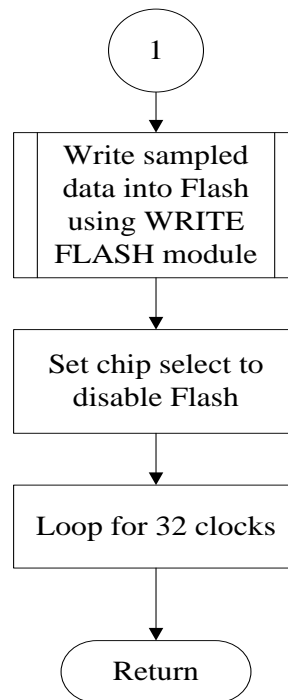


Figure 4-5: Recording Algorithm (Part 2) [3]

Why there was 32 clocks delay before returning to the next sample value? The PWM frequency was 31.25 KHz, while the crystal frequency of the microcontroller was 8 MHz. Hence, it will take 256 microcontroller cycles for crystal to update the value. Each ADC conversion consumes 14 cycles and each ADC cycle is 16 times the crystal cycle. Hence ADC conversion will take total of  $14 \times 16 = 224$  crystal cycles. ADC and flash write are parallel operations and that is why the number of clocks consumed by both the operations should be equal. Hence, we have to wait for  $256 - 224 = 32$  clock cycles before we can begin another ADC operation.

### 4.2.3 Playback

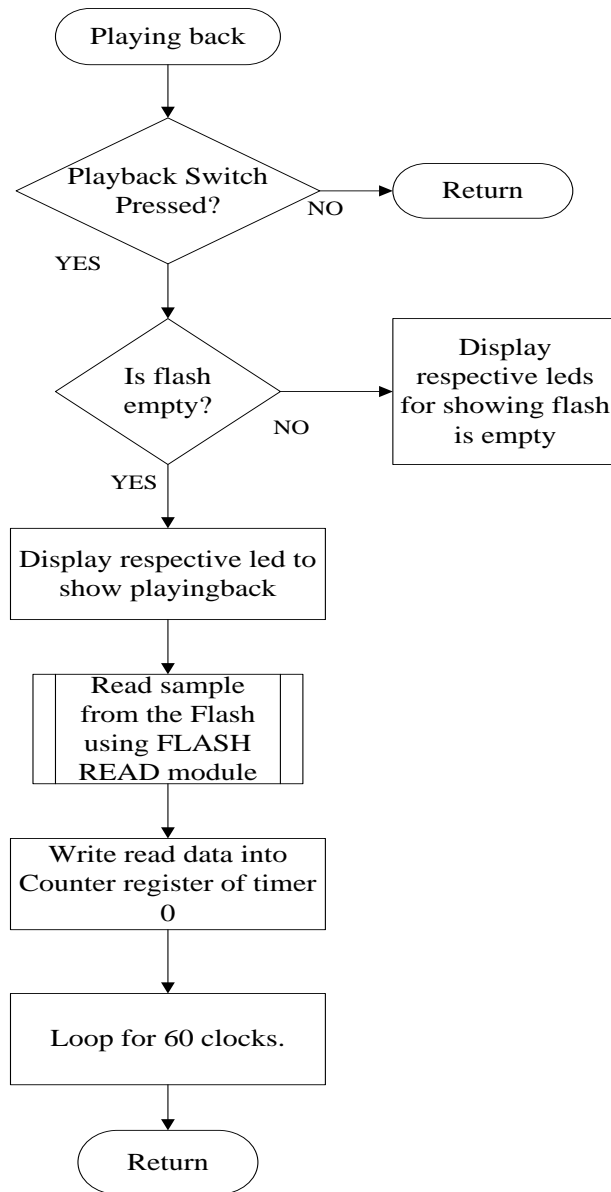


Figure 4-6: Playback Algorithm [3]

Figure 5-6 explains the Playback algorithm. The OCRO register of timer 0 stores the 8 bit sample read from flash memory. Now, why 64 clock delay before playing another sample? As explained earlier one PWM cycle is equal to 256 crystal cycles. Now flash reading takes 48 SPI cycles at 2 MHz frequency that is equal to 192 crystal cycles. Hence to synchronize the flash reading and PWM playback frequencies, we have to add  $256 - 192 = 64$  cycles delay in to our program. [2]

#### 4.2.4 Erase

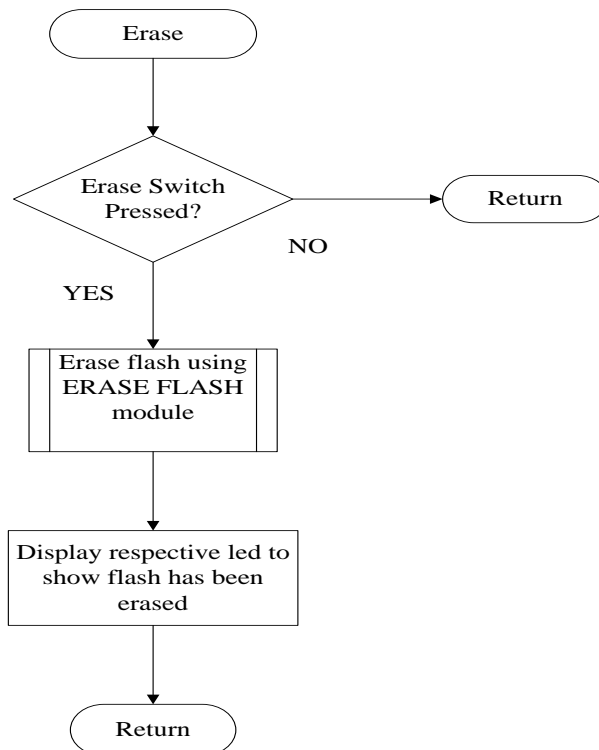


Figure 4-7: Erase Algorithm [3]

Erasing the flash is very straight forward process, as there is no need of synchronization. The process is explained in figure 4-7.

#### 4.4 PWM Filter

The processing that takes place in the embedded kit adds a significant amount of high frequency noise. Given the nature of the voice signal, two low pass filters were deployed to eliminate high frequency noise. The cut off frequency of the first order Chebyshev filter was chosen to be 3.3 KHz, while the cut off frequency of the 1<sup>st</sup> order Butterworth filter was chosen at 4 KHz. The design of the 1<sup>st</sup> order Butterworth filter was same as described recording section of this chapter. The Figure 4-8 explains the Chebyshev filter circuit in detail.

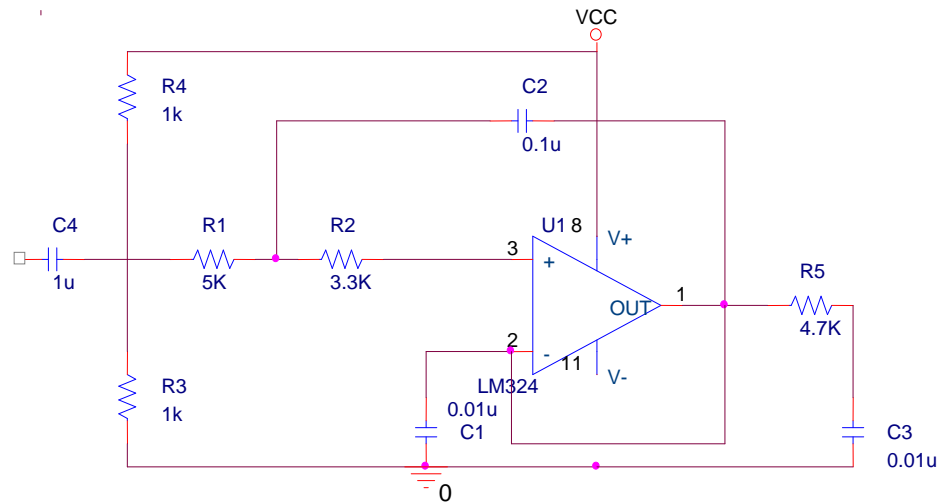


Figure 4-8: Analog Filter Circuit [4]

$$\text{The gain, } \frac{V_o}{V_{in}} = \frac{1}{a_2 s^2 + a_1 s + a_0}$$

Where,  $a_0 = 1$

$$a_1 = C1 * (R1 + R2)$$

$$a_2 = 2 * R1 * R2 * C1 * C2$$

$$\text{The cut off frequency, } fc = \frac{1}{2 * \pi * R4 * C4}$$

This was the process of sampling the sound storing in to a flash memory and playing it back.

After doing the practice experiments for a while we were able to achieve a significant amount of noise reduction in.

## Chapter 5

### CONCLUSION AND FUTURE WORK

After completing all three parts of project successfully, I was able to derive following conclusions:

1. With comparatively simplified ADPCM algorithms, we can achieve 2:1 compression ratio.
2. The error is very tolerable for the speech signals reproduced by ADPCM decoding algorithm.
3. The STK500 platform can be very efficiently utilized along with AVR studio to capture and reproduced sound signals.

In the future I would like to see someone building upon the C language programming and hardware work I have done so far and make a portable speech recorder/player that uses the ADPCM algorithm. The implementation as explained in the previous chapters will be simple and power efficient compared to other algorithms like MP3 and PCM.



## APPENDIX A

### MATLAB Codes

#### A.1 MATLAB Commands

```
>> Y = wavread("\gaia.ecs.csus.edu\jadejab\Project\Codes\EEE500.wav');  
>> plot(Y)  
>> Y_en = adpcm_encoder(Y);  
>> plot(Y_en)  
>> Y_de = adpcm_decoder(Y_en);  
>> plot(Y_de)  
>> wavwrite(Y_de, "\gaia.ecs.csus.edu\jadejab\Project\Codes\EEE500_decode.wav')
```

#### A.2 Encoder

```
IndexTable = [-1, -1, -1, -1, 2, 4, 6, 8];
```

```
StepSizeTable = [7, 8, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 20, 22, 24, 26, 29, 31, 34, 37, 41, 44,  
48, 53, 58, 63, 69, 75, 82, 89, 98, 107, 116, 127];
```

```
prevsample = 0;
```

```
previndex = 1;
```

```
Ns = length(raw_y);
```

```
n = 1;

raw_y = 127 * raw_y;      % 8-bit operation

while (n <= Ns)

    predsampl = prevsampl;

    index = previndex;

    step = StepSizeTable(index);

    diff = raw_y(n) - predsampl;

    if (diff >= 0)

        code = 0;

    else

        code = 8;

        diff = -diff;

    end

    tempstep = step;

    if (diff >= tempstep)

        code = bitor(code, 4);

        code = bitand(code, 15);

        diff = diff - tempstep;

    end

    tempstep = bitshift(tempstep, -1);
```

```

if (diff >= tempstep)

    code = bitor(code, 2);

    code = bitand(code, 15);

    diff = diff - tempstep;

end

tempstep = bitshift(tempstep, -1);

if (diff >= tempstep)

    code = bitor(code, 1);

    code = bitand(code, 15);

end

diffq = bitshift(step, -3);

if (bitand(code, 4))

    diffq = diffq + step;

end

if (bitand(code, 2))

    diffq = diffq + bitshift(step, -1);

end

if (bitand(code, 1))

    diffq = diffq + bitshift(step, -2);

end

if (bitand(code, 8))

    predsampl = predsampl - diffq;

```

```

else
    predsamp1e = predsamp1e + diffq;
end

```

```

if (predsamp1e > 127)
    predsamp1e = 127;
elseif (predsamp1e < -127)
    predsamp1e = -127;
end

```

```

x = bitand (code,7);
index = index + IndexTable(x + 1);

```

```

if (index < 1)
    index = 1;
end
if (index > 34)
    index = 34;
end

```

```

prevsample = predsamp1e;
previndex = index;

```

```

adpcm_y(n) = bitand(code, 15);
%adpcm_y(n) = code;

```

```

    n = n + 1;
end

```

### A.3 Decoder

```

IndexTable = [-1, -1, -1, -1, 2, 4, 6, 8];

```

```

StepSizeTable = [7, 8, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 20, 22, 24, 26, 29, 31, 34, 37, 41, 44,
48, 53, 58, 63, 69, 75, 82, 89, 98, 107, 116, 127];

```

```

prevsample = 0;
previndex = 1;

```

```

Ns = length(adpcm_y);
n = 1;

```

```

while (n <= Ns)
    predsampl = prevsample;
    index = previndex;
    step = StepSizeTable(index);
    code = adpcm_y(n);

    diffq = bitshift(step, -3);
    if (bitand(code, 4))

```

```

    diffq = diffq + step;
end
if (bitand(code, 2))
    diffq = diffq + bitshift(step, -1);
end
if (bitand(code, 1))
    diffq = diffq + bitshift(step, -2);
end

if (bitand(code, 8))
    predsampl = predsampl - diffq;
else
    predsampl = predsampl + diffq;
end

if (predsampl > 127)
    predsampl = 127;
elseif (predsampl < -127)
    predsampl = -127;
end

x = bitand(code, 7);
index = index + IndexTable(x + 1);

if (index < 1)

```

```
        index = 1;
    end
    if (index > 34)
        index = 34;
    end

    prevsample = predsampl;
    previndex = index;

    raw_y(n) = predsampl / 127;
    n = n + 1;
end
```

## APPENDIX B

### C Language Codes

#### B.1 Encoder

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

const char FILE_NAME_in[100] =
"C:\\Users\\bajadeja\\Desktop\\Project\\C_Programs\\raw.txt";

const char FILE_NAME_out[100] =
"C:\\Users\\bajadeja\\Desktop\\Project\\C_Programs\\encoded.txt";

#define fix2int(a)  ((signed char)((a)>>8))

#define SIGN_BIT 8

#define NUMSTEPS 34

#define int2fix(a)  (((int)(a))<<8)

#define MAX int2fix(127)

#define MIN int2fix(-127)

signed char ind[8] = {-1, -1, -1, -1, 2, 4, 6, 8};

signed char table4[NUMSTEPS] = {7, 8, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 20, 22,
24, 26, 29, 31, 34, 37, 41, 44, 48, 53, 58, 63, 69, 75, 82, 89, 98, 107, 116, 127};

static int  function_counter=0;
```



```
signed char currentIndex_e,currentIndex_d;

unsigned char temp, temp1,decoded_code;

signed int temp2,temp4;

signed int ss = 7;


int predict=0;

signed int d;

unsigned char NEG;

unsigned char code;


char adpcm_encoding (unsigned char sample_e);


int main(void) {

    printf("program starts");

    int count = 0; /* number of characters seen */

    FILE *in_file; /* input file */

    FILE *out_file; /* output file */


    int ch,x;


    in_file = fopen(FILE_NAME_in, "r");

    out_file = fopen(FILE_NAME_out, "w");


    if (in_file == NULL) {
```

```

printf("Cannot open %s\n", FILE_NAME_in);

    }

    printf("both file open\n");
while (1) {
    ch = fgetc(in_file);

        x =adpcm_encoding (ch);

        fputc(x, out_file);

        if (ch == EOF)

            break;

    ++count;

}

printf("Number of characters in %s is %d\n", FILE_NAME_in, count);


fclose(in_file);

    fclose (out_file);


getch ();

    return 0;

}

char adpcm_encoding (unsigned char sample_e)

{

```

```
        d = int2fix(sample_e) - predict;

code = 0;

if (d < 0)
{
    NEG = 1;
    code = SIGN_BIT;
    d = -d;
}
else
    NEG = 0;

temp2 = ss;
if (d >= temp2)
{
    code = code + 4;
    d = d - temp2;
}

temp2 = temp2/2;

if (d >= temp2)
{
    code = code + 2;
```

```
d = d - temp2;

}

temp2 = temp2/2;

    if (d >= temp2)
    {

        code = code + 1;
    }

temp = code & 0b00000111;


temp2 = ss/8;
if (temp & 1)
    temp2 = temp2 + ss/4;
if (temp & 2)
    temp2 = temp2 + ss/2;
if (temp & 4)
    temp2 = temp2 + ss;

if (NEG)
    temp2 = -temp2;
```

```

predict = predict + temp2;

if (predict > MAX)
    predict = MAX;
if (predict < MIN)
    predict = MIN;

currentIndex_e = currentIndex_e + ind[temp];

if (currentIndex_e < 0)
    currentIndex_e = 0;
if (currentIndex_e >= NUMSTEPS)
    currentIndex_e = NUMSTEPS-1;
ss = int2fix( table4[currentIndex_e] );
    function_counter++;

    printf("Line number = %d    Predict=%d    ss = %d    code=%d    d= %d
sample_e = %d\n",function_counter, predict,ss,code,d,sample_e);

return code;
}

```

## B.2 Decoder

```

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

static int function_counter;

const char FILE_NAME_in[100] =

"C:\\Users\\bajadeja\\Desktop\\Project\\C_Programs\\encoded.txt";

const char FILE_NAME_out[100] =

"C:\\Users\\bajadeja\\Desktop\\Project\\C_Programs\\decoded.txt";


#define fix2int(a)  (((signed char)((a)>>8))

#define SIGN_BIT 8

#define NUMSTEPS 34

#define int2fix(a)  (((int)(a))<<8)

#define MAX int2fix(127)

#define MIN int2fix(-127)


signed char ind[8] = {-1, -1, -1, -1, 2, 4, 6, 8};

signed char table4[NUMSTEPS] = {7, 8, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 20, 22,
24, 26, 29, 31, 34, 37, 41, 44, 48, 53, 58, 63, 69, 75, 82, 89, 98, 107, 116, 127};

signed char currentIndex_e,currentIndex_d;

unsigned char temp, temp1,decoded_code;

signed int temp2,temp4;

```

```

signed int ss = 7;

int predict=0,accum=0;           //predicted value of the current sample
signed int d;                    //difference between real value and predicted value
unsigned char NEG;               //indicates whether d is negative
unsigned char code;              //ADPCM code to be written to Flash

char adpcm_decoding (unsigned char sample_d);

int main(void) {
    printf("program starts");

    int count = 0;                /* number of characters seen */
    FILE *in_file;                /* input file */
    FILE *out_file;               /* output file */

    int ch,x;

    in_file = fopen(FILE_NAME_in, "r");
    out_file = fopen(FILE_NAME_out, "w");

    if (in_file == NULL) {
        printf("Cannot open %s\n", FILE_NAME_in);
    }
}

```

```

        printf("both file open");
while (1) {
    ch = fgetc(in_file);

        x = adpcm_decoding (ch);

        fputc(x, out_file);

        if (ch == EOF)

            break;

    ++count;
}

    printf("Number of characters in %s is %d\n", FILE_NAME_in, count);

fclose(in_file);

    fclose (out_file);

    getch ();

    return 0;
}

char adpcm_decoding (unsigned char sample_d)
{

    temp1 = sample_d & 0b00000111;

    temp4 = ss/8;

```



```
if (temp1 & 1)
    temp4 = temp4 + ss/4;
if (temp1 & 2)
    temp4 = temp4 + ss/2;
if (temp1 & 4)
    temp4 = temp4 + ss;

if (sample_d >= SIGN_BIT)
    temp4 = -temp4;

accum = accum + temp4;

if (accum > MAX)
    accum = MAX;
if (accum < MIN)
    accum = MIN;

currentIndex_d = currentIndex_d + ind[temp1];

if (currentIndex_d < 0)
    currentIndex_d = 0;
if (currentIndex_d >= NUMSTEPS)
    currentIndex_d = NUMSTEPS-1;
```

```
ss = int2fix( table4[currentIndex_d] );

    decoded_code = fix2int(accum);

    function_counter++;

    printf("Line number = %d    Predict=%d    ss = %d\n",
sample_d=%d    temp4 = %d    decoded_code = %d\n",function_counter,
accum,ss,sample_d,temp4,decoded_code);

    return decoded_code;

}
```

## REFERENCES

- [1] Rodger Richey - Microchip Technology Inc, "Adaptive Differential Pulse Code Modulation Using PIC® Microcontrollers", 2007.
- [2] Joel Chan & Jeremy Tan, "Sound effect processor", Cornell University, November 2003.
- [3] Atmel Corporation, "Application note AVR 335: digital sound recorder with AVR and data flash", 2005.
- [4] D. Gupta and V. Nandikonda, "Embedded system design 4840", Columbia University, 2004.
- [5] Perry Miller, "Aspects of data acquisition system design", Texas Instruments Inc., 2005.