ADPCM With A PIC32

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by:

Anthony Linley

MEng Field Advisor: Professor Bruce R. Land

Degree Date: August 2017

Table of Contents

Contents

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: ADPCM With A PIC32

Author: Anthony Linley

Abstract

The aim of this project, was to compress audio data in such a way so that the quality was preserved but was given at a lower bit rate, utilizing a PIC32 microcontroller. The speech compression algorithm used is known as adaptive differential pulse code modulation or ADPCM. The ADPCM algorithm can be broken down into two major components, the encoding process and the decoding process. In an effort to give audio capabilities to a microcontroller, a C implementation of a simplified ADPCM algorithm was developed and programmed onto the PIC32. During the testing phase, the C implementation for the PIC32 was compared to a working MATLAB implementation of the same algorithm, in order to confirm the numerical data was the same throughout the compression process. Once both the encoding and decoding processes produced identical outputs in both C and MATLAB, the code was put onto the microcontroller. The result was that audio compression was successful; the spectral content of the raw speech data and compressed speech data are the same and the entire process only used 1/5 of the CPU of the PIC32.

Executive Summary

This project was designed to implement speech and audio playback using a PIC32 microcontroller using adaptive differential pulse code modulation or ADPCM. ADPCM is a signal encoding process that takes audio data in and produces digital signals as an output. By only recording the differences between a sample and a predicted sample, the predictor can adjust itself appropriately which allows for signals to be produced at lower bit rates than when utilizing standard pulse code modulation.

This implementation is based on Microchip's documentation of a simplified ADPCM algorithm, designed to work on any PICmicro device. Before any actual development was done, a MATLAB implementation of Microchip's algorithm served as not only a basis for sound output but also for comparing the speech sample values as they were passed through the ADPCM encoder and decoder. Once confirming the MATLAB implementation worked, the development of the ADPCM algorithm in standard C began, using Code Blocks as the IDE. This allowed for a comparison of the numerical outputs of the C implementation of Microchip's ADPCM algorithm to the MATLAB implementation. After comparing both the outputs of the MATLAB and C codes with various sets of input data and confirming the correct functionality of the C program, the next step was porting the code to the PIC32.

Before being able to produce any sound output from the PIC32, a few peripherals needed to be set up. A 12-bit DAC, which communicated using SPI, was needed to convert the digital signals to analog signals so the audio could be played through speakers. A timer also needed to be set up, so the audio sampling rate could happen at 16kHz. This sampling frequency was found during the initial testing phases of the MATLAB code. The algorithm works successfully on the PIC32 microcontroller, as it does produce intelligible speech. When comparing the spectrograms of the raw speech data and the compressed audio data, the major features of the speech were unchanged. However, there was a small amount of noise in the compressed speech. It takes the PIC32 anywhere between 540 and 581 cycles to complete the entire ADPCM algorithm.

Introduction

Motivation

Being able to play recorded speech out of an electronic device is an extremely fascinating concept and makes the product itself that much more interesting. Especially in areas of interest such as robotics, adding the ability of speech playback is a great add on to any project. There are various ways to add the capability of speech playback to a device. One such way is to utilize adaptive differential pulse code modulation or ADPCM. Because consecutive speech samples are typically close together in value, the algorithm allows you to predict what the next speech sample is and adjust its encoding and decoding of the sample accordingly. This algorithm allows speech data to be compressed by only encoding the difference between the actual audio sample and a predicted audio sample, so it is given at a lower bit rate than the original data, making it easier to house this capability on a device such as a PIC32. The only other ways to add such capabilities to a project would be to use a special audio chip or processor and integrate it with your device somehow. By being able to handle the audio playback in software, without the need for additional hardware, you reduce the overall complexity of a device and don't lose any functionality.

This project is based on this need to have the ability of speech playback on a small device, such as a PIC32 microcontroller, without the need for additional hardware. This simplified ADPCM algorithm is based on guessing what the next speech sample is and adjusting its compression of the sample accordingly. This is actually the fundamental idea of the algorithm, and because it makes sound data much smaller it proves to be a much better choice when adding audio playback capabilities to any project.

Implementation

ADPCM Algorithm

The ADPCM algorithm has two major parts, the encoding process and the decoding process. The algorithm starts with the encoding part of the audio compression. The overall idea of the encoder is that it takes in a 16-bit speech sample and returns a 4-bit value which will be used to reconstruct the speech sample later on. Essentially, it takes a derivative of the speech sample. At a lower level, the 16-bit speech sample is passed into the encoder. After this, the values of the predicted audio sample and quantizer step size index from the end of the previous iteration of the encoding function are restored. Then the quantizer step size index is used to determine the actual quantizer step size for this iteration of the process. This is followed by finding the difference between the speech sample that is being encoded and the predicted sample. If the difference happens to be negative, then the absolute value of the difference is found and used instead. Next the difference is quantized into a 4-bit ADPCM code using the quantizer step size. This step is followed by the new ADPCM code being inverse quantized, or numerically integrated, into a predicted difference value, once again using the quantizer step size. Now to find the predicted sample to be used for the next iteration, the new predicted difference value is added to the old predicted sample value. In this version of the algorithm there is a value overflow check, to ensure all predicted sample values are 16-bit signed values. If the predicted sample value falls below -32768, it will cap that value at -32768; if the predicted value rises above 32767 than the value is reset to 32767, thus ensuring the predicted samples never leave the 16-bit signed range. Then the new quantizer step size index is found by using the ADPCM code as an index in a look up table containing various values. This value is added to the current index which gives us the new index for the next iteration. Both the new predicted sample and step size index are then saved and the 4-bit ADPCM code is output from the encoder.

The decoder process is not only simpler than the encoding process but also similar in some ways. The high level idea of the decoding process is that takes the 4-bit code from the encoder and outputs a 16-bit new speech sample. In more detail, the 4-bit code is passed into the decoder. The previous values of the quantizer step size index and the predicted sample are once again gotten from the previous loop of the process. As with the encoder, the quantizer step size is gotten from a table look up using the step size index. Then the 4-bit ADPCM code input is inverse quantized, or numerically integrated, into another predicted difference value. This value is then added to the old predicted speech sample value to get the new speech sample value. After this, there is a bounds check on the value to make sure it stays a 16-bit signed variable. Then the new step size index is found by adding the value from the table of index changes to the current index. Finally the new predicted sample and step size index are saved and the new 16-bit sample is output.

As mentioned earlier the decoding process is similar to the encoding but not exactly the same. The one major difference is the encoder does both quantization and inverse quantization while the decoder only does inverse quantization.

MATLAB Baseline

Before beginning to attempt to put the ADPCM algorithm onto a PIC32, I needed to find a way to not only understand how the algorithm worked but hear it for myself. With the suggestion from Professor Bruce Land, I decided to use MATLAB as a way to have a functioning version of Microchip's ADPCM algorithm. Having a working implementation of the algorithm in MATLAB served to be a great basis and tool of comparison throughout this development process.

First I used to the MATLAB code [1] to hear what the compressed audio should sound like running at 16kHz sampling frequency. Then after running the encoder and decoder functions, I converted my audio file into numerical data and output it to a text file. These numbers would serve as the speech samples needed for the ADPCM algorithm on the PIC and would be stored into an array.

C Implementation

After having a working implementation of Microchip's ADPCM algorithm in MATLAB, my next step in working on this project was to get the algorithm working in C. This was done to make sure that all the numerical values being passed were the same before attempting to put it on the PIC32. I began to implement just the encoding function in the appendix of the documentation from Microchip in C. I decided I would start with just the encoding function and once it was providing the same output as the MATLAB function, I would move on. After programming the encoding function, I took the floating point values from my MATLAB output text file and put those values into an array. However, the entire audio file was entirely too big to put into an array, so I chopped the down to the first 25000 entries. This also meant that my encoding function would be slightly different than the version in the documentation. Microchip's implementation takes a signed long integer as the input while mine takes floating point values as the input. However, the values still get stored into integers so the outcome is still the same. I used print statements and for loops to test random segments of the data in the audio floating point array. Then I would run the encoding function in MATLAB for those same segments of data, and compare the numbers. After testing and comparing numerous sets of data between the C implementation and MATLAB and receiving the same values, I moved to working on the decoding function. I followed the same process to test the decoder. After programming the function from the appendix in the documentation, I began to test the decoder with the outputs from my encoder. Because I was sure that my encoder worked from my testing I knew that whatever outputs my decoder had, should be correct. I once again compared the outputs of random sets of data from the MATLAB function and the C function and confirmed they were the same. Finally, I ran both codes in their entirety and compared the outputs and the numbers all matched up.

Working On The PIC32

The final part of the development part of this project was to move my C code to the PIC32. Even though I knew my ADPCM algorithm worked numerically, porting the code over to the PIC and having audio come out would be no easy feat. I started out by taking some of my old lab 2 code from ECE 4760 and using that as a starting point. Because lab 2 dealt with outputting sound through a 12-bit DAC, it was the perfect code to reference for help. First I added my functions from my standalone C code to my code that would go on the PIC32. Then I moved my array of floating point speech samples to the code. After this was done I began to write code to set up a 12-bit DAC. I needed a DAC in order to be able to output the compressed speech through speakers, to prove that the algorithm does indeed work. Otherwise, there would be no true indication of functionality or not. The first part in setting up the DAC was to define the channels for the DAC, which was promptly followed by setting up the SPI channel and SPI clock divider for the DAC to use. Next, I had to set up lines of code for the DAC to be able to output data through the speakers. Because I had code that used the same concept from ECE 4760, I copied the DAC output lines of code from that class into this code. Not only did this save me time from writing it, I already knew that it would function correctly. However, because the value from the decoder was a 32-bit long value, and the relevant information from the decoder was placed in the 16 most significant bits of this 32-bit value, I needed to shift the value 20 bits in order for the value to be passed through the 12-bit DAC. After setting up the DAC, I needed to set up the timer on the PIC. Because the PIC32 was running at 40MHz I needed to setup the timer, Timer 2, to overflow ever 2500 cycles. This would be needed to ensure that I was running this algorithm at a 16kHz sampling rate, the same as the MATLAB code. After setting up the timer, I moved onto creating the Interrupt Service Routine. The ISR is where the ADPCM algorithm occurs in this project. Every time Timer 2 overflows, the ISR encodes, decodes and outputs to the DAC which causes the speech to be heard. However, in order for the Protothreads threading library to run, there has to be at least one thread in the code. I simply made an empty thread and let it run in a round robin scheduler. After all of the code was written, I then began wiring up the DAC according to the datasheet which was simple and straightforward. Once all of this was done, I downloaded my code to the PIC and wired a speaker audio jack to the DAC and successfully heard the anticipated speech audio.

Results and Conclusions

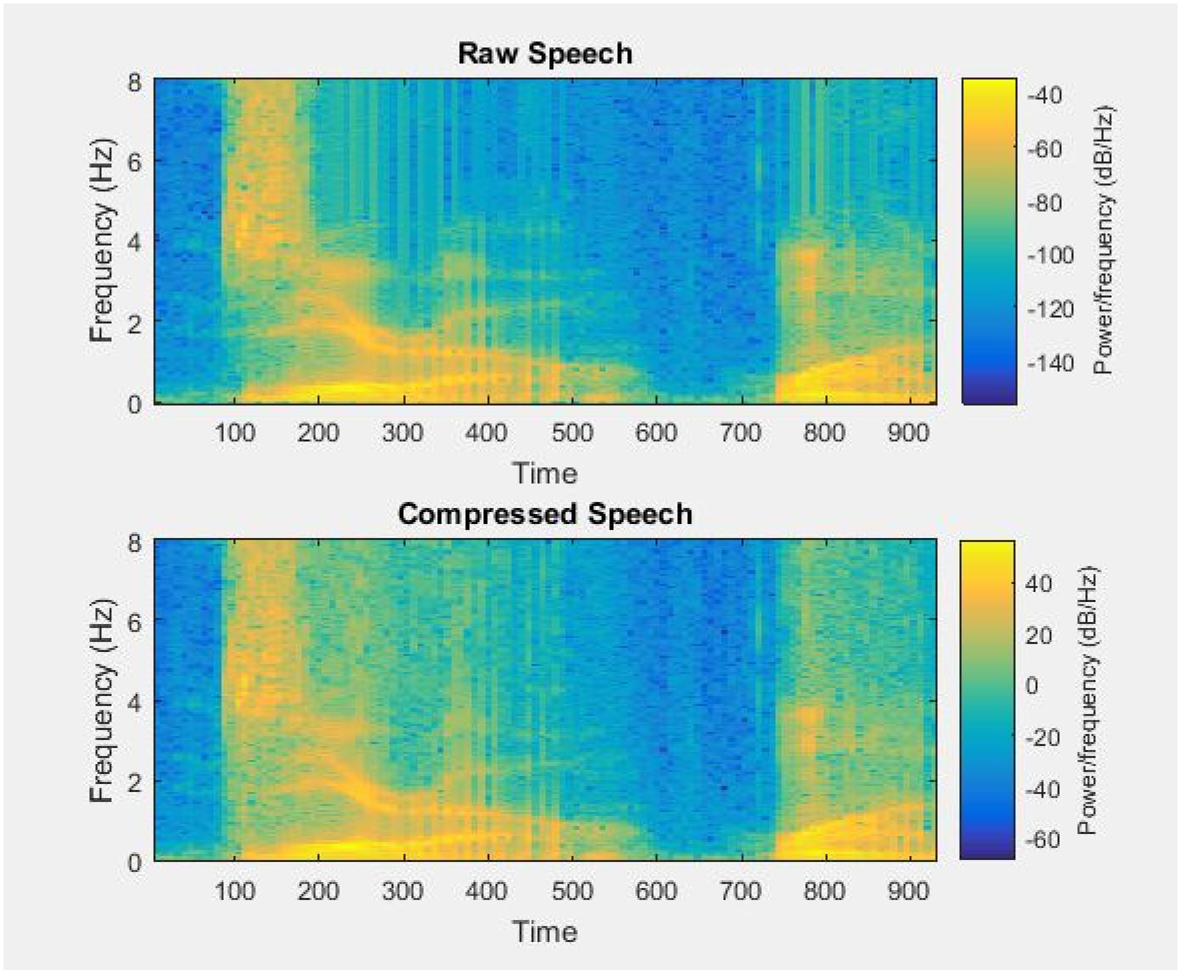


Figure 1: Raw Vs. Compressed Audio Data

Figure 1 contains the spectrograms for the raw audio data and the compressed audio data. As can been seen by comparing the two, the major features of the spectral content are almost identical. There is a bit more high frequencies that can be seen in the compressed spectrogram, as well as a bit of noise. Nevertheless, this quantitatively shows just how close the uncompressed and compressed data actually are. As far as performance on the PIC32 is concerned, the entire ADPCM process took anywhere from 540-581 cycles to complete. Because the timer would overflow at every 2500 cycles, I was roughly using about 1/5 of the CPU on the PIC.

Although the project was successful, there were many issues along the way. The first was when I first attempted the project. I immediately began trying to program the algorithm on the PIC and wiring up the DAC. During these attempts at just going straight to programming the device I did not hear anything coming out of the DAC. This confused me because I had idea whether I was having issues with the algorithm logic or was it a hardware issue. This caused me to end up getting rid of that entire version of that code. Next was trying to get MATLAB to not output it's numerical data in scientific notation. Because I would need strictly floating point values when working in C and would not want to have to write a particularly complicated conversion from scientific notation function, I had to search and find out about the formatSpec function in MATLAB. This gave all my floating point values in a way that I could easily translate into a C array. The third problem I had was loading the array of values onto the PIC. Because the entire audio file used with MATLAB was over 90000 speech samples, I had to trim the number of values down to 25000 when I was working with the C code in Code Blocks. However, once I tried to move that over the PIC, I did not have enough space on the device to hold that many values. This made me cut the number of values down again to 15000 but this still wasn't enough allow it to fit on the PIC32. In order to get around this running out of space issue, I declared the array of floating point values to be **const** so the compiler on the PIC would place the array in Flash instead of RAM. This particular solution was recommended by Professor Land. I also had an issue with hearing strange noises come from the speakers. This was due to the fact that I was attempting to run my algorithm in my main code, but after moving the algorithm to an ISR I began to hear intelligible speech. A fifth issue was after getting the speech to successfully be heard coming from the PIC, there was quite a lot of noise that was heard through the speakers. To solve this issue I used a 3 kHz low pass filter on the output of the DAC which greatly helped the quality of sound being heard. This was another solution proposed by Professor Land.

Acknowledgements

First I would like to give my greatest thanks to Professor Land, who is my advisor, for his help throughout this project, his door always being open for questions, sharing ideas or just to talk.

I would like to thank my girlfriend, soon to be fiancée. For always supporting me, being there to lift my spirits and most importantly for being strong and raising our daughter while I was completing this program. I would like to my parents for helping me get through any and every tough situation that arose throughout this school year. Lastly, I would like to thank my daughter, Aubrey. For being the sun on the many cloudy days I've had through this entire year; the mere thought of her has pushed me through the times when I felt like giving up.

Code Appendix

```
/*
 * File:        TFT_keypad_BRL4.c
 * Author:      Bruce Land
 * Adapted from:
 *              main.c by
 * Author:      Syed Tahmid Mahbub
 * Target PIC:  PIC32MX250F128B
 */

#include "config.h"
#include <xc.h> // need for pps
 // graphics libraries
#include "tft_master.h"
#include "tft_gfx.h"
// need for sine function
#include "math.h"
// the spectral amplitudes
//#include "audio_digits_wc_25000.h"

// threading library
//#define _SUPPRESS_PLIB_WARNING
//#define _DISABLE_OPENADC10_CONFIGPORT_WARNING
//#include <plib.h>
// config.h sets 40 MHz
#define    SYS_FREQ 40000000
//#define two32 4294967296.0 // 2^32
#define Fs 12000.0
#include "pt_cornell_1_2_1.h"
// use boolean type
#include "stdbool.h"

 // Configuration Bit settings
// SYSCLK = 40 MHz (8MHz Crystal/ FPLLIDIV * FPLLMUL / FPLLODIV)
// PBCLK = 40 MHz
// Primary Osc w/PLL (XT+,HS+,EC+PLL)
// WDT OFF
// Other options are don't care
//                      8MHZ                       4MHz
80MHz            40      <---    40MHz
//#pragma config FNOSC = FRCPLL, POSCMOD = OFF, FPLLIDIV = DIV_2, FPLLMUL
= MUL_20, FPBDIV = DIV_1, FPLLODIV = DIV_2
//#pragma config FWDTEN = OFF
//#pragma config FSOSCEN = OFF, JTAGEN = OFF, DEBUG = OFF


/* Demo code for interfacing TFT (ILI9340 controller) to PIC32
 * The library has been modified from a similar Adafruit library
 */
```

```
// Adafruit data:
/***************************************************
  This is an example sketch for the Adafruit 2.2" SPI display.
  This library works with the Adafruit 2.2" TFT Breakout w/SD card
  ----> http://www.adafruit.com/products/1480

  Check out the links above for our tutorials and wiring diagrams
  These displays use SPI to communicate, 4 or 5 pins are required to
  interface (RST is optional)
  Adafruit invests time and resources providing this open source code,
  please support Adafruit and open-source hardware by purchasing
  products from Adafruit!

  Written by Limor Fried/Ladyada for Adafruit Industries.
  MIT license, all text above must be included in any redistribution
 ****************************************************/
// === 16:16 fixed point macros ==========================================

static struct pt pt_print;

//deleted floating point sample values for sake of being able to keep code
//into draft. Array held 15000 speech samples all in floating point.
//values were taken from audio sample and printed to text file using
//MATLAB.
static const float audio_digits[]={/*insert 15000 floating point speech
sample values*/};


// === thread structures ============================================
// thread control structs
// note that UART input and output are threads
//static struct pt pt_key ;

// A-channel, 1x, active
#define DAC_config_chan_A 0b0011000000000000
#define DAC_config_chan_B 0b1011000000000000
//== Timer 2 interrupt handler ===========================================

volatile SpiChannel spiChn = SPI_CHANNEL2 ; // the SPI channel to use
volatile int spiClkDiv = 2 ; // 20 MHz max speed for this DAC


//encoding
volatile signed long prev_sample_e =0;
volatile int         prev_index_e  =0;

// decoding
volatile signed long prev_sample_d = 0;
volatile int         prev_index_d  = 0;

volatile int answer_d;
volatile char answer_e;
volatile int size = (sizeof(audio_digits)/sizeof(float));//for testing
volatile int n = 0;


const int IndexTable[16] = {
  -1, -1, -1, -1, 2, 4, 6, 8,
  -1, -1, -1, -1, 2, 4, 6, 8
};

/* Quantizer step size lookup table */
const long StepSizeTable[89] = {
  7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
  19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
  50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
  130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
  337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
  876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
  2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
  5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
  15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};


//function prototype
char encoder(float sample);
signed long decoder(char code);

void DAC_output(int output);

volatile int timer=0;

void DAC_output(int output){
    //============ channel A =================
    // CS low to start transaction
     mPORTBClearBits(BIT_4); // start transaction
    // test for ready
    while (TxBufFullSPI2());
    // write to spi2
    WriteSPI2(DAC_config_chan_A | (output & 0xfff));
    // test for done
    while (SPI2STATbits.SPIBUSY); // wait for end of transaction
    // CS high
    mPORTBSetBits(BIT_4); // end transaction


}

void __ISR(_TIMER_2_VECTOR, ipl2) Timer2Handler(void)
{
    mT2ClearIntFlag();

    answer_e = encoder(audio_digits[n]);
    answer_d = decoder(answer_e);
    DAC_output(answer_d>>20);
    n++;
    if(n==size){n=0;}//reset n back to zero
    timer= TMR2;

}

static PT_THREAD(protothread_print(struct pt *pt)){
    PT_BEGIN(pt);
    PT_YIELD_TIME_msec(10) ;
    printf("timer: %i",timer);
    PT_END(pt);

}//end print thread

// === Main  ======================================================
void main(void) {
 SYSTEMConfigPerformance(PBCLK);

  ANSELA = 0; ANSELB = 0; CM1CON = 0; CM2CON = 0;

  // Configure the device for maximum performance but do not change the
PBDIV
  // Given the options, this function will change the flash wait states,
RAM
  // wait state and enable prefetch cache but will not change the PBDIV.
  // The PBDIV value is already set via the pragma FPBDIV option above..
  SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

  // timer interrupt //////////////////////////
  // Set up timer2 on,  interrupts, internal clock, prescalar 1, toggle
rate
  // at 30 MHz PB clock 60 counts is two microsec
  // 2000 is 20 ksamp/sec
  OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_1, 2500);

  // set up the timer interrupt with a priority of 2
  ConfigIntTimer2(T2_INT_ON | T2_INT_PRIOR_2);
  mT2ClearIntFlag(); // and clear the interrupt flag

  // SCK2 is pin 26
  // SDO2 (MOSI) is in PPS output group 2, could be connected to RB5 which
is pin 14
  PPSOutput(2, RPB5, SDO2);

  // control CS for DAC
  mPORTBSetPinsDigitalOut(BIT_4);
  mPORTBSetBits(BIT_4);

  mPORTBSetPinsDigitalIn(BIT_13);

  // divide Fpb by 2, configure the I/O ports. Not using SS in this
example
  // 16 bit transfer CKP=1 CKE=1
  // possibles SPI_OPEN_CKP_HIGH;   SPI_OPEN_SMP_END;  SPI_OPEN_CKE_REV
  // For any given peripherial, you will need to match these
  SpiChnOpen(spiChn, SPI_OPEN_ON | SPI_OPEN_MODE16 | SPI_OPEN_MSTEN |
SPI_OPEN_CKE_REV , spiClkDiv);


  // === setup system wide interrupts  ========
  INTEnableSystemMultiVectoredInt();

  PT_setup();

  PT_INIT(&pt_print);


  while (1){

       PT_SCHEDULE(protothread_print(&pt_print));
      }//end while
} // main

// === end  ======================================================

char encoder(float sample){
  signed long diff; /* Difference between sample and predicted sample */
  long step; /* Quantizer step size */
  signed long predsample; /* Output of ADPCM predictor */
  signed long diffq; /* Dequantized predicted difference */
  int index; /* Index into step size table */

  int code; /* ADPCM output value */
  int tempstep; /* Temporary step size */

  /* Restore previous values of predicted sample and quantizer step
   size index
  */

  //dealing with 16 bit sample...gotten from matlab code
  sample = sample * 32767;



  predsample = prev_sample_e;
  index = prev_index_e;
  step = StepSizeTable[index];

  /* Compute the difference between the actual sample (sample) and the
   the predicted sample (predsample)
  */
  diff = sample - predsample;

  if(diff >= 0)
    code = 0;
  else {
    code = 8;
    diff = -diff;
  }


  /* Quantize the difference into the 4-bit ADPCM code using the
   the quantizer step size
  */
  tempstep = step;
  if( diff >= tempstep ) {
    code |= 4;
    diff -= tempstep;
  }

  tempstep >>= 1;
  if( diff >= tempstep ) {
    code |= 2;
    diff -= tempstep;
  }

  tempstep >>= 1;
  if( diff >= tempstep )
    code |= 1;

  /* Inverse quantize the ADPCM code into a predicted difference
   using the quantizer step size
  */
  diffq = step >> 3;
  if( code & 4 )
    diffq += step;
  if( code & 2 )
    diffq += step >> 1;
  if( code & 1 )
    diffq += step >> 2;

  /* Fixed predictor computes new predicted sample by adding the
   old predicted sample to predicted difference
  */
  if( code & 8 )
    predsample -= diffq;
  else
    predsample += diffq;
  /* Check for overflow of the new predicted sample
  */
  if( predsample > 32767 )
    predsample = 32767;
  else if( predsample < -32768 )
    predsample = -32768;
  /* Find new quantizer stepsize index by adding the old index
   to a table lookup using the ADPCM code
  */
  index += IndexTable[code];
  /* Check for overflow of the new quantizer step size index
  */
  if( index < 0 )
```

```
    index = 0;
  if( index > 88 )
    index = 88;
  /* Save the predicted sample and quantizer step size index for
   next iteration
  */
  prev_sample_e = predsample;
  prev_index_e = index;

  /* Return the new ADPCM code */
  return ( code & 0x0f );

}//end of encoder

signed long decoder( char code ) {

  long step; /* Quantizer step size */
  signed long pred_sample; /* Output of ADPCM predictor */
  signed long diffq; /* Dequantized predicted difference */
  int index; /* Index into step size table */

  /* Restore previous values of predicted sample and quantizer step
   size index
  */
  pred_sample = prev_sample_d;
  index = prev_index_d;

  /* Find quantizer step size from lookup table using index
  */
  step = StepSizeTable[index];
  /* Inverse quantize the ADPCM code into a difference using the
   quantizer step size
  */

  diffq = step >> 3;

  if( code & 4 )
    diffq += step;
  if( code & 2 )
    diffq += step >> 1;
  if( code & 1 )
    diffq += step >> 2;
  /* Add the difference to the predicted sample
  */
  if( code & 8 )
    pred_sample -= diffq;
  else
    pred_sample += diffq;



  /* Check for overflow of the new predicted sample
  */
  if( pred_sample > 32767 )
```

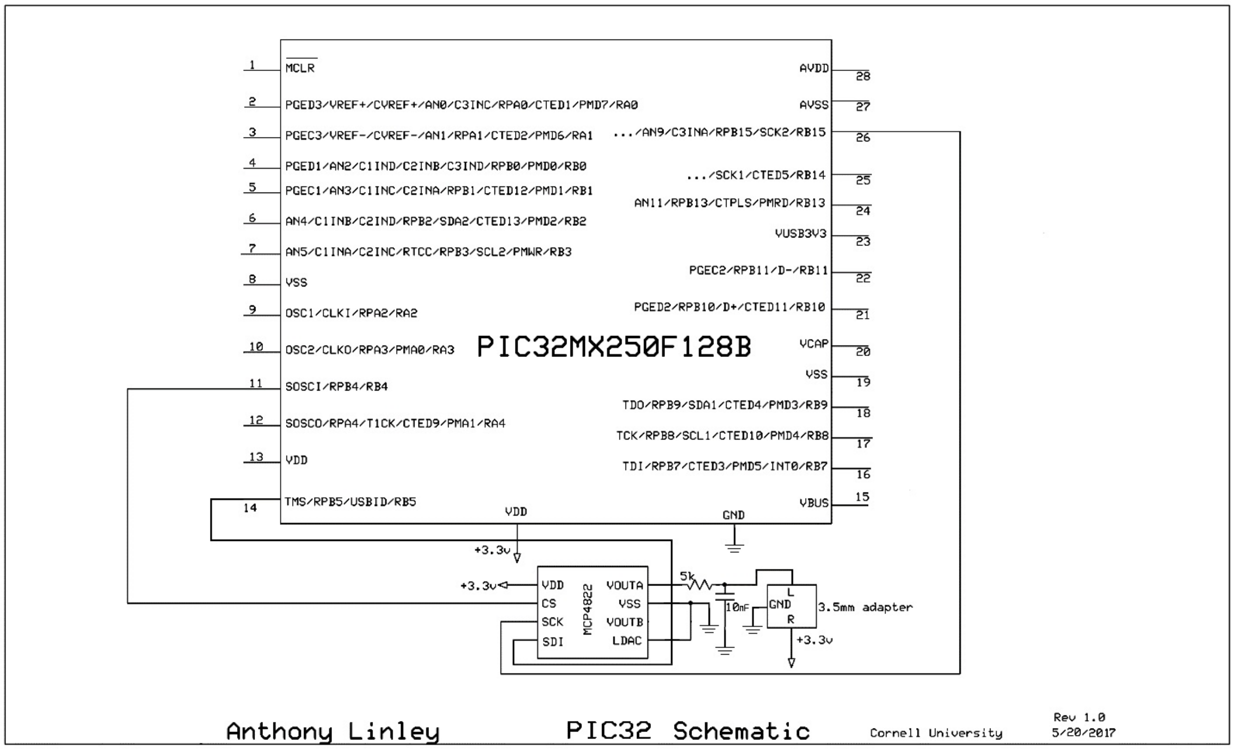
```
    pred_sample = 32767;
  else if( pred_sample < -32768 )
    pred_sample = -32768;
  /* Find new quantizer step size by adding the old index and a
   table lookup using the ADPCM code
  */


  index += IndexTable[code];
  /* Check for overflow of the new quantizer step size index
  */
  if( index < 0 )
    index = 0;
  if( index > 88 )
    index = 88;
  /* Save predicted sample and quantizer step size index for next
   iteration
  */


  prev_sample_d = pred_sample;
  prev_index_d = index;

  /* Return the new speech sample */
  return( pred_sample );
}
```

Schematic


PIC32MX250F128B
1 MCLR
2 PGED3/VREF+/CVREF+/AN0/C3INC/RPA0/CTED1/PMD7/RA0
3 PGEC3/VREF-/CVREF-/AN1/RPA1/CTED2/PMD6/RA1
4 PGED1/AN2/C1IND/C2INB/C3IND/RPB0/PMD0/RB0
5 PGEC1/AN3/C1INC/C2INA/RPB1/CTED12/PMD1/RB1
6 AN4/C1INB/C2IND/RPB2/SDA2/CTED13/PMD2/RB2
7 AN5/C1INA/C2INC/RTCC/RPB3/SCL2/PMWR/RB3
8 VSS
9 OSC1/CLKI/RPA2/RA2
10 OSC2/CLKO/RPA3/PMA0/RA3
11 SOSCI/RPB4/RB4
12 SOSCO/RPA4/T1CK/CTED9/PMA1/RA4
13 VDD
14 TMS/RPB5/USBID/RB5
AVDD 28
AVSS 27
.../AN9/C3INA/RPB15/SCK2/RB15 26
.../SCK1/CTED5/RB14 25
AN11/RPB13/CTPLS/PMRD/RB13 24
VUSB3V3 23
PGEC2/RPB11/D-/RB11 22
PGED2/RPB10/D+/CTED11/RB10 21
VCAP 20
VSS 19
TDO/RPB9/SDA1/CTED4/PMD3/RB9 18
TCK/RPB8/SCL1/CTED10/PMD4/RB8 17
TDI/RPB7/CTED3/PMD5/INT0/RB7 16
VBUS 15
VDD
GND
+3.3v
+3.3v
MCP4822
VDD
CS
SCK
SDI
VOUTA
VSS
VOUTB
LDAC
5k
10nF
L
GND
R
3.5mm adapter
+3.3v
Anthony Linley
PIC32 Schematic
Cornell University
Rev 1.0
5/20/2017

References

[1] https://www.mathworks.com/matlabcentral/fileexchange/6480-adpcm-encoder-and-decoder?focused=5056015&tab=function