18-551 Final Project Report
Spring 2003

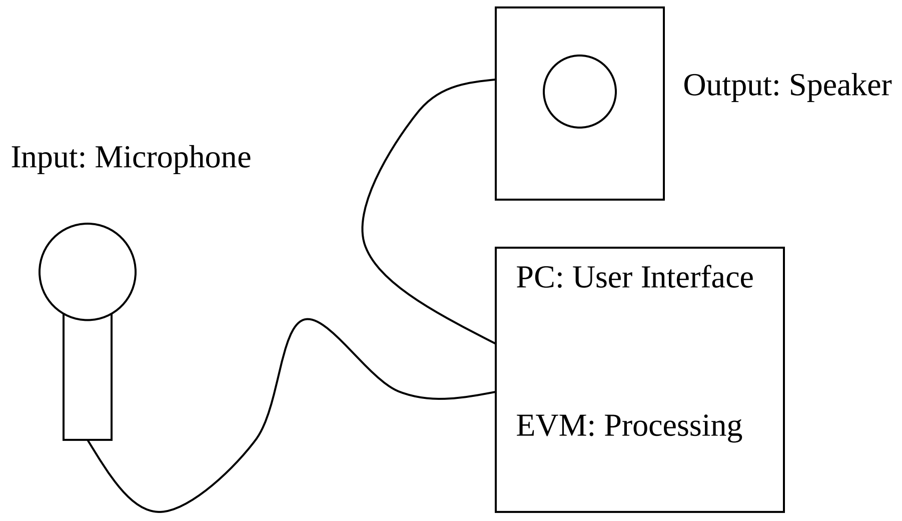
"Sing-Synth" voice driven synthesizer

Group 9:
Brian Jucha (bgj@)
Tadge Dryja (tpd@)
Yves Stauffer(ystautte@)

Project Overview

Problem

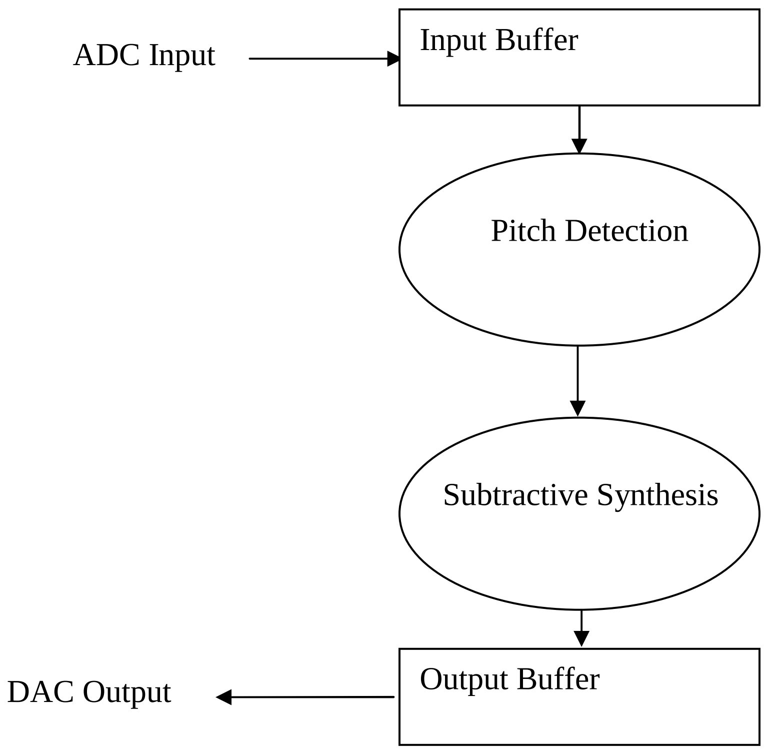
Have you ever tried to play a musical instrument, but found that it was too hard for you? Surely it would be easier if you didn't have to use a complex, unintuitive instrument. One way around this would be to have an instrument that could synthesize notes from your own voice. Everyone can sing, hum, or whistle, but these types of sounds are somewhat limited. A device that could synthesize complex instruments based on human music could attract a variety of listeners, as well as musicians who wouldn't otherwise be heard on traditional instruments.



Solution

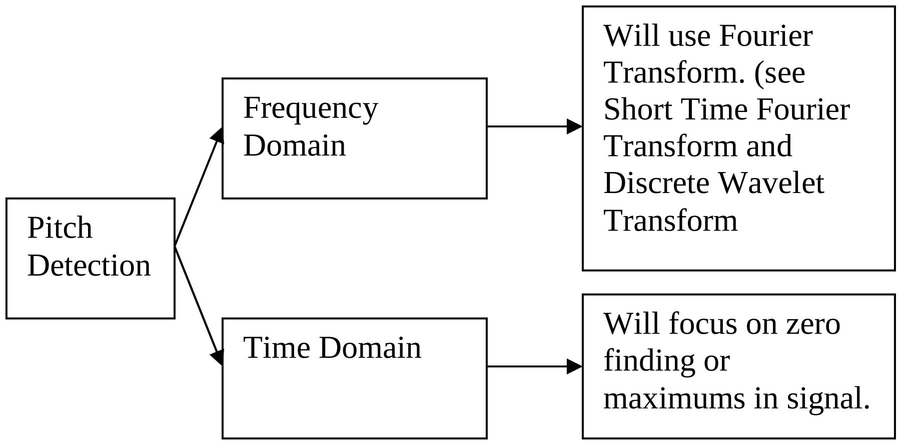
We proposed to develop a DSP software package that allows a user to synthesize music based on vocal input. This way, the user can hum into a microphone and have a selectable synthesized output based on the intensity and pitch of their voice. This package consists of pitch detection software and subtractive synthesis software. While individually these are widely available, and many algorithms exist for each, the pairing of the two into a music generation system is a novel concept. We are not aware of any existing commercial application of this nature.

What We'll Do



There are two main components to this project: the input portion and the output portion. For input, we have a microphone, which is mono, 16 bit sampled at 16 KHz. This input is analyzed for pitch, volume, and intonation, and then these parameters are passed on to the output module. The output module is a traditional subtractive synthesis model software synthesizer. Initially, we implemented a very simple subtractive synthesis model, and a very simple input model that only gives the pitch of the fundamental frequency of the voice, and detects when to cut off the synthesis (volume drops below a certain threshold).

Pitch Detection (Fundamental Frequency Estimation)


Pitch Detection
Frequency Domain
Time Domain
Will use Fourier Transform. (see Short Time Fourier Transform and Discrete Wavelet Transform
Will focus on zero finding or maximums in signal.

The first step is to get the right pitch. To do this several algorithms [1] were available. Some of them used time domain techniques (such as FFT) [2] and others work in time domain (and will use zero crossing…).

In the past years both techniques have been used. However frequency domain pitch detection seems to be less reliable and hard to implement on EVM. Also several projects involved musical instruments. Ours will use human voice, so this should also simplify the problem.

We have found several different algorithms, after testing them in MATLAB and in C, we decided that autocorrelation was the most accurate and the most efficient. Thus our pitch detection algorithm is based on autocorrelation.

Subtractive synthesis

Subtractive synthesis is a very well established method for generating realistic-sounding instrumental sounds. It is very robust in its range of possible pitches, durations, and volumes that it can produce. It is one of the oldest synthesis methods. While modern technology had introduced new types of synthesis such as FM synthesis, Wavetable synthesis, and even experimental synthesis methods such as grain synthesis and mathematical modeling synthesis, many commercial products are still based upon tried and true subtractive synthesis methods.

How It Works

The main components of subtractive synthesis are the initial waveform, the filter, and the envelope.

The initial waveform is generally a harmonically rich, easily generated waveform. Traditionally, the triangle wave, sawtooth wave, and square wave (or impulse train) are used. Sine waves are of little use for reasons that will become apparent. In the early days, these waveforms were generated with analog circuitry. In our project, the waveforms are created by equations in C code. Unfortunately, the entire purpose of these waveforms is that they contain lots of harmonics, which can cause aliasing problems if ignored. Normally when, say, a square wave is sampled, there will be an LPF in front of any analog to digital converter to filter out any frequencies higher than can be accurately sampled. If a square wave were created in the EVM's RAM using a simple function, it would be a perfect square wave sampled without any filtering. At low frequencies this is less noticeable, but aliasing can become more apparent. For this reason, bandlimited waveforms need to be generated, which is somewhat more complicated, but still not terribly computationally intensive.

The next step in the synthesis process is the filter. Generally this is a low-pass filter of some type that cuts out the higher order harmonics in the original signal. How many and to what extent they are diminished determines the quality of the sound that is created. The filter doesn't have to be LPF, but can instead be some type of notch, band-pass, or HPF to achieve interesting results. Generally, though, when trying to simulate real-world instruments, the filters are mostly LPF. In our project, this stage of the synthesis is implemented by an IIR filer function.

The third portion of the synthesis is the gain envelope. The envelope determines the volume of the signal, depending on the time passed and user input. It is generally organized in to four sections – attack, decay, sustain, and release. The attack section is the initial rise in volume from zero to the peak volume of the signal, which occurs when the note is first sounded. Decay is the tapering off of the attack portion, settling into a less loud state. Sustain is the longest state, which can be held indefinitely in some cases, of a moderate volume. Release is the final dropping off of the volume to zero. Once the envelope is done, the other steps can stop computing the waveform.

We decided on subtractive synthesis because it is a well established synthesis method, and can take up very little processing power. This allows the C67 DSP to be free for the pitch detection portion of our project, which is more computationally intensive, despite the fact that it will be dealing with data at a lower sampling rate. Also, the small memory requirements of the subtractive synthesis methods will possibly allow it to run with the internal EVM ram, making it faster still.

Putting It Together

Once we had the two portions of our code working, stitching them together shouldn't be too difficult. The pitch, duration, and volume data generated by the pitch detection module passes this data off to the synthesis module.

Week	Tasks	Who
2/24-3/02	Wrote Report Found Algorithms : - Pitch detection - Subtractive synth.	Brian, Yves Tadge
3/03-3/09	Tested different algorithms for pitch detection in Matlab. Subtractive synthesis in C, using PC sound card.	Brian, Yves Tadge
3/10-3/16	(same)	
3/17-3/23	17: Oral Update Implementation on EVM: - Pitch detection - Subtractive synth.	Brian, Yves Tadge
3/24-3/30		
3/31-4/6	Link pitch detection and subtractive synth.	All
4/7-4/13		
4/14-4/20	Debugging	TBD
4/21-4/27	Optimizing	
4/28-5/4	28: Demo	
5/5	Report Due	

Speed Issues

Initially we had planned for a fully real-time implementation of our program. We assumed that the C67 DSP would be more than powerful enough to do some autocorrelation, some filtering, and the MCBSP I/O. Ultimately, it seemed that without optimization, it would not be possible to have acceptable sound quality in real time.

Optimization, unfortunately, created more problems than it solved. We first got most of the algorithms working properly, then changed the compiler settings to optimize. The pitch tracking lost all accuracy, and the synthesis stopped working. It took us a while to confirm that it was in fact the optimizing compiler that was having this effect. The ability to graph contents of the EVM's RAM while running was an invaluable one. Using the graph functions, we could pinpoint where the data fell apart. The input data from the microphone ADC worked under all optimization settings. The data of the autocorrelation output, however, looks completely different once optimization is turned on. Here are examples of a normal, working autocorrelation, and an autocorrelation output that has been mangled by the optimization. (These are autocorrelations of the same input waveform):

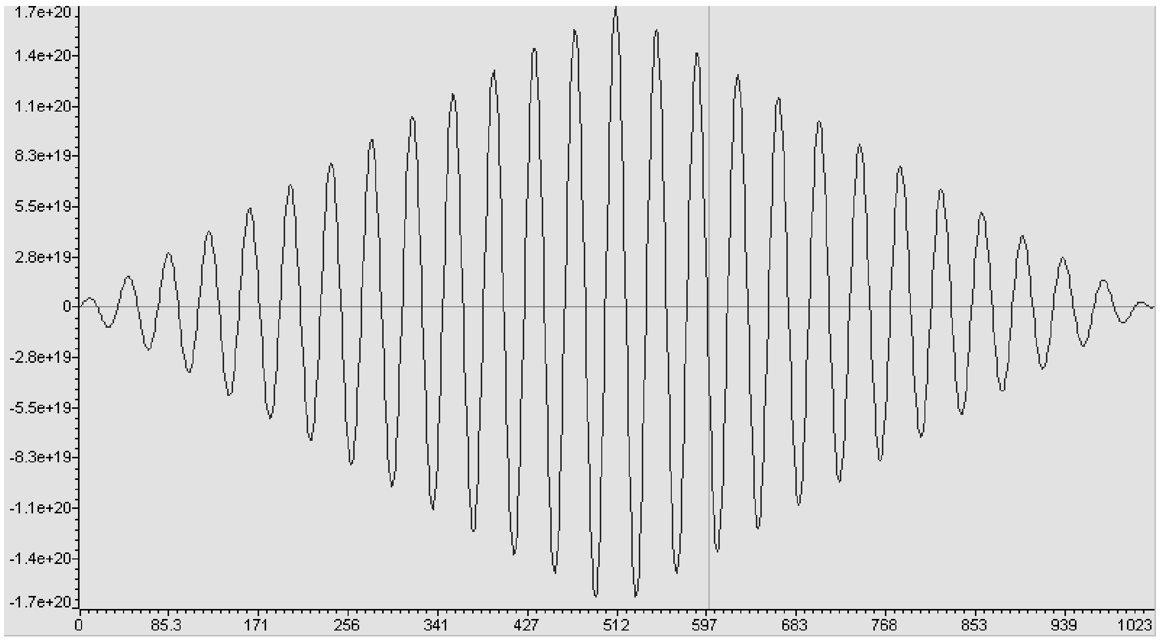


Figure 1: Normal Autocorrelation

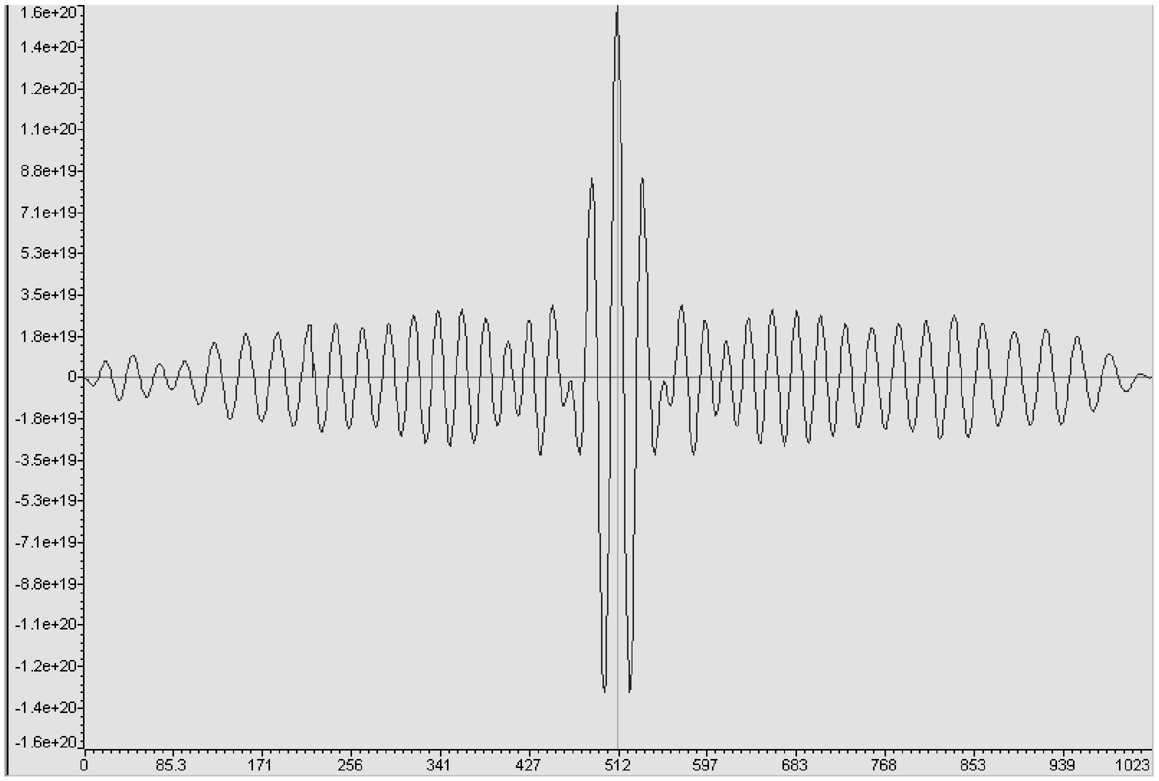


Figure 2: Autocorrelation with optimization

We tried to revise the autocorrelation algorithm, but seeing as it worked perfectly without optimization, it was difficult to find any problems with it. We tried a different method using FFTs to perform the autocorrelation, but that still gave incorrect output once the optimization was used.

We put the autocorrelation on the back burner, and did performance tests of everything else at given optimization levels. With optimization on, the EMV was capable of generating real time continuous signals for synthesis output. With the lower optimization levels that allowed to the autocorrelation to function, however, there were audible breaks in the output waveform. While these sometimes were not readily apparent on an oscilloscope reading, they could easily be heard as unwanted high frequency components of an otherwise low frequency signal.

Our compromise was then to move from a fully real-time implementation to a partially real time implementation. We accepted audio input from the microphone in real time, autocorrelated, did the pitch detection algorithm, and saved the value obtained in an array.

Once we detected that the user had stopped giving input, which was triggered by a few frames of relative quiet, we stopped the input and autocorrelation / pitch detection steps, and rendered the audio. This was the non-real time step, where the program read through the saved pitch values and synthesized notes for them. The synthesized notes were stored in a large buffer in the EVM’s off-chip RAM. Once the synthesis was complete, this stored waveform was sent out to the codec. Nothing else was running while the output buffer was being played, which ensured that the output waveform was clean and continuous.

Our theory for why the output became choppy is that the interrupts were not being serviced properly. When the synthesis and output were occurring in real-time, it seemed that the interrupts were either not being called as often as they should, or that they were being called and the program didn’t have the resources to move the data into the codec output registers.

Profiling

As we found in the earlier labs, the profiling function of the EVM software was not as robust and one might have hoped. Trying to profile more than one function at a time wouldn't work, but more importantly, the profiler seemed to slow the execution of the code so drastically that some functions would never complete in a reasonable amount of time. Enabling optimization seemed to alleviate this problem somewhat. We couldn't use optimization, however, and so the profiler couldn't generate useful data.
When profiling the peak detection, autocorrelation, or synthesis functions, the EMV never completed a single iteration of the function, even after as much as 5 minutes of

waiting. The elapsed cycles would slowly tick upwards, but at nowhere near the supposed clock rate of 166MHz.

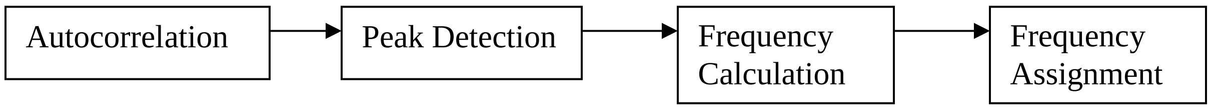
So, as many other groups did in the lab, we resorted to a much less exact, but functional profiling system: the stopwatch. We had code to do autocorrelations and peak detections, and would time how long different numbers of them took. In the end, it seemed that we got about 5 peak detection cycles per second. This wasn't quite fast enough for very quickly changing notes to be synthesized, but it could track notes held a bit longer quite well.

Description of Algorithms

Autocorrelation and Pitch Detection

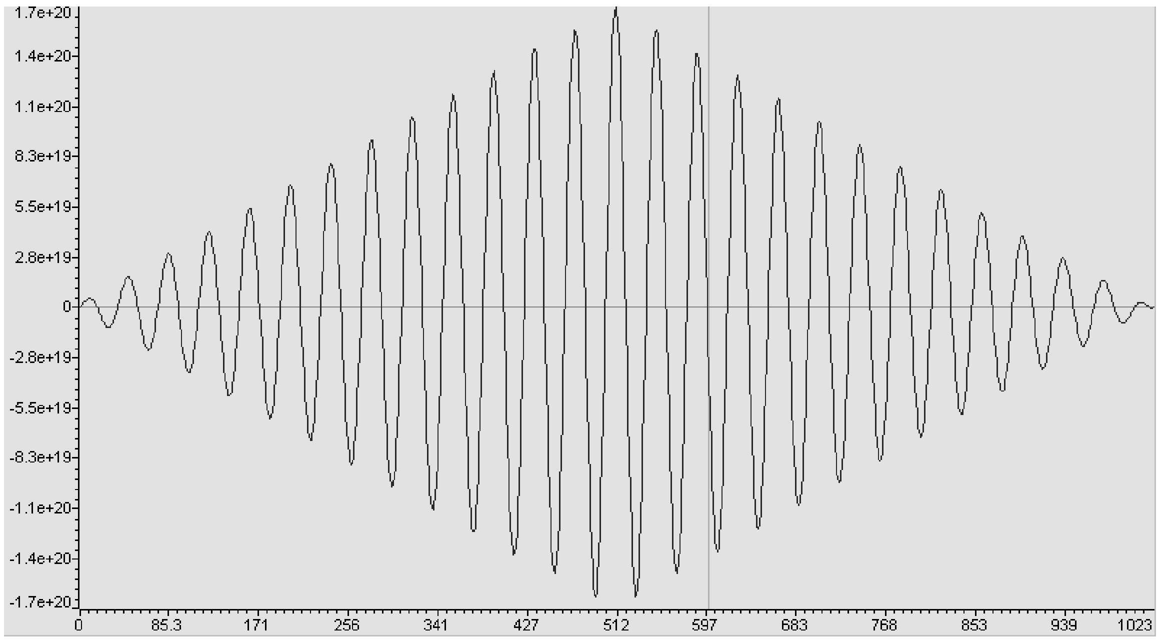
The Autocorrelation and Pitch Detection portion of our project works as follows:

1.) Autocorrelate the input (work) buffer.
2.) Use peak detection to find the number of peaks and the index of the last found peak.
3.) Calculate the frequency using the values in step 2.
4.) Assign the frequency to be the closest musical frequency to the detected frequency.



Autocorrelation

Our autocorrelation algorithm is very similar to what we did for Homework 1. Autocorrelation involves correlating a signal with itself. When this is done with a periodic signal, such as human humming, the result is a waveform with many peaks. From this waveform we can detect the pitch of in input signal.



Our input array, work[] is always of size 512, so the output array, corr, is always twice as long, or 1024. The algorithm consists mainly of 2 double for loops. The outer loop cycles through the corr array from k=0 to k=512. Then in the inner loop, which goes from 0 to k, we multiply the appropriate indices of work together and accumulate them in the corr index k.

```
for (k = 0; k<BUFFER_LEN; k++) {
        corr[k] = 0;
        blk=BUFFER_LEN-k;
        for (m=0; m<k+1; m++) {
              corr[k] += work[m] * work[blk+m-1];
        }
}
```

For example, when k=3, we multiply work[3], work[2], work[1], and work[0] with work[511], work[510], work[509], and work[508] respectively. The sum of all these products is then stored in corr[3]. The second double for loop is essentially the same as the first, except this time k moves through corr from 512 to 1023.

As we tested our peak detection and autocorrelation algorithms, we noticed that we would sometimes get a frequency and an output sound with no intentional sound going into the microphone. To correct this problem, we placed a power check at the beginning of our autocorrelation function. We obtained a value for the power in the signal by adding up all the values in the work array and dividing by 1,000,000. To stop the program from outputting unwanted sound, we only did autocorr if the power was above 7500. Creating this threshold stopped the program from outputting random sounds from random noise as input.

Peak Detection

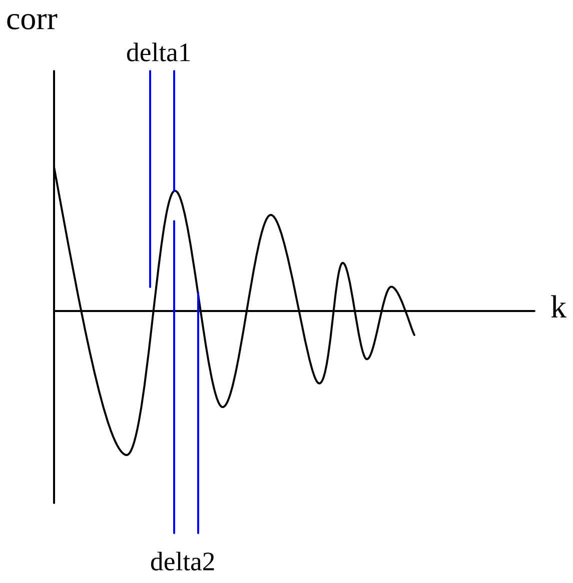
We used Group 9 – 2000 's peak detection algorithm as a model for ours. There are four conditions under which we check for a peak.

The first condition is that delta1, difference between the current index of corr and previous index of corr, is greater than zero.

delta1 = corr[k] – corr[k-1]

The second condition is that delta2, the difference between the next index of corr and the current index of corr, is less than zero.

delta2 = corr[k+1] - corr[k]



The third condition is that the value of the potential peak must be at least 80% of the value of the first peak. This is to ensure that peaks are not detected from imperfections in the corr array.

The fourth condition, which will be explained later, is that we have counted less than 6 peaks so far.

If all of these conditions are satisfied, then the algorithm is close to detecting a peak. Since we are not dealing with a perfect autocorrelation signal, it is necessary to examine the values around the current index. The function checks from 9 samples back to 14 samples forward to search for the maximum value. When the maximum value is found, the index of that value is stored and the number of peaks detected is increased.

```
int swit = 1;
for(n = -9; n < 14; n++){
        if(corr[k+l] > corr[k])
                swit = 0;
        if(swit){
                if(first == -1)
                first = k;
        }
 }
peakIndex = k;
peakCounter++;
```

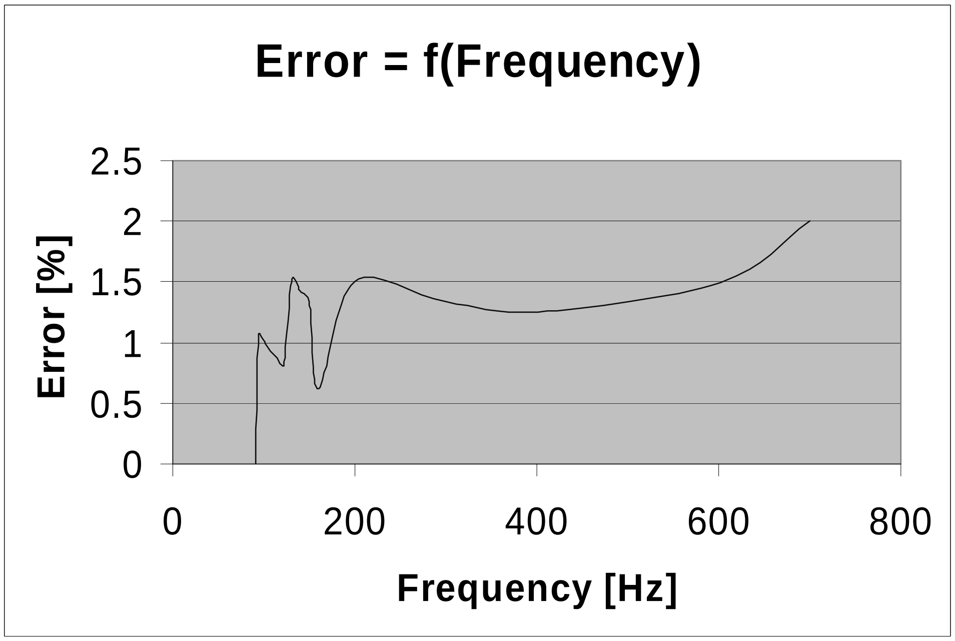
After we have detected a peak and stored its index, we can then calculate the frequency of the signal. The formula we used is

$$f = \frac{f_s \times peaks}{index}$$

Where *f* is the fundamental frequency, f_s is the sampling rate, *peaks* is the number of peaks counted, and *index* is the index of the last detected peak.

We found that the more peaks we counted, the more accurate this equation became. Since every signal we were working with would have at least 6 peaks, that is how many peaks we chose to count.

Once these two algorithms have been performed, we have now detected the fundamental frequency of our input signal. While testing these algorithms however, we found that we were accurate within 1.5% of the frequency of the input signal.



The detected frequency would change within these few Hertz and produce a change in output while the input voice remained constant. To correct this problem, we added a few lines of code that would assign the detected frequency to be the frequency to be the value of the nearest musical pitch.

To do this, we created an array, notes, of 36 musical frequencies from 65.406 Hz to 493.883 Hz. We then searched through this array until we found the first value that was greater than our detected value. Let notes[k] equal this value.

If the difference between notes[k] and the detected frequency was greater than the difference between the detected frequency and notes[k-1], then the detected frequency was assigned to be notes[k-1]. Otherwise it was assigned to be notes[k].

We now pass this frequency on to the synthesis portion of our project.

Pitch Stabilization

1) We scan the frequency array and if we have the following case:

100Hz 200Hz 100Hz

The middle frequency will be changed to 100Hz, resulting in a more stable pitch.

```
if(abs(farray[i-1]-farray[i+1])<1) // comparison between floats
        farray[i]=farray[i+1];
```

This modification will change the values in farray in figure A to those shown in figure B.

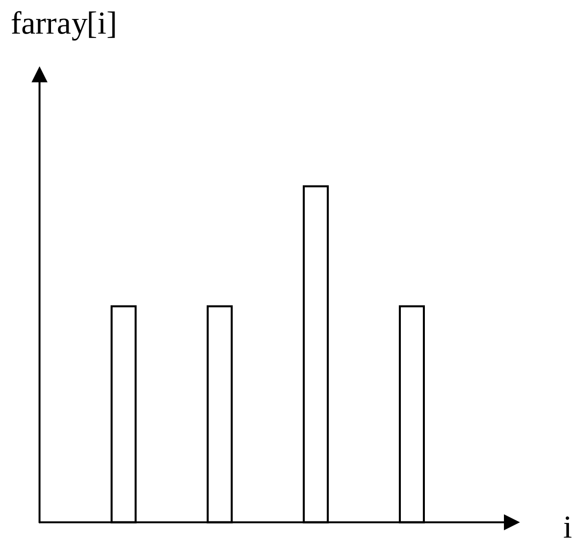


Fig. A

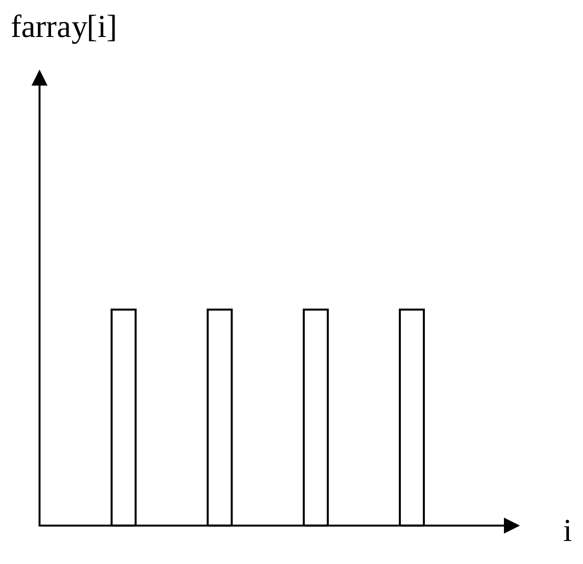
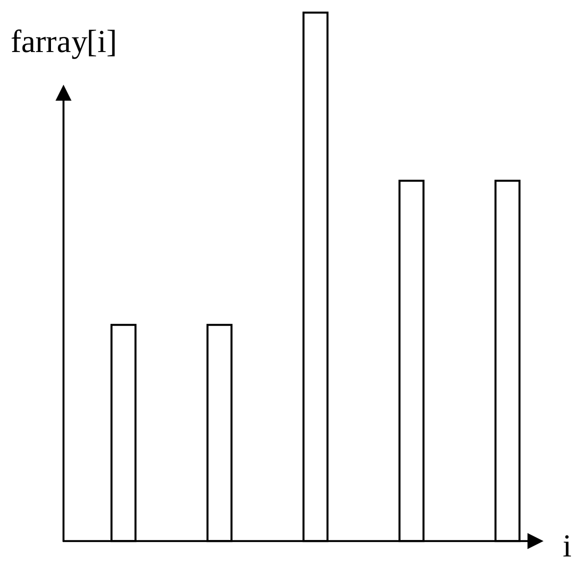
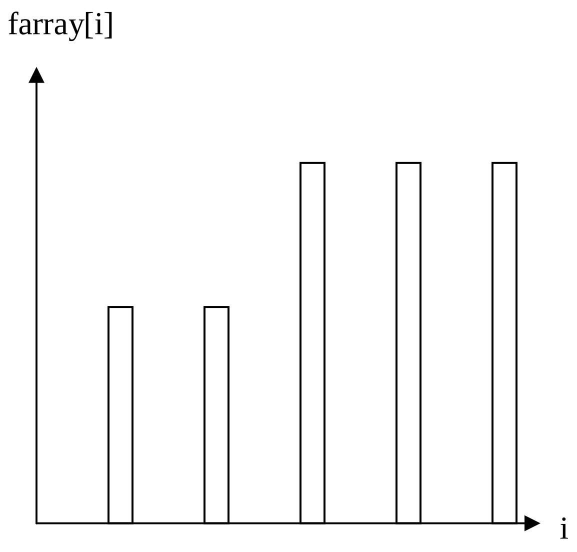


Fig. B

2) If we have 3 different frequencies in a row, 100Hz 200Hz 300Hz we will change this to 100 Hz 300 Hz 300 Hz

```
if((abs(farray[i-1]-farray[i+1])>1 )&& (abs(farray[i-1]-farray[i])>1) &&(abs(farray[i]-
farray[i+1])>1)){
                        farray[i]=farray[i+1];
```





These two algorithms take care of problems we had when the pitch changed within a frame.

Synthesis:

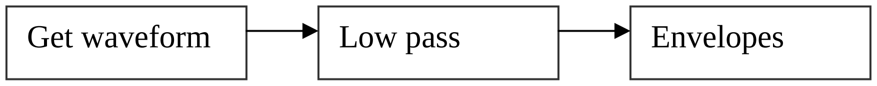
Intruduction

In order for our project to be complete we have to get some sound out of the EVM. The way we do it is called subtractive synthesis.

Some of the implementation methods and specifics came form the freely available CSound Porject [3].

This is done in three parts:

1) generate the right waveform at the detected frequency.
2) low pass
3) envelopes.



In the next lines I will explain each of these stages

1) Get waveform

Generating the waveform is pretty easy, the user has the choice between 4(sine, square, triangle, sawtooth).

The sinewave is generated the following way:

```
bigarray[y_index] = (short)(10000 * sinf(2*3.14159* y_index *
farray[frame]/OUTPUT_RATE));
```

Where bigarray is the output array (mallocated before), farray[frame] is the corrected frequency for the given time window and y_index gets incremented for each sample.
Notice that this has to be casted into a short.
To generate the 3 remaining waveforms the idea is the same, so I will just explain the triangle wave.

```
if( (y_index % (int)(16000/farray[frame])) < (8000/farray[frame]) )
                                bigarray[y_index] = 10000;
                                else
                                bigarray[y_index] = -10000;
```

(int)(16000/farray[frame])) is the number of samples in one period.
So y_index % (int)(16000/farray[frame] will go to zero each period.
(8000/farray[frame] is half the period.
So what the if statement does is simply switching between -10000 and 10000 each half period.

2) Filtering

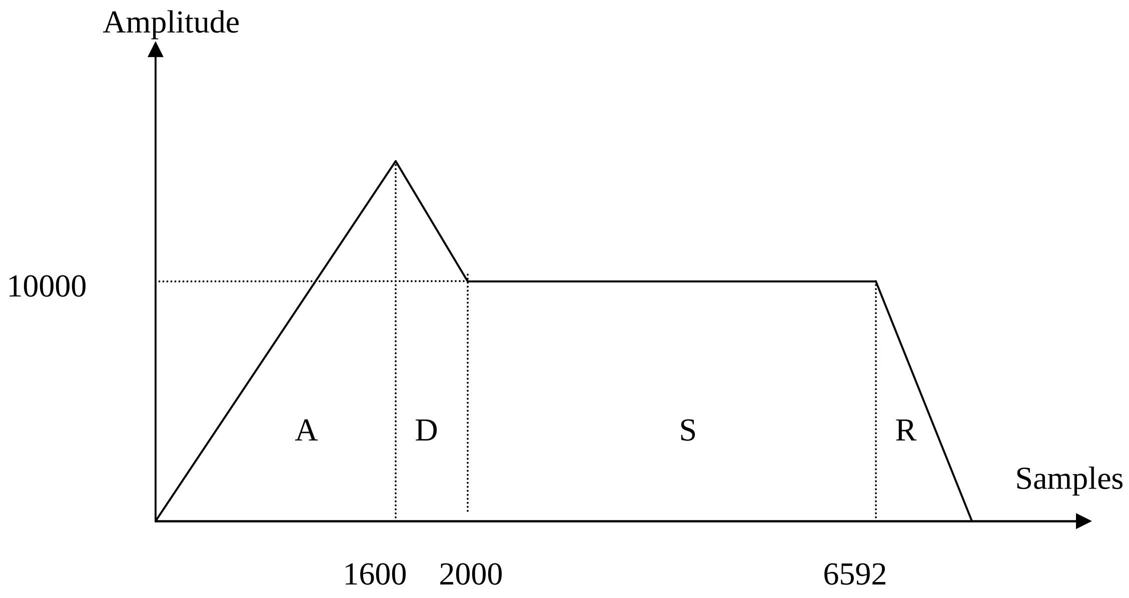
Filtering, first we hade implemented a low pass filter using IIR (matlab), later one we simply used the algorithm from lab 1.

```
bigarray[k] += (bigarray[k-1] >> 3) + (bigarray[k-2] >> 3) + (bigarray[k-3] >> 3) +
(bigarray[k-4] >> 3)
```

3) Envelopes

The tricky part of our code are the ADSR envolopes.
We made the following assumptions:

The attack phase is 100ms (1600 samples @ 16kHz)
The decay phase is 25ms
And the release phase is also 100ms.



Now I will briefly explain how "attack" works, the other phases use the same ideas.

```
if(((y_index%8192) < 1600)&&(farray[frame]!=farray[frame-1]))

        bigarray[y_index]=(short)((2.0*(float)bigarray[y_index]*(y_index%8192))/1600.)
```

the "if" statement will check if the given part of the output sample is in "attack", if yes we simply adjust the amplitude according to his position (we use a straight line starting at 0 and with a slope of 2).

Release will be exactly the same except for that the straight line will have a different equation.

Of course we have a release phase only if the next frequency window is different, so we add the following condition: &&(farray[frame]!=farray[frame+1])

Synthesis Conclusion:

We found lot's of papers describing subtractive synthesis, however they only give the general idea. Getting the actual numbers used to get a "real" sound out of the EVM were quite impossible to find. However each individual part of the synthesis algorithm works. It's not Mozart but it sounds like humming.

```
/****************************************************************
 * 18-551 Group 9:sing synth                                    *
 * echo.c                                                       *
 * Authors:(tpd,bgj,ystautte@andrew.cmu.edu                     *
 ****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <mcbsp.h>              /* mcbsp devlib */
#include <common.h>
#include <mcbspdrv.h>           /* mcbsp driver */
#include <board.h>              /* EVM library */
#include <codec.h>              /* codec library */
#include <mathf.h>              /* math library */
#include <intr.h>               /* interrupt library */
#include <linkage.h>
#include <string.h>
#include <timer.h>
#include <pci.h>

#define pi 3.14159
#define BUFFER_LEN 512
#define OUTPUT_RATE 16000
#define FILT_ORDER 5

short *bigarray;
float work2[BUFFER_LEN];
int rindex=0;                   /* Index for receive ISR */
unsigned long xindex=0;                   /* Index for transmission ISR
*/
int buffer[BUFFER_LEN*2];
float work[BUFFER_LEN];       /* Memory buffer for samples */
unsigned long y_index=0;
int silent_frames,i;
short fsmodf;
int frame = 0;
int final_frame;
char buffer_sel = 0;
char synth_go =0;
char playing = 0;
float corr[(2*BUFFER_LEN)-1];
int max;
int frequency;
int prevfreq;
int cur_cycle=0;
float farray[500];
float power=0.;
float first = -1;
int ii;
int type = 2;
float peakCounter = 0;
float peakIndex;
float delta1, delta2;
float notes[36]= {1, 69.296, 73.416, 77.782, 82.407, 87.307,
  92.499, 97.999, 103.826, 110.000, 116.541, 123.471,
  130.813, 138.591, 146.832, 155.563, 164.814, 174.614,
  184.997, 195.998, 207.652, 220.000, 233.082, 246.942,
  261.626, 277.183, 293.665, 311.127, 329.628, 349.228,
  369.994, 391.995, 415.305, 440.000, 466.164, 493.883};//natural notes


unsigned int f;
unsigned int message=0;
int blk;
```

```
//autocor actually does autocorr, peak detection and frequency
correction
void autocorr()
{

      int k, l;

      power = 0;
      for(k=0; k<BUFFER_LEN; k++)
      {
            work2[k] = work[k];
      }

      for(k=0; k<BUFFER_LEN; k++)
      {
            power += abs(work2[k]) / 1000000; // power gives an
indication of how strong the imput is
      }


      if(power>100000)
      {

            //autocorrelation
            for(k=0; k<BUFFER_LEN; k++)
            {
            corr[k] = 0;
            for (l=0; l<k+1; l++)
            {
                  corr[k] += work2[l] * work2[BUFFER_LEN-k+l-1];

            }
                  if(corr[k]>max)

      }




      for(k=BUFFER_LEN; k<(BUFFER_LEN*2)-1; k++)
      {
            corr[k] = 0;
            for(l=k-BUFFER_LEN+1; l<BUFFER_LEN; l++)
            {
                        corr[k] += work2[l] * work2[(BUFFER_LEN-
k)+l-1];
                     }

               if(corr[k]>max)

          }




      // peak detection
      first = -1;
      peakCounter = 0;

      for(k=520; k< 1000; k++)
      {
            delta1 = corr[k] - corr[k-1];
            delta2 = corr[k+1] - corr[k];
            // here we check if
            //1) it is a potential peak
            //2) it is at least 0.8 times the main peak (middle of
autocorr (sizeof(autocorr)=1024)
            //3) it is at most the 6th peak
            if( ( delta1 >= 0 ) && ( delta2 < 0 ) && (corr[k] >
corr[512] * 0.8)&&(peakCounter<6) )
            {
                  int swit = 1;
                  for(l=-9; l<14; l++) // look around if there is a
higher point
                  {
                        if(corr[k+l] > corr[k])
                              swit = 0;

                        if(swit) // it is a peak
                        {
                              if(first == -1)
                              first = k;




                        }



                  }
                  peakIndex = k; // index of the last peak
                        peakCounter++; // number of peaks

                        frequency= (8000.*(peakCounter)/(peakIndex-
512.)); // compute the frequency

            }





      }




      }
else{
            frequency = 1; // default case
            }




       k=0;
       // natural notes, will simply compare the note to the reference
array and round up to the closest
      while(k < 36)
      {
            if(frequency > notes[k])
            {
                  k++;
            }
            else
            {
                  if( ( notes[k] - frequency ) > (frequency - notes[k-
1]) )
```

```
                  {
                        frequency = notes[k-1];
                  }
                  else
                  {
                        frequency = notes[k];
                  }
                  k=36;
            }
      }


}

      //low pass filter
void filter()
{
      int k;

      for(k=1; k<y_index; k++)
      {
            bigarray[k] += (bigarray[k-1] >> 3) + (bigarray[k-2] >> 3) +
(bigarray[k-3] >> 3) + (bigarray[k-4] >> 3);
      }
}



interrupt void rcvISR(void)
{
      int q;
      buffer[rindex++]=MCBSP0_DRR;
            if (rindex>BUFFER_LEN*2)
            {
            for(q=0; q<BUFFER_LEN; q++)
            {
                  work[q] = (float)buffer[2*q];
            }
          rindex=0;
            }
}


interrupt void xmitISR(void)
{
  /* MCBSP0_DXR is the hardware register for outgoing samples */
      if(playing == 1) // variable that is set in main, once the outpub
buffer is full
      MCBSP0_DXR = bigarray[xindex++];
      else
      MCBSP0_DXR = 0;

}


/****************************************************************
 *    Name: main
            *
 ****************************************************************/
int main(void) {
  Mcbsp_dev dev;                /* Serial port device */

  evm_init();                   /* Standard board initialization */
  printf("Program Running\n");
  printf("Configuring MCBSP...");
  mcbsp_drv_init();             /* Call this before using McBSP
functions */

  f = 0;
  /* Open serial port */
  if (!(dev=mcbsp_open(0))) {
    return(ERROR);
  }
  /* Configure McBSP */
  mcbsp_setup(dev);   /* See bottom of this file */
  printf("done\n");

  /******************** configure CODEC **********************/
  printf("Configuring codec...");
  /* EXIT_ERROR is a macro which jumps to exit_err if the function
     returns an ERROR */
  EXIT_ERROR(codec_init());
  codec_change_sample_rate(16000, TRUE);
  /* A/D 0.0 dB gain, turn off 20dB mic gain, sel (L/R)LINE input */
  EXIT_ERROR(codec_adc_control(LEFT,0.0,TRUE,MIC_SEL));
  EXIT_ERROR(codec_adc_control(RIGHT,0.0,TRUE,AUX1_SEL));
  /* mute (L/R)LINE input to mixer */
  EXIT_ERROR(codec_line_in_control(LEFT,MIN_AUX_LINE_GAIN,FALSE));
  EXIT_ERROR(codec_line_in_control(RIGHT,MIN_AUX_LINE_GAIN,FALSE));
  /* D/A 0.0 dB atten, do not mute DAC outputs */
  EXIT_ERROR(codec_dac_control(LEFT, 0.0, FALSE));
  EXIT_ERROR(codec_dac_control(RIGHT, 0.0, FALSE));
  printf("done\n");

  /**************** setup interrupt routines ******************/
  printf("Initializing interrupts...");
  intr_init();
  /* Hook up serial transmit interrupt to CPU Interrupt 14 */
  intr_map(CPU_INT14,ISN_XINT0);
  INTR_CLR_FLAG(CPU_INT14);     /* Clear any old interrupts */
  intr_hook(xmitISR,CPU_INT14); /* Hook our own xmitISR into chain for
14 */
  /* Repeat the same process for the receive interrupt */
  intr_map(CPU_INT15,ISN_RINT0);
  INTR_CLR_FLAG(CPU_INT15);
  intr_hook(rcvISR,CPU_INT15);
  /* Enable all necessary interrupts */
  INTR_ENABLE(CPU_INT_NMI);     /* Non-maskable interrupt */
  INTR_ENABLE(CPU_INT14);
  INTR_ENABLE(CPU_INT15);
  INTR_GLOBAL_ENABLE();         /* Controls whether ANY interrupts
function */
  printf("done\n");

  /****************** Turn on the serial port **********************/
  MCBSP_ENABLE(dev->port,MCBSP_RX|MCBSP_TX);

  /* At this point, the program leaves main and enters an infinite
   *  idle loop.  Interrupts continue to function */


  bigarray = (short *)malloc(500*1024*sizeof(short)); //output array
  if(bigarray==NULL)
  {
    printf("Couldn't allocate memory...\n");
    exit(1);
  }
  playing = 0;


  while (1)
  {
      printf("waiting stage\n");
      while(! (power>100000 && (frequency <400) && (frequency >80) ) )
      {
            autocorr(); // waiting for some singing

      }

      printf("Recording stage\n");

      silent_frames = 0;

      while( frame < 500  && silent_frames < 2)
      {
            autocorr();

            farray[frame] = frequency; // we store the frequency
            frame++;//and jump to the next frame

            if (!(power>100000 && (frequency <400) && (frequency >80) )
)
            {
                  silent_frames++;
            }
            else
            {
                  silent_frames = 0;//reset silent frame if we hear
singing again
            }
      }

      final_frame = frame-2; // we don t want to play the two silent
frames

      // modification to get a better stability,if in the frequency
array we have the following sequence:
      // 100Hz 200Hz 100Hz we will change it to: 100Hz 100Hz 100Hz
      for( i=1; i<frame; i++){
            if(abs(farray[i-1]-farray[i+1])<1)
                  farray[i]=farray[i+1];
      }

      //other modification, 100Hz 200Hz 300Hz => 100Hz 300Hz 300Hz
      for(i=1; i<frame; i++){
            if((abs(farray[i-1]-farray[i+1])>1 )&& (abs(farray[i-1]-
farray[i])>1) &&(abs(farray[i]-farray[i+1])>1)){
                  farray[i]=farray[i+1];
            }
      }


      printf("Rendering stage, %d frames\n", final_frame);
      for(frame = 0; frame < final_frame; frame++)
      {
            for(y_index = 8192*frame; y_index < ((8192 * frame)+ 8192);
y_index++)
            {

                  if(type == 1)
                  {// sinewave
```

```
                        bigarray[y_index] = (short)(10000 *
sinf(2*3.14159* y_index * farray[frame]/OUTPUT_RATE));
                  }

                  if(type == 2)
                  {//square
                        if( (y_index % (int)(16000/farray[frame])) <
(8000/farray[frame]) )
                        bigarray[y_index] = 10000;
                        else
                        bigarray[y_index] = -10000;
                  }

                  if(type == 3)
                  {//triangle
                        if( (y_index % (int)(16000/farray[frame])) <
(8000/farray[frame]) )
                        bigarray[y_index] = -1000 +
(y_index%16000/farray[frame]) * (20000/(8000/farray[frame]));
                        else
                        bigarray[y_index] = 1000 -
(y_index%16000/farray[frame]) * (8000/farray[frame]);

                  }

                  if(type == 4)
                  {//sawtooth
                        bigarray[y_index] = -10000 +
(y_index%(int)(16000/farray[frame]))*(16000/farray[frame]);
                  }



            }
      }

      filter(); //we LPF the whole array

      //envelope
      for(frame = 0; frame < final_frame; frame++)
      {
            for(y_index = 8192*frame; y_index < ((8192 * frame)+ 8192);
y_index++)
            {//attack
             // the first condition sets the timing
             // the second checks if the frequency in this fram is
different (no attack otherwise)
                  if(((y_index%8192) <
1600)&&(farray[frame]!=farray[frame-1]))

      bigarray[y_index]=(short)((2.0*(float)bigarray[y_index]*(y_index%8
192))/1600.);

            //decay
                  if(((y_index%8192) >= 1600 )&&(y_index < 2000) &&
(farray[frame]!=farray[frame-1]))

      bigarray[y_index]=(short)((float)bigarray[y_index]*(6.-
((float)(y_index%8192))*(0.0025)));
            //sustain is the default case, so there is nothing to change

            //release
                  if(((y_index%8192) >
6592)&&(farray[frame]!=farray[frame+1]))
```

```
      bigarray[y_index]=(short)((float)bigarray[y_index]*(5.-
((float)(y_index%8192))*0.00061));
            }

      }


      printf("Playing stage\n");
      playing = 1;
      while(xindex < y_index)
      {}
      printf("Done, reset\n");

      playing = 0;
      xindex =0;
      y_index =0;
      frame = 0;

       }
  exit_err:
  return(ERROR);
}


/****************************************************************
 *    Name: mcbspSetup
 *  Inputs: Mcbsp_dev
 *  Output: none
 * Purpose: McBSP stands for Multi-Channel Buffered Serial Port.
 *  It is build onto the C67 processor itself, and is how the
 *  codec communicates with the processor.  This function sets
 *  up the serial port for communication with the codec, and
 *  should never need to be modified.
 ****************************************************************/
int mcbsp_setup(Mcbsp_dev dev) {
  /* Structure with all configuration parameters for serial port */
  Mcbsp_config mcbspConfig;
  memset(&mcbspConfig,0,sizeof(mcbspConfig)); /* Initialize everything
to 0 */

  mcbspConfig.loopback             = FALSE;
  mcbspConfig.tx.update            = TRUE;
  mcbspConfig.tx.clock_polarity    = CLKX_POL_RISING;
  mcbspConfig.tx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.tx.clock_mode        = CLK_MODE_EXT;
  mcbspConfig.tx.frame_sync_mode   = FSYNC_MODE_EXT;
  mcbspConfig.tx.phase_mode        = SINGLE_PHASE;
  mcbspConfig.tx.frame_length1     = 0;
  mcbspConfig.tx.word_length1      = WORD_LENGTH_32;
  mcbspConfig.tx.frame_ignore      = FRAME_IGNORE;
  mcbspConfig.tx.data_delay        = DATA_DELAY0;

  mcbspConfig.rx.update            = TRUE;
  mcbspConfig.rx.clock_polarity    = CLKR_POL_FALLING;
  mcbspConfig.rx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.rx.clock_mode        = CLK_MODE_EXT;
  mcbspConfig.rx.frame_sync_mode   = FSYNC_MODE_EXT;
  mcbspConfig.rx.phase_mode        = SINGLE_PHASE;
  mcbspConfig.rx.frame_length1     = 0;
  mcbspConfig.rx.word_length1      = WORD_LENGTH_32;
  mcbspConfig.rx.frame_ignore      = FRAME_IGNORE;
  mcbspConfig.rx.data_delay        = DATA_DELAY0;
  /* Pass entire structure to mcbsp_config, a library function which
   *  sets registers according to the contents of the structure */
  if(mcbsp_config(dev,&mcbspConfig) != OK) {
    printf("Couldn't configure McBSP device %i\n", dev);
    return(ERROR);
  }
  return(OK);
}
```

References

[1] “Comparative evaluation of F_o estimation algorithms” de Cheveigné, Kawahara. www.isit.or.jp/~ikuyo/paper.pdf, 2001

[2] “An Efficient Pitch-Tracking Algorithm Using a Combinations of Fourier Transforms” Marchand, www.csis.ul.ie/dafx01/proceedings/papers/marchand.pdf, 2001

[3] CSound Synthesis models. http://www.csounds.com/toots/index.html