Automatic Feedback Detection and Elimination Using TI 67xx DSP Processors
18-551, Digital Signal Processing and Communications Design Project
Final Report

7 May 2000
Group 17

Alison Greenwald
Ros Neplokh
Will Wong

1 Problem Description

1.1 What Feedback Is

Acoustic feedback is caused when an audio system boosts a signal so the microphone producing that signal receives it louder than the original level. This causes a signal to get continually louder until limited by some component of the system (such as a compressor/limiter or the microphone's maximum output). This is usually very distracting to an audience and can often be loud enough to injure people's ears.

Feedback can be problematic in a small setting, such as a speaker at a conference, though the problem is far worse at live musical concerts. This project aims to eliminate in a way conducive to such a setting. Any speaker can "ring" with feedback, though the most problematic are stage monitors. Artists use these to hear themselves, and therefore they are pointed directly at the microphone.

1.2 Traditional Elimination methods

Normally when a concert is being set up, engineers go through a process called "ringing out"; they attempt to induce feedback through the speakers then they reduce the gain on problematic frequencies with a graphic equalizer until a sufficient overall gain is reached.

This process requires talented ears to be able to detect which frequencies are ringing. Sometimes engineers use “real-time analyzers” or other methods of showing relative loudness of frequencies to help find the feedback, but it still takes a trained engineer to know what is truly the feedback. Occasionally there are rings during the performance, but ideally they should all be removed prior to the concert.

1.3 What feedback looks like

Feedback is generally constrained to a very narrow band of frequencies, and sometimes harmonics of those frequencies. Over time, feedback increases exponentially to where it reaches the limit of gain in the system. At that point it continues to ring at painful volumes until an engineer decreases gain or the performer move the microphone farther from the speakers.

In Figure 1 below, you see a song with feedback interjected in the middle of it (around 1.25 seconds in) at approximately 6kHz. The feedback grows very quickly from the noise floor and the frequency band is only about 4 bands wide (about 150 Hz). In figure 2, that one frequency is expanded so the development over time of this frequency can be watched. The y scale is in dB, so a linear growth actually represents an exponential growth in amplitude of the signal.

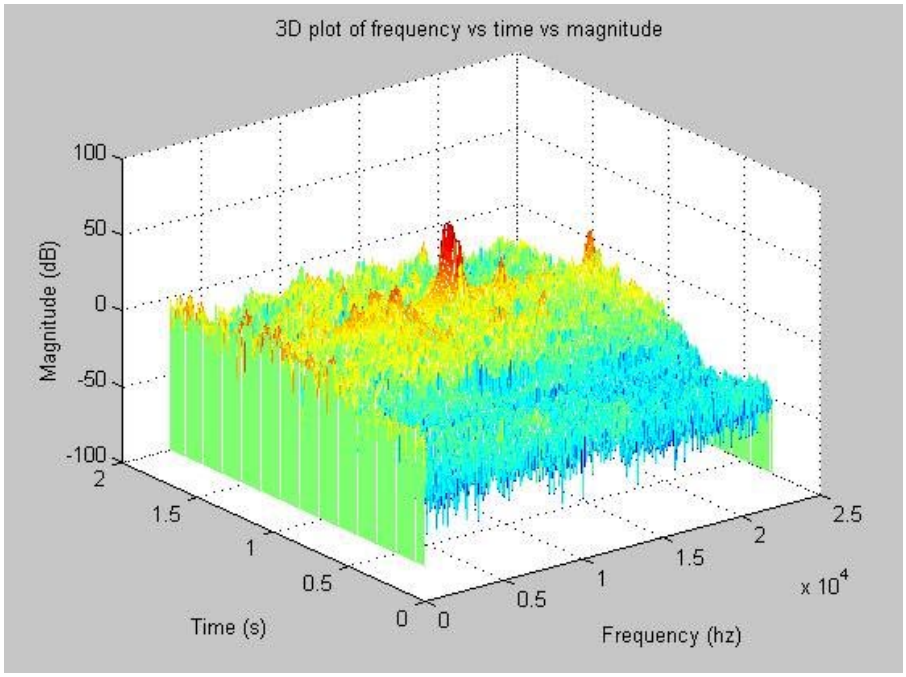

Figure 1

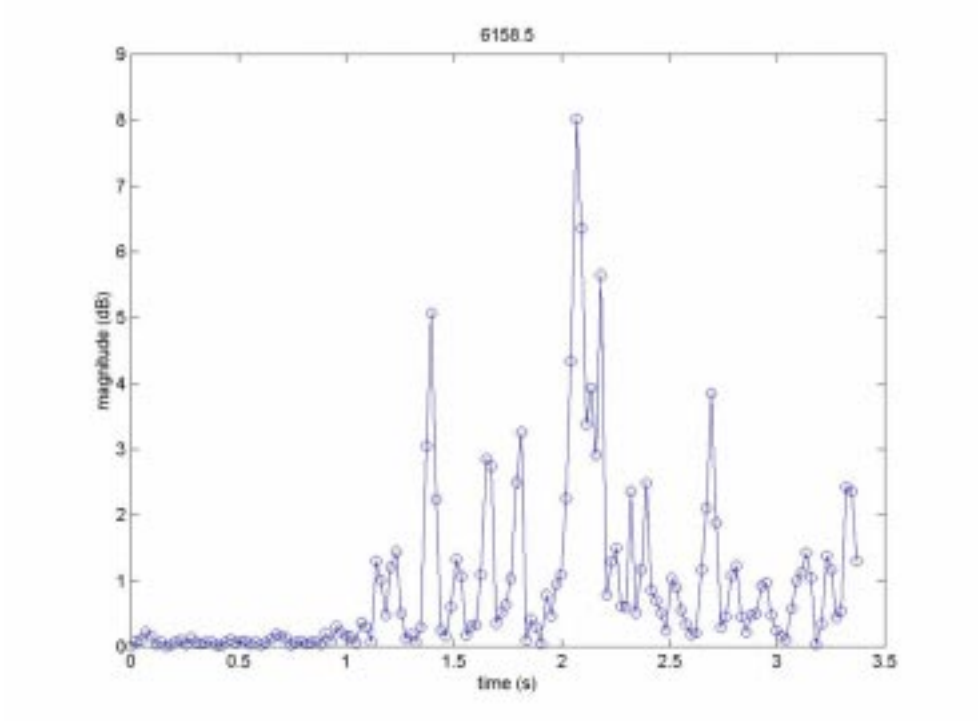

Figure 2

1.4 Approach to elimination

Sabine owns the patent on feedback elimination. Their method is looking at the FFT of a single time slice and determining what frequencies look strange. This is good, but since feedback grows as time goes on, in an exponential manner, a better method is to look at the changes of specific frequency components of the sound as time goes on.

Through experimentation, three types of feedback were found. Some of the feedback is barely sustained, and gets slightly louder over a large period of time. Unfortunately this is similar to a note that is being held and is hard to distinguish. Fortunately, only computers are capable of actually doing this, so the recommendation is to not use computerized devices or eliminate this feature.

The second type of feedback grows at a fast but measurable pace. By experimentation it seems to categorically increase by a factor of 2.5 every time slice (23 ms). 4-5 time samples are needed to verify that this is indeed feedback.

The final type of feedback increases quickly enough that it cannot be detected by watching increases over time. Within one or two time slices it reaches a threshold value for volume. The threshold method is rarely utilized in real life testing of the system.

2 Code

2.1 Algorithms

With this algorithm, the eight previous FFT values from the input sound are kept in a circular buffer. This number was chosen due to memory constraints, as all of the data is being kept on-chip. Were external memory to be used, many more previous samples could be kept. Current algorithms rely only on the immediately previous value. In Matlab, some more complex algorithms have been tested. These take into account multiple previous values and calculate a best-fit curve.

A general "value" on each frequency is held, this indicates the likelihood to be feedback. This value is overloaded for different operations as explained below.

2.1.1 Fast Growth, Slow Growth

Any time a frequency component is greater than or equal to its value in the previous time slice, "1" gets added to its feedback value. If this frequency component is greater than or equal to some factor times the previous value (2.5 in this implementation), it is marked as fast growing feedback and gets an additional "1000" added to it. If neither condition is satisfied, the value gets reset to 0.

```
    for(i=0;i<FFTSIZE/2;i++) {
        temp=buff[i];
        tracking[current][i]=temp;
        if(temp > MAXVOL) {
            finaltrack[i]=S_INAROW;
```

```
        } else if(((temp > THRESH) &&(temp > (tracking[((S_INAROW+current-
1)%S_INAROW)][i])*FACTOR))){
                if(time_running>10) {
                    finaltrack[i]+=1001; /* counts as fast and slow */
                }
        } else if(((temp > THRESH) &&(temp > (tracking[((S_INAROW+current-
1)%S_INAROW)][i])*S_FACTOR))){
                if(time_running>10) {
                   finaltrack[i]++;  /* just counts as slow */
                }
        } else {
                finaltrack[i]=0;
                time_running++;
                }
    }
```

Then, the value array is scanned to find any values that indicate feedback. For example, if a value modulo 1000 is greater than 8, it is marked as slow feedback and eliminated. If the value in itself is greater than 5000, it is fast feedback and eliminated as such.

```
    if(time_running>10) {
       for(i=0;i<FFTSIZE/2;i++) {
           if((((finaltrack[i]/1000) > (INAROW-1) )||
              ((finaltrack[i]%1000) > S_INAROW-1)) && (buff[i] > NOISEFLOOR)) {

               x=FREQ[i];
              insert_filter(i); /* code eliminated for space */
         }
       }
    }
```

2.1.2 Threshold

A certain threshold is set such that any time a frequency component exceeds that threshold even once, it is eliminated indiscriminately. The value is set to 8 instantaneously so it registers as feedback on the output.

2.1.3 Other Options

This value method is conducive to doing more calculations on the same value array, by using modulo 100 or 10000. In addition, only some amount could be subtracted rather

than resetting to 0 in the case of a dip. This would be helpful since feedback does not always grow monotonically.

2.2 Components

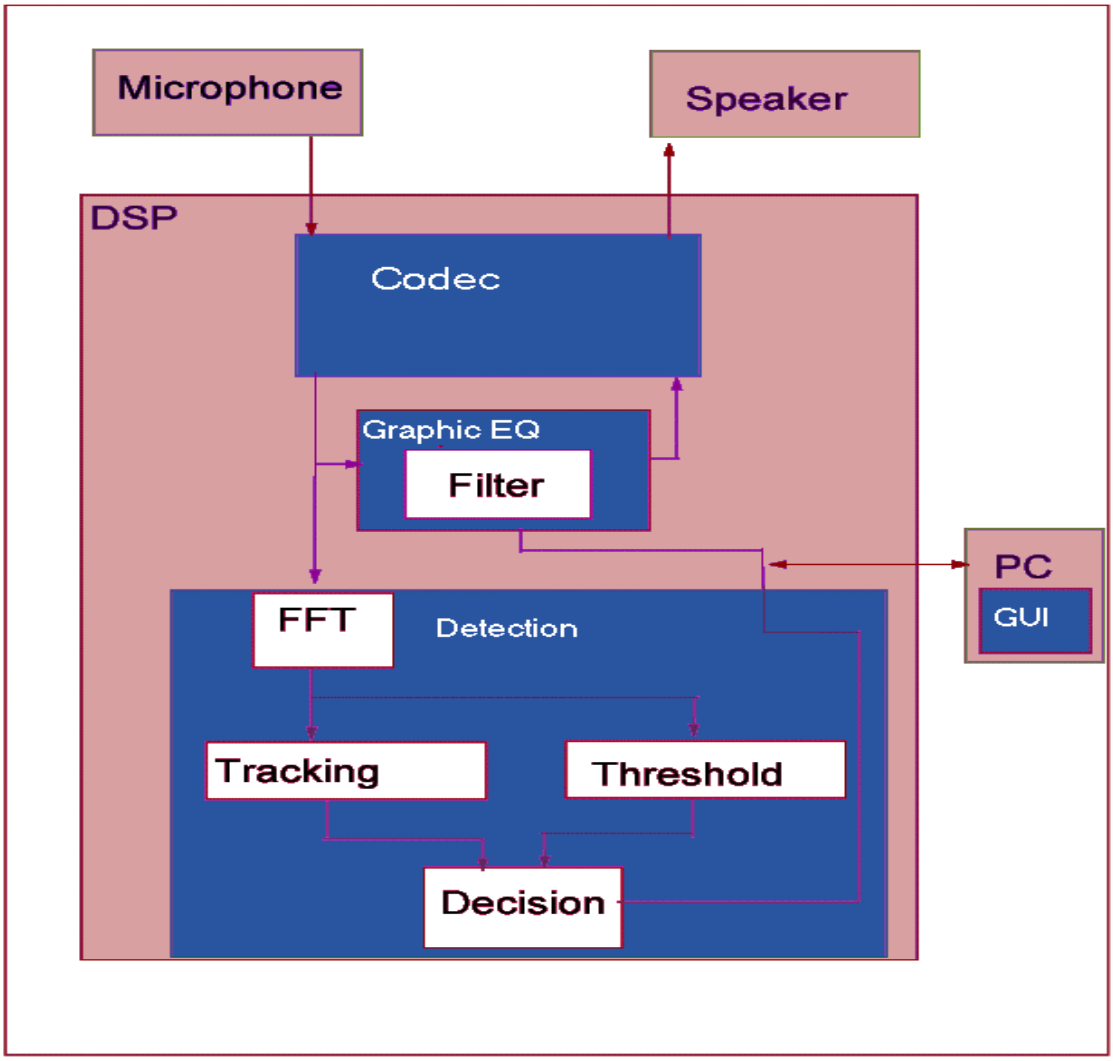

Figure 3

2.2.1 Codec

The Codec on the EVM board was operating at 44.1kHz. There is an automatic –6dB on the signal between input and output. This is ignored for the current project. It is trivial to counteract this by doubling the amplitude of the output on the transmit interrupt.

For preliminary tests, a function generator was used on the line in, and an oscilloscope on the line out. In real-time testing the function generator was replaced by a Shure SM-58 vocal microphone and a pair of Altec-Lansing computer speakers to generate real-time feedback replaced the oscilloscope. The microphone was chained through the Sound Blaster sound card in the PC for gain control and mixing with a CD.

2.2.2 FFT

The original design utilized the TI benchmark Radix-4 FFT in assembly. Due to constraints with the interrupts, this could not be used. The radix-4 FFT is not interruptible, so it must either be done inside one interrupt, or interrupts must be disabled. Since interrupts are critical to sound quality, disabling interrupts for any period of time is not an option. The Radix-4 FFT does not finish quickly enough on a 1024 long sample to be done within an interrupt.

Because of this, the c code supplied with the benchmark assembly to calculate the FFT was used. This completes in short enough time to fit into the detect algorithm and has no issues with interrupts.

2.2.3 Detection

The detection is completed within 1009 interrupts, almost 4 million clock cycles. Within this time, the detect function must perform an FFT, then traverse all 512 data points twice, once to calculate current value, then once to determine if it is now feedback. These could be combined into one step as an optimization. After traversing, if feedback is detected, a filter must be set in place. More details of this algorithm are provided above.

2.2.4 Equalization

The input signal passes through a 2^{nd} order Butterworth high pass filter at 160 Hz. This is necessary due to the properties of the speaker and microphone used. The quality of the output sound is greatly improved.

The output signal passes through the graphic equalizer, which consists of a maximum of 6 out of a possible 31 2^{nd} order Butterworth band pass filters centered 1/3 octave apart with a filter width of one half octave to either side of the center frequency.

2.3 Interface

A way was needed to get output from the EVM while it was working. This would help to identify run-time errors and ensure that the algorithm was functional. The standard console application was forgone in favor of a good interface. A Real-Time Analyzer

(RTA) was implemented which would show the Fourier Transform of the input signal. Such RTA outputs are a staple of sound-related software (i.e. mp3 players).

The interface was made using standard MFC components and OpenGL. The MFC App wizard was used to generate a dialog-based application. To this was added a CRichEdit control to handle text output and an OpenGL rendering context to display the RTA. Finally, a "pause" button was added, which would halt the redisplay of the output. Note that the pause button does not freeze the collection of data from the EVM.

For data collection, the dialog spawns a second thread called wait_thread(). This function waits for the EVM to call PC_PRINT. The first call from PC_PRINT shows what data is to follow in subsequent PC_PRINT calls. A "1" meant that a detected frequency was to follow. A "2" meant that an eliminated frequency was to follow. A "3" meant that amplitude information would come. And finally, a "4" would be followed by RTA data to display. This RTA data was sent as 39 consecutive floats and the OpenGL rendering context immediately displayed it.

A timer was instated for animation purposes. This refreshed the dialog contents every 30 milliseconds. So, while the data collection was occurring at odd intervals, the refresh occurred regularly. This was done in anticipation of the Real Time Analyzer.

The RTA was forgone, because it interfered with the best possible performance from the EVM. This was due to sending of the RTA data, which caused the EVM to lose

interrupts and hence skip sound data. Many trials were attempted to alleviate this issue. Less data was sent at longer intervals, and in compressed formats, but this experiment was not fully successful. This should be explored further.

3 DSP methods

3.1 General/Memory

Code from some of the labs was used, especially from labs 1 and 2 to get the Codec operating and the DSP performing the necessary operations to get data into the detect function.

All of the program and data was placed in on-chip memory, with the exception of parts of the libraries, which was left off chip at Pete Boettcher's suggestion. This allows for speedy memory accesses and execution of the program within the memory constraints.

```
  .rtstext      {-lrts6701.lib(.text)         /* putting libraries far away */
                 -ldev6x.lib(.text)
                 -ldrv6x.lib(.text)} > SDRAM0
  .rtsbss       {-lrts6701.lib(.bss)
                 -ldev6x.lib(.bss)
                 -ldrv6x.lib(.bss)
                 -lrts6701.lib(.far)
                 -ldev6x.lib(.far)
                 -ldrv6x.lib(.far)} > SDRAM0
  .rtsdata      {-lrts6701.lib(.const)
                 -ldev6x.lib(.const)
                 -ldrv6x.lib(.const)
                 -lrts6701.lib(.switch)
                 -ldev6x.lib(.switch)
                 -ldrv6x.lib(.switch)} > SDRAM0
```

The majority of the data memory used is occupied by historical FFT data, with 8*512*sizeof(float) in memory usage. The tracking table is another 1*512*sizeof(float) In addition the buffer is 1*1024*sizeof(float) and many local copies of that buffer.

3.2 Real-Time issues

The majority of the time was spent on real-time issues. As it is extremely important to not drop any interrupts, every new step that was added required different optimizations. Profiling data is not available because this information is irrelevant if interrupts are not taking place, and impossible to get with interrupts operating.

The FFT could not be conducted from inside an interrupt, and it failed if it was outside of the interrupt, so C was used code for that. The filtering code was also not interruptible, so it was chosen to put that inside the interrupts. The high pass filter went on the receive, and the graphic equalizer was put on the transmit, mostly for time reasons, but also because removing the low frequencies out before detect() even tried to eliminate it is desirable.

The buffers, that were written to on the interrupts needed to be marked as volatile in order to keep the compiler from optimizing out accesses to this memory. Before this step was taken, there were bad values in the FFT.

Making local copies of buffers that were accessed multiple times also sped up memory accesses substantially. For example, the following code could have been replaced with

input[2*I]=input[2*I]/6000; however this was much slower and actually caused interrupts to be dropped when done in a loop of 1024.

```
                        j=2*i;
                        z=input[j];
                        input[j]=z/6000;
```

The following code was necessary to make sure registers were not overwritten and that the correct data was received across the PCI bus.

```
        haha=(float)(input[I]);
               tosend=(unsigned int *)(&haha);
               pci_message_sync_send(*tosend,TRUE);
```

The sync_send was necessary to make sure data did not get dropped on the way over, async sends were not as reliable. In addition, if just (unsigned int *)(&haha) was sent, this operation fails the second time it is used. In the future, getting async send to work to allow for more data transfers should be investigated.

4 Areas for Improvement

4.1 Improve slow detection to not be over-aggressive

A sine wave input from a function generator is detected as feedback. This can be easily eliminated by changing the slow-frequency ratio to 1.01 instead of 1, however this paves the way for equilibrium feedback to manifest. More testing is needed in this area.

4.2 Reduce Threshold

The noise floor is sufficiently high that feedback cannot be detected until sound is already annoyingly loud. If feedback could be detected before it gets above the noise floor, this eliminator would be useful during the concert, not only for the ring-out process.

4.3 Parametric EQ

The fixed EQ has several issues. The first problem is an extremely sharp drop-off at the center frequency, approaching 60dB. The second problem is that while border frequencies are attenuated sufficiently, they are not attenuated as much as center frequencies, so dynamically setting the center frequency would provide better quality and finer filter resolution for less reduction in sound quality.

The first problem could be addressed by mixing the input signal with the filtered signal to provide a more gradual drop off. The second problem requires generating filter coefficients real-time, which may be a problem due to tight time constraints on the interrupts.

4.4 Transfers

Each pair of PCI data transfers causes an interrupt to be dropped. If this problem were eliminated, the real-time analyzer would work and more data could be transferred.

Bi-directional transfers without getting in the way of interrupts would be an additional desired feature. This way from the interface it would be possible to set modes of the detector or even to select frequencies to eliminate.

4.5 New features

One of the best features would be a gradual decay of all filters put in place. Because room properties and the relative location of microphone and speakers change, it would be nice for filters to expire without requiring explicit removal or resetting the system.

In addition, displaying a frequency response curve to the output window would allow for better analysis by the engineer operating the equipment.

There are many more desired features, such as audio compression and more operator control, but none as important as these two.

5 Results

As of the final demo, feedback elimination was working in that it was possible to get the detector to eliminate any feedback. Occasionally this feedback was not picked up until it was already painful, but during "ring-out" this is not a major threat to functionality.

The detector rarely picked up non-feedback. On a CD playing, at most it picked up one frequency per song, and those could actually be real feedback. With a function generator input, it picks the sine wave up as sustained feedback, but this is easily corrected by modifying the factor.

There is no distortion during normal operation, although one interrupt is lost each time data is transferred. This is not a major problem, as this is simultaneous with the filtering

which is a much larger distortion problem. After more than three frequencies are removed, there is audible distortion in music, but this is true of any equalization.

6 References

6.1 Feedback Elimination links

*Someone else's senior project:

http://www.isip.msstate.edu/publications/courses/ece_4012/1998/active_feedback/presentation/

*2 patents on the subject - these link to many others:

http://www.patents.ibm.com/details?&pn10=US04845757

http://www.patents.ibm.com/details?patent_number=5245665

*Paper on Feedback elimination - Sabine's Propaganda

http://www.SabineUSA.com/newsite/pdf/Positive-Feedback-v2.pdf

6.2 Audio Engineering links

*General Audio Engineering information - lots of really good links

http://www.hut.fi/Misc/Electronics/audiopro.html

*Audio Engineering Society

http://www.aes.org

*Sabine - make feedback exterminators

http://www.SabineUSA.com/

*Behringer - also make feedback exterminators

http://www.behringer.de/

*Shure - make microphone we will be using

http://www.shure.com/sm58.html

*EAW - make speaker we will be using in demo

http://www.eaw.com/

*BSS - make very good parametric and graphic EQ's - use DSP's in one EQ

http://www.bss.co.uk/e_fds.htm#fds366

6.3 Misc

*ABTech - people we are renting equipment for demo from

http://www.abtech.org/

*Wavelet tutorial

http://www.public.iastate.edu/~rpolikar/WAVELETS/WTtutorial.html

*Wavelet.org

http://www.wavelet.org/

*Matlab

http://www.mathworks.com/web_downloads/

*TI code

http://www.ti.com/sc/docs/products/dsp/c6000/67bench.html#filters