Part I

Echo Cancellation

Mahmoud Alizadeh, Priyanka Marigi, Jakub Górski, George Ryrstedt

Abstract

The project explores and implements a signal processing algorithm, Normalised Least Mean Square, for acoustic echo cancellation. The algorithm was first prototyped in Matlab and then implemented in C using *Code Composer Studio IDE* on *Texas Instruments DSK6713* DSP board. The project aims to cancel acoustic echo created by a system involving a near-end speaker and a loudspeaker delivering far-end signal. The results shows that the NLMS algorithm can reduce the echo.

Contents

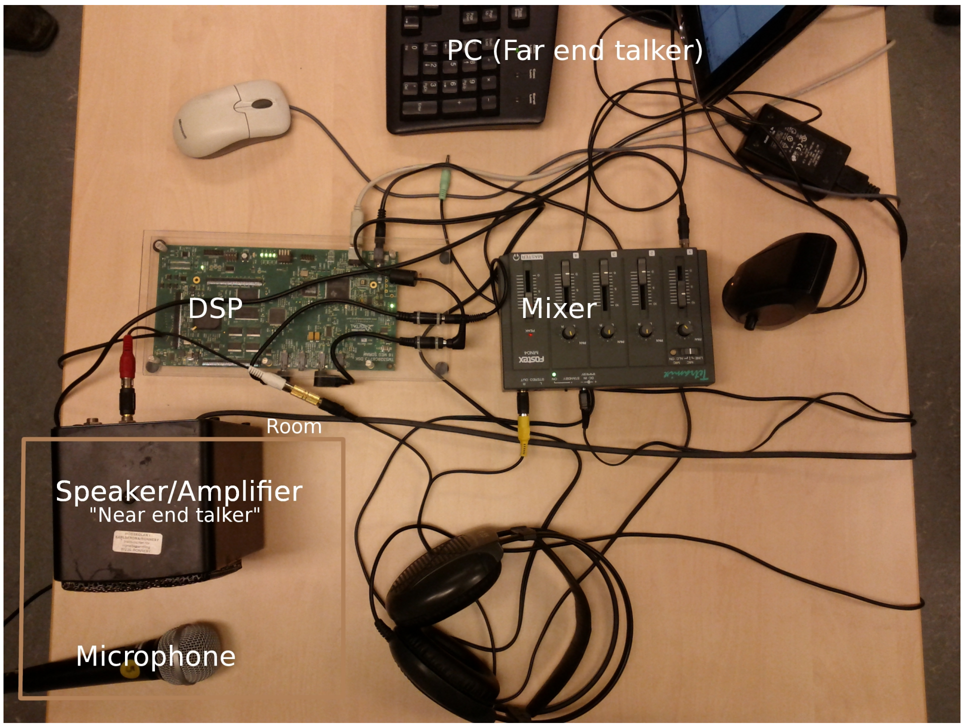

Figure 1: An overview of the general setup environment for the Echo Cancellation project.

1 Introduction

The most common application of acoustic echo cancellation is in hands-free systems like telephony and teleconferencing systems. The echo produced by the room decreases the quality of the voice received by the far end. Manufacturers of such systems are hard at work to improve voice quality by exploring different acoustic echo cancellation algorithms. As part of this project we decided to implement one of the most well known algorithms, Normalised Least Mean Square (NLMS) on DSP board.

The following sections discuss the theory behind echo cancellation, algorithm implementation and results.

2 Theory

The sound from a speaker, for instance in a teleconferencing system, gets reflected by the obstacles in a room like walls or other material surfaces. These reflected waves in turn get reflected and so on. This phenomenon creates an undesirable signal called echo which gets picked up by the microphone along with the desired signal of the near-end speaker. As such the sound quality received by the far-end listener gets reduced. The following subsection discusses a system implementing an adaptive filter algorithm to tackle the problem of removing the echo.

2.1 Algorithm

A high level block diagram of a system employing echo canceller is shown in figure 2. When the signal from the speaker passes through an air medium it does not follow a straight path, but rather gets reflected from different obstacles resulting in delayed versions of the signal. This delayed signal is what is called the echo as it will be picked by the microphone along with the near-end speaker which is the desired signal. Thus, the output of the system is the summation of the near-end signal and the echo. In order to extract only the near-end signal an adaptive filter that attempts to mimics the channel is used. This filter adapts the behaviour of the system gradually by learning from the actual output. The filter mimics the echo of the system and outputs it, which is subtracted from the system output resulting in a signal close to the desired signal.

Adaptive filters is useful when the behaviour of the channel varies over time. Figure (M1) shows an adaptive echo cancellation system. In fact, the adaptive filter should remove the echoes comes from the speaker into the microphone. As shown in the figure, in the presence of the far-end signal, $x(t)$, the echo signal comes into the microphone which is filtered by the channel, h_t, which is modelled as n-tap FIR filter:

$$h = [h(0)\ h(1) \ldots\ h(n-1)]^T \tag{1}$$

Then is the absence of near-end signal, $v(t)$, the output of the microphone is:

$$y(t) = h^T X(t) + w(t) \tag{2}$$

Where $w(t)$ is the additive noise and:

$$X(t) = [x(t)\ x(t-1) \ldots\ x(t-n+1)]^T \tag{3}$$

If the adaptive filter tries to estimate the channel, $hath$, as close as h, so the error will be the subtraction of microphone signal and the output of adaptive filter signal[1]:

$$e(t) = y(t) - \hat{h}^T X(t) \tag{4}$$

Some popular algorithms for the echo cancellation adaptive filters are LMS (NLMS) and RLS. The RLS method, is the most effective and robust but its more computational heavy which makes it unsuitable for use in an

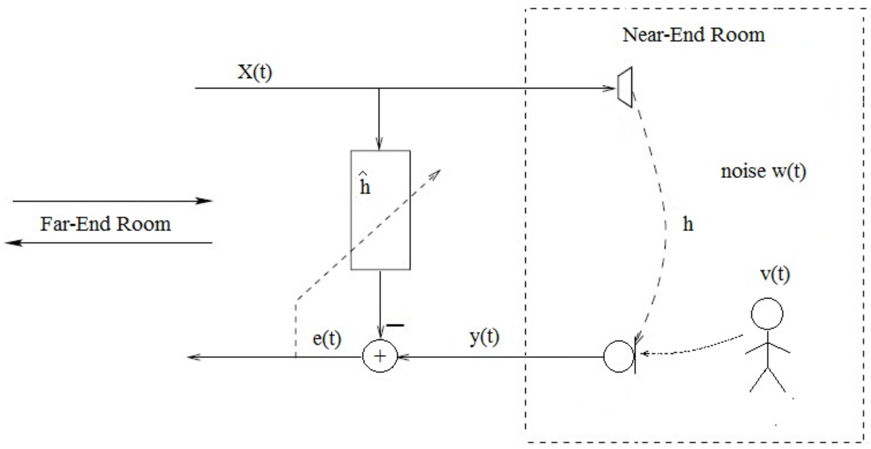

Figure 2: Adaptive filter[1].

embedded system[2]. Frequency-domain LMS algorithm is another sophisticated method for echo cancellation, because the time-domain impulse response of the speech (a non-stationary signal) requires very large filter taps. Also, the double talk separation in frequency domain[3] is more easy than time-domain. A recently developed popular algorithm is the affine project algorithm (APA). The main advantage of this algorithm in comparison to NLMS is faster convergence rate, especially for speech signals[4].

2.1.1 LMS

The least-mean-square (LMS) algorithm is one of the most popular adaptation algorithm for echo cancellation which have been used since in the mid 1960s[5]. In adaptive filters the filter coefficients are updated as follow:

$$\hat{h}(n+1) = \hat{h}(n) + \Delta\hat{h} \tag{5}$$

The LMS algorithm updates the filter coefficients according to the negated gradient of mean square error as follows:

$$\Delta\hat{h} = -\frac{\mu}{2}\nabla E[e^2(n)] \tag{6}$$

where μ is the step size that indicate the rate of convergence, ∇ is gradient and E is the expectation of square error, but for LMS algorithm, the instantaneous value is replaced and after some manipulation the updated filter coefficients are:

$$\hat{h}(n+1) = \hat{h}(n) + \mu e(n)x(n) \tag{7}$$

2.1.2 NLMS

There are some drawbacks in the LMS algorithm e.g that μ depends on the input vector; it means that for different values of the input signal, the step size should be changed, another one is that the error is proportional to the input signal $x(n)$, then the updated part of the filter coefficients are proportional to the power of the input signal $x(n)$, and it suffers from the presence of noise. One way to avoid this problem in the normalized LMS is to normalize the updated part with the power of the input signal $x(n)$:

$$\hat{h}(n+1) = \hat{h}(n) + \frac{\mu}{\| x(n) \|^2 + a} \tag{8}$$

Where a is a protection value to avoid division by zero, in this approach μ can be between $0 < \mu < 2$.

3 Simulation

In the initial stage of the project it was essential to implement the algorithm in Matlab. Such an implementation would make it possible to compare the precision of the C implemented algorithm and plotting results in Matlab, which allows for more convenient error detection. By plotting the delta function it was possible to note the time instance of an echo's detection. The appearance of a new echo can be noticed through observation of the function's deviation at that point. Once the Matlab and C implementation to some extent match in term of efficiency and error parameters then the algorithm is deemed as stable and ready for implementation on the DSP board.

3.1 Matlab

In order to generate the echoes the signal has to pass through the microphone

As shown in figure 2 the signal $X(t)$ travels through the channel which consequently picks up the noise $w(t)$ of the channel before it ends up at the microphone. Noside may come from the microphone, the environment and is dependent on the acoustics of the room. The inferred error is illustrated by figure 3 where the noise can be seen in the parts where there should be silence.

Because of this noise the filter coefficients $\hat{h}$ will have a difficulty converging, so that the echo can be filtered correctly. The longer a signal has been received by the board the less it will be affected by the additive noise,

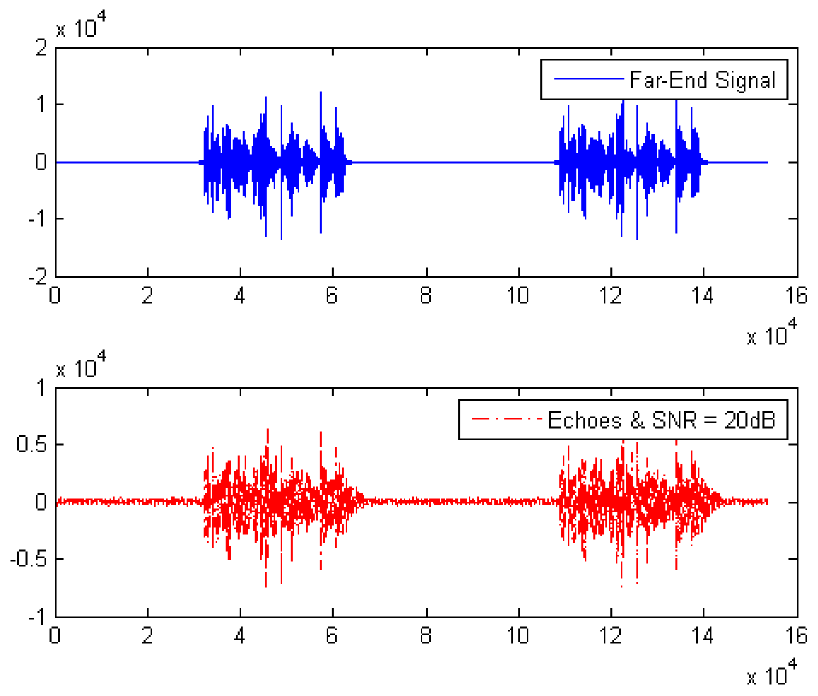

Figure 3: Graph comparing the illustrated error before the speaker, and after the microphone.

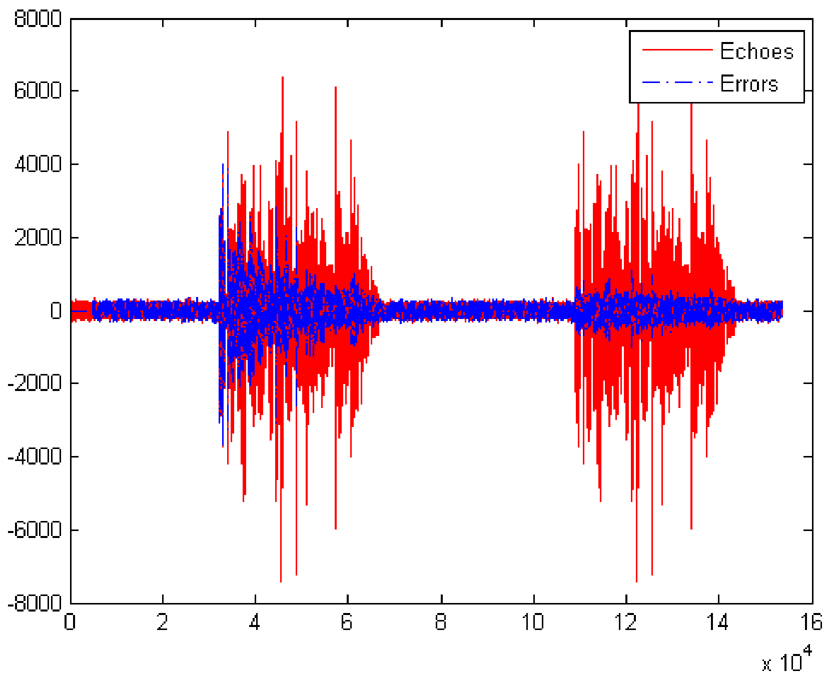

Figure 4: Shows graph of coefficients $\hat{h}$ converging towards the channel over time.

because the taps will come closer to accurately approximating the channel. The error goes down as shown in figure 3.1.

4 Implementation

4.1 Hardware constraints

The implementation of the algorithm was memory constraint-oriented. Due to the fact that the external DRAM memory proved to not be fast enough keeping all variables allocated on the stack was crucial. In order to keep such memory usage in check and be able to detect any spikes in such usage Code Composer Studio's CPU graph utility. Any CPU usage is displayed on the graph with CPU utilization on per-thread basis.

Greatest memory management improvement was observed once sampling frequency was lowered, which effectively lowered memory utilization, and allowed for a greater filter length. The latter provided a better performing echo canceller.

By using this information it was possible to push the size of the algorithm's adaptive coefficient vector to the out most limit without causing

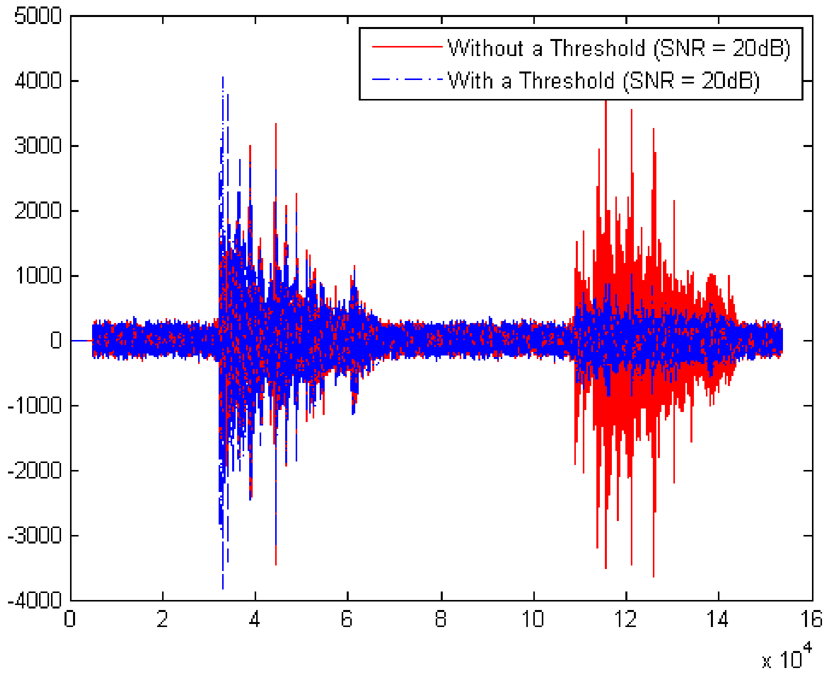

Figure 5: Graph comparing error $e(t)$ signal threshold filter enabled, as opposed to disabled.

unnecessary crashes due to memory overflows.

4.2 Optimizations in C

By implementing the algorithm in C according to the sliding window principle it was possible to bring down the memory usage significantly. The algorithm is written such that the input signals are read once at most, upon which their results are computed and adaptive coefficients updated with each iteration.

4.2.1 Signal threshold

In order to cope with noise in the signal a signal threshold was introduced, which ideally would not allow background noise to pass through. if the near-end signal is and there for only noise is received then `w[]` coefficient vector, which is named $\hat{h}$ in section 2 and equation 1, should not be updated.

According to figure 5 the error is unreasonably big if there is no threshold filter applied at the signal input block. Because the filter coefficients are updated when there is only noise present on the channel the coefficients are updated with the wrong audio data. This phenomenon is illustrated in figure 5 where the algorithm without the threshold filter presents significantly higher error values.

Therefore this graph shows that there is a need for a threshold which increases the overall accuracy of the echo canceller, which can also be seen in the figure.

Once the far-end signal is larger then the noise threshold, the filter coefficients $\hat{h}$ should be updated again. However, the threshold is not a fixed variable but calculated according to a signal variance algorithm:

$$Th = \frac{1}{N} \sum_{n=1}^{N} x(n)^2 \tag{9}$$

By using this algorithm the variance can be calculated for each audio block as well as the total number of audio blocks. Consequently the variance for total audio blocks and currently processed audio block will be compared according to:

$$Th_{total} > \frac{1}{N} \sum_{block=1}^{block_N} x(n)^2 \tag{10}$$

Should the total variance of the audio blocks be bigger than the variance for the current block, then signal is deemed as noise.

4.3 Problems

During the development of the system several problems where encountered. To start off the algorithm was implemented in a system agnostic way so as to allow regression testing against the Matlab implementation. This turned out to have been a very good choice as it was discovered that when something broke in the algorithm it was very seldom readily apparent. One example of such a devious error that occurred several times during different stages of the development of this project was that we accidentally introduced off by one errors when accessing the arrays. As we technically owned the memory accessed when running on the computer and the board lacks memory protection the program didnt give us a segmentation fault or any other indication that something was amiss. The way we noticed these errors was that when the output of the C implementation was compared to the Matlab version we had minute errors in the output, had we simply ran it on the board everything would have appeared to be working except that the board could potentially have crashed at random intervals, in reality it did not crash which means that such an error could easily have slip through to production code had we not had regression testing. Another difficulty that was encountered was that when something broke in the code it resulted in what appeared as random output meaning there was no easy way to tell

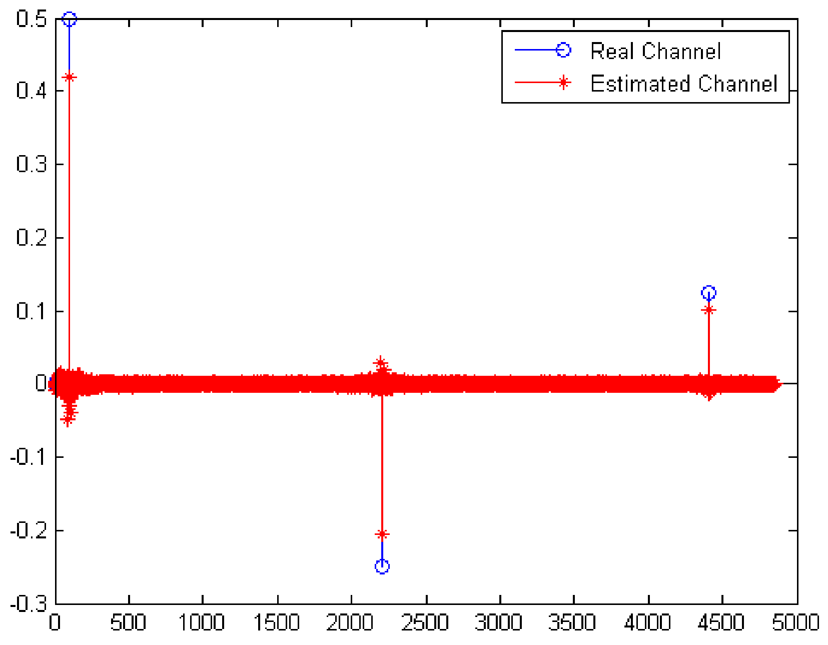

Figure 6: Channel estimation graph of C implementation (Estimated) compared to the real channel.

what was wrong. We countered this by only changing small portions of the code at a time and continuous tests of the changes we made.

5 Conclusion

The goal of the project, to implement a working echo cancellation algorithm, was achieved as illustrated by figure 6. It would have been good for the performance of the algorithm to implement double-talk[1] detection. However, the NLMS algorithm handles double-talk with sufficient accuracy for this project even without double talk detection.

Due to the hardware constraints mentioned previously in the report, the filter length could not be larger than an array of 700 floating point integers. Because of this the amount of echoes that can be potentially detected may not exceed $87.5\mu s$, which makes this another measure for the efficiency of the final filter.

Another improvement of this filter could have been made if a variable step-size algorithm was implemented, as it governs the rate of convergence towards the channel. This was not implemented due to the time constraint imposed by the project deadline.

References

[1] P. øAhgren and A. Jakobsson, *A Study of Doubletalk Detection Performance in the Presence of Acoustic Path Changes*, IEEE Transaction on Consumer Electronics, May 2006

[2] P. Sinha, *Speech Processing in Embedded System*, Springer, 2010

[3] F. Capman, J. Boudy, P.Lockwood, *Acoustic Echo Cancellation and Noise Reduction in the Frequency-Domain: A Global Optimisation*,

[4] C. Paleologu, J. Benesty, and S. Ciochina, *Variable step-size affine projection algorithm designed for acoustic echo cancellation*, IEEE Trans. Audio, Speech, Language Process., Nov. 2008

[5] M. M. Sondhi, *The History of Echo Cancellation*, IEEE Signal Processing Magazine, Sept. 2006

6 Appendix - A

```
#include <std.h>
#include <math.h>

#include <hst.h>
#include <log.h>
#include <pip.h>
#include <swi.h>
#include <sys.h>

#include <iom.h>
#include <pio.h>

#ifdef _6x_
extern far LOG_Obj trace;
extern far PIP_Obj pipRx;
extern far PIP_Obj pipTx;
extern far SWI_Obj swiEcho;
#else
extern LOG_Obj trace;
extern PIP_Obj pipRx;
extern PIP_Obj pipTx;
extern SWI_Obj swiEcho;
#endif

#define FILTER_LEN 700 // must be divisible by 2
#define AUDIO_LEN  256
#define BUFF_LEN AUDIO_LEN/2
short oldInData[FILTER_LEN] = { 0 };

float short_w[FILTER_LEN];
short oldInPos = 0;
float norm_u = 0;
double var_tot = 0;
float tot_num = 0;

void mexNewNLMS(size_t N_LEN, short *in, short *err, double mu, double a);

/*
 *  'pioRx' and 'pioTx' objects will be initialized by PIO_new().
 */
PIO_Obj pioRx, pioTx;
```

```
/*
 *  ======== main ========
 *
 *  Application startup funtion called by DSP/BIOS. Initialize the
 *  PIO adapter then return back into DSP/BIOS.
 */
main()
{

  int indx;
  for (indx = 0; indx < FILTER_LEN; indx++) {

    short_w[indx] = 0;
    oldInData[indx] = 0;
  }

  /*
   * Initialize PIO module
   */
  PIO_init();



  /* Bind the PIPs to the channels using the PIO class drivers */
  PIO_new(&pioRx, &pipRx, "/udevCodec", IOM_INPUT, NULL);
  PIO_new(&pioTx, &pipTx, "/udevCodec", IOM_OUTPUT, NULL);

  /*
   * Prime the transmit side with buffers of silence.
   * The transmitter should be started before the receiver.
   * This results in input-to-output latency being one full
   * buffer period if the pipes is configured for 2 frames.
   */
  PIO_txStart(&pioTx, PIP_getWriterNumFrames(&pipTx), 0);

  /* Prime the receive side with empty buffers to be filled. */
  PIO_rxStart(&pioRx, PIP_getWriterNumFrames(&pipRx));

  LOG_printf(&trace, "pip_audio started");
}


/*
 *  ======== echo ========
 *
 *  This function is called by the swiEcho DSP/BIOS SWI thread created
 *  statically with the DSP/BIOS configuration tool. The PIO adapter
 *  posts the swi when an the input PIP has a buffer of data and the
 *  output PIP has an empty buffer to put new data into. This function
 *  copies from the input PIP to the output PIP. You could easily
 *  replace the copy function with a signal processing algorithm.
 */
Void echo(Void)
{
  Int audiosize;
  short *audiosrc, *audiodst;

  /*
   * Check that the precondions are met, that is pipRx has a buffer of
   * data and pipTx has a free buffer.
   */
  if (PIP_getReaderNumFrames(&pipRx) <= 0) {
    LOG_error("echo: No reader frame!", 0);
    return;
  }
  if (PIP_getWriterNumFrames(&pipTx) <= 0) {
    LOG_error("echo: No writer frame!", 0);
    return;
  }

  /* get the full buffer from the receive PIP */
  PIP_get(&pipRx);
  audiosrc = PIP_getReaderAddr(&pipRx);
  audiosize = PIP_getReaderSize(&pipRx) * sizeof(short);

  /* get the empty buffer from the transmit PIP */
  PIP_alloc(&pipTx);
  audiodst = PIP_getWriterAddr(&pipTx);


  /* Do the data processing. */
  //    process(audiosrc,audiosize,audiodst);

  //mexNewNLMS( audiosrc,  audiodst, 0.5, 0.01, audiosrc);


  mexNewNLMS(audiosize, audiosrc, audiodst, 0.5, 0.01);
  /* Record the amount of actual data being sent */
  PIP_setWriterSize(&pipTx, PIP_getReaderSize(&pipRx));

  /* Free the receive buffer, put the transmit buffer */
  PIP_put(&pipTx);
  PIP_free(&pipRx);
}

/*
 *  ======== copy ========
 *
 *  This function is called by the swiCopy DSP/BIOS SWI thread created
 *  statically with the DSP/BIOS configuration tool. The PIO adapter
 *  posts the swi when an the input HST has a buffer of data and the
 *  output HST has an empty buffer to put new data into. This function
 *  copies from the input HST to the output HST. You could easily
 *  replace the copy function with a signal processing algorithm.
 */

Void copy(HST_Obj * input, HST_Obj * output)
{
  PIP_Obj *in, *out;
  Uns *datasrc, *datadst;
  Uns datasize;
  short i;

  in = HST_getpipe(input);
  out = HST_getpipe(output);

  if (PIP_getReaderNumFrames(in) == 0 || PIP_getWriterNumFrames(out) == 0) {
    LOG_error("echo: No data to read from file!", 0);
  }


  LOG_printf(&trace, "in copy");
  /* get input data and allocate output buffer */
  PIP_get(in);
  PIP_alloc(out);

  /* copy input data to output buffer */
  datasrc = PIP_getReaderAddr(in);
  datadst = PIP_getWriterAddr(out);

  datasize = PIP_getReaderSize(in);
  PIP_setWriterSize(out, datasize);

  for (i = 0; i < datasize; i++) {
    *datadst++ = *datasrc++;
  }

  /* output copied data and free input buffer */
  PIP_put(out);
  PIP_free(in);
}

short inline readEcho(short *data, int index)
{
  return data[index * 2 + 1];
}

short inline readClean(short *data, size_t index)
{
  return data[index * 2];
}

void writeStereo(short value, short *dest, int index)
{
  dest[index * 2] = value;
  dest[index * 2 + 1] = value;
}


short inline readOffset(size_t offset, short *oldInData, size_t index)
{
  offset = (index + offset) % FILTER_LEN;
  return oldInData[offset];
}

void mexNewNLMS(size_t N_LEN, short *in, short *err, double mu, double a)
{
  size_t t;
  double w_mul_in;
  double a_norm_u, temp;
  size_t buf_len = N_LEN / 2;
  short m;
  short value;

  double curDataSq;
double var_buf = 0;
short buf_num = 0;
double varb, vart;

// Matlab
//         uvec=u(n-Delay:-1:n-Delay-M+1);
// algorithm, using, a, mu, flen
///*
for (t = 0; t < buf_len; ++t) {
  //remove old value and add next from sliding window, oldindata will be init

  curDataSq = readClean(in, t) * readClean(in, t);
  var_buf += curDataSq;
  buf_num++;

  var_tot += curDataSq;
  tot_num++;
  varb = var_buf / buf_num;
  vart = var_tot / tot_num;


  norm_u += curDataSq;
  norm_u -= oldInData[t] * oldInData[t];

  // MATLAB:          e(n)=d(n)-w'*uvec;
  w_mul_in = 0; //reset for this iteration
  //calculate the echo for this sample

  for (m = 0; m <= t; ++m) {
    w_mul_in += short_w[m] * readClean(in, (t - m));
  }

  for (m = t + 1; m < FILTER_LEN; ++m) {
    w_mul_in += short_w[m] * oldInData[FILTER_LEN + t - m];
  }

  temp = readEcho(in, t) - w_mul_in;


  err[t * 2] = temp;
  err[1 + t * 2] = (short) temp;


  if (varb > vart) {
      a_norm_u = (mu / (a + norm_u)) * temp;
      for (m = 0; m <= t; ++m) {
short_w[m] += a_norm_u * readClean(in, (t - m));
      }

      for (m = t + 1; m < FILTER_LEN; ++m) {
short_w[m] += a_norm_u * oldInData[FILTER_LEN + t - m];
      }

    }

  }

  for (m = 0; m < FILTER_LEN; ++m) {
    if (m < (FILTER_LEN - buf_len)) {
      value = oldInData[buf_len + m]; //move old values forward
    } else {
      value = readClean(in, (m - (FILTER_LEN - buf_len))); //append new data
      oldInData[m] = value;
    }
  }
}
```