**Department of Electrical Engineering**

|  |  |
| --- | --- |
| **Faculty Member: Mr. Kalimullah** | **Dated: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** |
|  |  |
| **Course/Section: BEE-8D** | **Semester: 6th** |
|  |  |

**EE-330 Digital Signal Processing**

**Lab 10 # Frequency Delays and FIR Filtering**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **PLO4-CLO4** | | **PLO5-CLO5** | **PLO8-CLO6** | **PLO9-CLO7** |
| **Name** | **Reg. No** | **Viva / Quiz / Lab Performance** | **Analysis of data in Lab Report** | **Modern Tool Usage** | **Ethics and Safety** | **Individual and Team Work** |
|  |  | **5 Marks** | **5 Marks** | **5 Marks** | **5 Marks** | **5 Marks** |
| **Muhammad Haris Rehman** | **182797** |  |  |  |  |  |
| **Muhammad Hasnain Afridi** | **192414** |  |  |  |  |  |
| **Muhammad Anus Siddiqui** | **174308** |  |  |  |  |  |
|  |  |  |  |  |  |  |

**Lab10: Delays and FIR Filtering**

In this lab you will study sample by sample processing methods for FIR filtering and implement them on the TMS320C6713 processor.

**Objectives**

* Implementation of FIR filter on DSK
* Implementation of multiple Delay.
* Delay implementation using linear circular buffer
* Digital audio effect using delays

**Lab Instructions**

* The students should perform and demonstrate each lab task separately for step-wise evaluation (please ensure that course instructor/lab engineer has signed each step after ascertaining its functional verification)
* Each group shall submit one lab report on LMS within 6 days after lab is conducted. Lab report submitted via email will not be graded.

. Students are however encouraged to practice on their own in spare time for enhancing their

**Lab Report Instructions**

All questions should be answered precisely to get maximum credit. Lab report must ensure following items:

* Objective
* Introduction to DSK C6713 board
* Introduction to C6713 processor Chip
* Introduction to Code Composer studio
* Conclusion

# Delays and FIR Filtering

**Abstract:**

In the last lab you studied filtering by convolution, which is a block processing method. In this lab you will study sample by sample processing methods for FIR filters and implement them on the TMS320C6713 processor. Once you know how to implement a multiple delay on a sample by sample basis, it becomes straightforward to implement FIR and IIR filters. Multiple delays are also the key component in various digital audio effects. Delays can be implemented using linear or circular buffers, the latter being more efficient, especially for audio effects.

## Delays Using Linear and Circular Buffers

A D-fold delay, also referred to as a delay line, has transfer function H(z)= z−D and corresponds to a

time delay in seconds where T is the time interval between samples, related to the sampling rate by fs = 1/T. A block diagram



realization of the multiple delay is shown below:



Fig. 4.1 Tapped delay line

There are D registers whose contents are the “internal” states of the delay line. The dth state sd, i.e., the content of the dth register, represents the d-fold delayed version of the input, that is, at time n we have: sd(n)= x(n − d), for d = 1, . . . , D; the case d = 0 corresponds to the input so(n)= x(n). At each time instant, all D contents are available for processing and can be “tapped” out for further use (e.g., to implement FIR filters). For example, in the above diagram, the dth tap is being tapped, and the corresponding transfer function from the input x to the output y = sd is the partial delay z−d The D contents/states sd, d = 1, 2, . . . , D, and the input so = x must be stored in memory in a (D+1) dimensional array or buffer. But the manner in which they are stored and retrieved depends on whether a linear or a circular buffer is used. The two cases are depicted below.



In both cases, the buffer can be created in C by the declaration:

**float w[D+1];**

Its contents are retrieved as w[i], i = 0, 1, . . . , D. Thinking of w as pointer, the contents can also be retrieved by \*(w + i)= w[i], where \* denotes the de-referencing operator in C. In the linear buffer case, the states are stored in the buffer sequentially, or linearly, that is, the ith state is:



At each time instant, after the contents Si are used, the delay-line is updated in preparation for the next time instant by shifting its contents to the right from one register to the next, as suggested by the block diagram in Fig. 4.1. This follows from the definition Si(n)= x(n − i), which implies for the next time instant Si(n+1)= x(n+1−i)= Si−1(n). Thus, the current Si−1 becomes the next Si. Since Si = w[i], this leads to the following updating algorithm for the buffer contents:

for i=D down to i=1,

do: w[i]=w[i−1]

where the shifting is done from the right to the left to prevent the over-writing of the correct contents.

It is implemented by the C function delay() of the text [1]:

// delay.c -linear buffer updating// -------------------------------

void delay(int D, float \*w)

{

int i;

for (i=D; i>=1; i--)

w[i] = w[i-1];

}

// -------------------------------

For large values of D, this becomes an inefficient operation because it involves the shifting of large amounts of data from one memory location to the next. An alternative approach is to keep the data unshifted but to shift the beginning address of the buffer to the left by one slot. This leads to the concept of a circular buffer in which a movable pointer p is introduced that always points somewhere within the buffer array, and its current position allows one to retrieve the states by Si = \*(p + i), i = 0, 1, . . . , D. If the pointer p + i exceeds the bounds of the array to the right, it gets wrapped around to the beginning of the buffer. To update the delay line to the next time instant, the pointer is left-shifted, i.e., by the substitution p = p−1, or, −−p, and is wrapped to the right if it exceeds the array bounds to the left. Fig. 4.3 depicts the contents and pointer positions at two successive time instants for the linear and circular buffer cases f or D = 3. In both cases, the states are retrieved by Si = \*(p+i), i = 0, 1, 2, 3, but in the linear case, the pointer remains frozen at the beginning of the buffer, i.e., p = w, and the buffer contents shift forwards, whereas in the circular case, p shifts backwards, but the contents remain in place.



In the text [1], the functions tap() and cdelay() are used for extracting the states Si and for the circular back-shifting of the pointer. Although these two functions could be used in the CCS environment we prefer instead to use a single function called pwrap() that calculates the new pointer after performing the required wrapping. The function is declared in the common header file dsplab.h and defined in the file dsplab.c. Its listing is as follows:

// pwrap.c - pointer wrapping relative to circular buffer

// Usage: p\_new = pwrap(D,w,p)

// ------------------------------------------------------

float \*pwrap(int D, float \*w, float \*p)

{

if (p > w+D)

p -= D+1;

if (p < w)

p += D+1;

return p;

}

// ------------------------------------------------------

The ith state Si and the updating of the delay-line can be obtained by the function calls:



We will use this function in the implementation of FIR filters and in various audio effects. It will allow us to easily translate a sample processing algorithm expressed in pseudo-code to the actual C code. As an example, let us consider the circular buffer implementation of the partial delay z−d. The block diagram of Fig. 4.1 and the pseudo-code computational algorithm are as follows:



We may translate this into C by the following operations using pwrap:

*y = \*pwrap(D,w,p+d); // delay output*

*\*p = x; // delay-line input*

*p = pwrap(D,w,--p); // backshift circular buffer pointer*

In the last line, we must pre-decrement the pointer, that is, --p, instead of post-decrementing it, p--.

By comparison, the linear buffer implementation, using a (D+1)-dimensional buffer, is as follows:

*y = w[d]; // delay output*

*w[0] = x; // delay-line input*

*for (i=D; i>=0; i--) // update linear buffer*

*w[i] = w[i-1];*

An alternative approach to circular buffers is working with circular indices instead of circular pointers.The pointer p always points to some element of the buffer array w, that is, there is a unique integer q such that p = w + q, with corresponding content \*p = w[q]. This is depicted in Fig. 4.2. The index q is always bound by the limits 0 ≤ q ≤ D and wrapped modulo–(D+1) if it exceeds these limits. The textbook functions tap2() and cdelay2(), and their corresponding MATLAB versions given in the Appendix of [1], implement this approach. Again, however, we prefer to use the following function,qwrap(), also included in the common file dsplab.c, that calculates the required wrapped value of thecircular index:

*// qwrap.c - circular index wrapping*

*// Usage: q\_new = qwrap(D,q);*

*// -------------------------------------*

*int qwrap(int D, int q)*

*{*

*if (q > D)*

*q -= D + 1;*

*if (q < 0)*

*q += D + 1;*

*return q;*

*}*

*// -------------------------------------*

*In terms of this function, the above d-fold delay example is implemented as follows:*

*y = w[qwrap(D,q+d)]; // delayed output*

*w[q] = x; // delay-line input*

*q = qwrap(D,--q); // backshift pointer index*

*We note that in general, the ith state is:*

*Si = \*(p + i)=\*(w + q + i)= w[q + i]*

*where q + i must be wrapped as necessary. Thus, the precise way to extract the ith state is:*

*Si = w[qwrap(D, q + i)] , i= 1, 2, . . . , D*

### Lab Procedure 1

A complete C program that implements the above d-fold delay example on the TMS320C6713 processor is given below:

*// delay1.c - multiple delay example using circular buffer pointers (pwrap version)*

*// ----------------------------------------------------------------------------------*

*#include "dsplab.h" // init parameters and function prototypes*

*short xL, xR, yL, yR; // input and output samples from/to codec*

*#define D 8000 // max delay in samples (TD = D/fs = 8000/8000 = 1 sec)*

*short fs = 8; // sampling rate in kHz*

*float w[D+1], \*p, x, y; // circular delay-line buffer, circular pointer, input, output*

*int d = 4000; // must be d <= D*

*// ----------------------------------------------------------------------------------*

*void main() // main program executed first*

*{*

*int n;*

*for (n=0; n<=D; n++) // initialize circular buffer to zero*

*w[n] = 0;*

*p = w; // initialize pointer*

*initialize(); // initialize DSK board and codec, define interrupts*

*sampling\_rate(fs); // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz*

*audio\_source(MIC); // use LINE or MIC for line or microphone input*

*while(1); // keep waiting for interrupt, then jump to isr()*

*}*

*// ----------------------------------------------------------------------------------*

*interrupt void isr() // sample processing algorithm - interrupt service routine*

*{*

*read\_inputs(&xL, &xR); // read left and right input samples from codec*

*x = (float) xL; // work with left input only*

*y = \*pwrap(D,w,p+d); // delayed output - pwrap defined in dsplab.c*

*\*p = x; // delay-line input*

*p = pwrap(D,w,--p); // backshift pointer*

*yL = yR = (short) y;*

*write\_outputs(yL,yR); // write left and right output samples to codec*

*return;*

*}*

*// ----------------------------------------------------------------------------------*

Note the following features. The sampling rate is set to 8 kHz, therefore, the maximum delay D = 8000 corresponds to a delay of 1 sec, and the partial delay d = 4000, to 1/2 sec. The circular buffer array w has dimension D+1 = 8001 and its scope is global within this file. It is initialized to zero within main() and the pointer p is initialized to point to the beginning of w, that is, p = w.The left/right input samples, which are of the **short int** type, are cast to **float**, while the **float** output is cast to **short int** before it is sent out to the codec.

**Code:**

// template.c - to be used as starting point for interrupt-based programs

//

// 332:348 DSP Lab - Spring 2011 - S. J. Orfanidis

//

// ----------------------------------------------------------------------------------

**#include** "dsplab.h" // init parameters and function prototypes

**short** xL, xR, yL, yR; // input and output samples from/to codec

**#define** D 8000 // max delay in samples (TD = D/fs = 8000/8000 = 1 sec)

**short** fs = 8; // sampling rate in kHz

**float** w[D+1], \*p, x, y; // circular delay-line buffer, circular pointer, input, output

**int** d = 8000; // must be d <= D

// ----------------------------------------------------------------------------------

**void** **main**() // main program executed first

{

**int** n;

**for** (n=0; n<=D; n++) // initialize circular buffer to zero

w[n] = 0;

p = w; // initialize pointer

**initialize**(); // initialize DSK board and codec, define interrupts

**sampling\_rate**(fs); // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz

**audio\_source**(MIC); // use LINE or MIC for line or microphone input

**while**(1); // keep waiting for interrupt, then jump to isr()

}

// ----------------------------------------------------------------------------------

**interrupt** **void** **isr**() // sample processing algorithm - interrupt service routine

{

**read\_inputs**(&xL, &xR); // read left and right input samples from codec

x = (**float**) xL; // work with left input only

y = \***pwrap**(D,w,p+d); // delayed output - pwrap defined in dsplab.c

\*p = x; // delay-line input

p = **pwrap**(D,w,--p); // backshift pointer

yL = yR = (**short**) y;

**write\_outputs**(yL,yR); // write left and right output samples to codec

**return**;

}

1. Import and build provided a project on LMS for this program. Then, run it. Give the system an impulse by lightly tapping the table with the mike, and listen to the impulse response. Then, speak into the mike. You should hear repeated echoes.

**Done.**

1. Change the sampling rate to 16 kHz, recompile and reload keeping the value of d the same, that is, d = 4000. Listen to the impulse response. What is the duration of the delay in seconds now?

**Delay = 0.25s**

1. Reset the sampling rate back to 8 kHz, and this time change d to its maximum value d = D = 8000.Recompile, reload, and listen to the impulse response. Experiment with lower and lower values of d and listen to your delayed voice until you can no longer distinguish a separate echo. How many milliseconds of delay does this correspond to?

**d=8000, delay=1 second**

**d=4000, delay=0.5 second**

**d=2000, delay=0.25 second**

**At d = 2000, the echo was not distinguishable.**

1. In this part you will profile the computational cost of the sample processing algorithm. Open the source file template.c in a CCS window. Locate the read\_inputs line in the isr(), then right-click on that line and choose Breakpoint; a dot will appear in the margin. Do the same for the write\_outputs line. From the top menu of the CCS window, choose Run -> Clock ; a little yellow clock will appear on the right bottom status line of CCS. When you compile, load, and run your program, it will stop at the first breakpoint, with a arrow pointing to it. Reset the clock by chossing Run -> Clock🡪reset to clear the number of cycles, then click resume to continue running the program and it will stop at the second breakpoint. Read and record the number of cycles shown next to the profile clock.

**Break point 1 = 210796 cycles**

**Break point 2 = 343 cycles**

1. Change the code of tamplate.c to makes use of the function qwrap instead of pwrap. Repeat parts (a) and (d).

**Code:**

**Interrupt void isr**() // sample processing algorithm - interrupt service routine

{

**read\_inputs**(&xL, &xR); // read left and right input samples from codec

x = (**float**) xL; // work with left input only

yL = yR = (**short**) y;

y = w[**qwrap**(D,q+d)]; // delayed output

w[q] = x; // delay-line input

q = **qwrap**(D,--q); // backshift pointer index

**write\_outputs**(yL,yR); // write left and right output samples to codec

**return**;

}

**Approximate delay = 0.5 seconds.**

**Breakpoint 1 = 210716 cycles**

**Breakpoint 2 = 365 cycles cycles**

1. Next, Change the code of tamplate.c so that it uses linear buffers. Its isr() will be as follows:

interrupt void isr()

{

int i;

read\_inputs(&xL, &xR);

x = (float) xL;

w[0] = x; // delay-line input

y = w[d]; // delay output

for (i=D; i>=0; i--) // update linear buffer

w[i] = w[i-1];

yL = yR = (short) y;

write\_outputs(yL,yR);

return;

}

Build the project. You will find that it may not run (because the data shifts require too many cycles that over-run the sampling rate). Change the program parameters D and d to the following values D = 2000 and d = 1000. Rebuild and run the program. Repeat part (d) and record the number of cycles. Change the parameters D, d of the program delay1.c to the same values, and repeat part (d) for that. Comment on the required number of samples using the linear vs. the circular buffer implementation.

**When d=1000 and D=2000:**

**Breakpoint 1 = 102729 cycles**

**Breakpoint 2 = 49261 cycles**

**When d=4000 and D=8000:**

**Breakpoint 1 = 210737 cycles**

**Breakpoint 2 = 196256 cycles**

**Linear method requires iteration, as a result, it is not efficient. While the circular buffer implementation reduces the computation time.**

## FIR Comb Filters Using Circular Buffers

More interesting audio effects can be derived by combining several multiple delays. An example is the FIR comb filter defined by :



Its transfer function is given by Eq:



Its impulse response has a very sparse structure:

****

The comb-like structure of its frequency response and its zero-pattern on the z-plane are depicted . Instead of implementing it as a general FIR filter, a more efficient approach is to program the block diagram directly by using a single delay line of order 3D and tapping it out at taps 0, D, 2D, and 3D. The block diagram realization and corresponding sample processing algorithm are:



The translation of the sample processing algorithm into C is straightforward and can be incorporated into the following isr() function to be included in your main program:

interrupt void isr()

{

float s0, s1, s2, s3, y; // states & output

read\_inputs(&xL, &xR); // read inputs from codec

s0 = (float) xL; // work with left input only

s1 = \*pwrap(3\*D,w,p+D); // extract states relative to p

s2 = \*pwrap(3\*D,w,p+2\*D); // note, buffer length is 3D+1

s3 = \*pwrap(3\*D,w,p+3\*D);

y = s0 + a\*s1 + a\*a\*s2 + a\*a\*a\*s3; // output sample

\*p = s0; // delay-line input

p = pwrap(3\*D,w,--p); // backshift pointer

yL = yR = (short) y;

write\_outputs(yL,yR); // write outputs to codec

return;

}

### Lab Procedure 2

Set the sampling rate to 8 kHz and the audio source to microphone. Choose the delay to be D = 4000,corresponding to TD = 0.5 sec, so that the total duration of the filter is 3TD = 1.5 sec, and set a = 0.5.

1. Write a C program called comb.c that incorporates the above interrupt service routine. You will need to globally declare/define the parameters D, a, p, as well as the circular buffer w to be a 3D+1 dimensional float array. Make sure you initialize the buffer to zero inside main(), as was done in the previous example, and also initialize p = w.Build and run this project. Listen to the impulse response of the filter by tapping the table with the mike. Speak into the mike. Bring the mike.

**There are repeated echoes.**

1. Keeping the delay D the same, choose a = 0.2 and run the program again. What effect do you hear? Repeat for a = 0.1. Repeat with a = 1.

**The amplitude of the repeated echoes decreases.**

1. Set the audio input to LINE and play your favorite wave or MP3 song into the input. Experiment with reducing the value of D in order to match your song’s tempo to the repeated echoes.
2. The FIR comb can also be implemented recursively using the geometric series formula to rewrite its transfer function in the recursive form as shown in Eq. (8.2.9) of the text:



This requires a (4D+1)-dimensional delay-line buffer w. The canonical realization and the corresponding

sample processing algorithm are shown below:



Write a new program, comb2.c, that implements this algorithm. Remember to define the buffer to be a (4D+1)-dimensional float array. Using the values D = 1600 (corresponding to a 0.2 sec delay) and a = 0.5, recompile and run both the comb.c and comb2.c programs and listen to their outputs. In general, such recursive implementations of FIR filters are more prone to the accumulation of roundoff errors than the non-recursive versions. You may want to run these programs with a = 1 to observe this sensitivity.

**Code:**

**Interrupt void isr**() // sample processing algorithm - interrupt service routine

{

**float** s0, s1, s4, y; // states & output

**read\_inputs**(&xL, &xR); // read inputs from codec

s1 = \***pwrap**(4\*D,w,p+D); // extract states relative to p

s4=\***pwrap**(4\*D,w,p+4\*D);

s0=(**float**) xL+a\*s1;

y = s0 - a\*a\*a\*a\*s4; // output sample

\*p = s0; // delay-line input

p = **pwrap**(4\*D,w,--p); // backshift pointer

yL = yR = (**short**) y;

**write\_outputs**(yL,yR); // write outputs to codec

**return**;

}

**There are varying amplitudes of the repeated echoes and are different delays as compared to previous. Hence we hear reducing echo sound.**

**Conclusion:**

In this lab, we learnt, how to implement FIR Filters and in DSK trainer board. Different delays were implemented to produce different repeated echoes. Secondly, we FIR comb filters using the circular buffers and linear iterative buffer, and circular buffer was less time consuming.

## References

[1] S. J. Orfanidis, Introduction to Signal Processing, online book, 2010, available from:

http://www.ece.rutgers.edu/~orfanidi/intro2sp/

[2] R. Chassaing and D. Reay, Digital Signal Processing and Applications with the TMS320C6713 and

TMS320C6416 DSK, 2nd ed., Wiley, Hoboken, NJ, 2008.