Lab 3 – Delays and FIR Filtering

3.1. Introduction

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, such as reverb.

Delays can be implemented using linear or circular buffers, the latter being more efficient, especially for audio effects. The theory behind this lab is developed in Ch. 4 of the text [1] for FIR filters, and used in Ch. 8 for audio effects.

3.2. 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:

$$T_D = DT = \frac{D}{f_s} \quad\Rightarrow\quad D = f_s T_D \tag{3.1}$$

where T is the time interval between samples, related to the sampling rate by $f_s = 1/T$. A block diagram realization of the multiple delay is shown below:

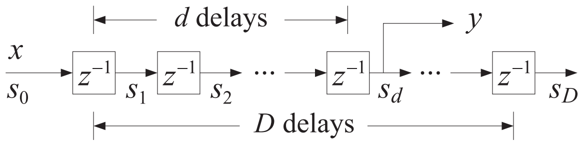


Fig. 3.1 Tapped delay line.

There are D registers whose contents are the "internal" states of the delay line. The dth state s_d, i.e., the content of the dth register, represents the d-fold delayed version of the input, that is, at time n we have: $s_d(n)= x(n-d)$, for $d = 1,\dots,D$; the case $d = 0$ corresponds to the input $s_0(n)= x(n)$.

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 $y = s_d$ is the partial delay z^{-d}.

The D contents/states s_d, $d = 1,2,\dots,D$, and the input $s_0 = x$ 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.

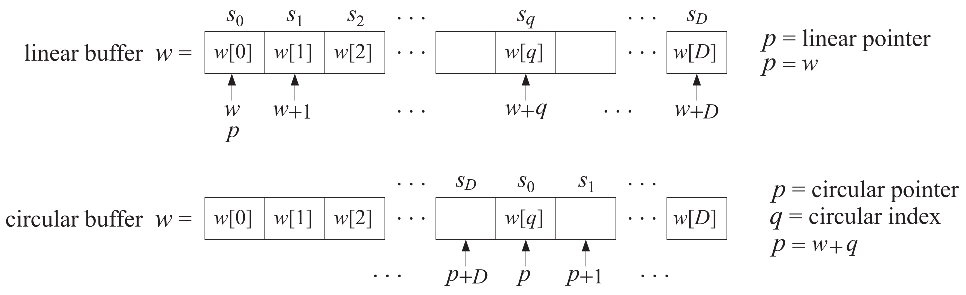


Fig. 3.2 Linear and circular buffers.

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

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

Its contents are retrieved as $w[i]$, $i = 0, 1, \dots, D$. Thinking of w as a 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:

$$s_i = w[i] = *(w + i)\,, \quad i = 0, 1, \dots, D$$

At each time instant, after the contents s_i 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. 3.1. This follows from the definition $s_i(n) = x(n - i)$, which implies for the next time instant $s_i(n+1) = x(n+1-i) = s_{i-1}(n)$. Thus, the current s_{i-1} becomes the next s_i. Since $s_i = w[i]$, 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 $s_i = *(p + i)$, $i = 0, 1, \dots, D$. 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 end of the buffer if it exceeds the array bounds to the left. Fig. 3.3 depicts the contents and pointer positions at two successive time instants for the linear and circular buffer cases for $D = 3$. In both cases, the states are retrieved by $s_i = *(p + i)$, $i = 0, 1, 2, 3$, 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.

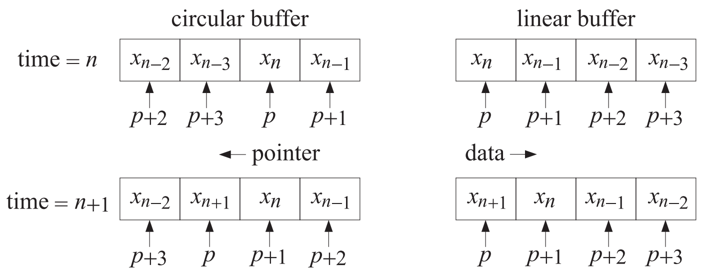


Fig. 3.3 Buffer contents at successive time instants for $D = 3$.

In the text [1], the functions `tap()` and `cdelay()` are used for extracting the states s_i 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` in the directory `C:\dsplab\common`. Its listing is as follows:

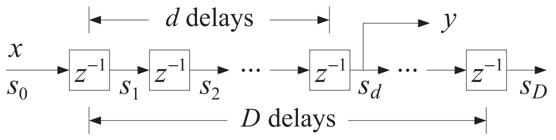
```
// pwrap.c - pointer wrapping relative to circular buffer
// Usage: p_new = pwrap(D,w,p)
// -------------------------------------------------------

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 s_i and the updating of the delay-line can be obtained by the function calls:

$$s_i = *\text{pwrap}(D, w, p+i)\,, \quad i = 1, 2, \dots, D$$

$$p_{\text{next}} = \text{pwrap}(D, w, --p)$$

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 into 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. 3.1 and the pseudo-code computational algorithm are as follows:



```
for each input x do:
    $y = s_d = *(p + d)$
    $*p = x$
    $--p$
```

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 inside `pwrap`, that is, `--p`, instead of post-decrementing it, `p--`. Why? 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 pointers. The pointer p always points at 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. 3.2. The index q is always bound by the limits $0 \le q \le D$ 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 the circular 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:

```
qd = qwrap(D,q+d);              // (q+d) mod (D+1)
y = w[qd];                      // delayed output
w[q] = x;                       // delay-line input
q = qwrap(D,--q);               // backshift pointer index
```

We note that in general, the ith state is:

$$s_i = *(p+i) = *(w+q+i) = w[q+i]$$

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

$$q_i = \text{qwrap}(D, q+i)\,, \quad s_i = w[q_i]\,, \quad i = 1, 2, \dots, D$$

Lab Procedure

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++) w[n] = 0;   // initialize circular buffer to zero
   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.

a. Create and build a project 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.

   Bring the mike near the speaker and then give the system an impulse. You should hear repeated echoes. If you bring the mike too close to the speakers the output goes unstable. Draw a block diagram realization that would explain the effect you are hearing. Experimentally determine the distance at which the echoes remain marginally stable, that is, neither die out nor diverge. (Technically speaking, the poles of your closed-loop system lie on the unit circle.)

b. 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?

c. 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. Set $d = 0$, recompile and reload. This should correspond to no delay at all. But what do you hear? Can you explain why? Can you fix it by changing the program? Will your modified program still work with $d \neq 0$? Is there any good reason for structuring the program the way it was originally?

e. In this part you will profile the computational cost of the sample processing algorithm. Open the source file `delay1.c` in a CCS window. Locate the `read_inputs` line in the `isr()`, then right-click on that line and choose *Toggle Software Breakpoint*; a red dot will appear in the margin. Do the same for the `write_outputs` line.

   From the top menu of the CCS window, choose *Profile -> Clock -> View*; 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 yellow arrow pointing to it. Double-click on the profile clock to clear the number of cycles, then type `F5` 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.

f. Write a new program, called `delay2.c`, that makes use of the function `qwrap` instead of `pwrap`. Repeat parts (a) and (e).

g. Next, write a new program, called `delay3.c`, that 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, d to the following values $D = 2000$ and $d = 1000$. Rebuild and run the program. Repeat part (e) and record the number of cycles. Change the parameters D, d of the program `delay1.c` to the same values, and repeat part (e) for that. Comment on the required number of samples using the linear vs. the circular buffer implementation.

3.3. 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 Eq. (8.2.8) of the text [1]:

$$y_n = x_n + a\,x_{n-D} + a^2 x_{n-2D} + a^3 x_{n-3D}$$

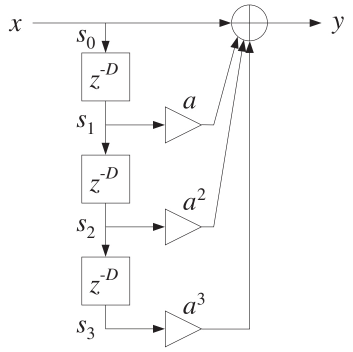
Its transfer function is given by Eq. (8.2.9):

$$H(z) = 1 + az^{-D} + a^2 z^{-2D} + a^3 z^{-3D}$$

Its impulse response has a very sparse structure:

$$\mathbf{h} = [1, \underbrace{0, 0, \dots, 0}_{D-1 \text{ zeros}}, a, \underbrace{0, 0, \dots, 0}_{D-1 \text{ zeros}}, a^2, \underbrace{0, 0, \dots, 0}_{D-1 \text{ zeros}}, a^3]$$

The comb-like structure of its frequency response and its zero-pattern on the z-plane are depicted in Fig. 8.2.5 of [1]. 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:



```
for each input x do:
    $s_0 = x$
    $s_1 = *(p + D)$
    $s_2 = *(p + 2D)$
    $s_3 = *(p + 3D)$
    $y = s_0 + as_1 + a^2 s_2 + a^3 s_3$
    $*p = s_0$
    $--p$
```

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

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

a. 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 close to the speakers and get a closed-loop feedback.

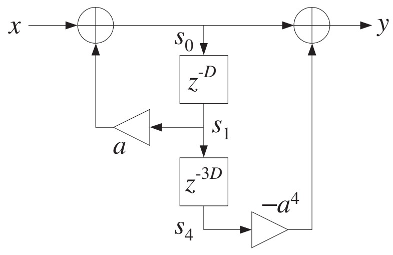
b. Keeping the delay D the same, choose $a = 0.2$ and run the program again. What effect do you hear? Repeat for $a = 0.1$. Repeat with $a = 1$.

c. Set the audio input to LINE and play your favorite wave file or MP3 into the input. Experiment with reducing the value of D in order to match your song's tempo to the repeated echoes. Some wave files are in the directory `c:\dsplab\wav` (e.g., try `dsummer`, `take5`.)

d. 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:

$$H(z) = 1 + az^{-D} + a^2z^{-2D} + a^3z^{-3D} = \frac{1 - a^4z^{-4D}}{1 - az^{-D}}$$

   This requires a $(4D+1)$-dimensional delay-line buffer w. The canonical realization and the corresponding sample processing algorithm are shown below:



```
for each input x do:
    $s_1 = *(p + D)$
    $s_4 = *(p + 4D)$
    $s_0 = x + as_1$
    $y = s_0 - a^4 s_4$
    $*p = s_0$
    $--p$
```

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.

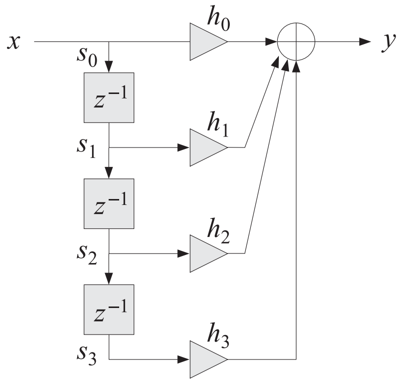
In general, such recursive implementations of FIR filters are more prone to the accumulation of round-off errors than the non-recursive versions. You may want to run these programs with $a = 1$ to observe this sensitivity.

3.4. FIR Filters with Linear and Circular Buffers

The sample-by-sample processing implementation of FIR filters is discussed in Sect. 4.2 of the text [1]. For an order-M filter, the input/output convolutional equation can be written as the dot product of the filter-coefficient vector $\mathbf{h} = [h_0, h_1, \dots, h_M]^T$ with the state vector $\mathbf{s}(n) = [x_n, x_{n-1}, \dots, x_{n-M}]^T$:

$$y_n = \sum_{m=0}^{M} h(m)x(n-m) = [h_0, h_1, \dots, h_M]\begin{bmatrix} x(n) \\ x(n-1) \\ \vdots \\ x(n-M) \end{bmatrix} = \mathbf{h}^T\mathbf{s}(n)\,, \quad \mathbf{s}(n) = \begin{bmatrix} x(n) \\ x(n-1) \\ \vdots \\ x(n-M) \end{bmatrix}$$

A block diagram realization for the case $M = 3$ is depicted below.



We note that the ith component of the state vector is $s_i(n) = x(n-i)$, $i = 0, 1, \dots, M$, and therefore, the states are the tap outputs of a multiple delay-line with M delays. Thus, the definition of the delay line and its time updating remains the same as in the previous sections. To realize the FIR filter, we must use the tapped outputs s_i from the delay line to calculate the dot product, and then update the delay line to the next time instant.

In this lab, we consider five implementations of FIR filters and study their relative efficiency in terms of machine cycles at different levels of compiler optimization:

```
y = fir(M, h, w, x);         - linear buffer implementation
y = firc(M, h, w, &p, x);    - circular buffer with pointers
y = firc2(M, h, w, &q, x);   - circular index with updating in loop
y = firq(M, h, w, &q, x);    - circular index with updating outside loop
y = fira(w, h, Lh, Nb, q);   - circular buffer in linear assembly
```

These functions are defined below. The function, `fir`, implements the linear buffer case:

```
// fir.c - FIR filter in direct form with linear buffer
// Usage: y = fir(M, h, w, x);
// ----------------------------------------------------------------

float fir(int M, float *h, float *w, float x)
{
   int i;
   float y;                         // y=output sample

   w[0] = x;                        // read current input sample x

   for (y=0, i=0; i<=M; i++)        // process current output sample
       y += h[i] * w[i];            // dot-product operation

   for (i=M; i>=1; i--)             // update states for next call
      w[i] = w[i-1];                // done in reverse order

   return y;
}

// ----------------------------------------------------------------
```

The function `firc` implements the circular buffer version using the pointer-wrapping function `pwrap`:

```
// firc.c - FIR filter implemented with circular pointer
// Usage: y = firc(M, h, w, &p, x);
// ----------------------------------------------------------------------

float *pwrap(int, float *, float *);               // defined in dsplab.c

float firc(int M, float *h, float *w, float **p, float x)
{
   int i;
   float y;

   **p = x;                          // read input sample x

   for (y=0, i=0; i<=M; i++) {
      y += (*h++) * (**p);           // i-th state s[i] = *pwrap(M,w,*p+i)
      *p = pwrap(M,w,++*p);          // pointer to state s[i+1] = *pwrap(M,w,*p+i+1)
      }

   *p = pwrap(M,w,--*p);             // update circular delay line

   return y;
}

// ----------------------------------------------------------------------
```

The function `firc2` is a circular buffer version using the pointer-index-wrapping function `qwrap`:

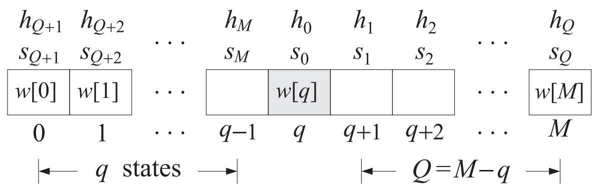
```
// firc2.c - FIR filter implemented with circular index
// Usage: y = firc2(M, h, w, &q, x);
// ----------------------------------------------------------------------

int qwrap(int, int);                               // defined in dsplab.c

float firc2(int M, float *h, float *w, int *q, float x)
{
   int i;
   float y;

   w[*q] = x;                         // read input sample x

   for (y=0, i=0; i<=M; i++) {
      y += *h++ * w[*q];              // i-th state s[i] = w[*q]
      *q = qwrap(M,++*q);             // pointer to state s[i+1] = w[*q+1]
      }

   *q = qwrap(M,--*q);                // update circular delay line

   return y;
}

// ----------------------------------------------------------------------
```

In both `firc` and `firc2`, the circular pointer or index are being wrapped during each pass through the for-loop that computes the output sample y. This is inefficient but necessary because C does not support circular arrays.

All modern DSP chips, including the C6713, support circular addressing in hardware, which does the required wrapping automatically without any extra instructions. The following function, `firq`, emulates the hardware version more closely by avoiding the repeated calls to `qwrap` inside the for-loop—it performs only one wrapping when it reaches the end of the buffer and wraps the index back to $q = 0$; furthermore, it wraps once more after the for-loops in order to backshift the pointer index:

```
// firq.c - FIR filter implemented with circular index
// Usage: y = fircq(M, h, w, &q, x);
// ----------------------------------------------------------------------

float firq(int M, float *h, float *w, int *q, float x)
{
   int i, Q;
   float y;

   Q = M - (*q);                          // number of states to end of buffer

   w[*q] = x;                             // read input sample x

   for (y=0, i=0; i<=Q; i++)              // loop from q to end of buffer
      y += h[i] * w[(*q)++];

   (*q) = 0;                              // wrap to beginning of buffer

   for (i=Q+1; i<=M; i++)                 // loop to q-1
      y += h[i] * w[(*q)++];

   (*q)--;  if (*q == -1) *q = M;         // backshift index

   return y;
}

// ----------------------------------------------------------------------
```

The for-loop is split into two parts, the first part starts at position q and loops until the end of the buffer, then it wraps to the beginning of the buffer; the second part loops till $q - 1$. The required states s_i of the FIR filter and their association with the filter coefficients h_i are depicted below.



Finally, we consider the *linear assembly* function, `fira.sa`, listed below, that exploits the hardware implementation of circular buffers on the C6713 processor. It is based on the function `convol1.sa` of Ref. [3], adapted here to our convention of counting the states and filter coefficients in forward order and updating the circular index by backshifting. Linear assembly functions have an extension `.sa` and may be included in a project just like C functions. The linear assembly optimizer determines which particular hardware registers to assign to the various local variables in the function.

```
; fira.sa - linear assembly version of FIR filter with circular buffer
;
; extern float fira(float *, float *, int, int, int);
;
; float w[Lw];
; #pragma DATA_ALIGN(w, Lb)
;
```

```
; usage: w[q] = x;                     read input sample
;        y = fira(w,h,Lh,Nb,q);        compute output sample
;        q--; if (q==-1) q = Lw-1;     update circular index by backshifting
;
; M                          = filter order
; Lh  = M+1                  = filter length
; Nb >= 1 + ceil(log2(Lh))   = circular buffer bytes-length exponent
; Lb  = 2^(Nb+1)             = circular buffer length in bytes
; Lw  = Lb/4 = 2^(Nb-1)      = circular buffer in 32-bit words
; ----------------------------------------------------------------------

        .global _fira
_fira   .cproc  w, h, Lh, Nb, q       ; function arguments
        .reg  Y, P, si, hi            ; local variables

      ADDAW   w, q, w       ; point to w[q] = x = current input
                            ; set up the circular buffer
      SHL Nb, 16, Nb        ; shift Nb to BK0 field
      set Nb, 8,8, Nb       ; set circular mode, BK0, B4
      MVC Nb, AMR           ; load mode into AMR

      ZERO  Y               ; output

loop: .trip 8, 500          ; assume between 8 and 500 taps

      LDW *w++, si          ; load i-th state, si = x(n-i)
      LDW *h++, hi          ; load i-th filter coeff, h(i)
      MPYSP  si,hi,P        ; multiply single precision, P = hi*si
      ADDSP P,Y,Y           ; Y = Y + P = accumulate output

 [Lh] SUB Lh, 1, Lh         ; decrement, Lh = Lh-1
 [Lh] B  loop               ; loop until Lh=0

      .return Y             ; put sum in A4 - C convention

      .endproc

; ----------------------------------------------------------------------
```

Lab Procedure

A lowpass FIR filter of order M and cutoff frequency f_0 can be designed using the Hamming window approach by the following equations (see Ch.11 of [1]):

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M}\right), \quad h(n) = w(n)\,\frac{\sin\big(\omega_0(n-M/2)\big)}{\pi(n-M/2)}, \quad 0 \le n \le M$$

where $\omega_0 = 2\pi f_0/f_s$, and $w(n)$ is the Hamming window.

a. Design such a filter with MATLAB using the following values: $f_s = 8$ kHz, $f_0 = 2$ kHz, and filter order $M = 100$. Then, using the built-in MATLAB function `freqz`, or the textbook function `dtft`, calculate and plot in dB the magnitude response of the filter over the frequency interval $0 \le f \le 4$ kHz. The designed filter response is shown in Fig. 3.4 in absolute units and in dB.

b. The designed 101-long impulse response coefficient vector **h** can be exported into a data file, `h.dat`, in a form that is readable by a C program by the following MATLAB command:

```
C_header('h.dat', 'h', 'M', h);
```

where `C_header` is a MATLAB function in the directory `c:\dsplab\common`. A few lines of the resulting data file are shown below:

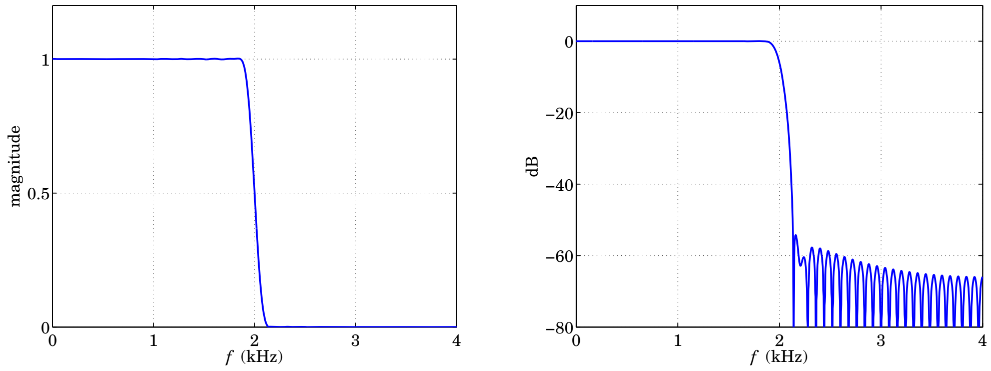


Fig. 3.4 Magnitude response of lowpass filter.

```
// h.dat  -  FIR impulse response coefficients
// exported from MATLAB using C_header.m
// ------------------------------------------

#define M 100        // filter order

float h[M+1] = {
   -0.000000000000000,
    0.000525586170315,
   -0.000000000000000,
   -0.000596985888698,
    0.000000000000000,
    0.000725138232653,
      --- etc. ---
   -0.000596985888698,
   -0.000000000000000,
    0.000525586170315,
   -0.000000000000000
   };

// ------------------------------------------
```

The following complete C program, `firex.c`, implements this example on the C6713 processor. The program reads the impulse response vector from the data file `h.dat`, and defines a 101-dimensional delay-line buffer array `w`. The FIR filtering operation is based on any of the choices, `fir, firc, firc2, firq,` depending on which lines are uncommented.

```
// firex.c - FIR filtering example
// ------------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes

//float fir(int, float *, float *, float);
//float firc(int, float *, float *, float **, float);
//float firc2(int, float *, float *, int *, float);
//float firq(int, float *, float *, int *, float);

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

#include "h.dat"    // contains M+1 = 101 filter coefficients

float w[M+1];       // filter delay lines
int on = 1;         // turn filter on
//float *p;
//int q;

// ------------------------------------------------------------------------------------
```

```
void main()
{
  int i;

  for (i=0; i<=M; i++) w[i] = 0;     // initialize delay-line buffer
  //p = w;                           // initialize circular pointer
  //q = 0;

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

  sampling_rate(8);         // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz
  audio_source(LINE);       // LINE or MIC for line or microphone input

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

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

interrupt void isr()
{
   float x, y;                         // filter input & output

   read_inputs(&xL, &xR);              // read audio samples from codec

   if (on) {
      x = (float)(xL);                 // work with left input only

      //y = fir(M,h,w,x);
      //y = firc(M,h,w,&p,x);
      //y = firc2(M, h, w, &q, x);
      //y = firq(M, h, w, &q, x);

      yL = (short)(y);
      }
   else                                // pass through if filter is off
      yL = xL;

  write_outputs(yL,yL);                // write audio samples to codec

  return;
}

// ------------------------------------------------------------------------------------
```

Create and build a project for this program. You will need to add one of the functions `fir, firc, firc2, firq` to the project. Using the following MATLAB code (same as in the aliasing example of Lab-1), generate a signal consisting of a 1-kHz segment, followed by a 3-kHz segment, followed by another 1-kHz segment, where all segments have duration of 1 sec:

```
fs = 8000; f1 = 1000; f2 = 3000; f3 = 1000;
L = 8000; n = (0:L-1);
A = 1/5;                           % adjust playback volume

x1 = A * cos(2*pi*n*f1/fs);
x2 = A * cos(2*pi*n*f2/fs);
x3 = A * cos(2*pi*n*f3/fs);

sound([x1,x2,x3], fs);
```

First, set the parameter `on=0` so that the filtering operation is bypassed. Send the above signal into the line input of the DSK and listen to the output. Then, set `on=1` to turn the filter on using the linear buffer version, `fir`, recompile and run the program, and send the same signal in. The middle 3-kHz segment should not be heard, since it lies in the filter's stopband.

c. Create breakpoints at the `read_inputs` and `write_outputs` lines of the `isr()` function, and start the profile clock. Run the program and record the number of cycles between reading the input samples and writing the computed outputs.

d. Uncomment the appropriate lines in the above program to implement the circular buffer versions using the functions `firc, firc2, firq`. You will need to add these to your project. Recompile and run your program with the same input.

   Then, repeat part (c) and record the number of computation cycles.

e. The compiler optimization thus far was set to "none". Using the keyboard combination "ALT-P P", or the CCS menu commands *Project -> Build Options*, change the optimization level to `-o0, -o1, -o2, -o3,` and for each level and each of the four filter implementations `fir, firc, firc2, firq,` repeat part (c) and record the number of cycles in a table form:

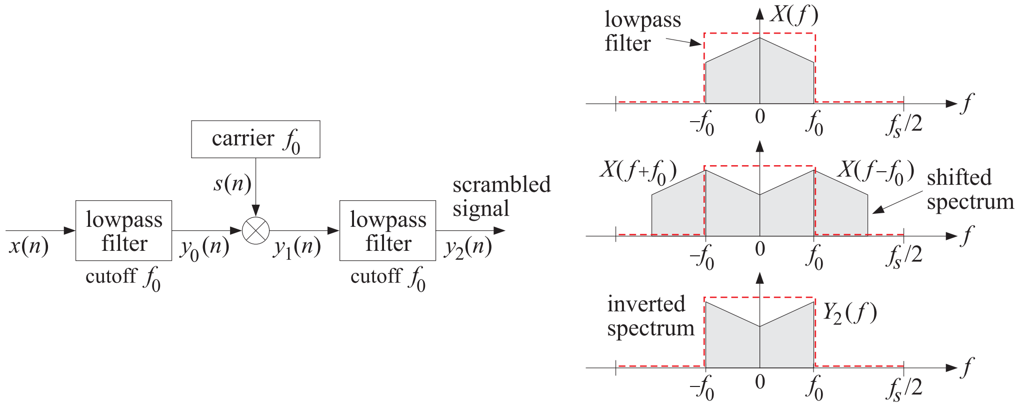
	none	-o0	-o1	-o2	-o3
fir					
firc					
firc2					
firq					
fira					

f. Add to the above table the results from the linear assembly version implemented by the following complete C program, `firexa.c`, and evaluate your results in terms of efficiency of implementation and optimization level.

```
// firexa.c - FIR filtering example using circular buffer with linear assembly
// ------------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes

extern float fira(float *, float *, int, int, int);

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

#include "h.dat"    // contains M+1 = 101 filter coefficients

#define Nb 8        // circular-buffer length (bytes) exponent, Nb = 1 + ceil(log2(M+1)) = 8
#define Lb 512      // circular-buffer length (bytes) = 2^(Nb+1)
#define Lw 128      // circular-buffer length (words) = 2^(Nb-1) = Lb/4
#define Lh 101      // filter length = M+1

float w[Lw];                  // circular buffer

#pragma DATA_ALIGN(w, Lb)     // align buffer at byte-boundary

int q;                        // circular-buffer index
int on = 1;                   // filter is ON or OFF

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

void main()
{
  int i;

  for (i=0; i<Lw; i++) w[i] = 0;    // initialize circular buffer to zero
  q = 0;                            // initialize index into buffer

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

  sampling_rate(8);          // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz
  audio_source(LINE);        // LINE or MIC for line or microphone input
  while(1);                  // keep waiting for interrupt, then jump to isr()
}

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

interrupt void isr()
{

  float x, y;                          // filter input & output

   read_inputs(&xL, &xR);

   if (on) {
      x = (float)(xL);                 // work with left input only

      w[q] = x;                        // put x into w[q],

      y = fira(w, h, Lh, Nb, q);       // fira does not update q

      q--;  if (q == -1) q = Lw-1;     // backshift to update q for next time instant

      yL = (short)(y);
      }
   else
      yL = xL;

  write_outputs(yL,yL);

  return;
}

// -----------------------------------------------------------------------------------
```

3.5. Voice Scrambler

A simple voice scrambler works by spectrum inversion. It is not the most secure way of encrypting speech, but we consider it in this lab as an application of low pass filtering and AM modulation. The main operations are depicted below.



First, the sampled speech signal $x(n)$ is filtered by a lowpass filter $h(n)$ whose cutoff frequency f_0 is high enough not to cause distortions of the speech signal (the figure depicts an ideal filter). The sampling rate f_s is chosen such that $4f_0 < f_s$. The filtering operation can be represented by the convolutional equation:

$$y_0(n) = \sum_m h(m)x(n-m) \tag{3.2}$$

Next, the filter output $y_0(n)$ modulates a cosinusoidal carrier signal whose frequency coincides with the filter's cutoff frequency f_0, resulting in the signal:

$$y_1(n) = s(n)y_0(n), \quad \text{where} \quad s(n) = 2\cos(\omega_0 n), \quad \omega_0 = \frac{2\pi f_0}{f_s} \tag{3.3}$$

The multiplication by the carrier signal causes the spectrum of the signal to be shifted and centered at $\pm f_0$, as shown above. Finally, the modulated signal $y_1(n)$ is passed through the same filter again which removes the spectral components with $|f| > f_0$, resulting in a signal $y_2(n)$ with inverted spectrum. The last filtering operation is:

$$y_2(n) = \sum_m h(m)y_1(n-m) \tag{3.4}$$

To unscramble the signal, one may apply the scrambling steps (3.2)–(3.4) to the scrambled signal itself. This works because the inverted spectrum will be inverted again, recovering in the original spectrum.

In this lab, you will study a real-time implementation of the above procedures. The lowpass filter will be designed with the parameters $f_s = 16$ kHz, $f_0 = 3.3$ kHz, and filter order $M = 100$ using the Hamming design method:

$$h(n) = w(n)\,\frac{\sin\big(\omega_0(n-M/2)\big)}{\pi(n-M/2)}, \quad 0 \le n \le M \tag{3.5}$$

where $\omega_0 = 2\pi f_0/f_s$, and $w(n)$ is the Hamming window:

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M}\right), \quad 0 \le n \le M \tag{3.6}$$

The following C program, `scrambler.c`, forms the basis of this lab. It is a variation of that discussed in the Chassaing-Reay text [2].

```
// scrambler.c - voice scrambler example
// ------------------------------------------------------------------------------------------

#include "dsplab.h"                // initialization declarations and function prototypes
#include <math.h>
#define PI 3.14159265358979

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

#define M 100                      // filter order
#define L 160                      // carrier period, note L*f0/fs = 160*3.3/16 = 33 cycles

float h[M+1], w1[M+1], w2[M+1];    // filter coefficients and delay-line buffers
int n=0;                           // time index for carrier, repeats with period L
int on=1;                          // turn scrambler on (off with on=0)

float w0, f0 = 3.3;                // f0 = 3.3 kHz
short fs = 16;                     // fs = 16 kHz

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

void main()
{
  int i;
  float wind;

  w0 = 2*PI*f0/fs;                             // carrier frequency in rads/sample

  for (i=0; i<=M; i++)  {                      // initialize buffers & design filter
      w1[i] = w2[i] = 0;
      wind = 0.54 - 0.46 * cos(2*PI*i/M);      // Hamming window
      if (i==M/2)
         h[i] = w0/PI;
      else
         h[i] = wind * sin(w0*(i-M/2)) / (PI*(i-M/2));
      }
  initialize();             // initialize DSK board and codec, define interrupts

  sampling_rate(fs);        // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz
  audio_source(LINE);       // 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
{
   int i;
   float y;

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

   if (on) {
      y = (float)(xL);                // work with left input only

      w1[0] = y;                      // first filter
      for (y=0, i=0; i<=M; i++)
         y += h[i] * w1[i];
      delay(M,w1);

      y *= 2*cos(w0*n);               // multiply y by carrier
      if (++n >= L) n = 0;



      w2[0] = y;                      // second filter
      for (y=0, i=0; i<=M; i++)
         y += h[i] * w2[i];
      delay(M,w2);

      yL = (short)(y);
      }
   else
      yL = xL;                        // pass through if on=0

  write_outputs(yL,yL);

  return;
}

// ------------------------------------------------------------------------------------
```

Two separate buffers, w_1, w_2, are used for the two lowpass filters. The filter coefficients are computed on the fly within `main()` using Eqs. (3.5) and (3.6). A linear buffer implementation is used for both filters. The sinusoidal carrier signal is defined by:

$$s[n] = 2\cos(\omega_0 n)\,, \quad \omega_0 = \frac{2\pi f_0}{f_s}$$

Since $f_s/f_0 = 16/3.3$ samples/cycle, it follows that the smallest number of samples containing an integral number of cycles will be:

$$L = \frac{16}{3.3}\cdot 33 = 160 \text{ samples}$$

that is, these 160 samples contain 33 cycles and will keep repeating. Therefore, the time index n of $s[n]$ is periodically cycled over the interval $0 \le n \le L-1$.

Lab Procedure

a. Explain why the factor 2 is needed in the carrier definition $s(n) = 2\cos(\omega_0 n)$. Explain why f_0 must be chosen such that $4f_0 < f_s$ in designing the lowpass filter.

b. Create and build a project for this program. The parameter `on=1` turns the scrambler on or off. Create a GEL file for this parameter and open it when you run the program.

c. Play the following two wave files through program:

```
JB.wav
JBs.wav
```

When you play the second, which is a scrambled version of the first, it will get unscrambled.

d. Open MATLAB and generate three sinusoids of frequencies 300 Hz, 3000 Hz, and 300 Hz, sampled at a rate of 16 kHz, each of duration of 1 second, and concatenate them to form a 3-second signal. Then play this out of the PCs sound card using the `sound()` function. For example, the following MATLAB code will do this:

```
fs = 16000; f1 = 300; f2 = 3000; f3 = 300;
L = 16000; n = (0:L-1);
A = 1/5;                         % adjust playback volume

x1 = A * cos(2*pi*n*f1/fs);
x2 = A * cos(2*pi*n*f2/fs);
x3 = A * cos(2*pi*n*f3/fs);

sound([x1,x2,x3], fs);
```

Play this signal through the DSK with the scrambler off. Then, play it with the scrambler on. What are the frequencies in Hz of the scrambled signal that you hear? Explain this in your report.

e. Instead of actually computing the cosine function at each call of `isr()`, a more efficient approach would be to pre-compute the L repeating samples of the carrier $s[n]$ and keep re-using them. This can be accomplished by replacing the two modulation instructions in `isr()` by:

```
y *= s[n];               // multiply y by carrier
if (++n >= L) n = 0;
```

where $s[n]$ must be initialized within `main()` to the L values, $s[n]= 2\cos(\omega_0 n)$, $n = 0, 1, \dots, L-1$.

Re-write the above program to take advantage of this suggestion. Test your program.

In Lab-4, we will reconsider the scrambler and implement the required spectrum inversion using the FFT.

3.6. 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.

[3] S. A. Tretter, *Communication System Design Using DSP Algorithms with Laboratory Experiments for the TMS320C6713 DSK*, Springer, New York, 2008, code available from:
`http://www.ece.umd.edu/~tretter`