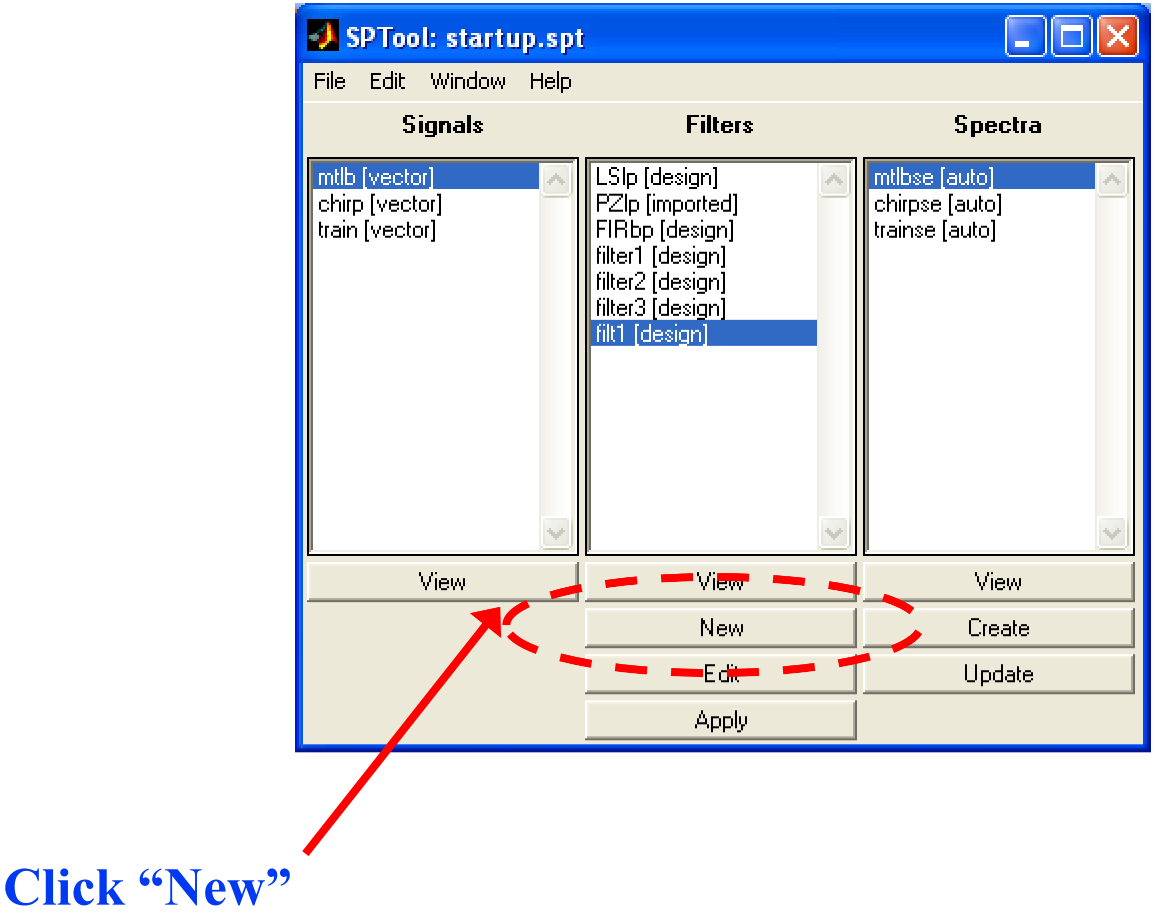
DSP Algorithms

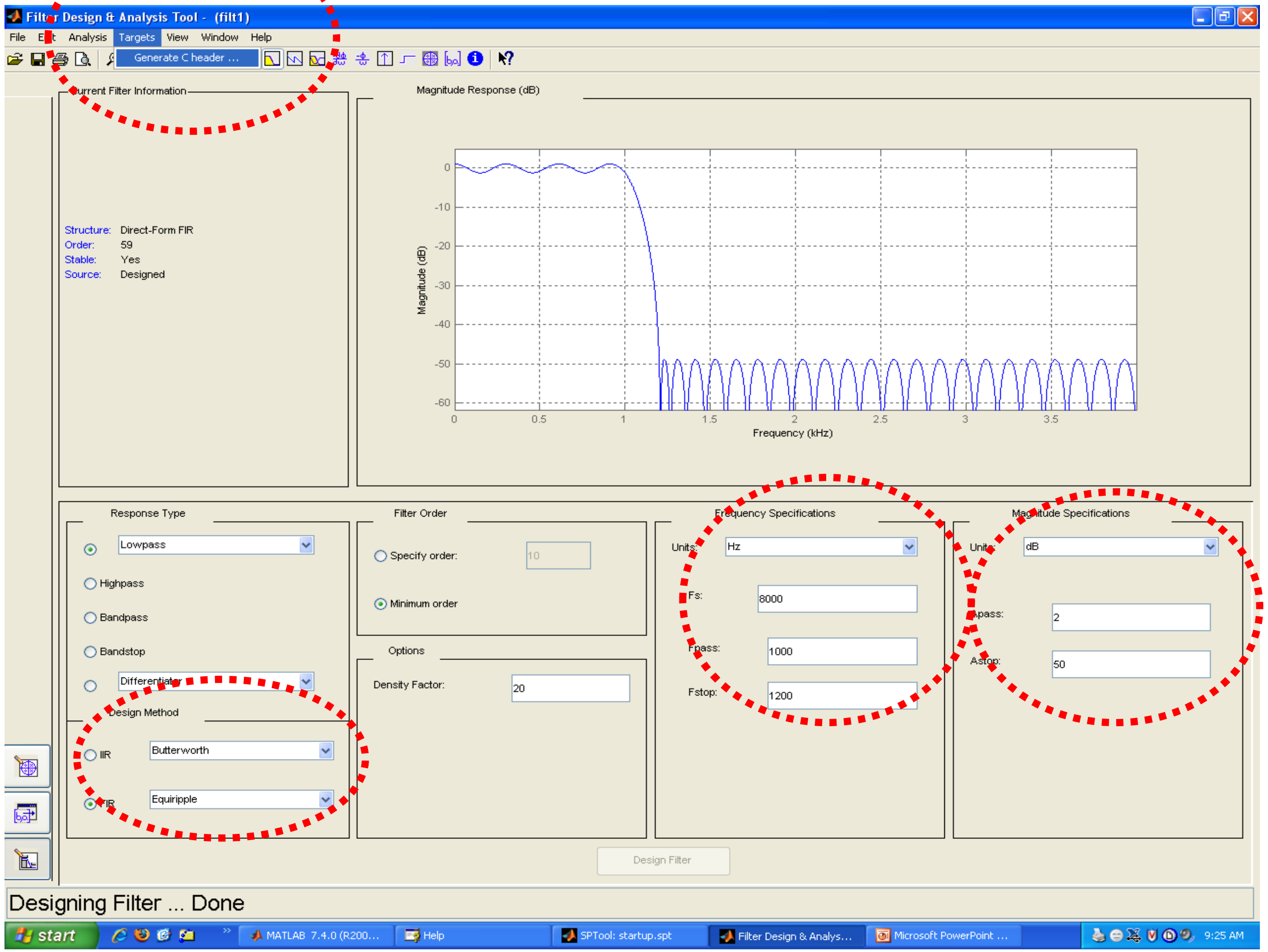
UNIVERSITY OF WASHINGTON
ELECTRICAL ENGINEERING

MATLAB GUI Filter Design

- Open Matlab
- type "sptool" at command line
- See start up window "startup.spt"
- There are 3 analysis options, we will discuss only Filters
- Click " New" for the first time.
- Click " Edit" when you want to change the old design.



Design Filters

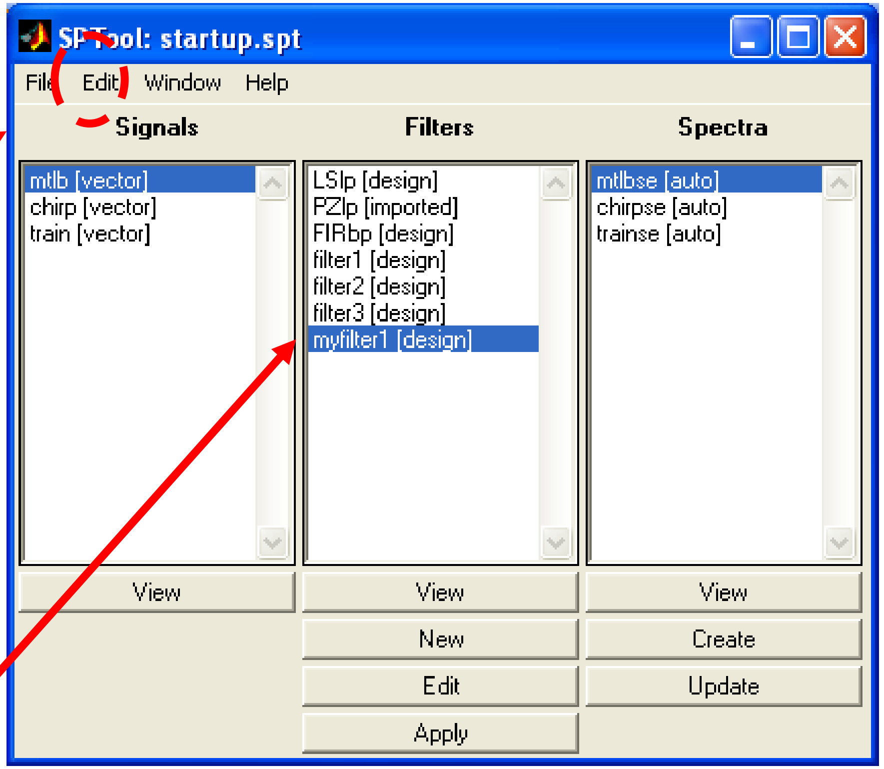


Design the filter based on given specification

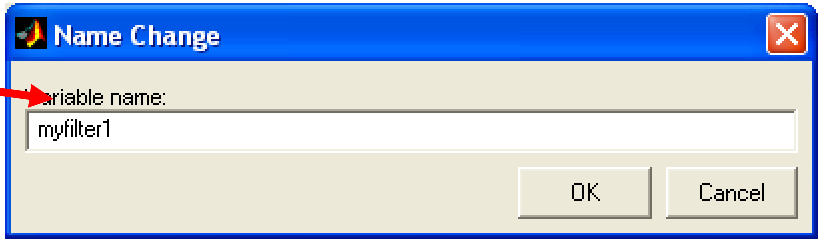
(can use Targets/Generate_C_Header to send to .h files)

Rename for Matlab Use

1. Click “Edit” → “Name” and choose a new name.

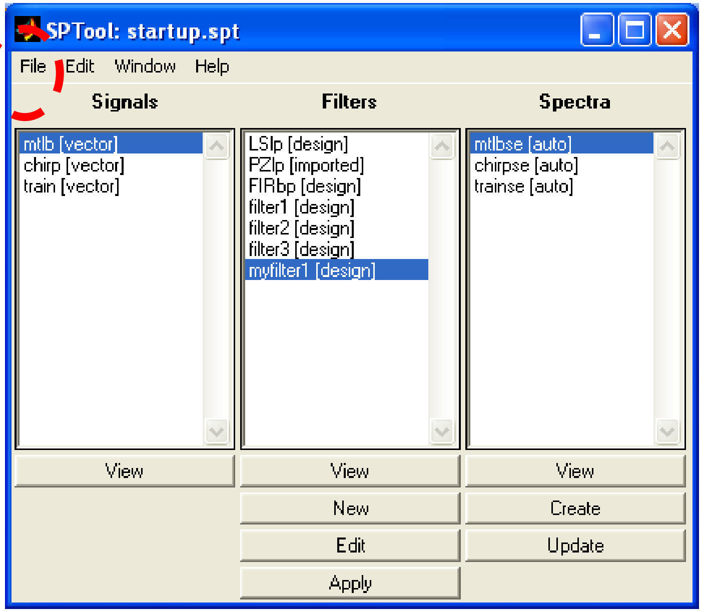


2. Type the new name, for example “myfilter1”

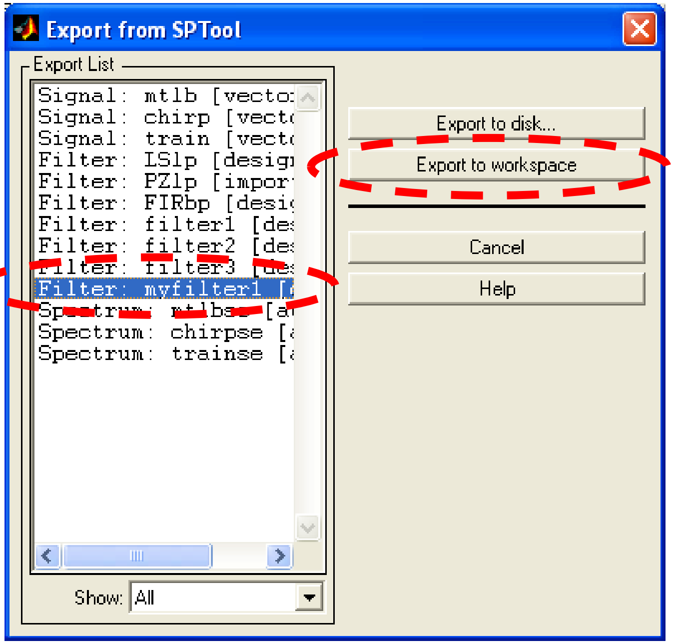


Export to Matlab

1. Click “File” → “Export”

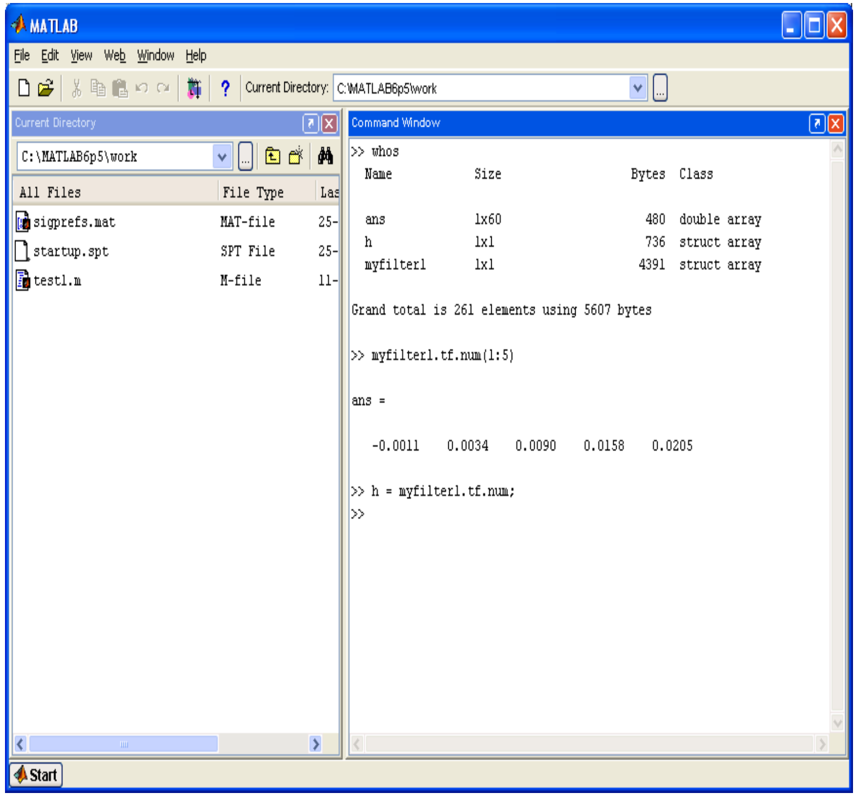


2. Choose “myfilter1[design]”



3. Export to workspace

Save Filter Coefficients




```
>> whos
  Name          Size                Bytes  Class

  ans           1x60                  480  double array
  h             1x1                   736  struct array
  myfilter1     1x1                  4391  struct array

Grand total is 261 elements using 5607 bytes

>> myfilter1.tf.num(1:5)

ans =

   -0.0011    0.0034    0.0090    0.0158    0.0205

>> h = myfilter1.tf.num;
```

- Use “ whos” to see variables
- myfilter1 is the structure array class.
- The coefficients are in myfilter1.tf.num
- Finally, h variable contains our designed coefficients.

FIR Filtering: Convolution

filter length = N ➔ delayed buffer = N (or 2N in case we store in bytes)

$$y[n] = \sum_{i=0}^{N-1} h[i]x[n-i]$$

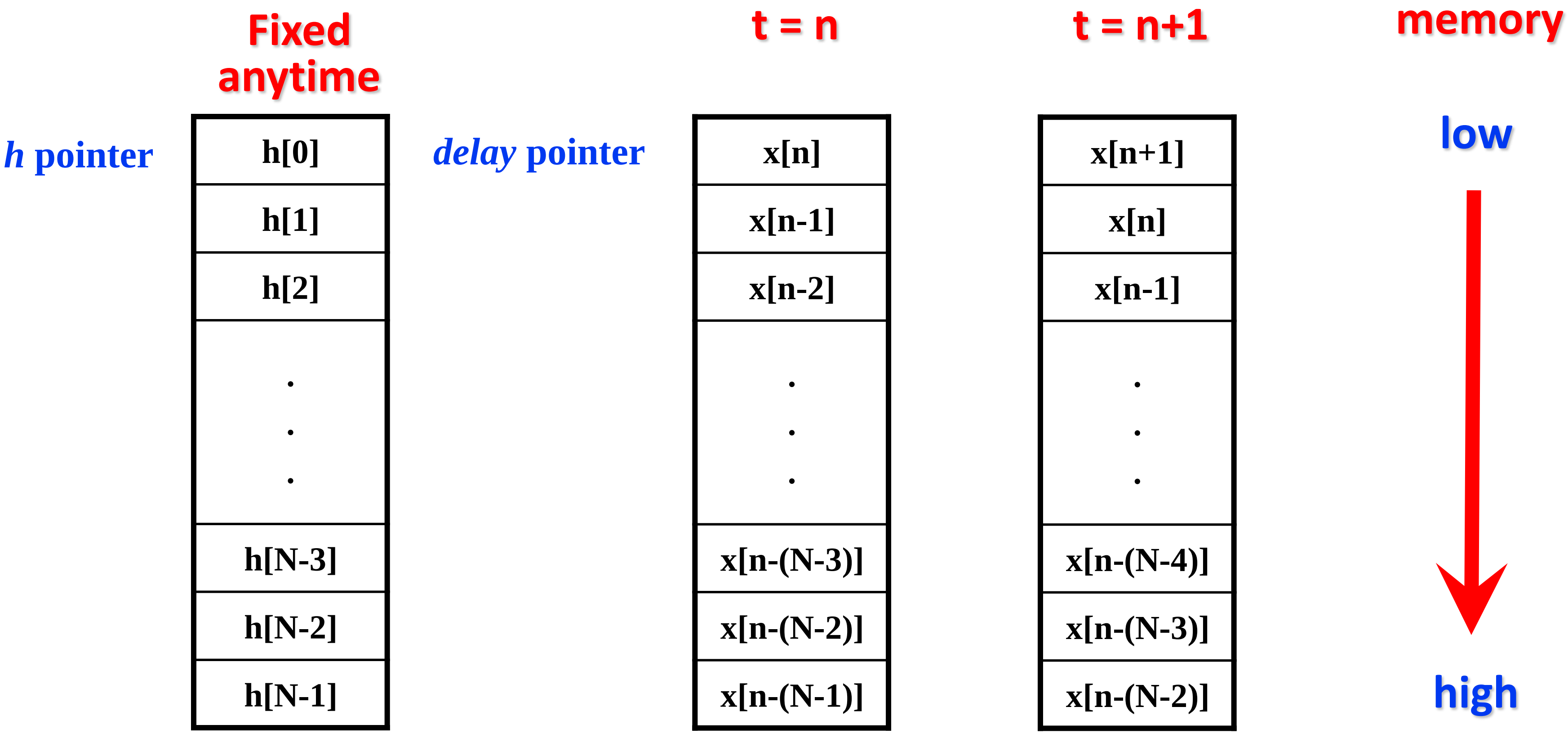
at time n

$$y[n] = h[0]x[n] + h[1]x[n-1] + \cdots + h[N-2]x[n-(N-2)] + h[N-1]x[n-(N-1)]$$

at time n+1

$$y[n+1] = h[0]x[n+1] + h[1]x[n] + \cdots + h[N-2]x[n-(N-3)] + h[N-1]x[n-(N-2)]$$

Intuitive Memory Organization



Only update the delay buffer. The newest sample will be at the lowest memory after shifting and deleting the old samples.

Software Implementation in C Code: Fixed Point

(can be Floating Point too)

The *bs2700.cof* contains the filter coefficients.

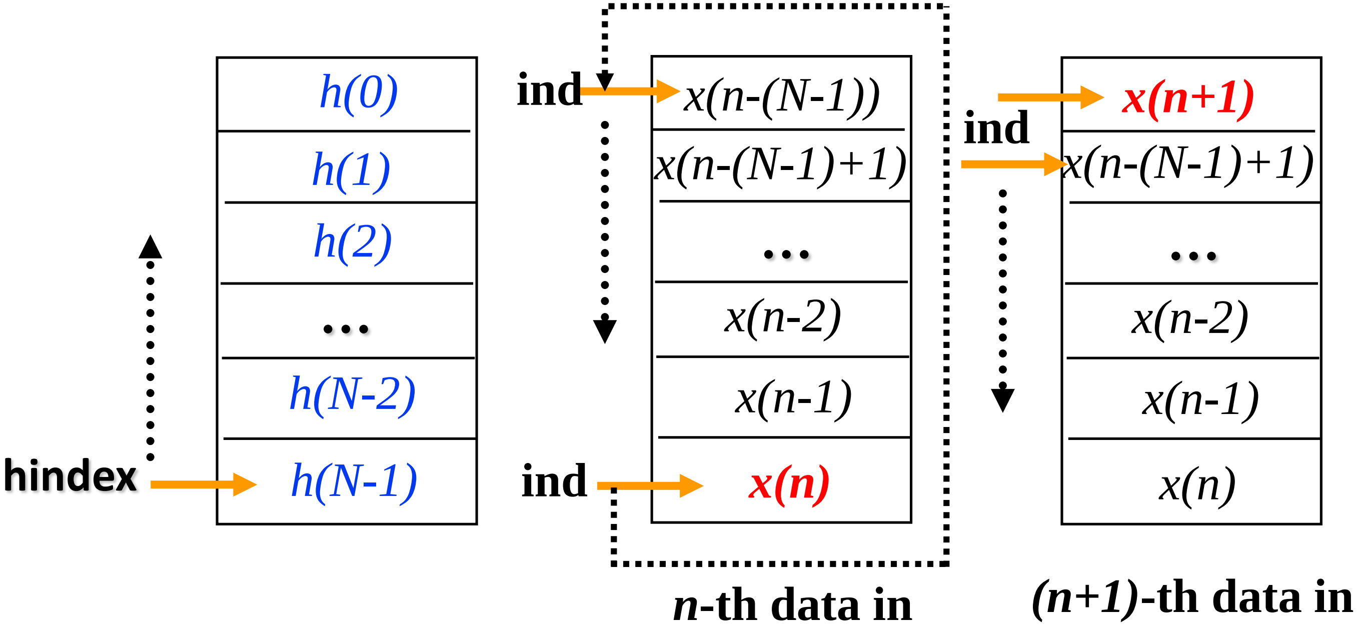
```
//Fir.c  FIR filter. Include coefficient file with length N

#include "bs2700.cof"                    //coefficient file BS @ 2700Hz
int yn = 0;                              //initialize filter's output
short dly[N];              //delay samples

interrupt void c_int11()          //ISR
{
        short i;

        dly[0] = input_sample();  //new input @ beginning of buffer "x[n]"
        yn = 0;                  //initialize filter's output
        for (i = 0; i< N; i++)
          yn += (h[i] * dly[i]);  //y(n) += h(i)* x(n-i)
        for (i = N-1; i > 0; i--)  //starting @ end of buffer
          dly[i] = dly[i-1];      //update delays with data move

        output_sample(yn >> 15);  //scale output (shift right 15 bits)
        return;
}

void main()
  {
        comm_intr();            //init DSK, codec, McBSP
        while(1);             //infinite loop
  }
```

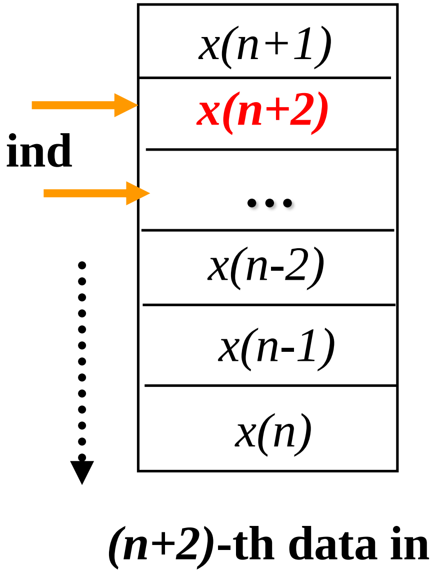
```
float indly[N], outdly[N];
short i;
float yn;                    //filter output
interrupt void c_int11()               //ISR
{
 indly[0]=(float)(input_sample()); //
 yn = 0.0;                   //
…
```

Circular Buffering

Don't want to move data all the time

(same pointer for data input and SOP calculaton)





$$y(n)=\sum_{i=0}^{N-1}h(i)x(n-i)=h(0)x(n)+h(1)x(n-1)$$
$$+h(2)x(n-2)+\ldots+h((N-1)-1)x(n-(N-1)+1)$$
$$+h(N-1)x(n-(N-1))$$
$$y(n+1)=\sum_{i=0}^{N-1}h(i)x(n+1-i)=h(0)x(n+1)+h(1)x(n)$$
$$+h(2)x(n-1)+\ldots+h((N-1)-1)x(n-(N-1)+2)$$
$$+h(N-1)x(n-(N-1)+1)$$

```
/*Real time convolution: Do a convolution calculation when a new data coming in.*/
void convolve(int dataindex, float hfunction[]){
    int index;
    int count;
    int convResult = 0;
    index = dataindex;
    for(count = 0; count < BUFFERSIZE; count++){
            convResult += hfunction[(calBuffersize - count)] * leftChannelData[index % BUFFERSIZE];
            index++;
    }
    convResultBuffer[convIndex] = convResult;
    convIndex = (convIndex + 1) % (CONVBUFFSIZE);
}
 %% count→ hindex, index → ind
```

This function is called inside each “ready” ISR. Once every new sample is arrived, we need to do a convolution and give us a new convolution result. The size of your h(n) functions affect the running time of the convolution. In our package we set the h(n) function size to 32 which is the maximum size to work on 48k sampling rate (32k sampling rate will have the perfect performance in both left and right channel). Sorry for this limitation, if we make a larger h(n) function size the calculation will not be done when a new data is arrived.

Main Program Calling Convolve Function

```
static void handle_leftready_interrupt_test(void* context, alt_u32 id) {
        volatile int* leftreadyptr = (volatile int *)context;
        *leftreadyptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(LEFTREADY_BASE);
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(LEFTREADY_BASE, 0);

    /*******Read, playback, store data*******/
        leftChannel = IORD_ALTERA_AVALON_PIO_DATA(LEFTDATA_BASE);
        IOWR_ALTERA_AVALON_PIO_DATA(LEFTSENDDATA_BASE, leftChannel);
        convolve(leftCount, h);
        leftChannelData[leftCount] = leftChannel;
    leftCount = (leftCount+1) % BUFFERSIZE;
}
```

IIR Realization Structures

- Direct Form II canonic realization:

$$H(z) = H_1(z)H_2(z) = \frac{1}{1+\sum_{k=1}^{M} a_k z^{-k}} \sum_{k=0}^{N} b_k z^{-k}; \quad \text{for N = M}$$

$$= \frac{P(z)}{X(z)} \cdot \frac{Y(z)}{P(z)}$$

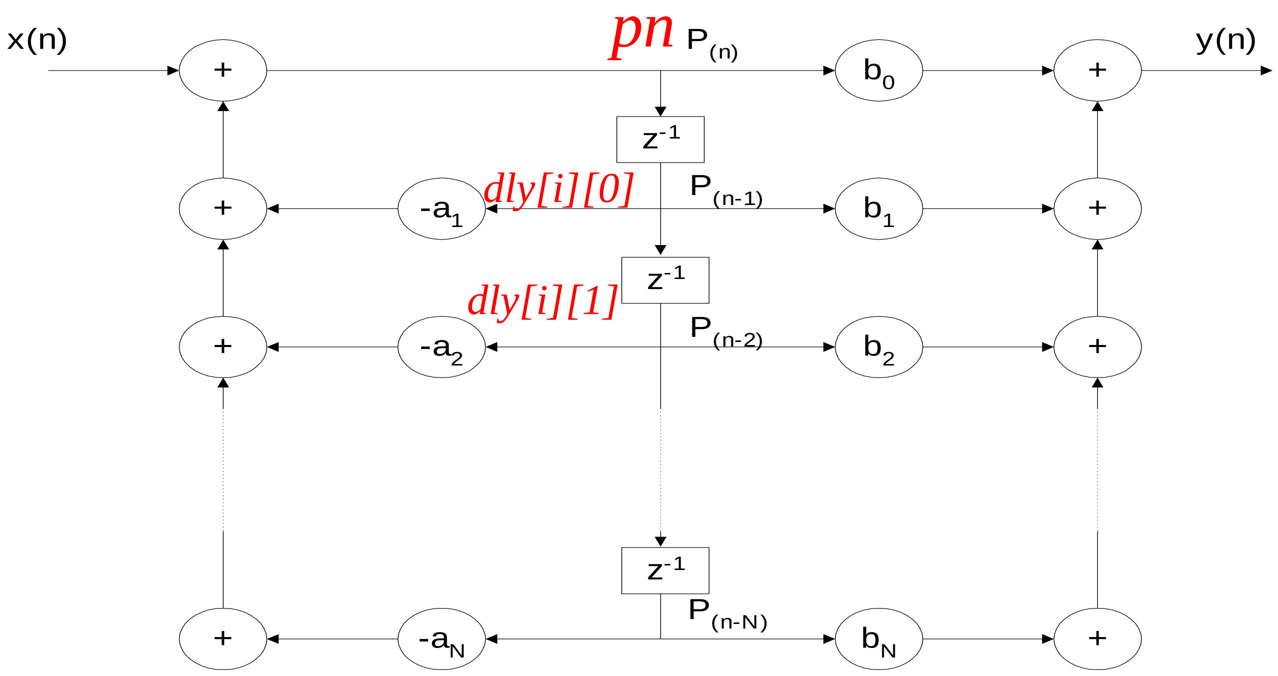
- where: $\frac{P(z)}{X(z)} = \frac{1}{1+\sum_{k=1}^{M} a_k z^{-k}}$ and $\frac{Y(z)}{P(z)} = \sum_{k=0}^{N} b_k z^{-k}$

- Taking the inverse of the z-transform of P(z) and Y(z) leads to:

$$p(n) = x(n) - \sum_{k=1}^{N} a_k p(n-k) \qquad y(n) = \sum_{k=0}^{N} b_k p(n-k)$$

Realization Structures: the Minimum Delay

- Direct Form II canonic realization:



$$\text{A Biquad}: \quad H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

$$p(n) = x(n) - a_1 p(n-1) - a_2 p(n-2)$$

$$y(n) = b_0 p(n) + b_1 p(n-1) + b_2 p(n-2)$$

Direct Form II Implementation

```
//IIR.c  IIR filter using cascaded Direct Form II

#include "bs1750.cof"                    //BS @ 1750 Hz coefficient file
short dly[stages][2] = {0};    //delay samples per stage

interrupt void c_int11()                 //ISR
{
 int i, input;
 int pn, yn;

 input = input_sample(); //input to 1st stage input= IORD_ALTERA_AVALON_PIO_DATA(LEFTDATA_BASE);

 for (i = 0; i < stages; i++)  //repeat for each stage
  {
   pn=input-((a[i][0]*dly[i][0])>>15) - ((a[i][1]*dly[i][1])>>15);

   yn=((b[i][0]*pn)>>15)+((b[i][1]*dly[i][0])>>15)+((b[i][2]*dly[i][1])>>15);

   dly[i][1] = dly[i][0];                //update delays
   dly[i][0] = pn;           //update delays
   input = yn;                                  //intermediate output->input to next stage
  }
   output_sample(yn);        //output final result for time n
   return;                                      //return from ISR
}

void main()
{
  comm_intr();               //init DSK, codec, McBSP
  while(1);                                     //infinite loop
}
```

Sine Generation Using IIR (1)

- A sinusoidal function can be represented as, *y(n)=Ay(n-1)+By(n-2)+Cx(n-1),* where *A=2coswT*, *B=-1*, and *C=sinwT*.

Biquad ➔ $$\sin(\omega Tn)u[n] \overset{z}{\longleftrightarrow} \frac{\sin(\omega T)z^{-1}}{1-(2\cos(\omega T))z^{-1}+z^{-2}}$$

$$\begin{aligned} H(z) &= \frac{\sin wT\ z^{-1}}{1-2\cos wT\ z^{-1}+z^{-2}} \\ &= \frac{1}{2j}\left[\frac{1}{1-e^{jwT}z^{-1}} - \frac{1}{1-e^{-jwT}z^{-1}}\right] \\ &= \frac{1}{2j}\left[\sum_{n=0}^{\infty} e^{jwTn} z^{-n} - \sum_{n=0}^{\infty} e^{-jwTn} z^{-n}\right] \\ &= \sum_{n=0}^{\infty}\frac{1}{2j}\left[e^{jwTn} - e^{-jwTn}\right]z^{-n} \\ &= \sum_{n=0}^{\infty}\sin wTn\ z^{-n} = \sum_{n=0}^{\infty} h(n)\ z^{-n} \end{aligned}$$

$$\begin{aligned} h(n) &= \sin(wTn)u(n) \\ &= \sin(2\pi fn/F_s)u(n) \end{aligned}$$

Sine Generation using IIR (2)

$$x(n) = \delta(n) \longrightarrow \boxed{\begin{array}{c} h(n)= \\ \sin wTn \end{array}} \longrightarrow h(n)$$

Alternative implementation

$$\begin{aligned} y(n) &= Ay(n-1) - y(n-2) \\ &= 2\cos(\omega T)\, y(n-1) - y(n-2) \end{aligned}$$

where

$$\begin{aligned} y(-1) &= -\sin(\omega T) \\ y(-2) &= -\sin(2\omega T) \end{aligned}$$

Goertzel Algorithm

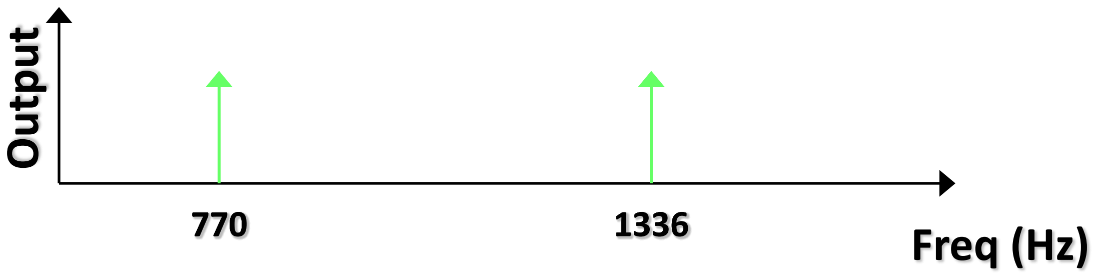
- The Goertzel algorithm is mainly used to detect tones for Dual Tone Multi-Frequency (DTMF) applications.
- DTMF is predominately used for push-button digital telephone sets which are an alternative to rotary telephone sets.
- DTMF has now been extended to electronic mail and telephone banking systems in which users select options from a menu by sending DTMF signals from a telephone.

DTMF Signaling

- In a DTMF signaling system a combination of two frequency tones represents a specific digit, character (A, B, C or D) or symbol (* or #).
- Two types of signal processing are involved:
  - Coding or generation.
  - Decoding or detection.
- For coding, two sinusoidal sequences of finite length are added in order to represent a digit, character or symbol as shown in the following example.

DTMF Tone Generation

	1209Hz	1336Hz	1477Hz	1633Hz
697Hz	1	2	3	A
770Hz	4	5	6	B
852Hz	7	8	9	C
941Hz	*	0	#	D



- Example: **Button 5** results in a 770Hz and a 1336Hz tone being generated simultaneously.

DTMF Tone Detection

- Detection of tones can be achieved by using a bank of filters or using the DFT/FFT.
- However, the Goertzel algorithm is more efficient for this application (checking 8 frequency components).
- The Goertzel algorithm is derived from the DFT and exploits the periodicity of the phase factor, *exp(-j*2πk/N),* to reduce the computational complexity associated with the DFT, as the FFT does.

$$X(k) = \sum_{l=0}^{N-1} x(l) e^{-j\frac{2\pi lk}{N}} = \sum_{l=0}^{N-1} x(l) e^{j\frac{2\pi(N-l)k}{N}}$$

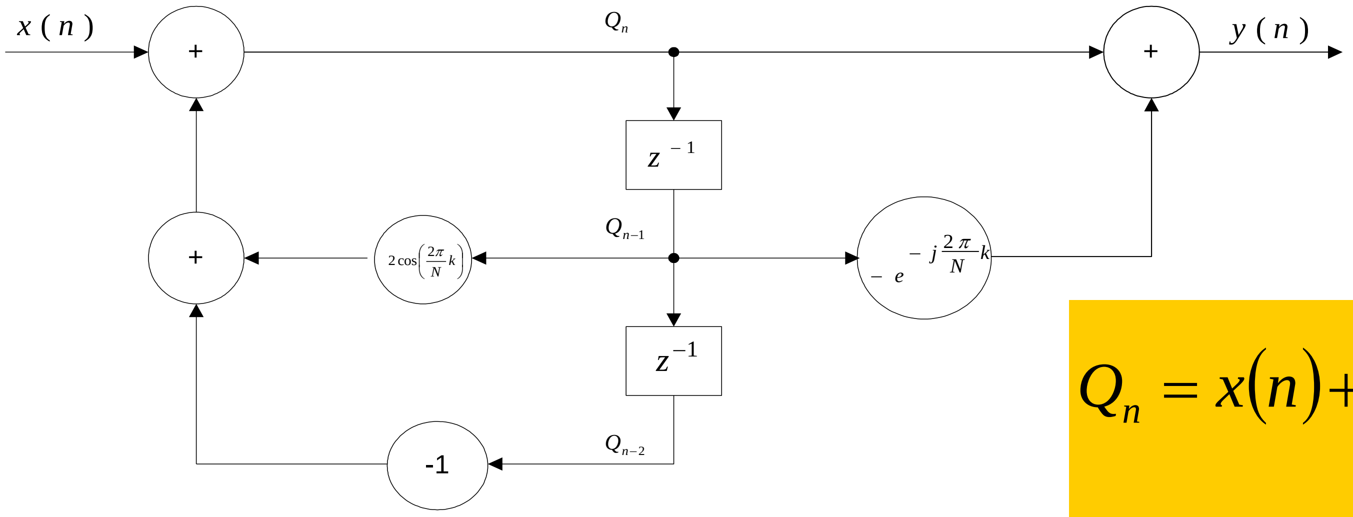
$$\textit{Define}\ \ y_k(n) = \sum_{l=0}^{n} x(l) e^{j\frac{2\pi(n-l)k}{N}} = x(n) * e^{j\frac{2\pi nk}{N}} = x(n) * h_k(n) \quad \text{convolution}$$

$$\text{and the DFT}\ \ X(k) = y_k(n)\big|_{n=N-1}$$

$$H_k(z) = \sum_{n=0}^{\infty} e^{j\frac{2\pi nk}{N}} z^{-n} = \frac{1}{1 - e^{j\frac{2\pi k}{N}} z^{-1}} = \frac{1 - e^{-j\frac{2\pi k}{N}} z^{-1}}{(1 - e^{j\frac{2\pi k}{N}} z^{-1})(1 - e^{-j\frac{2\pi k}{N}} z^{-1})} = \frac{1 - e^{-j\frac{2\pi k}{N}} z^{-1}}{1 - 2\cos(2\pi k / N) z^{-1} + z^{-2}}$$

Goertzel Algorithm Implementation

- To implement the Goertzel algorithm the following equations are required:



$$Q_n = x(n) + 2\cos\left(\frac{2\pi k}{N}\right)Q_{n-1} - Q_{n-2}$$

$$y_k(n) = Q_n - e^{-j\frac{2\pi k}{N}}Q_{n-1}$$

$$|y_k(N)|^2 = y_k(N)\cdot y_k^*(N)$$

$$= Q^2(N) + Q^2(N-1) - 2\cos\left(\frac{2\pi k}{N}\right)Q(N)Q(N-1)$$

Goertzel Algorithm Implementation

- Finally we need to calculate the constant, *k*.
- The value of this constant determines the tone we are trying to detect and is given by:

$$\boxed{k = N \times \frac{f_{tone}}{f_s}}$$

- where: f_{tone} = frequency of the tone.
  f_s = sampling frequency (8K Hz).
  N is set to 205.

- Now we can calculate the value of the coefficient $2\cos(2*\pi*k/N)$.

Goertzel Algorithm Implementation

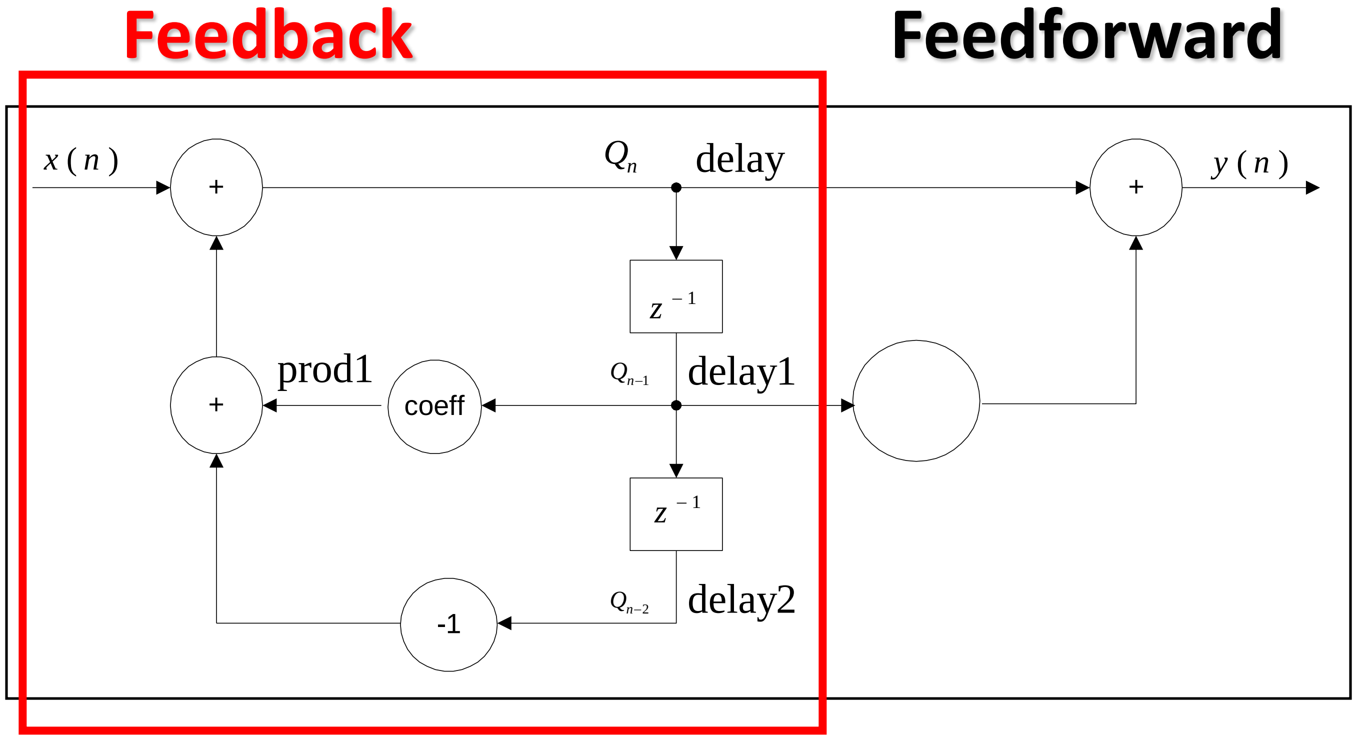
Frequency	k	Coefficient (decimal)
697	18	1.703275
770	20	1.635585
852	22	1.562297
941	24	1.482867
1209	31	1.163138
1336	34	1.008835
1477	38	0.790074
1633	42	0.559454

$2\cos(2*\pi*k/N)$.

$N = 205$

$fs = 8kHz$

Goertzel Implementation



$$Q_n \quad = x(n) - Q_{n-2} + coeff * Q_{n-1}; \quad 0 \leq n < N$$

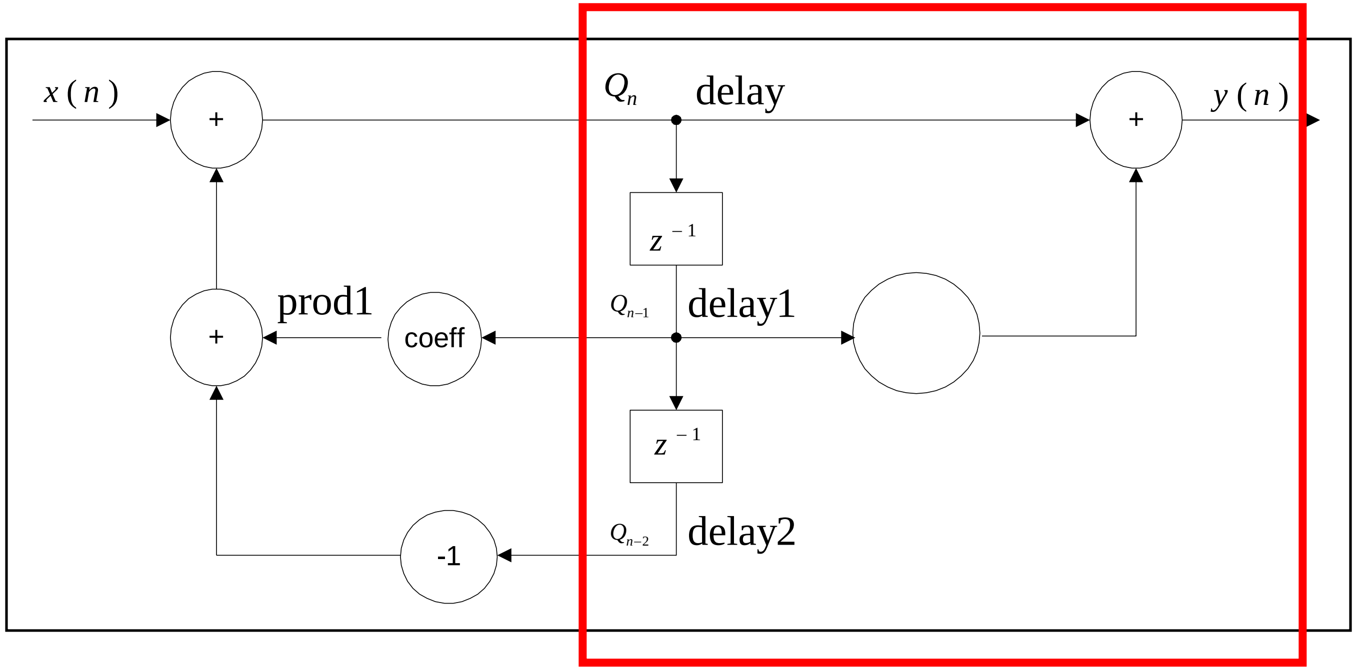
$$= \quad sum1 + prod1$$

where: $coeff = 2\cos(2\pi k/N)$

- The feedback section has to be repeated N times (N=205).

Goertzel Implementation

Feedback **Feedforward**



- Only interested in detecting the presence of a tone and not the phase → the square of the magnitude:

$$|Y_k(N)|^2 = Q^2(N) + Q^2(N\text{-}1) - coeff*Q(N)*Q(N\text{-}1)$$

where: $coeff = 2*\cos(2*\pi*k/N)$

‘C’ code

```c
void Goertzel (void)
{
      static short delay;
      static short delay_1 = 0;
      static short delay_2 = 0;
      static int N = 0;
      static int Goertzel_Value = 0;
      int I, prod1, prod2, prod3, sum, R_in, output;
      short input;
      short coef_1 = 0x4A70;                  // For detecting 1209 Hz

      R_in = input_sample();                   // Read the signal in

      input = (short) R_in;
      input = input >> 4;                      // Scale down input to prevent overflow

      prod1 = (delay_1*coef_1)>>14;
      delay = input + (short)prod1 - delay_2;
      delay_2 = delay_1;
      delay_1 = delay;
      N++;

      if (N==206)
      {
            prod1 = (delay_1 * delay_1);
            prod2 = (delay_2 * delay_2);
            prod3 = (delay_1 *  coef_1)>>14;
            prod3 = prod3 * delay_2;
            Goertzel_Value = (prod1 + prod2 - prod3) >> 15;
            Goertzel_Value <<= 4;              // Scale up value for sensitivity
            N = 0;
            delay_1 = delay_2 = 0;
      }

      output = (((short) R_in) * ((short)Goertzel_Value)) >> 15;

      output_sample(output);          // Send the signal out

   return;}
```

From DFT to FFT

$$X(k) = \sum_{n=0}^{N-1} x[n] W_N^{nk}$$

x[n] = input

X[k] = frequency bins

W = twiddle factors

$X(0) = x[0]W_N^{0} + x[1]W_N^{0*1} + \ldots + x[N-1]W_N^{0*(N-1)}$

$X(1) = x[0]W_N^{0} + x[1]W_N^{1*1} + \ldots + x[N-1]W_N^{1*(N-1)}$

:

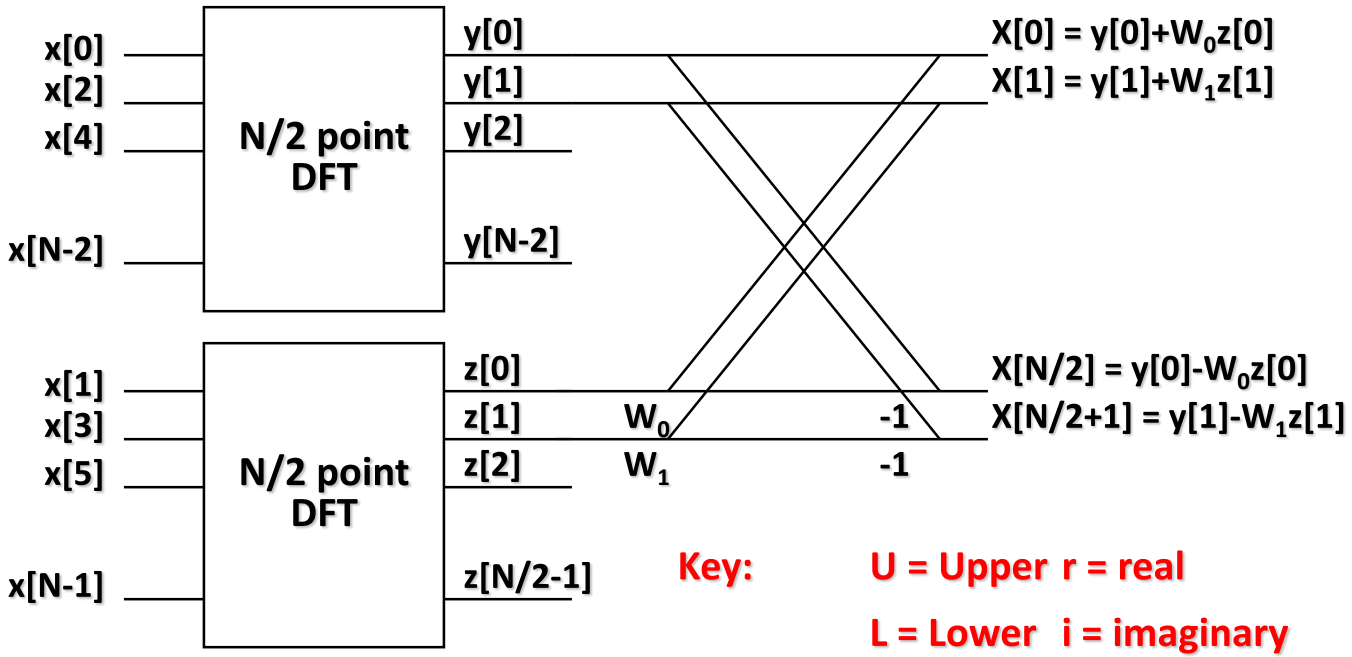
$X(k) = x[0]W_N^{0} + x[1]W_N^{k*1} + \ldots + x[N-1]W_N^{k*(N-1)}$

:

$X(N-1) = x[0]W_N^{0} + x[1]W_N^{(N-1)*1} + \ldots + x[N-1]W_N^{(N-1)(N-1)}$

Note: For N samples of x we have N frequencies representing the signal, A large amount of work has been devoted to reducing the computation time of a DFT → FFT.

Decimation-in-Time FFT



Key: U = Upper r = real

L = Lower i = imaginary

$$(L_r + jL_i)(W_r + jW_i) = L_rW_r + jL_rW_i + jL_iW_r - L_iW_i$$

$$\begin{aligned} U' &= [(L_rW_r - L_iW_i) + j(L_rW_i + L_iW_r)] + [U_r + jU_i] \\ &= (L_rW_r - L_iW_i + U_r) + j(L_rW_i + L_iW_r + U_i) \end{aligned}$$

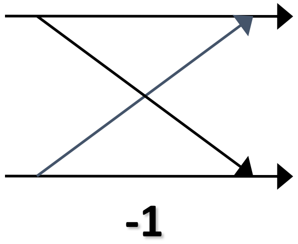
$$\begin{aligned} L' &= (U_r + jU_i) - [(L_rW_r - L_iW_i) + j(L_rW_i + L_iW_r)] \\ &= (U_r - L_rW_r + L_iW_i) + j(U_i - L_rW_i - L_iW_r) \end{aligned}$$

FFT Butterfly Calculations

- **Calculation of the output of a 'butterfly':**

$U_r + jU_i$

$(L_r + jL_i)(W_r + jW_i)$



$$U' = U_r' + jU_i' = (L_r W_r - L_i W_i + U_r) + j(L_r W_i + L_i W_r + U_i)$$

$$L' = L_r' + jL_i' = (U_r - L_r W_r + L_i W_i) + j(U_i - L_r W_i - L_i W_r)$$

- **To further minimize the number of operations (* and +), the following are calculated only once:**

$temp1 = L_r W_r$	$temp2 = L_i W_i$	$temp3 = L_r W_i$	$temp4 = L_i W_r$
$temp1\_2 = temp1 - temp2$		$temp3\_4 = temp3 + temp4$	

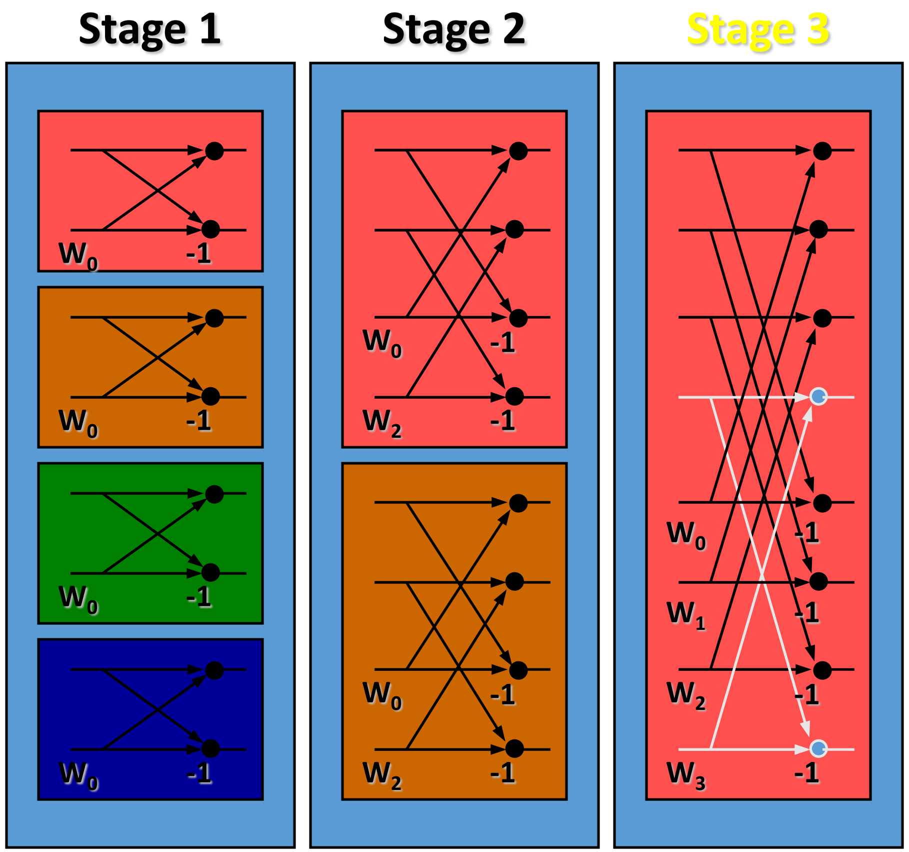
U_r'	$= temp1 - temp2 + U_r$	$= temp1\_2 + U_r$
U_i'	$= temp3 + temp4 + U_i$	$= temp3\_4 + U_i$
L_r'	$= U_r - temp1 + temp2$	$= U_r - temp1\_2$
L_i'	$= U_i - temp3 - temp4$	$= U_i - temp3\_4$

FFT Butterfly Calculations

- **Converting the butterfly calculation into ‘C’ code:**

```
temp1 = (y[lower].real * WR);
temp2 = (y[lower].imag * WI);
temp3 = (y[lower].real * WI);
temp4 = (y[lower].imag * WR);

temp1_2 = temp1 - temp2;
temp3_4 = temp 3 + temp4;

y[upper].real = temp1_2 + y[upper].real;
y[upper].imag = temp3_4 + y[upper].imag;
y[lower].imag = y[upper].imag - temp3_4;
y[lower].real = y[upper].real - temp1_2;
```

FFT Implementation



Example: 8 point FFT

(1) Number of stages:
  – $N_{stages} = 3$

(2) Blocks/stage:
  – Stage 1: $N_{blocks} = 4$
  – Stage 2: $N_{blocks} = 2$
  – Stage 3: $N_{blocks} = 1$

(3) B'flies/block:
  – Stage 1: $N_{btf} = 1$
  – Stage 2: $N_{btf} = 2$
  – Stage 3: $N_{btf} = 4$

- Decimation in time FFT:
  - Number of stages = $\log_2 N$
  - Number of blocks/stage = $N/2^{stage}$
  - Number of butterflies/block = $2^{stage-1}$

Decimation in Frequency (DIF) FFT

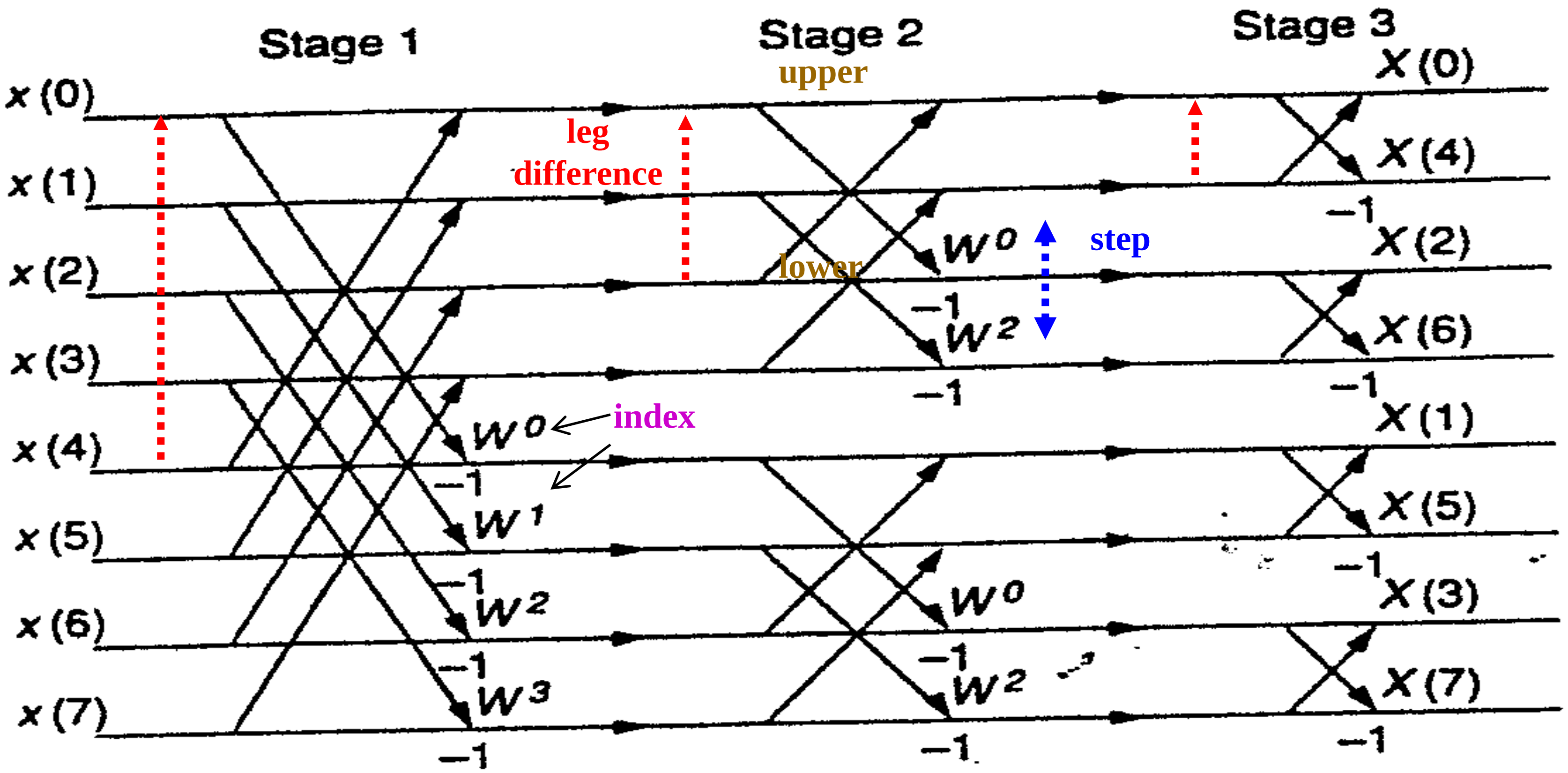


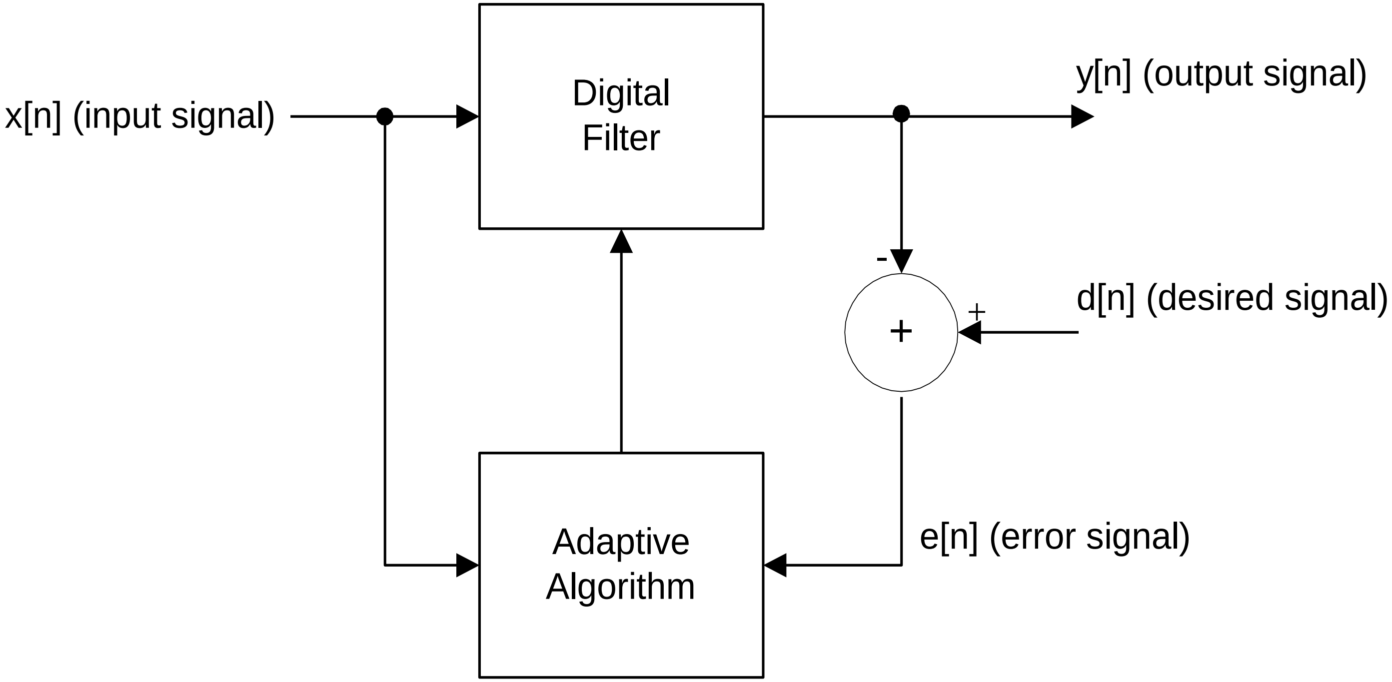
FIGURE 6.5 Eight-point FFT flow graph using decimation-in-frequency.

DIF FFT

```
i = 1;                    //log(base2) of N points= # of stages
do  {
  num_stages +=1; initial value = 0
  i = i*2;
 }while (i!=N); how many stages?
leg_diff = N/2;     //difference between upper&lower legs
step = 1;   (initial value) //step between values in twiddle.h
for (i = 0;i < num_stages; i++)  //for N-point FFT
 {
  index = 0;
  for (j = 0; j < leg_diff; j++)  // how many butterflies
   {
    for (upper_leg = j; upper_leg < N; upper_leg += (2*leg_diff))
     {
      lower_leg = upper_leg+leg_diff;
              temp1.real = (Y[upper_leg]).real + (Y[lower_leg]).real;
      temp1.imag = (Y[upper_leg]).imag + (Y[lower_leg]).imag;
      temp2.real = (Y[upper_leg]).real - (Y[lower_leg]).real;
      temp2.imag = (Y[upper_leg]).imag - (Y[lower_leg]).imag;
      (Y[lower_leg]).real = temp2.real*(w[index]).real -
temp2.imag*(w[index]).imag;
      (Y[lower_leg]).imag = temp2.real*(w[index]).imag
+temp2.imag*(w[index]).real;
      (Y[upper_leg]).real = temp1.real;
      (Y[upper_leg]).imag = temp1.imag;     }
index += step;
  }
  leg_diff = leg_diff/2;
  step *= 2;
  }
   j = 0;
   for (i = 1; i < (N-1); i++)
       //bit reversal for resequencing data
  {    k = N/2;
  while (k <= j)
  {     j = j - k;
   k = k/2;
  }
  j = j + k;
  if (i<j)
  {
   temp1.real = (Y[j]).real;
   temp1.imag = (Y[j]).imag;
   (Y[j]).real = (Y[i]).real;
   (Y[j]).imag = (Y[i]).imag;
   (Y[i]).real = temp1.real;
   (Y[i]).imag = temp1.imag;
  }   }
return;}
```

$(N=8)$, 04261537, $1 \leftrightarrow 4$, $3 \leftrightarrow 6$

A Typical Adaptive Filter Structure



FIR Adaptive Filter

- **Adaptive filters differ from other filters such as FIR and IIR in the sense that:**
  - **The coefficients are not determined by a set of desired specifications.**
  - **The coefficients are not fixed.**
- **With adaptive filters the specifications are not known and change with time.**
- **Applications include: process control, medical instrumentation, speech processing, echo and noise calculation and channel equalization.**
- **FIR adaptive filter is the most practical and widely used:**

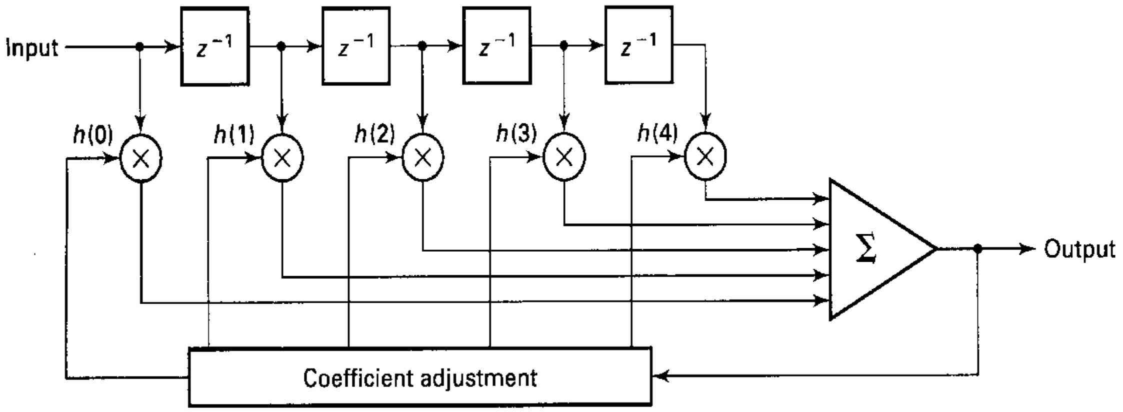


FIGURE 9.1 *Direct form adaptive FIR filter*

The LMS Update Algorithm

- **The basic premise of the LMS algorithm is the use of the instantaneous estimates of the gradient in the steepest descent algorithm:**

$$h_n(k) = h_{n-1}(k) + \beta\Delta_{n,k}.$$

β = step size parameter

$\Delta_{n,k}$ = gradient vector that makes h(n) approach the optimal value h_{opt}

- **It has been shown that (Widrow and Stearns, 1985):**

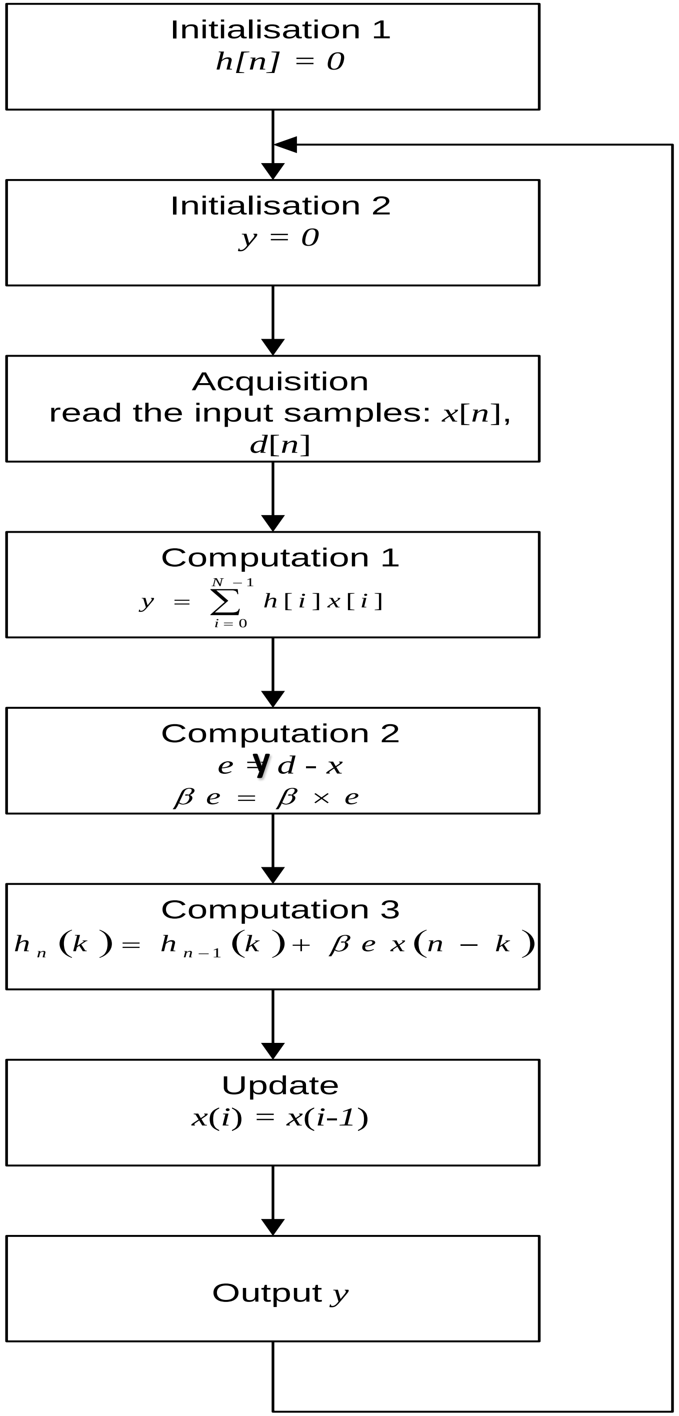
$$\Delta_{n,k} = e(n)x(n-k).$$

e(n) is the error signal, where: e(n) = d(n) - y(n)

- **Finally:**

$$h_n(k) = h_{n-1}(k) + \beta e(n)x(n-k).$$

LMS Algorithm Implementation



LMS algorithm Implementation

(NS: sample #, N: filter order)

```
float desired[NS], Y_out[NS], error[NS];

void main()
{
  long I, T;
  float D, Y, E;
  float W[N+1] = {0.0};
  float X[N+1] = {0.0};

  for (T = 0; T < NS; T++)            //start adaptive algorithm
  {
    X[0] = NOISE;                     //new noise sample (functional call)
    D = DESIRED;                      //desired signal (functional call)
    Y = 0;                            //filter'output set to zero
    for (I = 0; I <= N; I++)
      Y += (W[I] * X[I]);             //calculate filter output
    E = D - Y;                        //calculate error signal
    for (I = N; I >= 0; I--)
    {
      W[I] = W[I] + (beta*E*X[I]);    //update filter coefficients
      if (I != 0) X[I] = X[I-1];      //update data sample
    }
    desired[T] = D;
    Y_out[T] = Y;
    error[T] = E;
  }
  printf("done!\n");
}
```