Filter C Programming

(2A) FIR Filter

Please send corrections (or suggestions) to youngwlim@hotmail.com.
This document was produced by using LibreOffice.

Based on

Introduction to Signal Processing

S. J. Ofranidis

The necessities in DSP C Programming

FIR Filter (A.pdf) 20191114

Tapped Delay

Details will be found in

https://en.wikiversity.org/wiki/The_necessities_in_Filter_Theory#Digital_Filter_Realizations

The necessities in Filter Theory

- **Digital Filter Realizations**
  - **Tapped Delay (A.pdf)**

Circular Convolution

Details will be found in

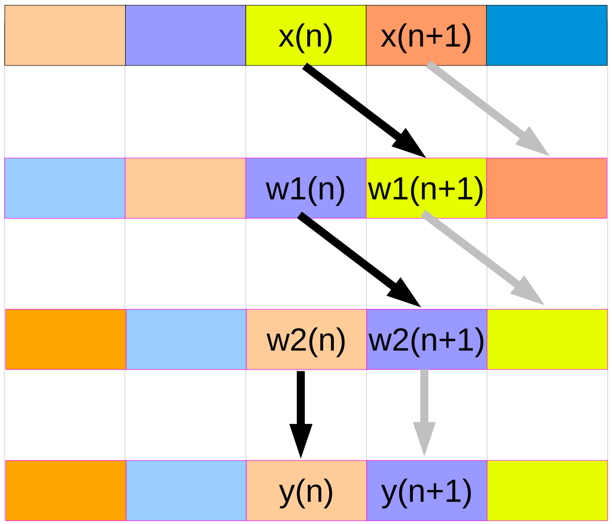
https://en.wikiversity.org/wiki/The_necessities_in_Linear_System_Theory#Time_Domain_System_Analysis_-_Discrete_Time

The necessities in Linear System Theory

- **Time Domain System Analysis - Discrete Time**
  - **Convolution (A.pdf, B.pdf)**

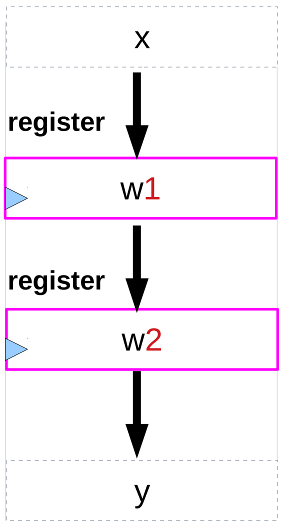
Delay C Model

Timing Chart



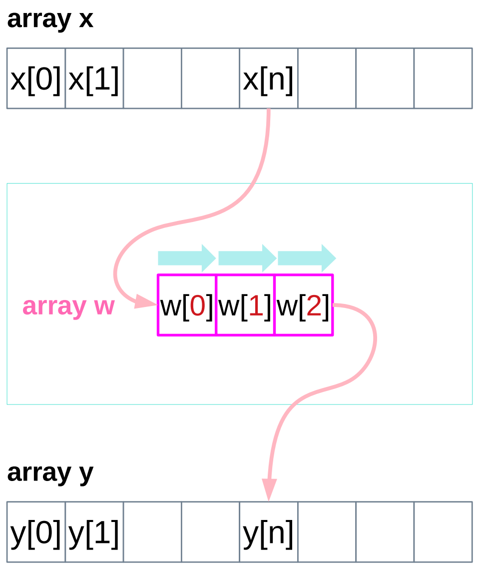
y(n) = w2(n)
w2(n+1) = w1(n)
w1(n+1) = x(n)

Register Transfer



y = w2
w2 = w1
w1 = x

DSP C Model for simulation

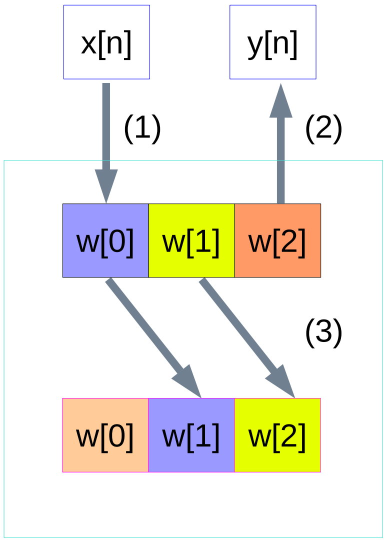


y[n] = w[2]
w[0] = x[n]
w[2] = w[1]
w[1] = w[0]

IO Equations for the Triple Delay

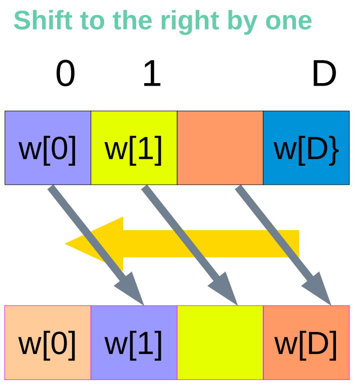
y(n) = w2(n)
w0(n) = x(n)
w2(n+1) = w1(n)
w1(n+1) = w2(n)   D = 2, 1

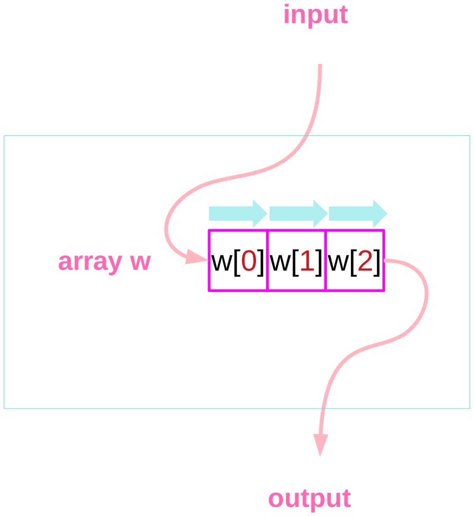
```
y[n] = w[2]        // get the output
w[0] = x[n]        // put the input
w[2] = w[1]        // shift
w[1] = w[0]        // shift
```



delay.c

```
/* delay.c - delay by D time samples */
/* w[0] = input, w[D] = output */

void delay(int D, double *w)
{
    int i;

    for (i=D; i>=1; i--)
        w[i] = w[i-1];

    // reverse-order updating
}
```





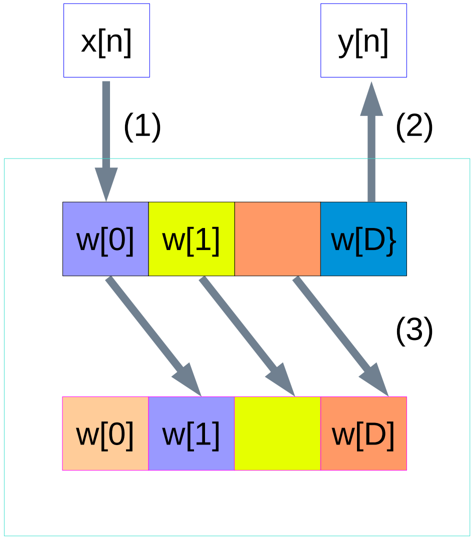
order of execution

w[D] = w[D-1]
… …
w[2] = w[1]
w[1] = w[0]

Using the delay function

```
double *w;
w = (double *) calloc(D+1, sizeof(double));    // (D+1)-dimensional
```

```
for (n = 0; n < Ntot; n++) {
    y[n] = w[D];            // (1) write output
    w[0] = x[n];            // (2) read input
    delay(D, w);            // (3) update delay line
}
```



Delay Functions

$y(n) = w_1(n)$
$w_1(n+1) = x(n)$

$y(n) = w_2(n)$
$w_2(n+1) = w_1(n)$
$w_1(n+1) = x(n)$

$y(n) = w_3(n)$
$w_3(n+1) = w_2(n)$
$w_2(n+1) = w_1(n)$
$w_1(n+1) = x(n)$

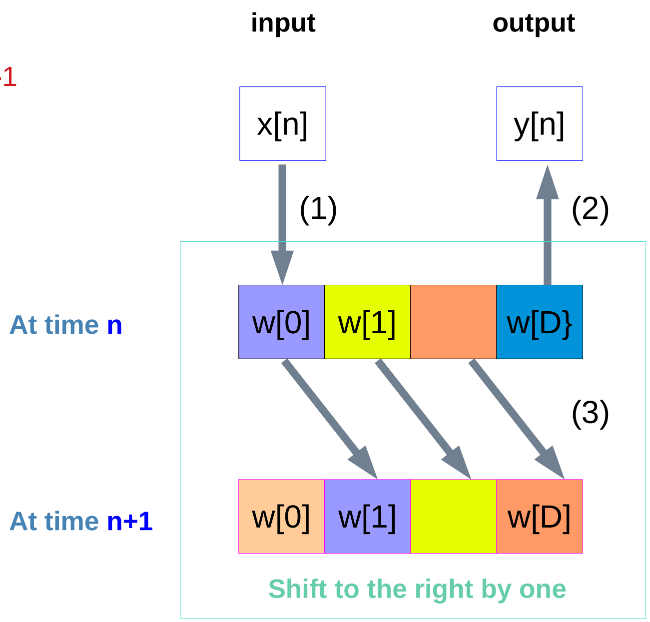
$y(n) = w_D(n)$
$w_0(n) = x(n)$
$w_i(n+1) = w_{i-1}(n),$
$i = D, D-1, \ldots, 2, 1$

time index : n

memory location : W_i
memory index : i

$$w_i(n+1) = w_{i-1}(n)$$

the current value at w_{i-1}
will become
the next value at w_i



Sample Processing Algorithms for Delay Functions

for each input sample x do:
$y := w_1$
$w_1 := x$

for each input sample x do:
$y := w_2$
$w_2 := w_1$
$w_1 := x$

for each input sample x do:
$y := w_3$
$w_3 := w_2$
$w_2 := w_1$
$w_1 := x$

for each input sample x do:
$y := w_D$
$w_0 := x$
for $i = D, D-1, \ldots, 1$ do:
$w_i := w_{i-1}$

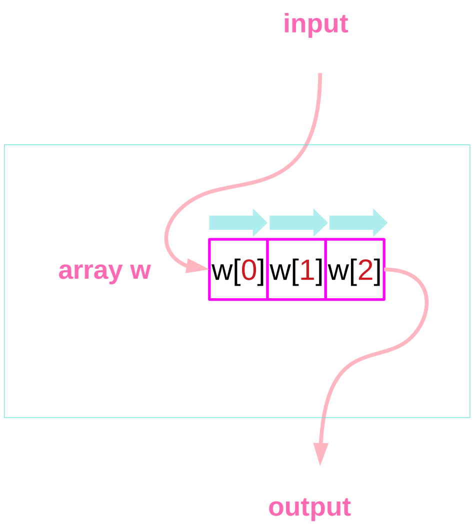
Holding a delayed input sequence

$w_0(n) = x(n)$

$w_1(n) = x(n-1) = w_0(n-1)$

$w_2(n) = x(n-2) = w_1(n-1)$

$w_3(n) = x(n-3) = w_2(n-1)$



FIR filter of order of M

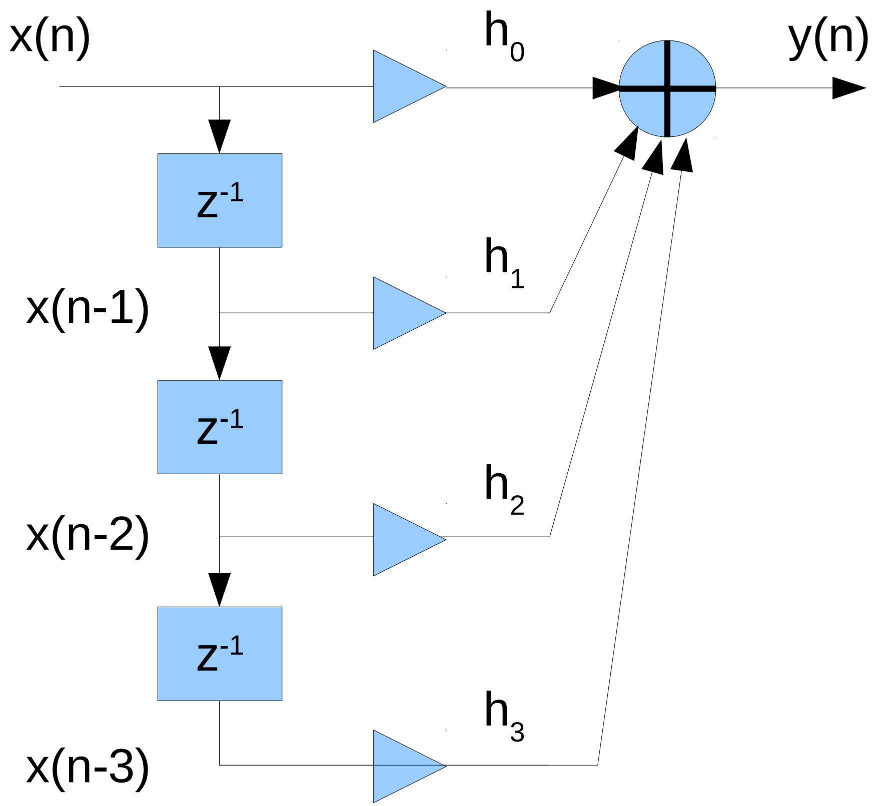
$y(n) = h_0x(n) + h_1x(n\text{-}1) + \ldots + h_M(n\text{-}M)$

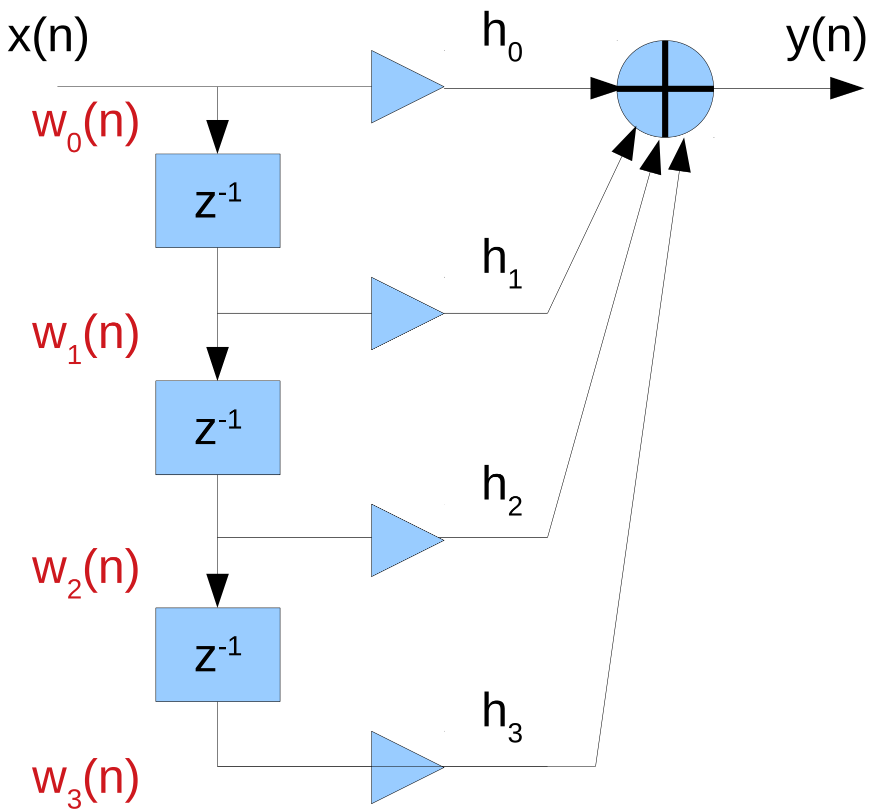
Impulse response

$\mathbf{h} = [h_0, h_1, \ldots , h_M]$

Direct form realization

$y(n) = h_0 x(n) + h_1 x(n-1) + \ldots + h_M x(n-M)$

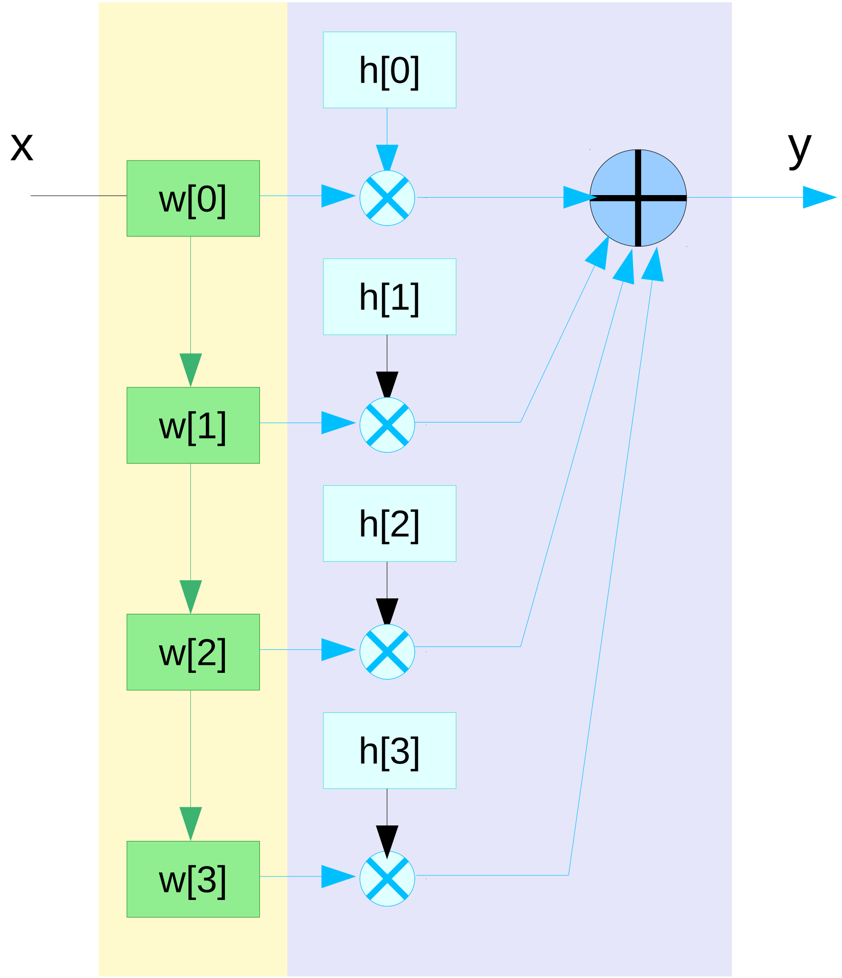


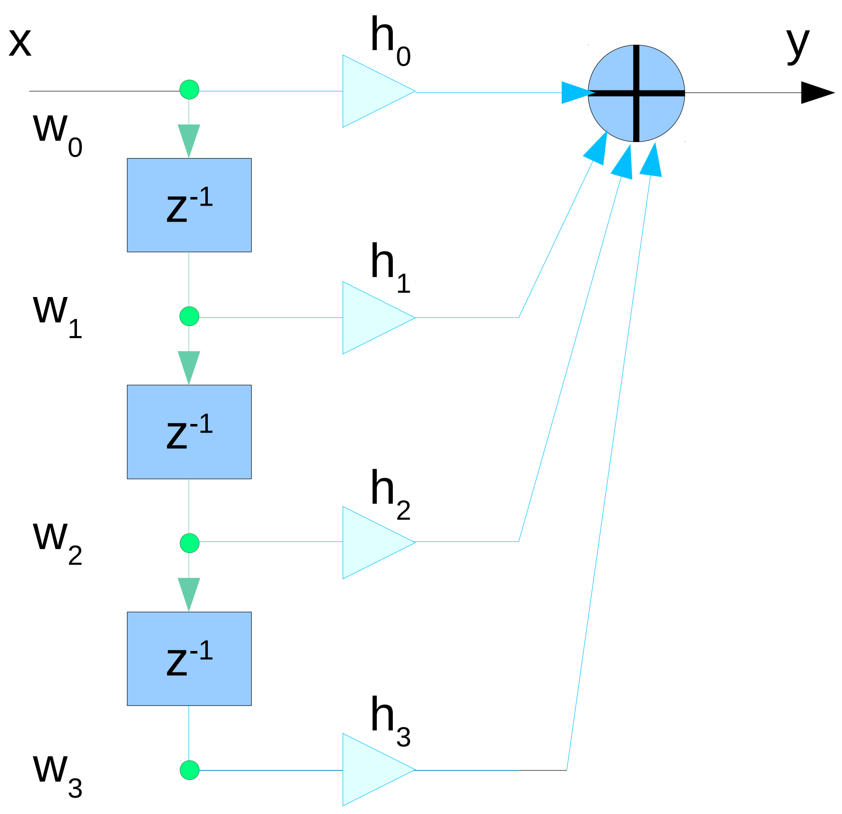


Internal state $w_0(n)$, $w_1(n)$, $w_2(n)$, $w_3(n)$

Block Diagram

$$y(n) = h_0 w_0 + h_1 w_1 + \ldots + h_M w_M$$





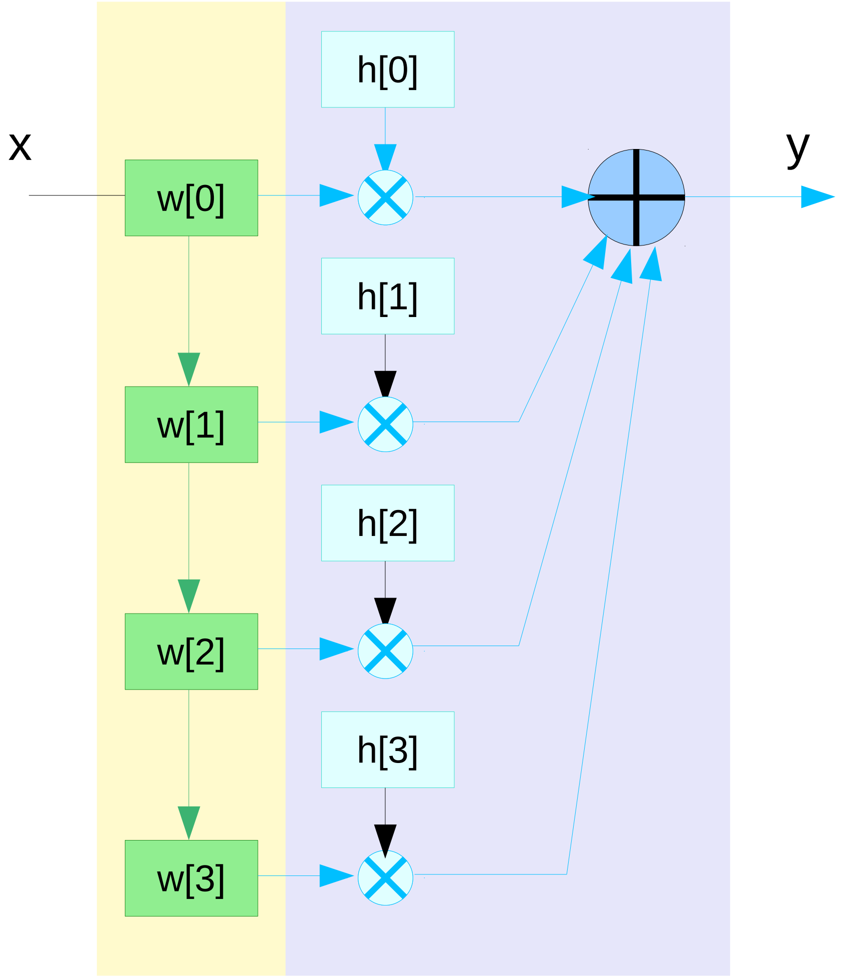
Sample Processing Algorithm

FIR filter equations

$w_0(n) = x(n)$
$y(n) = h_0 w_0(n)+h_1 w_1(n)+h_2 w_2(n)+h_3 w_3(n)$
$w_3(n+1) = w_2(n)$
$w_2(n+1) = w_1(n)$
$w_1(n+1) = w_0(n)$

$w_0(n) = x(n)$
$y(n) = h_0 w_0(n)+h_1 w_1(n)+ \cdot \cdot \cdot + h_M w_M(n)$
$w_i(n+1) = w_{i-1}(n),$
for $i = M, M-1, \ldots, 1$

$y(n) = h_0 w_0 + h_1 w_1 + \ldots + h_M w_M$



Sample Processing Algorithms for FIR filters

for each input sample x do:
$w_0 = x$
$y = h_0w_0 + h_1w_1 + h_2w_2 + h_3w_3$
$w_3 = w_2$
$w_2 = w_1$
$w_1 = w_0$

for each input sample x do:
$w_0 = x$
$y = h_0w_0 + h_1w_1 + \cdots + h_Mw_M$
for $i = M, M-1, \ldots, 1$ do:
$w_i = w_{i-1}$

for each input sample x do:
$w_0 = x$
y = **dot** (M, h , w)
delay (M, w)

fir.c

```
/* fir.c - FIR filter in direct form */
/* Usage: y = fir(M, h, w, x); */
/* M = filter order, h = filter, w = state, x = input sample */
double fir(int M, double *h, double *w, double x)
{
        int i;
        double y;                       /* output sample */

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

        for (y=0, i=0; i<=M; i++)
                y += h[i] * w[i];       /* compute current output sample y */

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

        return y;
}
```

Using **fir**

```
double *h, *w, x, y;
h = (double *) calloc(M+1, sizeof(double));          // (M+1)-dimensional
w = (double *) calloc(M+1, sizeof(double));          // (M+1)-dimensional


FILE *fpx, *fpy;
fpx = fopen("x.dat", "r");          // input file
fpy = fopen("y.dat", "w");          // output file

while (fscanf(fpx, "%lf", &x) != EOF) {          // read x from x.dat
        y = fir(M, h, w, x);                     // process x to get y
        fprintf(fpy, "%lf\n", y);                // write y into y.dat
}

for (i=0; i<M; i++) {
        y = fir(M, h, w, 0.0);          // M-input transients with x=0
        fprintf(fpy, "%lf\n", y);
}
```

dot.c

```
/* dot.c - dot product of two length-(M+1) vectors */
// Usage: y = dot(M, h, w);
// h = filter vector, w = state vector
// M = filter order
// compute dot product

double dot(int M, double *h, double *w)
{
        int i;
        double y;

        for (y=0, i=0; i<=M; i++)
                y += h[i] * w[i];

        return y;
}
```

fir2.c

```
/* fir2.c - FIR filter in direct form */
double dot(int M, double *h, double *w);
void delay(int D, double *w);

// Usage: y = fir2(M, h, w, x);
// M = filter order, h = filter, w = state, x = input
double fir2(int M, double *h, double *w, double *x)
{
    double y;

    w[0] = x;                   // read input

    y = dot(M, h, w);           // compute output

    delay(M, w);                // update states

    return y;
}
```

```
for (y=0, i=0; i<=M; i++)
    y += h[i] * w[i];
```

```
for (i=M; i>=1; i--)
    w[i] = w[i-1];
```

fir3.c

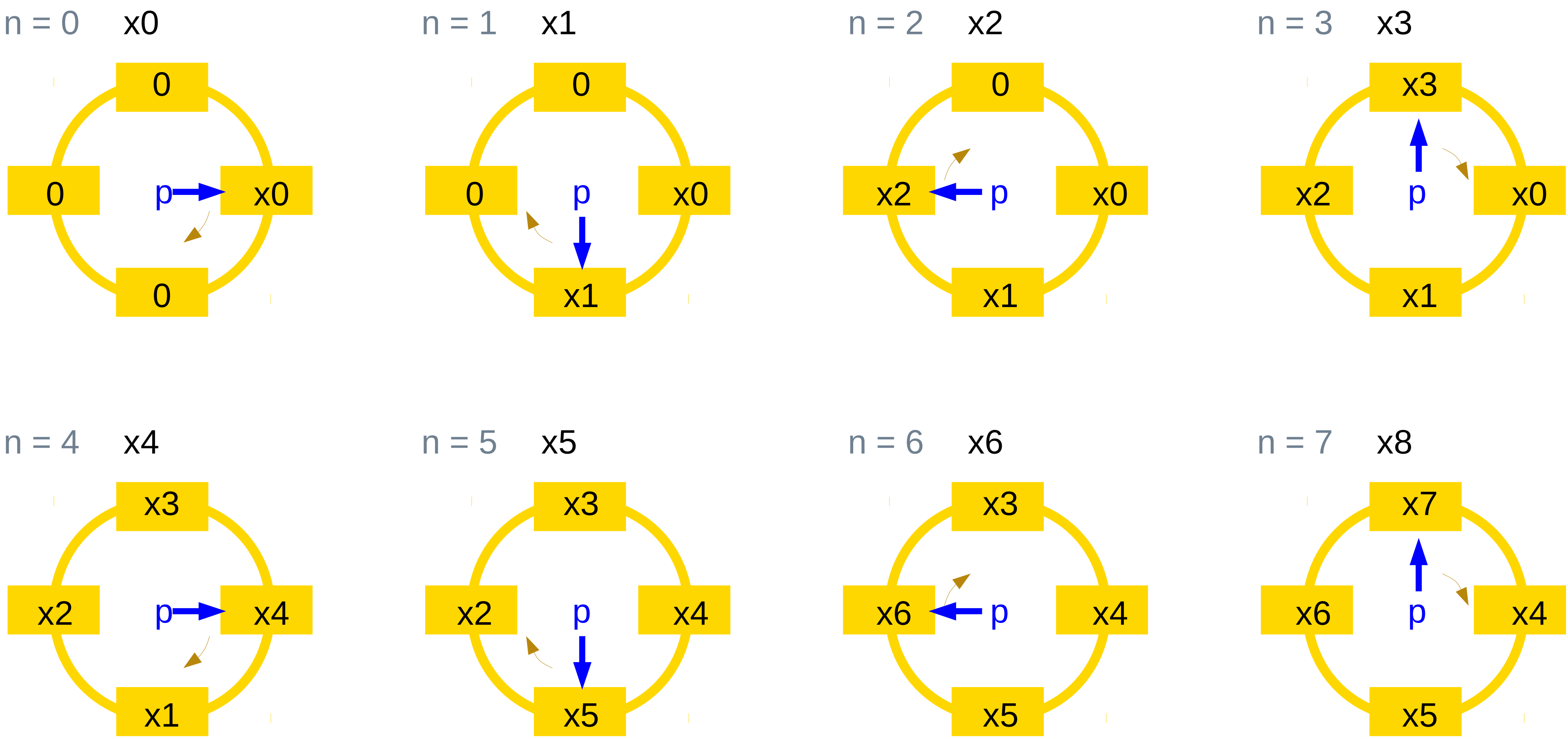
```
/* fir3.c - FIR filter emulating a DSP chip */
double fir3(int M, double *h, double *w, double x)
{
    int i;
    double y;

    w[0] = x;                                   // read input

    for (y=h[M]*w[M], i=M-1; i>=0; i--) {
        w[i+1] = w[i];                          // data shift instruction
        y += h[i] * w[i];                       // MAC instruction
    }
    return y;
}
```

```
for (y=0, i=0; i<=M; i++)
    y += h[i] * w[i];
```

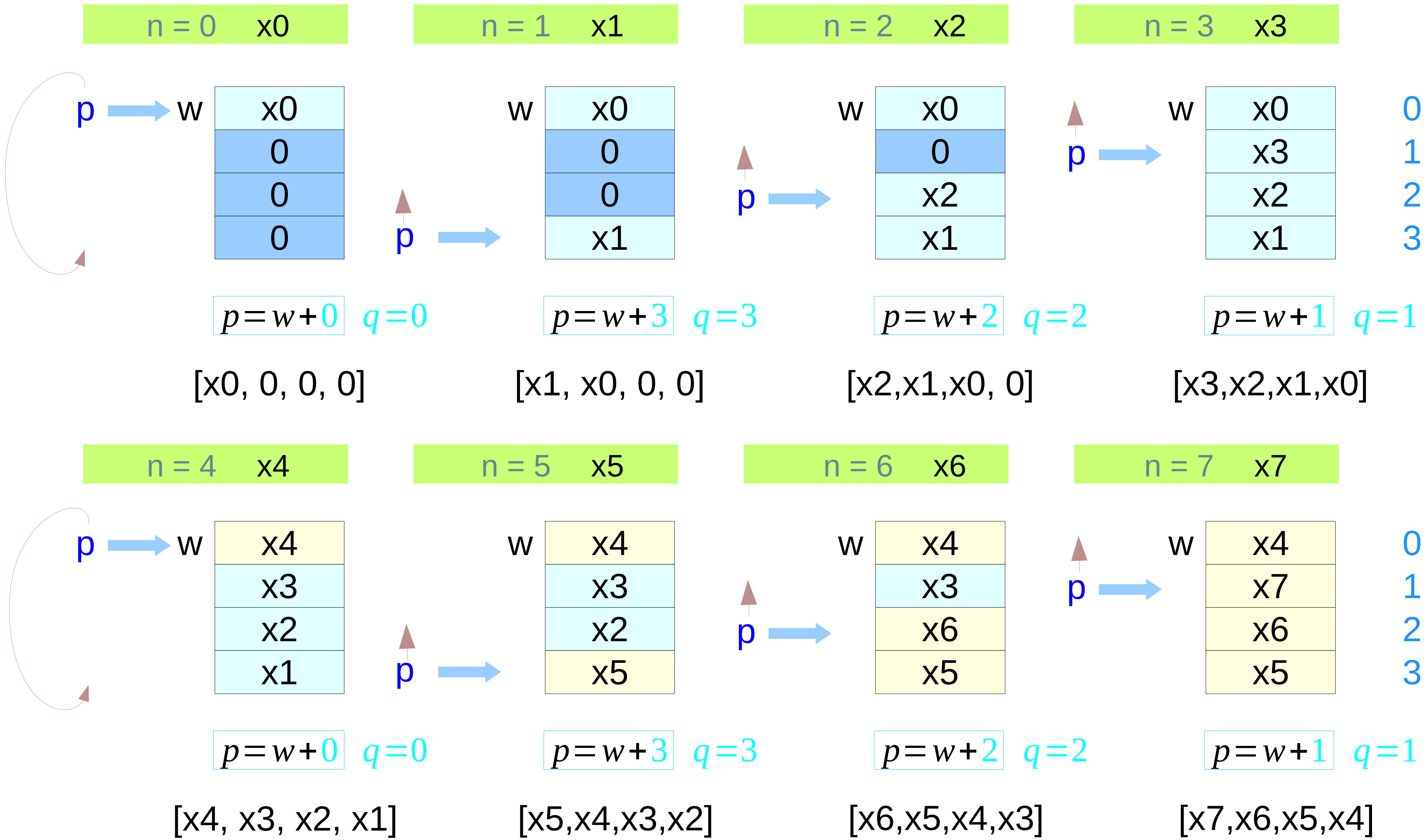
```
for (i=M; i>=1; i--)
    w[i] = w[i-1];
```

```
                          y  =  h[M]*w[M]
w[M]    = w[M-1];         y += h[M-1] * w[M-1];
w[M-1] = w[M-2];          y += h[M-2] * w[M-2];
w[M-2] = w[M-3];          y += h[M-3] * w[M-3];
     …                         …
w[2] = w[1];              y += h[1] * w[1];
w[1] = w[0];              y += h[0] * w[0];
```

Address pointer **p** and incoming inputs



Address pointer p and incoming inputs – linear array view



Internal states over the first 8 steps of n

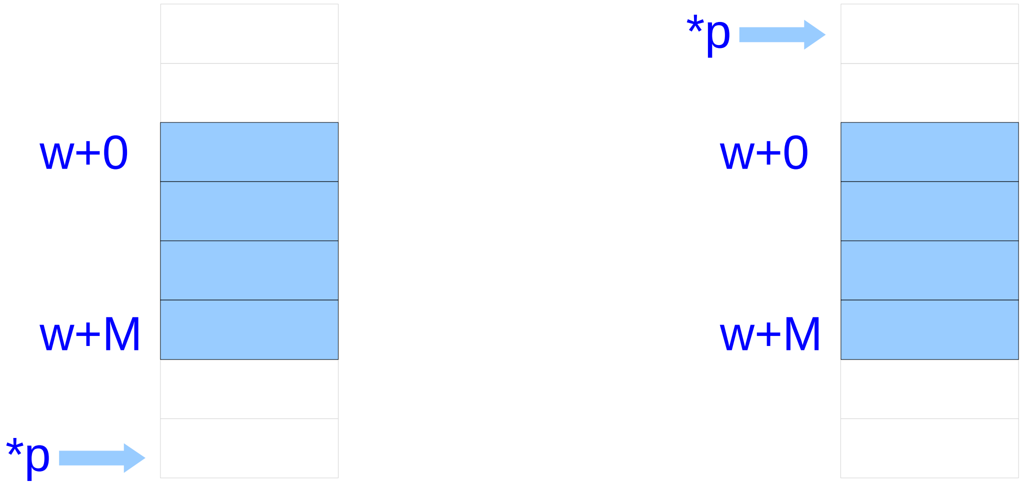
n	q	[w0]	[w1]	[w2]	[w3]	s0	s1	s2	s3	s0	s1	s2	s3
0	0	x0	0	0	0	[w0]	[w1]	[w2]	[w3]	x0	0	0	0
1	3	x0	0	0	x1	[w3]	[w0]	[w1]	[w2]	x1	x0	0	0
2	2	x0	0	x2	x1	[w2]	[w3]	[w0]	[w1]	x2	x1	x0	0
3	1	x0	x3	x2	x1	[w1]	[w2]	[w3]	[w0]	x3	x2	x1	x0
4	0	x4	x3	x2	x1	[w0]	[w1]	[w2]	[w3]	x4	x3	x2	x1
5	3	x4	x3	x2	x5	[w3]	[w0]	[w1]	[w2]	x5	x4	x3	x2
6	2	x4	x3	x6	x5	[w2]	[w3]	[w0]	[w1]	x6	x5	x4	x3
7	1	x4	x7	x6	x5	[w1]	[w2]	[w3]	[w0]	x7	x6	x5	x4

[w0] value at the buffer w0
[w1] value at the buffer w1
[w2] value at the buffer w2
[w3] value at the buffer w3

$$\begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} \quad \begin{pmatrix} p[0] \\ p[1] \\ p[2] \\ p[3] \end{pmatrix} \quad \begin{pmatrix} w[2] \\ w[3] \\ w[0] \\ w[1] \end{pmatrix}$$

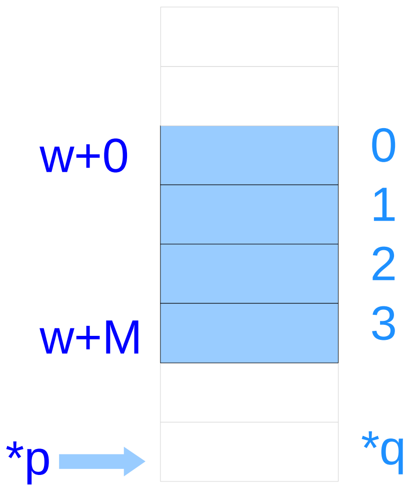
wrap.c

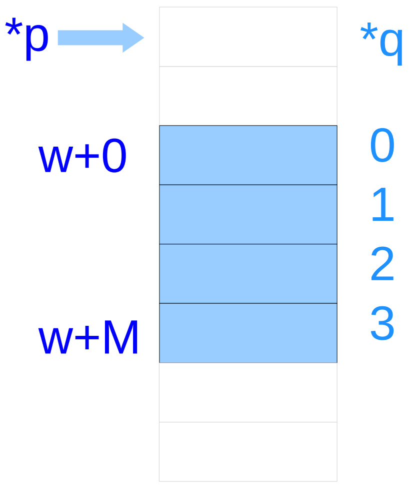
```
/* wrap.c - circular wrap of pointer p, relative to array w */

void wrap(int M, double *w, double **p)
{
        if (*p > w + M)
                *p -= M + 1;          /* when *p=w+M+1, it wraps around to *p=w */

        if (*p < w)
                *p += M + 1;          /* when *p=w-1, it wraps around to *p=w+M */
}
```



wrap2.c

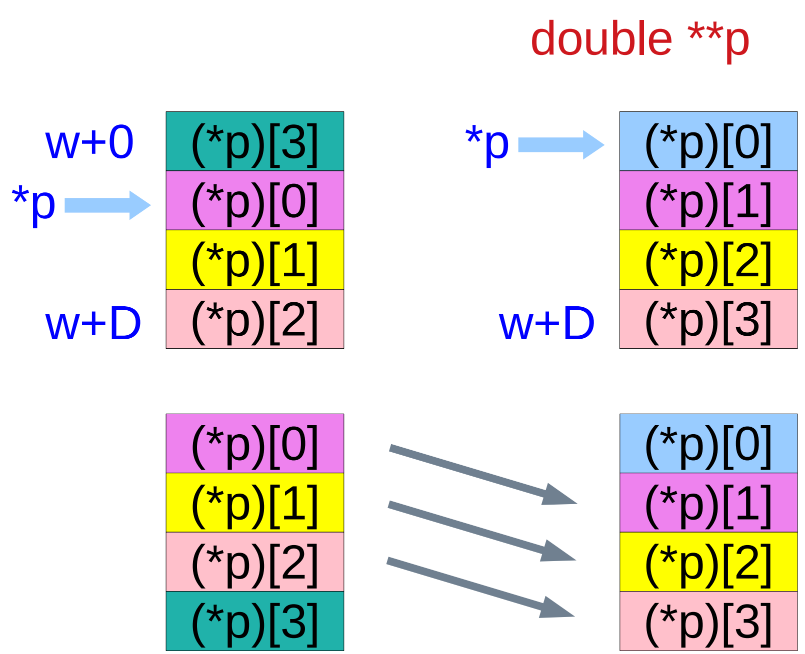
```
/* wrap2.c - circular wrap of pointer offset q, relative to array w */

void wrap2(int M, int *q)
{
        if (*q > M)
                *q -= M + 1;
        if (*q < 0)
                *q += M + 1;
}
```

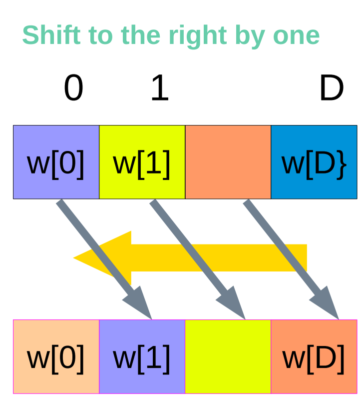




cdelay.c

```
/* cdelay.c - circular buffer implementation of D-fold delay */

void wrap(int M, double *w, double **p);

void cdelay(int D, double *w, double **p)
{
        (*p)--;
        wrap(D, w, p);
}
```

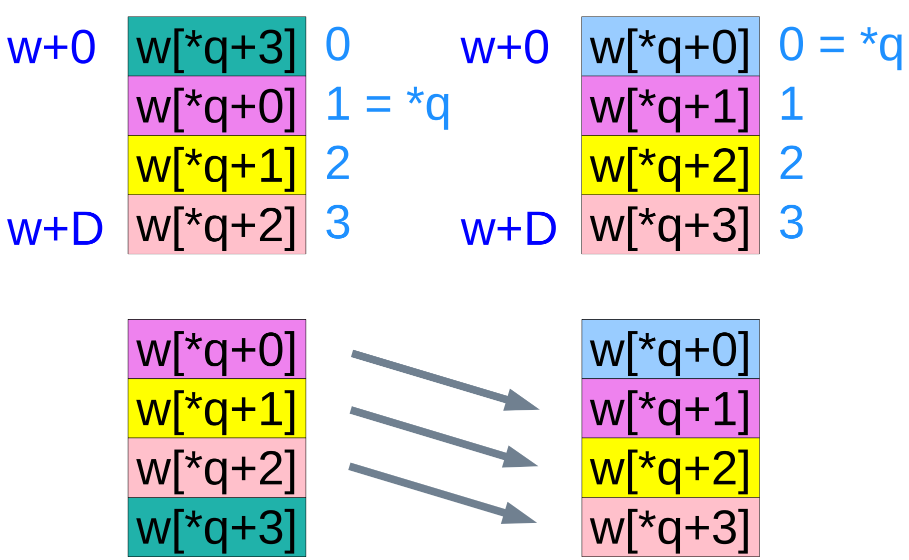


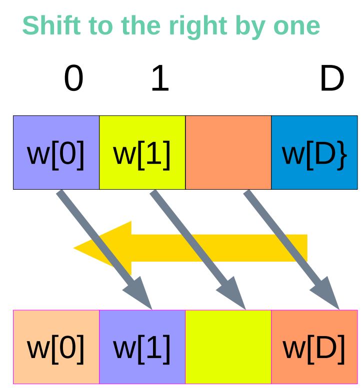


cdelay2.c

```
/* cdelay2.c - circular buffer implementation of D-fold delay */

void wrap2(int M, int *q);

void cdelay2(int D, int *q)
{
        (*q)--;
        wrap2(D, q);
}
```

int *q





Using **cdelay**

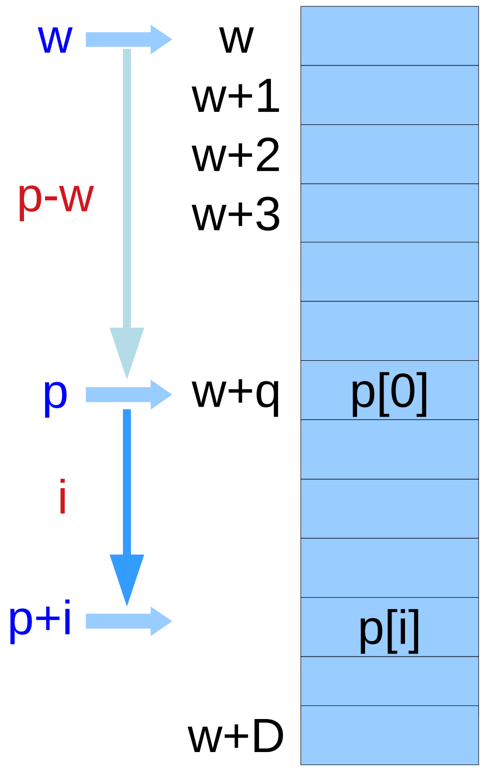
```
void wrap(int M, double *w, double **p);
void cdelay(int D, double *w, double **p);

// implementing the delay equation : y[n] = x[n-D]
double *p;

p = w;                              // initialize p

for (n = 0; n < Ntot; n++) {
    y[n] = w[(p-w+D)%(D+1)];        // write output
    *p = x[n];                      // read input; equivalently, p[0] = x[n]
    cdelay(D, w, &p);               // update delay line
}
```

Using **cdelay2**

```
void wrap2(int M, int *q);
void cdelay2(int D, int *q);

int q;

q = 0;                                  // initialize q

for (n = 0; n < Ntot; n++) {
        y[n] = w[(q+D)%(D+1)];          // alternatively, y[n] = tap2 (D, w, q, D) ;
        w[q] = x[n];                    // read input
        cdelay2(D, &q);                 // update delay line
}
```

tap.c

```
/* tap.c - i-th tap of circular delay-line buffer */
/* usage: si = tap2(D, w, p, i);                     */
/*              i = 0, 1, …, D                        */

double tap(int D, double *w, double *p, int i)
{
      return w[(p - w + i) % (D + 1)];
}
```

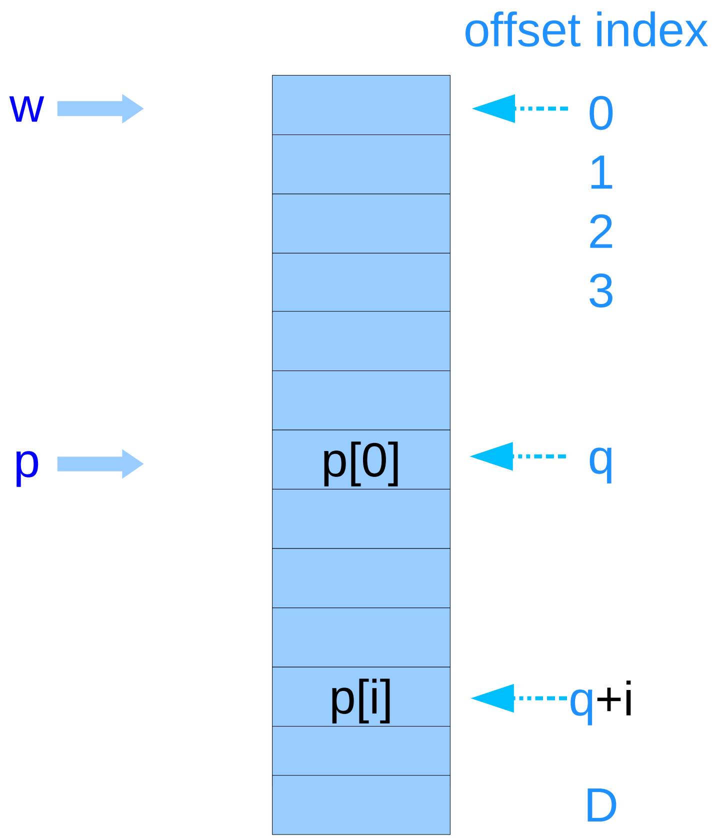


$$w[(p-w+i)\%(M+1)] = w[(q+i)\%(M+1)]$$

tap2.c

```
/* tap2.c - i-th tap of circular delay-line buffer */
/* usage: si = tap2(D, w, q, i);                     */
/*              i = 0, 1, …, D                        */

double tap2(int D, double *w, int q, int i)
{
     return w[(q + i) % (D + 1)];
}
```

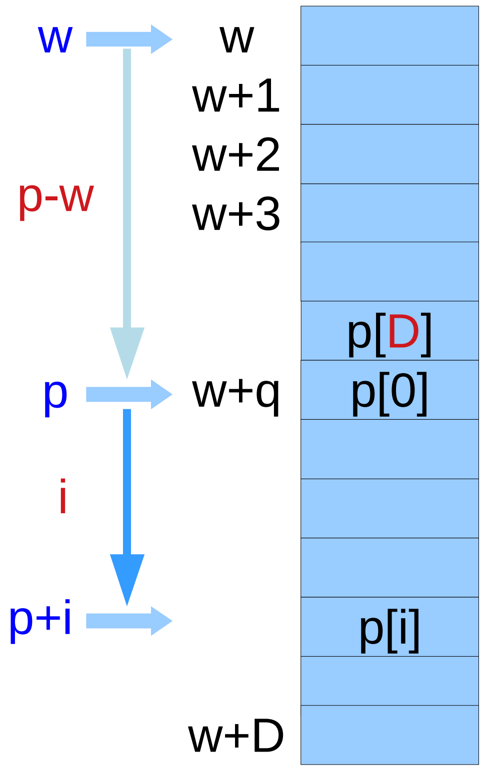
$$w[(p-w+i)\%(M+1)] = w[(q+i)\%(M+1)]$$



Using tap

```
double *p;
p = w;                                  // initialize p
for (n = 0; n < Ntot; n++) {
      y[n] = tap(D, w, p, D);           // D th component of state vector
      *p = x[n];                        // read input; equivalently, p[0]= x[n]
      cdelay(D, w, &p);                 // update delay line
}
```

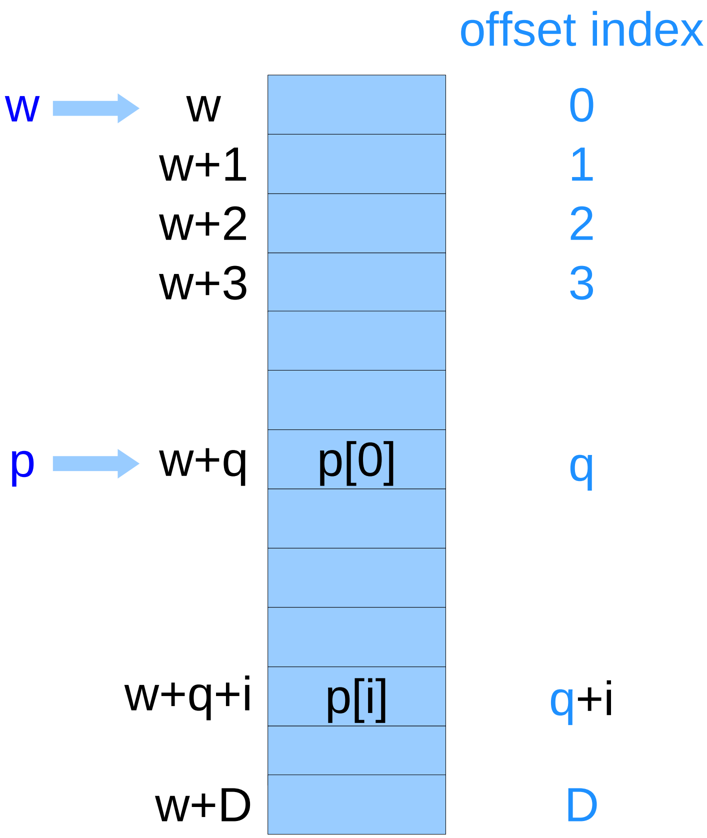
```
double tap(int D, double *w, double *p, int i)
{
      return w[(p - w + i) % (D + 1)];
}

void cdelay(int D, double *w, double **p);
```



Using **tap2**

```
int q;
q = 0;                              // initialize q
for (n = 0; n < Ntot; n++) {
      y[n] = tap2(D, w, q, D);      // D th component of state vector
      w[q] = x[n];                  // read input; equivalently, p[0]= x[n]
      cdelay2(D, w, &q);            // update delay line
}
```

```
double tap2(int D, double *w, int q, int i)
{
      return w[(q + i) % (D + 1)];
}

void cdelay2(int D, int *q);
```



Pointers with ++ and - - operators

x = (* p) ++;

x = (* p) −−;

x = ++ (* p);   x = ++*p;

x = −− (* p);   x = −−*p;

Access First

x = (* p) ++;

x = (* p) −−;

Update First

x = ++ (* p);

x = −− (* p);

Update Next

x = (* p) ++;

x = (* p) −−;

Access Next

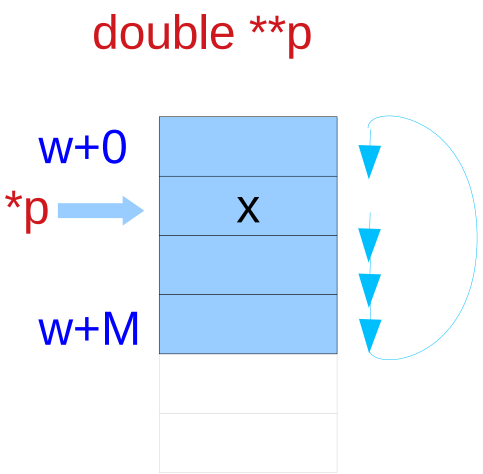
x = ++ (* p);

x = −− (* p);

https://upload.wikimedia.org/wikiversity/en/1/18/C05.Data3.Operators.1.A.20161219.pdf

cfir – moving (*p) pointer forward

```
**p = x;                                  // read input sample x
for (y=0, i=0; i<=M; i++) {               // compute output sample y

    y += (*h++) * (*(*p)++);

    y += (*h) * (*(*p));
    h++;
    (*p)++;

    wrap(M, w, p);
}
```



With respect to the initial h and *p

```
y += h[0] + (*p)[0];
y += h[1] + (*p)[1];

y += h[M] + (*p)[M];
```

cfir.c

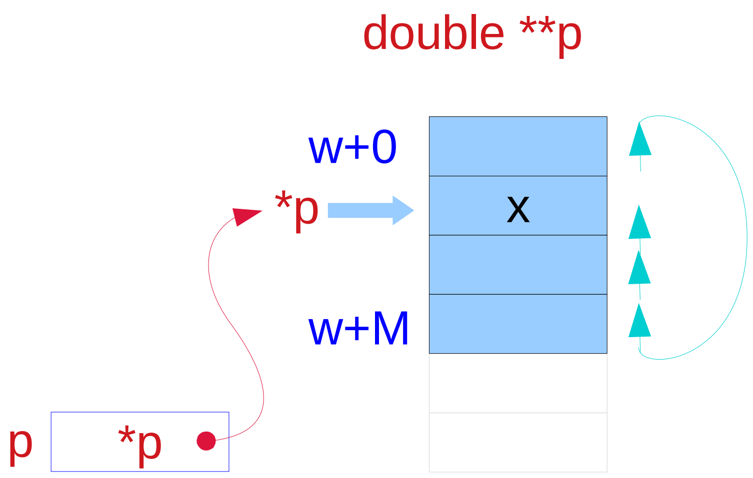
```
/* cfir.c - FIR filter implemented with circular delay-line buffer */
// p = circular pointer to w,  M = filter order

void wrap(int M, double *w, double **p);

double cfir(int M, double *h, double *w, double **p, double x)
{
    int i;
    double y;
    **p = x;                                // read input sample x

    for (y=0, i=0; i<=M; i++) {             // compute output sample y
        y += (*h++) * (*(*p)++);
        wrap(M, w, p);
    }
    (*p)--;                                 // update circular delay line
    wrap(M, w, p);
    return y;
}
```

Using **cfir**

```
double *h, *w, *p;                                    p  [ *p ]

h = (double *) calloc(M+1, sizeof(double));
w = (double *) calloc(M+1, sizeof(double));       // also, initializes w to zero
p = w;                                            // initialize p

for (n = 0; n < Ntot; n++)
      y[n] = cfir(M, h, w, &p, x[n]);             // p passed by address


double cfir(int M, double *h, double *w, double **p, double x);
```

cfir1 – moving (*p) pointer backward

```
*(*p)-- = x;     wrap(M, w, p);          /* p now points to sb{M} */

for (y=0, h+=M, i=M; i>=0; i--) {        /* h starts at hb{M} */
    y += (*h--) * (*(*p)--);

    y += (*h) * (*(*p));
    h--;
    (*p)--;

    wrap(M, w, p);
}
```



With respect to the initial h and *p

```
y += h[M]    + (*p)[M];
y += h[M-1] + (*p)[M-1];

y += h[0]     + (*p)[0];
```

cfir1.c

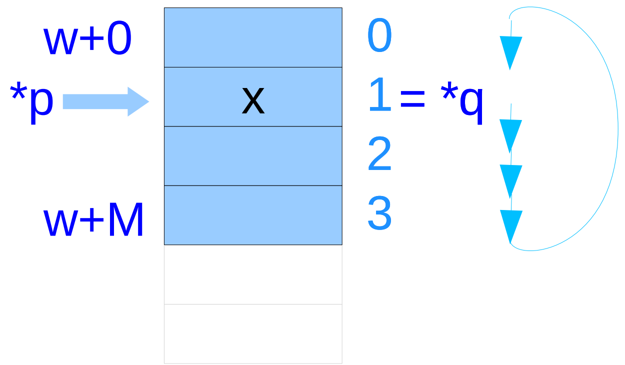
```
/* cfir1.c - FIR filter implemented with circular delay-line buffer */

void wrap(int M, double *w, double **p);

double cfir1(int M, double *h, double *w, double **p, double x)
{
        int i;
        double y;

        *(*p)-- = x;
        wrap(M, w, p);                          /* p now points to sb{M} */

        for (y=0, h+=M, i=M; i>=0; i--) {       /* h starts at hb{M} */
                y += (*h--) * (*(*p)--);
                wrap(M, w, p);
        }

        return y;
}
```

Using **cfir1**

```
double *h, *w, *p;                                    p  [ *p ]

h = (double *) calloc(M+1, sizeof(double));
w = (double *) calloc(M+1, sizeof(double));          // also, initializes w to zero
p = w;                                               // initialize p

for (n = 0; n < Ntot; n++)
      y[n] = cfir1(M, h, w, &p, x[n]);               // p passed by address


double cfir1(int M, double *h, double *w, double **p, double x);
```

cfir2 – incrementing offset integer (*q)

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

for (y=0, i=0; i<=M; i++) {                 // compute output sample y
    y += (*h++) * w[(*q)++];

    y += (*h) * w[(*q)];
    h++;
    (*q)++;

    wrap2(M, q);
}
```



With respect to the initial h and *p

```
y += h[0] + w[(*q)+0];
y += h[1] + w[(*q)+1];

y += h[M] + w[(*q)+M];
```

cfir2.c

```
/* cfir2.c - FIR filter implemented with circular delay-line buffer */
// q = circular offset index, M = filter order
void wrap2(int M, int *q);

double cfir2(int M, double *h, double *w, int *q, double x)
{
        int i;
        double y;

        w[*q] = x;                              // read input sample x
        for (y=0, i=0; i<=M; i++) {             // compute output sample y
                y += (*h++) * w[(*q)++];
                wrap2(M, q);
        }
        (*q)--;                                 // update circular delay line
        wrap2(M, q);

        return y;
}
```

Using **cfir2**

```
double *h, *w; int q;

h = (double *) calloc(M+1, sizeof(double));
w = (double *) calloc(M+1, sizeof(double));          // also, initializes w to zero
q = 0;                                               // initialize q

for (n = 0; n < Ntot; n++)
      y[n] = cfir2(M, h, w, &q, x[n]);               // q passed by address


double cfir2(int M, double *h, double *w, int *q, double x);
```

Loop in the **cfir**

```
for (y=0, i=0; i<=M; i++)
    y += h[i] * w[(*q+i)%(M+1)];          // used by cfir2.m of Appendix D
```

Using **cfir2**

```
double cfir2(int M, double *h, double *w, int *q, double x)

double *h, *w;
int q;

h = (double *) calloc(M+1, sizeof(double));
w = (double *) calloc(M+1, sizeof(double));            // also, initializes w to zero

q= 0;                                                   // initialize q
for (n = 0; n < Ntot; n++)
      y[n] = cfir2(M, h, w, &q, x[n]);                  // q passed by address
```

FIR Filters using **tap** and **cdelay** functions

for each input sample x do:
$w_0 := x$
$y := h_M w_M$
for i = M− 1 , . . . , 1 , 0 do:
    $w_{i+1} := w_i$
    $y := y + h_i w_i$

for each input sample x do:
$s_0 = *p = x$
for i = 1 , 2 , . . . , M determine states:
    s_i = **tap** (M, w , p, i)
$y = h_0 s_0 + h_1 s_1 + \cdots + h_M s_M$
cdelay (M, w , &p)

for each input sample x do:
$s_0 = w[q] = x$
for i = 1 , 2 , . . . , M determine states:
    s_i = **tap2** (M, w , q, i)
$y = h_0 s_0 + h_1 s_1 + \cdots + h_M s_M$
cdelay2 (M, &q)

FIR Filters using **tap** and **cdelay** functions

for each input sample x do:
s_0 = *p = x
for i = 1 , 2 , . . . , M determine states:
    s_i = **tap** (M, w , p, i)
$y = h_0 s_0 + h_1 s_1 + \cdots + h_M s_M$
cdelay (M, w , &p)

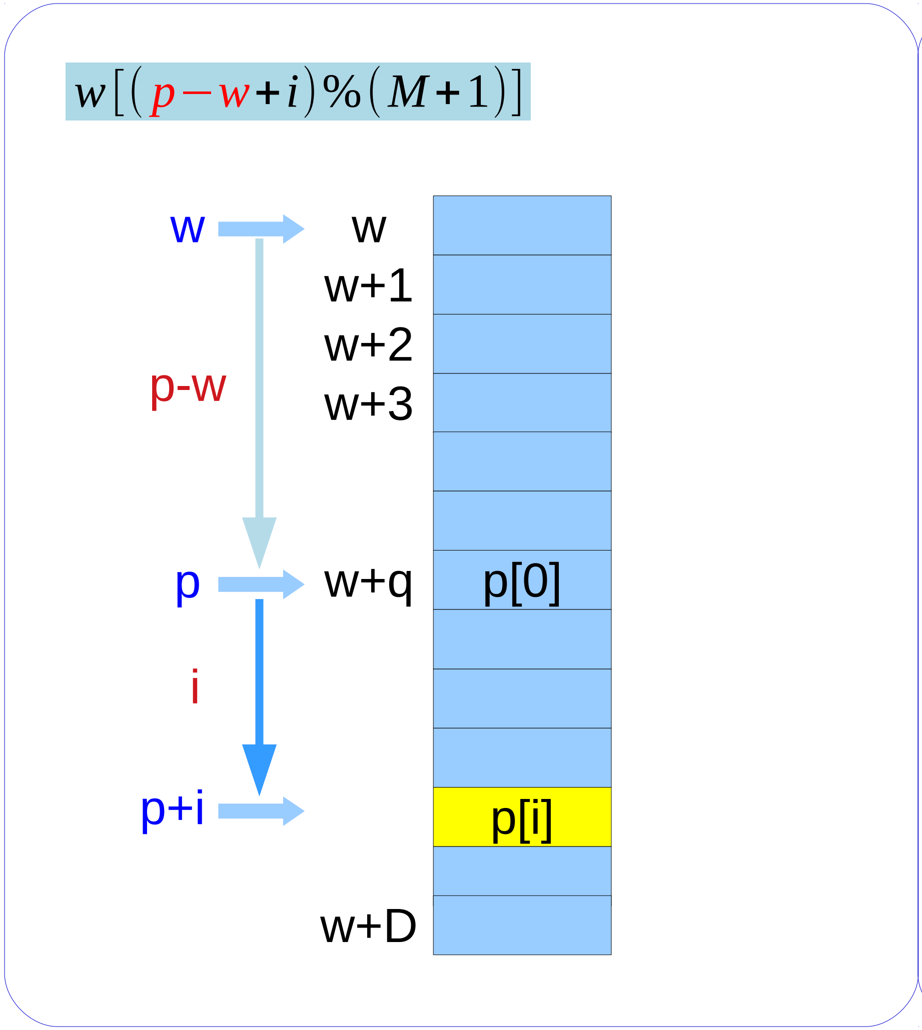
for each input sample x do:
s_0 = w[q]= x
for i = 1 , 2 , . . . , M determine states:
    s_i = **tap2** (M, w , q, i)
$y = h_0 s_0 + h_1 s_1 + \cdots + h_M s_M$
cdelay2 (M, &q)

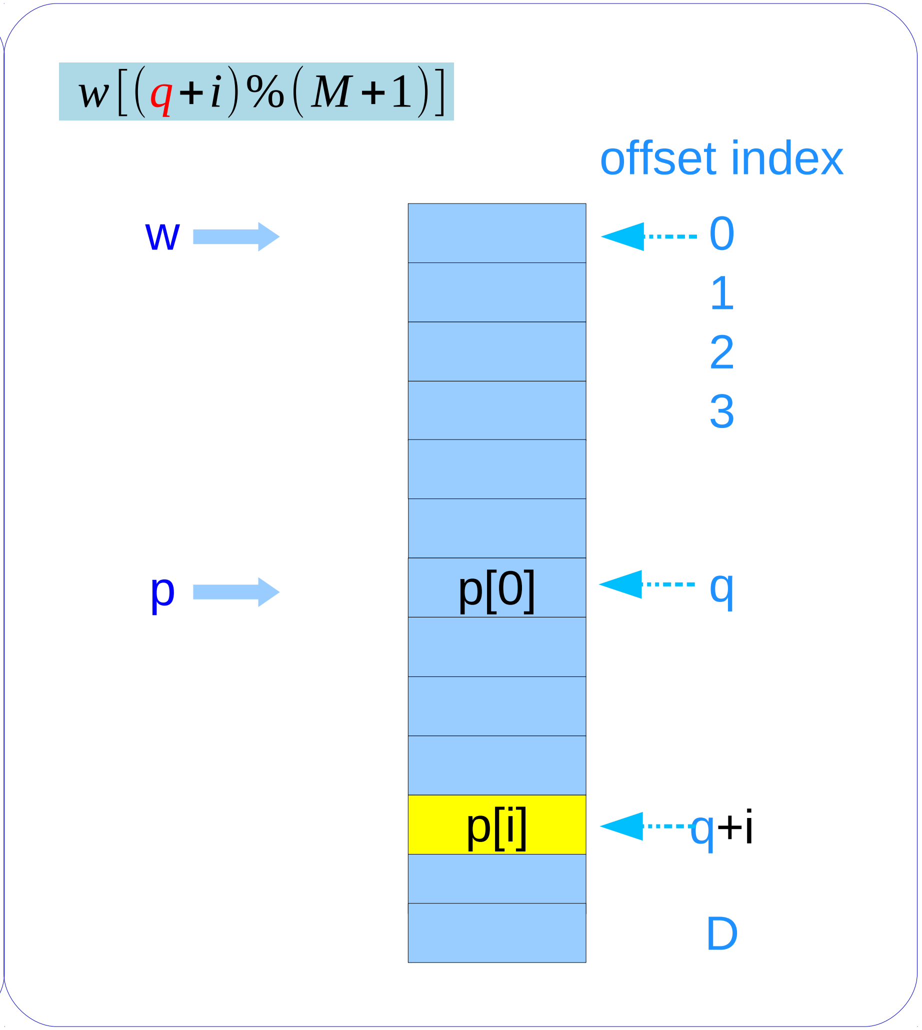
for each input sample x do:
s_0 = *p = x, $y = h_0 s_0$
for i = 1 , 2 , . . . , M determine states:
    s_i = **tap** (M, w , p, i)
    $y \mathrel{+}= h_i s_i$
cdelay (M, w , &p)

for each input sample x do:
s_0 = w[q]= x, $y = h_0 s_0$
for i = 1 , 2 , . . . , M determine states:
    s_i = **tap2** (M, w , q, i)
    $y \mathrel{+}= h_i s_i$
cdelay2 (M, &q)

tap and **tap2** – fetching p[i]

double **tap**(int D, double *w, double *p, int i) double **tap2**(int D, double *w, int q, int i)

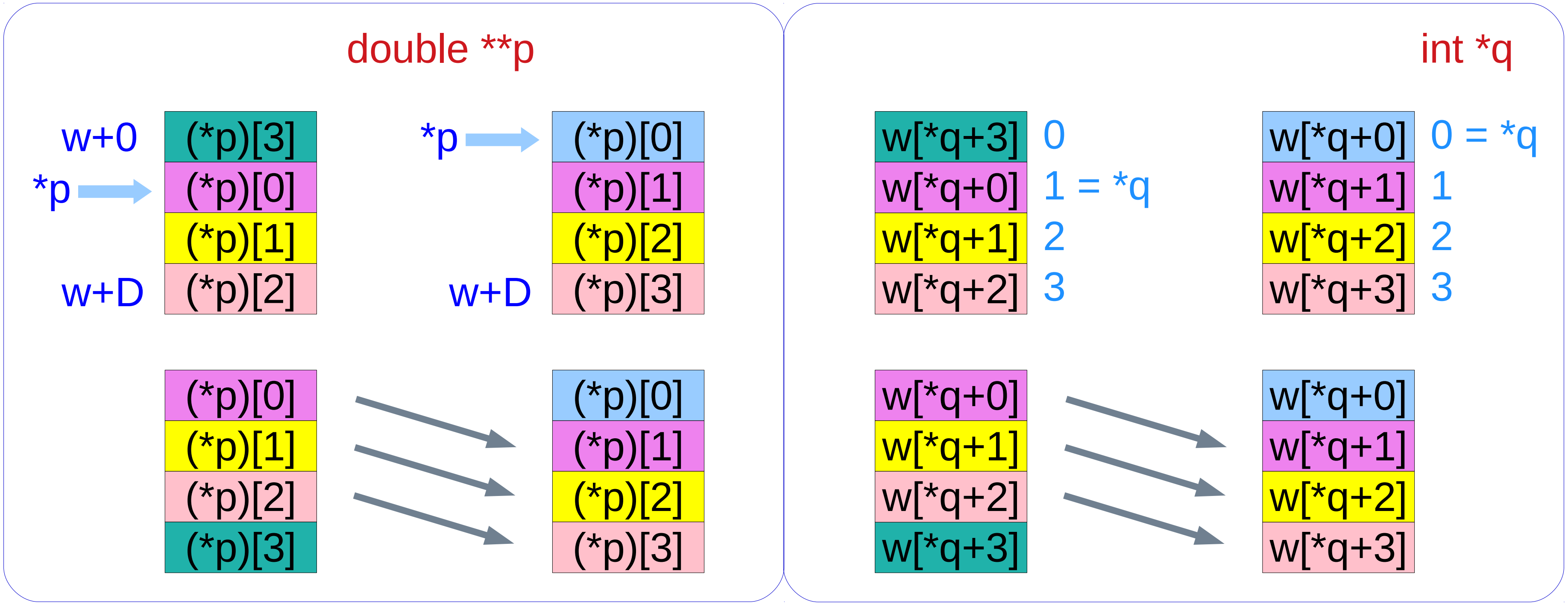




cdelay and cdelay2

void **cdelay**(int D, double *w, double **p)

void **cdelay2**(int D, int *q)



Sample processing algorithm ex 1

h = [1,2,-1,1],
x = [1,1,2 1,2,2,1,1]

$y(n)= x(n) + 2\, x(n-1) - x(n-2) + x(n-3)$

$w0(n) = x(n)$
$y(n) = w0(n) + 2\, w1(n) - w2(n) + w3(n)$
$w3(n+1) = w2(n)$
$w2(n+1) = w1(n)$
$w1(n+1) = w0(n)$

Sample processing algorithm ex 1

```
/* firexmpl.c - Example of FIR sample processing algorithm */
// h = [ 1,2,-1,1 ],
// x = [ 1,1,2,1,2,2,1,1 ]


#include <stdio.h>
#include <stdlib.h>                    // declares calloc

double x[8] = {1,1,2,1,2,2,1,1};       // input signal
double filter();
```

Sample processing algorithm ex 1

```
int main(void) {
        int n;
        double y, *w;

        w = (double *) calloc(4, sizeof(double));

        for (n=0; n<8; n++) {
                y = filter(x[n], w);
                printf("%lf\n", y);
        }

        for (n=8; n<11; n++) {
                y = filter(0.0, w);                // input-off transients
                printf("%lf\n", y);                // called with x = 0
        }

}
```

Sample processing algorithm ex 1

```
// Usage: y = filter(x, w);

double filter(double x, double *w)
{
    double y;

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

    y = w[0] + 2 * w[1] - w[2] + w[3];      // compute output sample

    w[3] = w[2];
    w[2] = w[1];
    w[1] = w[0];                            // update internal states

    return y;
}
```

Sample processing algorithm ex 1

n x w 0 w 1 w 2 w 3 w 4 0 1 1 0 0 0 0 1 1 1 1 0 0 0 1

2 2 2 1 1 0 0 2

3 1 1 2 1 1 0 1

4 2 2 1 2 1 1 1

5 2 2 2 1 2 1 1

6 1 1 2 2 1 2 7 1 1 1 2 2 1 − 1

8 0 0 1 1 2 2 9 0 0 0 1 1 2

10 0 0 0 0 1 1

11 0 0 0 0 0 1

y = w 0 − w 4

1

0

− 2

− 2

− 1

− 1

Sample processing algorithm ex 2

$y(n) = x(n) - x(n-4)$

x = [1,1,2,1,2,2,1,1]

$w0(n) = x(n)$
$w1(n) = x(n-1) = w0(n-1)$
$w2(n) = x(n-2) = w1(n-1)$
$w3(n) = x(n-3) = w2(n-1)$
$w4(n) = x(n-4) = w3(n-1)$

Sample processing algorithm ex 2

$w0(n) = x(n)$

$y(n) = w0(n) - w4(n)$

$w4(n+1) = w3(n)$
$w3(n+1) = w2(n)$
$w2(n+1) = w1(n)$
$w1(n+1) = w0(n)$

References

[1] S. J. Ofranidis , Introduction to Signal Processing