1 Introduction

Real time DSP (Discrete Signal Processing) applications

A standard MATLAB program provides a non-real time signal processing algorithm. Non – real time means all the input samples are available at the same time. Meanwhile, real- time application – which gives a continuous supply of input samples – differs from a standard MATLAB program. Because the input data is a continuous stream, it needs to be processed continuously also making the timing critical for the latency to be low. Real time processing in DSP uses a buffer-based strategy shown in figure 1.

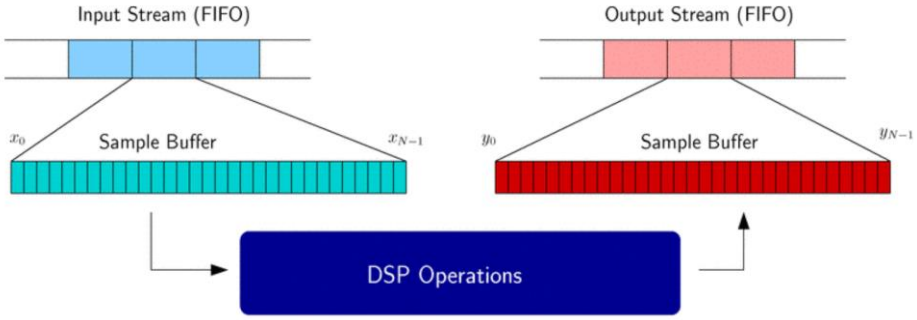


Figure 1: Buffer based strategy of real time DSP

The input is divided into buffers with N samples. The DSP algorithm operates on one buffer at a time giving N output samples. The information of previous buffers is stored in state for the following buffers to be processed correctly. In addition, a smaller buffer size gives less latency. However, it spends time storing and restoring state giving lower throughput. Furthermore, coding DSP in MATLAB gives faster processing time because of user-friendly environment.

Circular Buffers

Because Real-time DSP has a continuous flow of input samples, the samples are simulated by using finite-sized circular buffers. (Figure 2) The circular buffer is referred as first-in first-out (FIFO) buffer.

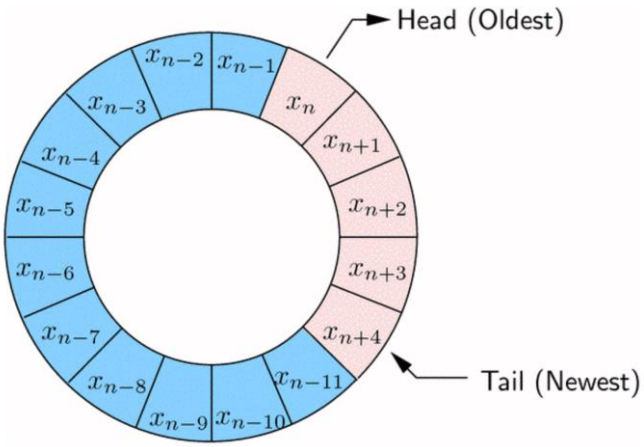


Figure 2: Circular buffer

The incoming data is stored in tail following by the incrementation of the tail pointer. The oldest not-processed item is taken from the head of the buffer also followed by the incrementation of the head pointer. This method is convenient because unnecessary

process of copying data is avoided. It can be processed simply by updating pointers giving more efficiency. In a processor, the data is arranged in a linear way – flattened circular buffer. (Figure 3)

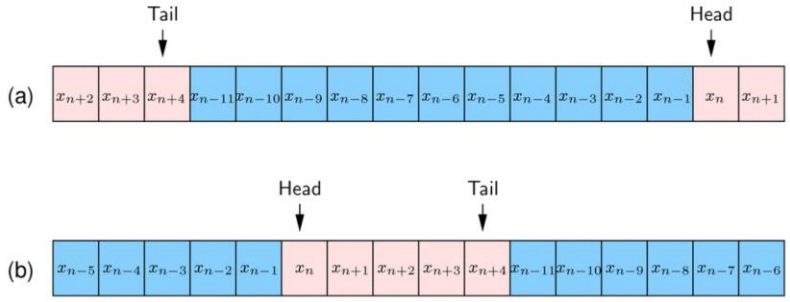


Figure 3: Linear data arrangement

The pointer at the end is wrapped back to the front by using modulo operation with bit masking operation similar to the code shown below.

```
int samples[16];
unsigned int head=0;
unsigned int tail=0;

void put_sample(int samp1)
{
  /* Put a sample at the tail of the buffer */
  samples[tail] = samp1;

  /* Increment.  By masking bits, make modulo 16 */
  tail = (tail + 1) & 15;
}

int get_sample()
{
    int samp1;

    /* Get a sample from head of buffer. */
    samp1 = samples[head];

    /* Increment to next value */
    head = (head + 1) & 15;
}
```

2 Execution/Evaluation

Delay Block Initialization

```
function [state] = delay_init(Nmax, N)
%initialize the delay block
% Inputs:
%   Nmax    Maximum delay supported by this block.
%   N   Initial delay
% Outputs:
%   state   State of block
%% 1. Save parameters
state.Nmax = Nmax;

% Store initial desired delay.
state.N = N;

%% 2. Create state variables

% Make the size of the buffer at least twice of the maximum delay.
% Allows us to copy in and then read out in just two steps.
state.M = 2^(ceil(log2(Nmax))+1);

% Get mask allowing us to wrap index easily
state.Mmask = state.M-1;

% Temporary storage for circular buffer
state.buff = zeros(state.M, 1);

% Set initial head and tail of buffer
state.n_h = 0;
state.n_t = state.N;
```

The function has input of Nmax and N and an output of a structure state. The function creates a circular buffer (M) of integer power 2 in size to store samples. The mask (Mmask) is created to do AND operation for modulo Nmax indexing.
The head indices was assigned as 0 and the tail was assigned as N to make a circular transition for the delay block.

Delay block processing function

The function processes a single buffer of samples.
The input state is stored in s. N input samples are copied to the tail of the buffer of length N with the increment of the tail pointer. Then, the N output sample are read from the head of the buffer with the increment of the head pointer. Then the output buffer is returned.

```
function [state_out, y] = delay(state_in, x);
% Delays a signal by the specified number of samples.
% Inputs:
%   state_in    Input state
%   x       Input buffer of samples
% Outputs:
%   state_out   Output state
%   y       Output buffer of samples

% Get input state
s = state_in;

% Copy in samples at tail
for ii=0:length(x)-1
  % Store a sample
  s.buff(s.n_t+1) = x(ii+1);
  % Increment head index (circular)
  s.n_t = bitand(s.n_t+1, s.Mmask);
end

% Get samples out from head
y = zeros(size(x));
for ii=0:length(y)-1
  % Get a sample
  y(ii+1) = s.buff(s.n_h+1);
  % Increment tail index
  s.n_h = bitand(s.n_h+1, s.Mmask);
end

% Output the updated state
state_out = s;
```

Testing delay block

Two plots are generated. Plot 1 shows the input and output vector. Plot 2 shows where the signals are plotted with shift “undone”

```
% test_delay1.m
%
% Script to test the delay block.  Set up to model the way
% samples would be processed in a DSP program.

% Global parameters
Nb = 10;    % Number of buffers
Ns = 128;   % Samples in each buffer
Nmax = 200; % Maximum delay

Nd = 10;    % Delay of block

% Initialize the delay block
state_delay1 = delay_init(Nmax, Nd);

% Generate some random samples.
x = randn(Ns*Nb, 1);
% Reshape into buffers
xb = reshape(x, Ns, Nb);

% Output samples
yb = zeros(Ns, Nb);

% Process each buffer
for bi=1:Nb,
  [state_delay1 yb(:,bi)] = delay(state_delay1, xb(:,bi));
end

% Convert individual buffers back into a contiguous signal.
y = reshape(yb, Ns*Nb, 1);

% Check if it worked right
n = [0:length(x)-1];

figure(1);
plot(n, x, n, y);

figure(2);
plot(n+Nd, x, n, y, 'x');

% Do a check and give a warning if it is not right.  Skip first buffer in check
% to avoid initial conditions.
n_chk = 1+[Ns:(Nb-1)*Ns-1];
if any(x(n_chk - Nd) ~= y(n_chk))
  warning('A mismatch was encountered.');
end
```

1) Delay of 10

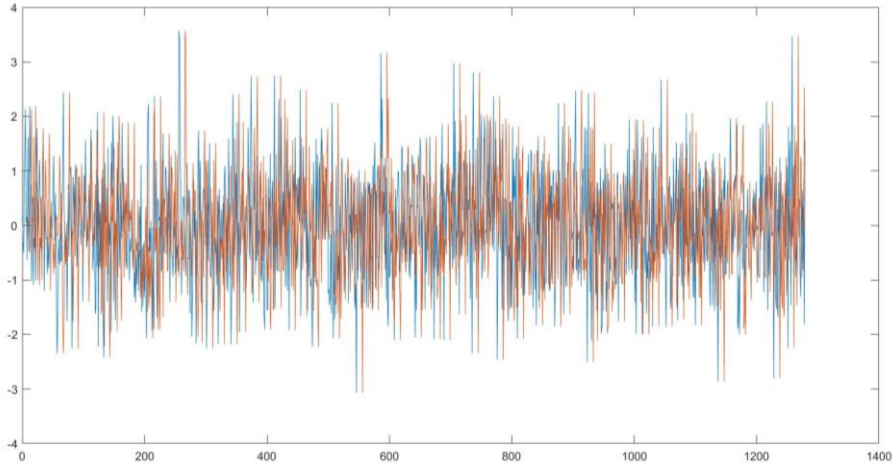


Figure 4: Plot 1

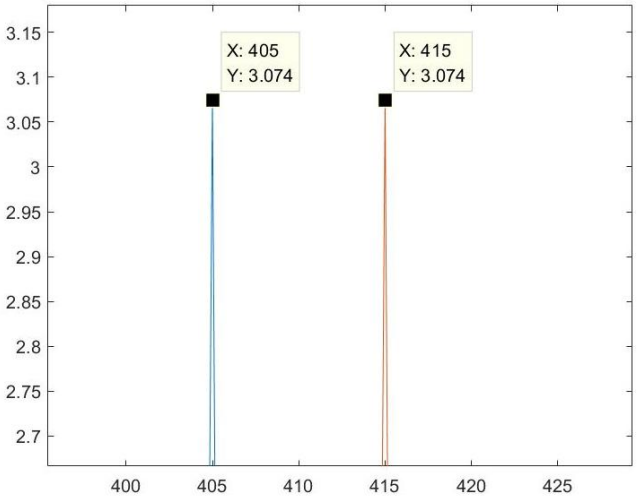


Figure 5: Plot 1 showing time delay

There is a time delay of 10. Which shows the correct response according to the code.

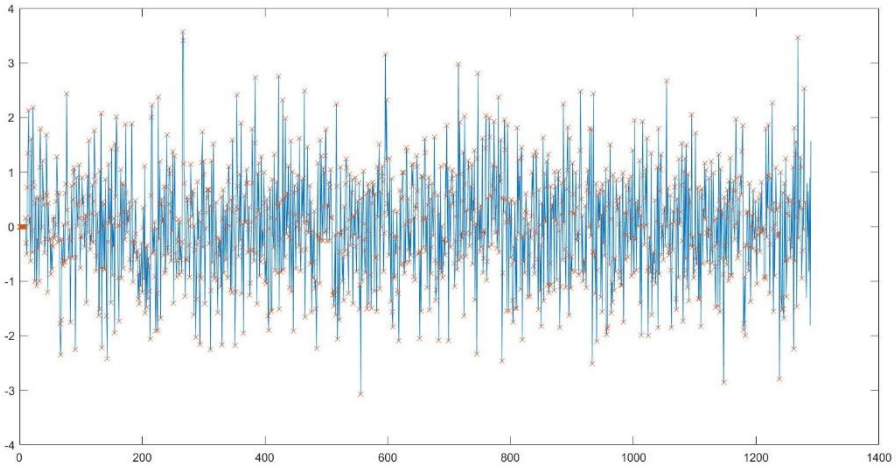
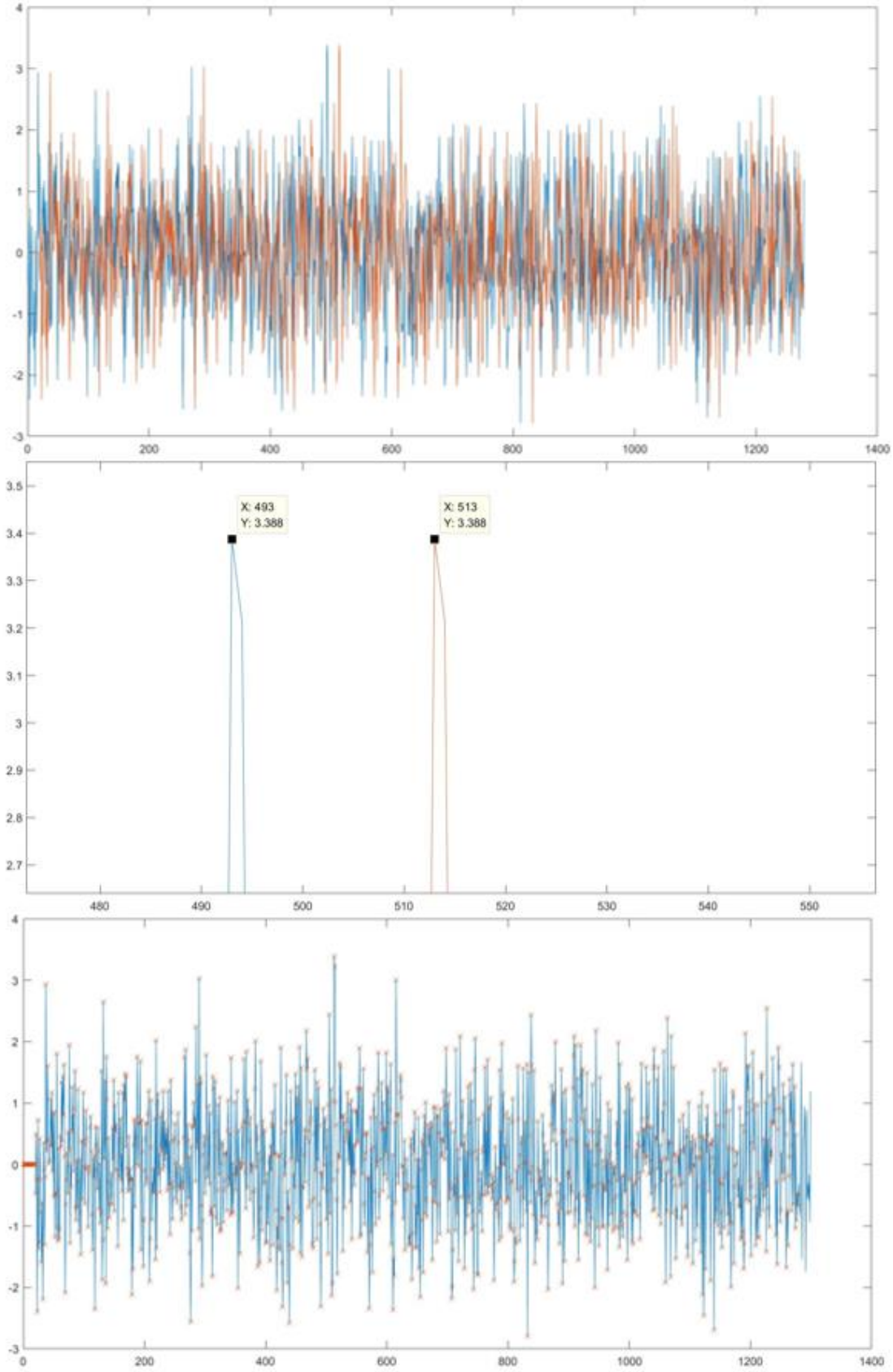


Figure 6: Plot 2

The undone shift are shown in plot 2. The samples lined up and matched correctly.

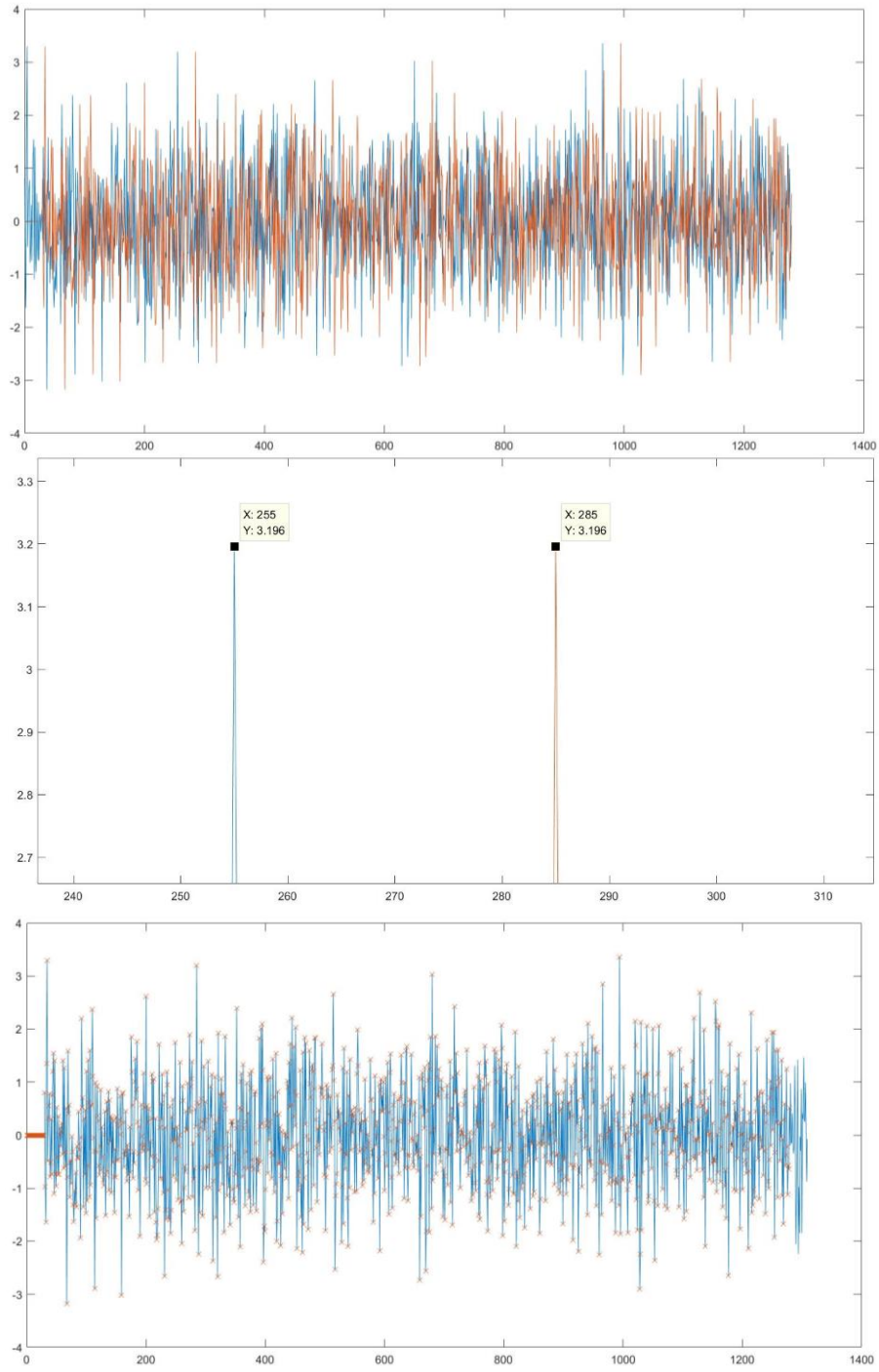
2) Delay of 20



The plot has a delay of 20. Showing the code works fine.

3) Cascade of two delay blocks

```
clear; close all; clear all;
%% test_delay1.m
% Global parameters
Nb = 10;    % Number of buffers
Ns = 128;   % Samples in each buffer
Nmax = 200; % Maximum delay

Nd = 10;    % Delay of block
Nd1 = 20;   % 2nd Delay

% Initialize the delay block
state_delay1 = delay_init(Nmax, Nd);
state_delay2 = delay_init(Nmax, Nd1);

% Generate some random samples.
x = randn(Ns*Nb, 1);
% Reshape into buffers
xb = reshape(x, Ns, Nb);

% Output samples
yb = zeros(Ns, Nb);
yb1 = zeros(Ns, Nb);

% Process each buffer
for bi=1:Nb
  [state_delay1 yb(:,bi)] = delay(state_delay1, xb(:,bi));
  [state_delay2 yb1(:,bi)] = delay(state_delay2, yb(:,bi)); %2nd delay
end

% Convert individual buffers back into a contiguous signal.
y = reshape(yb1, Ns*Nb, 1);

% Check if it worked right
n = [0:length(x)-1];

figure(1);
plot(n, x, n, y);

figure(2);
plot(n+Nd+Nd1, x, n, y, 'x');

% Do a check and give a warning if it is not right.  Skip first buffer in check
% to avoid initial conditions.
n_chk = 1+[Ns:(Nb-1)*Ns-1];
if any(x(n_chk - Nd) ~= y(n_chk))
  warning('A mismatch was encountered.');
end
```



Have a delay of 30, showing the cascade code worked well

Check – off

(1) Explain the difference between non-real-time MATLAB processing and real-time DSP code.

For usual MATLAB simulation, all the input samples are available in the same time therefore the time for algorithm to run is not important. Meanwhile, real-time DSP code has a continuous stream of input signal and the output should also be processed continuously. This makes the timing critical to make the latency- the input to output delay- low.
This is explained more in lab write up 1.

(2) Demonstrate that the delay block works correctly. Also show how you set the initial head and tail pointers and how this gives the delay you want.

The head indices was assigned as 0 and the tail was assigned as N to make a circular transition for the delay block. Also, the mask (Mmask) is created to do bitand operation for modulo Nmax indexing.

(3) Show that cascaded delay blocks work.
Shown in timing delay block part 3.

Lab Write up

(1) A short explanation of the difference between a usual MATLAB simulation and real-time DSP code. Also, briefly explain why buffer-oriented processing is needed and what affects the latency and throughput of an algorithm.

For usual MATLAB simulation, all the input samples are available in the same time therefore the time for algorithm to run is not important. Meanwhile, real-time DSP code has a continuous stream of input signal and the output should also be processed continuously. This makes the timing critical to make the latency- the input to output delay - low.
Buffer-oriented processing is needed because the input signal is not available at once. The input is divided into buffers of sample size N. It operates on one buffer at a time giving N outputs. The size of the buffer affects the latency. Smaller the buffer size gives a shorter time to fill the buffer – resulting in smaller latency. However, if the buffer size is small results having a lot of buffers. Thus it takes more time storing and restoring the state causing lower overall throughput. Therefore an ideal balance in the buffer sample size (N) needs to be made.

(2) A diagram showing conceptually how your Delay uses a circular buffer to implement the Delay.

Because Real-time DSP has a continuous flow of input samples, the samples are simulated by using finite-sized circular buffers. The incoming data is stored in tail following by the incrementation of the tail pointer. The oldest not-processed item is taken from the head of the buffer also followed by the incrementation of the head pointer. This method is convenient because unnecessary process of copying data is avoided. It can be processed simply by updating pointers giving more efficiency.

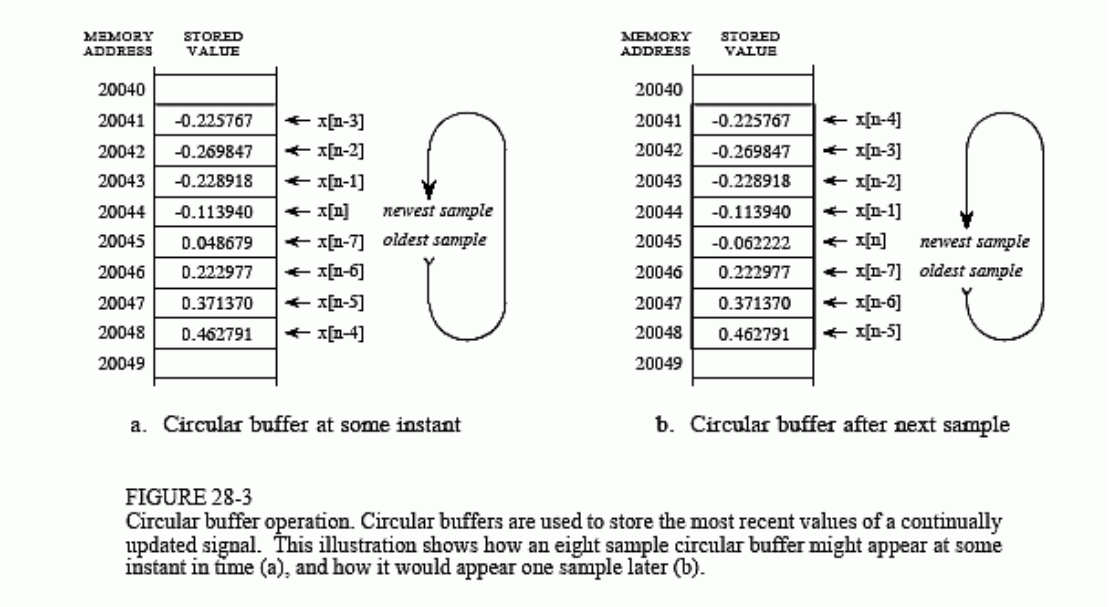


FIGURE 28-3
Circular buffer operation. Circular buffers are used to store the most recent values of a continually updated signal. This illustration shows how an eight sample circular buffer might appear at some instant in time (a), and how it would appear one sample later (b).

(3) A printout of the MATLAB code that implements your Delay with comments.

Code is attached above in the execution section.

3 **Conclusion**

The use of Matlab to code real time DSP and delay block was studied in the experiment. To operate real time processing, circular buffer was used coded by mask, AND operator and pointer. The coded delay block worked well, also cascade of delay blocks was coded and functioned well. There were some troubles when coding the cascade, but errors were easily fixed. Results turned out to be correct showing the right delay according to the code.

4 **References**

[1] http://dsp-fhu.user.jacobs-university.de /?page_id=142