8 CONCLUSIONS

The LMS algorithm is simple and has less computational complexity when compared with the RLS algorithm. But, the RLS algorithm has faster convergence rate than LMS algorithm at the cost of higher computational complexity. The λ plays a role in RLS algorithm similar to that of the step-size parameter μ in the LMS algorithm. RLS algorithm has better performance than LMS algorithm in the low signal to noise ratio.

The tracking performance, at first glance, it might be tempting to say that since the RLS algorithm has a faster rate of convergence than the LMS algorithm in the case of a stationary environment, therefore, it will track a nonstationary environment better than the LMS algorithm. Such an answer, however, is not justified because the tracking performance of an adaptive filtering algorithm is influenced not only the rate of convergence but also by fluctuation in the steady-state performance of the algorithm due to measurement and algorithm noise. But generally, the tracking performance of RLS algorithm is better than LMS type algorithms.

7 REFERENCES AND BIBLIOGRAPHY

1. Adaptive Signal Processing by Bernard Widrow and Samuel D Stearns.

2. Digital Signal Processing by Richard A Haddad and Thomas W Parsons.

3. Signal Processing Algorithms by Samuel D Stearns and Ruth A David.

4. Adaptive Filter Theory by Simon Haykin.

5 Essential MATLAB for Scientists and Engineers by Brian D. Hahn.

6 A Variable Step Size LMS Algorithm by Raymond H. Kwong, IEEE Transactions on Signal Processing Vol. 40 No. 7, July 1992.

7 A Robust Variable Step-Size LMS type Algorithm : Analysis and Simulations By Tyseer Aboulnasr, IEEE Transaction on Signal Processing Vol.45 No.3 March 1997.

8 Comparison of RLS, LMS, and Sign Algorithms for Tracking Randomly Time-Varying channels by Eweda Eweda, IEEE Transaction on Signal Processing, Vol. 42, No. 11, November 1994.

9. The Stability of Variable Step-Size LMS Algorithms by Saul B Gelfand, Yongbin Wei and James V Krogmeier, IEEE Transaction on Signal Processing, Vol. 47, No. 12, December 1999.

10. Complexity Reduction of the NLMS Algorithm via Selective coefficient Update by T. Aboulnasr and K. Mayyas, IEEE Transaction on Signal Processing, Vol. 47, No. 05, May 1999.

11. Convergence Analysis of the Multi-Variable Filtered-X LMS Algorithm with Application to Active Noise Control by A. Kuo Wang and Wei Ren, IEEE Transaction on Signal Processing, Vol. 47, No. 04, April 1999.

12. Nonlinear Effects in LMS Adaptive Equalizers by Michael Reuter and James R. Zeidler, IEEE Transaction on Signal Processing, Vol. 47, No. 06, June 1999.

13. Prediction in LMS-Type Adaptive Algorithms for Smoothly Time Varying Environments by Saeed Gazor, IEEE Transaction on Signal Processing, Vol. 47, No. 06, June 1999.

14. Analysis of the Frequency – Domain Block LMS Algorithm by B. Farhang-Boroujeny, IEEE Transaction on Signal Processing, Vol. 48, No. 8, August 2000.

15. Computationally Efficient Frequency-Domain LMS Algorithm with Constraints on the Adaptive filter by Boaz Rafaely and J. Elliott, IEEE Transaction on Signal Processing, Vol. 6, June 2000.

16. Mean Weight Behavior of the filtered – X LMS Algorithm by Orlando J. Tobias et all, IEEE Transactions on Signal Processing, Vol. 48, No. 4, April 2000.

17. 17. Regularized fast Recursive Least Squares Algorithms for Finite Memory Filtering by Ricardo Merched and Ali H. Sayed, IEEE Transactions on Signal Processing, Vol. 40, No. 04, April 1992.

18. Rotation-Based RLS Algorithms: Unified Derivations, Numerical Properties, and Parallel Implementation by Bin Yang and Johann F. Bohme, IEEE Transactions on Signal Processing, Vol. 40, No. 05, May 1992.

19. Delta Levinson and Schur-Type RLS Algorithms for Adaptive Signal Processing by H. Fan and Xiaqu Liu, IEEE Transactions on Signal Processing, Vol. 42, No. 07, June 1994.

20. A Fast Least-Square Algorithm for Linearly Constrained Adaptive Filtering by Leonardo S. Resende and Joao Marcos T. Romano, IEEE Transactions on Signal Processing, Vol. 44, No. 05, May 1996.

21. Pipelined RLS Adaptive Filtering Using Scaled Tangent Rotations (STAR) by Kalavai J. Raghunath and Keshab K. Parhi, IEEE Transactions on Signal Processing, Vol. 44, No. 10, October 1996.

22. Adaptive Recovery of a Chirped Signal Using the RLS Algorithm by Payl C. Wei, James R. Zeidler and Walter H. Ku, IEEE Transactions on Signal Processing, Vol. 45, No. 02, February 1997.

23. Finite-Precision Error Analysis of QRD-RLS and STAR-RLS Adaptive Filters by Kalavai J. Raghunath and Keshab K. Parhi, IEEE Transactions on Signal Processing, Vol. 45, No. 05, October 1997.

24. A Delta Least Squares Lattice Algorithm for Fast Sampling by Parthapratim De and H. Howard Fan, IEEE Transactions on Signal Processing, Vol. 47, No. 09, September 1999.

25. Convergence Analysis of the Sign Algorithm Without the Independence and Gaussian Assumptions by Eweda Eweda, IEEE Transactions on Signal Processing, Vol. 48, No. 09, September 2000.

26. Order-Recursive RLS Laguerre Adaptive Filtering by Ricardo Merched and Ali H. Sayed, IEEE Transactions on Signal Processing, Vol. 48, No. 11, November 2000.

8 Appendix A – LMS Algorithm Program Listing

8.1 Fixed Step Size LMS Algorithm

```
% Fixed step size LMS algorithm

% Y=FSSLMS(X,N,D,B,L,MU,SIG,AL,PX,IERROR)
% subroutine for LMS algorithm and implements the
equation
% B(K+1)= B(K)  + 2*MU*E*X(K)/((L+1)*SIG)
% X       = Data vector, Input sent and Output returned
% N       = Number of data in input X
% D       = Desired signal vector
% W       = Adaptive coefficients of Lth order FIR
filter
% L       = Order of adaptive system
% MU      = Convergence parameter
% SIG     = Input signal power estimate
% AL      = Forgetting factor alfa.
% PX      = Vector that retain the past input.
% NS    = Number of samples per cycle

N=500;
MU=0.1;
L=6;
SIG=0.005;
AL=0;
IERROR=0;
NS=20; % Number of sample per cycle
for K=1:N
   F=2*pi*K/NS;
  SI(K,1)=2*sin(F+pi/3);

D(K,1)=(sprand(12357)-0.5)  + SI(K);
%SPRAND - Generate uniformly distributed random
numbers
   X(K,1)= 0.3*sin(F);
end
subplot(2,2,1)
plot(SI)
title('Original Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,3)
plot(D)
hold on
```

```
plot(X,'r')
title('Noise contaminated and Reference Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')


disp('Executing LMS')
W=zeros(L+1,1);
PX=zeros(L+1,1);
MSE = zeros(N,1);
PSD=zeros(NS+L+1,1);
PSX=zeros(NS+L+1,1);
IERROR1 = 5;
C=0;
for K= 1 : N
  PX(1)=X(K);
  PSD(1)=D(K);
  PSX(1) = X(K);
  X(K)=  PX'*W;

   %calculate equation (4.1.1)
   if(abs(X(K))) > 1E10
      disp('output is too large')
      X(K)
      K
      return
   end

   E =D(K)-X(K);                           % calculate the
equation (4.1.2)
   SIG = AL*(PX(1)*PX(1)) + (1-AL)*SIG; % calculate
the equation (4.1.8)
   TMP=2*MU/((L+1)*SIG);
   W=W+TMP*E*PX;                   % calculate W(n+1) =
W(n) + TMP*E*X(n) equation (4.1.9)

   % Calculate MSE
   R =MakeR(PSX,length(PX),L,NS);
   P =MakeP(PSX,PSD,length(PX),L,NS);
   MSE(K) = mean(PSD(1:NS).^2) + W'*R*W - 2*P'*W;

   %Shift PX to right
  PX(2:L+1) = PX(1:L);
   PSD(2:NS+L+1) = PSD(1:NS+L);
   PSX(2:L+NS+1) = PSX(1:L+NS);

end
```

```
% Ploting Result

subplot(2,2,2)
plot(X)
title('Recovered Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,4)
plot(MSE)
title('Mean Squared Error')
xlabel('Iterations/Time')
ylabel('Amplitude')
gtext('Fixed Step Size LMS Algorithm')
gtext('MU = 0.1 and 15% Noise by amplitude')
```

8.2 Fixed Step Size LMS Algorithm – Noise as a reference Signal

```
% Fixed step size LMS algorithm, Noise as a reference

% Y      = Filter output
% B(K+1) = B(K) + 2*MU*E*X(K)/((L+1)*SIG)
% X      = Noise vector, Input sent and Output
returned
% N      = Number of data in input X
% D      = Desired signal vector
% W      = Adaptive coefficients of Lth order FIR
filter
% L      = Order of adaptive system
% MU     = Convergence parameter
% SIG    = Input signal power estimate
% AL     = Forgetting factor alfa.
% PX     = Vector that retain the past input.
% NS   = Number of samples per cycle

N=500;
MU=0.08;
L=6;
SIG=0.005;
AL=0;%0.01;
IERROR=0;
NS=20; % Number of sample per cycle
for K=1:N
   F=2*pi*K/NS;
  SI(K,1)=sin(F);
```

```
   X(K,1)=0.1*(sprand(12357)-0.5) ;%SPRAND - Generate
uniformly distributed random numbers
   D(K,1)= 4*X(K,1)+ SI(K,1);
end

subplot(2,2,1)
plot(SI)
title('FIG: A       Original Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,3)
plot(D)
hold on
plot(2*X,'r')
title('FIG: B   Noise contaminated Signal and
Reference Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

disp('Executing LMS')
W=ones(L+1,1);
PX=zeros(L+1,1);
MSE = zeros(N,1);
PSD=zeros(NS+L+1,1);
PSX=zeros(NS+L+1,1);
IERROR1 = 5;

for K= 1 : N
  PX(1)=X(K,1);
  PSD(1)=D(K,1);
  PSX(1) = X(K,1);
  Y(K,1)=  PX'*W;

   %calculate equation (4.1.1)
   if(abs(X(K))) > 1E10
      disp('output is too large')
      X(K)
      K
      return
   end

   E(K,1) =D(K,1)-Y(K,1);                          %
calculate the equation (4.1.2)
   SIG = AL*(PX(1)*PX(1)) + (1-AL)*SIG; % calculate
the equation (4.1.8)
   TMP=2*MU/((L+1)*SIG);
```

```
   W=W+TMP*E(K,1)*PX;              % calculate W(n+1)
= W(n) + TMP*E*X(n) equation (4.1.9)

   % Calculate MSE
   R =MakeR(PSX,length(PX),L,NS);
   P =MakeP(PSX,PSD,length(PX),L,NS);
   MSE(K) = mean(PSD(1:NS).^2) + W'*R*W - 2*P'*W;

   %Shift PX to right
   PX(2:L+1) = PX(1:L);
   PSD(2:NS+L+1) = PSD(1:NS+L);
   PSX(2:L+NS+1) = PSX(1:L+NS);

end

% Ploting Result

subplot(2,2,2)
plot(E)
title('FIG: C   Recovered Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,4)
plot(MSE)
title('FIG: D    Mean Squared Error')
xlabel('Iterations/Time')
ylabel('Amplitude')
gtext('Fixed Step Size LMS Algorithm')
gtext('FIG 4.1.12     LMS Algorithm with noise as a
refence signal, MU = 0.08 and Noise = -14 dB
23')
```


8.3 Variable Step Size LMS Algorithm

```
% Variable Step Size LMS Algorithm
% Y=VSSLMS(X,N,D,B,L,MU,SIG,AL,PX,IERROR)
% subroutine for Variable Step-Size LMS algorithm and
implements the equation
% B(K+1)= B(K) + MU*E*X(K)
% X      = Data vector, Input sent and Output returned
% PX     = Vector that retain the past input.
% N      = Number of data in input X
% D      = Desired signal vector
```

```
% W       = Adaptive coefficients of Lth order FIR
filter
% L       = Order of adaptive system
% MUMIN   = Minimum limit of MU
% MU      = Convergence parameter
% MUMAX   = Maximu limit of MU
% ALFA    = Forgetting factor alfa.
% GAMA    = Control the MU in conjuction with ALFA
usually small
% NS     = Number of samples per cycle
% In this code out put replaces the input

disp('Executing VSSLMS')
N = 500;
NS = 20;
MU = 0.001;
MUMIN = 0.01;

L = 8;
ALFA = 0.97;
GAMA = 5E-2;

for K=1:N
   F=2*pi*K/NS;
   SI(K,1)=sin(F+pi/3);
   D(K,1)=0.5*(sprand(12357)-0.5) + SI(K);
   %SPRAND - Generate uniformly distributed random
numbers
   X(K,1)=0.1*sin(F);
end

subplot(2,2,1)
plot(SI)
title('Original Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,3)
plot(D)
hold on
plot(X,'r')
title('Noise contaminated and Reference Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')


W = zeros(L+1,1);
```

```
PX = zeros(L+1,1);
MSE1=zeros(N,1);
PSD=zeros(NS+L+1,1);
PSX=zeros(NS+L+1,1);

for K= 1 :N

  PX(1)=X(K);
  PSD(1)=D(K);
  PSX(1) = X(K);

   X(K)=  PX'*W;
   if(abs(X(K))) > 1E10
      disp('Output is too large')
      X(K)
      K
      return
   end

   E =D(K)-X(K);
   W=W + 2*MU*E*PX;

   % Calculate MSE
   R =MakeR(PSX,length(PX),L,NS);
   P =MakeP(PSX,PSD,length(PX),L,NS);
   MSE(K) = mean(PSD(1:NS).^2) + W'*R*W - 2*P'*W;

   %Shift PX to right
   PX(2:L+1) = PX(1:L);
   PSD(2:NS+L+1) = PSD(1:NS+L);
   PSX(2:NS+L+1) = PSX(1:NS+L);

   MU = ALFA*MU + GAMA*E*E;
   MUMAX = 2/(3*trace(R));
   if MU<MUMIN
     MU=MUMIN;
   elseif MU > MUMAX
      MU = MUMAX;
   end

end

subplot(2,2,2)
plot(X)
title('Recovered Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')
```

```
subplot(2,2,4)
plot(MSE)
title('Mean Squared Error')
xlabel('Iterations/Time')
ylabel('Amplitude')
gtext('Variable Step Size LMS Algorithm')
gtext('50% Noise by Amlitude, ALFA=0.97 and Gama = 5E-
2')
```

8.3 Robust Variable Step Size

```
%Robust Variable Step Size LMS Algorithm
% subroutine for Robust VSSLMS algorithm and
implements the equation
% B(K+1) = B(K) + MU*E*X(K)
% X       = Data vector, Input sent and Output returned
% PX      = Vector that retain the past input.
% N       = Number of data in input X
% D       = Desired signal vector
% W       = Adaptive coefficients of Lth order FIR
filter
% L       = Order of adaptive system
% MUMIN   = Minimum limit of MU
% MU      = Convergence parameter
% MUMAX   = Maximu limit of MU
% ALFA    = Forgetting factor alfa.
% GAMA    = Control the MU in conjuction with ALFA
usually small
% BETA    = Positive constant control the averaging
time constant
% NS    = Number of samples per cycle
% In this code output replaces the input X

disp('Executing LMS')

N = 500;
MU = 0.01;
MUMIN = 0.00001;
MUMAX = 2.0;
```

```
L = 8;
NS=20;
ALFA = 0.97;
GAMA = 0.1;  %Control the convergence time and level of
misadjustment of algorithm
BETA = 0.9;    %Governs the averaging time constant

for K=1:N
   F=2*pi*K/NS;
   SI(K,1)=sin(F+pi/3);
   D(K,1)=(sprand(12357)-0.5) + SI(K);
   %SPRAND - Generate uniformly distributed random
numbers
   X(K,1)=0.1*sin(F);
end

subplot(2,2,1)
plot(SI)
title('Original Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,3)
plot(D)
hold on
plot(X,'r')
title('Noise contaminated and Reference Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

W = zeros(L+1,1);
PX = zeros(L+1,1);
MSE1=zeros(N,1);
PSD=zeros(NS+L+1,1);
PSX=zeros(NS+L+1,1);

P1=0;
E=0;
for K= 1 : N

  PX(1)=X(K);
  PSD(1)=D(K);
  PSX(1) = X(K);

   X(K)=  PX'*W;
   if(abs(X(K))) > 1E10
       disp('Error is too large')
```

```
        X(K)
        K
        return
    end
    E_OLD = E;
    E =D(K)-X(K);
    W=W + 2*MU*E*PX;

    % Calculate MSE
    R =MakeR(PSX,length(PX),L,NS);
    P =MakeP(PSX,PSD,length(PX),L,NS);
    MSE(K) = mean(PSD(1:NS).^2) + W'*R*W - 2*P'*W;

    %Shift PX to right
    PX(2:L+1) = PX(1:L);
    PSD(2:NS+L+1) = PSD(1:NS+L);
    PSX(2:NS+L+1) = PSX(1:NS+L);

    MU = ALFA*MU + GAMA*P1*P1;
    P1 = BETA*P1 + (1-BETA)*E_OLD*E;
    %MUMAX = 2/(3*trace(R))
    if MU<MUMIN
      MU=MUMIN;
    elseif MU > MUMAX
       MU = MUMAX;
    end
end

subplot(2,2,2)
plot(X)
title('Recovered Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,4)
plot(MSE)
title('Mean Squared Error')
xlabel('Iterations/Time')
ylabel('Amplitude')
gtext('Robust Variable Step Size LMS Algorithm')
gtext('100% Noise by Amlitude, ALFA=0.97, GAMA = 0.1
and BETA=0.9')
```

8.5 Common Routine for LMS Algorithms

```
function P = MakeP(PSX,PSD,len,L,NS)
% This fuction make the Cross Correlation matrix P
P = zeros(len,1);
for i = 1 : L+1
   for s = 1 : NS
       P(i) = P(i) + PSD(s)*PSX(s+i-1);
    end
end
P = P/NS;
```

```
function R = MakeR(PSX,len,L,NS)
% This fuction make the Autocorrelation matrix R

R = zeros(len);

    for j = 1: L+1
        for i = j : L+1

            for s = j : j+NS-1
               R(j,i) = R(j,i) + PSX(s)*PSX(s+i-j);
            end
             R(i,j)=R(j,i);
        end
    end

R = R/NS;
```

9 Appendix B – RLS Algorithm Program Listing

9.1 RLS Algorithm Version I

```
% RLS algorithm

N=499;
LAMDA=1;
DELTA = 0.01;
L=8;
NS=20;
P = eye(L)/DELTA;


for K=1:N
   F = 2*pi*K/NS;
  SI(1,K) = sin(F+pi/3);

   X(1,K) = 0.1*sin(F);
end
Noise = randn(1,N);
D = Noise + SI;

W(1,:)=zeros(1,L);
NoisePad=[zeros(1,L-1) Noise];


for n= 1:N;

   m=n+L-1;
   NoiseBlock=NoisePad(m-L+1:1:m)';
   Y(n) =W(n,:)*NoiseBlock;
   E(n)=D(n)-Y(n);
   Kn = P'*NoiseBlock/(LAMDA +
NoiseBlock'*P*NoiseBlock);
   P = (P - Kn*NoiseBlock'*P)/LAMDA;
   W(n+1,:) = W(n,:) + Kn'*E(n) ;


end
subplot(2,2,1)
plot(SI)
title('Original Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,2)
```

```
plot((Noise - Y).^2,'m')
title('Mean Squared Error')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,3)
plot(D)
title('Noise Contaminated Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,4)
plot(E)
title('Recovered Output')
xlabel('Iterations/Time')
ylabel('Amplitude')

gtext('RLS ALGORITHM VERSION I')
```

9.2 RLS Algorithm Version II

```
% RLS algorithm
N=499;
LAMDA=1;
DELTA = 0.01;
L=4;
NS=20;
P = eye(L)/DELTA;


for K=1:N
    F = 2*pi*K/NS;
   SI(1,K) = sin(F+pi/3);

    X(1,K) = 0.1*sin(F);
end
Noise = 100*randn(1,N);
D = Noise + SI;

W(1,:)=zeros(1,L);
NoisePad=[zeros(1,L-1) Noise];


for n= 1:N;
```

```
    m=n+L-1;
    NoiseBlock=NoisePad(m-L+1:1:m)';
    Y(n) =W(n,:)*NoiseBlock;
    E(n)=D(n)-Y(n);
   Kn = P*NoiseBlock/(LAMDA +NoiseBlock'*P*NoiseBlock);
    P = (P - Kn*NoiseBlock'*P)/LAMDA;
  W(n+1,:) = W(n,:) + Kn'*E(n) ;


end
subplot(2,2,1)
plot(SI)
title('Original Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,2)
plot((Noise - Y).^2,'m')
title('Mean Squared Error')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,3)
plot(D)
title('Noise Contaminated Signal')
xlabel('Iterations/Time')
ylabel('Amplitude')

subplot(2,2,4)
plot(E)
title('Recovered Output')
xlabel('Iterations/Time')
ylabel('Amplitude')
gtext('RLS ALGORITHM VERSION I')
```

10 Appendix C – LMS Algorithm ADSP 2181 Program Listing

```
.module/RAM/ABS=0 MEng_Program;
{LMS Algorithm Implementation}
{******** Constant Declaration ***********}

{Memory Mapped ADSP-2181 Control Registers}

.const IDMA                 =0x3fe0;
.const BDMA_BIAD            =0x3fe1;
.const BDMA_BEAD            =0x3fe2;
.const BDMA_BDMA_Ctrl             =0x3fe3;
.const BDMA_BWCOUNT               =0x3fe4;
.const PFDATA                     =0x3fe5;
.const PFTYPE                     =0x3fe6;
.const SPORT1_Autobuf             =0x3fef;
.const SPORT1_RFSDIV              =0x3ff0;
.const SPORT1_SCLKDIV             =0x3ff1;
.const SPORT1_Control_Reg=0x3ff2;
.const SPORT0_Autobuf             =0x3ff3;
.const SPORT0_RFSDIV              =0x3ff4;
.const SPORT0_SCLKDIV             =0x3ff5;
.const SPORT0_Control_Reg=0x3ff6;
.const SPORT0_TX_Channels0        =0x3ff7;
.const SPORT0_TX_Channels1        =0x3ff8;
.const SPORT0_RX_Channels0        =0x3ff9;
.const SPORT0_RX_Channels1        =0x3ffa;
.const TSCALE                     =0x3ffb;
.const TCOUNT                     =0x3ffc;
.const TPERIOD                    =0x3ffd;
.const DM_Wait_Reg          =0x3ffe;
.const System_Control_Reg   =0x3fff;

{****** Variable and Buffer Declaration *******}

.var/dm/ram/circ rx_buf[3];
.var/dm/ram/circ tx_buf[3];
.var/dm/ram/circ init_cmds[13];
.var/dm                stat_flag;

.var/dm/ram Mu;
.var/dm/ram nk[4];
.var/dm/ram Wk[4];
.var/dm/ram Xk[4];
.var/dm ek;
.var/dm        Yk;
```

```
{****** Variable and Buffer Initialization ******}

.init Mu : 1024;
.init Wk : 0,0,0,0;
.init Xk : 0,0,0,0;
.init nk : 0,0,0,0;
.init ek : 0;
.init Yk : 0;

.init tx_buf: 0xc000, 0x0000, 0x0000;
.init init_cmds:        0xc003,
                        0xc103,
                        0xc288,
                        0xc388,
                        0xc488,
                        0xc588,
                        0xc680,
                        0xc780,
                        0xc85b,
                        0xc909,
                        0xca00,
                        0xcc40,
                        0xcd00;
```



```
{****** Interrupt Vector Table ******}

        jump start;     {Location 0000 : reset}
        rti;
        rti;
        rti;

        rti;            {Location 0004 : IRQ2}
        rti;
        rti;
        rti;

        rti;            {Location 0008 : IRQL1}
        rti;
        rti;
        rti;

        rti;            {Location 000c : IRQL0}
        rti;
        rti;
```

```
        rti;

ar = dm(stat_flag);     {Location 0010 : SPORT0 tx}
ar = pass ar;
if eq rti;
jump next_cmd;

jump input_samples;  {Location 0014 : SPORT0 rx}
        rti;
        rti;
        rti;

        rti;             {Location 0018 : IRQE}
        rti;
        rti;
        rti;

        rti;             {Location 001c : BDMA}
        rti;
        rti;
        rti;

        rti;             {Location 0020 : SPORT1 tx or IRQ1}
        rti;
        rti;
        rti;

        rti;             {Location 0024 : SPORT1 rx or IRQ0}
        rti;
        rti;
        rti;

        rti;             {Location 0028 : timer}
        rti;
        rti;
        rti;

        rti;             {Location 002c : power down}
        rti;
        rti;
        rti;


{****** ADSP 2181 Initialization ******}

start:
```

```
    i0 = ^rx_buf;          {Address pointer to start of receive buffer}
    l0 = %rx_buf;          {Length register to size of receive buffer}
    i1 = ^tx_buf;          {Address pointer to start of transmit buffer}
    l1 = %tx_buf;          {Length register to size of transmit buffer}
    i3 = ^init_cmds;
    l3 = %init_cmds;
    m1 = 1;

    i4 = ^Wk;              {Coefficient buffer}
    m4 = 1;
    l4 = %Wk;

    m5 = 0;

    i2 = ^nk;              {Noise}
    m2 = 1;
    l2 = %nk;

{****** Serial port 0 Set up ******}

    ax0 = b#0000001010000111;
    dm(SPORT0_Autobuf) = ax0;

    ax0 = 0;
    dm(SPORT0_RFSDIV) = ax0;
    dm(SPORT0_SCLKDIV) = ax0;
    ax0 = b#1000011000001111;
    dm(SPORT0_Control_Reg) = ax0;

    ax0 = b#0000000000000111;
    dm(SPORT0_TX_Channels0) = ax0;
    ax0 = b#0000000000000111;
    dm(SPORT0_TX_Channels1) = ax0;
    ax0 = b#0000000000000111;
    dm(SPORT0_RX_Channels0) = ax0;
    ax0 = b#0000000000000111;
    dm(SPORT0_RX_Channels1) = ax0;


{****** Serial port 1 Set up ******}

    ax0 = 0;
    dm(SPORT1_Autobuf) = ax0;        {Auto buffer disabled}
    dm(SPORT1_RFSDIV) = ax0;         {RFSDIV not used}
    dm(SPORT1_SCLKDIV) = ax0;             {SCLKDIV not used}
    dm(SPORT1_Control_Reg) = ax0;    {ctrl fuction disabled}
```

```
{****** Timer set up ******}

        ax0 = 0;
        dm(TSCALE) = ax0;           {Timer not being used}
        dm(TCOUNT) = ax0;
        dm(TPERIOD) = ax0;


{****** System and memory set up ******}

        ax0 = b#0000000000000000;
        dm(DM_Wait_Reg) = ax0;

        ax0 = b#0001000000000000; {Enable SPORT0}
        dm(System_Control_Reg) = ax0;

        ifc = b#00000011111111;
        nop;

        icntl = b#00000;
        mstat = b#1000000;


{****** AD1847 Codec initialization ******}

        ax0 = 1;
        dm(stat_flag) = ax0;   {Clear flag}
        imask = b#0001000000;       {Enable transmit interrupt}
        ax0 = dm(i1,m1);            {Start interrupt}
        tx0 = ax0;

check_init:
        ax0 = dm(stat_flag);   {Wait for entire init buffer}
        af = pass ax0;         {to be sent to the codec}
        if ne jump check_init;

        ay0 = 2;

check_acih:
        ax0 = dm(rx_buf);      {Once initialized, wait }
        ar = ax0 and ay0;      {for codec to come out }
        if eq jump check_acih;      {of autocalibration }

check_acil:
```

```
        ax0 = dm(rx_buf);       {Once initialized, wait for}
        ar = ax0 and ay0;       {codec to come out of}
        if ne jump check_acil;  {autocalibration}

        idle;

        ay0 = 0xbf3f;           {Unmute left DAC}
        ax0 = dm(init_cmds + 6);
        ar = ax0 and ay0;
        dm(tx_buf) = ar;
        idle;

        ax0 = dm(init_cmds + 7); {Unmute right DAC}
        ar = ax0 and ay0;
        dm(tx_buf) = ar;
        idle;

        ax0 = 0xc901;           {Clear autocalibration request}
        dm(tx_buf) = ax0;
        idle;

        ax1 = 0x8000;           {Control word to clear}
        dm(tx_buf) = ax1;       {over-range flags}

        ifc = b#00000011111111;     {Clear any pending interrupt}
        nop;

        imask = b#0000100000;       {Enable rx0 interrupt}


{****** Wait for interrupt and loop forvevr ******}

talkthru:
        idle;

        jump talkthru;


{***************** Interrupt Service Routine *****************}

{****** Received Interrupt Used for loopback ******}

input_samples:
        ena sec_reg;            {Use shadow register Bank}
```

```
ax1 = dm(rx_buf + 1); {Get data from codec}
ay1 = dm(rx_buf + 2); {Get noise from codec,small cable noise}

{******* Code to be written to process input samples}

{dm(tx_buf + 1)=ax1;}
{dm(tx_buf + 2)=ay1;}

dm(i2,m2) = ay1;        {Fill the noise buffer}
ar = ax1 + ay1;
dm(tx_buf +2 ) = ar;
DM(Yk) = ar;            {Noise contaminated signal }

        {Calculation of equation 4.1.1}
                CNTR = 3;
        FIR:    mr = 0, mx0 = dm(i2,m2), my0 = pm(i4,m4);
                do SOP until ce;
        SOP:    mr = mr + mx0*my0(SS), mx0 = dm(i2,m2), my0 =
                 pm(i4,m4);
                mr = mr + mx0*my0(RND);
                if mv sat mr;
                dm(ek) = mr1;           {Estimated noise}
                {dm(tx_buf + 1) = mr1;}

        ay0 = mr1;
        ax0 = dm(Yk);
        ar = ax0 - ay0;         {calculation of equation 4.1.2}
        ay0 = ar;
        ax0 = ar;
        ar = ax0 + ay0;                 {Signal Amplified}
        dm(tx_buf + 1) = ar;    {Recovered Signal and rest of the code
                                 update filter coefficient}

        mr = 0, mx0 =ar;
        my0 = dm(Mu);
        mr = mx0*my0(RND);
        ax0 = mr1;
        ay0 = mr1;
        ar = ax0 + ay0;
        my0 = ar;               {calculated 2*Mu*er}

                CNTR = 3;
                do CAL until ce;
                mr = 0, ay0 = pm(i4,m5), mx0 = dm(i2,m2);
                mr1 = ay0;
```

```
                mr = mr + mx0*my0(RND);
          CAL:  pm(i4,m4) = mr1;


      rti;


{****** Transmit Interrupt used for Codec initialization ******}

next_cmd:
      ena sec_reg;
      ax0 = dm(i3,m1);          {Fetch next control word and}
      dm(tx_buf) = ax0;         {place in transmit slot 0}
      ax0 = i3;
      ay0 = ^init_cmds;
      ar = ax0 - ay0;
      if gt rti;                {rti if more control words else}
      ax0 = 0x8000;             {set done flag and remove MCE}
      dm(tx_buf) = ax0;         {if done with init.}
      ax0 = 0;
      dm(stat_flag) = ax0;      {reset status flag}
      rti;


.endmod;
```