Linear Predictive Coding 3

3.1 OVERVIEW

Digital Signal Processing Applications Using the ADSP-2100 Family, Volume 1, contains a chapter about Linear Predictive Coding. That chapter (Chapter 10) discusses the following topics:

- LPC theory
- Correlation functions (auto-correlation and cross-correlation)
- Levinson-Durbin Recursion
- Pitch Detection

Also, the chapter includes program listings and subroutines for the following applications:

- Correlation subroutine
- LPC coefficient calculation
- Pitch Detection
- LPC synthesis

After Volume 1 was published, additional application programs for 7.8 kbits/s and 2.4 kbits/s LPCwere developed to build on the usefulness of the information included in that text. This chapter introduces those applications.

3.2 LINEAR PREDICTION

Linear Predictive Coding is a speech coding technique that models the human vocal tract. According to the model, the human body produces two basic types of sounds: voiced and unvoiced. If the vocal folds vibrate when air from the lungs is forced through them, voiced sound is produced. Unvoiced sound is produced by the tongue, lips, teeth, and mouth.

For voiced and unvoiced sounds, the vocal tract can be modeled as a series of cylinders with different radii and different amounts of energy at the boundaries between the cylinders. Mathematically, you can represent this model as a linear filter excited by a fundamental frequency (voiced sound) or random noise (unvoiced sound).

3 Linear Predictive Coding

The goal of linear predictive analysis is to derive the necessary parameters for reconstructing the sound: voiced / unvoiced decision, fundamental frequency, system gain, and the coefficients that describe the filter. The objective of Linear Predictive Coding (LPC) is to predict the next output of the system based on previous outputs and inputs. This is an effective coding technique because speech is a highly correlated signal when considered during a short interval of time (frame length). That is, given a sequence of speech samples, subsequent speech samples can be predicted with a minimum of error over a short period of time.

LPC relies on a technique that uses a linear combination of previous outputs, past inputs, and an input excitation. The filter equation or speech output, s(n), can be represented as:

$$s(n) = -\sum_{k=1} a_k s(n-k) + G\sum_{j=0} b_j u(n-j), b0 = 1$$

where s() is the speech sequence and u() is the excitation sequence.

The resulting synthesis filter has the following domain representation:

$$H(z) = \frac{\tilde{s}(z)}{u(z)} = G * \frac{\quad}{\left(1 + a_j z^{-i}\right)}$$

To synthesize sound with the model described above, you must specify the complete parameter set for the synthesis filter. Table 3.1 defines these parameters.

Parameter	*Symbol*
Filter Coefficients	a_k, b_k
Gain	G
Excitation	
Voiced / Unvoiced	1 / 0
Pitch Period	P

Table 3.1 Parameter Set For The Sound Synthesis Model

3.3 7.8 kbits/s LPC

The 7.8 kbits/s LPC (which is of higher speech quality) is derived by using a data frame size of 180 samples sampled at 8 kSa/s. For each frame, the analysis section generates ten 16-bit coefficients and an additional 16-bit word containing the gain and pitch period. Therefore, the bit rate is derived by:

$$\text{Bit Rate} = \frac{(11 \text{ words}) * (16 \text{ bits / word})}{(180 \text{ samples})} * (125 \,\mu\text{s / sample}) = 7822 \text{ bits / s } (7.8 \text{ kbits / s})$$

Listing 3.1 shows the 7.8 kbits/s LPC Routine. This routine calls several subroutines listed in Section 3.4, "*LPC Subroutines.*"

```
.module/boot=3/abs=0      lpc7k8_through;
{   LPC7k8.DSP - talk through, encoding and decoding using LPC.
Input: Speech samples from microphone (using autobuffering) via sport0
Ouput: Speech samples to speeaker (using autobuffering) via sport0
Modules used:
   - pre_emphasize_speech          (PREEMP.DSP)
   - gain_calculation              (GAIN.DSP)
   - autocorrelation_of_speech     (AUTOCOR.DSP)
   - durbin_double/single          (DURBIN2.DSP/DURBIN.DSP)
   - pitch_detection               (PITCH.DSP)
   - lpc_sync_synth                (SSYNTH.DSP)
   - de_emphasize_speech           (DEEMP.DSP)
   - constant header               (LPC.H)
Description:
   This program implements a shell to demonstrate the LPC algorithm on an EZ-LAB
   board. Speech is autobuffered in from the codec, compressed, decompressed and
   autobuffered back out to the codec, providing a "talk-through" program.
NOTE: The framesyncs of sport0 (TFS0 & RFS0) SHOULD NOT be tied together EXTERNALY!!
   (On the EZ-LAB's serial connector it is pins 4 & 5)
}

{include constant definitions}
#include "lpc.h";

{Buffers used by autobuffering, input swaps between analys_buf & receive_buf,
   whereas output swaps between synth_buf & transit_buf}
.var/dm/ram/circ  analys_buf[FRAME_LENGTH];
.var/dm/ram/circ  receive_buf[FRAME_LENGTH];

{The LPC-parameters are stored in trans_line to "simulate" transmission}
.var/dm/ram       trans_line[WORDS_PR_LPCFRAME];
```

(listing continues on next page)

```
{Pointers to the buffers that are NOT currently being used by autobuffering}
.var/dm/ram       p2_analysis,p2_synth;

{Intermediate variables}
.var/pm/ram       autocor_speech[FRAME_LENGTH];
.var/dm/ram/circ  k[N];
.var/dm/ram       pitch, gain;
.var/dm/ram       lpc_flag;
.external         pre_emph;
.external         calc_gain;
.external         a_correlate;
.external         levinson;
.external         detect_pitch;
.external         clear_filter;
.external         synthesis;
.external         de_emph;

{load interrupt vectors}
jump start_test; nop; nop; nop;                 {reset interrupt}
   rti; nop; nop; nop;
   rti; rti; nop; nop;                          {sport0 transmit}
   call rcv_ir; rti; nop; nop;                  {sport0 receive}
   rti; nop; nop; nop;
   rti; nop; nop; nop;
   rti; nop; nop; nop;

change_demo:
   gd: if not flag_in jump gd;
   ay0 = 0x0338;                                {set bootforce bit}
   dm(0x3fff) = ay0;
   nop;
   rts;

start_test:
   {configure sports etc.}
   ax0 = 0x0000;
   dm(0x3ffe) = ax0;                            {dm waits = 0}
   ax0 = 0x6327;
   dm(0x3ff6) = ax0;                            {set sport0 control reg}
   ax0 = 2;
   dm(0x3ff5) = ax0;                            {set sclkfreq to 2.048 Mhz}
   ax0 = 255;
   dm(0x3ff4) = ax0;                            {set rclkfreq to 64 khz}

{Default register values, these values can always be asumed, and must
   be reset if altered}
   l0 = 0; l1 = 0; l2 = 0; l3 = 0;
   l4 = 0; l5 = 0; l6 = 0; l7 = 0;
```

```
{DEDICATED REGISTERS. These registers must NOT be altered by any routine at any
   time! (used by autobuffering)}
   m0 = 0;  m4 = 0;
   m1 = 1;  m5 = 1;
   i3 = ^receive_buf;
   l3 = %receive_buf;

{Setup and clear intermediate buffers}
   i6 = ^analys_buf;
   l6 = 0; dm(p2_analysis) = i6;

   ena sec_reg;
   ax0 = FRAME_LENGTH;
   af  = pass ax0;
   dis sec_reg;

   ax0 = 0;
   dm(lpc_flag) = ax0;

{clear the synthesis filter}
   call clear_filter;

{enable sport0}
   ax0 = 0x1038;
   dm(0x3fff) = ax0;

{enable sport0}
   icntl = b#00111;
   imask = b#001000;                                  {Enable receive}

   wait: idle;


   if not flag_in call change_demo;
   ax0 = dm(lpc_flag);
   ar  = pass ax0;
   if eq jump wait;
   ax0 = 0;
   dm(lpc_flag) = ax0;

{Parameters: i0 = p2_analysis (-> speech)
 Returns:    filtered speech}
   i0 = dm(p2_analysis);
   l0 = 0;
   call pre_emph;
```

(listing continues on next page)

```
{Parameters: i0  = p2_analysis (-> speech)
 Returns:    sr1 = gain}
   i0 = dm(p2_analysis);
   l0 = 0;
   call calc_gain;
   dm(gain) = sr1;

{Parameters: i0 = p2_analysis (-> speech)
 Returns:    autocor_speech[]}
   i0 = dm(p2_analysis); l0 = 0;
   i6 = ^autocor_speech; l6 = 0;
   call a_correlate;

{Parameters: i4 -> autocor_speech[])
 Returns:    i0 -> k[]}
   i4 = ^autocor_speech; l4 = 0;
   i0 = ^k; l0 = 0;
   call levinson;

{Parameters: i0 -> k[], i6 -> autocor_speech[]
 Returns:    si =  pitch}
   i0 = ^k; l0 = 0;
   i6 = ^autocor_speech; l6 = 0;
   call detect_pitch;
   dm(pitch) = si;

{TRANSMISSION LINE}

{Parameters: ax1 =  pitch, mx1 = gain, i0  -> k[]
 Returns:    i2  -> speech}
   i0 = ^k; l0 = 0;
   i1 = ^k + N - 1; l1 = 0;              {store k's in revers order - }
   cntr = 5;                             {N/2}{required by lattice routine}
   m2 = -1;
   do reverse_ks until ce;
         ay0 = dm(i0,m0);
         ay1 = dm(i1,m0);
         dm(i0,m1) = ay1;
   reverse_ks: dm(i1,m2) = ay0;
   ax1 = dm(pitch);
   mx1 = dm(gain);
   i1 = ^k; l1 = N;
   i2 = dm(p2_analysis); l2 = 0;
   call synthesis;

{Parameters: i0 = p2_analysis (-> speech)
 Returns:    filtered speech}
   i0 = dm(p2_analysis); l0 = 0;
   call de_emph;

jump wait;
```

```
{End of main routine}

{Autobuffering interrupt routines}
trns_ir:
rts;
rcv_ir:
   ena sec_reg;
   ax0 = dm(i3,m0);
   tx0 = ax0;
   ax0 = rx0;
   dm(i3,m1) = ax0;
   af = af - 1;
   if gt jump no_lpc;                              {switch pointers}
   ay0 = i3;
   ay1 = dm(p2_analysis);
   i3 = ay1; l3 = %receive_buf;
   dm(p2_analysis) = ay0;
   ax0 = FRAME_LENGTH;
   af  = pass ax0;
   ax0 = 1;
   dm(lpc_flag) = ax0;
   no_lpc:
   dis sec_reg;
   rts;

{END of main code}

.endmod;
```

Listing 3.1 7.8 kbits/s LPC Routine

3.4 2.4 kbits/s LPC

The speech quality of 2.4 kbits/s LPC is somewhat deminished from the quality of 7.8 kbits/s LPC, but the compression ration is much higher. Although the bit rate is more than 3 times slower than 7.8 kbits/s LPC, the quality is considered acceptable for most applications. The lower bit rate is achieved by reducing the number of bits per frame from 176 (11 words X 16 bits/word) unquantized bits to 54 quantized bits. Therefore, the bit rate for the 2.4 kbits/s LPC is derived by:

$$\text{Bit Rate} = \frac{(54\ \text{bits / frame})}{(180\ \text{samples})} * (125\ \mu\text{s / sample}) = 2400\ \text{bits / s}\ (2.4\ \text{kbits / s})$$

The quantization routines are called only by the 2.4 kbits/s version of the code (main module is LPC2K4.DSP); they are called encode.dsp and decode.dsp.

3 Linear Predictive Coding

Listing 3.2 shows the 2.4 kbits/s LPC Routine. This routine calls the subroutines listed in Section 3.4, "*LPC Subroutines.*"

```
.module/boot=4/abs=0      lpc2k4_through;
{   LPC2k4.DSP - talk through, encoding and decoding using LPC.
Input: Speech samples from microphone (using autobuffering) via sport0
Ouput:Speech samples to speeaker (using autobuffering) via sport0
Modules used:
   - pre_emphasize_speech           (PREEMP.DSP)
   - gain_calculation               (GAIN.DSP)
   - autocorrelation_of_speech      (AUTOCOR.DSP)
   - durbin_double/single           (DURBIN2.DSP/DURBIN.DSP)
   - pitch_detection                (PITCH.DSP)
   - lpc_sync_synth                 (SSYNTH.DSP)
   - de_emphasize_speech            (DEEMP.DSP)
   - constant header                (LPC.H)
Description:
   This program implements a shell to demonstrate the LPC algorithm on an EZ-LAB
   board. Speech is autobuffered in from the codec, compressed, decompressed and
   autobuffered back out to the codec, providing a "talk-through" program.
NOTE: The framesyncs of sport0 (TFS0 & RFS0) SHOULD NOT be tied together EXTERNALY!!
   (On the EZ-LAB's seriel connector it is pins 4 & 5)
}

{include constant definitions}
#include "lpc.h";

{Buffers used by autobuffering, input swaps between analys_buf & receive_buf,
   whereas output swaps between synth_buf & transit_buf}
.var/dm/ram/circ  analys_buf[FRAME_LENGTH];
.var/dm/ram/circ  receive_buf[FRAME_LENGTH];

{The LPC-parameters are stored in trans_line to "simulate" transmission}
.var/dm/ram       trans_line[WORDS_PR_LPCFRAME];

{Pointers to the buffers that are NOT currently being used by autobuffering}
 .var/dm/ram      p2_analysis,p2_synth;

{Intermediate variables}
.var/pm/ram       autocor_speech[FRAME_LENGTH];
.var/dm/ram/circ  k[N];
.var/dm/ram       pitch, gain;
.var/dm/ram       lpc_flag;


.external         pre_emph;
.external         calc_gain;
.external         a_correlate;
.external         levinson;
```

```
.external         detect_pitch;
.external         encode;
.external         decode;
.external         clear_filter;
.external         synthesis;
.external         de_emph;

{load interrupt vectors}
jump start_test; nop; nop; nop;                 {reset interrupt}
   rti; nop; nop; nop;
   rti; rti; nop; nop;                          {sport0 transmit}
   call rcv_ir; rti; nop; nop;                  {sport0 receive}
   rti; nop; nop; nop;
   rti; nop; nop; nop;
   rti; nop; nop; nop;
change_demo:
   gd: if not flag_in jump gd;
   ay0 = 0x0238;                                {set bootforce bit}
   dm(0x3fff) = ay0;
   nop;
   rts;

start_test:
   {configure sports etc.}
   ax0 = 0x0000;
   dm(0x3ffe) = ax0;                            {dm waits = 0}
   ax0 = 0x6327;
   dm(0x3ff6) = ax0;                            {set sport0 control reg}
   ax0 = 2;
   dm(0x3ff5) = ax0;                            {set sclkfreq to 2.048 Mhz}
   ax0 = 255;
   dm(0x3ff4) = ax0;                            {set rclkfreq to 64 khz}

{Default register values, these values can always be asumed, and must
   be reset if altered}
   l0 = 0; l1 = 0; l2 = 0; l3 = 0;
   l4 = 0; l5 = 0; l6 = 0; l7 = 0;

{DEDICATED REGISTERS. These registers must NOT be altered by any routine at
any time! (used by autobuffering)}
   m0 = 0;  m4 = 0;
   m1 = 1;  m5 = 1;
   i3 = ^receive_buf;
   l3 = %receive_buf;
```

(listing continues on next page)

```
{Setup and clear intermediate buffers}
   i6 = ^analys_buf;
   l6 = 0; dm(p2_analysis) = i6;

   ena sec_reg;

   ax0 = FRAME_LENGTH;
   af  = pass ax0;
   dis sec_reg;

   ax0 = 0;
   dm(lpc_flag) = ax0;

{clear the synthesis filter}
   call clear_filter;

{enable sport0}
   ax0 = 0x1038;
   dm(0x3fff) = ax0;

{enable sport0}
   icntl = b#00111;
   imask = b#001000;                        {Enable receive}


   wait: idle;


   if not flag_in call change_demo;


   ax0 = dm(lpc_flag);
   ar  = pass ax0;
   if eq jump wait;


   ax0 = 0;
   dm(lpc_flag) = ax0;

{Parameters: i0 = p2_analysis (-> speech)
 Returns:    filtered speech}
   i0 = dm(p2_analysis);
   l0 = 0;
   call pre_emph;

{Parameters: i0  = p2_analysis (-> speech)
 Returns:    sr1 = gain}
   i0 = dm(p2_analysis);
   l0 = 0;
   call calc_gain;
   dm(gain) = sr1;
```

```
{Parameters: i0 = p2_analysis (-> speech)
 Returns:    autocor_speech[]}
   i0 = dm(p2_analysis); l0 = 0;
   i6 = ^autocor_speech; l6 = 0;
   call a_correlate;

{Parameters: i4 -> autocor_speech[])
 Returns:    i0 -> k[]}
   i4 = ^autocor_speech;
   l4 = 0;
   i0 = ^k;
   l0 = 0;
   call levinson;

{Parameters: i0 -> k[],
          i6 -> autocor_speech[]
 Returns:    si =  pitch}
   i0 = ^k;
   l0 = 0;
   i6 = ^autocor_speech; l6 = 0;
   call detect_pitch;
   dm(pitch) = si;

{Parameters: i1 -> k[],
          ar = pitch,
          si = gain
Returns:  parameters encoded ar = pitch,
          si = gain}
   i1 = ^k; l1 = 0;
   ar = dm(pitch);
   si = dm(gain);
   call encode;
   dm(pitch) = ar;
   dm(gain)  = si;

{TRANSMISSION LINE}


{Parameters: i1 -> k[],
          si  = pitch,
          ax0 = gain
Returns:  k's decoded
          si  = pitch,
          ax0 = gain}
   i1  = ^k;   l1 = 0;
   si  = dm(pitch);
   ax0 = dm(gain);
   call decode;
   dm(pitch) = si;
   dm(gain)  = ax0;
```

(listing continues on next page)

```
{Parameters: ax1 =  pitch,
          mx1 =  gain,
          i0  -> k[]
Returns:  i2  -> speech}
   i0 = ^k;
   l0 = 0;
   i1 = ^k + N - 1;
   l1 = 0;                          {store k's in revers order - }
   cntr = 5;                        {N/2} {required by lattice routine}
   m2 = -1;
   do reverse_ks until ce;
         ay0 = dm(i0,m0);
         ay1 = dm(i1,m0);
         dm(i0,m1) = ay1;
   reverse_ks: dm(i1,m2) = ay0;
   ax1 = dm(pitch);
   mx1 = dm(gain);
   i1 = ^k;
   l1 = N;
   i2 = dm(p2_analysis); l2 = 0;
   call synthesis;

{Parameters: i0 = p2_analysis (-> speech)
 Returns:    filtered speech}
   i0 = dm(p2_analysis);  l0 = 0;
   call de_emph;
jump wait;


{End of main routine}

{Autobuffering interrupt routines}
trns_ir:
rts;

rcv_ir:
   ena sec_reg;
   ax0 = dm(i3,m0);
   tx0 = ax0;
   ax0 = rx0;
   dm(i3,m1) = ax0;
   af = af - 1;
   if gt jump no_lpc;
   {switch pointers}
   ay0 = i3;
   ay1 = dm(p2_analysis);
   i3 = ay1;
   l3 = %receive_buf;
   dm(p2_analysis) = ay0;
```

```
   ax0 = FRAME_LENGTH;
   af  = pass ax0;


   ax0 = 1;
   dm(lpc_flag) = ax0;
   no_lpc:


   dis sec_reg;
rts;

{END of main code}

.endmod;
```

Listing 3.2 2.4 kbits/s LPC Routine

3.5 LPC SUBROUTINES

This section contains the subroutines called by the 7.8 and 2.4 kbits/s LPC routines.

```
.module/boot=3/boot=4      autocorrelation_of_speech;
{   AUTOCOR.DSP - perform autocorrelation on input speech frame.
INPUT: i0 -> frame of speech (dm)
    l0 =  0
    i6 -> buffer for autocorrelation (pm)
    l6 =  0

OUTPUT: autocorrelation buffer filled

FUNCTIONS CALLED: None
DESCRIPTION:
   First the speech is autoscaled (IN PLACE!!), to avoid overflow.
   The autocorrelation is calculated, and normalized so r[0] = 1 (0x7fff). }

{include constant and macro definitions}
#include “lpc.h”;
.entry      a_correlate;
.external   overflow;

{.var/pm/ram/circ  copy_of_speech[FRAME_LENGTH];}
.var/dm     p2_speech;
.var/dm     p2_autocor_speech;
```

(listing continues on next page)

3 Linear Predictive Coding

```
a_correlate:

{store pointers for later use}
   dm(p2_speech) = i0;
   dm(p2_autocor_speech) = i6;

{auto scale input before correlating}
{first: detect largest exp in speech}
   {i0 -> speech)}
   cntr = FRAME_LENGTH;
   sb = -16;
   do max_speech until ce;
         si = dm(i0,m1);
   max_speech: sb = expadj si;

{adjust speech input: normalize to largest and then right shift to |AC_SHIFT|.
   (16-|AC_SHIFT|) format (done in one shift). At the same time copy to pm for
   correlation}
   i0 = dm(p2_speech);              l0 = 0;
   i5 = dm(p2_autocor_speech);      l5 = 0;
   cntr = FRAME_LENGTH;
   ax0 = sb;
   ay0 = AC_SHIFT;                  {scale down to avoid overflow (worst case)}
   ar = ay0 - ax0;                  {effective scale value}
   se = ar;
   do adj_speech until ce;
         si = dm(i0,m0);
         sr = ashift si (hi);
         pm(i5,m5) = sr1;
   adj_speech: dm(i0,m1) = sr1;

{do autocorrelation, R[i] = sum_of s[j]*s[i+j]}
{NOTE: the counter updating scheme, might cause a “acces to non-existing memory”
   in the simulator/emulator}
   i5 = dm(p2_autocor_speech); l5 = 0;               {s[i+j]}
   {i6 -> autocor_speech}                            {->R[i]}
   i2 = FRAME_LENGTH; l2 = 0;
   m2 = -1;
   cntr = FRAME_LENGTH;
   do corr_loop until ce;                            {i loop}
   i0 = dm(p2_speech); l0 = 0;                       {->s[j]}
   i4 = i5; l4 = 0;                                  {->s[i+j]} cntr=i2;
         mr=0, my0=pm(i4,m5), mx0=dm(i0,m1);
         do cor_data_loop until ce;  {j loop}
   cor_data_loop: mr=mr+mx0*my0(ss),my0=pm(i4,m5),mx0=dm(i0,m1);
         if mv call overflow;
   mx0 = dm(i2,m2), my0 = pm(i5,m5);                 {update counters: - }
                                                     {(innerloop cnt’er)—, i++)}
   corr_loop: pm(i6,m5) = mr1;                       {store R[i]}
```

```
{Normalize autocorrelation sequence}
{shift sequnece for maximum precision before division}
   i5 = dm(p2_autocor_speech); l5 = 0;
   cntr = FRAME_LENGTH - 1;
   si = pm(i5,m4);                        {R(0)}
   se = exp si (hi);
   sr = norm si (hi);
   pm(i5,m5) = sr1;                       {new R(0)}
   do sh_cor until ce;                    {shift remaining sequence accordingly}
         si = pm(i5,m4);
         sr = norm si (hi);
   sh_cor: pm(i5,m5) = sr1;

{calculate R(i)/R(0)}
   i5 = dm(p2_autocor_speech); l5 = 0;
   cntr = FRAME_LENGTH - 1;
   ax0 = pm(i5,m4);                       {ax0 = divisor = R(0)}
   ay0 = 0x7fff;
   pm(i5,m5) = ay0;                       {new R(0) = 1}
   do nrm_cor until ce;
   ay1 = pm(i5,m4);                       {ay1 = MSW of dividend }
   ay0 = 0x0000;                          {ay0 = LSW of dividend}
   divide(ax0,ay1);
   nrm_cor: pm(i5,m5) = ay0;
rts;

.endmod;
```

Listing 3.3 AUTOCOR.DSP Subroutine

3 Linear Predictive Coding

```
.module/boot=4        decode_parameters;
{  DECODE.DSP - decompresses the lpc parameters.
INPUT:
   i1  -> k  (reflection coeffs) l1 = 0
   si  = pitch
   ax0 = gain

OUTPUT:
   k's decoded inplace
   si  = pitch
   ax0 = gain

The log coded parameters are decompressed using:
     k[i] = (10^(g[i]*4)+1)/(10^(g[i]*4)-1)
}

#include "lpc.h"
.const   DELOG_ORDER = 8;
.var/pm  delog_coeffs[2*DELOG_ORDER];          {Ci lsb, Ci msb, Ci-1 lsb .....}
.init       delog_coeffs: <delog.cff>;         {scaled down by 512 = 2^9}
.const   LAR_ORDER = 16;
.var/pm  lar_coeffs[2*LAR_ORDER];              {Ci lsb, Ci msb, Ci-1 lsb .....}
.init       lar_coeffs: <dec.cff>;             {scaled down by 1024 = 2^10}
.var/dm  temp_pitch;
.var/dm  temp_gain;
.entry   decode;
.external   poly_approx;

decode:
{decode}
   {si = pitch}
   se = -9;
   sr = lshift si (lo);
   dm(temp_pitch) = sr0;

         {ax0 = gain}
         sr1 = ax0;
         ar  = pass ax0;
         if eq jump zero_gain;
            my0 = 0x0000;                 {gain lsb}
            my1 = ax0;                    {gain msb}
            ax0 = DELOG_ORDER - 1;
            i6  = ^delog_coeffs; l6 = 0;
            call poly_approx;             {log10 function}
            si = mx0;
         sr = lshift si by 9 (lo);        {scale up by 512, comes with the coeff's}
         sr = sr or ashift ar by 9 (hi);
         zero_gain:
         dm(temp_gain) = sr1;
```

```
         cntr = 2 {N};
         do dec_k until ce;
            my0 = 0x0000;                {k lsb}
            my1 = dm(i1,m0);             {k msb}
            ax0 = LAR_ORDER - 1;
            i6  = ^lar_coeffs; l6 = 0;
            call poly_approx;            {log area ratio function}
            si = mx0;
         sr = lshift si by 10 (lo);      {scale up by 1024}
         sr = sr or ashift ar by 10 (hi);
         dec_k: dm(i1,m1) = sr1;
   {setup return parameters}
   si  = dm(temp_pitch);
   ax0 = dm(temp_gain);
rts;

.endmod;
```

Listing 3.4 DECODE.DSP Subroutine

3 Linear Predictive Coding

```
.module/boot=3/boot=4        de_emphasize_speech;
{  DEEMP.DSP - deemphasizes a speech frame. (filters it)
INPUT:
   i0 -> frame of speech
   l0 =  0
OUTPUT: speech deemphasized
FUNCTIONS CALLED:
    None
DESCRIPTION:
   Filters speech using: H(z) = 1/(1 - 0.75)
}

{Include constant definitions}
#include “lpc.h”
.entry   de_emph;
.external   overflow;
.var/dm/ram delay;

de_emph:
   {deemphasize}
   mx0 = 0x6000;                                {a1 = 0.75}
   cntr = FRAME_LENGTH;
   do filt_speech until ce;
         mr  = 0;
         mr1 = dm(i0,m0);                       {x(n)}
         my0 = dm(delay);                       {y(n-1)}
         mr = mr + mx0*my0 (ss);
            if mv call overflow;
   dm(delay) = mr1;                             {update delay with y(n)}
   filt_speech: dm(i0,m1) = mr1;
rts;

.endmod;
```

Listing 3.5 DEEMP.DSP Subroutine

```
.module/boot=3/boot=4        durbin_single;
{  DURBIN.DSP - single precision Levinso-Durbin routine
INPUT:
   i4 -> buffer with autocorrelated speech (pm)
   l4 =  0
   i0 -> buffer for reflection coeffs
   l0 =  0
OUTPUT: reflection coeffs calculated
     mr1 = Ep (minimum total squared prediction error
FUNCTIONS CALLED:
     None
DESCRIPTON:
   The routine implements Durbins recursion method of solving a set of linear
   equations forming a Toeplitz matrix. The algorithm in C is as follows:

   Where R[] is the autocorrelation, and k[] the reflection coeffs. e[] is the
   total squared error, and a[][] is the predictor coeff matrix (since only the
   i'th and the i+1'th column is used at any one time, the matrix is implemented
   as two (a_old and a_new) columns swapping place after each iteration.


         e[0] = R[0]
         k[1] = R[1] / e[0]
         alpha[1][1] = k[1]
         e[1] = (1 - k[1]*k[1]) * e[0]

         for (i=2; i<=N; i++)
         begin
         k[i] = 0
         for (j=1; j<=i-1; j++)
         k[i] = k[i] + R[i-j] * alpha[i-1][j]
         k[i] = R[i] - k[i]
         k[i] = k[i] / e[i-1]
         alpha[i][i] = k[i]
         for (j=i-1; j>0; j++)
         alpha[i][j] = alpha[i-1][j] - k[i]*alpha[i-1][i-j]
         e[i] = (1 - k[i]*k[i]) * e[i-1]
         end
}

{Include constant definitions}
#include "lpc.h"
.entry   levinson;
.external   overflow;
.global e;
```

(listing continues on next page)

3 Linear Predictive Coding

```
.var/dm/ram i_1;
.var/dm/ram e[N+1];                 {error values}
.var/dm/ram a_new[N],a_old[N];
.var/dm/ram ap_new,ap_old;          {pointers to a_*}
.var/dm/ram p2_k_i;                 {pointer to k[i]}
.var/dm  p2_autocor_speech;

{determines the format that a-values are stored in format: (SBITS+1).(16-SBITS-1)}
.const   SBITS  =  3;
.const   NSBITS = -SBITS;

levinson:
   i1 = ^a_new; l1 = 0;
   dm(ap_new) = i1;
   i2 = ^a_old; l2 = 0;
   dm(ap_old) = i2;
   dm(p2_autocor_speech) = i4;
   i5 = ^e;     l5 = 0;
   m2 = -1;
   m6 = -1;

se = NSBITS;

   {e[0] = R[0]}
   ax0 = pm(i4,m5);
   dm(i5,m5) = ax0;

   {k[1] = R[1]/e[0]}
   {ax0 = e[0] = divisor}
   ay1 = pm(i4,m4);                 {MSW of dividend}
   ay0 = 0000;                      {LSW of dividend}
   divide(ax0,ay1);
   ar = -ay0;                       {reverse sign of k before storing}
   dm(i0,m1)  = ar;
   dm(p2_k_i) = i0;

   {a_old[1] = k[1]}
   si = ay0;
   sr = ashift si (hi);             {store in (SBITS+1).(16-SBITS-1) format}
   dm(i2,m0) = sr1;

   {e[1] = (1 - k[1]*k[1])*e[0]}
                                    {ay0 = k[1]}
   mx0 = ay0;
   my0 = ay0;
   mr0 = 0xffff;                    {mr = 1 in 1.31 format}
   mr1 = 0x7fff;
   mr = mr - mx0*my0 (ss);
```

```
                                          {ax0 = e(0)}
   my0 = ax0;
   mr = mr1 * my0 (ss);
   dm(i5,m4) = mr1;

   {for(i = 2; i <= N; i++)}
   cntr = N-1;
   ax0 = 1;                               {i-1}
   dm(i_1) = ax0;
   do pass_two until ce;

   {k[i] = 0}
         mr = 0;
{for(j = 1; j <= i-1; j++)}
         ay0 = dm(i_1);
         cntr = ay0;
         m3 = ay0;                        {i-1}
         m7 = ay0;                        {i-1}

{prepare: k[i] = k[i] + R[i-j]*a_old[j]}
         i2 = dm(ap_old);
         l2 = 0;
         i4 = dm(p2_autocor_speech);
         l4 = 0;
         modify(i4,m7);  {->R[i-1]}

         {loop}
         do calc_ks until ce;
            mx0 = pm(i4,m6);
            my0 = dm(i2,m1);
         calc_ks: mr = mr + mx0*my0 (ss);
         if mv call overflow;

         {k[i] = R[i] - k[i]}
         i4 = dm(p2_autocor_speech); l4 = 0;
   modify(i4,m7);
   modify(i4,m5);                         {->R[i]}
   si = pm(i4,m4);                        {R[i]}
   sr = ashift si (hi);                   {shift to (SBITS+1).(16-SBITS-1) format}
   ay1 = mr1;     {k[i]}
   ar = sr1 - ay1;
   if av call overflow;

{k[i] = k[i]/e[i-1]}
   i5 = ^e; l5 = 0;
   modify(i5,m7);
   ax0 = dm(i5,m5);                       {e[i-1]}
   ay1 = ar;                              {MSW of k[i]}
   ay0 = 0000;                            {LSW of k[i]}
```

(listing continues on next page)

```
{overflow check}
   si = ax0;
   sr = ashift si (hi);
   ar = sr1 - ay1; {e[i-1] - k[i]}
   if ge jump e_ok;

{call overflow;}
         si = 0x7fff;                      {sat k[i]}
         sr = ashift si (hi);

{ay1 = sr1;}
         e_ok:
         divide(ax0,ay1);
         si = ay0;
         sr = ashift si by SBITS(hi);      {shift to 1.15 format before
storing}
         i0 = dm(p2_k_i);  0 = 0;
         ay1 = sr1;
         ar  = -ay1;                       {reverse sign of k before
storing}
         dm(i0,m1) = ar;                   {k[i] store}
         dm(p2_k_i) = i0;

{a_new[i] = k[i]}
   i1 = dm(ap_new); l1 = 0;
   modify(i1,m3);                          {->a_new[i]}
   dm(i1,m2) = ay0;

{for(j = i-1; j>0; j—)}
   cntr = dm(i_1);

{prepare: a_new[j] = a_old[j] - k[i]*a_old[i-j]}
   i2 = dm(ap_old);
   l2 = 0;
   modify(i2,m3);                          {modify by j (= i-1)}
   modify(i2,m2);                          {-> a_old[j]}
   i0 = dm(ap_old);                        {-> a_old[i-j]}
   l0 = 0; mx0 = sr1;                      {k[i]}

{loop}
   do calc_as until ce;
         mr0 = 0;
         mr1 = dm(i2,m2);                  {a_old[j]}
         my0 = dm(i0,m1);                  {a_old[i-j]}
         mr  = mr - mx0*my0 (ss);
   if mv {sat mr} call overflow;

   calc_as: dm(i1,m2) = mr1;
```

```
{e[i] = (1 - k[i]*k[i]) * e[i-1]}         {ay0 = k[i]}
         mx0 = sr1;
         my0 = sr1;
         mr0 = 0xffff;                    {mr = 1 in 1.31 format}
         mr1 = 0x7fff;
         mr = mr - mx0*my0 (ss);
         if mv call overflow;

         {ax0 = e(i-1)}
         my0 = ax0;
         mr = mr1 * my0 (ss);
         dm(i5,m4) = mr1;

         {switch the a pointers}
         ax0 = dm(ap_old);
         ay0 = dm(ap_new);
         dm(ap_new) = ax0;
         dm(ap_old) = ay0;

         {i++ }
         ay0 = dm(i_1);
         ar  = ay0 + 1;
         pass_two: dm(i_1) = ar;
   rts;

.endmod;
```

Listing 3.6 DURBIN.DSP Subroutine

3 Linear Predictive Coding

```
.module/boot=3/boot=4        durbin_double;
{  DURBIN2.DSP - single precision Levinso-Durbin routine
INPUT:

   i4 -> buffer with autocorrelated speech (pm)
   l4 = 0
   i0 -> buffer for reflection coeffs
   l0 =  0

OUTPUT  reflection coeffs calculated
    mr1 = Vp minimum total squared prediction error (normalized)
FUNCTIONS CALLED:
    None
DESCRIPTON:
   The routine implements Durbins recursion method of solving a set of linear
   equations forming a Toeplitz matrix. The algorithm in C is as follows:

   Where R[] is the autocorrelation, and k[] the reflection coeffs. e[] is the
   total squared error, and a[][] is the predictor coeff matrix (Since only the
   i'th and the i+1'th column is used at any one time, the matrix is implemented
   as two (a_old and a_new) columns swapping place after each iteration.


   e[0] = R[0]
   k[1] = R[1] / e[0]
   alpha[1][1] = k[1]
   e[1] = (1 - k[1]*k[1]) * e[0]


   for (i=2; i<=N; i++)
   begin
   k[i] = 0
   for (j=1; j<=i-1; j++)
   k[i] = k[i] + R[i-j] * alpha[i-1][j] k[i] = R[i] - k[i]
   k[i] = k[i] / e[i-1]
   alpha[i][i] = k[i]
   for (j=i-1; j>0; j++)
   alpha[i][j] = alpha[i-1][j] - k[i]*alpha[i-1][i-j]
   e[i] = (1 - k[i]*k[i]) * e[i-1]
   end

In this version the alpha's (a's) are stored as 32 bit numbers.
}
```

Linear Predictive Coding 3

```
#include “lpc.h”
.entry   levinson;
.external   overflow;
.global e;
.var/dm/ram i_1;
.var/dm/ram e[N+1];                          {error values}
.var/dm/ram a_new[2*N],a_old[2*N];           {msw0, lsw0, msw1, lsw1,.....}
.var/dm/ram ap_new,ap_old;                   {pointers to a_*}
.var/dm/ram p2_k_i;                          {pointer to k[i]}
.var/dm  p2_autocor_speech;

{determines the format that a-values are stored in format: (SBITS+1).(32-SBITS-1)}
.const   SBITS  = 4;
.const   NSBITS = -SBITS;

levinson:
   i1 = ^a_new; l1 = 0;
   dm(ap_new) = i1;
   i2 = ^a_old; l2 = 0;
   dm(ap_old) = i2; dm(p2_autocor_speech) = i4;
   i5 = ^e; l5 = 0;
   m2 = -1;
   m6 = -1;
   se = NSBITS;

{e[0] = R[0] }
   ax0 = pm(i4,m5);
   dm(i5,m5) = ax0;

{k[1] = R[1]/e[0]}
{ax0 = e[0] = divisor}
   ay1 = pm(i4,m4);                          {MSW of dividend}
   ay0 = 0000;                               {LSW of dividend}
   divide(ax0,ay1);
   ar = -ay0;                                {reverse sign of k before storing}
   dm(i0,m1) = ar;
   dm(p2_k_i) = i0;

{a_old[1] = k[1]}
   si = ay0;
   sr = ashift si (hi);                      {store in (SBITS+1).(32-SBITS-1) format}
   dm(i2,m1) = sr1;
   dm(i2,m1) = sr0;

{e[1] = (1 - k[1]*k[1])*e[0]}
{ay0 = k[1]}
   mx0 = ay0;
   my0 = ay0;
   mr0 = 0xffff;                             {mr = 1 in 1.31 format}
   mr1 = 0x7fff;
   mr = mr - mx0*my0 (ss);
```

(listing continues on next page)

```
{ax0 = e(0)}
   my0 = ax0;
   mr = mr1 * my0 (ss);
   dm(i5,m4) = mr1;

{for(i = 2; i <= N; i++)}
   cntr = N-1;
   ax0 = 1;                              {i-1}
   dm(i_1) = ax0;
   do pass_two until ce;

         {k[i] = 0}
         ay0 = 0;                        {LSW}
         ay1 = 0;                        {MSW}

{for(j = 1; j <= i-1; j++)}
         ax0 = dm(i_1);
         cntr = ax0;
         m3 = ax0;                       {i-1}
         m7 = ax0;                       {i-1}

{prepare: k[i] = k[i] + R[i-j]*a_old[j]}
   i2 = dm(ap_old); l2 = 0;
   i4 = dm(p2_autocor_speech); l4 = 0;
   modify(i4,m7);                        {->R[i-1]}

   {loop}
   do calc_ks until ce;
         my1 = pm(i4,m6);                {R[i-j]}
         mx1 = dm(i2,m1);                {msw of a_old[j]}
         mx0 = dm(i2,m1);                {lsw of a_old[j]}
         mr  = mx0 * my1 (us);           {lsw * msw}
         mr0 = mr1;                      {shift down 16 bits}
         mr1 = mr2;
         mr  = mr + mx1*my1 (ss);        {msw * msw}
         if mv call overflow;
         ar  = mr0 + ay0;                {acum. lsw's}
         ay0 = ar;
         ar  = mr1 + ay1 + c;            {acum. msw's}
         if av call overflow;

   calc_ks: ay1 = ar;

   {k[i] = R[i] - k[i]}
   i4 = dm(p2_autocor_speech); l4 = 0;
   modify(i4,m7);
   modify(i4,m5);                        {->R[i]}
   si = pm(i4,m4);                       {R[i]}
   sr = ashift si (hi);                  {shift to (SBITS+1).(32-SBITS-1) format}
                                         {ay0 = LSW of k[i]}
```

```
   ar = sr0 - ay0;
   si = ar;                                {store for double precision upshift}
                                           {ay1 = MSW of k[i]}
   ar = sr1 - ay1 + c - 1;
   if av call overflow;
{        sr  = lshift si by SBITS (lo);
         si  = ar;
         se  = exp si (hi);
         ay1 = se;
         se  = NSBITS;
         ax1 = SBITS;
         ar  = ax1 + ay1;
         if gt call overflow;
         sr = sr or ashift si by SBITS (hi);
}

{k[i] = k[i]/e[i-1]}
   i5 = ^e; l5 = 0;
   modify(i5,m7);                          {->e[i-1]}
   ax0 = dm(i5,m5);                        {e[i-1]}
   ay1 = ar {sr1};                         {MSW of k[i]}

   {overflow check}
   ar  = abs ax0;
   ay0 = ar;
   ar  = pass ay1;
   ar  = abs ar;
   ar  = ar - ay0;                         {abs(k[i]) - abs(e[i-1])}
   if gt call overflow;
   ay0 = si {sr0};                         {LSW of k[i]}
   divide(ax0,ay1);


   si = ay0;
   sr = ashift si by SBITS (hi);
   ay0 = sr1;
   i0 = dm(p2_k_i); l0 = 0;
   ay1 = sr1;
   ar  = -ay1;                             {reverse sign of k before storing}
   dm(i0,m1) = ar;                         {k[i] store}
   dm(p2_k_i) = i0;
```

(listing continues on next page)

3 Linear Predictive Coding

```
{a_new[i] = k[i]}
   si = ay0;
   sr = ashift si (hi);                    {store in (SBITS+1).(32-SBITS-1) format}
   i1 = dm(ap_new); l1 = 0;
         modify(i1,m3);
         modify(i1,m3);                    {->a_new[i].msw}
         modify(i1,m1);                    {->a_new[i].lsw}
         dm(i1,m2) = sr0;                  {store lsw}
         dm(i1,m2) = sr1;                  {store msw}

         {for(j = i-1; j>0; j—)}
         cntr = dm(i_1);

{prepare: a_new[j] = a_old[j] - k[i]*a_old[i-j]}
         i2 = dm(ap_old);  l2 = 0;
         modify(i2,m3);
         modify(i2,m3);                    {-> a_old[j+1].msw}
         modify(i2,m2);                    {-> a_old[j].lsw}
         i0 = dm(ap_old);                  {-> a_old[i-j].msw}
         l0 = 0; my1 = ay0;                {k[i]}

         {loop}
         do calc_as until ce;
            ay0 = dm(i2,m2);               {a_old[j].lsw}
            ay1 = dm(i2,m2);               {a_old[j].msw}
            mx1 = dm(i0,m1);               {a_old[i-j].msw}
            mx0 = dm(i0,m1);               {a_old[i-j].lsw}
            mr  = mx0*my1 (us)             {lsw * msw}
            mr0 = mr1;                     {shift down by 16 bits}
            mr1 = mr2;
            mr  = mr + mx1*my1 (ss);       {msw * msw}
            if mv call overflow;
            ar  = ay0 - mr0;               {acum. lsw’s}
            dm(i1,m2) = ar;
            ar  = ay1 - mr1 + c - 1;       {acum. msw’s}
            if av call overflow;
         calc_as: dm(i1,m2) = ar;

{e[i] = (1 - k[i]*k[i]) * e[i-1]}          {my1 = k[i]}
         mx0 = my1;
         mr0 = 0xffff;                     {mr = 1 in 1.31 format}
         mr1 = 0x7fff;
         mr = mr - mx0*my1 (ss);
         if mv call overflow;

         {ax0 = e(i-1)}
         my0 = ax0;
         mr = mr1 * my0 (ss);
         dm(i5,m4) = mr1;
```

```
         {switch the a pointers}
         ax0 = dm(ap_old);
         ay0 = dm(ap_new);
         dm(ap_new) = ax0;
         dm(ap_old) = ay0;

         {i++}
         ay0 = dm(i_1);
         ar  = ay0 + 1;
   pass_two: dm(i_1) = ar;
rts;

.endmod;
```

Listing 3.7 DURBIN2.DSP Subroutine

3 Linear Predictive Coding

```
.module/boot=4        encode_parameters;
{  ENCODE.DSP - truncates/compresses  the lpc parameters.

INPUT:
    i1 -> k (reflection coeffs) l1 = 0
    ar = pitch
    si = gain

OUTPUT: k's encoded inplace
     ar = pitch
     si = gain

The parameters are truncated into the required nr of bits, either by linearly (gain)
or logarithmic (k's) quantization.

Logarithmic: g[i] = log10((1+k[i])/(1-k[i]))/4
}

#include "lpc.h"
.const   LOG_ORDER = 8;
.var/pm  log_coeffs[2*LOG_ORDER];              {Ci lsb, Ci msb, Ci-1 lsb .....}
.init    log_coeffs: <log.cff>;                {scaled down by 512 = 2^9}
.const   LAR_ORDER = 16;
.var/pm  lar_coeffs[2*LAR_ORDER];              {Ci lsb, Ci msb, Ci-1 lsb .....}
.init       lar_coeffs: <enc.cff>;             {scaled down by 512 = 2^9}
.var/pm  round[WORDS_PR_LPCFRAME];
.init       round:
         0x000900, {pitch,    7 bit, (9=16-7) shifting NOT rounding}
         0x000600, {gain,     6 bit}
         0x000600, {k1, 6 bit}
         0x000600, {k2, 6 bit}
         0x000500, {k3, 5 bit}
         0x000500, {k4, 5 bit}
         0x000400, {k5, 4 bit}
         0x000400, {k6, 4 bit}
         0x000300, {k7, 3 bit}
         0x000300, {k8, 3 bit}
         0x000300, {k9, 3 bit}
         0x000200; {k10,      2 bit}
                {      __
                  54 bit/frame}
.var/dm  temp_pitch;
.var/dm  temp_gain;
.entry   encode;
.external   poly_approx;
encode:
```

```
{encode parameters}
   i4 = ^round; l4 = 0;

{ar = pitch}
   se = pm(i4,m5);                        {nr of bits to shift}
   sr = lshift ar (lo);
   dm(temp_pitch) = sr0;

         {si = gain}
         sr0 = si;
         ar  = pass sr0;
         if eq jump zero_gain;
            my0 = 0x0000;                 {gain lsb}
            my1 = si;                     {gain msb}
            ax0 = LOG_ORDER - 1;
            i6  = ^log_coeffs; l6 = 0;
            call poly_approx;             {log10 function}
                  si = mx0;
                  sr = lshift si by 9 (lo);     {scale up by 512, comes with}
                                                {the coeff's}
                  sr = sr or ashift ar by 9 (hi);
                  si = sr1;
                  se = pm(i4,m5);         {nr of bits to round to}
                  call do_round;
            zero_gain:
            dm(temp_gain) = sr0;
            cntr = 2 {N};
            do enc2_k until ce;
                  my0 = 0x0000;           {k lsb}
                  my1 = dm(i1,m0);        {k msb}
                  ax0 = LAR_ORDER - 1;
                  i6  = ^lar_coeffs; l6 = 0;
                  call poly_approx;       {log area ratio function}
                  si = mx0;
                  sr = lshift si by 9 (lo);     {scale up by 512, comes with}
                                                {the coeff's}
                  sr = sr or ashift ar by 9 (hi);
                  si = sr1;
                  se = pm(i4,m5);
                  call do_round;
            enc2_k: dm(i1,m1) = sr0;
            cntr = N-2;
            do enc_k until ce;
                  si = dm(i1,m0);
                  se = pm(i4,m5);
                  call do_round;
            enc_k: dm(i1,m1) = sr0;
```

(listing continues on next page)

```
          {setup return parameters}
          ar = dm(temp_pitch);
          si = dm(temp_gain);
             rts;

{ ROUNDING routine
   Input si = value to be rounded
   se = nr of bits to round to
   Output sr0 = rounded value}

   do_round:
          sr = ashift si (lo);
          mr0 = sr0;
          mr1 = sr1;
          mr  = mr (rnd), ay0 = se;
          ar  = -ay0;
          se  = ar;
          sr  = ashift mr1 (hi);
rts;

.endmod;
```

Listing 3.8 ENCODE.DSP Subroutine

```
.module/boot=3/boot=4        gain_calculation;
{  GAIN.DSP - Calculates the gain factor for a speech frame.

INPUT:
    i0 -> speech frame
    l0 = 0

OUTPUT:
    sr1 = gain

FUNCTIONS CALLED:
   poly_approx - used to aproximate sqrt function
DESCRIPTION:
   The gain of a frame is calculated as:
   gain = sqrt(sum_over_frame(x(n)^2))

   A simple no-speech detection is implemented, if the gain is lower than
   NOISE_FLOOR the gain is set to zero. The result is scaled appropriately by
   GAIN_SCALE.
}

{Include constant definitions}
#include "lpc.h"
.const   NOISE_FLOOR = 0x0000;            {found as gain when no input is present}
.const   GAIN_SCALE  = 0;                 {appropriate scale value}
.entry   calc_gain;
.external   sqrt;

calc_gain:
{calulate energy of frame, R(0), as sum of input squared}
   mr=0;
   cntr = FRAME_LENGTH;
   do cor_data_loop until ce;
         si = dm(i0,m1);
         sr = ashift si by G_INP_SHIFT (hi);    {scale to avoid overflow}
         my0 = sr1;
   cor_data_loop: mr=mr+sr1*my0(ss);

{set gain = 0 if energy is under noise level}
   ay0 = NOISE_FLOOR;
   ar = mr1 - ay0;
   if gt jump speech;
         sr1 = 0;
         jump from_noise;
   speech:
```

(listing continues on next page)

```
         {calc the gain as the squareroot of R(0)}
   sr = lshift mr0 by -12 (lo);                {shift to 16.16 format}
         sr = sr or ashift mr1 by -12 (hi);
         mr1 = sr1;                            {msw of gain^2}
         mr0 = sr0;                            {lsw of gain^2}
   call sqrt;                                  {result is in unsigned 8.8 format}
         sr  = lshift sr1 by 7 (hi);           {shift back to 1.15 format}
         sr  = lshift sr1 by GAIN_SCALE (hi);
   from_noise:
rts;

.endmod;
```

Listing 3.9 GAIN.DSP Subroutine

```
.module/boot=3/boot=4   ovfl;
.entry                  overflow;

{used to break on overflows during debug}
   overflow:
rts;

.endmod;
```

Listing 3.10 OVERFLOW.DSP Subroutine

```
.module/boot=3/boot=4     pitch_detection;
{  PITCH.DSP - extracts the pitch period, and makes a voiced/unvoiced decision.

INPUT:
    i0 -> k[N]
    l0 =  0
    i6 -> autocor_speech[FRAME_LENGTH]
    l6 =  0

OUTPUT:
    si = pitch (= 0 if unvoiced)
CONST:
PITCH_DETECT_LENGTH = part of frame used for pitch detection starting at 3 msec.
     mSEC_3 = sample to start pitch detection at
FUNCIOTNS CALLED:
     None
DESCRIPTION:
   The k's are autoscaled, and then autocorrelated. The new values are correlated
   with the autocorrelated speech R[]. The resulting sequence is searched for the
   largest peak in the interval mSEC_3 .....
   mSEC_3+PITCH_DETECT_LENGTH Depending on the relative size of the peak (to
   re[0]), a decision of voiced/unvoiced is made. In case of voiced the location
   is equal to the pitch period.
}

{Include constant definition}
#include "lpc.h"
.entry   detect_pitch;
.external   overflow;
.var/dm  rk[N];                    {autocorrelation of k[]}
{.var/pm re[FRAME_LENGTH];}        {cross correlation of R[] and rk[]}
.var/dm  k_dm[N];                  {scratch copy's of k}
.var/dm  p2_k;
.var/dm  p2_autocor_speech;
.var/dm  zero_crossings;           {in re[]}

detect_pitch:
{store pointers for later use}
   dm(p2_k) = i0;
   dm(p2_autocor_speech) = i6;

{autoscale before autocorrelation}
{detect largest value in k's}
{i0 = ^k;    l0 = 0;}
   cntr = N;
   sb = -16;
   do max_k until ce;
         si = dm(i0,m1);
   max_k: sb = expadj si;
```

(listing continues on next page)

3 Linear Predictive Coding

```
{adjust k input: normalize to largest and then right shift to |P_K_SHIFT|.
   (16-|P_K_SHIFT|) format (done in one shift). At the same time copy to
   pm for correlation}
   i0 = dm(p2_k); l0 = 0;
   i1 = ^k_dm;    l1 = 0;
   cntr = N;
   ax0 = sb;
   ay0 = P_K_SHIFT;                {scale down to avoid overflow (worst case)}
   ar = ay0 - ax0;
   se = ar;
   do adj_k until ce;
         si = dm(i0,m1);
         sr = ashift si (hi);
   adj_k: dm(i1,m1) = sr1;

{calculate autocorrelation of k[], rk[i] = sum_of k[j]*k[i+j]}
   i5 = ^k_dm; l5 = 0;             {k[i+j]}
   i2 = N;     l2 = 0;             {innerloop counter 'refill'}
   i1 = ^rk;   l1 = 0;
   m2 = -1;
   cntr = N;
   do corr_loop until ce;
         i0 = ^k_dm; l0 = 0;       {k[j]}
         i4 = i5;    l4 = 0;
         cntr = i2;
         mr=0, my0 = dm(i4,m5);
         mx0=dm(i0,m1);
   do cor_data_loop until ce; mr=mr+mx0*my0(ss),my0=dm(i4,m5);
         cor_data_loop: mx0=dm(i0,m1);
         if mv call overflow;
         mx0 = dm(i2,m2);
         my0 = dm(i5,m5);          {(innerloop cnt'er)—, i++}
   corr_loop: dm(i1,m1) = mr1;

{shift down R[] (autocor_speech) to |P_R_SHIFT|.(16-|P_R_SHIFT|) format}
   cntr = FRAME_LENGTH;

{i6 = ^autocor_speech; l6 = 0;}
   do shft_ac until ce;
         si = pm(i6,m4);
         sr = ashift si by P_R_SHIFT (hi);
   shft_ac: pm(i6,m5) = sr1;

{Setup rk[] and R[] (autocor_speech) for correlation. Only calculate the necessary
   correlation coefficients, equivalent to the 0-15 ms portion of the frame
   (samples 0-120 (out of 160), re(0) is necesary for later voiced/unvoiced calc}
```

Linear Predictive Coding 3

```
   { re[i] = sum_of(rk[j]*R[i+j]) }
   i5 = dm(p2_autocor_speech);      l5 = 0;     {R[i+j]}
   i2 = N;                     l2 = 0;
   i6 = dm(p2_autocor_speech);      l6 = 0;
   ay0 = 0;                                    {last 'sign' for zerocrossing count}
   af = pass ay0;                              {zerocrossing counter}
   cntr = PITCH_DETECT_LENGTH + mSEC_3;        {0-15 msec} {correlate rk's and R's}
         do cor_loop until ce;
            i0 = ^rk; l0 = 0;                  {rk[j]}
            i4 = i5;  l4 = 0;
            cntr=i2;
            mr=0, my0=pm(i4,m5), mx0=dm(i0,m1);
            do cor_inner_loop until ce;
            cor_inner_loop: mr=mr+mx0*my0(ss),my0=pm(i4,m5),mx0=dm(i0,m1);
            ar = mr1 xor ay0;                  {test for sign switch = zerocrossing}
            if ge jump no_crossing;
                  af  = af + 1;                {inc zerocrossing counter}
                  ay0 = mr1;                   {store new sign}
                  no_crossing:
                  modify(i5,m5);               {i++}
         cor_loop: pm(i6,m5) = mr1;
         ar = pass af;
         i5 = ar;                              {zerocrossing count}

{find the largest peak in range 3-15 msec. The index of the largest peak is
equal to the pitch period. At the same time count the nr of zerocrossings in
re[]}
   si  = 0;                                    {store for pitch of max peak}
   ay1 = 0;                                    {store for value of max peak}
   i6  = dm(p2_autocor_speech); l6 = 0;
   ax0 = pm(i6,m4);                            {save re(0) for voiced/unvoiced check}
   m7  = mSEC_3;
   modify(i6,m7);                              {-> re(mSEC_3)}
   i2  = mSEC_3; l2 = 0;                       {pitch counter}
   cntr = PITCH_DETECT_LENGTH;
   do find_max_peak until ce;
         ax1 = pm(i6,m5);                      {re[j]}
         ar  = ax1 - ay1;                      {re[j] > max?}
         if le jump not_bigger;
            ay1 = ax1;                         {new max value}
            si = i2;                           {corresponding pitch value}
            not_bigger:
            nop;
   find_max_peak: modify(i2,m1);               {(pitch period cnt'er)++}
```

(listing continues on next page)

```
{Check for voiced/unvoiced excitation. If unvoiced set pitch = 0}
                                        {ax0 = re(0)}
                                        {ay1 = MSW of peakvalue}
   ay0 = 0000;                          {LSB of peakvalue}
   divide(ax0,ay1);                     {ay0 = re[peak]/re[0]}
   ax0 = i5;                            {nr of zerocrossings in re[]}
   ay1 = 70;
   ar = ax0 - ay1;                      {i5>treshold => unvoiced}
   if ge jump not_voiced;
   ax1 = 0x1999;                        {= 0.20}
   ar  = ay0 - ax1;                     {re(j)/re(0) - 0.20}
   if lt jump not_voiced;
   ax1 = 0x2666;                        {= 0.30}
   ar  = ay0 - ax1;                     {re(j)/re(0) - 0.30}
   if gt jump frame_voiced;
   ay1 = 60;
   ar = ax0 - ay1;                      {i5>treshold => unvoiced}
   if lt jump frame_voiced;
   not_voiced:
   si = 0;                          {unvoiced => pitch period = 0}
   i0 = dm(p2_k); l0 = 0;           {zero out k5 - k10 for unvoiced speech}
         m3 = 4;
         modify(i0,m3);
         ax0 = 0;
         cntr = 6;
         do zero_ks until ce;
         zero_ks:  dm(i0,m1) = ax0;
   frame_voiced:
rts;

.endmod;
```

Listing 3.11 PITCH.DSP Subroutine

```
.module/boot=3/boot=4        approximate_func;
{  POLY.DSP - Calculates the polynomial approximation to a function given by the
   coefficients. Uses 32 bit; y = f(x);

INPUT:
    my0 = x lsb
    my1 = x msb
    ax0 = POLY_ORDER - 1
    i6  = -> coeffs (in pm) (Ci lsb, Ci msb, Ci-1 lsb .....)

OUTPUT:
    mx0 = y msb
     ar  = y lsb
     f(x) is approximated by a polynomial:
     f(x) = C[0] + C[1]*X^1 .... + C[(POLY_ORDER-1)]*X^(POLY_ORDER-1)
}

#include “lpc.h”
.entry   poly_approx;

poly_approx:
   mx0 = pm(i6,m5);                              {c lsb}
   ar  = pm(i6,m5);                              {c msb}
   cntr = ax0;
   do approx_loop until ce;
   mr = mx0 * my1 (us), ay0 = pm(i6,m5);         {c[i]lsb*xmsb, c[i-1] lsb}
   mr = mr + ar * my0 (su), ay1 = pm(i6,m5);     {c[i]msb*xlsb, c[i-1] msb}
   mr0 = mr1;
   mr1 = mr2;                                    {shift down by 16 bits}
   mr = mr + ar * my1 (ss);                      {c[i]msb*xmsb}
   ar = mr0 + ay0;                               {c[i]*x lsb + c[i-1] lsb}
   mx0 = ar;
   approx_loop: ar = mr1 + ay1 + c;              {c[i]*x msb + c[i-1] msb}
rts;

.endmod;
```

Listing 3.12 POLY.DSP Subroutine

```
.module/boot=3/boot=4   pre_emphasize_speech;
{  PREEMP.DSP - pre-emphasizes a frame of speech. (filters it)

INPUT:
    i0 -> speech frame to be filtered
    l0 =  0
    Frame is altered!!

OUTPUT:
    frame of speech is emphasized
FUNCTIONS CALLED:
    None
DESCRIPTION:
    Filters the speech using   H(z) = 1 - 0.9375*z-1
}

{Include constant definitions}
#include “lpc.h”
.entry   pre_emph;
.external   overflow;
.var/dm/ram delay;

pre_emph:

   {preemphasize}
   mx0 = 0x8801;                          {u = -0.9375}
   cntr = FRAME_LENGTH;
   do filt_speech until ce;
         mr  = 0;
         my0 = dm(delay);                 {x(n-1)}
         mr1 = dm(i0,m0);                 {x(n)}
         dm(delay) = mr1;                 {update delay with x(n)}
         mr  = mr  + mx0*my0 (ss);        {x(n) + u*x(n-1)}
                  if mv call overflow;
         filt_speech: dm(i0,m1) = mr1;    {store filtered sample}
rts;

.endmod;
```

Listing 3.13 PREEMP.DSP Subroutine

```
.module/boot=3/boot=4           random;
{  RANDOM.DSP - Random number function.

INPUT :
    sr1 = msw of seed
    sr0 = lsw of seed

OUTPUT:
    for best result use ONLY sr1 as random number
    sr1 = msw of new seed between 0 and 2^32
    sr0 = lsw of new seed
FUNCTIONS CALLED:
    None
DESCRIPTION:
    The function (taken from the APPS handbook) implements
         x(n+1) = (a*x(n) + c) mod m
         m = 2^32 a = 1,664,525 c = 32767
}

.entry noise_rand;
noise_rand:
   my1 = 25;                            {upper half of a}
   my0 = 26125;                         {lower half of a}
   mr = sr0*my1 (uu);
   mr = mr + sr1*my0 (uu);              {a(hi)*x(lo)}
   si = mr1;                            {a(hi)*x(lo) + a(lo)*x(hi)}
   mr1 = mr0;
   mr2 = si;
   mr0 = 0xfffe;                        {c = 32767, leftshifted by 1}
   mr = mr + sr0*my0 (uu);              {(above) + a(lo)*x(lo) + c}
   sr = ashift mr2 by 15 (hi);
   sr = sr or lshift mr1 by -1 (hi);    {right shift by 1}
   sr = sr or lshift mr0 by -1 (lo);
rts;

.endmod;
```

Listing 3.14 RANDOM.DSP Subroutine

3 Linear Predictive Coding

```
.module/boot=3/boot=4   square_root;
{  SQRT.DSP - Calculate the squareroot

INPUT:
    mr1 = msw of x in 16.16 format
    mr0 = lsw of x
    m1  = 5

OUTPUT:
    sr1 = y in 8.8 unsigned format
CALLED FUNCTIONS:
    None
DESCRIPTION:
    Approximates the squareroot of x by a Taylor series y = sqrt(x)
COMPUTATION TIME:
    75 cycles (maximum)
}

.const   BASE=h#0d49, SQRT2=h#5a82;
.var/pm  sqrt_coeff[5];
.init       sqrt_coeff: h#5d1d00, h#a9ed00, h#46d600, h#ddaa00, h#072d00;
.entry   sqrt;

sqrt:
   i6=^sqrt_coeff; l6 = 0;                {pointer to coeff. buffer}
   se=exp mr1 (hi);                       {check for redundant bits}
   se=exp mr0 (lo);
   ax0=se, sr=norm mr1 (hi);              {remove redundant bits}
   sr=sr or norm mr0 (lo);
   my0=sr1, ar=pass sr1;
   if eq rts;
   mr=0;
   mr1=BASE;                              {load constant value}
   mf=ar*my0 (rnd), mx0=pm(i6,m5);        {mf = x**2}
   mr=mr+mx0*my0 (ss), mx0=pm(i6,m5);     {mr = BASE + c1*x}
   cntr=4;
         do approx until ce;
            mr=mr+mx0*mf (ss), mx0=pm(i6,m5);
         approx: mf=ar*mf (rnd);
            ay0=15;
            my0=mr1, ar=ax0+ay0;          {se + 15 = 0?}
            if ne jump scale;             {no, compute sqrt(s)}
            sr=ashift mr1 by -6 (hi);
            rts;
```

```
scale:      mr=0;
            mr1=SQRT2;                    {load 1/sqrt(2)}
            my1=mr1, ar=abs ar;
            ay0=ar;
            ar=ay0-1;
            if eq jump pwr_ok;
            cntr=ar;                      {compute (1/sqrt(2))^(se+15)}
            do compute until ce;
compute: mr=mr1*my1 (rnd);
pwr_ok: if neg jump frac;
   ay1=h#0080;                            {load a 1 in 9.23 format}
   ay0=0;                                 {compute reciprocal of mr}
   divs ay1, mr1;
   divq mr1; divq mr1; divq mr1;
   divq mr1; divq mr1; divq mr1;
   divq mr1; divq mr1; divq mr1;
   divq mr1; divq mr1; divq mr1;
   divq mr1; divq mr1; divq mr1;
   mx0=ay0;
   mr=0;
   mr0=h#2000;
   mr=mr+mx0*my0 (us);
   sr=ashift mr1 by 2 (hi);
   sr=sr or lshift mr0 by 2 (lo);
   rts;
frac:    mr=mr1*my0 (rnd);
   sr=ashift mr1 by -6 (hi);
rts;

.endmod;
```

Listing 3.15 SQRT.DSP Subroutine

3 Linear Predictive Coding

```
.module/boot=3/boot=4   lpc_sync_synth;
{  SSYNTH.DSP - synthesizes lpc speech on a pitch syncronious boundry.

INPUT:
    i1  -> (negative) reflection coefficient (k[]'s)
    l1  =  0
    i2  -> output (speech) buffer
    l2  =  0
    ax1 =  pitch period
    mx1 =  gain

OUTPUT:
    Ouput buffer filled
FUNCTIONS CALLED:
    noise_rand          (random number generator)
DESCRIPTION:
    clear_filter: clears delay line, initializes variables (no arguments required)


    synthesis:  updates the frame delay line -> new -> old -> then synthesises
                a frame of speech, based on interpolated parameters (cur_) from
                the last frame (old_) and the latest (new_). When a frame is
                considered voiced the filter is excited with an impuls on a
                pitchsyncronous boundry. When a frame is unvoiced, the filter is
                excited whith random noise. Before input to the filter the
                excitation is scaled to an appropriate value depending on
                voiced/unvoiced status and the gain. The lattice filter is
                taken from the apps handbook.

In order to interpolate the parameter a DELAY OF ONE FRAME is introduced!
}

{include constant definitions}
#include    "lpc.h";

.entry      synthesis, clear_filter;

.external         noise_rand;


.var/pm/ram/circ  e_back[N];                          {delay line}
.var/dm/ram       lo_noise_seed,hi_noise_seed;
.var/pm/ram       new_k[N];
.var/dm/ram       new_gain;
.var/dm/ram       new_pitch;
.var/pm/ram       old_k[N];
.var/dm/ram       old_gain;
.var/dm/ram       old_pitch;
.var/dm/ram/circ  cur_k[N];
```

```
.var/dm/ram       cur_gain;
.var/dm/ram       cur_pitch;
.var/dm/ram       pif_cnt;                        {Place In Frame - cntr}
.var/dm/ram       pp_cnt;                         {pitch period - cntr}
.var/dm/ram first_time;

{clear filter and return}
clear_filter:
   {clear the filter}
   i4 = ^e_back; l4 = 0;
   ar = 0;
   cntr = N;
   do clear_loop until ce;
   clear_loop: pm(i4,m5) = ar;

{initialize seed value for random nr generation}
   ax0 = 0;
   dm(lo_noise_seed) = ax0;
   dm(hi_noise_seed) = ax0;
   ax0 = 1;
   dm(first_time) = ax0;                         {first_time = TRUE}
   rts;

{generate one frame of data and output to speech buffer}
synthesis:
   ax0 = dm(first_time);
   ar  = pass ax0;
   if eq jump not_first;

{copy parm's to new}                             {first_time = TRUE}
   dm(new_pitch) = ax1;
   dm(new_gain) = mx1;
   i6 = ^new_k; l6 = 0;
   cntr = N;
   do move_to_new until ce;
            ax0 = dm(i1,m1);
   move_to_new: pm(i6,m5) = ax0;

   {start of by interpolating}
   ax0 = 1;
   dm(pp_cnt) = ax0;

         {don't do this anymore}
         ax0 = 0;
         dm(first_time) = ax0;
         jump done;
```

(listing continues on next page)

```
not_first:                          {first_time = FALSE}
                                    {move parm's from old to new and update new}
   ax0 = dm(new_pitch);
   mx0 = dm(new_gain);
   dm(new_pitch) = ax1;
   dm(new_gain)  = mx1;
   dm(old_pitch) = ax0;
   dm(old_gain)  = mx0;
   i6 = ^new_k; l6 = 0;
   i5 = ^old_k; l5 = 0;
   cntr = N;
   do move_to_old until ce;
         ay0 = pm(i6,m4), ax0 = dm(i1,m1);
         pm(i5,m5) = ay0;
         move_to_old: pm(i6,m5) = ax0;
         {setup for lattice filter}
         i0 = ^cur_k;  l0 = N;
         i4 = ^e_back; l4 = N;
         m2 = -1;
         m6 = 3;
         m7 = -2;

{setup pitch_period cntr, in temp var (=af)}
         ax0 = dm(pp_cnt);
         af = pass ax0;

         {synthesize a whole frame}
         cntr = FRAME_LENGTH;
         ax0 = 0;
         dm(pif_cnt) = ax0;
         do frame_loop until ce;
            my0 = 0;                {excitation default to 0}
            af = af - 1;
            if gt jump not_pitch_time;

         {time to interpolate}

{calculate interpolation factor = pif_cnt/(FRAME_LENGTH - 1)}
         mx0 = dm(pif_cnt);
         my0 = INTERP_FACTOR;       {= 1/(FRAME_LENGHT - 1) shift -1}
         mr  = mx0 * my0 (ss);      {product is in 17.15 format}
         my1 = mr0;                 {get 1.15 format}

         {calc interpolated gain}
         ax0 = dm(new_gain);
         ay0 = dm(old_gain);
         ar  = ax0 - ay0;           {new - old}
         mr  = ar * my1 (ss);       {(new-old)*int_factor}
         ar  = mr1 + ay0;           { + old}
         dm(cur_gain) = ar;
```

```
{test for transition between voiced/unvoiced and unvoiced/voiced}
         ay0 = dm(old_pitch);
         ar  = pass ay0;
         if eq jump old_unv;
         ax0 = dm(new_pitch);
         ar  = pass ax0;
         if eq jump new_unv;
                                  {voiced - voiced}
         {calc interpolated pitch}
         ar  = ax0 - ay0;         {new - old}
         mr  = ar * my1 (ss);     {(new-old)*int_factor}
         ar  = mr1 + ay0;         { + old}
         dm(cur_pitch) = ar;

         {"interpolate" k's}
         i1 = ^cur_k;  l1 = 0;
         i5 = ^old_k;  l5 = 0;
         cntr = N;
         do interpolate_k until ce; ay0 = pm(i5,m5);
         interpolate_k: dm(i1,m1) = ay0; new_unv:

         {reinitialize pitch cntr}
         ar = dm(cur_pitch);
         af = pass ar;

{set my0 = excitation impuls, in case of voiced frame}
{multiply excitation by gain*(pitch/FRAME_LENGTH)}
   mx1 = ONE_OVER_FRAMEL;         { = 1/FRAME_LENGTH}
   my1 = dm(cur_pitch);
   mr = mx1 * my1 (ss);           {pitch/FRAME_LENGTH, result in 16.16 format}
my1 = dm(cur_gain);
   mr  = mr0 * my1 (ss);          {gain*pitch/FRAME_LENGTH}
   my1 = 0x7fff;                  {impuls}
   mr  = mr1 * my1 (ss);
   my0 = mr1;
   jump not_pitch_time;
   old_unv:            {unvoiced - *}
         ax0 = 1;      {set the pitch_period cntr to a apropriate spacing}
         af  = pass ax0;
         ax0 = 0;
         dm(cur_pitch) = ax0;      {set voiced state to unvoiced}

         {copy old_k to cur_k}
         i1 = ^cur_k;  l1 = 0;
         i5 = ^old_k;  l5 = 0;
         cntr = N;
         do move_to_cur until ce;
         ay0 = pm(i5,m5);
         move_to_cur:dm(i1,m1) = ay0;

         not_pitch_time:
```

(listing continues on next page)

```
            {calculate driving sample if noised}
            ax0 = dm(cur_pitch);
            ar = pass ax0;                {check for voiced/unvoiced}
            if ne jump voiced;
            sr1 = dm(hi_noise_seed);      {noised, old_pitch = 0}
            sr0 = dm(lo_noise_seed);
         call noise_rand;                 {random: 16 bit nr}
         dm(hi_noise_seed) = sr1;
            dm(lo_noise_seed) = sr0;
            ar  = abs sr1;
            sr  = ashift ar by -3 (hi);
            my1 = dm(cur_gain);           {multiply by gain}
            mr  = sr1 * my1 (ss);
            my0 = mr1;                    {my0 = excitation}
            jump do_filter;
            voiced:
            do_filter:

         {do allpole lattice filter (from apps book)}
            cntr = N - 1;
            mr = 0;
            mr1 = my0;
            mx0 = dm(i0,m1), my0 = pm(i4,m5);
            mr = mr - mx0*my0 (ss), mx0 = dm(i0,m1), my0 = pm(i4,m5);
            do dataloop until ce;
                  mr = mr - mx0*my0 (ss);
                  my1 = mr1, mr = 0;
                  mr1 = my0;
                  mr = mr + mx0*my1 (ss), mx0 = dm(i0,m1), my0 = pm(i4,m7);
                  pm(i4,m6) = mr1, mr = 0;
                  dataloop:  mr1 = my1;
                  my0 = pm(i4,m7), mx0 = dm(i0,m2);
                  dm(i2,m1) = my1;        {store synthesized sample}
                  pm(i4,m5) = mr1;        {store newest value in delay line}

                  {increment place_in_frame cntr}
                  ay0 = dm(pif_cnt);
                  ar          = ay0 + 1;
frame_loop: dm(pif_cnt) = ar;
   {store pitch_period cntr for next iteration}
   ar = pass af;
   dm(pp_cnt) = ar;
   done:
   l4 = 0;
rts;

.endmod;
```

Listing 3.16 SSYNTH.DSP Subroutine