Internet Draft
Document: draft-ietf-avt-ilbc-codec-05.txt
Category: Experimental
May 29th, 2004
Expires: November 29th, 2004

S. V. Andersen
Aalborg University
A. Duric
Telio
H. Astrom
R. Hagen
W. B. Kleijn
J. Linden
Global IP Sound

Internet Low Bit Rate Codec

Status of this Memo

This document specifies an Internet experimental standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

By submitting this Internet-Draft, we certify that any applicable patent or other IPR claims of which we am aware have been disclosed, and any of which we become aware will be disclosed, in accordance with RFC 3668.

Copyright Notice

Abstract

This document specifies a speech codec suitable for robust voice communication over IP. The codec is developed by Global IP Sound (GIPS). It is designed for narrow band speech and results in a payload bit rate of 13.33 kbit/s for 30 ms frames and 15.20 kbit/s for 20 ms frames. The codec enables graceful speech quality degradation in the case of lost frames, which occurs in connection with lost or delayed IP packets.

Table of Contents

1. INTRODUCTION

This document contains the description of an algorithm for the coding of speech signals sampled at 8 kHz. The algorithm, called iLBC, uses a block-independent linear-predictive coding (LPC) algorithm and has support for two basic frame lengths: 20 ms at 15.2 kbit/s and 30 ms at 13.33 kbit/s. When the codec operates at block lengths of 20 ms, it produces 304 bits per block which SHOULD be packetized as in [1]. Similarly, for block lengths of 30 ms it produces 400 bits per block which SHOULD be packetized as in [1]. The two modes for the different frame sizes operate in a very similar way. When they differ it is explicitly stated in the text, usually with the notation x/y, where x refers to the 20 ms mode and y refers to the 30 ms mode.

The described algorithm results in a speech coding system with a controlled response to packet losses similar to what is known from pulse code modulation (PCM) with packet loss concealment (PLC), such as the ITU-T G.711 standard [4] which operates at a fixed bit rate of 64 kbit/s. At the same time, the described algorithm enables fixed bit rate coding with a quality-versus-bit rate tradeoff close to state-of-the-art. A suitable RTP payload format for the iLBC codec is specified in [1].

Some of the applications for which this coder is suitable are: real time communications such as telephony and videoconferencing, streaming audio, archival, and messaging.

Cable Television Laboratories (CableLabs(R)) intends to adapt iLBC as a PacketCable(TM) audio codec standard for VoIP over Cable applications [3].

This document is organized as follows. In Section 2 a brief outline of the codec is given. The specific encoder and decoder algorithms are explained in Sections 3 and 4, respectively. A c-code reference implementation is provided in Appendix A.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [2].

2. OUTLINE OF THE CODEC

The codec consists of an encoder and a decoder described in Section 2.1 and 2.2, respectively.

The essence of the codec is LPC and block based coding of the LPC residual signal. For each 160/240 (20ms/30 ms) sample block, the following major steps are performed: A set of LPC filters are computed and the speech signal is filtered through them to produce the residual signal. The codec uses scalar quantization of the dominant part, in terms of energy, of the residual signal for the block. The dominant state is of length 57/58 (20 ms/30 ms) samples

and forms a start state for dynamic codebooks constructed from the already coded parts of the residual signal. These dynamic codebooks are used to code the remaining parts of the residual signal. By this method, coding independence between blocks is achieved, resulting in elimination of propagation of perceptual degradations due to packet loss. The method facilitates high-quality packet loss concealment (PLC).

2.1 Encoder

The input to the encoder SHOULD be 16 bit uniform PCM sampled at 8 kHz. It SHOULD be partitioned into blocks of BLOCKL=160/240 samples for the 20/30 ms frame size. Each block is divided into NSUB=4/6 consecutive sub-blocks of SUBL=40 samples each. For 30 ms frame size, the encoder performs two LPC_FILTERORDER=10 linear-predictive coding (LPC) analyses. The first analysis applies a smooth window centered over the 2nd sub-block and extending to the middle of the 5th sub-block. The second LPC analysis applies a smooth asymmetric window centered over the 5th sub-block and extending to the end of the 6th sub-block. For 20 ms frame size one LPC_FILTERORDER=10 linear-predictive coding (LPC) analysis is performed with a smooth window centered over the 3rd sub-frame.

For each of the LPC analyses, a set of line-spectral frequencies (LSFs) are obtained, quantized and interpolated to obtain LSF coefficients for each sub-block. Subsequently, the LPC residual is computed using the quantized and interpolated LPC analysis filters.

The two consecutive sub-blocks of the residual exhibiting the maximal weighted energy are identified. Within these 2 sub-blocks, the start state (segment) is selected from two choices: the first 57/58 samples or the last 57/58 samples of the 2 consecutive sub-blocks. The selected segment is the one of higher energy. The start state is encoded with scalar quantization.

A dynamic codebook encoding procedure is used to encode 1) the 23/22 (20 ms/30 ms) remaining samples in the 2 sub-blocks containing the start state; 2) encoding of the sub-blocks after the start state in time; 3) encoding of the sub-blocks before the start state in time. Thus, the encoding target can be either the 23/22 samples remaining of the 2 sub-blocks containing the start state or a 40 sample sub-block. This target can consist of samples that are indexed forwards in time or backwards in time depending on the location of the start state.

The coding is based on an adaptive codebook that is built from a codebook memory which contains decoded LPC excitation samples from the already encoded part of the block. These samples are indexed in the same time direction as the target vector and ending at the sample instant prior to the first sample instant represented in the target vector. The codebook is used in CB_NSTAGES=3 stages in a successive refinement approach and the resulting 3 code vector gains are encoded with 5, 4, and 3 bit scalar quantization, respectively.

The codebook search method employs noise shaping derived from the LPC filters and the main decision criterion is minimizing the squared error between the target vector and the code vectors. Each code vector in this codebook comes from one of CB_EXPAND=2 codebook sections. The first section is filled with delayed, already encoded residual vectors. The code vectors of the second codebook section are constructed by predefined linear combinations of vectors in the first section of the codebook.

Since codebook encoding with squared-error matching is known to produce a coded signal of less power than the scalar quantized start state signal, a gain re-scaling method is implemented by a refined search for a better set of codebook gains in terms of power matching after encoding. This is done by searching for a higher value of the gain factor for the first stage codebook since the subsequent stage codebook gains are scaled by the first stage gain.

2.2 Decoder

For packet communications, typically a jitter buffer placed at the receiving end decides whether the packet containing an encoded signal block has been received or lost. This logic is not part of the codec described here. For each received encoded signal block the decoder performs a decoding. For each lost signal block the decoder performs a PLC operation.

The decoding for each block starts by decoding and interpolating the LPC coefficients. Subsequently the start state is decoded.

For codebook encoded segments, each segment is decoded by constructing the 3 code vectors given by the received codebook indices in the same way as the code vectors were constructed in the encoder. The 3 gain factors are also decoded and the resulting decoded signal is given by the sum of the 3 codebook vectors scaled with respective gain.

An enhancement algorithm is applied on the reconstructed excitation signal. This enhancement augments the periodicity of voiced speech regions. The enhancement is optimized under the constraint that the modification signal (defined as the difference between the enhanced excitation and the excitation signal prior to enhancement) has a short-time energy that does not exceed a preset fraction of the short-time energy of the excitation signal prior to enhancement.

A packet loss concealment (PLC) operation is easily embedded in the decoder. The PLC operation can, e.g., be based on repetition of LPC filters and obtaining the LPC residual signal using a long term prediction estimate from previous residual blocks.

3. ENCODER PRINCIPLES

The following block diagram is an overview of all the components of the iLBC encoding procedure. The description of the blocks contains references to the section where that particular procedure is described further.

```
           +------------+    +----------+    +----------+
 speech -> | 1. Pre P   | -> | 2. LPC   | -> | 3. Ana   | ->
           +------------+    +----------+    +----------+
```

```
          +---------------+   +--------------+
       -> | 4. Start Sel  | ->| 5. Scalar Qu | ->
          +---------------+   +--------------+

          +--------------+    +---------------+
       -> |6. CB Search  | -> | 7. Packetize  | -> payload
       |  +--------------+ |  +---------------+
       ----<---------<------
    sub-frame 0..2/4 (20 ms/30 ms)
```

Figure 3.1. Flow chart of the iLBC encoder

1. Pre process speech with a HP filter if needed (section 3.1)
2. Compute LPC parameters, quantize and interpolate (section 3.2)
3. Use analysis filters on speech to compute residual (section 3.3)
4. Select position of 57/58 sample start state (section 3.5)
5. Quantize the 57/58 sample start state with scalar quantization (section 3.5)
6. Search the codebook for each sub-frame. Start with 23/22 sample block, then encode sub-blocks forward in time and then encode sub-blocks backward in time. For each block the steps in figure 3.4 are performed (section 3.6)
7. Packetize the bits into the payload specified in table 3.2.

The input to the encoder SHOULD be 16 bit uniform PCM sampled at 8 kHz. Also it SHOULD be partitioned into blocks of BLOCKL=160/240 samples. Each block input to the encoder is divided into NSUB=4/6 consecutive sub-blocks of SUBL=40 samples each.


```
0        39        79       119       159
+---------------------------------------+
|    1    |    2    |    3    |    4    |
+---------------------------------------+
               20 ms frame
```

```
   0        39        79       119       159       199       239
   +-----------------------------------------------------------+
   |    1    |    2    |    3    |    4    |    5    |    6    |
   +-----------------------------------------------------------+
                              30 ms frame
```

Figure 3.2. One input block to the encoder for 20 ms (with 4 sub-frames) and 30 ms (with 6 sub-frames).

3.1 Pre-processing

In some applications the recorded speech signal contains DC level and/or 50/60 Hz noise. If these components have not been removed prior to the encoder call, they should be removed by a high-pass filter. A reference implementation of this, using a filter with cut off frequency 90 Hz, can be found in Appendix A.28.

3.2 LPC Analysis and Quantization

The input to the LPC analysis module is a possibly high-pass filtered speech buffer, speech_hp, that contains 240/300 (LPC_LOOKBACK + BLOCKL = 80/60 + 160/240 = 240/300) speech samples
,
where samples 0 through 79/59 are from the previous block and samples 80/60 through 239/299 are from the current block. No look-ahead into the next block is used. For the very first block processed, the look back samples are assumed to be zeros.

For each input block, the LPC analysis calculates one/two set(s) of LPC_FILTERORDER=10 LPC filter coefficients using the autocorrelation method and the Levinson-Durbin recursion. These coefficients are converted to the Line Spectrum Frequency representation. In the 20 ms case the set, lsf, represents the spectral characteristics as measured at the center of the third sub-block. For 30 ms frames the first set, lsf1, represents the spectral properties of the input signal at the center of the second sub-block while the other set, lsf2, represents the spectral characteristics as measured at the center of the fifth sub-block. The details of the computation for 30 ms frames are described in 3.2.1 through 3.2.6. Section 3.2.7 explains how the LPC Analysis and Quantization differs for 20 ms frames.

3.2.1 Computation of Autocorrelation Coefficients

The first step in the LPC analysis procedure is to calculate autocorrelation coefficients using windowed speech samples. This windowing is the only difference in the LPC analysis procedure for the two sets of coefficients. For the first set, a 240 sample long

standard symmetric Hanning window is applied to samples 0 through 239 of the input data. The first window, lpc_winTbl, is defined as:

```
lpc_winTbl[i]= 0.5 * (1.0 - cos((2*PI*(i+1))/(BLOCKL+1)));
         i=0,...,119
lpc_winTbl[i] = winTbl[BLOCKL - i - 1]; i=120,...,239
```

The windowed speech speech_hp_win1 is then obtained by multiplying the 240 first samples of the input speech buffer with the window coefficients:

```
speech_hp_win1[i] = speech_hp[i] * lpc_winTbl[i];
         i=0,...,BLOCKL-1
```

From these 240 windowed speech samples, 11 (LPC_FILTERORDER + 1) autocorrelation coefficients, acf1, are calculated:

```
acf1[lag] += speech_hp_win1[n] * speech_hp_win1[n + lag];
         lag=0,...,LPC_FILTERORDER; n=0,...,BLOCKL-lag-1
```

In order to make the analysis more robust against numerical precision problems, a spectral smoothing procedure is applied by windowing the autocorrelation coefficients before the LPC coefficients are computed. Also, a white noise floor is added to the autocorrelation function by multiplying coefficient zero by 1.0001 (40dB below the energy of the windowed speech signal). These two steps are implemented by multiplying the autocorrelation coefficients with the following window:

```
lpc_lagwinTbl[0] = 1.0001;
lpc_lagwinTbl[i] = exp(-0.5 * ((2 * PI * 60.0 * i) /FS)^2);
         i=1,...,LPC_FILTERORDER
         where FS=8000 is the sampling frequency
```

Then, the windowed acf function acf1_win is obtained by:

```
acf1_win[i] = acf1[i] * lpc_lagwinTbl[i];
         i=0,...,LPC_FILTERORDER
```

The second set of autocorrelation coefficients, acf2_win are obtained in a similar manner. The window, lpc_asymwinTbl, is applied to samples 60 through 299, i.e., the entire current block. The window consists of two segments; the first (samples 0 to 219) being half a Hanning window with length 440 and the second being a quarter of a cycle of a cosine wave. By using this asymmetric window, an LPC analysis centered in the fifth sub-block is obtained without the need for any look-ahead, which would have added delay. The asymmetric window is defined as:

```
lpc_asymwinTbl[i] = (sin(PI * (i + 1) / 441))^2; i=0,...,219


lpc_asymwinTbl[i] = cos((i - 220) * PI / 40); i=220,...,239
```


and the windowed speech is computed by:

```
speech_hp_win2[i] = speech_hp[i + LPC_LOOKBACK] *
         lpc_asymwinTbl[i];  i=0,....BLOCKL-1
```

The windowed autocorrelation coefficients are then obtained in exactly the same way as for the first analysis instance.

The generation of the windows lpc_winTbl, lpc_asymwinTbl, and lpc_lagwinTbl are typically done in advance and the arrays are stored in ROM rather than repeating the calculation for every block.

3.2.2 Computation of LPC Coefficients

From the 2 x 11 smoothed autocorrelation coefficients, acf1_win and acf2_win, the 2 x 11 LPC coefficients, lp1 and lp2, are calculated in the same way for both analysis locations using the well known Levinson-Durbin recursion. The first LPC coefficient is always 1.0, resulting in 10 unique coefficients.

After determining the LPC coefficients, a bandwidth expansion procedure is applied in order to smooth the spectral peaks in the short-term spectrum. The bandwidth addition is obtained by the following modification of the LPC coefficients:

```
lp1_bw[i] = lp1[i] * chirp^i; i=0,...,LPC_FILTERORDER
lp2_bw[i] = lp2[i] * chirp^i; i=0,...,LPC_FILTERORDER
```

where "chirp" is a real number between 0 and 1. It is RECOMMENDED to use a value of 0.9.

3.2.3 Computation of LSF Coefficients from LPC Coefficients

Thusfar, two sets of LPC coefficients that represent the short-term spectral characteristics of the speech signal for two different time locations within the current block have been determined. These coefficients SHOULD be quantized and interpolated. Before doing so, it is advantageous to convert the LPC parameters into another type of representation called Line Spectral Frequencies (LSF). The LSF parameters are used because they are better suited for quantization and interpolation than the regular LPC coefficients. Many computationally efficient methods for calculating the LSFs from the LPC coefficients have been proposed in the literature. The detailed implementation of one applicable method can be found in Appendix A.26. The two arrays of LSF coefficients obtained, lsf1 and lsf2, are of dimension 10 (LPC_FILTERORDER).

3.2.4 Quantization of LSF Coefficients

Since the LPC filters defined by the two sets of LSFs are needed also in the decoder, the LSF parameters need to be quantized and transmitted as side information. The total number of bits required to represent the quantization of the two LSF representations for one block of speech is 40 with 20 bits used for each of lsf1 and lsf2.

For computational and storage reasons, the LSF vectors are quantized using 3-split vector quantization (VQ). That is, the LSF vectors are split into three sub-vectors which are each quantized with a regular VQ. The quantized versions of lsf1 and lsf2, qlsf1 and qlsf2, are obtained by using the same memoryless split VQ. The length of each of these two LSF vectors is 10 and they are split into 3 sub-vectors containing 3, 3 and 4 values respectively.

For each of the sub-vectors, a separate codebook of quantized values has been designed using a standard VQ training method for a large database containing speech from a large number of speakers recorded under various conditions. The size of each of the three codebooks associated with the split definitions above is:

```
int size_lsfCbTbl[LSF_NSPLIT] = {64,128,128};
```

The actual values of the vector quantization codebook that must be used can be found in the reference code of appendix A. Both sets of LSF coefficients, lsf1 and lsf2, are quantized with a standard memoryless split vector quantization (VQ) structure using the squared error criterion in the LSF domain. The split VQ quantization consists of the following steps:

1) Quantize the first 3 LSF coefficients (1 - 3) with a VQ codebook of size 64.
2) Quantize the LSF coefficients 4, 5, and 6 with VQ a codebook of

size 128.
3) Quantize the last 4 LSF coefficients (7 - 10) with a VQ codebook of size 128.

This procedure, repeated for lsf1 and lsf2, gives 6 quantization indices and the quantized sets of LSF coefficients qlsf1 and qlsf2. Each set of three indices is encoded with 6 + 7 + 7 = 20 bits. The total number of bits used for LSF quantization in a block is thus 40 bits.

3.2.5 Stability Check of LSF Coefficients

The LSF representation of the LPC filter has the nice property that the coefficients are ordered by increasing value, i.e., lsf(n-1) < lsf(n), 0 < n < 10, if the corresponding synthesis filter is stable. Since we are employing a split VQ scheme it is possible that at the split boundaries the LSF coefficients are not ordered correctly and hence the corresponding LP filter is unstable. To ensure that the filter used is stable, a stability check is performed for the quantized LSF vectors. If it turns out that the coefficients are not ordered appropriately (with a safety margin of 50 Hz to ensure that formant peaks are not too narrow) they will be moved apart. The detailed method for this can be found in Appendix A.40. The same procedure is performed in the decoder. This ensures that exactly the same LSF representations are used in both encoder and decoder.

3.2.6 Interpolation of LSF Coefficients

From the two sets of LSF coefficients that are computed for each block of speech, different LSFs are obtained for each sub-block by means of interpolation. This procedure is performed for the original LSFs (lsf1 and lsf2), as well as the quantized versions qlsf1 and qlsf2 since both versions are used in the encoder. Here follows a brief summary of the interpolation scheme while the details are found in the c-code of Appendix A. In the first sub-block, the average of the second LSF vector from the previous block and the first LSF vector in the current block is used. For sub-blocks two through five the LSFs used are obtained by linear interpolation from lsf1 (and qlsf1) to lsf2 (and qlsf2) with lsf1 used in sub-block two and lsf2 in sub-block five. In the last sub-block, lsf2 is used. For the very first block it is assumed that the last LSF vector of the previous block is equal to a predefined vector, lsfmeanTbl, that was obtained by calculating the mean LSF vector of the LSF design database.

```
lsfmeanTbl[LPC_FILTERORDER] = {0.281738, 0.445801, 0.663330,
```

0.962524, 1.251831, 1.533081, 1.850586, 2.137817,
2.481445, 2.777344}

The interpolation method is standard linear interpolation in the LSF domain. The interpolated LSF values are converted to LPC coefficients for each sub-block. The unquantized and quantized LPC coefficients form two sets of filters respectively. The unquantized analysis filter for sub-block k:

```
              ___
              \
Ak(z)= 1 + > ak(i)*z^(-i)
              /__
           i=1...LPC_FILTERORDER
```

And the quantized analysis filter for sub-block k:

```
               ___
               \
A~k(z)= 1 + > a~k(i)*z^(-i)
               /__
           i=1...LPC_FILTERORDER
```

A reference implementation of the lsf encoding is given in Appendix A.38. A reference implementation of the corresponding decoding can be found in Appendix A.36.

3.2.7 LPC Analysis and Quantization for 20 ms frames

As stated before, the codec only calculates one set of LPC parameters for the 20 ms frame size as opposed to two sets for 30 ms frames. A single set of autocorrelation coefficients is calculated on the LPC_LOOKBACK + BLOCKL = 80 + 160 = 240 samples. These sampl
es
are windowed with the asymmetric window lpc_asymwinTbl, centered over the third sub-frame, to form speech_hp_win. Autocorrelation coefficients, acf, are calculated on the 240 samples in

speech_hp_win and then windowed exactly as in 3.2.1 (resulting in acf_win).

This single set of windowed autocorrelation coefficients is used to calculate LPC Coefficients, LSF Coefficients and quantized LSF coefficients in exactly the same manner as in 3.2.3 to 3.2.4. As for the 30 ms frame size, the 10 LSF coefficients are divided into three sub-vectors of size 3, 3, 4 and quantized using the same scheme and codebook as in 3.2.4 to finally get 3 quantization indices. The quantized LSF coefficients are stabilized with the algorithm described in 3.2.5.

From the set of LSF coefficients that was computed for this block together with the LSF coefficients from the previous block, different LSFs are obtained for each sub-block by means of interpolation. The interpolation is done linearly in the LSF domain over the 4 sub-blocks, so that the n-th sub-frame uses the weight (4-n)/4 for the LSF from old frame and the weight n/4 of the LSF from the current frame. For the very first block the mean LSF, lsfmeanTbl, is used as the LSF from the previous block. Similar to 3.2.6, both unquantized, A(z), and quantized, A~(z), analysis filters are calculated for each of the four sub-blocks.

3.3 Calculation of the Residual

The block of speech samples is filtered by the quantized and interpolated LPC analysis filters to yield the residual signal. In particular, the corresponding LPC analysis filter for each 40 sample sub-block is used to filter the speech samples for the same sub-block. The filter memory at the end of each sub-block is carried over to the LPC filter of the next sub-block. The signal at the output of each LP analysis filter constitutes the residual signal for the corresponding sub-block.

A reference implementation of the LPC analysis filters is given in Appendix A.10.

3.4 Perceptual Weighting Filter

In principle any good design of a perceptual weighting filter can be applied in the encoder without compromising this codec definition. It is however RECOMMENDED to use the perceptual weighting filter specified below:

```
Weighting filter for sub-block k:

Wk(z)=1/Ak(z/LPC_CHIRP_WEIGHTDENUM), where
                             LPC_CHIRP_WEIGHTDENUM = 0.4222
```

This is a simple design with low complexity that is applied in the LPC residual domain. Here Ak(z) is the filter obtained from unquantized but interpolated LSF coefficients.

3.5 Start State Encoder

The start state is quantized using a common 6-bit scalar quantizer for the block and a 3-bit scalar quantizer operating on scaled samples in the weighted speech domain. In the following we describe the state encoding in greater detail.

3.5.1 Start State Estimation

The two sub-blocks containing the start state are determined by finding the two consecutive sub-blocks in the block having the highest power. Advantageously, down-weighting is used in the beginning and end of the sub-frames. I.e., the following measure is computed (NSUB=4/6 for 20/30 ms frame size):

```
      nsub=1,...,NSUB-1
      ssqn[nsub] = 0.0;
      for (i=(nsub-1)*SUBL; i<(nsub-1)*SUBL+5; i++)
               ssqn[nsub] += sampEn_win[i-(nsub-1)*SUBL]*
                                 residual[i]*residual[i];
      for (i=(nsub-1)*SUBL+5; i<(nsub+1)*SUBL-5; i++)
               ssqn[nsub] += residual[i]*residual[i];
      for (i=(nsub+1)*SUBL-5; i<(nsub+1)*SUBL; i++)
               ssqn[nsub] += sampEn_win[(nsub+1)*SUBL-i-1]*
                                 residual[i]*residual[i];
```

where sampEn_win[5]={1/6, 2/6, 3/6, 4/6, 5/6}; MAY be used. The sub-frame number corresponding to the maximum value of ssqEn_win[nsub-1]*ssqn[nsub] is selected as the start state indicator. A weighting of ssqEn_win[]={0.8,0.9,1.0,0.9,0.8} for 30 ms frames and ssqEn_win[]={0.9,1.0,0.9} for 20 ms frames; MAY advantageously be used to bias the start state towards the middle of the frame.

For 20 ms frames there are 3 possible positions of the two-sub-block length maximum power segment, the start state position is encoded using 2 bits. The start state position, start, MUST be encoded as:

```
      start=1: start state in sub-frame 0 and 1
      start=2: start state in sub-frame 1 and 2
      start=3: start state in sub-frame 2 and 3
```

For 30 ms frames there are 5 possible positions of the two-sub-block length maximum power segment, the start state position is encoded using 3 bits. The start state position, start, MUST be encoded as:

```
      start=1: start state in sub-frame 0 and 1
      start=2: start state in sub-frame 1 and 2
      start=3: start state in sub-frame 2 and 3
      start=4: start state in sub-frame 3 and 4
      start=5: start state in sub-frame 4 and 5
```

hence, in both cases, index 0 is not utilized. In order to shorten the start state for bit rate efficiency, the start state is brought down to STATE_SHORT_LEN=57 samples for 20 ms frames and

STATE_SHORT_LEN=58 samples for 30 ms frames. The power of the first 23/22 and last 23/22 samples of the 2 sub-frame block identified above is computed as the sum of the squared signal sample values and the 23/22 sample segment with the lowest power is excluded from the start state. One bit is transmitted to indicate which of the 2 possible 57/58 sample segments is used. The start state position within the 2 sub-frames determined above, state_first, MUST be encoded as:

```
state_first=1: start state is first STATE_SHORT_LEN samples
state_first=0: start state is last STATE_SHORT_LEN samples
```

3.5.2 All-Pass Filtering and Scale Quantization

The block of residual samples in the start state is first filtered by an all-pass filter with the quantized LPC coefficients as denominator and reversed quantized LPC coefficients as numerator. The purpose of this phase-dispersion filter is to get a more even distribution of the sample values in the residual signal. The filtering is performed by circular convolution, where the initial filter memory is set to zero.

```
res(0..(STATE_SHORT_LEN-1))   = uncoded start state residual
res((STATE_SHORT_LEN)..(2*STATE_SHORT_LEN-1)) = 0

Pk(z) = A~rk(z)/A~k(z), where
                             ___
                             \
A~rk(z)= z^(-LPC_FILTERORDER)+>a~k(i+1)*z^(i-(LPC_FILTERORDER-1))


                             /__
                         i=0...(LPC_FILTERORDER-1)

and A~k(z) is taken from the block where the start state begins

res -> Pk(z) -> filtered

ccres(k) = filtered(k) + filtered(k+STATE_SHORT_LEN),
                                  k=0..(STATE_SHORT_LEN-1)
```

The all pass filtered block is searched for its largest magnitude sample. The 10-logarithm of this magnitude is quantized with a 6-bit quantizer, state_frgqTbl, by finding the nearest representation. This results in an index, idxForMax, corresponding to a quantized value, qmax. The all-pass filtered residual samples in the block are then multiplied with a scaling factor scal=4.5/(10^qmax) to yield normalized samples.

```
state_frgqTbl[64] = {1.000085, 1.071695, 1.140395, 1.206868,
               1.277188, 1.351503, 1.429380, 1.500727, 1.569049,
               1.639599, 1.707071, 1.781531, 1.840799, 1.901550,
               1.956695, 2.006750, 2.055474, 2.102787, 2.142819,
               2.183592, 2.217962, 2.257177, 2.295739, 2.332967,
               2.369248, 2.402792, 2.435080, 2.468598, 2.503394,
               2.539284, 2.572944, 2.605036, 2.636331, 2.668939,
```



```
               2.698780, 2.729101, 2.759786, 2.789834, 2.818679,
               2.848074, 2.877470, 2.906899, 2.936655, 2.967804,
               3.000115, 3.033367, 3.066355, 3.104231, 3.141499,
               3.183012, 3.222952, 3.265433, 3.308441, 3.350823,
               3.395275, 3.442793, 3.490801, 3.542514, 3.604064,
               3.666050, 3.740994, 3.830749, 3.938770, 4.101764}
```

3.5.3 Scalar Quantization

The normalized samples are quantized in the perceptually weighted speech domain by a sample-by-sample scalar DPCM quantization as depicted in Figure 3.3. Each sample in the block is filtered by a weighting filter Wk(z), specified in section 3.4, to form a weighted speech sample x[n]. The target sample d[n] is formed by subtracting a predicted sample y[n], where the prediction filter is given by

$$Pk(z) = 1 - 1 / Wk(z).$$

```
            +-------+  x[n] +    d[n] +-----------+ u[n]
residual -->| Wk(z) |-------->(+)---->| Quantizer |------> quantized
            +-------+       - /|\     +-----------+    |   residual
                               |                      \|/
                         y[n] +--------------------->(+)
                               |                       |
                               |        +------+       |
                               +--------| Pk(z)|<------+
                                        +------+
```

Figure 3.3. Quantization of start state samples by DPCM in weighted speech domain.

The coded state sample u[n] is obtained by quantizing d[n] with a 3-bit quantizer with quantization table state_sq3Tbl.

```
state_sq3Tbl[8] = {-3.719849, -2.177490, -1.130005, -0.309692,
```

               0.444214, 1.329712, 2.436279, 3.983887}

The quantized samples are transformed back to the residual domain by 1) scaling with 1/scal 2) time-reversing the scaled samples 3) filtering the time-reversed samples by the same all-pass filter as in section 3.5.2, using circular convolution 4) time-reversing the filtered samples. (More detailed in section 4.2)

A reference implementation of the start state encoding can be found in Appendix A.46.

3.6 Encoding the remaining samples

A dynamic codebook is used to encode 1) the 23/22 remaining samples in the 2 sub-blocks containing the start state; 2) encoding of the sub-blocks after the start state in time; 3) encoding of the sub-blocks before the start state in time. Thus, the encoding target can be either the 23/22 samples remaining of the 2 sub-blocks containing the start state or a 40 sample sub-block. This target can consist of

samples that are indexed forwards in time or backwards in time depending on the location of the start state. The length of the target is denoted by lTarget.

The coding is based on an adaptive codebook that is built from a codebook memory which contains decoded LPC excitation samples from the already encoded part of the block. These samples are indexed in the same time direction as the target vector and ending at the sample instant prior to the first sample instant represented in the target vector. The codebook memory has length lMem which is equal to CB_MEML=147 for the two/four 40 sample sub-blocks and 85 for the 23/22 sample sub-block.

The following figure shows an overview of the encoding procedure.

```
         +------------+    +---------------+    +-------------+
      -> | 1. Decode  | -> | 2. Mem setup  | -> | 3. Perc. W. | ->
         +------------+    +---------------+    +-------------+

         +------------+    +-----------------+
      -> | 4. Search  | -> | 5. Upd. Target  | ------------------>
       | +------------+    +------------------ |
       ----<-------------<-----------<----------
                     stage=0..2

         +----------------+
      -> | 6. Recalc G[0] | ---------------> gains and CB indices
         +----------------+
```

Figure 3.4. Flow chart of the codebook search in the iLBC encoder

1. Decode the part of the residual that has been encoded so far, using the codebook without perceptual weighting
2. Set up the memory by taking data from the decoded residual. This memory is used to construct codebooks from. For blocks preceding the start state, both the decoded residual and the target are time reversed (section 3.6.1)
3. Filter the memory + target with the perceptual weighting filter (section 3.6.2)
4. Search for the best match between the target and the codebook vector. Compute the optimal gain for this match and quantize that gain (section 3.6.4)
5. Update the perceptually weighted target by subtracting the contribution from the selected codebook vector from the perceptually weighted memory (quantized gain times selected vector). Repeat 4. and 5. for the 2 additional stages
6. Calculate the energy loss due to encoding of the residual. If needed, compensate for this loss by an upscaling and requantization of the gain for the first stage (section 3.7)

The following sections provide an in-depth description of the different blocks of figure 3.4.

3.6.1 Codebook Memory

The codebook memory is based on the already encoded sub-blocks so the available data for encoding increases for each new sub-block that has been encoded. Until enough sub-blocks have been encoded to fill the codebook memory with data it is padded with zeros. The following figure shows an example of the order in which the sub-blocks are encoded for the 30 ms frame size if the start state is located in the last 58 samples of sub-block 2 and 3.

```
+-----------------------------------------------------+
|  5     | 1  |///|////////|    2   |    3   |    4   |
+-----------------------------------------------------+
```

Figure 3.5. The order from 1 to 5 in which the sub-blocks are encoded. The slashed area is the start state.

The first target sub-block to be encoded is number 1 and the corresponding codebook memory is shown in the following figure. Since the target vector is before the start state in time the codebook memory and target vector are time reversed. By reversing them in time, the search algorithm can be reused. Since only the start state has been encoded so far the last samples of the codebook memory are padded with zeros.

```
+-------------------------
|zeros|\\\\\\\\|\\\\|  1 |
+-------------------------
```

Figure 3.6. The codebook memory, length lMem=85 samples, and the target vector 1, length 22 samples.

The next step is to encode sub-block 2 using the memory which now has increased since sub-block 1 has been encoded. The following figure shows the codebook memory for encoding of sub-block 2.

```
+-----------------------------------
| zeros | 1  |///|////////|    2   |
+-----------------------------------
```

Figure 3.7. The codebook memory, length lMem=147 samples, and the target vector 2, length 40 samples.

The next step is to encode sub-block 3 using the memory which now has increased yet again since sub-blocks 1 and 2 have been encoded but it still has to be padded with a few zeros. The following figure shows the codebook memory for encoding of sub-block 3.

```
+------------------------------------------
|zeros| 1  |///|////////|    2   |   3    |
+------------------------------------------
```

Figure 3.8. The codebook memory, length lMem=147 samples, and the target vector 3, length 40 samples.

The next step is to encode sub-block 4 using the memory which now has increased yet again since sub-blocks 1, 2 and 3 have been encoded. This time the memory does not have to be padded with zeros. The following figure shows the codebook memory for encoding of sub-block 4.

```
+------------------------------------------
|1|///|////////|    2   |   3    |   4    |
+------------------------------------------
```

Figure 3.9. The codebook memory, length lMem=147 samples, and the target vector 4, length 40 samples.

The final target sub-block to be encoded is number 5 and the corresponding codebook memory is shown in the following figure. Since the target vector is before the start state in time the codebook memory and target vector are time reversed.

```
+------------------------------------------
|  3  |   2    |\\\\\\\\|\\\\|  1 |   5    |
+------------------------------------------
```

Figure 3.10. The codebook memory, length lMem=147 samples, and the target vector 5, length 40 samples.

For the case of 20 ms frames the encoding procedure looks almost exactly the same. The only difference is that the size of the start state is 57 samples and that there are only 3 sub-blocks to be encoded. The encoding order is the same as above starting with the 23 sample target and then encoding the two remaining 40 sample sub-blocks, first going forward in time and then going backwards in time relative to the start state.

3.6.2 Perceptual Weighting of Codebook Memory and Target

To provide a perceptual weighting of the coding error, a concatenation of the codebook memory and the target to be coded is all pole filtered with the perceptual weighting filter specified in section 3.4. The filter state of the weighting filter is set to zero.

```
in(0..(lMem-1))            = unweighted codebook memory
in(lMem..(lMem+lTarget-1)) = unweighted target signal


in -> Wk(z) -> filtered,
    where Wk(z) is taken from the sub-block of the target

weighted codebook memory = filtered(0..(lMem-1))
weighted target signal = filtered(lMem..(lMem+lTarget-1))
```

The codebook search is done using the weighted codebook memory and the weighted target, while the decoding and the codebook memory update uses the unweighted codebook memory.

3.6.3 Codebook Creation

The codebook for the search is created from the perceptually weighted codebook memory. It consists of two sections where the first is referred to as the base codebook and the second as the expanded codebook since it is created by linear combinations of the first. Each of these two sections also has a subsection referred to as the augmented codebook. The augmented codebook is only created and used for the coding of the 40 sample sub-blocks and not for the 23/22 sample sub-block case. The codebook size used for the different sub-blocks and different stages are summarized in the table below.

		Stage 1		Stage 2 & 3	
	22	128	(64+0)*2	128	(64+0)*2
Sub-	1:st 40	256	(108+20)*2	128	(44+20)*2
Blocks	2:nd 40	256	(108+20)*2	256	(108+20)*2
	3:rd 40	256	(108+20)*2	256	(108+20)*2
	4:th 40	256	(108+20)*2	256	(108+20)*2

Table 3.1. Codebook sizes for the 30 ms mode

The table 3.1 shows the codebook size for the different sub-blocks and stages for 30 ms frames. Inside the parenthesis it shows how the number of codebook vectors is distributed, within the two sections, between the base/expanded codebook and the augmented base/expanded codebook. It should be interpreted in the following way: (base/expanded cb + augmented base/expanded cb). The total number of codebook vectors for a specific sub-block and stage is given by the following formula:

Tot. cb vectors = base cb + aug. base cb + exp. cb + aug. exp. cb

The corresponding values to figure 3.1 for 20 ms frames are only slightly modified. The short sub-block is 23 instead of 22 samples and the 3:rd and 4:th sub-frame are not present.

3.6.3.1 Creation of a Base Codebook

The base codebook is given by the perceptually weighted codebook memory that is mentioned in section 3.5.3. The different codebook vectors are given by sliding a window of length 23/22 or 40, given by variable lTarget, over the lMem long perceptually weighted codebook memory. The indices are ordered so that the codebook vector containing sample(lMem-lTarget-n) to (lMem-n-1) of the codebook memory vector has index n, where n=0..lMem-lTarget. Thus the total number of base codebook vectors is lMem-lTarget+1 and the indices are ordered from sample delay lTarget (23/22 or 40) to lMem+1 (86 or 148).

3.6.3.2 Codebook Expansion

The base codebook is expanded by a factor of 2, creating an additional section in the codebook. This new section is obtained by filtering the base codebook, base_cb, with a FIR filter with filter length CB_FILTERLEN=8. The delay of four samples introduced by the FIR filter is compensated for in the construction of the expanded codebook.

```
cbfiltersTbl[CB_FILTERLEN]={-0.033691, 0.083740, -0.144043,
               0.713379, 0.806152, -0.184326,
               0.108887, -0.034180};


                ___
                \
   exp_cb(k)=  + > cbfiltersTbl(i)*x(k-i+4)
                /__
         i=0...(LPC_FILTERORDER-1)

   where x(j) = base_cb(j) for j=0..lMem-1 and 0 otherwise
```

The individual codebook vectors of the new filtered codebook, exp_cb, and their indices are obtained in the same fashion as described above for the base codebook.

3.6.3.3 Codebook Augmentation

For the cases when encoding entire sub-blocks, i.e. cbveclen=40, the base and expanded codebooks are augmented to increase codebook richness. The codebooks are augmented by vectors produced by interpolation of segments. The base and expanded codebook, constructed above, consists of vectors corresponding to sample delays in the range from cbveclen to lMem. The codebook augmentation attempts to augment these codebooks with vectors corresponding to sample delays from 20 to 39. However, not all of these samples are present in the base codebook and expanded codebook respectively. Therefore, the augmentation vectors are constructed as linear combinations between samples corresponding to sample delays in the range 20 to 39. The general idea of this procedure is presented in the following figures and text. The procedure is performed for both the base codebook and the expanded codebook.

```
    - - ------------------------|
 codebook memory                |
    - - ------------------------|
              |-5-|---15---|-5-|
```

```
               pi  pp       po

                   |        |                       Codebook vector
                   |---15---|-5-|-----20-----|   <- corresponding to
                       i     ii      iii            sample delay 20
```

Figure 3.11. Generation of the first augmented codebook

The figure 3.11 shows the codebook memory with pointers pi, pp and po where pi points to sample 25, pp to sample 20 and po to sample 5. Below the codebook memory, the augmented codebook vector corresponding to sample delay 20 is drawn. Segment i consists of 15 samples from pointer pp and forward in time. Segment ii consists of 5 interpolated samples from pi and forward and from po and forward. The samples are linearly interpolated with weights [0.0, 0.2, 0.4, 0.6, 0.8] for pi and weights [1.0, 0.8, 0.6, 0.4, 0.2] for po. Segment iii consists of 20 samples from pp and forward. The augmented codebook vector corresponding to sample delay 21 is produced by moving pointers pp and pi one sample backwards in time. That gives us the following figure.

```
    - - ------------------------|
 codebook memory                |
    - - ------------------------|
               |-5-|---16---|-5-|
               pi  pp       po

                   |        |                       Codebook vector
                   |---16---|-5-|-----19-----|   <- corresponding to
                       i     ii      iii            sample delay 21
```

Figure 3.12. Generation of the second augmented codebook

The figure 3.12 shows the codebook memory with pointers pi, pp and po where pi points to sample 26, pp to sample 21 and po to sample 5. Below the codebook memory, the augmented codebook vector corresponding to sample delay 21 is drawn. Segment i does now consist of 16 samples from pp and forward. Segment ii consists of 5 interpolated samples from pi and forward and po and forward and the interpolation weights are the same throughout the procedure. Segment iii consists of 19 samples from pp and forward. The same procedure of moving the two pointers is continued until the last augmented vector corresponding to sample delay 39 has been created. This gives a total of 20 new codebook vectors to each of the two sections. Thus the total number of codebook vectors for each of the two sections, when including the augmented codebook becomes lMem-SUBL+1+SUBL/2. This is provided that augmentation is evoked, i.e., that lTarget=SUBL.

3.6.4 Codebook Search

The codebook search uses the codebooks described in the sections above to find the best match of the perceptually weighted target, see section 3.6.2. The search method is a multi-stage gain-shape matching performed as follows. At each stage the best shape vector is identified, then the gain is calculated and quantized, and finally the target is updated in preparation for the next codebook search stage. The number of stages is CB_NSTAGES=3.

If the target is the 23/22 sample vector the codebooks are indexed in the order: base codebook followed by the expanded codebook. If the target is 40 samples the order is: base codebook, augmented base

codebook, expanded codebook and finally augmented expanded codebook. The size of each codebook section and its corresponding augmented section is given by table 3.1 in section 3.6.3.

For example when coding the second 40 sample sub-block indices 0-107 correspond to the base codebook, 108-127 correspond to the augmented base codebook, 128-235 correspond to the expanded codebook and finally indices 236-255 correspond to the augmented expanded codebook. The indices are divided in the same fashion for all stages in the example. Only in the case of coding the first 40 sample sub-block is there a difference between stages (see Table 3.1).

3.6.4.1 Codebook Search at Each Stage

The codebooks are searched to find the best match to the target at each stage. When the best match is found the target is updated and the next-stage search is started. The three chosen codebook vectors and their corresponding gains constitute the encoded sub-block. The best match is decided by the following three criteria:

1. Compute the measure

```
(target*cbvec)^2 / ||cbvec||^2
```

for all codebook vectors, cbvec, and choose the codebook vector maximizing the measure. The expression (target*cbvec) is the dot product between the target vector to be coded and the codebook vector for which we compute the measure. The norm, ||x||, is defined as the square root of (x*x).

2. The absolute value of the gain, corresponding to the chosen

codebook vector, cbvec, must be smaller than a fixed limit, CB_MAXGAIN=1.3:

```
|gain| < CB_MAXGAIN
```

where the gain is computed in the following way:

```
gain = (target*cbvec) / ||cbvec||^2
```

3. For the first stage the dot product of the chosen codebook vector and target must be positive:

```
target*cbvec > 0
```

In practice the above criteria are used in a sequential search through all codebook vectors. The best match is found by registering a new max measure and index whenever the previously registered max measure is surpassed and all other criteria are fulfilled. If none of the codebook vectors fulfill (2) and (3), the first codebook vector is selected.

3.6.4.2 Gain Quantization at Each Stage

The gain follows as a result of the computation:

```
gain = (target*cbvec) / ||cbvec||^2
```

for the optimal codebook vector that was found by the procedure from section 3.6.4.1.

The three stages quantize the gain using 5, 4 and 3 bits respectively. In the first stage, the gain is limited to positive values. This gain is quantized by finding the nearest value in the quantization table gain_sq5Tbl.

```
gain_sq5Tbl[32]={0.037476, 0.075012, 0.112488, 0.150024, 0.187500,
               0.224976, 0.262512, 0.299988, 0.337524, 0.375000,
               0.412476, 0.450012, 0.487488, 0.525024, 0.562500,
               0.599976, 0.637512, 0.674988, 0.712524, 0.750000,
               0.787476, 0.825012, 0.862488, 0.900024, 0.937500,
               0.974976, 1.012512, 1.049988, 1.087524, 1.125000,
               1.162476, 1.200012}
```

The gains of the subsequent two stages can be either positive or negative. The gains are quantized using a quantization table times a scale factor. The second stage uses the table gain_sq4Tbl and the third stage uses gain_sq3Tbl. The scale factor equates 0.1 or the absolute value of the quantized gain representation value obtained in the previous stage, whichever is the larger. Again, the resulting gain index is the index to the nearest value of the quantization table times the scale factor.


```
     gainQ = scaleFact * gain_sqXTbl[index]

gain_sq4Tbl[16]={-1.049988, -0.900024, -0.750000, -0.599976,
               -0.450012, -0.299988, -0.150024, 0.000000, 0.150024,
               0.299988, 0.450012, 0.599976, 0.750000, 0.900024,
               1.049988, 1.200012}

gain_sq3Tbl[8]={-1.000000, -0.659973, -0.330017,0.000000,
               0.250000, 0.500000, 0.750000, 1.00000}
```

3.6.4.3 Preparation of Target for Next Stage

Before performing the search for the next stage the perceptually weighted target vector is updated by subtracting from it the selected codebook vector (from the perceptually weighted codebook) times the corresponding quantized gain.

```
     target[i] = target[i] - gainQ * selected_vec[i];
```

A reference implementation of the codebook encoding is found in Appendix A.34.

3.7 Gain Correction Encoding

The start state is quantized in a relatively model independent manner using 3 bits per sample. In contrast to this, the remaining parts of the block is encoded using an adaptive codebook. This codebook will produce high matching accuracy whenever there is a high correlation between the target and the best codebook vector. For unvoiced speech segments and background noises, this is not necessarily so, which, due to the nature of the squared error criterion, results in a coded signal with less power than the target signal. As the coded start state has good power matching to the target, the result is a power fluctuation within the encoded frame. Perceptually, the main problem with this is that the time envelope of the signal energy becomes unsteady. To overcome this problem, the gains for the codebooks are re-scaled after the codebook encoding by searching for a new gain factor for the first stage codebook that provides better power matching.

First the energy for the target signal, tene, is computed along with the energy for the coded signal, cene, given by the addition of the 3 gain scaled codebook vectors. Since the gains of the 2nd and 3rd stage scale with the gain of the first stage, by changing the first stage gain from gain[0] to gain_sq5Tbl[i], the energy of the coded signal changes from cene to

```
cene*(gain_sq5Tbl[i]*gain_sq5Tbl[i])/(gain[0]*gain[0])
```

where gain[0] is the gain for the first stage found in the original codebook search. A refined search is performed by testing the gain indices i=0 to 31, and as long as the new codebook energy as given above is less than tene, the gain index for stage 1 is increased. A restriction is applied so that the new gain value for stage 1 cannot

be more than 2 times higher than the original value found in the codebook search. Note that by using this method the shape of the encoded vector is not changed, only the gain or amplitude.

3.8 Bitstream Definition

The total number of bits used to describe one frame of 20 ms speech is 304, which fits in 38 bytes and results in a bit rate of 15.20 kbit/s. For the case with a frame length of 30 ms speech the total number of bits used is 400, which fits in 50 bytes and results in a bit rate of 13.33 kbit/s. In the bitstream definition the bits are distributed into three classes according to their bit error or loss sensitivity. The most sensitive bits (class 1) are placed first in the bitstream for each frame. The less sensitive bits (class 2) are placed after the class 1 bits. The least sensitive bits (class 3) are placed at the end of the bitstream for each frame.

Looking at the 20/30 ms frame length cases for each class: The class 1 bits occupy a total of 6/8 bytes (48/64 bits), the class 2 bits occupy 8/12 bytes (64/96 bits), and the class 3 bits occupy 24/30 bytes (191/239 bits). This distribution of the bits enable the use of uneven level protection (ULP) as is exploited in the payload format definition for iLBC [1]. The detailed bit allocation is shown in the table below. When a quantization index is distributed between more classes the more significant bits belong to the lowest class.

Bitstream structure:

Parameter	Bits Class <1,2,3>	
	20 ms frame	30 ms frame

```
----------------------------------+---------------+---------------+
                         Split 1  |   6 <6,0,0>   |   6 <6,0,0>   |
                LSF 1    Split 2  |   7 <7,0,0>   |   7 <7,0,0>   |
LSF                      Split 3  |   7 <7,0,0>   |   7 <7,0,0>   |
                ------------------+---------------+---------------+
                         Split 1  | NA (Not Appl.)|   6 <6,0,0>   |
                LSF 2    Split 2  |      NA       |   7 <7,0,0>   |
                         Split 3  |      NA       |   7 <7,0,0>   |
                ------------------+---------------+---------------+
                Sum               |  20 <20,0,0>  |  40 <40,0,0>  |
----------------------------------+---------------+---------------+
Block Class.                      |   2 <2,0,0>   |   3 <3,0,0>   |
----------------------------------+---------------+---------------+
Position 22 sample segment        |   1 <1,0,0>   |   1 <1,0,0>   |
----------------------------------+---------------+---------------+
Scale Factor State Coder          |   6 <6,0,0>   |   6 <6,0,0>   |
----------------------------------+---------------+---------------+
                Sample 0          |   3 <0,1,2>   |   3 <0,1,2>   |
Quantized       Sample 1          |   3 <0,1,2>   |   3 <0,1,2>   |
Residual           :              |   :    :      |   :    :      |
State              :              |   :    :      |   :    :      |
Samples            :              |   :    :      |   :    :      |
                Sample 56         |   3 <0,1,2>   |   3 <0,1,2>   |
                Sample 57         |      NA       |   3 <0,1,2>   |
                ------------------+---------------+---------------+
                Sum               | 171 <0,57,114>| 174 <0,58,116>|
----------------------------------+---------------+---------------+
                         Stage 1  |   7 <6,0,1>   |   7 <4,2,1>   |
CB for 22/23             Stage 2  |   7 <0,0,7>   |   7 <0,0,7>   |
sample block             Stage 3  |   7 <0,0,7>   |   7 <0,0,7>   |
                ------------------+---------------+---------------+
                Sum               |  21 <6,0,15>  |  21 <4,2,15>  |
----------------------------------+---------------+---------------+
                         Stage 1  |   5 <2,0,3>   |   5 <1,1,3>   |
Gain for 22/23           Stage 2  |   4 <1,1,2>   |   4 <1,1,2>   |
sample block             Stage 3  |   3 <0,0,3>   |   3 <0,0,3>   |
                ------------------+---------------+---------------+
                Sum               |  12 <3,1,8>   |  12 <2,2,8>   |
```

	sub-block 1	Stage 1	8 <7,0,1>	8 <6,1,1>
		Stage 2	7 <0,0,7>	7 <0,0,7>
		Stage 3	7 <0,0,7>	7 <0,0,7>
	sub-block 2	Stage 1	8 <0,0,8>	8 <0,7,1>
		Stage 2	8 <0,0,8>	8 <0,0,8>
Indices for CB sub-blocks		Stage 3	8 <0,0,8>	8 <0,0,8>
	sub-block 3	Stage 1	NA	8 <0,7,1>
		Stage 2	NA	8 <0,0,8>
		Stage 3	NA	8 <0,0,8>
	sub-block 4	Stage 1	NA	8 <0,7,1>
		Stage 2	NA	8 <0,0,8>
		Stage 3	NA	8 <0,0,8>
	Sum		46 <7,0,39>	94 <6,22,66>
	sub-block 1	Stage 1	5 <1,2,2>	5 <1,2,2>
		Stage 2	4 <1,1,2>	4 <1,2,1>
		Stage 3	3 <0,0,3>	3 <0,0,3>
	sub-block 2	Stage 1	5 <1,1,3>	5 <0,2,3>
		Stage 2	4 <0,2,2>	4 <0,2,2>
		Stage 3	3 <0,0,3>	3 <0,0,3>
Gains for sub-blocks	sub-block 3	Stage 1	NA	5 <0,1,4>
		Stage 2	NA	4 <0,1,3>
		Stage 3	NA	3 <0,0,3>
	sub-block 4	Stage 1	NA	5 <0,1,4>
		Stage 2	NA	4 <0,1,3>
		Stage 3	NA	3 <0,0,3>
	Sum		24 <3,6,15>	48 <2,12,34>
Empty frame indicator			1 <0,0,1>	1 <0,0,1>
SUM			304 <48,64,192>	400 <64,96,240>

Table 3.2. The bitstream definition for iLBC for both the 20 ms frame size mode and the 30 ms frame size mode.

When packetized into the payload the bits MUST be sorted as: All the class 1 bits in the order (from top to bottom) as they were specified in the table, all the class 2 bits (from top to bottom) and finally all the class 3 bits in the same sequential order. The last bit, the empty frame indicator, SHOULD be set to zero by the encoder. If this bit is set to one the decoder SHOULD treat the data as a lost frame. For example this bit can be set to 1 to indicate lost frame for file storage format as in [1].

4. DECODER PRINCIPLES

This section describes the principles of each component of the decoder algorithm.

```
            +-------------+    +--------+    +---------------+
 payload -> | 1. Get para | -> | 2. LPC | -> | 3. Sc Dequant | ->
            +-------------+    +--------+    +---------------+

            +-------------+    +------------------+
         -> | 4 Mem setup | -> | 5. Construct res |------->
         |  +-------------+    +-------------------   |
         ---------<-----------<-----------<------------
                   Sub-frame 0...2/4 (20 ms/30 ms)

            +----------------+    +----------+
         -> | 6. Enhance res | -> | 7. Synth | ------------>
            +----------------+    +----------+

            +-----------------+
         -> | 8. Post Process | ----------------> decoded speech
            +-----------------+
```

Figure 4.1. Flow chart of the iLBC decoder. If a frame was lost steps 1 to 5 SHOULD be replaced by a PLC algorithm.

1. Extract the parameters from the bitstream
2. Decode the LPC and interpolate (section 4.1)
3. Construct the 57/58 sample start state (section 4.2)
4. Set up the memory using data from the decoded residual. This memory is used for codebook construction. For blocks preceding the start state, both the decoded residual and the target are time reversed. Sub-frames are decoded in the same order as they were encoded
5. Construct the residuals of this sub-frame (gain[0]*cbvec[0] + gain[1]*cbvec[1] + gain[2]*cbvec[2]). Repeat 4 and 5 until the residual of all sub-blocks have been constructed
6. Enhance the residual with the post filter (section 4.6)
7. Synthesis of the residual (section 4.7)
8. Post process with HP filter if desired (section 4.8)

4.1 LPC Filter Reconstruction

The decoding of the LP filter parameters is very straightforward. For a set of three/six indices the corresponding LSF vector(s) are found by simple table look up. For each of the LSF vectors the three split vectors are concatenated to obtain qlsf1 and qlsf2, respectively (in the 20 ms mode only one LSF vector, qlsf, is constructed). The next step is the stability check described in Section 3.2.5 followed by the interpolation scheme described in Section 3.2.6 (3.2.7 for 20 ms frames). The only difference is that only the quantized LSFs are known at the decoder and hence the unquantized LSFs are not processed.

A reference implementation of the LPC filter reconstruction is given in Appendix A.36.

4.2 Start State Reconstruction

The scalar encoded STATE_SHORT_LEN=58 (STATE_SHORT_LEN=57 in the 2
0
ms mode) state samples are reconstructed by 1) forming a set of samples (by table look-up) from the index stream idxVec[n] 2) multiplying the set with 1/scal=(10^qmax)/4.5 3) time reversing the 57/58 samples 4) filtering the time reversed block with the dispersion (all-pass) filter used in the encoder (as described in section 3.5.2). This compensates for the phase distortion of the earlier filter operation. 5) Reversing the 57/58 samples from the previous step

```
in(0..(STATE_SHORT_LEN-1)) = time reversed samples from table
                             look-up,
                             idxVecDec((STATE_SHORT_LEN-1)..0)

in(STATE_SHORT_LEN..(2*STATE_SHORT_LEN-1)) = 0

Pk(z) = A~rk(z)/A~k(z), where
                               ___
                               \
A~rk(z)= z^(-LPC_FILTERORDER) + > a~ki*z^(i-(LPC_FILTERORDER-1))
                               /__
                           i=0...(LPC_FILTERORDER-1)

and A~k(z) is taken from the block where the start state begins

in -> Pk(z) -> filtered

out(k) = filtered(STATE_SHORT_LEN-1-k) +
                        filtered(2*STATE_SHORT_LEN-1-k),
```

k=0..(STATE_SHORT_LEN-1)

The remaining 23/22 samples in the state are reconstructed by the same adaptive codebook technique as described in section 4.3. The location bit determines whether these are the first or the last 23/22 samples of the 80 sample state vector. If the remaining 23/22 samples are the first samples of the state vector, then the scalar encoded STATE_SHORT_LEN state samples are time-reversed before initialization of the adaptive codebook memory vector.

A reference implementation of the start state reconstruction is given in Appendix A.44.

4.3 Excitation Decoding Loop

The decoding of the LPC excitation vector proceeds in the same order in which the residual was encoded at the encoder. That is, after the decoding of the entire 80 sample state vector, the forward sub-blocks (corresponding to samples occurring after the state vector samples) are decoded, and then the backward sub-blocks

(corresponding to samples occurring before the state vector) are decoded, resulting in a fully decoded block of excitation signal samples.

In particular, each sub-block is decoded using the multistage adaptive codebook decoding module which is described in section 4.4. This module relies upon an adaptive codebook memory that is constructed before each run of the adaptive codebook decoding. The construction of the adaptive codebook memory in the decoder is identical to the method outlined in section 3.6.3, except that it is done on the codebook memory without perceptual weighting.

For the initial forward sub-block, the last STATE_LEN=80 samples of the length CB_LMEM=147 adaptive codebook memory are filled with the samples of the state vector. For subsequent forward sub-blocks, the first SUBL=40 samples of the adaptive codebook memory are discarded, the remaining samples are shifted by SUBL samples towards the beginning of the vector, while the newly decoded SUBL=40 samples are placed at the end of the adaptive codebook memory. For backward sub-blocks, the construction is similar except that every vector of samples involved is first time-reversed.

A reference implementation of the excitation decoding loop is found in Appendix A.5.

4.4 Multistage Adaptive Codebook Decoding

The Multistage Adaptive Codebook Decoding module is used at both the sender (encoder) and the receiver (decoder) ends to produce a synthetic signal in the residual domain that is eventually used to produce synthetic speech. The module takes the index values used to construct vectors that are scaled and summed together to produce a synthetic signal that is the output of the module.

4.4.1 Construction of the Decoded Excitation Signal

The unpacked index values provided at the input to the module are references to extended codebooks, which are constructed as described in Section 3.6.3 with the only difference that it is based on the codebook memory without the perceptual weighting. The unpacked 3 indices are used to look up 3 codebook vectors. The unpacked 3 gain indices are used to decode the corresponding 3 gains. In this decoding the successive rescaling as described in Section 3.6.4.2 is applied.

A reference implementation of the adaptive codebook decoding is listed in Appendix A.32.

4.5 Packet Loss Concealment

If packet loss occurs, the decoder receives a signal saying that information regarding a block is lost. For such blocks it is RECOMMENDED to use a Packet Loss Concealment (PLC) unit to create a decoded signal which masks the effect of that packet loss. In the following we will describe an example of a PLC unit that can be used with the iLBC codec. As the PLC unit is used only at the decoder, the PLC unit does not affect interoperability between implementations. Other PLC implementations MAY therefore be used.

The example PLC described operates on the LP filters and the excitation signals and is based on the following principles:

4.5.1 Block Received Correctly and Previous Block also Received

If the block is received correctly, the PLC only records state information of the current block that can be used in case the next block is lost. The LP filter coefficients for each sub-block and the entire decoded excitation signal are all saved in the decoder state structure. All this information will be needed if the following block is lost.

4.5.2 Block Not Received

If the block is not received, the block substitution is based on doing a pitch synchronous repetition of the excitation signal which is filtered by the last LP filter of the previous block. The previous block's information is stored in the decoder state structure.

A correlation analysis is performed on the previous block's excitation signal in order to detect the amount of pitch periodicity and a pitch value. The correlation measure is also used to decide on the voicing level (the degree to which the previous block's excitation was a voiced or roughly periodic signal). The excitation in the previous block is used to create an excitation for the block to be substituted such that the pitch of the previous block is maintained. Therefore, the new excitation is constructed in a pitch synchronous manner. In order to avoid a buzzy sounding substituted block, a random excitation is mixed with the new pitch periodic excitation and the relative use of the two components is computed from the correlation measure (voicing level).

For the block to be substituted, the newly constructed excitation signal is then passed through the LP filter to produce the speech that will be substituted for the lost block.

For several consecutive lost blocks, the packet loss concealment continues in a similar manner. The correlation measure of the last received block is still used along with the same pitch value. The LP filters of the last received block are also used again. The energy of the substituted excitation for consecutive lost blocks is

decreased, leading to a dampened excitation, and therefore dampened speech.

4.5.3 Block Received Correctly When Previous Block Not Received

For the case in which a block is received correctly when the previous block was not received, the correctly received block's directly decoded speech (based solely on the received block) is not used as the actual output. The reason for this is that the directly decoded speech does not necessarily smoothly merge into the synthetic speech generated for the previous lost block. If the two signals are not smoothly merged, an audible discontinuity is accidentally produced. Therefore, a correlation analysis between the two blocks of excitation signal (the excitation of the previous concealed block and the excitation of the current received block) is performed to find the best phase match. Then a simple overlap-add procedure is performed to smoothly merge the previous excitation into the current block's excitation.

The exact implementation of the packet loss concealment does not influence interoperability of the codec.

A reference implementation of the packet loss concealment is suggested in Appendix A.14. Exact compliance with this suggested algorithm is not needed for a reference implementation to be fully compatible with the overall codec specification.

4.6 Enhancement

The decoder contains an enhancement unit that operates on the reconstructed excitation signal. The enhancement unit increases the perceptual quality of the reconstructed signal by reducing the speech-correlated noise in the voiced speech segments. Compared to traditional postfilters, the enhancer has the advantage that it can only modify the excitation signal slightly. This means that there is no risk of over enhancement. The enhancer works very similar for both the 20 ms frame size mode and for the 30 ms frame size mode.

For the mode with 20 ms frame size, the enhancer uses a memory of six 80 sample excitation blocks prior in time plus the two new 80 sample excitation blocks. For each block of 160 new unenhanced excitation samples, 160 enhanced excitation samples are produced. The enhanced excitation is 40 sample delayed compared to the unenhanced excitation since the enhancer algorithm uses lookahead.

For the mode with 30 ms frame size, the enhancer uses a memory of five 80 sample excitation blocks prior in time plus the three new 80 sample excitation blocks. For each block of 240 new unenhanced excitation samples, 240 enhanced excitation samples are produced. The enhanced excitation is 80 sample delayed compared to the unenhanced excitation since the enhancer algorithm uses lookahead.

OUTLINE of Enhancer

The speech enhancement unit operates on sub-blocks of 80 samples, which means that there are two/three 80 sample sub-blocks per frame. Each of these two/three sub-blocks is enhanced separately, but in an analogous manner.

```
unenhanced residual
        |
        |   +---------------+    +--------------+
        +-> | 1. Pitch Est  | -> | 2. Find PSSQ | -------->
            +---------------+  | +--------------+
                               +-----<-------<------<--+
            +------------+         enh block 0..1/2    |
         -> | 3. Smooth  |                             |
            +------------+                             |
              \                                        |
              /\                                       |
             /  \   Already                            |
            / 4. \----------->----------->-----------+ |
            \Crit/ Fulfilled                         | |
             \? /                                    v |
              \/                                     | |
               \  +-----------------+    +---------+ | |
           Not +->| 5. Use Constr.  | -> | 6. Mix  | ----->
        Fulfilled +-----------------+    +---------+

         ---------------> enhanced residual
```

Figure 4.2. Flow chart of the enhancer

1. Pitch estimation of each of the two/three new 80 sample blocks
2. Find the pitch-period-synchronous sequence n (for block k) by a search around the estimated pitch value. Do this for
   n=1,2,3,-1,-2,-3
3. Calculate the smoothed residual generated by the 6 pitch-period-synchronous sequences from prior step
4. Check if the smoothed residual satisfies the criterion (section 4.6.4)
5. Use constraint to calculate mixing factor (section 4.6.5)
6. Mix smoothed signal with unenhanced residual (pssq(n) n=0)

The main idea of the enhancer is to find three 80 sample blocks before and three 80 sample blocks after the analyzed unenhanced sub-block and use these to improve the quality of the excitation in that sub-block. The six blocks are chosen so that they have the highest possible correlation with the unenhanced sub-block that is being enhanced. In other words the 6 blocks are pitch-period-synchronous sequences to the unenhanced sub-block.

A linear combination of the six pitch-period-synchronous sequences is calculated that approximates the sub-block. If the squared error between the approximation and the unenhanced sub-block is small enough, the enhanced residual is set equal to this approximation. For the cases when the squared error criterion is not fulfilled, a

linear combination of the approximation and the unenhanced residual forms the enhanced residual.

4.6.1 Estimating the pitch

Pitch estimates are needed to determine the locations of the pitch-period-synchronous sequences in a complexity efficient way. For each of the new two/three sub-blocks a pitch estimate is calculated by finding the maximum correlation in the range from lag 20 to lag 120.

These pitch estimates are used to narrow down the search for the best possible pitch-period-synchronous sequences.

4.6.2 Determination of the Pitch-Synchronous Sequences

Upon receiving the pitch estimates from the prior step, the enhancer analyzes and enhances one 80 sample sub-block at a time. The pitch-period-synchronous-sequences pssq(n) can be viewed as vectors of length 80 samples each shifted n*lag samples from the current sub-block. The six pitch-period-synchronous-sequences, pssq(-3) to pssq(-1) and pssq(1) to pssq(3), are found one at a time by the steps below:

1) Calculate the estimate of the position of the pssq(n). For pssq(n) in front of pssq(0) (n > 0), the location of the pssq(n) is estimated by moving one pitch estimate forward in time from the exact location of pssq(n-1). Similarly for pssq(n) behind pssq(0) (n < 0) is estimated by moving one pitch estimate backward in time from the exact location of pssq(n+1). If the estimated pssq(n) vector location is totally within the enhancer memory (figure 4.3) step 2,3, and 4 are performed, otherwise the pssq(n) is set to zeros.

2) Compute the correlation between the unenhanced excitation and vectors around the estimated location interval of pssq(n). The correlation is calculated in the interval estimated location +/- 2 samples. This results in 5 correlation values.

3) The 5 correlation values are upsampled by a factor 4, using sinc upsampling filters (four MA filters with coefficients upsFilter1

.. upsFilter4). Within these the maximum value is found, which specifies the best pitch-period with a resolution of a quarter of a sample.

```
  upsFilter1[7]={0.000000 0.000000 0.000000 1.000000
         0.000000 0.000000 0.000000}
  upsFilter2[7]={0.015625 -0.076904 0.288330 0.862061
        -0.106445 0.018799 -0.015625}
  upsFilter3[7]={0.023682 -0.124268 0.601563 0.601563
        -0.124268 0.023682 -0.023682}
  upsFilter4[7]={0.018799 -0.106445 0.862061 0.288330
        -0.076904 0.015625 -0.018799}
```


4) Generate the pssq(n) vector by upsampling of the excitation memory and extracting the sequence that corresponds to the lag delay that was calculated in prior step.

With the steps above all the pssq(n) can be found in an iterative manner, first moving backward in time from pssq(0) and then forward in time from pssq(0).

```
0              159             319             479             639
+---------------------------------------------------------------+
|  -5   |  -4   |  -3   |  -2   |  -1   |   0   |   1   |   2   |
+---------------------------------------------------------------+
                                            |pssq 0 |
                                       |pssq -1| |pssq 1 |
                                    |pssq -2|       |pssq 2 |
                                 |pssq -3|             |pssq 3 |
```

Figure 4.3. Enhancement for 20 ms frame size

Figure 4.3. depicts pitch-period-synchronous sequences in the enhancement of the first 80 sample block in the 20 ms frame size mode. The unenhanced signal input is stored in the two last sub-blocks (1-2), and the six other sub-blocks contain unenhanced residual prior-in-time. We perform the enhancement algorithm on two blocks of 80 samples, where the first of the two blocks consist of the last 40 samples of sub-block 0 and the first 40 samples of sub-block 1. The second 80 sample block consists of the last 40 samples of sub-block 1 and the first 40 samples of sub-block 2.

```
0              159             319             479             639
```

```
+-----------------------------------------------------------------+
|  -4   |  -3   |  -2   |  -1   |   0   |   1   |   2   |   3   |
+-----------------------------------------------------------------+
                                |pssq 0 |
                           |pssq -1| |pssq 1 |
                        |pssq -2|       |pssq 2 |
                     |pssq -3|             |pssq 3 |
```

Figure 4.4. Enhancement for 30 ms frame size

Figure 4.4. depicts pitch-period-synchronous sequences in the enhancement of the first 80 sample block in the 30 ms frame size mode. The unenhanced signal input is stored in the three last sub-blocks (1-3). The five other sub-blocks contain unenhanced residual prior-in-time. The enhancement algorithm is performed on the three 80 sample sub-blocks 0, 1 and 2.

4.6.3 Calculation of the smoothed excitation

A linear combination of the six pssq(n) (n!=0) form a smoothed approximation, z, of pssq(0). Most of the weight is put on the sequences that are close to pssq(0) since these are most likely to be most similar to pssq(0). The smoothed vector is also rescaled, so that the energy of z is the same as the energy of pssq(0).


```
      ___
      \
   y = > pssq(i) * pssq_weight(i)
      /__
   i=-3,-2,-1,1,2,3

   pssq_weight(i) = 0.5*(1-cos(2*pi*(i+4)/(2*3+2)))

   z = C * y, where C = ||pssq(0)||/||y||
```

4.6.4 Enhancer criterion

The criterion of the enhancer is that the enhanced excitation is not allowed to differ much from the unenhanced excitation. This criterion is checked for each 80 sample sub-block.

```
   e < (b * ||pssq(0)||^2), where b=0.05 and   (Constraint 1)

   e = (pssq(0)-z)*(pssq(0)-z), and "*" means the dot product
```

4.6.5 Enhancing the excitation

From the criterion in the previous section it is clear that the excitation is not allowed to change much. The purpose of this constraint is to prevent the creation of an enhanced signal that is significantly different from the original signal. This also means that the constraint limits the numerical size of the errors that the enhancement procedure can make. That is especially important in unvoiced segments and background noise segments where increased periodicity could lead to lower perceived quality.

When the constraint in the prior section is not met, the enhanced residual is instead calculated through a constrained optimization using the Lagrange multiplier technique. The new constraint is that:

```
e = (b * ||pssq(0)||^2)                     (Constraint 2)
```

We distinguish two solution regions for the optimization: 1) the region where the first constraint is fulfilled and 2) the region where the first constraint is not fulfilled so the second constraint must be used.

In the first case, where the second constraint is not needed, the optimized re-estimated vector is simply z, the energy scaled version of y.

In the second case, where the second constraint is activated and becomes an equality constraint, we have that

```
z= A*y + B*pssq(0)
```

where


```
A = sqrt((b-b^2/4)*(w00*w00)/ (w11*w00 + w10*w10)) and

w11 = pssq(0)*pssq(0)
w00 = y*y
w10 = y*pssq(0)    (* symbolizes the dot product)
```

and

```
B = 1 - b/2 - A * w10/w00
```

Appendix A.16 contains a listing of a reference implementation for the enhancement method.

4.7 Synthesis Filtering

Upon decoding or PLC of the LP excitation block, the decoded speech block is obtained by running the decoded LP synthesis filter, 1/A~k(z), over the block. The synthesis filters have to be shifted to compensate for the delay in the enhancer. For 20 ms frame size mode they SHOULD be shifted one 40 sample sub-block and for 30 ms frame size mode they SHOULD be shifted two 40 sample sub-blocks. The LP coefficients SHOULD be changed at the first sample of every sub-block while keeping the filter state. For PLC blocks, one solution is to apply the last LP coefficients of the last decoded speech block for all sub-blocks.

The reference implementation for the synthesis filtering can be found in Appendix A.48.

4.8 Post Filtering

If desired the decoded block can be filtered by a high-pass filter. This removes the low frequencies of the decoded signal. A reference implementation of this, with cut off at 65 Hz, is shown in Appendix A.30.

5. IANA CONSIDERATIONS

This algorithm for the coding of speech signals does not have any IANA considerations that need to be addressed.

6. SECURITY CONSIDERATIONS

This algorithm for the coding of speech signals is not subject of any known security consideration; however, its RTP payload format [1] is subject of several considerations which are addressed there. Confidentiality of the media streams is achieved by encryption, therefore external mechanisms, such as SRTP [5], MAY be used for that purpose.

7. EVALUATION OF THE ILBC IMPLEMENTATIONS

It is possible and suggested to evaluate certain iLBC implementation by utilizing methodology and tools available at http://www.ilbcfreeware.org/evaluation.html

8. REFERENCES

8.1 Normative

[1] A. Duric and S. V. Andersen, "RTP Payload Format for iLBC Speech", IETF Draft, May 2004

[2] S. Bradner, "Key words for use in RFCs to Indicate requirement Levels", BCP 14, RFC 2119, March 1997.

[3] PacketCable(TM) Audio/Video Codecs Specification, Cable Television Laboratories, Inc.

8.2 Informative

[4] ITU-T Recommendation G.711, available online from the ITU bookstore at http://www.itu.int.

[5] Baugher, et al., "The Secure Real Time Transport Protocol", IETF RFC 3711, March 2004.

9. ACKNOWLEDGEMENTS

This extensive work, beside listed authors, has the following authors, which could not been listed among "official" authors (due to IESG confines in number of authors which can be listed):

Manohar N Murthi (Dpt of El and Comp Eng, University of Miami), Fredrik Galschiodt, Julian Spittka and Jan Skoglund (Global IP Sound)

The authors are deeply indebted to them all and thank them sincerely:

Henry Sinnreich, Patrik Faltstrom and Alan Johnston

for great support of the iLBC initiative and for valuable feedback and comments.

Peter Vary, Frank Mertz and Christoph Erdmann (RWTH Aachen); Vladimir Cuperman (Niftybox LLC); Thomas Eriksson (Chalmers Univ of Tech) and Gernot Kubin (TU Graz)

for thorough review of the iLBC draft and their valuable feedback and remarks.

10. AUTHOR'S ADDRESSES

Soren Vang Andersen
Department of Communication Technology
Aalborg University
Fredrik Bajers Vej 7A
9200 Aalborg
Denmark
Phone: ++45 9 6358627
Email: sva@kom.auc.dk

Alan Duric
Telio AS
Stortingsgt. 8
Oslo, N-0161
Norway
Phone: +47 21673555
Email: alan.duric@telio.no

Henrik Astrom
Global IP Sound AB
Rosenlundsgatan 54
Stockholm, S-11863
Sweden
Phone: +46 8 54553040
Email: henrik.astrom@globalipsound.com

Roar Hagen
Global IP Sound AB
Rosenlundsgatan 54
Stockholm, S-11863
Sweden
Phone: +46 8 54553040
Email: roar.hagen@globalipsound.com

W. Bastiaan Kleijn
Global IP Sound AB
Rosenlundsgatan 54
Stockholm, S-11863
Sweden
Phone: +46 8 54553040
Email: bastiaan.kleijn@globalipsound.com

Jan Linden

Global IP Sound Inc.
900 Kearny Street, suite 500
San Francisco, CA-94133
USA
Phone: +1 415 397 2555
Email: jan.linden@globalipsound.com

Full Copyright Statement

Intellectual Property

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

APPENDIX A REFERENCE IMPLEMENTATION

This appendix contains the complete c-code for a reference implementation of encoder and decoder for the specified codec.

The c-code consists of the following files with highest level functions:

iLBC_test.c: main function for evaluation purpose
iLBC_encode.h: encoder header
iLBC_encode.c: encoder function
iLBC_decode.h: decoder header
iLBC_decode.c: decoder function

the following files containing global defines and constants:

iLBC_define.h: global defines
constants.h: global constants header
constants.c: global constants memory allocations

and the following files containing subroutines:

anaFilter.h: lpc analysis filter header
anaFilter.c: lpc analysis filter function
createCB.h: codebook construction header

```
      createCB.c: codebook construction function
      doCPLC.h: packet loss concealment header
      doCPLC.c: packet loss concealment function
      enhancer.h: signal enhancement header
      enhancer.c: signal enhancement function
      filter.h: general filter header
      filter.c: general filter functions
      FrameClassify.h: start state classification header
      FrameClassify.c: start state classification function
      gainquant.h: gain quantization header
      gainquant.c: gain quantization function
      getCBvec.h: codebook vector construction header
      getCBvec.c: codebook vector construction function
      helpfun.h: general purpose header
      helpfun.c: general purpose functions
      hpInput.h: input high pass filter header
      hpInput.c: input high pass filter function
      hpOutput.h: output high pass filter header
      hpOutput.c: output high pass filter function
      iCBConstruct.h: excitation decoding header
      iCBConstruct.c: excitation decoding function
      iCBSearch.h: excitation encoding header
      iCBSearch.c: excitation encoding function
      LPCdecode.h: lpc decoding header
      LPCdecode.c: lpc decoding function
      LPCencode.h: lpc encoding header
      LPCencode.c: lpc encoding function
      lsf.h: line spectral frequencies header
      lsf.c: line spectral frequencies functions
```



```
      packing.h: bitstream packetization header
      packing.c: bitstream packetization functions
      StateConstructW.h: state decoding header
      StateConstructW.c: state decoding functions
      StateSearchW.h: state encoding header
      StateSearchW.c: state encoding function
      syntFilter.h: lpc synthesis filter header
      syntFilter.c: lpc synthesis filter function
```

The implementation is portable and should work on many different platforms. However, it is not difficult to optimize the implementation on particular platforms, an exercise left to the reader.

A.1 iLBC_test.c

```
/******************************************************************
```

```
    iLBC Speech Coder ANSI-C Source Code

    iLBC_test.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "iLBC_define.h"
#include "iLBC_encode.h"
#include "iLBC_decode.h"

/* Runtime statistics */
#include <time.h>

#define ILBCNOOFWORDS_MAX   (NO_OF_BYTES_30MS/2)

/*----------------------------------------------------------------*
 *  Encoder interface function
 *---------------------------------------------------------------*/

short encode(   /* (o) Number of bytes encoded */
    iLBC_Enc_Inst_t *iLBCenc_inst,
                                /* (i/o) Encoder instance */
    short *encoded_data,    /* (o) The encoded bytes */
    short *data                 /* (i) The signal block to encode*/
){
    float block[BLOCKL_MAX];
    int k;
```



```
    /* convert signal to float */

    for (k=0; k<iLBCenc_inst->blockl; k++)
        block[k] = (float)data[k];

    /* do the actual encoding */

    iLBC_encode((unsigned char *)encoded_data, block, iLBCenc_inst);


    return (iLBCenc_inst->no_of_bytes);
}
```

```
/*----------------------------------------------------------------*
 *  Decoder interface function
 *---------------------------------------------------------------*/

short decode(       /* (o) Number of decoded samples */
    iLBC_Dec_Inst_t *iLBCdec_inst,  /* (i/o) Decoder instance */
    short *decoded_data,        /* (o) Decoded signal block*/
    short *encoded_data,        /* (i) Encoded bytes */
    short mode                       /* (i) 0=PL, 1=Normal */
){
    int k;
    float decblock[BLOCKL_MAX], dtmp;

    /* check if mode is valid */

    if (mode<0 || mode>1) {
        printf("\nERROR - Wrong mode - 0, 1 allowed\n"); exit(3);}

    /* do actual decoding of block */

    iLBC_decode(decblock, (unsigned char *)encoded_data,
        iLBCdec_inst, mode);

    /* convert to short */

    for (k=0; k<iLBCdec_inst->blockl; k++){
        dtmp=decblock[k];

        if (dtmp<MIN_SAMPLE)
            dtmp=MIN_SAMPLE;
        else if (dtmp>MAX_SAMPLE)
            dtmp=MAX_SAMPLE;
        decoded_data[k] = (short) dtmp;
    }

    return (iLBCdec_inst->blockl);
}

/*---------------------------------------------------------------*
 *  Main program to test iLBC encoding and decoding
 *
```



```
 *  Usage:
 *    exefile_name.exe <infile> <bytefile> <outfile> <channel>
 *
 *    <infile>   : Input file, speech for encoder (16-bit pcm file)
 *    <bytefile> : Bit stream output from the encoder
 *    <outfile>  : Output file, decoded speech (16-bit pcm file)
```

```
 *    <channel>  : Bit error file, optional (16-bit)
 *                     1 - Packet received correctly
 *                     0 - Packet Lost
 *
 *--------------------------------------------------------------*/

int main(int argc, char* argv[])
{

    /* Runtime statistics */

    float starttime;
    float runtime;
    float outtime;

    FILE *ifileid,*efileid,*ofileid, *cfileid;
    short data[BLOCKL_MAX];
    short encoded_data[ILBCNOOFWORDS_MAX], decoded_data[BLOCKL_MAX];
    int len;
    short pli, mode;
    int blockcount = 0;
    int packetlosscount = 0;

    /* Create structs */
    iLBC_Enc_Inst_t Enc_Inst;
    iLBC_Dec_Inst_t Dec_Inst;

    /* get arguments and open files */

    if ((argc!=5) && (argc!=6)) {
        fprintf(stderr,
        "\n*-----------------------------------------------*\n");
        fprintf(stderr,
        "   %s <20,30> input encoded decoded (channel)\n\n",
            argv[0]);
        fprintf(stderr,
        "   mode    : Frame size for the encoding/decoding\n");
        fprintf(stderr,
        "                 20 - 20 ms\n");
        fprintf(stderr,
        "                 30 - 30 ms\n");
        fprintf(stderr,
        "   input   : Speech for encoder (16-bit pcm file)\n");
        fprintf(stderr,
        "   encoded : Encoded bit stream\n");
        fprintf(stderr,
        "   decoded : Decoded speech (16-bit pcm file)\n");
        fprintf(stderr,
```

```
    "   channel : Packet loss pattern, optional (16-bit)\n");
    fprintf(stderr,
    "                  1 - Packet received correctly\n");
    fprintf(stderr,
    "                  0 - Packet Lost\n");
    fprintf(stderr,
    "*-----------------------------------------------*\n\n");
    exit(1);
}
mode=atoi(argv[1]);
if (mode != 20 && mode != 30) {
    fprintf(stderr,"Wrong mode %s, must be 20, or 30\n",
        argv[1]);
    exit(2);
}
if ( (ifileid=fopen(argv[2],"rb")) == NULL) {
    fprintf(stderr,"Cannot open input file %s\n", argv[2]);
    exit(2);}
if ( (efileid=fopen(argv[3],"wb")) == NULL) {
    fprintf(stderr, "Cannot open encoded file file %s\n",
        argv[3]); exit(1);}
if ( (ofileid=fopen(argv[4],"wb")) == NULL) {
    fprintf(stderr, "Cannot open decoded file %s\n",
        argv[4]); exit(1);}
if (argc==6) {
    if( (cfileid=fopen(argv[5],"rb")) == NULL) {
        fprintf(stderr, "Cannot open channel file %s\n",
            argv[5]);
        exit(1);
    }
} else {
    cfileid=NULL;
}

/* print info */

fprintf(stderr, "\n");
fprintf(stderr,
    "*---------------------------------------------------*\n");
fprintf(stderr,
    "*                                                   *\n");
fprintf(stderr,
    "*      iLBC test program                            *\n");
fprintf(stderr,
    "*                                                   *\n");
fprintf(stderr,
    "*                                                   *\n");
fprintf(stderr,
    "*---------------------------------------------------*\n");
fprintf(stderr,"\nMode           : %2d ms\n", mode);
```

```
    fprintf(stderr,"Input file     : %s\n", argv[2]);
    fprintf(stderr,"Encoded file   : %s\n", argv[3]);
    fprintf(stderr,"Output file    : %s\n", argv[4]);
    if (argc==6) {
```



```
        fprintf(stderr,"Channel file   : %s\n", argv[5]);
    }
    fprintf(stderr,"\n");

    /* Initialization */

    initEncode(&Enc_Inst, mode);
    initDecode(&Dec_Inst, mode, 1);

    /* Runtime statistics */

    starttime=clock()/(float)CLOCKS_PER_SEC;

    /* loop over input blocks */

    while (fread(data,sizeof(short),Enc_Inst.blockl,ifileid)==
            Enc_Inst.blockl) {

        blockcount++;

        /* encoding */

        fprintf(stderr, "--- Encoding block %i --- ",blockcount);
        len=encode(&Enc_Inst, encoded_data, data);
        fprintf(stderr, "\r");

        /* write byte file */

        fwrite(encoded_data, sizeof(unsigned char), len, efileid);

        /* get channel data if provided */
        if (argc==6) {
            if (fread(&pli, sizeof(short), 1, cfileid)) {
                if ((pli!=0)&&(pli!=1)) {
                    fprintf(stderr, "Error in channel file\n");
                    exit(0);
                }
                if (pli==0) {
                    /* Packet loss -> remove info from frame */
                    memset(encoded_data, 0,
                        sizeof(short)*ILBCNOOFWORDS_MAX);
                    packetlosscount++;
                }
            } else {
```

```
                fprintf(stderr, "Error. Channel file too short\n");
                exit(0);
            }
        } else {
            pli=1;
        }

        /* decoding */

        fprintf(stderr, "--- Decoding block %i --- ",blockcount);
```



```
        len=decode(&Dec_Inst, decoded_data, encoded_data, pli);
        fprintf(stderr, "\r");

        /* write output file */

        fwrite(decoded_data,sizeof(short),len,ofileid);
    }

    /* Runtime statistics */

    runtime = (float)(clock()/(float)CLOCKS_PER_SEC-starttime);
    outtime = (float)((float)blockcount*(float)mode/1000.0);
    printf("\n\nLength of speech file: %.1f s\n", outtime);
    printf("Packet loss          : %.1f%%\n",
        100.0*(float)packetlosscount/(float)blockcount);
    printf("Time to run iLBC     :");
    printf(" %.1f s (%.1f %% of realtime)\n\n", runtime,
        (100*runtime/outtime));

    /* close files */

    fclose(ifileid);  fclose(efileid); fclose(ofileid);
    if (argc==6) {
        fclose(cfileid);
    }
    return(0);
}
```

A.2 iLBC_encode.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    iLBC_encode.h

    Copyright (C) The Internet Society (2004).
```

```
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_ILBCENCODE_H
#define __iLBC_ILBCENCODE_H

#include "iLBC_define.h"

short initEncode(                   /* (o) Number of bytes
                                       encoded */
    iLBC_Enc_Inst_t *iLBCenc_inst,  /* (i/o) Encoder instance */
    int mode                    /* (i) frame size mode */
);

void iLBC_encode(
```



```
    unsigned char *bytes,           /* (o) encoded data bits iLBC */
    float *block,                   /* (o) speech vector to
                                       encode */
    iLBC_Enc_Inst_t *iLBCenc_inst   /* (i/o) the general encoder
                                       state */
);

#endif
```

A.3 iLBC_encode.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    iLBC_encode.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <math.h>
#include <stdlib.h>
#include <string.h>

#include "iLBC_define.h"
#include "LPCencode.h"
#include "FrameClassify.h"
#include "StateSearchW.h"
```

```
#include "StateConstructW.h"
#include "helpfun.h"
#include "constants.h"
#include "packing.h"
#include "iCBSearch.h"
#include "iCBConstruct.h"
#include "hpInput.h"
#include "anaFilter.h"
#include "syntFilter.h"

/*----------------------------------------------------------------*
 *  Initiation of encoder instance.
 *---------------------------------------------------------------*/

short initEncode(                   /* (o) Number of bytes
                                       encoded */
    iLBC_Enc_Inst_t *iLBCenc_inst,  /* (i/o) Encoder instance */
    int mode                    /* (i) frame size mode */
){
    iLBCenc_inst->mode = mode;
    if (mode==30) {
        iLBCenc_inst->blockl = BLOCKL_30MS;
        iLBCenc_inst->nsub = NSUB_30MS;
```



```
        iLBCenc_inst->nasub = NASUB_30MS;
        iLBCenc_inst->lpc_n = LPC_N_30MS;
        iLBCenc_inst->no_of_bytes = NO_OF_BYTES_30MS;
        iLBCenc_inst->no_of_words = NO_OF_WORDS_30MS;
        iLBCenc_inst->state_short_len=STATE_SHORT_LEN_30MS;
        /* ULP init */
        iLBCenc_inst->ULP_inst=&ULP_30msTbl;
    }
    else if (mode==20) {
        iLBCenc_inst->blockl = BLOCKL_20MS;
        iLBCenc_inst->nsub = NSUB_20MS;
        iLBCenc_inst->nasub = NASUB_20MS;
        iLBCenc_inst->lpc_n = LPC_N_20MS;
        iLBCenc_inst->no_of_bytes = NO_OF_BYTES_20MS;
        iLBCenc_inst->no_of_words = NO_OF_WORDS_20MS;
        iLBCenc_inst->state_short_len=STATE_SHORT_LEN_20MS;
        /* ULP init */
        iLBCenc_inst->ULP_inst=&ULP_20msTbl;
    }
    else {
        exit(2);
    }

    memset((*iLBCenc_inst).anaMem, 0,
```

```
        LPC_FILTERORDER*sizeof(float));
    memcpy((*iLBCenc_inst).lsfold, lsfmeanTbl,
        LPC_FILTERORDER*sizeof(float));
    memcpy((*iLBCenc_inst).lsfdeqold, lsfmeanTbl,
        LPC_FILTERORDER*sizeof(float));
    memset((*iLBCenc_inst).lpc_buffer, 0,
        (LPC_LOOKBACK+BLOCKL_MAX)*sizeof(float));
    memset((*iLBCenc_inst).hpimem, 0, 4*sizeof(float));

    return (iLBCenc_inst->no_of_bytes);
}

/*----------------------------------------------------------------*
 *  main encoder function
 *---------------------------------------------------------------*/

void iLBC_encode(
    unsigned char *bytes,           /* (o) encoded data bits iLBC */
    float *block,                   /* (o) speech vector to
                                           encode */
    iLBC_Enc_Inst_t *iLBCenc_inst   /* (i/o) the general encoder
                                           state */
){

    float data[BLOCKL_MAX];
    float residual[BLOCKL_MAX], reverseResidual[BLOCKL_MAX];

    int start, idxForMax, idxVec[STATE_LEN];
    float reverseDecresidual[BLOCKL_MAX], mem[CB_MEML];
    int n, k, meml_gotten, Nfor, Nback, i, pos;
```



```
    int gain_index[CB_NSTAGES*NASUB_MAX],
        extra_gain_index[CB_NSTAGES];
    int cb_index[CB_NSTAGES*NASUB_MAX],extra_cb_index[CB_NSTAGES];
    int lsf_i[LSF_NSPLIT*LPC_N_MAX];
    unsigned char *pbytes;
    int diff, start_pos, state_first;
    float en1, en2;
    int index, ulp, firstpart;
    int subcount, subframe;
    float weightState[LPC_FILTERORDER];
    float syntdenum[NSUB_MAX*(LPC_FILTERORDER+1)];
    float weightdenum[NSUB_MAX*(LPC_FILTERORDER+1)];
    float decresidual[BLOCKL_MAX];

    /* high pass filtering of input signal if such is not done
           prior to calling this function */
```

```
    hpInput(block, iLBCenc_inst->blockl,
                data, (*iLBCenc_inst).hpimem);

    /* otherwise simply copy */

    /*memcpy(data,block,iLBCenc_inst->blockl*sizeof(float));*/

    /* LPC of hp filtered input data */

    LPCencode(syntdenum, weightdenum, lsf_i, data, iLBCenc_inst);


    /* inverse filter to get residual */

    for (n=0; n<iLBCenc_inst->nsub; n++) {
        anaFilter(&data[n*SUBL], &syntdenum[n*(LPC_FILTERORDER+1)],
            SUBL, &residual[n*SUBL], iLBCenc_inst->anaMem);
    }

    /* find state location */

    start = FrameClassify(iLBCenc_inst, residual);

    /* check if state should be in first or last part of the
    two subframes */

    diff = STATE_LEN - iLBCenc_inst->state_short_len;
    en1 = 0;
    index = (start-1)*SUBL;
    for (i = 0; i < iLBCenc_inst->state_short_len; i++) {
        en1 += residual[index+i]*residual[index+i];
    }
    en2 = 0;
    index = (start-1)*SUBL+diff;
    for (i = 0; i < iLBCenc_inst->state_short_len; i++) {
        en2 += residual[index+i]*residual[index+i];
    }
```



```
    if (en1 > en2) {
        state_first = 1;
        start_pos = (start-1)*SUBL;
    } else {
        state_first = 0;
        start_pos = (start-1)*SUBL + diff;
    }

    /* scalar quantization of state */
```

```
    StateSearchW(iLBCenc_inst, &residual[start_pos],
        &syntdenum[(start-1)*(LPC_FILTERORDER+1)],
        &weightdenum[(start-1)*(LPC_FILTERORDER+1)], &idxForMax,
        idxVec, iLBCenc_inst->state_short_len, state_first);

    StateConstructW(idxForMax, idxVec,
        &syntdenum[(start-1)*(LPC_FILTERORDER+1)],
        &decresidual[start_pos], iLBCenc_inst->state_short_len);

    /* predictive quantization in state */

    if (state_first) { /* put adaptive part in the end */

        /* setup memory */

        memset(mem, 0,
            (CB_MEML-iLBCenc_inst->state_short_len)*sizeof(float));
        memcpy(mem+CB_MEML-iLBCenc_inst->state_short_len,
            decresidual+start_pos,
            iLBCenc_inst->state_short_len*sizeof(float));
        memset(weightState, 0, LPC_FILTERORDER*sizeof(float));

        /* encode sub-frames */

        iCBSearch(iLBCenc_inst, extra_cb_index, extra_gain_index,
            &residual[start_pos+iLBCenc_inst->state_short_len],
            mem+CB_MEML-stMemLTbl,
            stMemLTbl, diff, CB_NSTAGES,
            &weightdenum[start*(LPC_FILTERORDER+1)],
            weightState, 0);

        /* construct decoded vector */

        iCBConstruct(
            &decresidual[start_pos+iLBCenc_inst->state_short_len],
            extra_cb_index, extra_gain_index,
            mem+CB_MEML-stMemLTbl,
            stMemLTbl, diff, CB_NSTAGES);

    }
    else { /* put adaptive part in the beginning */
```



```
        /* create reversed vectors for prediction */

        for (k=0; k<diff; k++) {
            reverseResidual[k] = residual[(start+1)*SUBL-1
                -(k+iLBCenc_inst->state_short_len)];
```

```
        }

        /* setup memory */

        meml_gotten = iLBCenc_inst->state_short_len;
        for (k=0; k<meml_gotten; k++) {
            mem[CB_MEML-1-k] = decresidual[start_pos + k];
        }
        memset(mem, 0, (CB_MEML-k)*sizeof(float));
        memset(weightState, 0, LPC_FILTERORDER*sizeof(float));

        /* encode sub-frames */

        iCBSearch(iLBCenc_inst, extra_cb_index, extra_gain_index,
            reverseResidual, mem+CB_MEML-stMemLTbl, stMemLTbl,
            diff, CB_NSTAGES,
            &weightdenum[(start-1)*(LPC_FILTERORDER+1)],
            weightState, 0);

        /* construct decoded vector */

        iCBConstruct(reverseDecresidual, extra_cb_index,
            extra_gain_index, mem+CB_MEML-stMemLTbl, stMemLTbl,
            diff, CB_NSTAGES);

        /* get decoded residual from reversed vector */

        for (k=0; k<diff; k++) {
            decresidual[start_pos-1-k] = reverseDecresidual[k];
        }
    }

    /* counter for predicted sub-frames */

    subcount=0;

    /* forward prediction of sub-frames */

    Nfor = iLBCenc_inst->nsub-start-1;


    if ( Nfor > 0 ) {

        /* setup memory */

        memset(mem, 0, (CB_MEML-STATE_LEN)*sizeof(float));
        memcpy(mem+CB_MEML-STATE_LEN, decresidual+(start-1)*SUBL,
            STATE_LEN*sizeof(float));
        memset(weightState, 0, LPC_FILTERORDER*sizeof(float));
```

```
    /* loop over sub-frames to encode */

    for (subframe=0; subframe<Nfor; subframe++) {

        /* encode sub-frame */

        iCBSearch(iLBCenc_inst, cb_index+subcount*CB_NSTAGES,
            gain_index+subcount*CB_NSTAGES,
            &residual[(start+1+subframe)*SUBL],
            mem+CB_MEML-memLfTbl[subcount],
            memLfTbl[subcount], SUBL, CB_NSTAGES,
            &weightdenum[(start+1+subframe)*
                        (LPC_FILTERORDER+1)],
            weightState, subcount+1);

        /* construct decoded vector */

        iCBConstruct(&decresidual[(start+1+subframe)*SUBL],
            cb_index+subcount*CB_NSTAGES,
            gain_index+subcount*CB_NSTAGES,
            mem+CB_MEML-memLfTbl[subcount],
            memLfTbl[subcount], SUBL, CB_NSTAGES);

        /* update memory */

        memcpy(mem, mem+SUBL, (CB_MEML-SUBL)*sizeof(float));
        memcpy(mem+CB_MEML-SUBL,
            &decresidual[(start+1+subframe)*SUBL],
            SUBL*sizeof(float));
        memset(weightState, 0, LPC_FILTERORDER*sizeof(float));

        subcount++;
    }
}


/* backward prediction of sub-frames */

Nback = start-1;


if ( Nback > 0 ) {

    /* create reverse order vectors */

    for (n=0; n<Nback; n++) {
        for (k=0; k<SUBL; k++) {
            reverseResidual[n*SUBL+k] =
                residual[(start-1)*SUBL-1-n*SUBL-k];
            reverseDecresidual[n*SUBL+k] =
```

```
                decresidual[(start-1)*SUBL-1-n*SUBL-k];
        }
    }
```



```
    /* setup memory */

    meml_gotten = SUBL*(iLBCenc_inst->nsub+1-start);


    if ( meml_gotten > CB_MEML ) {
        meml_gotten=CB_MEML;
    }
    for (k=0; k<meml_gotten; k++) {
        mem[CB_MEML-1-k] = decresidual[(start-1)*SUBL + k];
    }
    memset(mem, 0, (CB_MEML-k)*sizeof(float));
    memset(weightState, 0, LPC_FILTERORDER*sizeof(float));

    /* loop over sub-frames to encode */

    for (subframe=0; subframe<Nback; subframe++) {

        /* encode sub-frame */

        iCBSearch(iLBCenc_inst, cb_index+subcount*CB_NSTAGES,
            gain_index+subcount*CB_NSTAGES,
            &reverseResidual[subframe*SUBL],
            mem+CB_MEML-memLfTbl[subcount],
            memLfTbl[subcount], SUBL, CB_NSTAGES,
            &weightdenum[(start-2-subframe)*
                        (LPC_FILTERORDER+1)],
            weightState, subcount+1);

        /* construct decoded vector */

        iCBConstruct(&reverseDecresidual[subframe*SUBL],
            cb_index+subcount*CB_NSTAGES,
            gain_index+subcount*CB_NSTAGES,
            mem+CB_MEML-memLfTbl[subcount],
            memLfTbl[subcount], SUBL, CB_NSTAGES);

        /* update memory */

        memcpy(mem, mem+SUBL, (CB_MEML-SUBL)*sizeof(float));
        memcpy(mem+CB_MEML-SUBL,
            &reverseDecresidual[subframe*SUBL],
            SUBL*sizeof(float));
        memset(weightState, 0, LPC_FILTERORDER*sizeof(float));
```

```
            subcount++;

        }

        /* get decoded residual from reversed vector */

        for (i=0; i<SUBL*Nback; i++) {
            decresidual[SUBL*Nback - i - 1] =
```



```
                reverseDecresidual[i];
        }
    }
    /* end encoding part */

    /* adjust index */
    index_conv_enc(cb_index);

    /* pack bytes */

    pbytes=bytes;
    pos=0;

    /* loop over the 3 ULP classes */

    for (ulp=0; ulp<3; ulp++) {

        /* LSF */
        for (k=0; k<LSF_NSPLIT*iLBCenc_inst->lpc_n; k++) {
            packsplit(&lsf_i[k], &firstpart, &lsf_i[k],
                iLBCenc_inst->ULP_inst->lsf_bits[k][ulp],
                iLBCenc_inst->ULP_inst->lsf_bits[k][ulp]+
                iLBCenc_inst->ULP_inst->lsf_bits[k][ulp+1]+
                iLBCenc_inst->ULP_inst->lsf_bits[k][ulp+2]);
            dopack( &pbytes, firstpart,
                iLBCenc_inst->ULP_inst->lsf_bits[k][ulp], &pos);
        }

        /* Start block info */

        packsplit(&start, &firstpart, &start,
            iLBCenc_inst->ULP_inst->start_bits[ulp],
            iLBCenc_inst->ULP_inst->start_bits[ulp]+
            iLBCenc_inst->ULP_inst->start_bits[ulp+1]+
            iLBCenc_inst->ULP_inst->start_bits[ulp+2]);
        dopack( &pbytes, firstpart,
            iLBCenc_inst->ULP_inst->start_bits[ulp], &pos);

        packsplit(&state_first, &firstpart, &state_first,
```

```
            iLBCenc_inst->ULP_inst->startfirst_bits[ulp],
            iLBCenc_inst->ULP_inst->startfirst_bits[ulp]+
            iLBCenc_inst->ULP_inst->startfirst_bits[ulp+1]+
            iLBCenc_inst->ULP_inst->startfirst_bits[ulp+2]);
        dopack( &pbytes, firstpart,
            iLBCenc_inst->ULP_inst->startfirst_bits[ulp], &pos);

        packsplit(&idxForMax, &firstpart, &idxForMax,
            iLBCenc_inst->ULP_inst->scale_bits[ulp],
            iLBCenc_inst->ULP_inst->scale_bits[ulp]+
            iLBCenc_inst->ULP_inst->scale_bits[ulp+1]+
            iLBCenc_inst->ULP_inst->scale_bits[ulp+2]);
        dopack( &pbytes, firstpart,
            iLBCenc_inst->ULP_inst->scale_bits[ulp], &pos);
```



```
        for (k=0; k<iLBCenc_inst->state_short_len; k++) {
            packsplit(idxVec+k, &firstpart, idxVec+k,
                iLBCenc_inst->ULP_inst->state_bits[ulp],
                iLBCenc_inst->ULP_inst->state_bits[ulp]+
                iLBCenc_inst->ULP_inst->state_bits[ulp+1]+
                iLBCenc_inst->ULP_inst->state_bits[ulp+2]);
            dopack( &pbytes, firstpart,
                iLBCenc_inst->ULP_inst->state_bits[ulp], &pos);
        }

        /* 23/22 (20ms/30ms) sample block */

        for (k=0;k<CB_NSTAGES;k++) {
            packsplit(extra_cb_index+k, &firstpart,
                extra_cb_index+k,
                iLBCenc_inst->ULP_inst->extra_cb_index[k][ulp],
                iLBCenc_inst->ULP_inst->extra_cb_index[k][ulp]+
                iLBCenc_inst->ULP_inst->extra_cb_index[k][ulp+1]+
                iLBCenc_inst->ULP_inst->extra_cb_index[k][ulp+2]);
            dopack( &pbytes, firstpart,
                iLBCenc_inst->ULP_inst->extra_cb_index[k][ulp],
                &pos);
        }

        for (k=0;k<CB_NSTAGES;k++) {
            packsplit(extra_gain_index+k, &firstpart,
                extra_gain_index+k,
                iLBCenc_inst->ULP_inst->extra_cb_gain[k][ulp],
                iLBCenc_inst->ULP_inst->extra_cb_gain[k][ulp]+
                iLBCenc_inst->ULP_inst->extra_cb_gain[k][ulp+1]+
                iLBCenc_inst->ULP_inst->extra_cb_gain[k][ulp+2]);
            dopack( &pbytes, firstpart,
```

```
                iLBCenc_inst->ULP_inst->extra_cb_gain[k][ulp],
                &pos);
        }

        /* The two/four (20ms/30ms) 40 sample sub-blocks */

        for (i=0; i<iLBCenc_inst->nasub; i++) {
            for (k=0; k<CB_NSTAGES; k++) {
                packsplit(cb_index+i*CB_NSTAGES+k, &firstpart,
                    cb_index+i*CB_NSTAGES+k,
                    iLBCenc_inst->ULP_inst->cb_index[i][k][ulp],
                    iLBCenc_inst->ULP_inst->cb_index[i][k][ulp]+
                    iLBCenc_inst->ULP_inst->cb_index[i][k][ulp+1]+
                    iLBCenc_inst->ULP_inst->cb_index[i][k][ulp+2]);
                dopack( &pbytes, firstpart,
                    iLBCenc_inst->ULP_inst->cb_index[i][k][ulp],
                    &pos);
            }
        }

        for (i=0; i<iLBCenc_inst->nasub; i++) {
            for (k=0; k<CB_NSTAGES; k++) {
```



```
                packsplit(gain_index+i*CB_NSTAGES+k, &firstpart,
                    gain_index+i*CB_NSTAGES+k,
                    iLBCenc_inst->ULP_inst->cb_gain[i][k][ulp],
                    iLBCenc_inst->ULP_inst->cb_gain[i][k][ulp]+
                    iLBCenc_inst->ULP_inst->cb_gain[i][k][ulp+1]+
                    iLBCenc_inst->ULP_inst->cb_gain[i][k][ulp+2]);
                dopack( &pbytes, firstpart,
                    iLBCenc_inst->ULP_inst->cb_gain[i][k][ulp],
                    &pos);
            }
        }
    }

    /* set the last bit to zero (otherwise the decoder
       will treat it as a lost frame) */
    dopack( &pbytes, 0, 1, &pos);
}
```

A.4 iLBC_decode.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code
```

```
    iLBC_decode.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_ILBCDECODE_H
#define __iLBC_ILBCDECODE_H

#include "iLBC_define.h"

short initDecode(                   /* (o) Number of decoded
                                           samples */
    iLBC_Dec_Inst_t *iLBCdec_inst,  /* (i/o) Decoder instance */
    int mode,                       /* (i) frame size mode */
    int use_enhancer                /* (i) 1 to use enhancer
                                           0 to run without
                                             enhancer */
);

void iLBC_decode(
    float *decblock,            /* (o) decoded signal block */
    unsigned char *bytes,           /* (i) encoded signal bits */
    iLBC_Dec_Inst_t *iLBCdec_inst,  /* (i/o) the decoder state
                                             structure */
    int mode                    /* (i) 0: bad packet, PLC,
                                           1: normal */
);
```



```
#endif
```

A.5 iLBC_decode.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    iLBC_decode.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <math.h>
#include <stdlib.h>
```

```
#include "iLBC_define.h"
#include "StateConstructW.h"
#include "LPCdecode.h"
#include "iCBConstruct.h"
#include "doCPLC.h"
#include "helpfun.h"
#include "constants.h"
#include "packing.h"
#include "string.h"
#include "enhancer.h"
#include "hpOutput.h"
#include "syntFilter.h"

/*----------------------------------------------------------------*
 *  Initiation of decoder instance.
 *---------------------------------------------------------------*/

short initDecode(                   /* (o) Number of decoded
                                           samples */
    iLBC_Dec_Inst_t *iLBCdec_inst,  /* (i/o) Decoder instance */
    int mode,                       /* (i) frame size mode */
    int use_enhancer                /* (i) 1 to use enhancer
                                           0 to run without
                                             enhancer */
){
    int i;

    iLBCdec_inst->mode = mode;

    if (mode==30) {
        iLBCdec_inst->blockl = BLOCKL_30MS;
        iLBCdec_inst->nsub = NSUB_30MS;
        iLBCdec_inst->nasub = NASUB_30MS;
        iLBCdec_inst->lpc_n = LPC_N_30MS;
```



```
        iLBCdec_inst->no_of_bytes = NO_OF_BYTES_30MS;
        iLBCdec_inst->no_of_words = NO_OF_WORDS_30MS;
        iLBCdec_inst->state_short_len=STATE_SHORT_LEN_30MS;
        /* ULP init */
        iLBCdec_inst->ULP_inst=&ULP_30msTbl;
    }
    else if (mode==20) {
        iLBCdec_inst->blockl = BLOCKL_20MS;
        iLBCdec_inst->nsub = NSUB_20MS;
        iLBCdec_inst->nasub = NASUB_20MS;
        iLBCdec_inst->lpc_n = LPC_N_20MS;
        iLBCdec_inst->no_of_bytes = NO_OF_BYTES_20MS;
```

```
        iLBCdec_inst->no_of_words = NO_OF_WORDS_20MS;
        iLBCdec_inst->state_short_len=STATE_SHORT_LEN_20MS;
        /* ULP init */
        iLBCdec_inst->ULP_inst=&ULP_20msTbl;
    }
    else {
        exit(2);
    }

    memset(iLBCdec_inst->syntMem, 0,
        LPC_FILTERORDER*sizeof(float));
    memcpy((*iLBCdec_inst).lsfdeqold, lsfmeanTbl,
        LPC_FILTERORDER*sizeof(float));

    memset(iLBCdec_inst->old_syntdenum, 0,
        ((LPC_FILTERORDER + 1)*NSUB_MAX)*sizeof(float));
    for (i=0; i<NSUB_MAX; i++)
        iLBCdec_inst->old_syntdenum[i*(LPC_FILTERORDER+1)]=1.0;

    iLBCdec_inst->last_lag = 20;

    iLBCdec_inst->prevLag = 120;
    iLBCdec_inst->per = 0.0;
    iLBCdec_inst->consPLICount = 0;
    iLBCdec_inst->prevPLI = 0;
    iLBCdec_inst->prevLpc[0] = 1.0;
    memset(iLBCdec_inst->prevLpc+1,0,
        LPC_FILTERORDER*sizeof(float));
    memset(iLBCdec_inst->prevResidual, 0, BLOCKL_MAX*sizeof(float));
    iLBCdec_inst->seed=777;

    memset(iLBCdec_inst->hpomem, 0, 4*sizeof(float));

    iLBCdec_inst->use_enhancer = use_enhancer;
    memset(iLBCdec_inst->enh_buf, 0, ENH_BUFL*sizeof(float));
    for (i=0;i<ENH_NBLOCKS_TOT;i++)
        iLBCdec_inst->enh_period[i]=(float)40.0;

    iLBCdec_inst->prev_enh_pl = 0;

    return (iLBCdec_inst->blockl);
}
```



```
/*----------------------------------------------------------------*
 *  frame residual decoder function (subrutine to iLBC_decode)
 *---------------------------------------------------------------*/

void Decode(
```

```
    iLBC_Dec_Inst_t *iLBCdec_inst,  /* (i/o) the decoder state
                                             structure */
    float *decresidual,             /* (o) decoded residual frame */
    int start,                      /* (i) location of start
                                           state */
    int idxForMax,                  /* (i) codebook index for the
                                           maximum value */
    int *idxVec,                /* (i) codebook indexes for the
                                           samples  in the start
                                           state */
    float *syntdenum,               /* (i) the decoded synthesis
                                           filter coefficients */
    int *cb_index,                  /* (i) the indexes for the
                                           adaptive codebook */
    int *gain_index,            /* (i) the indexes for the
                                           corresponding gains */
    int *extra_cb_index,        /* (i) the indexes for the
                                           adaptive codebook part
                                           of start state */
    int *extra_gain_index,          /* (i) the indexes for the
                                           corresponding gains */
    int state_first                 /* (i) 1 if non adaptive part
                                           of start state comes
                                           first 0 if that part
                                           comes last */
){
    float reverseDecresidual[BLOCKL_MAX], mem[CB_MEML];
    int k, meml_gotten, Nfor, Nback, i;
    int diff, start_pos;
    int subcount, subframe;

    diff = STATE_LEN - iLBCdec_inst->state_short_len;

    if (state_first == 1) {
        start_pos = (start-1)*SUBL;
    } else {
        start_pos = (start-1)*SUBL + diff;
    }

    /* decode scalar part of start state */

    StateConstructW(idxForMax, idxVec,
        &syntdenum[(start-1)*(LPC_FILTERORDER+1)],
        &decresidual[start_pos], iLBCdec_inst->state_short_len);


    if (state_first) { /* put adaptive part in the end */
```

```
        /* setup memory */

        memset(mem, 0,
            (CB_MEML-iLBCdec_inst->state_short_len)*sizeof(float));
        memcpy(mem+CB_MEML-iLBCdec_inst->state_short_len,
            decresidual+start_pos,
            iLBCdec_inst->state_short_len*sizeof(float));

        /* construct decoded vector */

        iCBConstruct(
            &decresidual[start_pos+iLBCdec_inst->state_short_len],
            extra_cb_index, extra_gain_index, mem+CB_MEML-stMemLTbl,
            stMemLTbl, diff, CB_NSTAGES);

    }
    else {/* put adaptive part in the beginning */

        /* create reversed vectors for prediction */

        for (k=0; k<diff; k++) {
            reverseDecresidual[k] =
                decresidual[(start+1)*SUBL-1-
                        (k+iLBCdec_inst->state_short_len)];
        }

        /* setup memory */

        meml_gotten = iLBCdec_inst->state_short_len;
        for (k=0; k<meml_gotten; k++){
            mem[CB_MEML-1-k] = decresidual[start_pos + k];
        }
        memset(mem, 0, (CB_MEML-k)*sizeof(float));

        /* construct decoded vector */

        iCBConstruct(reverseDecresidual, extra_cb_index,
            extra_gain_index, mem+CB_MEML-stMemLTbl, stMemLTbl,
            diff, CB_NSTAGES);

        /* get decoded residual from reversed vector */

        for (k=0; k<diff; k++) {
            decresidual[start_pos-1-k] = reverseDecresidual[k];
        }
    }

    /* counter for predicted sub-frames */

    subcount=0;
```

```
    /* forward prediction of sub-frames */

    Nfor = iLBCdec_inst->nsub-start-1;
```



```
    if ( Nfor > 0 ){

        /* setup memory */

        memset(mem, 0, (CB_MEML-STATE_LEN)*sizeof(float));
        memcpy(mem+CB_MEML-STATE_LEN, decresidual+(start-1)*SUBL,
            STATE_LEN*sizeof(float));

        /* loop over sub-frames to encode */

        for (subframe=0; subframe<Nfor; subframe++) {

            /* construct decoded vector */

            iCBConstruct(&decresidual[(start+1+subframe)*SUBL],
                cb_index+subcount*CB_NSTAGES,
                gain_index+subcount*CB_NSTAGES,
                mem+CB_MEML-memLfTbl[subcount],
                memLfTbl[subcount], SUBL, CB_NSTAGES);

            /* update memory */

            memcpy(mem, mem+SUBL, (CB_MEML-SUBL)*sizeof(float));
            memcpy(mem+CB_MEML-SUBL,
                &decresidual[(start+1+subframe)*SUBL],
                SUBL*sizeof(float));

            subcount++;

        }

    }

    /* backward prediction of sub-frames */

    Nback = start-1;

    if ( Nback > 0 ) {

        /* setup memory */

        meml_gotten = SUBL*(iLBCdec_inst->nsub+1-start);

        if ( meml_gotten > CB_MEML ) {
            meml_gotten=CB_MEML;
```

```
        }
        for (k=0; k<meml_gotten; k++) {
            mem[CB_MEML-1-k] = decresidual[(start-1)*SUBL + k];
        }
        memset(mem, 0, (CB_MEML-k)*sizeof(float));

        /* loop over subframes to decode */
```



```
        for (subframe=0; subframe<Nback; subframe++) {

            /* construct decoded vector */

            iCBConstruct(&reverseDecresidual[subframe*SUBL],
                cb_index+subcount*CB_NSTAGES,
                gain_index+subcount*CB_NSTAGES,
                mem+CB_MEML-memLfTbl[subcount], memLfTbl[subcount],
                SUBL, CB_NSTAGES);

            /* update memory */

            memcpy(mem, mem+SUBL, (CB_MEML-SUBL)*sizeof(float));
            memcpy(mem+CB_MEML-SUBL,
                &reverseDecresidual[subframe*SUBL],
                SUBL*sizeof(float));

            subcount++;
        }

        /* get decoded residual from reversed vector */

        for (i=0; i<SUBL*Nback; i++)
            decresidual[SUBL*Nback - i - 1] =
            reverseDecresidual[i];
    }
}

/*----------------------------------------------------------------*
 *  main decoder function
 *---------------------------------------------------------------*/

void iLBC_decode(
    float *decblock,            /* (o) decoded signal block */
    unsigned char *bytes,           /* (i) encoded signal bits */
    iLBC_Dec_Inst_t *iLBCdec_inst,  /* (i/o) the decoder state
                                             structure */
    int mode                    /* (i) 0: bad packet, PLC,
                                           1: normal */
){
```

```
    float data[BLOCKL_MAX];
    float lsfdeq[LPC_FILTERORDER*LPC_N_MAX];
    float PLCresidual[BLOCKL_MAX], PLClpc[LPC_FILTERORDER + 1];
    float zeros[BLOCKL_MAX], one[LPC_FILTERORDER + 1];
    int k, i, start, idxForMax, pos, lastpart, ulp;
    int lag, ilag;
    float cc, maxcc;
    int idxVec[STATE_LEN];
    int check;
    int gain_index[NASUB_MAX*CB_NSTAGES],
        extra_gain_index[CB_NSTAGES];
    int cb_index[CB_NSTAGES*NASUB_MAX], extra_cb_index[CB_NSTAGES];
    int lsf_i[LSF_NSPLIT*LPC_N_MAX];
    int state_first;
```



```
    int last_bit;
    unsigned char *pbytes;
    float weightdenum[(LPC_FILTERORDER + 1)*NSUB_MAX];
    int order_plus_one;
    float syntdenum[NSUB_MAX*(LPC_FILTERORDER+1)];
    float decresidual[BLOCKL_MAX];

    if (mode>0) { /* the data are good */

        /* decode data */

        pbytes=bytes;
        pos=0;

        /* Set everything to zero before decoding */

        for (k=0; k<LSF_NSPLIT*LPC_N_MAX; k++) {
            lsf_i[k]=0;
        }
        start=0;
        state_first=0;
        idxForMax=0;
        for (k=0; k<iLBCdec_inst->state_short_len; k++) {
            idxVec[k]=0;
        }
        for (k=0; k<CB_NSTAGES; k++) {
            extra_cb_index[k]=0;
        }
        for (k=0; k<CB_NSTAGES; k++) {
            extra_gain_index[k]=0;
        }
        for (i=0; i<iLBCdec_inst->nasub; i++) {
            for (k=0; k<CB_NSTAGES; k++) {
```

```
                cb_index[i*CB_NSTAGES+k]=0;
            }
        }
        for (i=0; i<iLBCdec_inst->nasub; i++) {
            for (k=0; k<CB_NSTAGES; k++) {
                gain_index[i*CB_NSTAGES+k]=0;
            }
        }

        /* loop over ULP classes */

        for (ulp=0; ulp<3; ulp++) {

            /* LSF */
            for (k=0; k<LSF_NSPLIT*iLBCdec_inst->lpc_n; k++){
                unpack( &pbytes, &lastpart,
                    iLBCdec_inst->ULP_inst->lsf_bits[k][ulp], &pos);
                packcombine(&lsf_i[k], lastpart,
                    iLBCdec_inst->ULP_inst->lsf_bits[k][ulp]);
            }
```



```
            /* Start block info */

            unpack( &pbytes, &lastpart,
                iLBCdec_inst->ULP_inst->start_bits[ulp], &pos);
            packcombine(&start, lastpart,
                iLBCdec_inst->ULP_inst->start_bits[ulp]);

            unpack( &pbytes, &lastpart,
                iLBCdec_inst->ULP_inst->startfirst_bits[ulp], &pos);
            packcombine(&state_first, lastpart,
                iLBCdec_inst->ULP_inst->startfirst_bits[ulp]);

            unpack( &pbytes, &lastpart,
                iLBCdec_inst->ULP_inst->scale_bits[ulp], &pos);
            packcombine(&idxForMax, lastpart,
                iLBCdec_inst->ULP_inst->scale_bits[ulp]);

            for (k=0; k<iLBCdec_inst->state_short_len; k++) {
                unpack( &pbytes, &lastpart,
                    iLBCdec_inst->ULP_inst->state_bits[ulp], &pos);
                packcombine(idxVec+k, lastpart,
                    iLBCdec_inst->ULP_inst->state_bits[ulp]);
            }

            /* 23/22 (20ms/30ms) sample block */

            for (k=0; k<CB_NSTAGES; k++) {
```

```
                unpack( &pbytes, &lastpart,
                    iLBCdec_inst->ULP_inst->extra_cb_index[k][ulp],
                    &pos);
                packcombine(extra_cb_index+k, lastpart,
                    iLBCdec_inst->ULP_inst->extra_cb_index[k][ulp]);
            }
            for (k=0; k<CB_NSTAGES; k++) {
                unpack( &pbytes, &lastpart,
                    iLBCdec_inst->ULP_inst->extra_cb_gain[k][ulp],
                    &pos);
                packcombine(extra_gain_index+k, lastpart,
                    iLBCdec_inst->ULP_inst->extra_cb_gain[k][ulp]);
            }

            /* The two/four (20ms/30ms) 40 sample sub-blocks */

            for (i=0; i<iLBCdec_inst->nasub; i++) {
                for (k=0; k<CB_NSTAGES; k++) {
                    unpack( &pbytes, &lastpart,
                    iLBCdec_inst->ULP_inst->cb_index[i][k][ulp],
                        &pos);
                    packcombine(cb_index+i*CB_NSTAGES+k, lastpart,
                    iLBCdec_inst->ULP_inst->cb_index[i][k][ulp]);
                }
            }

            for (i=0; i<iLBCdec_inst->nasub; i++) {
```



```
                for (k=0; k<CB_NSTAGES; k++) {
                    unpack( &pbytes, &lastpart,

                    iLBCdec_inst->ULP_inst->cb_gain[i][k][ulp],
                        &pos);
                    packcombine(gain_index+i*CB_NSTAGES+k, lastpart,
                        iLBCdec_inst->ULP_inst->cb_gain[i][k][ulp]);
                }
            }
        }
        /* Extract last bit. If it is 1 this indicates an
           empty/lost frame */
        unpack( &pbytes, &last_bit, 1, &pos);

        /* Check for bit errors or empty/lost frames */
        if (start<1)
            mode = 0;
        if (iLBCdec_inst->mode==20 && start>3)
            mode = 0;
        if (iLBCdec_inst->mode==30 && start>5)
```

```
            mode = 0;
        if (last_bit==1)
            mode = 0;

        if (mode==1) { /* No bit errors was detected,
                          continue decoding */

            /* adjust index */
            index_conv_dec(cb_index);

            /* decode the lsf */

            SimplelsfDEQ(lsfdeq, lsf_i, iLBCdec_inst->lpc_n);
            check=LSF_check(lsfdeq, LPC_FILTERORDER,
                iLBCdec_inst->lpc_n);
            DecoderInterpolateLSF(syntdenum, weightdenum,
                lsfdeq, LPC_FILTERORDER, iLBCdec_inst);

            Decode(iLBCdec_inst, decresidual, start, idxForMax,
                idxVec, syntdenum, cb_index, gain_index,
                extra_cb_index, extra_gain_index,
                state_first);

            /* preparing the plc for a future loss! */

            doThePLC(PLCresidual, PLClpc, 0, decresidual,
                syntdenum +
                (LPC_FILTERORDER + 1)*(iLBCdec_inst->nsub - 1),
                (*iLBCdec_inst).last_lag, iLBCdec_inst);


            memcpy(decresidual, PLCresidual,
                iLBCdec_inst->blockl*sizeof(float));
        }
```



```
    }

    if (mode == 0) {
        /* the data is bad (either a PLC call
         * was made or a severe bit error was detected)
         */

        /* packet loss conceal */

        memset(zeros, 0, BLOCKL_MAX*sizeof(float));

        one[0] = 1;
        memset(one+1, 0, LPC_FILTERORDER*sizeof(float));
```

```
            start=0;

            doThePLC(PLCresidual, PLClpc, 1, zeros, one,
                (*iLBCdec_inst).last_lag, iLBCdec_inst);
            memcpy(decresidual, PLCresidual,
                iLBCdec_inst->blockl*sizeof(float));

            order_plus_one = LPC_FILTERORDER + 1;
            for (i = 0; i < iLBCdec_inst->nsub; i++) {
                memcpy(syntdenum+(i*order_plus_one), PLClpc,
                    order_plus_one*sizeof(float));
            }
        }

        if (iLBCdec_inst->use_enhancer == 1) {

            /* post filtering */

            iLBCdec_inst->last_lag =
                enhancerInterface(data, decresidual, iLBCdec_inst);

            /* synthesis filtering */

            if (iLBCdec_inst->mode==20) {
                /* Enhancer has 40 samples delay */
                i=0;
                syntFilter(data + i*SUBL,
                    iLBCdec_inst->old_syntdenum +
                    (i+iLBCdec_inst->nsub-1)*(LPC_FILTERORDER+1),
                    SUBL, iLBCdec_inst->syntMem);
                for (i=1; i < iLBCdec_inst->nsub; i++) {
                    syntFilter(data + i*SUBL,
                        syntdenum + (i-1)*(LPC_FILTERORDER+1),
                        SUBL, iLBCdec_inst->syntMem);
                }
            } else if (iLBCdec_inst->mode==30) {
                /* Enhancer has 80 samples delay */
                for (i=0; i < 2; i++) {
                    syntFilter(data + i*SUBL,
```



```
                        iLBCdec_inst->old_syntdenum +
                        (i+iLBCdec_inst->nsub-2)*(LPC_FILTERORDER+1),
                        SUBL, iLBCdec_inst->syntMem);
                }
                for (i=2; i < iLBCdec_inst->nsub; i++) {
                    syntFilter(data + i*SUBL,
                        syntdenum + (i-2)*(LPC_FILTERORDER+1), SUBL,
                        iLBCdec_inst->syntMem);
```

```
            }
        }

    } else {

        /* Find last lag */
        lag = 20;
        maxcc = xCorrCoef(&decresidual[BLOCKL_MAX-ENH_BLOCKL],
            &decresidual[BLOCKL_MAX-ENH_BLOCKL-lag], ENH_BLOCKL);

        for (ilag=21; ilag<120; ilag++) {
            cc = xCorrCoef(&decresidual[BLOCKL_MAX-ENH_BLOCKL],
                &decresidual[BLOCKL_MAX-ENH_BLOCKL-ilag],
                ENH_BLOCKL);

            if (cc > maxcc) {
                maxcc = cc;
                lag = ilag;
            }
        }
        iLBCdec_inst->last_lag = lag;

        /* copy data and run synthesis filter */

        memcpy(data, decresidual,
            iLBCdec_inst->blockl*sizeof(float));
        for (i=0; i < iLBCdec_inst->nsub; i++) {
            syntFilter(data + i*SUBL,
                syntdenum + i*(LPC_FILTERORDER+1), SUBL,
                iLBCdec_inst->syntMem);
        }
    }

    /* high pass filtering on output if desired, otherwise
       copy to out */

    hpOutput(data, iLBCdec_inst->blockl,
                decblock,iLBCdec_inst->hpomem);

    /* memcpy(decblock,data,iLBCdec_inst->blockl*sizeof(float));*/

    memcpy(iLBCdec_inst->old_syntdenum, syntdenum,

        iLBCdec_inst->nsub*(LPC_FILTERORDER+1)*sizeof(float));

    iLBCdec_inst->prev_enh_pl=0;
```



```
    if (mode==0) { /* PLC was used */
```

```
        iLBCdec_inst->prev_enh_pl=1;
    }
}
```

A.6 iLBC_define.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    iLBC_define.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/
#include <string.h>

#ifndef __iLBC_ILBCDEFINE_H
#define __iLBC_ILBCDEFINE_H

/* general codec settings */

#define FS                          (float)8000.0
#define BLOCKL_20MS                 160
#define BLOCKL_30MS                 240
#define BLOCKL_MAX                  240
#define NSUB_20MS                   4
#define NSUB_30MS                   6
#define NSUB_MAX                6
#define NASUB_20MS                  2
#define NASUB_30MS                  4
#define NASUB_MAX                   4
#define SUBL                    40
#define STATE_LEN                   80
#define STATE_SHORT_LEN_30MS        58
#define STATE_SHORT_LEN_20MS        57

/* LPC settings */

#define LPC_FILTERORDER             10
#define LPC_CHIRP_SYNTDENUM         (float)0.9025
#define LPC_CHIRP_WEIGHTDENUM       (float)0.4222
#define LPC_LOOKBACK            60
#define LPC_N_20MS                  1
#define LPC_N_30MS                  2
#define LPC_N_MAX                   2
#define LPC_ASYMDIFF            20
#define LPC_BW                      (float)60.0
#define LPC_WN                      (float)1.0001
#define LSF_NSPLIT                  3
```

```
#define LSF_NUMBER_OF_STEPS     4
#define LPC_HALFORDER           (LPC_FILTERORDER/2)

/* cb settings */

#define CB_NSTAGES              3
#define CB_EXPAND               2
#define CB_MEML                 147
#define CB_FILTERLEN        2*4
#define CB_HALFFILTERLEN    4
#define CB_RESRANGE             34
#define CB_MAXGAIN              (float)1.3

/* enhancer */

#define ENH_BLOCKL              80  /* block length */
#define ENH_BLOCKL_HALF         (ENH_BLOCKL/2)
#define ENH_HL                  3   /* 2*ENH_HL+1 is number blocks
                                       in said second sequence */
#define ENH_SLOP            2   /* max difference estimated and
                                       correct pitch period */
#define ENH_PLOCSL              20  /* pitch-estimates and pitch-
                                       locations buffer length */
#define ENH_OVERHANG        2
#define ENH_UPS0            4   /* upsampling rate */
#define ENH_FL0                 3   /* 2*FLO+1 is the length of
                                       each filter */
#define ENH_VECTL               (ENH_BLOCKL+2*ENH_FL0)
#define ENH_CORRDIM             (2*ENH_SLOP+1)
#define ENH_NBLOCKS             (BLOCKL_MAX/ENH_BLOCKL)
#define ENH_NBLOCKS_EXTRA       5
#define ENH_NBLOCKS_TOT         8   /* ENH_NBLOCKS +
                                       ENH_NBLOCKS_EXTRA */
#define ENH_BUFL            (ENH_NBLOCKS_TOT)*ENH_BLOCKL
#define ENH_ALPHA0              (float)0.05

/* Down sampling */

#define FILTERORDER_DS          7
#define DELAY_DS            3
#define FACTOR_DS               2

/* bit stream defs */

#define NO_OF_BYTES_20MS    38
#define NO_OF_BYTES_30MS    50
#define NO_OF_WORDS_20MS    19
#define NO_OF_WORDS_30MS    25
```

```
#define STATE_BITS              3
#define BYTE_LEN            8
#define ULP_CLASSES             3

/* help parameters */
```



```
#define FLOAT_MAX               (float)1.0e37
#define EPS                     (float)2.220446049250313e-016
#define PI                      (float)3.14159265358979323846
#define MIN_SAMPLE              -32768
#define MAX_SAMPLE              32767
#define TWO_PI                  (float)6.283185307
#define PI2                     (float)0.159154943

/* type definition encoder instance */
typedef struct iLBC_ULP_Inst_t_ {
    int lsf_bits[6][ULP_CLASSES+2];
    int start_bits[ULP_CLASSES+2];
    int startfirst_bits[ULP_CLASSES+2];
    int scale_bits[ULP_CLASSES+2];
    int state_bits[ULP_CLASSES+2];
    int extra_cb_index[CB_NSTAGES][ULP_CLASSES+2];
    int extra_cb_gain[CB_NSTAGES][ULP_CLASSES+2];
    int cb_index[NSUB_MAX][CB_NSTAGES][ULP_CLASSES+2];
    int cb_gain[NSUB_MAX][CB_NSTAGES][ULP_CLASSES+2];
} iLBC_ULP_Inst_t;

/* type definition encoder instance */
typedef struct iLBC_Enc_Inst_t_ {

    /* flag for frame size mode */
    int mode;

    /* basic parameters for different frame sizes */
    int blockl;
    int nsub;
    int nasub;
    int no_of_bytes, no_of_words;
    int lpc_n;
    int state_short_len;
    const iLBC_ULP_Inst_t *ULP_inst;

    /* analysis filter state */
    float anaMem[LPC_FILTERORDER];

    /* old lsf parameters for interpolation */
    float lsfold[LPC_FILTERORDER];
    float lsfdeqold[LPC_FILTERORDER];
```

```
    /* signal buffer for LP analysis */
    float lpc_buffer[LPC_LOOKBACK + BLOCKL_MAX];

    /* state of input HP filter */
    float hpimem[4];

} iLBC_Enc_Inst_t;

/* type definition decoder instance */
typedef struct iLBC_Dec_Inst_t_ {
```



```
    /* flag for frame size mode */
    int mode;

    /* basic parameters for different frame sizes */
    int blockl;
    int nsub;
    int nasub;
    int no_of_bytes, no_of_words;
    int lpc_n;
    int state_short_len;
    const iLBC_ULP_Inst_t *ULP_inst;

    /* synthesis filter state */
    float syntMem[LPC_FILTERORDER];

    /* old LSF for interpolation */
    float lsfdeqold[LPC_FILTERORDER];

    /* pitch lag estimated in enhancer and used in PLC */
    int last_lag;

    /* PLC state information */
    int prevLag, consPLICount, prevPLI, prev_enh_pl;
    float prevLpc[LPC_FILTERORDER+1];
    float prevResidual[NSUB_MAX*SUBL];
    float per;
    unsigned long seed;

    /* previous synthesis filter parameters */
    float old_syntdenum[(LPC_FILTERORDER + 1)*NSUB_MAX];

    /* state of output HP filter */
    float hpomem[4];

    /* enhancer state information */
    int use_enhancer;
```

```
    float enh_buf[ENH_BUFL];
    float enh_period[ENH_NBLOCKS_TOT];

} iLBC_Dec_Inst_t;

#endif
```

A.7 constants.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    constants.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.
```



```
******************************************************************/

#ifndef __iLBC_CONSTANTS_H
#define __iLBC_CONSTANTS_H

#include "iLBC_define.h"


/* ULP bit allocation */

extern const iLBC_ULP_Inst_t ULP_20msTbl;
extern const iLBC_ULP_Inst_t ULP_30msTbl;

/* high pass filters */

extern float hpi_zero_coefsTbl[];
extern float hpi_pole_coefsTbl[];
extern float hpo_zero_coefsTbl[];
extern float hpo_pole_coefsTbl[];

/* low pass filters */
extern float lpFilt_coefsTbl[];

/* LPC analysis and quantization */

extern float lpc_winTbl[];
extern float lpc_asymwinTbl[];
extern float lpc_lagwinTbl[];
extern float lsfCbTbl[];
extern float lsfmeanTbl[];
```

```
extern int   dim_lsfCbTbl[];
extern int   size_lsfCbTbl[];
extern float lsf_weightTbl_30ms[];
extern float lsf_weightTbl_20ms[];

/* state quantization tables */

extern float state_sq3Tbl[];
extern float state_frgqTbl[];

/* gain quantization tables */

extern float gain_sq3Tbl[];
extern float gain_sq4Tbl[];
extern float gain_sq5Tbl[];

/* adaptive codebook definitions */

extern int search_rangeTbl[5][CB_NSTAGES];
extern int memLfTbl[];
extern int stMemLTbl;
extern float cbfiltersTbl[CB_FILTERLEN];
```



```
/* enhancer definitions */

extern float polyphaserTbl[];
extern float enh_plocsTbl[];

#endif
```

A.8 constants.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    constants.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include "iLBC_define.h"

/* ULP bit allocation */

    /* 20 ms frame */
```

```
const iLBC_ULP_Inst_t ULP_20msTbl = {
    /* LSF */
    {   {6,0,0,0,0}, {7,0,0,0,0}, {7,0,0,0,0},
        {0,0,0,0,0}, {0,0,0,0,0}, {0,0,0,0,0}},
    /* Start state location, gain and samples */
    {2,0,0,0,0},
    {1,0,0,0,0},
    {6,0,0,0,0},
    {0,1,2,0,0},
    /* extra CB index and extra CB gain */
    {{6,0,1,0,0}, {0,0,7,0,0}, {0,0,7,0,0}},
    {{2,0,3,0,0}, {1,1,2,0,0}, {0,0,3,0,0}},
    /* CB index and CB gain */
    {   {{7,0,1,0,0}, {0,0,7,0,0}, {0,0,7,0,0}},
        {{0,0,8,0,0}, {0,0,8,0,0}, {0,0,8,0,0}},
        {{0,0,0,0,0}, {0,0,0,0,0}, {0,0,0,0,0}},
        {{0,0,0,0,0}, {0,0,0,0,0}, {0,0,0,0,0}}},
    {   {{1,2,2,0,0}, {1,1,2,0,0}, {0,0,3,0,0}},
        {{1,1,3,0,0}, {0,2,2,0,0}, {0,0,3,0,0}},
        {{0,0,0,0,0}, {0,0,0,0,0}, {0,0,0,0,0}},
        {{0,0,0,0,0}, {0,0,0,0,0}, {0,0,0,0,0}}}
};

    /* 30 ms frame */

const iLBC_ULP_Inst_t ULP_30msTbl = {
    /* LSF */
    {   {6,0,0,0,0}, {7,0,0,0,0}, {7,0,0,0,0},
```



```
        {6,0,0,0,0}, {7,0,0,0,0}, {7,0,0,0,0}},
    /* Start state location, gain and samples */
    {3,0,0,0,0},
    {1,0,0,0,0},
    {6,0,0,0,0},
    {0,1,2,0,0},
    /* extra CB index and extra CB gain */
    {{4,2,1,0,0}, {0,0,7,0,0}, {0,0,7,0,0}},
    {{1,1,3,0,0}, {1,1,2,0,0}, {0,0,3,0,0}},
    /* CB index and CB gain */
    {   {{6,1,1,0,0}, {0,0,7,0,0}, {0,0,7,0,0}},
        {{0,7,1,0,0}, {0,0,8,0,0}, {0,0,8,0,0}},
        {{0,7,1,0,0}, {0,0,8,0,0}, {0,0,8,0,0}},
        {{0,7,1,0,0}, {0,0,8,0,0}, {0,0,8,0,0}}},
    {   {{1,2,2,0,0}, {1,2,1,0,0}, {0,0,3,0,0}},
        {{0,2,3,0,0}, {0,2,2,0,0}, {0,0,3,0,0}},
        {{0,1,4,0,0}, {0,1,3,0,0}, {0,0,3,0,0}},
        {{0,1,4,0,0}, {0,1,3,0,0}, {0,0,3,0,0}}}
};
```

```
/* HP Filters */

float hpi_zero_coefsTbl[3] = {
    (float)0.92727436, (float)-1.8544941, (float)0.92727436
};
float hpi_pole_coefsTbl[3] = {
    (float)1.0, (float)-1.9059465, (float)0.9114024
};
float hpo_zero_coefsTbl[3] = {
    (float)0.93980581, (float)-1.8795834, (float)0.93980581
};
float hpo_pole_coefsTbl[3] = {
    (float)1.0, (float)-1.9330735, (float)0.93589199
};

/* LP Filter */

float lpFilt_coefsTbl[FILTERORDER_DS]={
    (float)-0.066650, (float)0.125000, (float)0.316650,
    (float)0.414063, (float)0.316650,
    (float)0.125000, (float)-0.066650
};

/* State quantization tables */

float state_sq3Tbl[8] = {
    (float)-3.719849, (float)-2.177490, (float)-1.130005,
    (float)-0.309692, (float)0.444214, (float)1.329712,
    (float)2.436279, (float)3.983887
};

float state_frgqTbl[64] = {
    (float)1.000085, (float)1.071695, (float)1.140395,
    (float)1.206868, (float)1.277188, (float)1.351503,
```



```
    (float)1.429380, (float)1.500727, (float)1.569049,
    (float)1.639599, (float)1.707071, (float)1.781531,
    (float)1.840799, (float)1.901550, (float)1.956695,
    (float)2.006750, (float)2.055474, (float)2.102787,
    (float)2.142819, (float)2.183592, (float)2.217962,
    (float)2.257177, (float)2.295739, (float)2.332967,
    (float)2.369248, (float)2.402792, (float)2.435080,
    (float)2.468598, (float)2.503394, (float)2.539284,
    (float)2.572944, (float)2.605036, (float)2.636331,
    (float)2.668939, (float)2.698780, (float)2.729101,
    (float)2.759786, (float)2.789834, (float)2.818679,
    (float)2.848074, (float)2.877470, (float)2.906899,
```

```
    (float)2.936655, (float)2.967804, (float)3.000115,
    (float)3.033367, (float)3.066355, (float)3.104231,
    (float)3.141499, (float)3.183012, (float)3.222952,
    (float)3.265433, (float)3.308441, (float)3.350823,
    (float)3.395275, (float)3.442793, (float)3.490801,
    (float)3.542514, (float)3.604064, (float)3.666050,
    (float)3.740994, (float)3.830749, (float)3.938770,
    (float)4.101764
};

/* CB tables */

int search_rangeTbl[5][CB_NSTAGES]={{58,58,58}, {108,44,44},
            {108,108,108}, {108,108,108}, {108,108,108}};
int stMemLTbl=85;
int memLfTbl[NASUB_MAX]={147,147,147,147};

/* expansion filter(s) */

float cbfiltersTbl[CB_FILTERLEN]={
    (float)-0.034180, (float)0.108887, (float)-0.184326,
    (float)0.806152,  (float)0.713379, (float)-0.144043,
    (float)0.083740,  (float)-0.033691
};

/* Gain Quantization */

float gain_sq3Tbl[8]={
    (float)-1.000000,  (float)-0.659973,  (float)-0.330017,
    (float)0.000000, (float)0.250000, (float)0.500000,
    (float)0.750000, (float)1.00000};

float gain_sq4Tbl[16]={
    (float)-1.049988, (float)-0.900024, (float)-0.750000,
    (float)-0.599976, (float)-0.450012, (float)-0.299988,
    (float)-0.150024, (float)0.000000, (float)0.150024,
    (float)0.299988, (float)0.450012, (float)0.599976,
    (float)0.750000, (float)0.900024, (float)1.049988,
    (float)1.200012};

float gain_sq5Tbl[32]={
    (float)0.037476, (float)0.075012, (float)0.112488,
```



```
    (float)0.150024, (float)0.187500, (float)0.224976,
    (float)0.262512, (float)0.299988, (float)0.337524,
    (float)0.375000, (float)0.412476, (float)0.450012,
    (float)0.487488, (float)0.525024, (float)0.562500,
    (float)0.599976, (float)0.637512, (float)0.674988,
    (float)0.712524, (float)0.750000, (float)0.787476,
```

```
    (float)0.825012, (float)0.862488, (float)0.900024,
    (float)0.937500, (float)0.974976, (float)1.012512,
    (float)1.049988, (float)1.087524, (float)1.125000,
    (float)1.162476, (float)1.200012};

/* Enhancer - Upsamling a factor 4 (ENH_UPS0 = 4) */
float polyphaserTbl[ENH_UPS0*(2*ENH_FL0+1)]={
    (float)0.000000, (float)0.000000, (float)0.000000,
(float)1.000000,
        (float)0.000000, (float)0.000000, (float)0.000000,
    (float)0.015625, (float)-0.076904, (float)0.288330,
(float)0.862061,
        (float)-0.106445, (float)0.018799, (float)-0.015625,
    (float)0.023682, (float)-0.124268, (float)0.601563,
(float)0.601563,
        (float)-0.124268, (float)0.023682, (float)-0.023682,
    (float)0.018799, (float)-0.106445, (float)0.862061,
(float)0.288330,
        (float)-0.076904, (float)0.015625, (float)-0.018799};

float enh_plocsTbl[ENH_NBLOCKS_TOT] = {(float)40.0, (float)120.0,
            (float)200.0, (float)280.0, (float)360.0,
            (float)440.0, (float)520.0, (float)600.0};

/* LPC analysis and quantization */

int dim_lsfCbTbl[LSF_NSPLIT] = {3, 3, 4};
int size_lsfCbTbl[LSF_NSPLIT] = {64,128,128};


float lsfmeanTbl[LPC_FILTERORDER] = {
    (float)0.281738, (float)0.445801, (float)0.663330,
    (float)0.962524, (float)1.251831, (float)1.533081,
    (float)1.850586, (float)2.137817, (float)2.481445,
    (float)2.777344};

float lsf_weightTbl_30ms[6] = {(float)(1.0/2.0), (float)1.0,
(float)(2.0/3.0),
    (float)(1.0/3.0), (float)0.0, (float)0.0};

float lsf_weightTbl_20ms[4] = {(float)(3.0/4.0), (float)(2.0/4.0),
    (float)(1.0/4.0), (float)(0.0)};

/* Hanning LPC window */
float lpc_winTbl[BLOCKL_MAX]={
    (float)0.000183, (float)0.000671, (float)0.001526,
    (float)0.002716, (float)0.004242, (float)0.006104,
    (float)0.008301, (float)0.010834, (float)0.013702,
```

```
(float)0.016907, (float)0.020416, (float)0.024261,
(float)0.028442, (float)0.032928, (float)0.037750,
(float)0.042877, (float)0.048309, (float)0.054047,
(float)0.060089, (float)0.066437, (float)0.073090,
(float)0.080017, (float)0.087219, (float)0.094727,
(float)0.102509, (float)0.110535, (float)0.118835,
(float)0.127411, (float)0.136230, (float)0.145294,
(float)0.154602, (float)0.164154, (float)0.173920,
(float)0.183899, (float)0.194122, (float)0.204529,
(float)0.215149, (float)0.225952, (float)0.236938,
(float)0.248108, (float)0.259460, (float)0.270966,
(float)0.282654, (float)0.294464, (float)0.306396,
(float)0.318481, (float)0.330688, (float)0.343018,
(float)0.355438, (float)0.367981, (float)0.380585,
(float)0.393280, (float)0.406067, (float)0.418884,
(float)0.431763, (float)0.444702, (float)0.457672,
(float)0.470673, (float)0.483704, (float)0.496735,
(float)0.509766, (float)0.522797, (float)0.535828,
(float)0.548798, (float)0.561768, (float)0.574677,
(float)0.587524, (float)0.600342, (float)0.613068,
(float)0.625732, (float)0.638306, (float)0.650787,
(float)0.663147, (float)0.675415, (float)0.687561,
(float)0.699585, (float)0.711487, (float)0.723206,
(float)0.734802, (float)0.746216, (float)0.757477,
(float)0.768585, (float)0.779480, (float)0.790192,
(float)0.800720, (float)0.811005, (float)0.821106,
(float)0.830994, (float)0.840668, (float)0.850067,
(float)0.859253, (float)0.868225, (float)0.876892,
(float)0.885345, (float)0.893524, (float)0.901428,
(float)0.909058, (float)0.916412, (float)0.923492,
(float)0.930267, (float)0.936768, (float)0.942963,
(float)0.948853, (float)0.954437, (float)0.959717,
(float)0.964691, (float)0.969360, (float)0.973694,
(float)0.977692, (float)0.981384, (float)0.984741,
(float)0.987762, (float)0.990479, (float)0.992828,
(float)0.994873, (float)0.996552, (float)0.997925,
(float)0.998932, (float)0.999603, (float)0.999969,
(float)0.999969, (float)0.999603, (float)0.998932,
(float)0.997925, (float)0.996552, (float)0.994873,
(float)0.992828, (float)0.990479, (float)0.987762,
(float)0.984741, (float)0.981384, (float)0.977692,
(float)0.973694, (float)0.969360, (float)0.964691,
(float)0.959717, (float)0.954437, (float)0.948853,
(float)0.942963, (float)0.936768, (float)0.930267,
(float)0.923492, (float)0.916412, (float)0.909058,
(float)0.901428, (float)0.893524, (float)0.885345,
(float)0.876892, (float)0.868225, (float)0.859253,
(float)0.850067, (float)0.840668, (float)0.830994,
(float)0.821106, (float)0.811005, (float)0.800720,
(float)0.790192, (float)0.779480, (float)0.768585,
(float)0.757477, (float)0.746216, (float)0.734802,
```

```
    (float)0.723206, (float)0.711487, (float)0.699585,
    (float)0.687561, (float)0.675415, (float)0.663147,
    (float)0.650787, (float)0.638306, (float)0.625732,
```



```
    (float)0.613068, (float)0.600342, (float)0.587524,
    (float)0.574677, (float)0.561768, (float)0.548798,
    (float)0.535828, (float)0.522797, (float)0.509766,
    (float)0.496735, (float)0.483704, (float)0.470673,
    (float)0.457672, (float)0.444702, (float)0.431763,
    (float)0.418884, (float)0.406067, (float)0.393280,
    (float)0.380585, (float)0.367981, (float)0.355438,
    (float)0.343018, (float)0.330688, (float)0.318481,
    (float)0.306396, (float)0.294464, (float)0.282654,
    (float)0.270966, (float)0.259460, (float)0.248108,
    (float)0.236938, (float)0.225952, (float)0.215149,
    (float)0.204529, (float)0.194122, (float)0.183899,
    (float)0.173920, (float)0.164154, (float)0.154602,
    (float)0.145294, (float)0.136230, (float)0.127411,
    (float)0.118835, (float)0.110535, (float)0.102509,
    (float)0.094727, (float)0.087219, (float)0.080017,
    (float)0.073090, (float)0.066437, (float)0.060089,
    (float)0.054047, (float)0.048309, (float)0.042877,
    (float)0.037750, (float)0.032928, (float)0.028442,
    (float)0.024261, (float)0.020416, (float)0.016907,
    (float)0.013702, (float)0.010834, (float)0.008301,
    (float)0.006104, (float)0.004242, (float)0.002716,
    (float)0.001526, (float)0.000671, (float)0.000183
};

/* Asymmetric LPC window */
float lpc_asymwinTbl[BLOCKL_MAX]={
    (float)0.000061, (float)0.000214, (float)0.000458,
    (float)0.000824, (float)0.001282, (float)0.001831,
    (float)0.002472, (float)0.003235, (float)0.004120,
    (float)0.005066, (float)0.006134, (float)0.007294,
    (float)0.008545, (float)0.009918, (float)0.011383,
    (float)0.012939, (float)0.014587, (float)0.016357,
    (float)0.018219, (float)0.020172, (float)0.022217,
    (float)0.024353, (float)0.026611, (float)0.028961,
    (float)0.031372, (float)0.033905, (float)0.036530,
    (float)0.039276, (float)0.042084, (float)0.044983,
    (float)0.047974, (float)0.051086, (float)0.054260,
    (float)0.057526, (float)0.060883, (float)0.064331,
    (float)0.067871, (float)0.071503, (float)0.075226,
    (float)0.079010, (float)0.082916, (float)0.086884,
    (float)0.090942, (float)0.095062, (float)0.099304,
    (float)0.103607, (float)0.107971, (float)0.112427,
    (float)0.116974, (float)0.121582, (float)0.126282,
```

```
     (float)0.131073, (float)0.135895, (float)0.140839,
     (float)0.145813, (float)0.150879, (float)0.156006,
     (float)0.161224, (float)0.166504, (float)0.171844,
     (float)0.177246, (float)0.182709, (float)0.188263,
     (float)0.193848, (float)0.199524, (float)0.205231,
     (float)0.211029, (float)0.216858, (float)0.222778,
     (float)0.228729, (float)0.234741, (float)0.240814,
     (float)0.246918, (float)0.253082, (float)0.259308,
     (float)0.265564, (float)0.271881, (float)0.278259,
     (float)0.284668, (float)0.291107, (float)0.297607,
```



```
     (float)0.304138, (float)0.310730, (float)0.317322,
     (float)0.323975, (float)0.330658, (float)0.337372,
     (float)0.344147, (float)0.350922, (float)0.357727,
     (float)0.364594, (float)0.371460, (float)0.378357,
     (float)0.385284, (float)0.392212, (float)0.399170,
     (float)0.406158, (float)0.413177, (float)0.420197,
     (float)0.427246, (float)0.434296, (float)0.441376,
     (float)0.448456, (float)0.455536, (float)0.462646,
     (float)0.469757, (float)0.476868, (float)0.483978,
     (float)0.491089, (float)0.498230, (float)0.505341,
     (float)0.512451, (float)0.519592, (float)0.526703,
     (float)0.533813, (float)0.540924, (float)0.548004,
     (float)0.555084, (float)0.562164, (float)0.569244,
     (float)0.576294, (float)0.583313, (float)0.590332,
     (float)0.597321, (float)0.604309, (float)0.611267,
     (float)0.618195, (float)0.625092, (float)0.631989,
     (float)0.638855, (float)0.645660, (float)0.652466,
     (float)0.659241, (float)0.665985, (float)0.672668,
     (float)0.679352, (float)0.685974, (float)0.692566,
     (float)0.699127, (float)0.705658, (float)0.712128,
     (float)0.718536, (float)0.724945, (float)0.731262,
     (float)0.737549, (float)0.743805, (float)0.750000,
     (float)0.756134, (float)0.762238, (float)0.768280,
     (float)0.774261, (float)0.780182, (float)0.786072,
     (float)0.791870, (float)0.797638, (float)0.803314,
     (float)0.808960, (float)0.814514, (float)0.820038,
     (float)0.825470, (float)0.830841, (float)0.836151,
     (float)0.841400, (float)0.846558, (float)0.851654,
     (float)0.856689, (float)0.861633, (float)0.866516,
     (float)0.871338, (float)0.876068, (float)0.880737,
     (float)0.885315, (float)0.889801, (float)0.894226,
     (float)0.898560, (float)0.902832, (float)0.907013,
     (float)0.911102, (float)0.915100, (float)0.919037,
     (float)0.922882, (float)0.926636, (float)0.930328,
     (float)0.933899, (float)0.937408, (float)0.940796,
     (float)0.944122, (float)0.947357, (float)0.950470,
     (float)0.953522, (float)0.956482, (float)0.959351,
```

```
    (float)0.962097, (float)0.964783, (float)0.967377,
    (float)0.969849, (float)0.972229, (float)0.974518,
    (float)0.976715, (float)0.978821, (float)0.980835,
    (float)0.982727, (float)0.984528, (float)0.986237,
    (float)0.987854, (float)0.989380, (float)0.990784,
    (float)0.992096, (float)0.993317, (float)0.994415,
    (float)0.995422, (float)0.996338, (float)0.997162,
    (float)0.997864, (float)0.998474, (float)0.998962,
    (float)0.999390, (float)0.999695, (float)0.999878,
    (float)0.999969, (float)0.999969, (float)0.996918,
    (float)0.987701, (float)0.972382, (float)0.951050,
    (float)0.923889, (float)0.891022, (float)0.852631,
    (float)0.809021, (float)0.760406, (float)0.707092,
    (float)0.649445, (float)0.587799, (float)0.522491,
    (float)0.453979, (float)0.382690, (float)0.309021,
    (float)0.233459, (float)0.156433, (float)0.078461
};
```



```
/* Lag window for LPC */
float lpc_lagwinTbl[LPC_FILTERORDER + 1]={
    (float)1.000100, (float)0.998890, (float)0.995569,
        (float)0.990057, (float)0.982392,
    (float)0.972623, (float)0.960816, (float)0.947047,
        (float)0.931405, (float)0.913989, (float)0.894909};

/* LSF quantization*/
float lsfCbTbl[64 * 3 + 128 * 3 + 128 * 4] = {
(float)0.155396, (float)0.273193, (float)0.451172,
(float)0.390503, (float)0.648071, (float)1.002075,
(float)0.440186, (float)0.692261, (float)0.955688,
(float)0.343628, (float)0.642334, (float)1.071533,
(float)0.318359, (float)0.491577, (float)0.670532,
(float)0.193115, (float)0.375488, (float)0.725708,
(float)0.364136, (float)0.510376, (float)0.658691,
(float)0.297485, (float)0.527588, (float)0.842529,
(float)0.227173, (float)0.365967, (float)0.563110,
(float)0.244995, (float)0.396729, (float)0.636475,
(float)0.169434, (float)0.300171, (float)0.520264,
(float)0.312866, (float)0.464478, (float)0.643188,
(float)0.248535, (float)0.429932, (float)0.626099,
(float)0.236206, (float)0.491333, (float)0.817139,
(float)0.334961, (float)0.625122, (float)0.895752,
(float)0.343018, (float)0.518555, (float)0.698608,
(float)0.372803, (float)0.659790, (float)0.945435,
(float)0.176880, (float)0.316528, (float)0.581421,
(float)0.416382, (float)0.625977, (float)0.805176,
(float)0.303223, (float)0.568726, (float)0.915039,
```

```
(float)0.203613, (float)0.351440, (float)0.588135,
(float)0.221191, (float)0.375000, (float)0.614746,
(float)0.199951, (float)0.323364, (float)0.476074,
(float)0.300781, (float)0.433350, (float)0.566895,
(float)0.226196, (float)0.354004, (float)0.507568,
(float)0.300049, (float)0.508179, (float)0.711670,
(float)0.312012, (float)0.492676, (float)0.763428,
(float)0.329956, (float)0.541016, (float)0.795776,
(float)0.373779, (float)0.604614, (float)0.928833,
(float)0.210571, (float)0.452026, (float)0.755249,
(float)0.271118, (float)0.473267, (float)0.662476,
(float)0.285522, (float)0.436890, (float)0.634399,
(float)0.246704, (float)0.565552, (float)0.859009,
(float)0.270508, (float)0.406250, (float)0.553589,
(float)0.361450, (float)0.578491, (float)0.813843,
(float)0.342651, (float)0.482788, (float)0.622437,
(float)0.340332, (float)0.549438, (float)0.743164,
(float)0.200439, (float)0.336304, (float)0.540894,
(float)0.407837, (float)0.644775, (float)0.895142,
(float)0.294678, (float)0.454834, (float)0.699097,
(float)0.193115, (float)0.344482, (float)0.643188,
(float)0.275757, (float)0.420776, (float)0.598755,
(float)0.380493, (float)0.608643, (float)0.861084,
(float)0.222778, (float)0.426147, (float)0.676514,
```



```
(float)0.407471, (float)0.700195, (float)1.053101,
(float)0.218384, (float)0.377197, (float)0.669922,
(float)0.313232, (float)0.454102, (float)0.600952,
(float)0.347412, (float)0.571533, (float)0.874146,
(float)0.238037, (float)0.405396, (float)0.729492,
(float)0.223877, (float)0.412964, (float)0.822021,
(float)0.395264, (float)0.582153, (float)0.743896,
(float)0.247925, (float)0.485596, (float)0.720581,
(float)0.229126, (float)0.496582, (float)0.907715,
(float)0.260132, (float)0.566895, (float)1.012695,
(float)0.337402, (float)0.611572, (float)0.978149,
(float)0.267822, (float)0.447632, (float)0.769287,
(float)0.250610, (float)0.381714, (float)0.530029,
(float)0.430054, (float)0.805054, (float)1.221924,
(float)0.382568, (float)0.544067, (float)0.701660,
(float)0.383545, (float)0.710327, (float)1.149170,
(float)0.271362, (float)0.529053, (float)0.775513,
(float)0.246826, (float)0.393555, (float)0.588623,
(float)0.266846, (float)0.422119, (float)0.676758,
(float)0.311523, (float)0.580688, (float)0.838623,
(float)1.331177, (float)1.576782, (float)1.779541,
(float)1.160034, (float)1.401978, (float)1.768188,
(float)1.161865, (float)1.525146, (float)1.715332,
```

```
(float)0.759521, (float)0.913940, (float)1.119873,
(float)0.947144, (float)1.121338, (float)1.282471,
(float)1.015015, (float)1.557007, (float)1.804932,
(float)1.172974, (float)1.402100, (float)1.692627,
(float)1.087524, (float)1.474243, (float)1.665405,
(float)0.899536, (float)1.105225, (float)1.406250,
(float)1.148438, (float)1.484741, (float)1.796265,
(float)0.785645, (float)1.209839, (float)1.567749,
(float)0.867798, (float)1.166504, (float)1.450684,
(float)0.922485, (float)1.229858, (float)1.420898,
(float)0.791260, (float)1.123291, (float)1.409546,
(float)0.788940, (float)0.966064, (float)1.340332,
(float)1.051147, (float)1.272827, (float)1.556641,
(float)0.866821, (float)1.181152, (float)1.538818,
(float)0.906738, (float)1.373535, (float)1.607910,
(float)1.244751, (float)1.581421, (float)1.933838,
(float)0.913940, (float)1.337280, (float)1.539673,
(float)0.680542, (float)0.959229, (float)1.662720,
(float)0.887207, (float)1.430542, (float)1.800781,
(float)0.912598, (float)1.433594, (float)1.683960,
(float)0.860474, (float)1.060303, (float)1.455322,
(float)1.005127, (float)1.381104, (float)1.706909,
(float)0.800781, (float)1.363892, (float)1.829102,
(float)0.781860, (float)1.124390, (float)1.505981,
(float)1.003662, (float)1.471436, (float)1.684692,
(float)0.981323, (float)1.309570, (float)1.618042,
(float)1.228760, (float)1.554321, (float)1.756470,
(float)0.734375, (float)0.895752, (float)1.225586,
(float)0.841797, (float)1.055664, (float)1.249268,
(float)0.920166, (float)1.119385, (float)1.486206,
(float)0.894409, (float)1.539063, (float)1.828979,
```



```
(float)1.283691, (float)1.543335, (float)1.858276,
(float)0.676025, (float)0.933105, (float)1.490845,
(float)0.821289, (float)1.491821, (float)1.739868,
(float)0.923218, (float)1.144653, (float)1.580566,
(float)1.057251, (float)1.345581, (float)1.635864,
(float)0.888672, (float)1.074951, (float)1.353149,
(float)0.942749, (float)1.195435, (float)1.505493,
(float)1.492310, (float)1.788086, (float)2.039673,
(float)1.070313, (float)1.634399, (float)1.860962,
(float)1.253296, (float)1.488892, (float)1.686035,
(float)0.647095, (float)0.864014, (float)1.401855,
(float)0.866699, (float)1.254883, (float)1.453369,
(float)1.063965, (float)1.532593, (float)1.731323,
(float)1.167847, (float)1.521484, (float)1.884033,
(float)0.956055, (float)1.502075, (float)1.745605,
(float)0.928711, (float)1.288574, (float)1.479614,
```

```
(float)1.088013, (float)1.380737, (float)1.570801,
(float)0.905029, (float)1.186768, (float)1.371948,
(float)1.057861, (float)1.421021, (float)1.617432,
(float)1.108276, (float)1.312500, (float)1.501465,
(float)0.979492, (float)1.416992, (float)1.624268,
(float)1.276001, (float)1.661011, (float)2.007935,
(float)0.993042, (float)1.168579, (float)1.331665,
(float)0.778198, (float)0.944946, (float)1.235962,
(float)1.223755, (float)1.491333, (float)1.815674,
(float)0.852661, (float)1.350464, (float)1.722290,
(float)1.134766, (float)1.593140, (float)1.787354,
(float)1.051392, (float)1.339722, (float)1.531006,
(float)0.803589, (float)1.271240, (float)1.652100,
(float)0.755737, (float)1.143555, (float)1.639404,
(float)0.700928, (float)0.837280, (float)1.130371,
(float)0.942749, (float)1.197876, (float)1.669800,
(float)0.993286, (float)1.378296, (float)1.566528,
(float)0.801025, (float)1.095337, (float)1.298950,
(float)0.739990, (float)1.032959, (float)1.383667,
(float)0.845703, (float)1.072266, (float)1.543823,
(float)0.915649, (float)1.072266, (float)1.224487,
(float)1.021973, (float)1.226196, (float)1.481323,
(float)0.999878, (float)1.204102, (float)1.555908,
(float)0.722290, (float)0.913940, (float)1.340210,
(float)0.673340, (float)0.835938, (float)1.259521,
(float)0.832397, (float)1.208374, (float)1.394165,
(float)0.962158, (float)1.576172, (float)1.912842,
(float)1.166748, (float)1.370850, (float)1.556763,
(float)0.946289, (float)1.138550, (float)1.400391,
(float)1.035034, (float)1.218262, (float)1.386475,
(float)1.393799, (float)1.717773, (float)2.000244,
(float)0.972656, (float)1.260986, (float)1.760620,
(float)1.028198, (float)1.288452, (float)1.484619,
(float)0.773560, (float)1.258057, (float)1.756714,
(float)1.080322, (float)1.328003, (float)1.742676,
(float)0.823975, (float)1.450806, (float)1.917725,
(float)0.859009, (float)1.016602, (float)1.191895,
(float)0.843994, (float)1.131104, (float)1.645020,
```



```
(float)1.189697, (float)1.702759, (float)1.894409,
(float)1.346680, (float)1.763184, (float)2.066040,
(float)0.980469, (float)1.253784, (float)1.441650,
(float)1.338135, (float)1.641968, (float)1.932739,
(float)1.223267, (float)1.424194, (float)1.626465,
(float)0.765747, (float)1.004150, (float)1.579102,
(float)1.042847, (float)1.269165, (float)1.647461,
(float)0.968750, (float)1.257568, (float)1.555786,
(float)0.826294, (float)0.993408, (float)1.275146,
```

```
(float)0.742310, (float)0.950439, (float)1.430542,
(float)1.054321, (float)1.439819, (float)1.828003,
(float)1.072998, (float)1.261719, (float)1.441895,
(float)0.859375, (float)1.036377, (float)1.314819,
(float)0.895752, (float)1.267212, (float)1.605591,
(float)0.805420, (float)0.962891, (float)1.142334,
(float)0.795654, (float)1.005493, (float)1.468506,
(float)1.105347, (float)1.313843, (float)1.584839,
(float)0.792236, (float)1.221802, (float)1.465698,
(float)1.170532, (float)1.467651, (float)1.664063,
(float)0.838257, (float)1.153198, (float)1.342163,
(float)0.968018, (float)1.198242, (float)1.391235,
(float)1.250122, (float)1.623535, (float)1.823608,
(float)0.711670, (float)1.058350, (float)1.512085,
(float)1.204834, (float)1.454468, (float)1.739136,
(float)1.137451, (float)1.421753, (float)1.620117,
(float)0.820435, (float)1.322754, (float)1.578247,
(float)0.798706, (float)1.005005, (float)1.213867,
(float)0.980713, (float)1.324951, (float)1.512939,
(float)1.112305, (float)1.438843, (float)1.735596,
(float)1.135498, (float)1.356689, (float)1.635742,
(float)1.101318, (float)1.387451, (float)1.686523,
(float)0.849854, (float)1.276978, (float)1.523438,
(float)1.377930, (float)1.627563, (float)1.858154,
(float)0.884888, (float)1.095459, (float)1.287476,
(float)1.289795, (float)1.505859, (float)1.756592,
(float)0.817505, (float)1.384155, (float)1.650513,
(float)1.446655, (float)1.702148, (float)1.931885,
(float)0.835815, (float)1.023071, (float)1.385376,
(float)0.916626, (float)1.139038, (float)1.335327,
(float)0.980103, (float)1.174072, (float)1.453735,
(float)1.705688, (float)2.153809, (float)2.398315, (float)2.743408,
(float)1.797119, (float)2.016846, (float)2.445679, (float)2.701904,
(float)1.990356, (float)2.219116, (float)2.576416, (float)2.813477,
(float)1.849365, (float)2.190918, (float)2.611572, (float)2.835083,
(float)1.657959, (float)1.854370, (float)2.159058, (float)2.726196,
(float)1.437744, (float)1.897705, (float)2.253174, (float)2.655396,
(float)2.028687, (float)2.247314, (float)2.542358, (float)2.875854,
(float)1.736938, (float)1.922119, (float)2.185913, (float)2.743408,
(float)1.521606, (float)1.870972, (float)2.526855, (float)2.786987,
(float)1.841431, (float)2.050659, (float)2.463623, (float)2.857666,
(float)1.590088, (float)2.067261, (float)2.427979, (float)2.794434,
(float)1.746826, (float)2.057373, (float)2.320190, (float)2.800781,
(float)1.734619, (float)1.940552, (float)2.306030, (float)2.826416,
(float)1.786255, (float)2.204468, (float)2.457520, (float)2.795288,
```



```
(float)1.861084, (float)2.170532, (float)2.414551, (float)2.763672,
(float)2.001465, (float)2.307617, (float)2.552734, (float)2.811890,
```

```
(float)1.784424, (float)2.124146, (float)2.381592, (float)2.645508,
(float)1.888794, (float)2.135864, (float)2.418579, (float)2.861206,
(float)2.301147, (float)2.531250, (float)2.724976, (float)2.913086,
(float)1.837769, (float)2.051270, (float)2.261963, (float)2.553223,
(float)2.012939, (float)2.221191, (float)2.440186, (float)2.678101,
(float)1.429565, (float)1.858276, (float)2.582275, (float)2.845703,
(float)1.622803, (float)1.897705, (float)2.367310, (float)2.621094,
(float)1.581543, (float)1.960449, (float)2.515869, (float)2.736450,
(float)1.419434, (float)1.933960, (float)2.394653, (float)2.746704,
(float)1.721924, (float)2.059570, (float)2.421753, (float)2.769653,
(float)1.911011, (float)2.220703, (float)2.461060, (float)2.740723,
(float)1.581177, (float)1.860840, (float)2.516968, (float)2.874634,
(float)1.870361, (float)2.098755, (float)2.432373, (float)2.656494,
(float)2.059692, (float)2.279785, (float)2.495605, (float)2.729370,
(float)1.815674, (float)2.181519, (float)2.451538, (float)2.680542,
(float)1.407959, (float)1.768311, (float)2.343018, (float)2.668091,
(float)2.168701, (float)2.394653, (float)2.604736, (float)2.829346,
(float)1.636230, (float)1.865723, (float)2.329102, (float)2.824219,
(float)1.878906, (float)2.139526, (float)2.376709, (float)2.679810,
(float)1.765381, (float)1.971802, (float)2.195435, (float)2.586914,
(float)2.164795, (float)2.410889, (float)2.673706, (float)2.903198,
(float)2.071899, (float)2.331055, (float)2.645874, (float)2.907104,
(float)2.026001, (float)2.311523, (float)2.594849, (float)2.863892,
(float)1.948975, (float)2.180786, (float)2.514893, (float)2.797852,
(float)1.881836, (float)2.130859, (float)2.478149, (float)2.804199,
(float)2.238159, (float)2.452759, (float)2.652832, (float)2.868286,
(float)1.897949, (float)2.101685, (float)2.524292, (float)2.880127,
(float)1.856445, (float)2.074585, (float)2.541016, (float)2.791748,
(float)1.695557, (float)2.199097, (float)2.506226, (float)2.742676,
(float)1.612671, (float)1.877075, (float)2.435425, (float)2.732910,
(float)1.568848, (float)1.786499, (float)2.194580, (float)2.768555,
(float)1.953369, (float)2.164551, (float)2.486938, (float)2.874023,
(float)1.388306, (float)1.725342, (float)2.384521, (float)2.771851,
(float)2.115356, (float)2.337769, (float)2.592896, (float)2.864014,
(float)1.905762, (float)2.111328, (float)2.363525, (float)2.789307,
(float)1.882568, (float)2.332031, (float)2.598267, (float)2.827637,
(float)1.683594, (float)2.088745, (float)2.361938, (float)2.608643,
(float)1.874023, (float)2.182129, (float)2.536133, (float)2.766968,
(float)1.861938, (float)2.070435, (float)2.309692, (float)2.700562,
(float)1.722168, (float)2.107422, (float)2.477295, (float)2.837646,
(float)1.926880, (float)2.184692, (float)2.442627, (float)2.663818,
(float)2.123901, (float)2.337280, (float)2.553101, (float)2.777466,
(float)1.588135, (float)1.911499, (float)2.212769, (float)2.543945,
(float)2.053955, (float)2.370850, (float)2.712158, (float)2.939941,
(float)2.210449, (float)2.519653, (float)2.770386, (float)2.958618,
(float)2.199463, (float)2.474731, (float)2.718262, (float)2.919922,
(float)1.960083, (float)2.175415, (float)2.608032, (float)2.888794,
(float)1.953735, (float)2.185181, (float)2.428223, (float)2.809570,
(float)1.615234, (float)2.036499, (float)2.576538, (float)2.834595,
(float)1.621094, (float)2.028198, (float)2.431030, (float)2.664673,
(float)1.824951, (float)2.267456, (float)2.514526, (float)2.747925,
```

(float)1.994263, (float)2.229126, (float)2.475220, (float)2.833984,


```
(float)1.746338, (float)2.011353, (float)2.588257, (float)2.826904,
(float)1.562866, (float)2.135986, (float)2.471680, (float)2.687256,
(float)1.748901, (float)2.083496, (float)2.460938, (float)2.686279,
(float)1.758057, (float)2.131470, (float)2.636597, (float)2.891602,
(float)2.071289, (float)2.299072, (float)2.550781, (float)2.814331,
(float)1.839600, (float)2.094360, (float)2.496460, (float)2.723999,
(float)1.882202, (float)2.088257, (float)2.636841, (float)2.923096,
(float)1.957886, (float)2.153198, (float)2.384399, (float)2.615234,
(float)1.992920, (float)2.351196, (float)2.654419, (float)2.889771,
(float)2.012817, (float)2.262451, (float)2.643799, (float)2.903076,
(float)2.025635, (float)2.254761, (float)2.508423, (float)2.784058,
(float)2.316040, (float)2.589355, (float)2.794189, (float)2.963623,
(float)1.741211, (float)2.279541, (float)2.578491, (float)2.816284,
(float)1.845337, (float)2.055786, (float)2.348511, (float)2.822021,
(float)1.679932, (float)1.926514, (float)2.499756, (float)2.835693,
(float)1.722534, (float)1.946899, (float)2.448486, (float)2.728760,
(float)1.829834, (float)2.043213, (float)2.580444, (float)2.867676,
(float)1.676636, (float)2.071655, (float)2.322510, (float)2.704834,
(float)1.791504, (float)2.113525, (float)2.469727, (float)2.784058,
(float)1.977051, (float)2.215088, (float)2.497437, (float)2.726929,
(float)1.800171, (float)2.106689, (float)2.357788, (float)2.738892,
(float)1.827759, (float)2.170166, (float)2.525879, (float)2.852417,
(float)1.918335, (float)2.132813, (float)2.488403, (float)2.728149,
(float)1.916748, (float)2.225098, (float)2.542603, (float)2.857666,
(float)1.761230, (float)1.976074, (float)2.507446, (float)2.884521,
(float)2.053711, (float)2.367432, (float)2.608032, (float)2.837646,
(float)1.595337, (float)2.000977, (float)2.307129, (float)2.578247,
(float)1.470581, (float)2.031250, (float)2.375854, (float)2.647583,
(float)1.801392, (float)2.128052, (float)2.399780, (float)2.822876,
(float)1.853638, (float)2.066650, (float)2.429199, (float)2.751465,
(float)1.956299, (float)2.163696, (float)2.394775, (float)2.734253,
(float)1.963623, (float)2.275757, (float)2.585327, (float)2.865234,
(float)1.887451, (float)2.105469, (float)2.331787, (float)2.587402,
(float)2.120117, (float)2.443359, (float)2.733887, (float)2.941406,
(float)1.506348, (float)1.766968, (float)2.400513, (float)2.851807,
(float)1.664551, (float)1.981079, (float)2.375732, (float)2.774414,
(float)1.720703, (float)1.978882, (float)2.391479, (float)2.640991,
(float)1.483398, (float)1.814819, (float)2.434448, (float)2.722290,
(float)1.769043, (float)2.136597, (float)2.563721, (float)2.774414,
(float)1.810791, (float)2.049316, (float)2.373901, (float)2.613647,
(float)1.788330, (float)2.005981, (float)2.359131, (float)2.723145,
(float)1.785156, (float)1.993164, (float)2.399780, (float)2.832520,
(float)1.695313, (float)2.022949, (float)2.522583, (float)2.745117,
(float)1.584106, (float)1.965576, (float)2.299927, (float)2.715576,
(float)1.894897, (float)2.249878, (float)2.655884, (float)2.897705,
(float)1.720581, (float)1.995728, (float)2.299438, (float)2.557007,
```

```
(float)1.619385, (float)2.173950, (float)2.574219, (float)2.787964,
(float)1.883179, (float)2.220459, (float)2.474365, (float)2.825073,
(float)1.447632, (float)2.045044, (float)2.555542, (float)2.744873,
(float)1.502686, (float)2.156616, (float)2.653320, (float)2.846558,
(float)1.711548, (float)1.944092, (float)2.282959, (float)2.685791,
(float)1.499756, (float)1.867554, (float)2.341064, (float)2.578857,
(float)1.916870, (float)2.135132, (float)2.568237, (float)2.826050,
(float)1.498047, (float)1.711182, (float)2.223267, (float)2.755127,
```



```
(float)1.808716, (float)1.997559, (float)2.256470, (float)2.758545,
(float)2.088501, (float)2.402710, (float)2.667358, (float)2.890259,
(float)1.545044, (float)1.819214, (float)2.324097, (float)2.692993,
(float)1.796021, (float)2.012573, (float)2.505737, (float)2.784912,
(float)1.786499, (float)2.041748, (float)2.290405, (float)2.650757,
(float)1.938232, (float)2.264404, (float)2.529053, (float)2.796143
};
```

A.9 anaFilter.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    anaFilter.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_ANAFILTER_H
#define __iLBC_ANAFILTER_H

void anaFilter(
    float *In,  /* (i) Signal to be filtered */
    float *a,   /* (i) LP parameters */
    int len,/* (i) Length of signal */
    float *Out, /* (o) Filtered signal */
    float *mem  /* (i/o) Filter state */
);

#endif
```

A.10 anaFilter.c

```
/******************************************************************
```

```
    iLBC Speech Coder ANSI-C Source Code

    anaFilter.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <string.h>
#include "iLBC_define.h"

/*----------------------------------------------------------------*
 *  LP analysis filter.
```



```
 *---------------------------------------------------------------*/

void anaFilter(
    float *In,  /* (i) Signal to be filtered */
    float *a,   /* (i) LP parameters */
    int len,/* (i) Length of signal */
    float *Out, /* (o) Filtered signal */
    float *mem  /* (i/o) Filter state */
){
    int i, j;
    float *po, *pi, *pm, *pa;

    po = Out;

    /* Filter first part using memory from past */

    for (i=0; i<LPC_FILTERORDER; i++) {
        pi = &In[i];
        pm = &mem[LPC_FILTERORDER-1];
        pa = a;
        *po=0.0;
        for (j=0; j<=i; j++) {
            *po+=(*pa++)*(*pi--);
        }
        for (j=i+1; j<LPC_FILTERORDER+1; j++) {

            *po+=(*pa++)*(*pm--);
        }
        po++;
    }

    /* Filter last part where the state is entierly
       in the input vector */
```

```
    for (i=LPC_FILTERORDER; i<len; i++) {
        pi = &In[i];
        pa = a;
        *po=0.0;
        for (j=0; j<LPC_FILTERORDER+1; j++) {
            *po+=(*pa++)*(*pi--);
        }
        po++;
    }

    /* Update state vector */

    memcpy(mem, &In[len-LPC_FILTERORDER],
        LPC_FILTERORDER*sizeof(float));
}
```

A.11 createCB.h

```
/******************************************************************
```



```
    iLBC Speech Coder ANSI-C Source Code

    createCB.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_CREATECB_H
#define __iLBC_CREATECB_H

void filteredCBvecs(
    float *cbvectors,   /* (o) Codebook vector for the
                               higher section */
    float *mem,         /* (i) Buffer to create codebook
                               vectors from */
    int lMem        /* (i) Length of buffer */
);

void searchAugmentedCB(
    int low,        /* (i) Start index for the search */
    int high,           /* (i) End index for the search */
    int stage,          /* (i) Current stage */
    int startIndex,     /* (i) CB index for the first
                               augmented vector */
    float *target,      /* (i) Target vector for encoding */
```

```
    float *buffer,      /* (i) Pointer to the end of the
                               buffer for augmented codebook
                               construction */
    float *max_measure, /* (i/o) Currently maximum measure */
    int *best_index,/* (o) Currently the best index */
    float *gain,    /* (o) Currently the best gain */
    float *energy,      /* (o) Energy of augmented
                               codebook vectors */
    float *invenergy/* (o) Inv energy of aug codebook
                               vectors */
);

void createAugmentedVec(
    int index,          /* (i) Index for the aug vector
                               to be created */
    float *buffer,      /* (i) Pointer to the end of the
                               buffer for augmented codebook
                               construction */
    float *cbVec    /* (o) The construced codebook vector */
);

#endif
```

A.12 **createCB.c**


```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    createCB.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include "iLBC_define.h"
#include "constants.h"
#include <string.h>
#include <math.h>

/*----------------------------------------------------------------*
 *  Construct an additional codebook vector by filtering the
 *  initial codebook buffer. This vector is then used to expand
 *  the codebook with an additional section.
 *---------------------------------------------------------------*/
```

```
void filteredCBvecs(
    float *cbvectors,   /* (o) Codebook vectors for the
                               higher section */
    float *mem,         /* (i) Buffer to create codebook
                               vector from */
    int lMem        /* (i) Length of buffer */
){
    int j, k;
    float *pp, *pp1;
    float tempbuff2[CB_MEML+CB_FILTERLEN];
    float *pos;

    memset(tempbuff2, 0, (CB_HALFFILTERLEN-1)*sizeof(float));
    memcpy(&tempbuff2[CB_HALFFILTERLEN-1], mem, lMem*sizeof(float));
    memset(&tempbuff2[lMem+CB_HALFFILTERLEN-1], 0,
        (CB_HALFFILTERLEN+1)*sizeof(float));

    /* Create codebook vector for higher section by filtering */

    /* do filtering */
    pos=cbvectors;
    memset(pos, 0, lMem*sizeof(float));
    for (k=0; k<lMem; k++) {
        pp=&tempbuff2[k];
        pp1=&cbfiltersTbl[CB_FILTERLEN-1];
        for (j=0;j<CB_FILTERLEN;j++) {
            (*pos)+=(*pp++)*(*pp1--);
        }
        pos++;
    }
}
```



```
/*----------------------------------------------------------------*
 *  Search the augmented part of the codebook to find the best
 *  measure.
 *----------------------------------------------------------------*/

void searchAugmentedCB(
    int low,        /* (i) Start index for the search */
    int high,           /* (i) End index for the search */
    int stage,          /* (i) Current stage */
    int startIndex,     /* (i) Codebook index for the first
                               aug vector */
    float *target,      /* (i) Target vector for encoding */
    float *buffer,      /* (i) Pointer to the end of the buffer for
                               augmented codebook construction */
    float *max_measure, /* (i/o) Currently maximum measure */
```

```
    int *best_index,/* (o) Currently the best index */
    float *gain,    /* (o) Currently the best gain */
    float *energy,      /* (o) Energy of augmented codebook
                               vectors */
    float *invenergy/* (o) Inv energy of augmented codebook
                               vectors */
) {
    int icount, ilow, j, tmpIndex;
    float *pp, *ppo, *ppi, *ppe, crossDot, alfa;
    float weighted, measure, nrjRecursive;
    float ftmp;

    /* Compute the energy for the first (low-5)
       noninterpolated samples */
    nrjRecursive = (float) 0.0;
    pp = buffer - low + 1;
    for (j=0; j<(low-5); j++) {
        nrjRecursive += ( (*pp)*(*pp) );
        pp++;
    }
    ppe = buffer - low;


    for (icount=low; icount<=high; icount++) {

        /* Index of the codebook vector used for retrieving
           energy values */
        tmpIndex = startIndex+icount-20;

        ilow = icount-4;

        /* Update the energy recursively to save complexity */
        nrjRecursive = nrjRecursive + (*ppe)*(*ppe);
        ppe--;
        energy[tmpIndex] = nrjRecursive;

        /* Compute cross dot product for the first (low-5)
           samples */
        crossDot = (float) 0.0;
```



```
        pp = buffer-icount;
        for (j=0; j<ilow; j++) {
            crossDot += target[j]*(*pp++);
        }

        /* interpolation */
        alfa = (float) 0.2;
        ppo = buffer-4;
        ppi = buffer-icount-4;
```

```
        for (j=ilow; j<icount; j++) {
            weighted = ((float)1.0-alfa)*(*ppo)+alfa*(*ppi);
            ppo++;
            ppi++;
            energy[tmpIndex] += weighted*weighted;
            crossDot += target[j]*weighted;
            alfa += (float)0.2;
        }

        /* Compute energy and cross dot product for the
           remaining samples */
        pp = buffer - icount;
        for (j=icount; j<SUBL; j++) {
            energy[tmpIndex] += (*pp)*(*pp);
            crossDot += target[j]*(*pp++);
        }

        if (energy[tmpIndex]>0.0) {
            invenergy[tmpIndex]=(float)1.0/(energy[tmpIndex]+EPS);
        } else {
            invenergy[tmpIndex] = (float) 0.0;
        }

        if (stage==0) {
            measure = (float)-10000000.0;

            if (crossDot > 0.0) {
                measure = crossDot*crossDot*invenergy[tmpIndex];
            }
        }
        else {
            measure = crossDot*crossDot*invenergy[tmpIndex];
        }

        /* check if measure is better */
        ftmp = crossDot*invenergy[tmpIndex];

        if ((measure>*max_measure) && (fabs(ftmp)<CB_MAXGAIN)) {
            *best_index = tmpIndex;
            *max_measure = measure;
            *gain = ftmp;
        }
    }
}
```



```
/*----------------------------------------------------------------*
 *  Recreate a specific codebook vector from the augmented part.
```

```
 *
 *---------------------------------------------------------------*/

void createAugmentedVec(
    int index,      /* (i) Index for the augmented vector
                           to be created */
    float *buffer,  /* (i) Pointer to the end of the buffer for
                           augmented codebook construction */
    float *cbVec/* (o) The construced codebook vector */
) {
    int ilow, j;
    float *pp, *ppo, *ppi, alfa, alfa1, weighted;

    ilow = index-5;

    /* copy the first noninterpolated part */

    pp = buffer-index;
    memcpy(cbVec,pp,sizeof(float)*index);

    /* interpolation */

    alfa1 = (float)0.2;
    alfa = 0.0;
    ppo = buffer-5;
    ppi = buffer-index-5;
    for (j=ilow; j<index; j++) {
        weighted = ((float)1.0-alfa)*(*ppo)+alfa*(*ppi);
        ppo++;
        ppi++;
        cbVec[j] = weighted;
        alfa += alfa1;
    }

    /* copy the second noninterpolated part */

    pp = buffer - index;
    memcpy(cbVec+index,pp,sizeof(float)*(SUBL-index));
}
```

A.13 doCPLC.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    doCPLC.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.
```

```
******************************************************************/

#ifndef __iLBC_DOLPC_H
#define __iLBC_DOLPC_H

void doThePLC(
    float *PLCresidual, /* (o) concealed residual */
    float *PLClpc,      /* (o) concealed LP parameters */
    int PLI,        /* (i) packet loss indicator
                               0 - no PL, 1 = PL */
    float *decresidual, /* (i) decoded residual */
    float *lpc,         /* (i) decoded LPC (only used for no PL) */
    int inlag,          /* (i) pitch lag */
    iLBC_Dec_Inst_t *iLBCdec_inst
                        /* (i/o) decoder instance */
);

#endif
```

A.14 doCPLC.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    doCPLC.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <math.h>
#include <string.h>
#include <stdio.h>

#include "iLBC_define.h"

/*----------------------------------------------------------------*
 *  Compute cross correlation and pitch gain for pitch prediction
 *  of last subframe at given lag.
 *---------------------------------------------------------------*/

void compCorr(
    float *cc,      /* (o) cross correlation coefficient */
    float *gc,      /* (o) gain */
    float *pm,
```

```
    float *buffer,  /* (i) signal buffer */
    int lag,    /* (i) pitch lag */
    int bLen,       /* (i) length of buffer */
    int sRange      /* (i) correlation search length */
){
```



```
    int i;
    float ftmp1, ftmp2, ftmp3;

    /* Guard against getting outside buffer */
    if ((bLen-sRange-lag)<0) {
        sRange=bLen-lag;
    }

    ftmp1 = 0.0;
    ftmp2 = 0.0;
    ftmp3 = 0.0;
    for (i=0; i<sRange; i++) {
        ftmp1 += buffer[bLen-sRange+i] *
            buffer[bLen-sRange+i-lag];
        ftmp2 += buffer[bLen-sRange+i-lag] *
                buffer[bLen-sRange+i-lag];
        ftmp3 += buffer[bLen-sRange+i] *
                buffer[bLen-sRange+i];
    }

    if (ftmp2 > 0.0) {
        *cc = ftmp1*ftmp1/ftmp2;
        *gc = (float)fabs(ftmp1/ftmp2);
        *pm=(float)fabs(ftmp1)/
            ((float)sqrt(ftmp2)*(float)sqrt(ftmp3));
    }
    else {
        *cc = 0.0;
        *gc = 0.0;
        *pm=0.0;
    }
}

/*----------------------------------------------------------------*
 *  Packet loss concealment routine. Conceals a residual signal
 *  and LP parameters. If no packet loss, update state.
 *---------------------------------------------------------------*/

void doThePLC(
    float *PLCresidual, /* (o) concealed residual */
    float *PLClpc,      /* (o) concealed LP parameters */
    int PLI,        /* (i) packet loss indicator
```

```
                              0 - no PL, 1 = PL */
    float *decresidual, /* (i) decoded residual */
    float *lpc,         /* (i) decoded LPC (only used for no PL) */
    int inlag,          /* (i) pitch lag */
    iLBC_Dec_Inst_t *iLBCdec_inst
                        /* (i/o) decoder instance */
){
    int lag=20, randlag;
    float gain, maxcc;
    float use_gain;
    float gain_comp, maxcc_comp, per, max_per;
    int i, pick, use_lag;
```



```
    float ftmp, randvec[BLOCKL_MAX], pitchfact, energy;

    /* Packet Loss */

    if (PLI == 1) {

        iLBCdec_inst->consPLICount += 1;

        /* if previous frame not lost,
           determine pitch pred. gain */

        if (iLBCdec_inst->prevPLI != 1) {

            /* Search around the previous lag to find the
               best pitch period */

            lag=inlag-3;
            compCorr(&maxcc, &gain, &max_per,
                iLBCdec_inst->prevResidual,
                lag, iLBCdec_inst->blockl, 60);
            for (i=inlag-2;i<=inlag+3;i++) {
                compCorr(&maxcc_comp, &gain_comp, &per,
                    iLBCdec_inst->prevResidual,
                    i, iLBCdec_inst->blockl, 60);

                if (maxcc_comp>maxcc) {
                    maxcc=maxcc_comp;
                    gain=gain_comp;
                    lag=i;
                    max_per=per;
                }
            }

        }

        /* previous frame lost, use recorded lag and periodicity */
```

```
        else {
            lag=iLBCdec_inst->prevLag;
            max_per=iLBCdec_inst->per;
        }

        /* downscaling */

        use_gain=1.0;
        if (iLBCdec_inst->consPLICount*iLBCdec_inst->blockl>320)
            use_gain=(float)0.9;
        else if (iLBCdec_inst->consPLICount*
                        iLBCdec_inst->blockl>2*320)
            use_gain=(float)0.7;
        else if (iLBCdec_inst->consPLICount*
                        iLBCdec_inst->blockl>3*320)
            use_gain=(float)0.5;
        else if (iLBCdec_inst->consPLICount*
```



```
                        iLBCdec_inst->blockl>4*320)
            use_gain=(float)0.0;

        /* mix noise and pitch repeatition */
        ftmp=(float)sqrt(max_per);
        if (ftmp>(float)0.7)
            pitchfact=(float)1.0;
        else if (ftmp>(float)0.4)
            pitchfact=(ftmp-(float)0.4)/((float)0.7-(float)0.4);
        else
            pitchfact=0.0;


        /* avoid repetition of same pitch cycle */
        use_lag=lag;
        if (lag<80) {
            use_lag=2*lag;
        }

        /* compute concealed residual */

        energy = 0.0;
        for (i=0; i<iLBCdec_inst->blockl; i++) {

            /* noise component */

            iLBCdec_inst->seed=(iLBCdec_inst->seed*69069L+1) &
                (0x80000000L-1);
            randlag = 50 + ((signed long) iLBCdec_inst->seed)%70;
            pick = i - randlag;
```

```
            if (pick < 0) {
                randvec[i] =
                    iLBCdec_inst->prevResidual[
                                iLBCdec_inst->blockl+pick];
            } else {
                randvec[i] =  randvec[pick];
            }

            /* pitch repeatition component */
            pick = i - use_lag;

            if (pick < 0) {
                PLCresidual[i] =
                    iLBCdec_inst->prevResidual[
                                iLBCdec_inst->blockl+pick];
            } else {
                PLCresidual[i] = PLCresidual[pick];
            }

            /* mix random and periodicity component */

            if (i<80)
                PLCresidual[i] = use_gain*(pitchfact *
```



```
                            PLCresidual[i] +
                            ((float)1.0 - pitchfact) * randvec[i]);
            else if (i<160)
                PLCresidual[i] = (float)0.95*use_gain*(pitchfact *
                            PLCresidual[i] +
                            ((float)1.0 - pitchfact) * randvec[i]);
            else
                PLCresidual[i] = (float)0.9*use_gain*(pitchfact *
                            PLCresidual[i] +
                            ((float)1.0 - pitchfact) * randvec[i]);

            energy += PLCresidual[i] * PLCresidual[i];
        }

        /* less than 30 dB, use only noise */

        if (sqrt(energy/(float)iLBCdec_inst->blockl) < 30.0) {
            gain=0.0;
            for (i=0; i<iLBCdec_inst->blockl; i++) {
                PLCresidual[i] = randvec[i];
            }
        }

        /* use old LPC */
```

```
            memcpy(PLClpc,iLBCdec_inst->prevLpc,
                (LPC_FILTERORDER+1)*sizeof(float));

        }

        /* no packet loss, copy input */

        else {
            memcpy(PLCresidual, decresidual,
                iLBCdec_inst->blockl*sizeof(float));
            memcpy(PLClpc, lpc, (LPC_FILTERORDER+1)*sizeof(float));
            iLBCdec_inst->consPLICount = 0;
        }

        /* update state */

        if (PLI) {
            iLBCdec_inst->prevLag = lag;
            iLBCdec_inst->per=max_per;
        }

        iLBCdec_inst->prevPLI = PLI;
        memcpy(iLBCdec_inst->prevLpc, PLClpc,
            (LPC_FILTERORDER+1)*sizeof(float));
        memcpy(iLBCdec_inst->prevResidual, PLCresidual,
            iLBCdec_inst->blockl*sizeof(float));
    }
```


A.15 enhancer.h

```
   /******************************************************************

       iLBC Speech Coder ANSI-C Source Code

       enhancer.h

       Copyright (C) The Internet Society (2004).
       All Rights Reserved.

   ******************************************************************/

   #ifndef __ENHANCER_H
   #define __ENHANCER_H

   #include "iLBC_define.h"
```

```
   float xCorrCoef(
       float *target,      /* (i) first array */
       float *regressor,   /* (i) second array */
       int subl        /* (i) dimension arrays */
   );

   int enhancerInterface(
       float *out,         /* (o) the enhanced recidual signal */
       float *in,          /* (i) the recidual signal to enhance */
       iLBC_Dec_Inst_t *iLBCdec_inst
                           /* (i/o) the decoder state structure */
   );

   #endif
```

A.16 enhancer.c

```
   /******************************************************************

       iLBC Speech Coder ANSI-C Source Code

       enhancer.c

       Copyright (C) The Internet Society (2004).
       All Rights Reserved.

   ******************************************************************/

   #include <math.h>
   #include <string.h>
   #include "iLBC_define.h"
   #include "constants.h"
   #include "filter.h"

   /*----------------------------------------------------------------*
```



```
    * Find index in array such that the array element with said
    * index is the element of said array closest to "value"
    * according to the squared-error criterion
    *---------------------------------------------------------------*/

   void NearestNeighbor(
       int   *index,   /* (o) index of array element closest
                           to value */
       float *array,   /* (i) data array */
       float value,/* (i) value */
       int arlength/* (i) dimension of data array */
   ){
```

```
    int i;
    float bestcrit,crit;

    crit=array[0]-value;
    bestcrit=crit*crit;
    *index=0;
    for (i=1; i<arlength; i++) {
        crit=array[i]-value;
        crit=crit*crit;

        if (crit<bestcrit) {
            bestcrit=crit;
            *index=i;
        }
    }
}

/*----------------------------------------------------------------*
 * compute cross correlation between sequences
 *---------------------------------------------------------------*/

void mycorr1(
    float* corr,    /* (o) correlation of seq1 and seq2 */
    float* seq1,    /* (i) first sequence */
    int dim1,           /* (i) dimension first seq1 */
    const float *seq2,  /* (i) second sequence */
    int dim2        /* (i) dimension seq2 */
){
    int i,j;

    for (i=0; i<=dim1-dim2; i++) {
        corr[i]=0.0;
        for (j=0; j<dim2; j++) {
            corr[i] += seq1[i+j] * seq2[j];
        }
    }
}

/*----------------------------------------------------------------*
 * upsample finite array assuming zeros outside bounds
 *---------------------------------------------------------------*/
```



```
void enh_upsample(
    float* useq1,   /* (o) upsampled output sequence */
    float* seq1,/* (i) unupsampled sequence */
    int dim1,       /* (i) dimension seq1 */
    int hfl         /* (i) polyphase filter length=2*hfl+1 */
```

```
){
    float *pu,*ps;
    int i,j,k,q,filterlength,hfl2;
    const float *polyp[ENH_UPS0]; /* pointers to
                                     polyphase columns */
    const float *pp;

    /* define pointers for filter */

    filterlength=2*hfl+1;

    if ( filterlength > dim1 ) {
        hfl2=(int) (dim1/2);
        for (j=0; j<ENH_UPS0; j++) {
            polyp[j]=polyphaserTbl+j*filterlength+hfl-hfl2;
        }
        hfl=hfl2;
        filterlength=2*hfl+1;
    }
    else {
        for (j=0; j<ENH_UPS0; j++) {
            polyp[j]=polyphaserTbl+j*filterlength;
        }
    }

    /* filtering: filter overhangs left side of sequence */

    pu=useq1;
    for (i=hfl; i<filterlength; i++) {
        for (j=0; j<ENH_UPS0; j++) {
            *pu=0.0;
            pp = polyp[j];
            ps = seq1+i;
            for (k=0; k<=i; k++) {
                *pu += *ps-- * *pp++;
            }
            pu++;
        }
    }

    /* filtering: simple convolution=inner products */

    for (i=filterlength; i<dim1; i++) {
        for (j=0;j<ENH_UPS0; j++){
            *pu=0.0;
            pp = polyp[j];
            ps = seq1+i;
            for (k=0; k<filterlength; k++) {
                *pu += *ps-- * *pp++;
```

```
            }
            pu++;
        }
    }

    /* filtering: filter overhangs right side of sequence */

    for (q=1; q<=hfl; q++) {
        for (j=0; j<ENH_UPS0; j++) {
            *pu=0.0;
            pp = polyp[j]+q;
            ps = seq1+dim1-1;
            for (k=0; k<filterlength-q; k++) {
                *pu += *ps-- * *pp++;
            }
            pu++;
        }
    }
}


/*----------------------------------------------------------------*
 * find segment starting near idata+estSegPos that has highest
 * correlation with idata+centerStartPos through
 * idata+centerStartPos+ENH_BLOCKL-1 segment is found at a
 * resolution of ENH_UPSO times the original of the original
 * sampling rate
 *---------------------------------------------------------------*/

void refiner(
    float *seg,         /* (o) segment array */
    float *updStartPos, /* (o) updated start point */
    float* idata,       /* (i) original data buffer */
    int idatal,         /* (i) dimension of idata */
    int centerStartPos, /* (i) beginning center segment */
    float estSegPos,/* (i) estimated beginning other segment */
    float period    /* (i) estimated pitch period */
){
    int estSegPosRounded,searchSegStartPos,searchSegEndPos,corrdim;
    int tloc,tloc2,i,st,en,fraction;
    float vect[ENH_VECTL],corrVec[ENH_CORRDIM],maxv;
    float corrVecUps[ENH_CORRDIM*ENH_UPS0];

    /* defining array bounds */

    estSegPosRounded=(int)(estSegPos - 0.5);

    searchSegStartPos=estSegPosRounded-ENH_SLOP;

    if (searchSegStartPos<0) {
```

```
        searchSegStartPos=0;
    }
    searchSegEndPos=estSegPosRounded+ENH_SLOP;
```



```
    if (searchSegEndPos+ENH_BLOCKL >= idatal) {
        searchSegEndPos=idatal-ENH_BLOCKL-1;
    }
    corrdim=searchSegEndPos-searchSegStartPos+1;

    /* compute upsampled correlation (corr33) and find
       location of max */

    mycorr1(corrVec,idata+searchSegStartPos,
        corrdim+ENH_BLOCKL-1,idata+centerStartPos,ENH_BLOCKL);
    enh_upsample(corrVecUps,corrVec,corrdim,ENH_FL0);
    tloc=0; maxv=corrVecUps[0];
    for (i=1; i<ENH_UPS0*corrdim; i++) {

        if (corrVecUps[i]>maxv) {
            tloc=i;
            maxv=corrVecUps[i];
        }
    }

    /* make vector can be upsampled without ever running outside
       bounds */

    *updStartPos= (float)searchSegStartPos +
        (float)tloc/(float)ENH_UPS0+(float)1.0;
    tloc2=(int)(tloc/ENH_UPS0);

    if (tloc>tloc2*ENH_UPS0) {
        tloc2++;
    }
    st=searchSegStartPos+tloc2-ENH_FL0;

    if (st<0) {
        memset(vect,0,-st*sizeof(float));
        memcpy(&vect[-st],idata, (ENH_VECTL+st)*sizeof(float));
    }
    else {
        en=st+ENH_VECTL;

        if (en>idatal) {
            memcpy(vect, &idata[st],
                (ENH_VECTL-(en-idatal))*sizeof(float));
            memset(&vect[ENH_VECTL-(en-idatal)], 0,
                (en-idatal)*sizeof(float));
```

```
        }
        else {
            memcpy(vect, &idata[st], ENH_VECTL*sizeof(float));
        }
    }
    fraction=tloc2*ENH_UPS0-tloc;

    /* compute the segment (this is actually a convolution) */

    mycorr1(seg,vect,ENH_VECTL,polyphaserTbl+(2*ENH_FL0+1)*fraction,
```



```
        2*ENH_FL0+1);
}

/*----------------------------------------------------------------*
 * find the smoothed output data
 *---------------------------------------------------------------*/

void smath(
    float *odata,   /* (o) smoothed output */
    float *sseq,/* (i) said second sequence of waveforms */
    int hl,         /* (i) 2*hl+1 is sseq dimension */
    float alpha0/* (i) max smoothing energy fraction */
){
    int i,k;
    float w00,w10,w11,A,B,C,*psseq,err,errs;
    float surround[BLOCKL_MAX]; /* shape contributed by other than
                                   current */
    float wt[2*ENH_HL+1];       /* waveform weighting to get
                                   surround shape */
    float denom;

    /* create shape of contribution from all waveforms except the
       current one */

    for (i=1; i<=2*hl+1; i++) {
        wt[i-1] = (float)0.5*(1 - (float)cos(2*PI*i/(2*hl+2)));
    }
    wt[hl]=0.0; /* for clarity, not used */
    for (i=0; i<ENH_BLOCKL; i++) {
        surround[i]=sseq[i]*wt[0];
    }
    for (k=1; k<hl; k++) {
        psseq=sseq+k*ENH_BLOCKL;
        for(i=0;i<ENH_BLOCKL; i++) {
            surround[i]+=psseq[i]*wt[k];
        }
    }
```

```
    for (k=hl+1; k<=2*hl; k++) {
        psseq=sseq+k*ENH_BLOCKL;
        for(i=0;i<ENH_BLOCKL; i++) {
            surround[i]+=psseq[i]*wt[k];
        }
    }

    /* compute some inner products */

    w00 = w10 = w11 = 0.0;
    psseq=sseq+hl*ENH_BLOCKL; /* current block  */
    for (i=0; i<ENH_BLOCKL;i++) {
        w00+=psseq[i]*psseq[i];
        w11+=surround[i]*surround[i];
        w10+=surround[i]*psseq[i];
    }
```



```
    if (fabs(w11) < 1.0) {
        w11=1.0;
    }
    C = (float)sqrt( w00/w11);

    /* first try enhancement without power-constraint */

    errs=0.0;
    psseq=sseq+hl*ENH_BLOCKL;
    for (i=0; i<ENH_BLOCKL; i++) {
        odata[i]=C*surround[i];
        err=psseq[i]-odata[i];
        errs+=err*err;
    }

    /* if constraint violated by first try, add constraint */

    if (errs > alpha0 * w00) {
        if ( w00 < 1) {
            w00=1;
        }
        denom = (w11*w00-w10*w10)/(w00*w00);

        if (denom > 0.0001) { /* eliminates numerical problems
                                 for if smooth */
            A = (float)sqrt( (alpha0- alpha0*alpha0/4)/denom);
            B = -alpha0/2 - A * w10/w00;
            B = B+1;
        }
        else { /* essentially no difference between cycles;
                  smoothing not needed */
```

```
            A= 0.0;
            B= 1.0;
        }

        /* create smoothed sequence */

        psseq=sseq+hl*ENH_BLOCKL;
        for (i=0; i<ENH_BLOCKL; i++) {
            odata[i]=A*surround[i]+B*psseq[i];
        }
    }
}

/*----------------------------------------------------------------*
 * get the pitch-synchronous sample sequence
 *---------------------------------------------------------------*/

void getsseq(
    float *sseq,    /* (o) the pitch-synchronous sequence */
    float *idata,       /* (i) original data */
    int idatal,         /* (i) dimension of data */
    int centerStartPos, /* (i) where current block starts */
    float *period,      /* (i) rough-pitch-period array */
```



```
    float *plocs,       /* (i) where periods of period array
                               are taken */
    int periodl,    /* (i) dimension period array */
    int hl              /* (i) 2*hl+1 is the number of sequences */
){
    int i,centerEndPos,q;
    float blockStartPos[2*ENH_HL+1];
    int lagBlock[2*ENH_HL+1];
    float plocs2[ENH_PLOCSL];
    float *psseq;

    centerEndPos=centerStartPos+ENH_BLOCKL-1;

    /* present */

    NearestNeighbor(lagBlock+hl,plocs,
        (float)0.5*(centerStartPos+centerEndPos),periodl);

    blockStartPos[hl]=(float)centerStartPos;
    psseq=sseq+ENH_BLOCKL*hl;
    memcpy(psseq, idata+centerStartPos, ENH_BLOCKL*sizeof(float));

    /* past */

    for (q=hl-1; q>=0; q--) {
```

```
        blockStartPos[q]=blockStartPos[q+1]-period[lagBlock[q+1]];
        NearestNeighbor(lagBlock+q,plocs,
            blockStartPos[q]+
            ENH_BLOCKL_HALF-period[lagBlock[q+1]], periodl);


        if (blockStartPos[q]-ENH_OVERHANG>=0) {
            refiner(sseq+q*ENH_BLOCKL, blockStartPos+q, idata,
                idatal, centerStartPos, blockStartPos[q],
                period[lagBlock[q+1]]);
        } else {
            psseq=sseq+q*ENH_BLOCKL;
            memset(psseq, 0, ENH_BLOCKL*sizeof(float));
        }
    }

    /* future */

    for (i=0; i<periodl; i++) {
        plocs2[i]=plocs[i]-period[i];
    }
    for (q=hl+1; q<=2*hl; q++) {
        NearestNeighbor(lagBlock+q,plocs2,
            blockStartPos[q-1]+ENH_BLOCKL_HALF,periodl);

        blockStartPos[q]=blockStartPos[q-1]+period[lagBlock[q]];
        if (blockStartPos[q]+ENH_BLOCKL+ENH_OVERHANG<idatal) {
            refiner(sseq+ENH_BLOCKL*q, blockStartPos+q, idata,
                idatal, centerStartPos, blockStartPos[q],
```



```
                period[lagBlock[q]]);
        }
        else {
            psseq=sseq+q*ENH_BLOCKL;
            memset(psseq, 0, ENH_BLOCKL*sizeof(float));
        }
    }
}

/*----------------------------------------------------------------*
 * perform enhancement on idata+centerStartPos through
 * idata+centerStartPos+ENH_BLOCKL-1
 *---------------------------------------------------------------*/

void enhancer(
    float *odata,       /* (o) smoothed block, dimension blockl */
    float *idata,       /* (i) data buffer used for enhancing */
    int idatal,         /* (i) dimension idata */
```

```
    int centerStartPos, /* (i) first sample current block
                               within idata */
    float alpha0,       /* (i) max correction-energy-fraction
                              (in [0,1]) */
    float *period,      /* (i) pitch period array */
    float *plocs,       /* (i) locations where period array
                               values valid */
    int periodl         /* (i) dimension of period and plocs */
){
    float sseq[(2*ENH_HL+1)*ENH_BLOCKL];

    /* get said second sequence of segments */

    getsseq(sseq,idata,idatal,centerStartPos,period,
        plocs,periodl,ENH_HL);

    /* compute the smoothed output from said second sequence */

    smath(odata,sseq,ENH_HL,alpha0);

}

/*----------------------------------------------------------------*
 * cross correlation
 *---------------------------------------------------------------*/

float xCorrCoef(
    float *target,      /* (i) first array */
    float *regressor,   /* (i) second array */
    int subl        /* (i) dimension arrays */
){
    int i;
    float ftmp1, ftmp2;

    ftmp1 = 0.0;
    ftmp2 = 0.0;
```



```
    for (i=0; i<subl; i++) {
        ftmp1 += target[i]*regressor[i];
        ftmp2 += regressor[i]*regressor[i];
    }

    if (ftmp1 > 0.0) {
        return (float)(ftmp1*ftmp1/ftmp2);
    }
    else {
        return (float)0.0;
    }
}
```

```
/*----------------------------------------------------------------*
 * interface for enhancer
 *---------------------------------------------------------------*/

int enhancerInterface(
    float *out,                     /* (o) enhanced signal */
    float *in,                      /* (i) unenhanced signal */
    iLBC_Dec_Inst_t *iLBCdec_inst   /* (i) buffers etc */
){
    float *enh_buf, *enh_period;
    int iblock, isample;
    int lag=0, ilag, i, ioffset;
    float cc, maxcc;
    float ftmp1, ftmp2;
    float *inPtr, *enh_bufPtr1, *enh_bufPtr2;
    float plc_pred[ENH_BLOCKL];

    float lpState[6], downsampled[(ENH_NBLOCKS*ENH_BLOCKL+120)/2];
    int inLen=ENH_NBLOCKS*ENH_BLOCKL+120;
    int start, plc_blockl, inlag;

    enh_buf=iLBCdec_inst->enh_buf;
    enh_period=iLBCdec_inst->enh_period;

    memmove(enh_buf, &enh_buf[iLBCdec_inst->blockl],
        (ENH_BUFL-iLBCdec_inst->blockl)*sizeof(float));

    memcpy(&enh_buf[ENH_BUFL-iLBCdec_inst->blockl], in,
        iLBCdec_inst->blockl*sizeof(float));

    if (iLBCdec_inst->mode==30)
        plc_blockl=ENH_BLOCKL;
    else
        plc_blockl=40;

    /* when 20 ms frame, move processing one block */
    ioffset=0;
    if (iLBCdec_inst->mode==20) ioffset=1;

    i=3-ioffset;
    memmove(enh_period, &enh_period[i],
```



```
        (ENH_NBLOCKS_TOT-i)*sizeof(float));

    /* Set state information to the 6 samples right before
       the samples to be downsampled. */

    memcpy(lpState,
```

```
        enh_buf+(ENH_NBLOCKS_EXTRA+ioffset)*ENH_BLOCKL-126,
        6*sizeof(float));

    /* Down sample a factor 2 to save computations */

    DownSample(enh_buf+(ENH_NBLOCKS_EXTRA+ioffset)*ENH_BLOCKL-120,
                lpFilt_coefsTbl, inLen-ioffset*ENH_BLOCKL,
                lpState, downsampled);

    /* Estimate the pitch in the down sampled domain. */
    for (iblock = 0; iblock<ENH_NBLOCKS-ioffset; iblock++) {

        lag = 10;
        maxcc = xCorrCoef(downsampled+60+iblock*
            ENH_BLOCKL_HALF, downsampled+60+iblock*
            ENH_BLOCKL_HALF-lag, ENH_BLOCKL_HALF);
        for (ilag=11; ilag<60; ilag++) {
            cc = xCorrCoef(downsampled+60+iblock*
                ENH_BLOCKL_HALF, downsampled+60+iblock*
                ENH_BLOCKL_HALF-ilag, ENH_BLOCKL_HALF);

            if (cc > maxcc) {
                maxcc = cc;
                lag = ilag;
            }
        }

        /* Store the estimated lag in the non-downsampled domain */
        enh_period[iblock+ENH_NBLOCKS_EXTRA+ioffset] = (float)lag*2;


    }


    /* PLC was performed on the previous packet */
    if (iLBCdec_inst->prev_enh_pl==1) {

        inlag=(int)enh_period[ENH_NBLOCKS_EXTRA+ioffset];

        lag = inlag-1;
        maxcc = xCorrCoef(in, in+lag, plc_blockl);
        for (ilag=inlag; ilag<=inlag+1; ilag++) {
            cc = xCorrCoef(in, in+ilag, plc_blockl);

            if (cc > maxcc) {
                maxcc = cc;
                lag = ilag;
            }
        }
```

```
            enh_period[ENH_NBLOCKS_EXTRA+ioffset-1]=(float)lag;

            /* compute new concealed residual for the old lookahead,
               mix the forward PLC with a backward PLC from
               the new frame */

            inPtr=&in[lag-1];

            enh_bufPtr1=&plc_pred[plc_blockl-1];

            if (lag>plc_blockl) {
                start=plc_blockl;
            } else {
                start=lag;
            }

            for (isample = start; isample>0; isample--) {
                *enh_bufPtr1-- = *inPtr--;
            }

            enh_bufPtr2=&enh_buf[ENH_BUFL-1-iLBCdec_inst->blockl];
            for (isample = (plc_blockl-1-lag); isample>=0; isample--)
{
                *enh_bufPtr1-- = *enh_bufPtr2--;
            }

            /* limit energy change */
            ftmp2=0.0;
            ftmp1=0.0;
            for (i=0;i<plc_blockl;i++) {
                ftmp2+=enh_buf[ENH_BUFL-1-iLBCdec_inst->blockl-i]*
                    enh_buf[ENH_BUFL-1-iLBCdec_inst->blockl-i];
                ftmp1+=plc_pred[i]*plc_pred[i];
            }
            ftmp1=(float)sqrt(ftmp1/(float)plc_blockl);
            ftmp2=(float)sqrt(ftmp2/(float)plc_blockl);
            if (ftmp1>(float)2.0*ftmp2 && ftmp1>0.0) {
                for (i=0;i<plc_blockl-10;i++) {
                    plc_pred[i]*=(float)2.0*ftmp2/ftmp1;
                }
                for (i=plc_blockl-10;i<plc_blockl;i++) {
                    plc_pred[i]*=(float)(i-plc_blockl+10)*
                        ((float)1.0-(float)2.0*ftmp2/ftmp1)/(float)(10)+
                        (float)2.0*ftmp2/ftmp1;
                }
            }

            enh_bufPtr1=&enh_buf[ENH_BUFL-1-iLBCdec_inst->blockl];
            for (i=0; i<plc_blockl; i++) {
```

```
            ftmp1 = (float) (i+1) / (float) (plc_blockl+1);
            *enh_bufPtr1 *= ftmp1;
            *enh_bufPtr1 += ((float)1.0-ftmp1)*
                                plc_pred[plc_blockl-1-i];
            enh_bufPtr1--;
        }
```



```
    }

    if (iLBCdec_inst->mode==20) {
        /* Enhancer with 40 samples delay */
        for (iblock = 0; iblock<2; iblock++) {
            enhancer(out+iblock*ENH_BLOCKL, enh_buf,
                ENH_BUFL, (5+iblock)*ENH_BLOCKL+40,
                ENH_ALPHA0, enh_period, enh_plocsTbl,
                    ENH_NBLOCKS_TOT);
        }
    } else if (iLBCdec_inst->mode==30) {
        /* Enhancer with 80 samples delay */
        for (iblock = 0; iblock<3; iblock++) {
            enhancer(out+iblock*ENH_BLOCKL, enh_buf,
                ENH_BUFL, (4+iblock)*ENH_BLOCKL,
                ENH_ALPHA0, enh_period, enh_plocsTbl,
                    ENH_NBLOCKS_TOT);
        }
    }

    return (lag*2);
}
```

A.17 filter.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    filter.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_FILTER_H
#define __iLBC_FILTER_H

void AllPoleFilter(
    float *InOut,   /* (i/o) on entrance InOut[-orderCoef] to
```

```
                            InOut[-1] contain the state of the
                            filter (delayed samples). InOut[0] to
                            InOut[lengthInOut-1] contain the filter
                            input, on en exit InOut[-orderCoef] to
                            InOut[-1] is unchanged and InOut[0] to
                            InOut[lengthInOut-1] contain filtered
                            samples */
    float *Coef,/* (i) filter coefficients, Coef[0] is assumed
                            to be 1.0 */
    int lengthInOut,/* (i) number of input/output samples */
    int orderCoef   /* (i) number of filter coefficients */
);
```



```
void AllZeroFilter(
    float *In,      /* (i) In[0] to In[lengthInOut-1] contain
                            filter input samples */
    float *Coef,/* (i) filter coefficients (Coef[0] is assumed
                            to be 1.0) */
    int lengthInOut,/* (i) number of input/output samples */
    int orderCoef,  /* (i) number of filter coefficients */
    float *Out      /* (i/o) on entrance Out[-orderCoef] to Out[-1]
                            contain the filter state, on exit Out[0]
                            to Out[lengthInOut-1] contain filtered
                            samples */
);

void ZeroPoleFilter(
    float *In,      /* (i) In[0] to In[lengthInOut-1] contain filter
                            input samples In[-orderCoef] to In[-1]
                            contain state of all-zero section */
    float *ZeroCoef,/* (i) filter coefficients for all-zero
                            section (ZeroCoef[0] is assumed to
                            be 1.0) */
    float *PoleCoef,/* (i) filter coefficients for all-pole section
                            (ZeroCoef[0] is assumed to be 1.0) */
    int lengthInOut,/* (i) number of input/output samples */
    int orderCoef,  /* (i) number of filter coefficients */
    float *Out      /* (i/o) on entrance Out[-orderCoef] to Out[-1]
                            contain state of all-pole section. On
                            exit Out[0] to Out[lengthInOut-1]
                            contain filtered samples */
);

void DownSample (
    float  *In,     /* (i) input samples */
    float  *Coef,   /* (i) filter coefficients */
    int lengthIn,   /* (i) number of input samples */
```

```
    float  *state,  /* (i) filter state */
    float  *Out     /* (o) downsampled output */
);

#endif
```

A.18 filter.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    filter.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/
```



```
#include "iLBC_define.h"

/*----------------------------------------------------------------*
 *  all-pole filter
 *---------------------------------------------------------------*/

void AllPoleFilter(
    float *InOut,   /* (i/o) on entrance InOut[-orderCoef] to
                           InOut[-1] contain the state of the
                           filter (delayed samples). InOut[0] to
                           InOut[lengthInOut-1] contain the filter
                           input, on en exit InOut[-orderCoef] to
                           InOut[-1] is unchanged and InOut[0] to
                           InOut[lengthInOut-1] contain filtered
                           samples */
    float *Coef,/* (i) filter coefficients, Coef[0] is assumed
                           to be 1.0 */
    int lengthInOut,/* (i) number of input/output samples */
    int orderCoef   /* (i) number of filter coefficients */
){
    int n,k;

    for(n=0;n<lengthInOut;n++){
        for(k=1;k<=orderCoef;k++){
            *InOut -= Coef[k]*InOut[-k];
        }
        InOut++;
    }
```

```
}

/*----------------------------------------------------------------*
 *  all-zero filter
 *---------------------------------------------------------------*/

void AllZeroFilter(
    float *In,      /* (i) In[0] to In[lengthInOut-1] contain
                           filter input samples */
    float *Coef,/* (i) filter coefficients (Coef[0] is assumed
                           to be 1.0) */
    int lengthInOut,/* (i) number of input/output samples */
    int orderCoef,  /* (i) number of filter coefficients */
    float *Out      /* (i/o) on entrance Out[-orderCoef] to Out[-1]
                           contain the filter state, on exit Out[0]
                           to Out[lengthInOut-1] contain filtered
                           samples */
){
    int n,k;

    for(n=0;n<lengthInOut;n++){
        *Out = Coef[0]*In[0];
        for(k=1;k<=orderCoef;k++){
            *Out += Coef[k]*In[-k];
        }
```



```
        Out++;
        In++;
    }
}

/*----------------------------------------------------------------*
 *  pole-zero filter
 *---------------------------------------------------------------*/

void ZeroPoleFilter(
    float *In,      /* (i) In[0] to In[lengthInOut-1] contain
                           filter input samples In[-orderCoef] to
                           In[-1] contain state of all-zero
                           section */
    float *ZeroCoef,/* (i) filter coefficients for all-zero
                           section (ZeroCoef[0] is assumed to
                           be 1.0) */
    float *PoleCoef,/* (i) filter coefficients for all-pole section
                           (ZeroCoef[0] is assumed to be 1.0) */
    int lengthInOut,/* (i) number of input/output samples */
    int orderCoef,  /* (i) number of filter coefficients */
    float *Out      /* (i/o) on entrance Out[-orderCoef] to Out[-1]
```

```
                           contain state of all-pole section. On
                           exit Out[0] to Out[lengthInOut-1]
                           contain filtered samples */
){
    AllZeroFilter(In,ZeroCoef,lengthInOut,orderCoef,Out);
    AllPoleFilter(Out,PoleCoef,lengthInOut,orderCoef);
}

/*----------------------------------------------------------------*
 * downsample (LP filter and decimation)
 *---------------------------------------------------------------*/

void DownSample (
    float  *In,     /* (i) input samples */
    float  *Coef,   /* (i) filter coefficients */
    int lengthIn,   /* (i) number of input samples */
    float  *state,  /* (i) filter state */
    float  *Out     /* (o) downsampled output */
){
    float   o;
    float *Out_ptr = Out;
    float *Coef_ptr, *In_ptr;
    float *state_ptr;
    int i, j, stop;

    /* LP filter and decimate at the same time */

    for (i = DELAY_DS; i < lengthIn; i+=FACTOR_DS)
    {
        Coef_ptr = &Coef[0];
        In_ptr = &In[i];
        state_ptr = &state[FILTERORDER_DS-2];
```



```
        o = (float)0.0;

        stop = (i < FILTERORDER_DS) ? i + 1 : FILTERORDER_DS;

        for (j = 0; j < stop; j++)
        {
            o += *Coef_ptr++ * (*In_ptr--);
        }
        for (j = i + 1; j < FILTERORDER_DS; j++)
        {
            o += *Coef_ptr++ * (*state_ptr--);
        }

        *Out_ptr++ = o;
    }
```

```
    /* Get the last part (use zeros as input for the future) */

    for (i=(lengthIn+FACTOR_DS); i<(lengthIn+DELAY_DS);
            i+=FACTOR_DS) {

        o=(float)0.0;

        if (i<lengthIn) {
            Coef_ptr = &Coef[0];
            In_ptr = &In[i];
            for (j=0; j<FILTERORDER_DS; j++) {
                    o += *Coef_ptr++ * (*Out_ptr--);
            }
        } else {
            Coef_ptr = &Coef[i-lengthIn];
            In_ptr = &In[lengthIn-1];
            for (j=0; j<FILTERORDER_DS-(i-lengthIn); j++) {
                    o += *Coef_ptr++ * (*In_ptr--);
            }
        }
        *Out_ptr++ = o;
    }
}
```

A.19 FrameClassify.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    FrameClassify.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/
```



```
#ifndef __iLBC_FRAMECLASSIFY_H
#define __iLBC_FRAMECLASSIFY_H

int FrameClassify(      /* index to the max-energy sub-frame */
    iLBC_Enc_Inst_t *iLBCenc_inst,
                        /* (i/o) the encoder state structure */
    float *residual     /* (i) lpc residual signal */
);
```

```
#endif
```

A.20 FrameClassify.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    FrameClassify.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include "iLBC_define.h"

/*---------------------------------------------------------------*
 *  Classification of subframes to localize start state
 *--------------------------------------------------------------*/

int FrameClassify(      /* index to the max-energy sub-frame */
    iLBC_Enc_Inst_t *iLBCenc_inst,
                        /* (i/o) the encoder state structure */
    float *residual     /* (i) lpc residual signal */
) {
    float max_ssqEn, fssqEn[NSUB_MAX], bssqEn[NSUB_MAX], *pp;
    int n, l, max_ssqEn_n;
    const float ssqEn_win[NSUB_MAX-1]={(float)0.8,(float)0.9,
        (float)1.0,(float)0.9,(float)0.8};
    const float sampEn_win[5]={(float)1.0/(float)6.0,
        (float)2.0/(float)6.0, (float)3.0/(float)6.0,
        (float)4.0/(float)6.0, (float)5.0/(float)6.0};

    /* init the front and back energies to zero */

    memset(fssqEn, 0, NSUB_MAX*sizeof(float));
    memset(bssqEn, 0, NSUB_MAX*sizeof(float));

    /* Calculate front of first seqence */

    n=0;
    pp=residual;
```



```
    for (l=0; l<5; l++) {
        fssqEn[n] += sampEn_win[l] * (*pp) * (*pp);
        pp++;
    }
```

```
for (l=5; l<SUBL; l++) {
    fssqEn[n] += (*pp) * (*pp);
    pp++;
}

/* Calculate front and back of all middle sequences */

for (n=1; n<iLBCenc_inst->nsub-1; n++) {
    pp=residual+n*SUBL;
    for (l=0; l<5; l++) {
        fssqEn[n] += sampEn_win[l] * (*pp) * (*pp);
        bssqEn[n] += (*pp) * (*pp);
        pp++;
    }
    for (l=5; l<SUBL-5; l++) {
        fssqEn[n] += (*pp) * (*pp);
        bssqEn[n] += (*pp) * (*pp);
        pp++;
    }
    for (l=SUBL-5; l<SUBL; l++) {
        fssqEn[n] += (*pp) * (*pp);
        bssqEn[n] += sampEn_win[SUBL-l-1] * (*pp) * (*pp);
        pp++;
    }
}

/* Calculate back of last seqence */

n=iLBCenc_inst->nsub-1;
pp=residual+n*SUBL;
for (l=0; l<SUBL-5; l++) {
    bssqEn[n] += (*pp) * (*pp);
    pp++;
}
for (l=SUBL-5; l<SUBL; l++) {
    bssqEn[n] += sampEn_win[SUBL-l-1] * (*pp) * (*pp);
    pp++;
}

/* find the index to the weighted 80 sample with
   most energy */

if (iLBCenc_inst->mode==20) l=1;
else                        l=0;

max_ssqEn=(fssqEn[0]+bssqEn[1])*ssqEn_win[l];
max_ssqEn_n=1;
for (n=2; n<iLBCenc_inst->nsub; n++) {

    l++;
```

```
        if ((fssqEn[n-1]+bssqEn[n])*ssqEn_win[l] > max_ssqEn) {
            max_ssqEn=(fssqEn[n-1]+bssqEn[n]) *
                            ssqEn_win[l];
            max_ssqEn_n=n;
        }
    }

    return max_ssqEn_n;
}
```

A.21 gainquant.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    gainquant.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_GAINQUANT_H
#define __iLBC_GAINQUANT_H

float gainquant(/* (o) quantized gain value */
    float in,       /* (i) gain value */
    float maxIn,/* (i) maximum of gain value */
    int cblen,      /* (i) number of quantization indices */
    int *index      /* (o) quantization index */
);

float gaindequant(  /* (o) quantized gain value */
    int index,      /* (i) quantization index */
    float maxIn,/* (i) maximum of unquantized gain */
    int cblen       /* (i) number of quantization indices */
);

#endif
```

A.22 gainquant.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code
```

```
    gainquant.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.
```



```
******************************************************************/

#include <string.h>
#include <math.h>
#include "constants.h"
#include "filter.h"

/*----------------------------------------------------------------*
 *  quantizer for the gain in the gain-shape coding of residual
 *---------------------------------------------------------------*/

float gainquant(/* (o) quantized gain value */
    float in,       /* (i) gain value */
    float maxIn,/* (i) maximum of gain value */
    int cblen,      /* (i) number of quantization indices */
    int *index      /* (o) quantization index */
){
    int i, tindex;
    float minmeasure,measure, *cb, scale;

    /* ensure a lower bound on the scaling factor */

    scale=maxIn;

    if (scale<0.1) {
        scale=(float)0.1;
    }

    /* select the quantization table */

    if (cblen == 8) {
        cb = gain_sq3Tbl;
    } else if (cblen == 16) {
        cb = gain_sq4Tbl;
    } else  {
        cb = gain_sq5Tbl;
    }

    /* select the best index in the quantization table */

    minmeasure=10000000.0;
    tindex=0;
    for (i=0; i<cblen; i++) {
```

```
        measure=(in-scale*cb[i])*(in-scale*cb[i]);

        if (measure<minmeasure) {
            tindex=i;
            minmeasure=measure;
        }
    }
    *index=tindex;

    /* return the quantized value */
```



```
    return scale*cb[tindex];
}

/*----------------------------------------------------------------*
 *  decoder for quantized gains in the gain-shape coding of
 *  residual
 *---------------------------------------------------------------*/

float gaindequant(  /* (o) quantized gain value */
    int index,      /* (i) quantization index */
    float maxIn,/* (i) maximum of unquantized gain */
    int cblen       /* (i) number of quantization indices */
){
    float scale;

    /* obtain correct scale factor */

    scale=(float)fabs(maxIn);

    if (scale<0.1) {
        scale=(float)0.1;
    }

    /* select the quantization table and return the decoded value */

    if (cblen==8) {
        return scale*gain_sq3Tbl[index];
    } else if (cblen==16) {
        return scale*gain_sq4Tbl[index];
    }
    else if (cblen==32) {
        return scale*gain_sq5Tbl[index];
    }

    return 0.0;
}
```

A.23 getCBvec.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    getCBvec.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_GETCBVEC_H
#define __iLBC_GETCBVEC_H
```



```
void getCBvec(
    float *cbvec,   /* (o) Constructed codebook vector */
    float *mem,     /* (i) Codebook buffer */
    int index,      /* (i) Codebook index */
    int lMem,       /* (i) Length of codebook buffer */
    int cbveclen/* (i) Codebook vector length */
);

#endif
```

A.24 getCBvec.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    getCBvec.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include "iLBC_define.h"
#include "constants.h"
#include <string.h>

/*----------------------------------------------------------------*
 *  Construct codebook vector for given index.
 *---------------------------------------------------------------*/
```

```
void getCBvec(
    float *cbvec,   /* (o) Constructed codebook vector */
    float *mem,     /* (i) Codebook buffer */
    int index,      /* (i) Codebook index */
    int lMem,       /* (i) Length of codebook buffer */
    int cbveclen/* (i) Codebook vector length */
){
    int j, k, n, memInd, sFilt;
    float tmpbuf[CB_MEML];
    int base_size;
    int ilow, ihigh;
    float alfa, alfa1;

    /* Determine size of codebook sections */

    base_size=lMem-cbveclen+1;

    if (cbveclen==SUBL) {
        base_size+=cbveclen/2;
    }

    /* No filter -> First codebook section */
```



```
    if (index<lMem-cbveclen+1) {

        /* first non-interpolated vectors */

        k=index+cbveclen;
        /* get vector */
        memcpy(cbvec, mem+lMem-k, cbveclen*sizeof(float));

    } else if (index < base_size) {

        k=2*(index-(lMem-cbveclen+1))+cbveclen;

        ihigh=k/2;
        ilow=ihigh-5;

        /* Copy first noninterpolated part */

        memcpy(cbvec, mem+lMem-k/2, ilow*sizeof(float));

        /* interpolation */

        alfa1=(float)0.2;
        alfa=0.0;
        for (j=ilow; j<ihigh; j++) {
            cbvec[j]=((float)1.0-alfa)*mem[lMem-k/2+j]+
```

```
                    alfa*mem[lMem-k+j];
                alfa+=alfa1;
            }

            /* Copy second noninterpolated part */

            memcpy(cbvec+ihigh, mem+lMem-k+ihigh,
                (cbveclen-ihigh)*sizeof(float));

        }

        /* Higher codebbok section based on filtering */

        else {

            /* first non-interpolated vectors */

            if (index-base_size<lMem-cbveclen+1) {
                float tempbuff2[CB_MEML+CB_FILTERLEN+1];
                float *pos;
                float *pp, *pp1;

                memset(tempbuff2, 0,
                    CB_HALFFILTERLEN*sizeof(float));
                memcpy(&tempbuff2[CB_HALFFILTERLEN], mem,
                    lMem*sizeof(float));
                memset(&tempbuff2[lMem+CB_HALFFILTERLEN], 0,
                    (CB_HALFFILTERLEN+1)*sizeof(float));
```



```
                k=index-base_size+cbveclen;
                sFilt=lMem-k;
                memInd=sFilt+1-CB_HALFFILTERLEN;

                /* do filtering */
                pos=cbvec;
                memset(pos, 0, cbveclen*sizeof(float));
                for (n=0; n<cbveclen; n++) {
                    pp=&tempbuff2[memInd+n+CB_HALFFILTERLEN];
                    pp1=&cbfiltersTbl[CB_FILTERLEN-1];
                    for (j=0; j<CB_FILTERLEN; j++) {
                        (*pos)+=(*pp++)*(*pp1--);
                    }
                    pos++;
                }
            }

            /* interpolated vectors */
```

```
        else {
            float tempbuff2[CB_MEML+CB_FILTERLEN+1];

            float *pos;
            float *pp, *pp1;
            int i;

            memset(tempbuff2, 0,
                CB_HALFFILTERLEN*sizeof(float));
            memcpy(&tempbuff2[CB_HALFFILTERLEN], mem,
                lMem*sizeof(float));
            memset(&tempbuff2[lMem+CB_HALFFILTERLEN], 0,
                (CB_HALFFILTERLEN+1)*sizeof(float));

            k=2*(index-base_size-
                (lMem-cbveclen+1))+cbveclen;
            sFilt=lMem-k;
            memInd=sFilt+1-CB_HALFFILTERLEN;

            /* do filtering */
            pos=&tmpbuf[sFilt];
            memset(pos, 0, k*sizeof(float));
            for (i=0; i<k; i++) {
                pp=&tempbuff2[memInd+i+CB_HALFFILTERLEN];
                pp1=&cbfiltersTbl[CB_FILTERLEN-1];
                for (j=0; j<CB_FILTERLEN; j++) {
                    (*pos)+=(*pp++)*(*pp1--);
                }
                pos++;
            }

            ihigh=k/2;
            ilow=ihigh-5;
```



```
            /* Copy first noninterpolated part */

            memcpy(cbvec, tmpbuf+lMem-k/2,
                ilow*sizeof(float));

            /* interpolation */

            alfa1=(float)0.2;
            alfa=0.0;
            for (j=ilow; j<ihigh; j++) {
                cbvec[j]=((float)1.0-alfa)*
                    tmpbuf[lMem-k/2+j]+alfa*tmpbuf[lMem-k+j];
                alfa+=alfa1;
            }
```

```
            /* Copy second noninterpolated part */

            memcpy(cbvec+ihigh, tmpbuf+lMem-k+ihigh,
                (cbveclen-ihigh)*sizeof(float));
        }
    }
}
```

A.25 helpfun.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    helpfun.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_HELPFUN_H
#define __iLBC_HELPFUN_H

void autocorr(
    float *r,       /* (o) autocorrelation vector */
    const float *x, /* (i) data vector */
    int N,          /* (i) length of data vector */
    int order       /* largest lag for calculated
                       autocorrelations */
);

void window(
    float *z,       /* (o) the windowed data */
    const float *x, /* (i) the original data vector */
    const float *y, /* (i) the window */
    int N           /* (i) length of all vectors */
);
```



```
void levdurb(
    float *a,       /* (o) lpc coefficient vector starting
                           with 1.0 */
    float *k,       /* (o) reflection coefficients */
    float *r,       /* (i) autocorrelation vector */
    int order       /* (i) order of lpc filter */
);
```

```
void interpolate(
    float *out,      /* (o) the interpolated vector */
    float *in1,      /* (i) the first vector for the
                            interpolation */
    float *in2,      /* (i) the second vector for the
                            interpolation */
    float coef,      /* (i) interpolation weights */
    int length       /* (i) length of all vectors */
);

void bwexpand(
    float *out,      /* (o) the bandwidth expanded lpc
                            coefficients */
    float *in,       /* (i) the lpc coefficients before bandwidth
                            expansion */
    float coef,      /* (i) the bandwidth expansion factor */
    int length       /* (i) the length of lpc coefficient vectors */
);

void vq(
    float *Xq,       /* (o) the quantized vector */
    int *index,      /* (o) the quantization index */
    const float *CB,/* (i) the vector quantization codebook */
    float *X,        /* (i) the vector to quantize */
    int n_cb,        /* (i) the number of vectors in the codebook */
    int dim          /* (i) the dimension of all vectors */
);

void SplitVQ(
    float *qX,       /* (o) the quantized vector */
    int *index,      /* (o) a vector of indexes for all vector
                            codebooks in the split */
    float *X,        /* (i) the vector to quantize */
    const float *CB,/* (i) the quantizer codebook */
    int nsplit,      /* the number of vector splits */
    const int *dim, /* the dimension of X and qX */
    const int *cbsize /* the number of vectors in the codebook */
);


void sort_sq(
    float *xq,       /* (o) the quantized value */
    int *index,      /* (o) the quantization index */
    float x,    /* (i) the value to quantize */
    const float *cb,/* (i) the quantization codebook */
```



```
    int cb_size      /* (i) the size of the quantization codebook */
```

```
);

int LSF_check(      /* (o) 1 for stable lsf vectors and 0 for
                           nonstable ones */
    float *lsf,     /* (i) a table of lsf vectors */
    int dim,    /* (i) the dimension of each lsf vector */
    int NoAn    /* (i) the number of lsf vectors in the
                           table */
);

#endif
```

A.26 helpfun.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    helpfun.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <math.h>

#include "iLBC_define.h"
#include "constants.h"

/*----------------------------------------------------------------*
 *  calculation of auto correlation
 *---------------------------------------------------------------*/

void autocorr(
    float *r,       /* (o) autocorrelation vector */
    const float *x, /* (i) data vector */
    int N,          /* (i) length of data vector */
    int order       /* largest lag for calculated
                       autocorrelations */
){
    int     lag, n;
    float   sum;

    for (lag = 0; lag <= order; lag++) {
        sum = 0;
        for (n = 0; n < N - lag; n++) {
            sum += x[n] * x[n+lag];
        }
        r[lag] = sum;
    }
```

```
}
```



```
/*----------------------------------------------------------------*
 *  window multiplication
 *---------------------------------------------------------------*/

void window(
    float *z,       /* (o) the windowed data */
    const float *x, /* (i) the original data vector */
    const float *y, /* (i) the window */
    int N           /* (i) length of all vectors */
){
    int     i;

    for (i = 0; i < N; i++) {
        z[i] = x[i] * y[i];
    }
}

/*----------------------------------------------------------------*
 *  levinson-durbin solution for lpc coefficients
 *---------------------------------------------------------------*/

void levdurb(
    float *a,       /* (o) lpc coefficient vector starting
                           with 1.0 */
    float *k,       /* (o) reflection coefficients */
    float *r,       /* (i) autocorrelation vector */
    int order       /* (i) order of lpc filter */
){
    float  sum, alpha;
    int     m, m_h, i;

    a[0] = 1.0;

    if (r[0] < EPS) { /* if r[0] <= 0, set LPC coeff. to zero */
        for (i = 0; i < order; i++) {
            k[i] = 0;
            a[i+1] = 0;
        }
    } else {
        a[1] = k[0] = -r[1]/r[0];
        alpha = r[0] + r[1] * k[0];
        for (m = 1; m < order; m++){
            sum = r[m + 1];
            for (i = 0; i < m; i++){
                sum += a[i+1] * r[m - i];
```

```
            }
            k[m] = -sum / alpha;
            alpha += k[m] * sum;
            m_h = (m + 1) >> 1;
            for (i = 0; i < m_h; i++){
                sum = a[i+1] + k[m] * a[m - i];
                a[m - i] += k[m] * a[i+1];
                a[i+1] = sum;
```



```
            }
            a[m+1] = k[m];
        }
    }
}

/*----------------------------------------------------------------*
 *  interpolation between vectors
 *---------------------------------------------------------------*/

void interpolate(
    float *out,      /* (o) the interpolated vector */
    float *in1,     /* (i) the first vector for the
                           interpolation */
    float *in2,     /* (i) the second vector for the
                           interpolation */
    float coef,      /* (i) interpolation weights */
    int length      /* (i) length of all vectors */
){
    int i;
    float invcoef;

    invcoef = (float)1.0 - coef;
    for (i = 0; i < length; i++) {
        out[i] = coef * in1[i] + invcoef * in2[i];
    }
}

/*----------------------------------------------------------------*
 *  lpc bandwidth expansion
 *---------------------------------------------------------------*/

void bwexpand(
    float *out,      /* (o) the bandwidth expanded lpc
                           coefficients */
    float *in,      /* (i) the lpc coefficients before bandwidth
                           expansion */
    float coef,     /* (i) the bandwidth expansion factor */
    int length      /* (i) the length of lpc coefficient vectors */
```

```
){
    int i;
    float  chirp;

    chirp = coef;

    out[0] = in[0];
    for (i = 1; i < length; i++) {
        out[i] = chirp * in[i];
        chirp *= coef;
    }
}

/*----------------------------------------------------------------*
 *  vector quantization
```



```
 *---------------------------------------------------------------*/

void vq(
    float *Xq,      /* (o) the quantized vector */
    int *index,     /* (o) the quantization index */
    const float *CB,/* (i) the vector quantization codebook */
    float *X,       /* (i) the vector to quantize */
    int n_cb,       /* (i) the number of vectors in the codebook */
    int dim         /* (i) the dimension of all vectors */
){
    int     i, j;
    int     pos, minindex;
    float   dist, tmp, mindist;

    pos = 0;
    mindist = FLOAT_MAX;
    minindex = 0;
    for (j = 0; j < n_cb; j++) {
        dist = X[0] - CB[pos];
        dist *= dist;
        for (i = 1; i < dim; i++) {
            tmp = X[i] - CB[pos + i];
            dist += tmp*tmp;
        }

        if (dist < mindist) {
            mindist = dist;
            minindex = j;
        }
        pos += dim;
    }
    for (i = 0; i < dim; i++) {
```

```
            Xq[i] = CB[minindex*dim + i];
        }
        *index = minindex;
    }

    /*----------------------------------------------------------------*
     *  split vector quantization
     *---------------------------------------------------------------*/

    void SplitVQ(
        float *qX,      /* (o) the quantized vector */
        int *index,     /* (o) a vector of indexes for all vector
                               codebooks in the split */
        float *X,       /* (i) the vector to quantize */
        const float *CB,/* (i) the quantizer codebook */
        int nsplit,     /* the number of vector splits */
        const int *dim, /* the dimension of X and qX */
        const int *cbsize /* the number of vectors in the codebook */
    ){
        int    cb_pos, X_pos, i;

        cb_pos = 0;
```



```
        X_pos= 0;
        for (i = 0; i < nsplit; i++) {
            vq(qX + X_pos, index + i, CB + cb_pos, X + X_pos,
                cbsize[i], dim[i]);
            X_pos += dim[i];
            cb_pos += dim[i] * cbsize[i];
        }
    }

    /*----------------------------------------------------------------*
     *  scalar quantization
     *---------------------------------------------------------------*/

    void sort_sq(
        float *xq,      /* (o) the quantized value */
        int *index,     /* (o) the quantization index */
        float x,    /* (i) the value to quantize */
        const float *cb,/* (i) the quantization codebook */
        int cb_size      /* (i) the size of the quantization codebook */
    ){
        int i;

        if (x <= cb[0]) {
            *index = 0;
            *xq = cb[0];
```

```
    } else {
        i = 0;
        while ((x > cb[i]) && i < cb_size - 1) {
            i++;
        }

        if (x > ((cb[i] + cb[i - 1])/2)) {
            *index = i;
            *xq = cb[i];
        } else {
            *index = i - 1;
            *xq = cb[i - 1];
        }
    }
}

/*----------------------------------------------------------------*
 *  check for stability of lsf coefficients
 *---------------------------------------------------------------*/

int LSF_check(    /* (o) 1 for stable lsf vectors and 0 for
                           nonstable ones */
    float *lsf,     /* (i) a table of lsf vectors */
    int dim,    /* (i) the dimension of each lsf vector */
    int NoAn    /* (i) the number of lsf vectors in the
                           table */
){
    int k,n,m, Nit=2, change=0,pos;
    float tmp;
```



```
    static float eps=(float)0.039; /* 50 Hz */
    static float eps2=(float)0.0195;
    static float maxlsf=(float)3.14; /* 4000 Hz */
    static float minlsf=(float)0.01; /* 0 Hz */

    /* LSF separation check*/

    for (n=0; n<Nit; n++) { /* Run through a couple of times */
        for (m=0; m<NoAn; m++) { /* Number of analyses per frame */
            for (k=0; k<(dim-1); k++) {
                pos=m*dim+k;

                if ((lsf[pos+1]-lsf[pos])<eps) {

                    if (lsf[pos+1]<lsf[pos]) {
                        tmp=lsf[pos+1];
                        lsf[pos+1]= lsf[pos]+eps2;
                        lsf[pos]= lsf[pos+1]-eps2;
```

```
                    } else {
                        lsf[pos]-=eps2;
                        lsf[pos+1]+=eps2;
                    }
                    change=1;
                }

                if (lsf[pos]<minlsf) {
                    lsf[pos]=minlsf;
                    change=1;
                }

                if (lsf[pos]>maxlsf) {
                    lsf[pos]=maxlsf;
                    change=1;
                }
            }
        }
    }

    return change;
}
```

A.27 hpInput.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    hpInput.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/
```



```
#ifndef __iLBC_HPINPUT_H
#define __iLBC_HPINPUT_H

void hpInput(
    float *In,  /* (i) vector to filter */
    int len,    /* (i) length of vector to filter */
    float *Out, /* (o) the resulting filtered vector */
    float *mem  /* (i/o) the filter state */
);

#endif
```

A.28 hpInput.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    hpInput.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include "constants.h"

/*----------------------------------------------------------------*
 *  Input high-pass filter
 *---------------------------------------------------------------*/

void hpInput(
    float *In,  /* (i) vector to filter */
    int len,    /* (i) length of vector to filter */
    float *Out, /* (o) the resulting filtered vector */
    float *mem  /* (i/o) the filter state */
){
    int i;
    float *pi, *po;

    /* all-zero section*/

    pi = &In[0];
    po = &Out[0];
    for (i=0; i<len; i++) {
        *po = hpi_zero_coefsTbl[0] * (*pi);
        *po += hpi_zero_coefsTbl[1] * mem[0];
        *po += hpi_zero_coefsTbl[2] * mem[1];

        mem[1] = mem[0];
        mem[0] = *pi;
        po++;
```



```
        pi++;

    }

    /* all-pole section*/
```

```
    po = &Out[0];
    for (i=0; i<len; i++) {
        *po -= hpi_pole_coefsTbl[1] * mem[2];
        *po -= hpi_pole_coefsTbl[2] * mem[3];

        mem[3] = mem[2];
        mem[2] = *po;
        po++;
    }
}
```

A.29 hpOutput.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    hpOutput.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_HPOUTPUT_H
#define __iLBC_HPOUTPUT_H

void hpOutput(
    float *In,  /* (i) vector to filter */
    int len,/* (i) length of vector to filter */
    float *Out, /* (o) the resulting filtered vector */
    float *mem  /* (i/o) the filter state */
);

#endif
```

A.30 hpOutput.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    hpOutput.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.
```

```
*******************************************************************/

#include "constants.h"

/*----------------------------------------------------------------*
 *  Output high-pass filter
 *---------------------------------------------------------------*/

void hpOutput(
    float *In,  /* (i) vector to filter */
    int len,/* (i) length of vector to filter */
    float *Out, /* (o) the resulting filtered vector */
    float *mem  /* (i/o) the filter state */
){
    int i;
    float *pi, *po;

    /* all-zero section*/

    pi = &In[0];
    po = &Out[0];
    for (i=0; i<len; i++) {
        *po = hpo_zero_coefsTbl[0] * (*pi);
        *po += hpo_zero_coefsTbl[1] * mem[0];
        *po += hpo_zero_coefsTbl[2] * mem[1];

        mem[1] = mem[0];
        mem[0] = *pi;
        po++;
        pi++;

    }

    /* all-pole section*/

    po = &Out[0];
    for (i=0; i<len; i++) {
        *po -= hpo_pole_coefsTbl[1] * mem[2];
        *po -= hpo_pole_coefsTbl[2] * mem[3];

        mem[3] = mem[2];
        mem[2] = *po;
        po++;
    }
}
```

A.31 iCBConstruct.h

```
/******************************************************************
```

```
    iLBC Speech Coder ANSI-C Source Code
```



```
    iCBConstruct.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_ICBCONSTRUCT_H
#define __iLBC_ICBCONSTRUCT_H

void index_conv_enc(
    int *index          /* (i/o) Codebook indexes */
);

void index_conv_dec(
    int *index          /* (i/o) Codebook indexes */
);

void iCBConstruct(
    float *decvector,   /* (o) Decoded vector */
    int *index,         /* (i) Codebook indices */
    int *gain_index,/* (i) Gain quantization indices */
    float *mem,         /* (i) Buffer for codevector construction */
    int lMem,           /* (i) Length of buffer */
    int veclen,         /* (i) Length of vector */
    int nStages         /* (i) Number of codebook stages */
);

#endif
```

A.32 iCBConstruct.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    iCBConstruct.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <math.h>
```

```
#include "iLBC_define.h"
#include "gainquant.h"
#include "getCBvec.h"

/*----------------------------------------------------------------*
 *  Convert the codebook indexes to make the search easier
 *---------------------------------------------------------------*/
```



```
void index_conv_enc(
    int *index          /* (i/o) Codebook indexes */
){
    int k;

    for (k=1; k<CB_NSTAGES; k++) {

        if ((index[k]>=108)&&(index[k]<172)) {
            index[k]-=64;
        } else if (index[k]>=236) {
            index[k]-=128;
        } else {
            /* ERROR */
        }
    }
}

void index_conv_dec(
    int *index          /* (i/o) Codebook indexes */
){
    int k;

    for (k=1; k<CB_NSTAGES; k++) {

        if ((index[k]>=44)&&(index[k]<108)) {
            index[k]+=64;
        } else if ((index[k]>=108)&&(index[k]<128)) {
            index[k]+=128;
        } else {
            /* ERROR */
        }
    }
}

/*----------------------------------------------------------------*
 *  Construct decoded vector from codebook and gains.
 *---------------------------------------------------------------*/

void iCBConstruct(
    float *decvector,   /* (o) Decoded vector */
```

```
    int *index,         /* (i) Codebook indices */
    int *gain_index,/* (i) Gain quantization indices */
    float *mem,         /* (i) Buffer for codevector construction */
    int lMem,           /* (i) Length of buffer */
    int veclen,         /* (i) Length of vector */
    int nStages         /* (i) Number of codebook stages */
){
    int j,k;
    float gain[CB_NSTAGES];
    float cbvec[SUBL];

    /* gain de-quantization */

    gain[0] = gaindequant(gain_index[0], 1.0, 32);
```



```
    if (nStages > 1) {
        gain[1] = gaindequant(gain_index[1],
            (float)fabs(gain[0]), 16);
    }
    if (nStages > 2) {
        gain[2] = gaindequant(gain_index[2],
            (float)fabs(gain[1]), 8);
    }

    /* codebook vector construction and construction of
    total vector */

    getCBvec(cbvec, mem, index[0], lMem, veclen);
    for (j=0;j<veclen;j++){
        decvector[j] = gain[0]*cbvec[j];
    }
    if (nStages > 1) {
        for (k=1; k<nStages; k++) {
            getCBvec(cbvec, mem, index[k], lMem, veclen);
            for (j=0;j<veclen;j++) {
                decvector[j] += gain[k]*cbvec[j];
            }
        }
    }
}
```

A.33 iCBSearch.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    iCBSearch.h
```

```
******************************************************************/

#ifndef __iLBC_ICBSEARCH_H
#define __iLBC_ICBSEARCH_H

void iCBSearch(
    iLBC_Enc_Inst_t *iLBCenc_inst,
                        /* (i) the encoder state structure */
    int *index,         /* (o) Codebook indices */
    int *gain_index,/* (o) Gain quantization indices */
    float *intarget,/* (i) Target vector for encoding */
    float *mem,         /* (i) Buffer for codebook construction */
    int lMem,           /* (i) Length of buffer */
    int lTarget,    /* (i) Length of vector */
    int nStages,    /* (i) Number of codebook stages */
    float *weightDenum, /* (i) weighting filter coefficients */
```



```
    float *weightState, /* (i) weighting filter state */
    int block           /* (i) the sub-block number */
);

#endif
```

A.34 iCBSearch.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    iCBSearch.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <math.h>
#include <string.h>

#include "iLBC_define.h"
#include "gainquant.h"
#include "createCB.h"
#include "filter.h"
#include "constants.h"
```

```
/*----------------------------------------------------------------*
 *  Search routine for codebook encoding and gain quantization.
 *----------------------------------------------------------------*/

void iCBSearch(
    iLBC_Enc_Inst_t *iLBCenc_inst,
                        /* (i) the encoder state structure */
    int *index,         /* (o) Codebook indices */
    int *gain_index,/* (o) Gain quantization indices */
    float *intarget,/* (i) Target vector for encoding */
    float *mem,         /* (i) Buffer for codebook construction */
    int lMem,           /* (i) Length of buffer */
    int lTarget,    /* (i) Length of vector */
    int nStages,    /* (i) Number of codebook stages */
    float *weightDenum, /* (i) weighting filter coefficients */
    float *weightState, /* (i) weighting filter state */
    int block           /* (i) the sub-block number */
){
    int i, j, icount, stage, best_index, range, counter;
    float max_measure, gain, measure, crossDot, ftmp;
    float gains[CB_NSTAGES];
    float target[SUBL];
    int base_index, sInd, eInd, base_size;
    int sIndAug=0, eIndAug=0;
    float buf[CB_MEML+SUBL+2*LPC_FILTERORDER];
```



```
    float invenergy[CB_EXPAND*128], energy[CB_EXPAND*128];
    float *pp, *ppi=0, *ppo=0, *ppe=0;
    float cbvectors[CB_MEML];
    float tene, cene, cvec[SUBL];
    float aug_vec[SUBL];

    memset(cvec,0,SUBL*sizeof(float));

    /* Determine size of codebook sections */

    base_size=lMem-lTarget+1;

    if (lTarget==SUBL) {
        base_size=lMem-lTarget+1+lTarget/2;
    }

    /* setup buffer for weighting */

    memcpy(buf,weightState,sizeof(float)*LPC_FILTERORDER);
    memcpy(buf+LPC_FILTERORDER,mem,lMem*sizeof(float));
    memcpy(buf+LPC_FILTERORDER+lMem,intarget,lTarget*sizeof(float));
```

```
    /* weighting */

    AllPoleFilter(buf+LPC_FILTERORDER, weightDenum,
        lMem+lTarget, LPC_FILTERORDER);

    /* Construct the codebook and target needed */

    memcpy(target, buf+LPC_FILTERORDER+lMem, lTarget*sizeof(float));

    tene=0.0;
    for (i=0; i<lTarget; i++) {
        tene+=target[i]*target[i];
    }

    /* Prepare search over one more codebook section. This section
       is created by filtering the original buffer with a filter. */

    filteredCBvecs(cbvectors, buf+LPC_FILTERORDER, lMem);

    /* The Main Loop over stages */

    for (stage=0; stage<nStages; stage++) {

        range = search_rangeTbl[block][stage];

        /* initialize search measure */

        max_measure = (float)-10000000.0;
        gain = (float)0.0;
        best_index = 0;

        /* Compute cross dot product between the target
```



```
           and the CB memory */

        crossDot=0.0;
        pp=buf+LPC_FILTERORDER+lMem-lTarget;
        for (j=0; j<lTarget; j++) {
            crossDot += target[j]*(*pp++);
        }

        if (stage==0) {

            /* Calculate energy in the first block of
              'lTarget' sampels. */
            ppe = energy;
            ppi = buf+LPC_FILTERORDER+lMem-lTarget-1;
            ppo = buf+LPC_FILTERORDER+lMem-1;
```

```
            *ppe=0.0;
            pp=buf+LPC_FILTERORDER+lMem-lTarget;
            for (j=0; j<lTarget; j++) {
                *ppe+=(*pp)*(*pp++);
            }

            if (*ppe>0.0) {
                invenergy[0] = (float) 1.0 / (*ppe + EPS);
            } else {
                invenergy[0] = (float) 0.0;
            }
            ppe++;

            measure=(float)-10000000.0;

            if (crossDot > 0.0) {
                   measure = crossDot*crossDot*invenergy[0];
            }
        }
        else {
            measure = crossDot*crossDot*invenergy[0];
        }

        /* check if measure is better */
        ftmp = crossDot*invenergy[0];

        if ((measure>max_measure) && (fabs(ftmp)<CB_MAXGAIN)) {
            best_index = 0;
            max_measure = measure;
            gain = ftmp;
        }

        /* loop over the main first codebook section,
           full search */

        for (icount=1; icount<range; icount++) {

            /* calculate measure */
```



```
            crossDot=0.0;
            pp = buf+LPC_FILTERORDER+lMem-lTarget-icount;

            for (j=0; j<lTarget; j++) {
                crossDot += target[j]*(*pp++);
            }

            if (stage==0) {
                *ppe++ = energy[icount-1] + (*ppi)*(*ppi) -
```

```
                (*ppo)*(*ppo);
            ppo--;
            ppi--;

            if (energy[icount]>0.0) {
                invenergy[icount] =
                    (float)1.0/(energy[icount]+EPS);
            } else {
                invenergy[icount] = (float) 0.0;
            }

            measure=(float)-10000000.0;

            if (crossDot > 0.0) {
                measure = crossDot*crossDot*invenergy[icount];
            }
        }
        else {
            measure = crossDot*crossDot*invenergy[icount];
        }

        /* check if measure is better */
        ftmp = crossDot*invenergy[icount];

        if ((measure>max_measure) && (fabs(ftmp)<CB_MAXGAIN)) {
            best_index = icount;
            max_measure = measure;
            gain = ftmp;
        }
    }

    /* Loop over augmented part in the first codebook
     * section, full search.
     * The vectors are interpolated.
     */

    if (lTarget==SUBL) {

        /* Search for best possible cb vector and
           compute the CB-vectors' energy. */
        searchAugmentedCB(20, 39, stage, base_size-lTarget/2,
            target, buf+LPC_FILTERORDER+lMem,
            &max_measure, &best_index, &gain, energy,
            invenergy);
```



```
    }

    /* set search range for following codebook sections */
```

```
       base_index=best_index;

       /* unrestricted search */

       if (CB_RESRANGE == -1) {
           sInd=0;
           eInd=range-1;
           sIndAug=20;
           eIndAug=39;
       }

       /* restriced search around best index from first
       codebook section */

       else {
           /* Initialize search indices */
           sIndAug=0;
           eIndAug=0;
           sInd=base_index-CB_RESRANGE/2;
           eInd=sInd+CB_RESRANGE;

           if (lTarget==SUBL) {

               if (sInd<0) {

                   sIndAug = 40 + sInd;
                   eIndAug = 39;
                   sInd=0;

               } else if ( base_index < (base_size-20) ) {

                   if (eInd > range) {
                       sInd -= (eInd-range);
                       eInd = range;
                   }
               } else { /* base_index >= (base_size-20) */

                   if (sInd < (base_size-20)) {
                       sIndAug = 20;
                       sInd = 0;
                       eInd = 0;
                       eIndAug = 19 + CB_RESRANGE;

                       if(eIndAug > 39) {
                           eInd = eIndAug-39;
                           eIndAug = 39;
                       }
                   } else {
                       sIndAug = 20 + sInd - (base_size-20);
                       eIndAug = 39;
```

```
                sInd = 0;
                eInd = CB_RESRANGE - (eIndAug-sIndAug+1);
            }
        }

    } else { /* lTarget = 22 or 23 */

        if (sInd < 0) {
            eInd -= sInd;
            sInd = 0;
        }

        if(eInd > range) {
            sInd -= (eInd - range);
            eInd = range;
        }
    }
}

/* search of higher codebook section */

/* index search range */
counter = sInd;
sInd += base_size;
eInd += base_size;


if (stage==0) {
    ppe = energy+base_size;
    *ppe=0.0;

    pp=cbvectors+lMem-lTarget;
    for (j=0; j<lTarget; j++) {
        *ppe+=(*pp)*(*pp++);
    }

    ppi = cbvectors + lMem - 1 - lTarget;
    ppo = cbvectors + lMem - 1;

    for (j=0; j<(range-1); j++) {
        *(ppe+1) = *ppe + (*ppi)*(*ppi) - (*ppo)*(*ppo);
        ppo--;
        ppi--;
        ppe++;
    }
}

/* loop over search range */

for (icount=sInd; icount<eInd; icount++) {
```

```
            /* calculate measure */

            crossDot=0.0;
```



```
            pp=cbvectors + lMem - (counter++) - lTarget;

            for (j=0;j<lTarget;j++) {
                crossDot += target[j]*(*pp++);
            }

            if (energy[icount]>0.0) {
                invenergy[icount] =(float)1.0/(energy[icount]+EPS);
            } else {
                invenergy[icount] =(float)0.0;
            }

            if (stage==0) {

                measure=(float)-10000000.0;

                if (crossDot > 0.0) {
                    measure = crossDot*crossDot*
                        invenergy[icount];
                }
            }
            else {
                measure = crossDot*crossDot*invenergy[icount];
            }

            /* check if measure is better */
            ftmp = crossDot*invenergy[icount];

            if ((measure>max_measure) && (fabs(ftmp)<CB_MAXGAIN)) {
                best_index = icount;
                max_measure = measure;
                gain = ftmp;
            }
        }

        /* Search the augmented CB inside the limited range. */

        if ((lTarget==SUBL)&&(sIndAug!=0)) {
            searchAugmentedCB(sIndAug, eIndAug, stage,
                2*base_size-20, target, cbvectors+lMem,
                &max_measure, &best_index, &gain, energy,
                invenergy);
        }
```

```
        /* record best index */

        index[stage] = best_index;

        /* gain quantization */

        if (stage==0){

            if (gain<0.0){
                gain = 0.0;
```



```
            }

            if (gain>CB_MAXGAIN) {
                gain = (float)CB_MAXGAIN;
            }
            gain = gainquant(gain, 1.0, 32, &gain_index[stage]);
        }
        else {
            if (stage==1) {
                gain = gainquant(gain, (float)fabs(gains[stage-1]),
                    16, &gain_index[stage]);
            } else {
                gain = gainquant(gain, (float)fabs(gains[stage-1]),
                    8, &gain_index[stage]);
            }
        }

        /* Extract the best (according to measure)
           codebook vector */

        if (lTarget==(STATE_LEN-iLBCenc_inst->state_short_len)) {

            if (index[stage]<base_size) {
                pp=buf+LPC_FILTERORDER+lMem-lTarget-index[stage];
            } else {
                pp=cbvectors+lMem-lTarget-
                    index[stage]+base_size;
            }
        } else {

            if (index[stage]<base_size) {
                if (index[stage]<(base_size-20)) {
                    pp=buf+LPC_FILTERORDER+lMem-
                        lTarget-index[stage];
                } else {
                    createAugmentedVec(index[stage]-base_size+40,
                            buf+LPC_FILTERORDER+lMem,aug_vec);
                    pp=aug_vec;
```

```
                }
            } else {
                int filterno, position;

                filterno=index[stage]/base_size;
                position=index[stage]-filterno*base_size;


                if (position<(base_size-20)) {
                    pp=cbvectors+filterno*lMem-lTarget-
                        index[stage]+filterno*base_size;
                } else {
                    createAugmentedVec(
                        index[stage]-(filterno+1)*base_size+40,
                        cbvectors+filterno*lMem,aug_vec);
                    pp=aug_vec;
```



```
                }
            }
        }

        /* Subtract the best codebook vector, according
           to measure, from the target vector */

        for (j=0;j<lTarget;j++) {
            cvec[j] += gain*(*pp);
            target[j] -= gain*(*pp++);
        }

        /* record quantized gain */

        gains[stage]=gain;

    }/* end of Main Loop. for (stage=0;... */

    /* Gain adjustment for energy matching */
    cene=0.0;
    for (i=0; i<lTarget; i++) {
        cene+=cvec[i]*cvec[i];
    }
    j=gain_index[0];

    for (i=gain_index[0]; i<32; i++) {
        ftmp=cene*gain_sq5Tbl[i]*gain_sq5Tbl[i];

        if ((ftmp<(tene*gains[0]*gains[0])) &&
            (gain_sq5Tbl[j]<(2.0*gains[0]))) {
            j=i;
        }
```

```
       }
       gain_index[0]=j;
   }
```

A.35 LPCdecode.h

```
   /******************************************************************

       iLBC Speech Coder ANSI-C Source Code

       LPC_decode.h

       Copyright (C) The Internet Society (2004).
       All Rights Reserved.

   ******************************************************************/

   #ifndef __iLBC_LPC_DECODE_H
   #define __iLBC_LPC_DECODE_H

   void LSFinterpolate2a_dec(
```



```
       float *a,           /* (o) lpc coefficients for a sub-frame */
       float *lsf1,    /* (i) first lsf coefficient vector */
       float *lsf2,    /* (i) second lsf coefficient vector */
       float coef,         /* (i) interpolation weight */
       int length          /* (i) length of lsf vectors */
   );

   void SimplelsfDEQ(
       float *lsfdeq,      /* (o) dequantized lsf coefficients */
       int *index,         /* (i) quantization index */
       int lpc_n           /* (i) number of LPCs */
   );

   void DecoderInterpolateLSF(
       float *syntdenum,   /* (o) synthesis filter coefficients */
       float *weightdenum, /* (o) weighting denumerator
                                  coefficients */
       float *lsfdeq,      /* (i) dequantized lsf coefficients */
       int length,         /* (i) length of lsf coefficient vector */
       iLBC_Dec_Inst_t *iLBCdec_inst
                           /* (i) the decoder state structure */
   );

   #endif
```

A.36 LPCdecode.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    LPC_decode.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <math.h>
#include <string.h>

#include "helpfun.h"
#include "lsf.h"
#include "iLBC_define.h"
#include "constants.h"

/*---------------------------------------------------------------*
 *  interpolation of lsf coefficients for the decoder
 *--------------------------------------------------------------*/

void LSFinterpolate2a_dec(
    float *a,           /* (o) lpc coefficients for a sub-frame */
    float *lsf1,    /* (i) first lsf coefficient vector */
```



```
    float *lsf2,    /* (i) second lsf coefficient vector */
    float coef,         /* (i) interpolation weight */
    int length          /* (i) length of lsf vectors */
){
    float  lsftmp[LPC_FILTERORDER];

    interpolate(lsftmp, lsf1, lsf2, coef, length);
    lsf2a(a, lsftmp);
}

/*---------------------------------------------------------------*
 *  obtain dequantized lsf coefficients from quantization index
 *--------------------------------------------------------------*/

void SimplelsfDEQ(
    float *lsfdeq,    /* (o) dequantized lsf coefficients */
    int *index,         /* (i) quantization index */
    int lpc_n           /* (i) number of LPCs */
){
    int i, j, pos, cb_pos;
```

```
    /* decode first LSF */

    pos = 0;
    cb_pos = 0;
    for (i = 0; i < LSF_NSPLIT; i++) {
        for (j = 0; j < dim_lsfCbTbl[i]; j++) {
            lsfdeq[pos + j] = lsfCbTbl[cb_pos +
                (long)(index[i])*dim_lsfCbTbl[i] + j];
        }
        pos += dim_lsfCbTbl[i];
        cb_pos += size_lsfCbTbl[i]*dim_lsfCbTbl[i];
    }

    if (lpc_n>1) {

        /* decode last LSF */

        pos = 0;
        cb_pos = 0;
        for (i = 0; i < LSF_NSPLIT; i++) {
            for (j = 0; j < dim_lsfCbTbl[i]; j++) {
                lsfdeq[LPC_FILTERORDER + pos + j] =
                    lsfCbTbl[cb_pos +
                    (long)(index[LSF_NSPLIT + i])*
                    dim_lsfCbTbl[i] + j];
            }
            pos += dim_lsfCbTbl[i];
            cb_pos += size_lsfCbTbl[i]*dim_lsfCbTbl[i];
        }
    }
}

/*----------------------------------------------------------------*
```



```
 *  obtain synthesis and weighting filters form lsf coefficients
 *---------------------------------------------------------------*/

void DecoderInterpolateLSF(
    float *syntdenum, /* (o) synthesis filter coefficients */
    float *weightdenum, /* (o) weighting denumerator
                               coefficients */
    float *lsfdeq,       /* (i) dequantized lsf coefficients */
    int length,         /* (i) length of lsf coefficient vector */
    iLBC_Dec_Inst_t *iLBCdec_inst
                        /* (i) the decoder state structure */
){
    int    i, pos, lp_length;
```

```
    float  lp[LPC_FILTERORDER + 1], *lsfdeq2;

    lsfdeq2 = lsfdeq + length;
    lp_length = length + 1;

    if (iLBCdec_inst->mode==30) {
        /* sub-frame 1: Interpolation between old and first */

        LSFinterpolate2a_dec(lp, iLBCdec_inst->lsfdeqold, lsfdeq,
            lsf_weightTbl_30ms[0], length);
        memcpy(syntdenum,lp,lp_length*sizeof(float));
        bwexpand(weightdenum, lp, LPC_CHIRP_WEIGHTDENUM,
            lp_length);

        /* sub-frames 2 to 6: interpolation between first
           and last LSF */

        pos = lp_length;
        for (i = 1; i < 6; i++) {
            LSFinterpolate2a_dec(lp, lsfdeq, lsfdeq2,
                lsf_weightTbl_30ms[i], length);
            memcpy(syntdenum + pos,lp,lp_length*sizeof(float));
            bwexpand(weightdenum + pos, lp,
                LPC_CHIRP_WEIGHTDENUM, lp_length);
            pos += lp_length;
        }
    }
    else {
        pos = 0;
        for (i = 0; i < iLBCdec_inst->nsub; i++) {
            LSFinterpolate2a_dec(lp, iLBCdec_inst->lsfdeqold,
                lsfdeq, lsf_weightTbl_20ms[i], length);
            memcpy(syntdenum+pos,lp,lp_length*sizeof(float));
            bwexpand(weightdenum+pos, lp, LPC_CHIRP_WEIGHTDENUM,
                lp_length);
            pos += lp_length;
        }
    }

    /* update memory */
```



```
    if (iLBCdec_inst->mode==30)
        memcpy(iLBCdec_inst->lsfdeqold, lsfdeq2,
                    length*sizeof(float));
    else
        memcpy(iLBCdec_inst->lsfdeqold, lsfdeq,
                    length*sizeof(float));
```

```
}
```

A.37 LPCencode.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    LPCencode.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_LPCENCOD_H
#define __iLBC_LPCENCOD_H

void LPCencode(
    float *syntdenum,   /* (i/o) synthesis filter coefficients
                               before/after encoding */
    float *weightdenum, /* (i/o) weighting denumerator coefficients
                               before/after encoding */
    int *lsf_index,     /* (o) lsf quantization index */
    float *data,    /* (i) lsf coefficients to quantize */
    iLBC_Enc_Inst_t *iLBCenc_inst
                        /* (i/o) the encoder state structure */
);

#endif
```

A.38 LPCencode.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    LPCencode.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <string.h>
```



```
#include "iLBC_define.h"
```

```
#include "helpfun.h"
#include "lsf.h"
#include "constants.h"

/*----------------------------------------------------------------*
 *  lpc analysis (subrutine to LPCencode)
 *---------------------------------------------------------------*/

void SimpleAnalysis(
    float *lsf,         /* (o) lsf coefficients */
    float *data,    /* (i) new data vector */
    iLBC_Enc_Inst_t *iLBCenc_inst
                        /* (i/o) the encoder state structure */
){
    int k, is;
    float temp[BLOCKL_MAX], lp[LPC_FILTERORDER + 1];
    float lp2[LPC_FILTERORDER + 1];
    float r[LPC_FILTERORDER + 1];

    is=LPC_LOOKBACK+BLOCKL_MAX-iLBCenc_inst->blockl;
    memcpy(iLBCenc_inst->lpc_buffer+is,data,
        iLBCenc_inst->blockl*sizeof(float));

    /* No lookahead, last window is asymmetric */

    for (k = 0; k < iLBCenc_inst->lpc_n; k++) {

        is = LPC_LOOKBACK;

        if (k < (iLBCenc_inst->lpc_n - 1)) {
            window(temp, lpc_winTbl,
                iLBCenc_inst->lpc_buffer, BLOCKL_MAX);
        } else {
            window(temp, lpc_asymwinTbl,
                iLBCenc_inst->lpc_buffer + is, BLOCKL_MAX);
        }

        autocorr(r, temp, BLOCKL_MAX, LPC_FILTERORDER);
        window(r, r, lpc_lagwinTbl, LPC_FILTERORDER + 1);

        levdurb(lp, temp, r, LPC_FILTERORDER);
        bwexpand(lp2, lp, LPC_CHIRP_SYNTDENUM, LPC_FILTERORDER+1);

        a2lsf(lsf + k*LPC_FILTERORDER, lp2);
    }
    is=LPC_LOOKBACK+BLOCKL_MAX-iLBCenc_inst->blockl;
    memmove(iLBCenc_inst->lpc_buffer,
        iLBCenc_inst->lpc_buffer+LPC_LOOKBACK+BLOCKL_MAX-is,
        is*sizeof(float));
}

/*----------------------------------------------------------------*
```

```
 *  lsf interpolator and conversion from lsf to a coefficients
 *  (subrutine to SimpleInterpolateLSF)
 *---------------------------------------------------------------*/

void LSFinterpolate2a_enc(
    float *a,       /* (o) lpc coefficients */
    float *lsf1,/* (i) first set of lsf coefficients */
    float *lsf2,/* (i) second set of lsf coefficients */
    float coef,     /* (i) weighting coefficient to use between
                           lsf1 and lsf2 */
    long length      /* (i) length of coefficient vectors */
){
    float  lsftmp[LPC_FILTERORDER];

    interpolate(lsftmp, lsf1, lsf2, coef, length);
    lsf2a(a, lsftmp);
}

/*----------------------------------------------------------------*
 *  lsf interpolator (subrutine to LPCencode)
 *---------------------------------------------------------------*/

void SimpleInterpolateLSF(
    float *syntdenum,   /* (o) the synthesis filter denominator
                               resulting from the quantized
                               interpolated lsf */
    float *weightdenum, /* (o) the weighting filter denominator
                               resulting from the unquantized
                               interpolated lsf */
    float *lsf,         /* (i) the unquantized lsf coefficients */
    float *lsfdeq,      /* (i) the dequantized lsf coefficients */
    float *lsfold,      /* (i) the unquantized lsf coefficients of
                               the previous signal frame */
    float *lsfdeqold, /* (i) the dequantized lsf coefficients of
                               the previous signal frame */
    int length,         /* (i) should equate LPC_FILTERORDER */
    iLBC_Enc_Inst_t *iLBCenc_inst
                        /* (i/o) the encoder state structure */
){
    int    i, pos, lp_length;
    float  lp[LPC_FILTERORDER + 1], *lsf2, *lsfdeq2;

    lsf2 = lsf + length;
    lsfdeq2 = lsfdeq + length;
    lp_length = length + 1;

    if (iLBCenc_inst->mode==30) {
```

```
        /* sub-frame 1: Interpolation between old and first
           set of lsf coefficients */

        LSFinterpolate2a_enc(lp, lsfdeqold, lsfdeq,
            lsf_weightTbl_30ms[0], length);
        memcpy(syntdenum,lp,lp_length*sizeof(float));
        LSFinterpolate2a_enc(lp, lsfold, lsf,
```



```
            lsf_weightTbl_30ms[0], length);
        bwexpand(weightdenum, lp, LPC_CHIRP_WEIGHTDENUM, lp_length);

        /* sub-frame 2 to 6: Interpolation between first
           and second set of lsf coefficients */

        pos = lp_length;
        for (i = 1; i < iLBCenc_inst->nsub; i++) {
            LSFinterpolate2a_enc(lp, lsfdeq, lsfdeq2,
                lsf_weightTbl_30ms[i], length);
            memcpy(syntdenum + pos,lp,lp_length*sizeof(float));

            LSFinterpolate2a_enc(lp, lsf, lsf2,
                lsf_weightTbl_30ms[i], length);
            bwexpand(weightdenum + pos, lp,
                LPC_CHIRP_WEIGHTDENUM, lp_length);
            pos += lp_length;
        }
    }
    else {
        pos = 0;
        for (i = 0; i < iLBCenc_inst->nsub; i++) {
            LSFinterpolate2a_enc(lp, lsfdeqold, lsfdeq,
                lsf_weightTbl_20ms[i], length);
            memcpy(syntdenum+pos,lp,lp_length*sizeof(float));
            LSFinterpolate2a_enc(lp, lsfold, lsf,
                lsf_weightTbl_20ms[i], length);
            bwexpand(weightdenum+pos, lp,
                LPC_CHIRP_WEIGHTDENUM, lp_length);
            pos += lp_length;
        }
    }

    /* update memory */

    if (iLBCenc_inst->mode==30) {
        memcpy(lsfold, lsf2, length*sizeof(float));
        memcpy(lsfdeqold, lsfdeq2, length*sizeof(float));
    }
    else {
```

```
        memcpy(lsfold, lsf, length*sizeof(float));
        memcpy(lsfdeqold, lsfdeq, length*sizeof(float));
    }
}

/*----------------------------------------------------------------*
 *  lsf quantizer (subrutine to LPCencode)
 *---------------------------------------------------------------*/

void SimplelsfQ(
    float *lsfdeq,    /* (o) dequantized lsf coefficients
                         (dimension FILTERORDER) */
    int *index,     /* (o) quantization index */
    float *lsf,      /* (i) the lsf coefficient vector to be
```



```
                           quantized (dimension FILTERORDER ) */
    int lpc_n     /* (i) number of lsf sets to quantize */
){
    /* Quantize first LSF with memoryless split VQ */
    SplitVQ(lsfdeq, index, lsf, lsfCbTbl, LSF_NSPLIT,
        dim_lsfCbTbl, size_lsfCbTbl);

    if (lpc_n==2) {
        /* Quantize second LSF with memoryless split VQ */
        SplitVQ(lsfdeq + LPC_FILTERORDER, index + LSF_NSPLIT,
            lsf + LPC_FILTERORDER, lsfCbTbl, LSF_NSPLIT,
            dim_lsfCbTbl, size_lsfCbTbl);
    }
}

/*----------------------------------------------------------------*
 *  lpc encoder
 *---------------------------------------------------------------*/

void LPCencode(
    float *syntdenum, /* (i/o) synthesis filter coefficients
                           before/after encoding */
    float *weightdenum, /* (i/o) weighting denumerator
                           coefficients before/after
                           encoding */
    int *lsf_index,     /* (o) lsf quantization index */
    float *data,    /* (i) lsf coefficients to quantize */
    iLBC_Enc_Inst_t *iLBCenc_inst
                        /* (i/o) the encoder state structure */
){
    float lsf[LPC_FILTERORDER * LPC_N_MAX];
    float lsfdeq[LPC_FILTERORDER * LPC_N_MAX];
    int change=0;
```

```
    SimpleAnalysis(lsf, data, iLBCenc_inst);
    SimplelsfQ(lsfdeq, lsf_index, lsf, iLBCenc_inst->lpc_n);
    change=LSF_check(lsfdeq, LPC_FILTERORDER, iLBCenc_inst->lpc_n);
    SimpleInterpolateLSF(syntdenum, weightdenum,
        lsf, lsfdeq, iLBCenc_inst->lsfold,
        iLBCenc_inst->lsfdeqold, LPC_FILTERORDER, iLBCenc_inst);
}
```

A.39 lsf.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    lsf.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.
```



```
******************************************************************/

#ifndef __iLBC_LSF_H
#define __iLBC_LSF_H

void a2lsf(
    float *freq,/* (o) lsf coefficients */
    float *a    /* (i) lpc coefficients */
);

void lsf2a(
    float *a_coef,  /* (o) lpc coefficients */
    float *freq     /* (i) lsf coefficients */
);

#endif
```

A.40 lsf.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    lsf.c

    Copyright (C) The Internet Society (2004).
```

```
******************************************************************/

#include <string.h>
#include <math.h>

#include "iLBC_define.h"

/*----------------------------------------------------------------*
 *  conversion from lpc coefficients to lsf coefficients
 *---------------------------------------------------------------*/

void a2lsf(
    float *freq,/* (o) lsf coefficients */
    float *a    /* (i) lpc coefficients */
){
    float steps[LSF_NUMBER_OF_STEPS] =
        {(float)0.00635, (float)0.003175, (float)0.0015875,
        (float)0.00079375};
    float step;
    int step_idx;
    int lsp_index;
    float p[LPC_HALFORDER];
    float q[LPC_HALFORDER];
    float p_pre[LPC_HALFORDER];
```



```
    float q_pre[LPC_HALFORDER];
    float old_p, old_q, *old;
    float *pq_coef;
    float omega, old_omega;
    int i;
    float hlp, hlp1, hlp2, hlp3, hlp4, hlp5;

    for (i=0; i<LPC_HALFORDER; i++) {
        p[i] = (float)-1.0 * (a[i + 1] + a[LPC_FILTERORDER - i]);
        q[i] = a[LPC_FILTERORDER - i] - a[i + 1];
    }

    p_pre[0] = (float)-1.0 - p[0];
    p_pre[1] = - p_pre[0] - p[1];
    p_pre[2] = - p_pre[1] - p[2];
    p_pre[3] = - p_pre[2] - p[3];
    p_pre[4] = - p_pre[3] - p[4];
    p_pre[4] = p_pre[4] / 2;

    q_pre[0] = (float)1.0 - q[0];
    q_pre[1] = q_pre[0] - q[1];
    q_pre[2] = q_pre[1] - q[2];
```

```
      q_pre[3] = q_pre[2] - q[3];
      q_pre[4] = q_pre[3] - q[4];
      q_pre[4] = q_pre[4] / 2;

      omega = 0.0;
      old_omega = 0.0;

      old_p = FLOAT_MAX;
      old_q = FLOAT_MAX;

      /* Here we loop through lsp_index to find all the
         LPC_FILTERORDER roots for omega. */

      for (lsp_index = 0; lsp_index<LPC_FILTERORDER; lsp_index++) {

          /* Depending on lsp_index being even or odd, we
          alternatively solve the roots for the two LSP equations. */


          if ((lsp_index & 0x1) == 0) {
              pq_coef = p_pre;
              old = &old_p;
          } else {
              pq_coef = q_pre;
              old = &old_q;
          }

          /* Start with low resolution grid */

          for (step_idx = 0, step = steps[step_idx];
              step_idx < LSF_NUMBER_OF_STEPS;){
```



```
              /*  cos(10piw) + pq(0)cos(8piw) + pq(1)cos(6piw) +
              pq(2)cos(4piw) + pq(3)cod(2piw) + pq(4) */

              hlp = (float)cos(omega * TWO_PI);
              hlp1 = (float)2.0 * hlp + pq_coef[0];
              hlp2 = (float)2.0 * hlp * hlp1 - (float)1.0 +
                  pq_coef[1];
              hlp3 = (float)2.0 * hlp * hlp2 - hlp1 + pq_coef[2];
              hlp4 = (float)2.0 * hlp * hlp3 - hlp2 + pq_coef[3];
              hlp5 = hlp * hlp4 - hlp3 + pq_coef[4];


              if (((hlp5 * (*old)) <= 0.0) || (omega >= 0.5)){

                  if (step_idx == (LSF_NUMBER_OF_STEPS - 1)){
```

```
                    if (fabs(hlp5) >= fabs(*old)) {
                        freq[lsp_index] = omega - step;
                    } else {
                        freq[lsp_index] = omega;
                    }


                    if ((*old) >= 0.0){
                        *old = (float)-1.0 * FLOAT_MAX;
                    } else {
                        *old = FLOAT_MAX;
                    }

                    omega = old_omega;
                    step_idx = 0;

                    step_idx = LSF_NUMBER_OF_STEPS;
                } else {

                    if (step_idx == 0) {
                        old_omega = omega;
                    }

                    step_idx++;
                    omega -= steps[step_idx];

                    /* Go back one grid step */

                    step = steps[step_idx];
                }
            } else {

            /* increment omega until they are of different sign,
            and we know there is at least one root between omega
            and old_omega */
                *old = hlp5;
                omega += step;
            }
```



```
        }
    }

    for (i = 0; i<LPC_FILTERORDER; i++) {
        freq[i] = freq[i] * TWO_PI;
    }
}

/*----------------------------------------------------------------*
 *  conversion from lsf coefficients to lpc coefficients
```

```
 *---------------------------------------------------------------*/

void lsf2a(
    float *a_coef,  /* (o) lpc coefficients */
    float *freq     /* (i) lsf coefficients */
){
    int i, j;
    float hlp;
    float p[LPC_HALFORDER], q[LPC_HALFORDER];
    float a[LPC_HALFORDER + 1], a1[LPC_HALFORDER],
        a2[LPC_HALFORDER];
    float b[LPC_HALFORDER + 1], b1[LPC_HALFORDER],
        b2[LPC_HALFORDER];

    for (i=0; i<LPC_FILTERORDER; i++) {
        freq[i] = freq[i] * PI2;
    }

    /* Check input for ill-conditioned cases.  This part is not
    found in the TIA standard.  It involves the following 2 IF
    blocks. If "freq" is judged ill-conditioned, then we first
    modify freq[0] and freq[LPC_HALFORDER-1] (normally
    LPC_HALFORDER = 10 for LPC applications), then we adjust
    the other "freq" values slightly */


    if ((freq[0] <= 0.0) || (freq[LPC_FILTERORDER - 1] >= 0.5)){


        if (freq[0] <= 0.0) {
            freq[0] = (float)0.022;
        }


        if (freq[LPC_FILTERORDER - 1] >= 0.5) {
            freq[LPC_FILTERORDER - 1] = (float)0.499;
        }

        hlp = (freq[LPC_FILTERORDER - 1] - freq[0]) /
            (float) (LPC_FILTERORDER - 1);

        for (i=1; i<LPC_FILTERORDER; i++) {
            freq[i] = freq[i - 1] + hlp;
        }
```



```
    }

    memset(a1, 0, LPC_HALFORDER*sizeof(float));
    memset(a2, 0, LPC_HALFORDER*sizeof(float));
```

```
    memset(b1, 0, LPC_HALFORDER*sizeof(float));
    memset(b2, 0, LPC_HALFORDER*sizeof(float));
    memset(a, 0, (LPC_HALFORDER+1)*sizeof(float));
    memset(b, 0, (LPC_HALFORDER+1)*sizeof(float));

    /* p[i] and q[i] compute cos(2*pi*omega_{2j}) and
    cos(2*pi*omega_{2j-1} in eqs. 4.2.2.2-1 and 4.2.2.2-2.
    Note that for this code p[i] specifies the coefficients
    used in .Q_A(z) while q[i] specifies the coefficients used
    in .P_A(z) */

    for (i=0; i<LPC_HALFORDER; i++) {
        p[i] = (float)cos(TWO_PI * freq[2 * i]);
        q[i] = (float)cos(TWO_PI * freq[2 * i + 1]);
    }

    a[0] = 0.25;
    b[0] = 0.25;

    for (i= 0; i<LPC_HALFORDER; i++) {
        a[i + 1] = a[i] - 2 * p[i] * a1[i] + a2[i];
        b[i + 1] = b[i] - 2 * q[i] * b1[i] + b2[i];
        a2[i] = a1[i];
        a1[i] = a[i];
        b2[i] = b1[i];
        b1[i] = b[i];
    }

    for (j=0; j<LPC_FILTERORDER; j++) {

        if (j == 0) {
            a[0] = 0.25;
            b[0] = -0.25;
        } else {
            a[0] = b[0] = 0.0;
        }

        for (i=0; i<LPC_HALFORDER; i++) {
            a[i + 1] = a[i] - 2 * p[i] * a1[i] + a2[i];
            b[i + 1] = b[i] - 2 * q[i] * b1[i] + b2[i];
            a2[i] = a1[i];
            a1[i] = a[i];
            b2[i] = b1[i];
            b1[i] = b[i];
        }

        a_coef[j + 1] = 2 * (a[LPC_HALFORDER] + b[LPC_HALFORDER]);
    }

    a_coef[0] = 1.0;
```

```
}
```

A.41 packing.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    packing.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __PACKING_H
#define __PACKING_H

void packsplit(
    int *index,                 /* (i) the value to split */
    int *firstpart,             /* (o) the value specified by most
                                       significant bits */
    int *rest,                  /* (o) the value specified by least
                                       significant bits */
    int bitno_firstpart,    /* (i) number of bits in most
                                       significant part */
    int bitno_total             /* (i) number of bits in full range
                                       of value */
);

void packcombine(
    int *index,                 /* (i/o) the msb value in the
                                       combined value out */
    int rest,                   /* (i) the lsb value */
    int bitno_rest              /* (i) the number of bits in the
                                       lsb part */
);

void dopack(
    unsigned char **bitstream,  /* (i/o) on entrance pointer to
                                       place in bitstream to pack
                                       new data, on exit pointer
                                       to place in bitstream to
                                       pack future data */
    int index,                  /* (i) the value to pack */
    int bitno,                  /* (i) the number of bits that the
                                       value will fit within */
    int *pos                /* (i/o) write position in the
                                       current byte */
```

```
   );

   void unpack(
       unsigned char **bitstream,  /* (i/o) on entrance pointer to
```



```
                                          place in bitstream to
                                          unpack new data from, on
                                          exit pointer to place in
                                          bitstream to unpack future
                                          data from */
       int *index,                 /* (o) resulting value */
       int bitno,                  /* (i) number of bits used to
                                          represent the value */
       int *pos                /* (i/o) read position in the
                                          current byte */
   );

   #endif
```

A.42 packing.c

```
   /******************************************************************

       iLBC Speech Coder ANSI-C Source Code

       packing.c

       Copyright (C) The Internet Society (2004).
       All Rights Reserved.

   ******************************************************************/

   #include <math.h>
   #include <stdlib.h>

   #include "iLBC_define.h"
   #include "constants.h"
   #include "helpfun.h"
   #include "string.h"

   /*----------------------------------------------------------------*
    *  splitting an integer into first most significant bits and
    *  remaining least significant bits
    *---------------------------------------------------------------*/

   void packsplit(
       int *index,                 /* (i) the value to split */
       int *firstpart,             /* (o) the value specified by most
```

```
                                       significant bits */
    int *rest,                  /* (o) the value specified by least
                                       significant bits */
    int bitno_firstpart,    /* (i) number of bits in most
                                       significant part */
    int bitno_total             /* (i) number of bits in full range
                                       of value */
){
    int bitno_rest = bitno_total-bitno_firstpart;
```



```
    *firstpart = *index>>(bitno_rest);
    *rest = *index-(*firstpart<<(bitno_rest));
}

/*----------------------------------------------------------------*
 *  combining a value corresponding to msb's with a value
 *  corresponding to lsb's
 *---------------------------------------------------------------*/

void packcombine(
    int *index,                 /* (i/o) the msb value in the
                                       combined value out */
    int rest,                   /* (i) the lsb value */
    int bitno_rest              /* (i) the number of bits in the
                                       lsb part */
){
    *index = *index<<bitno_rest;
    *index += rest;
}

/*----------------------------------------------------------------*
 *  packing of bits into bitstream, i.e., vector of bytes
 *---------------------------------------------------------------*/

void dopack(
    unsigned char **bitstream,  /* (i/o) on entrance pointer to
                                       place in bitstream to pack
                                       new data, on exit pointer
                                       to place in bitstream to
                                       pack future data */
    int index,                  /* (i) the value to pack */
    int bitno,                  /* (i) the number of bits that the
                                       value will fit within */
    int *pos                /* (i/o) write position in the
                                       current byte */
){
    int posLeft;
```

```
    /* Clear the bits before starting in a new byte */

    if ((*pos)==0) {
        **bitstream=0;
    }

    while (bitno>0) {

        /* Jump to the next byte if end of this byte is reached*/

        if (*pos==8) {
            *pos=0;
            (*bitstream)++;
            **bitstream=0;
        }
```



```
        posLeft=8-(*pos);

        /* Insert index into the bitstream */

        if (bitno <= posLeft) {
            **bitstream |= (unsigned char)(index<<(posLeft-bitno));
            *pos+=bitno;
            bitno=0;
        } else {
            **bitstream |= (unsigned char)(index>>(bitno-posLeft));

            *pos=8;
            index-=((index>>(bitno-posLeft))<<(bitno-posLeft));

            bitno-=posLeft;
        }
    }
}

/*----------------------------------------------------------------*
 *  unpacking of bits from bitstream, i.e., vector of bytes
 *---------------------------------------------------------------*/

void unpack(
    unsigned char **bitstream,  /* (i/o) on entrance pointer to
                                       place in bitstream to
                                       unpack new data from, on
                                       exit pointer to place in
                                       bitstream to unpack future
                                       data from */
    int *index,                 /* (o) resulting value */
```

```
    int bitno,                  /* (i) number of bits used to
                                       represent the value */
    int *pos                /* (i/o) read position in the
                                       current byte */
){
    int BitsLeft;

    *index=0;

    while (bitno>0) {

        /* move forward in bitstream when the end of the
           byte is reached */

        if (*pos==8) {
            *pos=0;
            (*bitstream)++;
        }

        BitsLeft=8-(*pos);

        /* Extract bits to index */
```



```
        if (BitsLeft>=bitno) {
            *index+=((((**bitstream)<<(*pos)) & 0xFF)>>(8-bitno));

            *pos+=bitno;
            bitno=0;
        } else {

            if ((8-bitno)>0) {
                *index+=((((**bitstream)<<(*pos)) & 0xFF)>>
                    (8-bitno));
                *pos=8;
            } else {
                *index+=(((int)(((**bitstream)<<(*pos)) & 0xFF))<<
                    (bitno-8));
                *pos=8;
            }
            bitno-=BitsLeft;
        }
    }
}
```

A.43 StateConstructW.h

```
/******************************************************************
```

```
    iLBC Speech Coder ANSI-C Source Code

    StateConstructW.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_STATECONSTRUCTW_H
#define __iLBC_STATECONSTRUCTW_H

void StateConstructW(
    int idxForMax,      /* (i) 6-bit index for the quantization of
                               max amplitude */
    int *idxVec,    /* (i) vector of quantization indexes */
    float *syntDenum,   /* (i) synthesis filter denumerator */
    float *out,         /* (o) the decoded state vector */
    int len             /* (i) length of a state vector */
);

#endif
```

A.44 StateConstructW.c

```
/******************************************************************
```



```
    iLBC Speech Coder ANSI-C Source Code

    StateConstructW.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <math.h>
#include <string.h>

#include "iLBC_define.h"
#include "constants.h"
#include "filter.h"

/*----------------------------------------------------------------*
 *  decoding of the start state
 *---------------------------------------------------------------*/
```

```
void StateConstructW(
    int idxForMax,      /* (i) 6-bit index for the quantization of
                               max amplitude */
    int *idxVec,    /* (i) vector of quantization indexes */
    float *syntDenum,   /* (i) synthesis filter denumerator */
    float *out,         /* (o) the decoded state vector */
    int len             /* (i) length of a state vector */
){
    float maxVal, tmpbuf[LPC_FILTERORDER+2*STATE_LEN], *tmp,
        numerator[LPC_FILTERORDER+1];
    float foutbuf[LPC_FILTERORDER+2*STATE_LEN], *fout;
    int k,tmpi;

    /* decoding of the maximum value */

    maxVal = state_frgqTbl[idxForMax];
    maxVal = (float)pow(10,maxVal)/(float)4.5;

    /* initialization of buffers and coefficients */

    memset(tmpbuf, 0, LPC_FILTERORDER*sizeof(float));
    memset(foutbuf, 0, LPC_FILTERORDER*sizeof(float));
    for (k=0; k<LPC_FILTERORDER; k++) {
        numerator[k]=syntDenum[LPC_FILTERORDER-k];
    }
    numerator[LPC_FILTERORDER]=syntDenum[0];
    tmp = &tmpbuf[LPC_FILTERORDER];
    fout = &foutbuf[LPC_FILTERORDER];

    /* decoding of the sample values */

    for (k=0; k<len; k++) {
        tmpi = len-1-k;
        /* maxVal = 1/scal */
```



```
        tmp[k] = maxVal*state_sq3Tbl[idxVec[tmpi]];
    }

    /* circular convolution with all-pass filter */

    memset(tmp+len, 0, len*sizeof(float));
    ZeroPoleFilter(tmp, numerator, syntDenum, 2*len,
        LPC_FILTERORDER, fout);
    for (k=0;k<len;k++) {
        out[k] = fout[len-1-k]+fout[2*len-1-k];
    }
}
```

A.45 StateSearchW.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    StateSearchW.h

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#ifndef __iLBC_STATESEARCHW_H
#define __iLBC_STATESEARCHW_H

void AbsQuantW(
    iLBC_Enc_Inst_t *iLBCenc_inst,
                        /* (i) Encoder instance */
    float *in,          /* (i) vector to encode */
    float *syntDenum,   /* (i) denominator of synthesis filter */
    float *weightDenum, /* (i) denominator of weighting filter */
    int *out,           /* (o) vector of quantizer indexes */
    int len,        /* (i) length of vector to encode and
                               vector of quantizer indexes */
    int state_first     /* (i) position of start state in the
                               80 vec */
);

void StateSearchW(
    iLBC_Enc_Inst_t *iLBCenc_inst,
                        /* (i) Encoder instance */
    float *residual,/* (i) target residual vector */
    float *syntDenum,   /* (i) lpc synthesis filter */
    float *weightDenum, /* (i) weighting filter denuminator */
    int *idxForMax,     /* (o) quantizer index for maximum
                               amplitude */
    int *idxVec,    /* (o) vector of quantization indexes */
    int len,        /* (i) length of all vectors */
    int state_first     /* (i) position of start state in the
```



```
                               80 vec */
);


#endif
```

A.46 StateSearchW.c

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    StateSearchW.c

    Copyright (C) The Internet Society (2004).
    All Rights Reserved.

******************************************************************/

#include <math.h>
#include <string.h>

#include "iLBC_define.h"
#include "constants.h"
#include "filter.h"
#include "helpfun.h"

/*----------------------------------------------------------------*
 *  predictive noise shaping encoding of scaled start state
 *  (subrutine for StateSearchW)
 *---------------------------------------------------------------*/

void AbsQuantW(
    iLBC_Enc_Inst_t *iLBCenc_inst,
                        /* (i) Encoder instance */
    float *in,          /* (i) vector to encode */
    float *syntDenum,   /* (i) denominator of synthesis filter */
    float *weightDenum, /* (i) denominator of weighting filter */
    int *out,           /* (o) vector of quantizer indexes */
    int len,        /* (i) length of vector to encode and
                               vector of quantizer indexes */
    int state_first     /* (i) position of start state in the
                               80 vec */
){
    float *syntOut;
    float syntOutBuf[LPC_FILTERORDER+STATE_SHORT_LEN_30MS];
    float toQ, xq;
    int n;
    int index;

    /* initialization of buffer for filtering */

    memset(syntOutBuf, 0, LPC_FILTERORDER*sizeof(float));
```

```
/* initialization of pointer for filtering */

syntOut = &syntOutBuf[LPC_FILTERORDER];

/* synthesis and weighting filters on input */

if (state_first) {
    AllPoleFilter (in, weightDenum, SUBL, LPC_FILTERORDER);
} else {
    AllPoleFilter (in, weightDenum,
        iLBCenc_inst->state_short_len-SUBL,
        LPC_FILTERORDER);
}

/* encoding loop */

for (n=0; n<len; n++) {

    /* time update of filter coefficients */

    if ((state_first)&&(n==SUBL)){
        syntDenum += (LPC_FILTERORDER+1);
        weightDenum += (LPC_FILTERORDER+1);

        /* synthesis and weighting filters on input */
        AllPoleFilter (&in[n], weightDenum, len-n,
            LPC_FILTERORDER);

    } else if ((state_first==0)&&
        (n==(iLBCenc_inst->state_short_len-SUBL))) {
        syntDenum += (LPC_FILTERORDER+1);
        weightDenum += (LPC_FILTERORDER+1);

        /* synthesis and weighting filters on input */
        AllPoleFilter (&in[n], weightDenum, len-n,
            LPC_FILTERORDER);

    }

    /* prediction of synthesized and weighted input */

    syntOut[n] = 0.0;
    AllPoleFilter (&syntOut[n], weightDenum, 1,
        LPC_FILTERORDER);

    /* quantization */

    toQ = in[n]-syntOut[n];
    sort_sq(&xq, &index, toQ, state_sq3Tbl, 8);
    out[n]=index;
    syntOut[n] = state_sq3Tbl[out[n]];
```

```
        /* update of the prediction filter */
```



```
        AllPoleFilter(&syntOut[n], weightDenum, 1,
            LPC_FILTERORDER);
    }
}

/*----------------------------------------------------------------*
 *  encoding of start state
 *---------------------------------------------------------------*/

void StateSearchW(
    iLBC_Enc_Inst_t *iLBCenc_inst,
                        /* (i) Encoder instance */
    float *residual,/* (i) target residual vector */
    float *syntDenum,   /* (i) lpc synthesis filter */
    float *weightDenum, /* (i) weighting filter denuminator */
    int *idxForMax,     /* (o) quantizer index for maximum
                               amplitude */
    int *idxVec,    /* (o) vector of quantization indexes */
    int len,        /* (i) length of all vectors */
    int state_first     /* (i) position of start state in the
                               80 vec */
){
    float dtmp, maxVal;
    float tmpbuf[LPC_FILTERORDER+2*STATE_SHORT_LEN_30MS];
    float *tmp, numerator[1+LPC_FILTERORDER];
    float foutbuf[LPC_FILTERORDER+2*STATE_SHORT_LEN_30MS], *fout;
    int k;
    float qmax, scal;

    /* initialization of buffers and filter coefficients */

    memset(tmpbuf, 0, LPC_FILTERORDER*sizeof(float));
    memset(foutbuf, 0, LPC_FILTERORDER*sizeof(float));
    for (k=0; k<LPC_FILTERORDER; k++) {
        numerator[k]=syntDenum[LPC_FILTERORDER-k];
    }
    numerator[LPC_FILTERORDER]=syntDenum[0];
    tmp = &tmpbuf[LPC_FILTERORDER];
    fout = &foutbuf[LPC_FILTERORDER];

    /* circular convolution with the all-pass filter */

    memcpy(tmp, residual, len*sizeof(float));
    memset(tmp+len, 0, len*sizeof(float));
    ZeroPoleFilter(tmp, numerator, syntDenum, 2*len,
```

```
        LPC_FILTERORDER, fout);
    for (k=0; k<len; k++) {
        fout[k] += fout[k+len];
    }

    /* identification of the maximum amplitude value */

    maxVal = fout[0];
```



```
    for (k=1; k<len; k++) {

        if (fout[k]*fout[k] > maxVal*maxVal){
            maxVal = fout[k];
        }
    }
    maxVal=(float)fabs(maxVal);

    /* encoding of the maximum amplitude value */

    if (maxVal < 10.0) {
        maxVal = 10.0;
    }
    maxVal = (float)log10(maxVal);
    sort_sq(&dtmp, idxForMax, maxVal, state_frgqTbl, 64);

    /* decoding of the maximum amplitude representation value,
       and corresponding scaling of start state */

    maxVal=state_frgqTbl[*idxForMax];
    qmax = (float)pow(10,maxVal);
    scal = (float)(4.5)/qmax;
    for (k=0; k<len; k++){
        fout[k] *= scal;
    }

    /* predictive noise shaping encoding of scaled start state */

    AbsQuantW(iLBCenc_inst, fout,syntDenum,
        weightDenum,idxVec, len, state_first);
}
```

A.47 syntFilter.h

```
/******************************************************************

    iLBC Speech Coder ANSI-C Source Code

    syntFilter.h
```

```
       Copyright (C) The Internet Society (2004).
       All Rights Reserved.

   ******************************************************************/

   #ifndef __iLBC_SYNTFILTER_H
   #define __iLBC_SYNTFILTER_H

   void syntFilter(
       float *Out,     /* (i/o) Signal to be filtered */
       float *a,       /* (i) LP parameters */
       int len,    /* (i) Length of signal */
       float *mem      /* (i/o) Filter state */
```



```
   );

   #endif
```

A.48 syntFilter.c

```
   /******************************************************************

       iLBC Speech Coder ANSI-C Source Code

       syntFilter.c

       Copyright (C) The Internet Society (2004).
       All Rights Reserved.

   ******************************************************************/

   #include "iLBC_define.h"

   /*----------------------------------------------------------------*
    *  LP synthesis filter.
    *---------------------------------------------------------------*/

   void syntFilter(
       float *Out,     /* (i/o) Signal to be filtered */
       float *a,       /* (i) LP parameters */
       int len,    /* (i) Length of signal */
       float *mem      /* (i/o) Filter state */
   ){
       int i, j;
       float *po, *pi, *pa, *pm;

       po=Out;
```

```
    /* Filter first part using memory from past */

    for (i=0; i<LPC_FILTERORDER; i++) {
        pi=&Out[i-1];
        pa=&a[1];
        pm=&mem[LPC_FILTERORDER-1];
        for (j=1; j<=i; j++) {
            *po-=(*pa++)*(*pi--);
        }
        for (j=i+1; j<LPC_FILTERORDER+1; j++) {
            *po-=(*pa++)*(*pm--);
        }
        po++;
    }

    /* Filter last part where the state is entierly in
       the output vector */

    for (i=LPC_FILTERORDER; i<len; i++) {
```



```
        pi=&Out[i-1];
        pa=&a[1];
        for (j=1; j<LPC_FILTERORDER+1; j++) {
            *po-=(*pa++)*(*pi--);
        }
        po++;
    }

    /* Update state vector */

    memcpy(mem, &Out[len-LPC_FILTERORDER],
        LPC_FILTERORDER*sizeof(float));
}
```