INTERNATIONAL TELECOMMUNICATION UNION

CCITT G.728

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

(09/92)

GENERAL ASPECTS OF DIGITAL TRANSMISSION SYSTEMS;

TERMINAL EQUIPMENTS

CODING OF SPEECH AT 16 kbit/s USING LOW-DELAY CODE EXCITED LINEAR PREDICTION

Recommendation G.728

Geneva, 1992

FOREWORD

The CCITT (the International Telegraph and Telephone Consultative Committee) is a permanent organ of the International Telecommunication Union (ITU). CCITT is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The Plenary Assembly of CCITT which meets every four years, establishes the topics for study and approves Recommendations prepared by its Study Groups. The approval of Recommendations by the members of CCITT between Plenary Assemblies is covered by the procedure laid down in CCITT Resolution No. 2 (Melbourne, 1988).

Recommendation G.796 was prepared by Study Group XV and was approved under the Resolution No. 2 procedure on the 1st of September 1992.

CCITT NOTES

1) In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized private operating agency.

2) A list of abbreviations used in this Recommendation can be found in Annex F.

© ITU 1992

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the ITU.

Recommendation G.728

CODING OF SPEECH AT 16 kbit/s USING LOW-DELAY CODE EXCITED LINEAR PREDICTION

(1992)

1 Introduction

This Recommendation contains the description of an algorithm for the coding of speech signals at 16 kbit/s using low-delay code excited linear prediction (LD-CELP). This Recommendation is organized as follows.

In § 2 a brief outline of the LD-CELP algorithm is given. In §§ 3 and 4, the LD-CELP encoder and LD-CELP decoder principles are discussed, respectively. In § 5, the computational details pertaining to each functional algorithmic block are defined. Annexes A, B, C and D contain tables of constants used by the LD-CELP algorithm. In Annex E the sequencing of variable adaptation and use is given. Finally, in Appendix I information is given on procedures applicable to the implementation verification of the algorithm.

Under further study is the future incorporation of three additional appendices (to be published separately) consisting of LD-CELP network aspects, LD-CELP fixed-point implementation description, and LD-CELP fixed-point verification procedures.

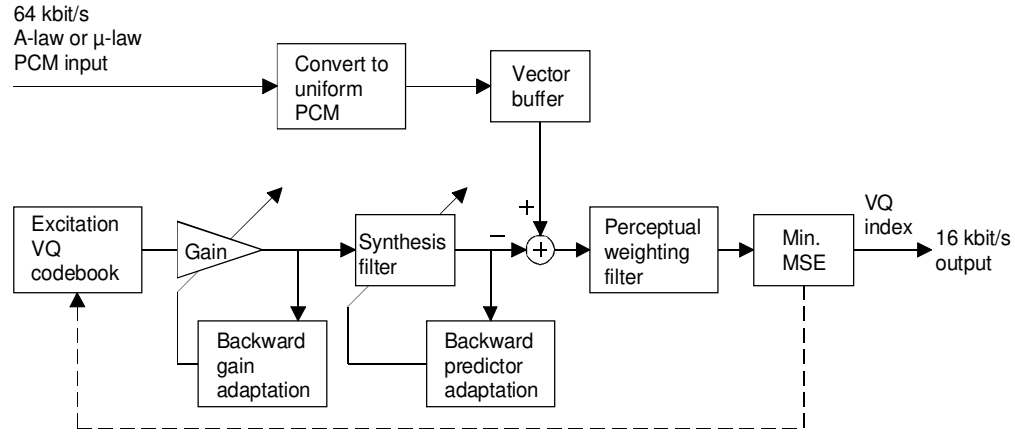
2 Outline of LD-CELP

The LD-CELP algorithm consists of an encoder and a decoder described in §§ 2.1 and 2.2 respectively, and illustrated in Figure 1/G.728.

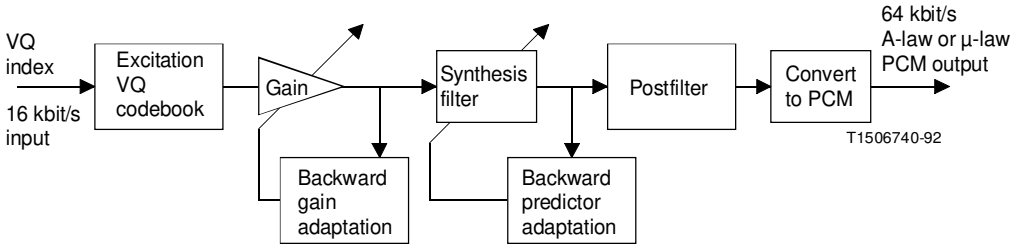
The essence of CELP techniques, which is an analysis-by-synthesis approach to codebook search, is retained in LD-CELP. The LD-CELP however, uses backward adaptation of predictors and gain to achieve an algorithmic delay of 0.625 ms. Only the index to the excitation codebook is transmitted. The predictor coefficients are updated through LPC analysis of previously quantized speech. The excitation gain is updated by using the gain information embedded in the previously quantized excitation. The block size for the excitation vector and gain adaptation is five samples only. A perceptual weighting filter is updated using LPC analysis of the unquantized speech.

2.1 *LD-CELP encoder*

After the conversion from A-law or μ-law PCM to uniform PCM, the input signal is partitioned into blocks of five-consecutive input signal samples. For each input block, the encoder passes each of 1024 candidate codebook vectors (stored in an excitation codebook) through a gain scaling unit and a synthesis filter. From the resulting 1024 candidate quantized signal vectors, the encoder identifies the one that minimizes a frequency-weighted mean-squared error measure with respect to the input signal vector. The 10-bit codebook index of the corresponding best codebook vector (or "codevector"), which gives rise to that best candidate quantized signal vector, is transmitted to the decoder. The best codevector is then passed through the gain scaling unit and the synthesis filter to establish the correct filter memory in preparation for the encoding of the next signal vector. The synthesis filter coefficients and the gain are updated periodically in a backward adaptive manner based on the previously quantized signal and gain-scaled excitation.



a) LD-CELP encoder



b) LD-CELP decoder

FIGURE 1/G.728

Simplified block diagram of LD-CELP coder

2.2 *LD-CELP decoder*

The decoding operation is also performed on a block-by-block basis. Upon receiving each 10-bit index, the decoder performs a table look-up to extract the corresponding codevector from the excitation codebook. The extracted codevector is then passed through a gain scaling unit and a synthesis filter to produce the current decoded signal vector. The synthesis filter coefficients and the gain are then updated in the same way as in the encoder. The decoded signal vector is then passed through an adaptive postfilter to enhance the perceptual quality. The postfilter coefficients are updated periodically using the information available at the decoder. The five samples of the postfilter signal vector are next converted to five A-law or μ-law PCM output samples.

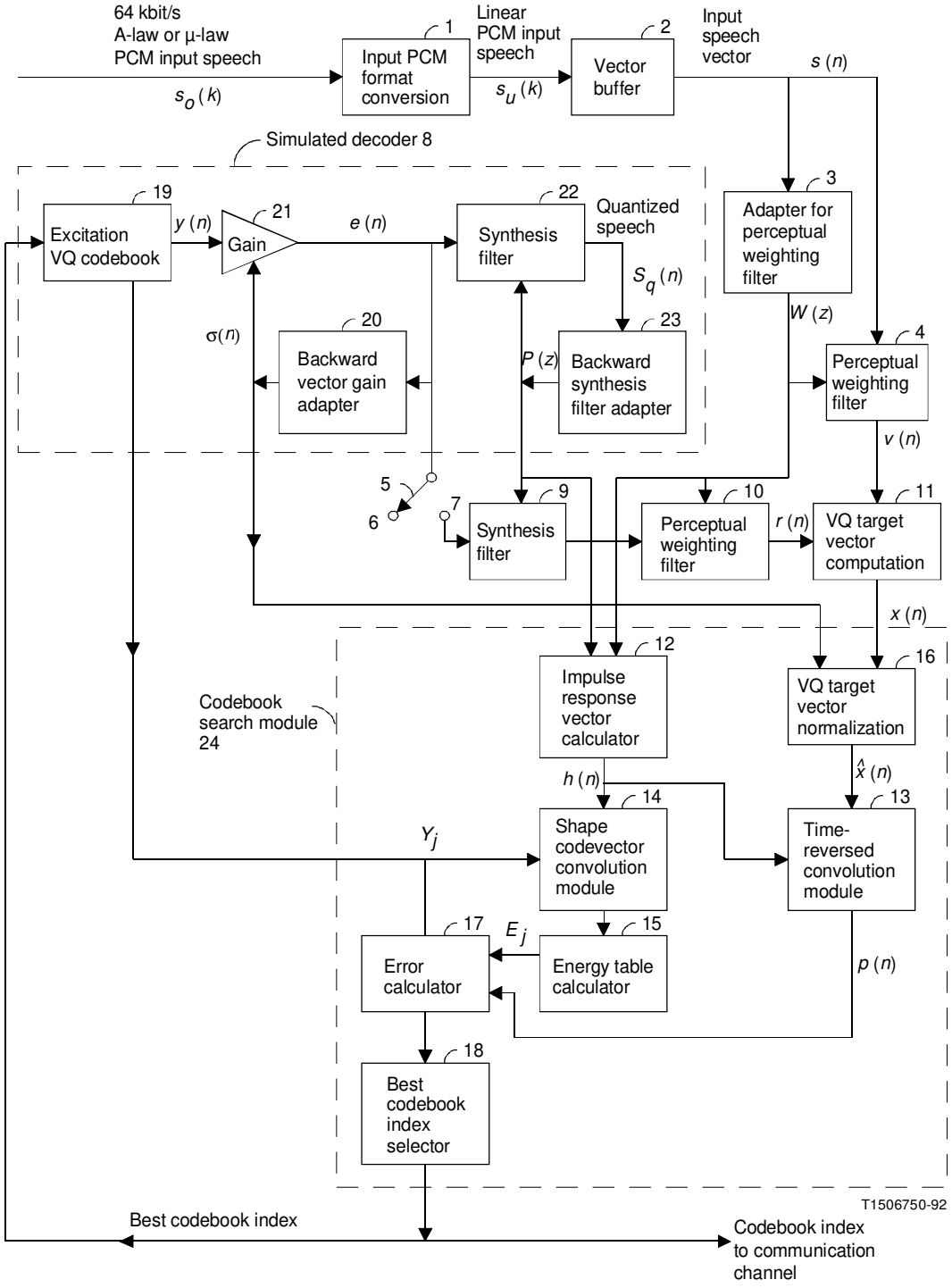


FIGURE 2/G.728

LD-CELP encoder block schematic

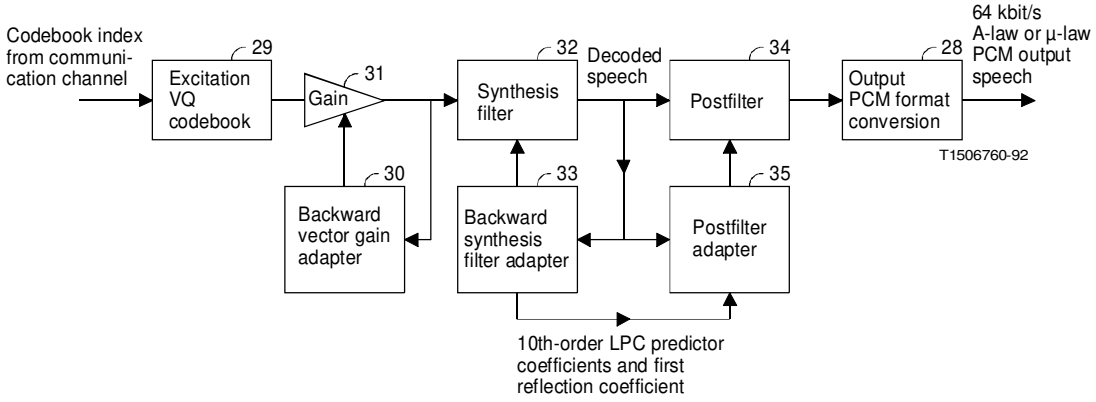


FIGURE 3/G.728

LD-CELP decoder block schematic

3 LD-CELP (encoder principles)

Figure 2/G.728 is a detailed block schematic of the LD-CELP encoder. The encoder in Figure 2/G.728 is mathematically equivalent to the encoder previously shown in Figure 1/G.728 but is computationally more efficient to implement.

In the following description:

a) for each variable to be described, k is the sampling index and samples are taken at 125 μs intervals;

b) a group of five consecutive samples in a given signal is called a *vector* of that signal. For example, five consecutive speech samples form a speech vector, five excitation samples form an excitation vector, and so on;

c) we use n to denote the vector index, which is different from the sample index k;

d) four consecutive vectors build one *adaptation cycle*. In a later section, we also refer to adaptation cycles as *frames*. The two terms are used interchangeably.

The excitation vector quantization (VQ) codebook index is the only information explicitly transmitted from the encoder to the decoder. Three other types of parameters will be periodically updated: the excitation gain, the synthesis filter coefficients, and the perceptual weighting filter coefficients. These parameters are derived in a backward adaptive manner from signals that occur prior to the current signal vector. The excitation gain is updated once per vector, while the synthesis filter coefficients and the perceptual weighting filter coefficients are updated once every four vectors (i.e. a 20-sample, or 2.5 ms update period). Note that, although the processing sequence in the algorithm has an adaptation cycle of four vectors (20 samples), the basic buffer size is still only one vector (five samples). This small buffer size makes it possible to achieve a one-way delay less than 2 ms.

A description of each block of the encoder is given below. Since the LD-CELP coder is mainly used for encoding speech, for convenience of description, in the following we will assume that the input signal is speech, although in practice it can be other non-speech signals as well.

3.1 *Input PCM format conversion*

This block converts the input A-law or μ-law PCM signal $s_o(k)$ to a uniform PCM signal $s_u(k)$.

3.1.1 *Internal linear PCM levels*

In converting from A-law or μ-law to linear PCM, different internal representations are possible, depending on the device. For example, standard tables for μ-law PCM define a linear range of –4 015.5 to +4 015.5. The corresponding range for A-law PCM is –2 016 to +2 016. Both tables list some output values having a fractional part of 0.5. These fractional parts cannot be represented in an integer device unless the entire table is multiplied by 2 to make all of the values integers. In fact, this is what is most commonly done in fixed point digital signal processing (DSP) chips. On the other hand, floating point DSP chips can represent the same values listed in the tables. Throughout this document it is assumed that the input signal has a maximum range of –4 095 to +4 095. This encompasses both the μ-law and A-law cases. In the case of A-law it implies that when the linear conversion results in a range of –2 016 to +2 016, those values should be scaled up by a factor of 2 before continuing to encode the signal. In the case of μ-law input to a fixed point processor where the input range is converted to –8 031 to +8 031, it implies that values should be scaled down by a factor of 2 before beginning the encoding process. Alternatively, these values can be treated as being in Q1 format, meaning there is one bit to the right of the decimal point. All computation involving the data would then need to take this bit into account.

For the case of 16-bit linear PCM input signals having full dynamic range of –32 768 to +32 767, the input values should be considered to be in Q3 format. This means that the input values should be scaled down (divided) by a factor of 8. On output at the decoder the factor of 8 would be restored for these signals.

3.2 *Vector buffer*

This block buffers five consecutive speech samples $s_u(5n)$, $s_u(5n + 1)$, ..., $s_u(5n + 4)$ to form a 5-dimensional speech vector $s(n) = [s_u(5n), s_u(5n + 1), ..., s_u(5n + 4)]$.

3.3 *Adapter for perceptual weighting filter*

Figure 4/G.728 shows the detailed operation of the perceptual weighting filter adapter (block 3 in Figure 2/G.728). This adapter calculates the coefficients of the perceptual weighting filter once every four speech vectors based on linear prediction analysis (often referred to as LPC analysis) of unquantized speech. The coefficient updates occur at the third speech vector of every 4-vector adaptation cycle. The coefficients are held constant in between updates.

Refer to Figure 4a)/G.728. The calculation is performed as follows. First, the input (unquantized) speech vector is passed through a hybrid windowing module (block 36) which places a window on previous speech vectors and calculates the first 11 autocorrelation coefficients of the windowed speech signal as the output. The Levinson-Durbin recursion module (block 37) then converts these autocorrelation coefficients to predictor coefficients. Based on these predictor coefficients, the weighting filter coefficient calculator (block 38) derives the desired coefficients of the weighting filter. These three blocks are discussed in more detail below.

First, let us describe the principles of hybrid windowing. Since this hybrid windowing technique will be used in three different kinds of LPC analyses, we first give a more general description of the technique and then specialize it to different cases. Suppose the LPC analysis is to be performed once every L signal samples. To be general, assume that the signal samples corresponding to the current LD-CELP adaptation cycle are $s_u(m)$, $s_u(m + 1)$, $s_u(m + 2)$, ..., $s_u(m + L – 1)$. Then, for backward-adaptive LPC analysis, the hybrid window is applied to all previous signal samples with a sample index less than m (as shown in Figure 4b)/G.728). Let there be N non-recursive samples in the hybrid window function. Then, the signal samples $s_u(m – 1)$, $s_u(m – 2)$, ..., $s_u(m – N)$ are all weighted by the non-recursive portion of the window. Starting with $s_u(m – N – 1)$, all signal samples to the left of (and including) this sample are weighted by the recursive portion of the window, which has values b, $b\alpha$, $b\alpha^2$, ..., where $0 < b < 1$ and $0 < \alpha < 1$.

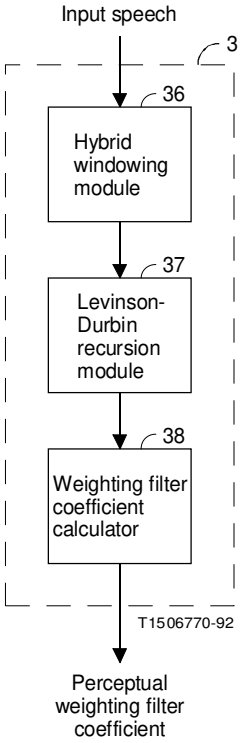


FIGURE 4a)/G.728

Perceptual weighting filter adapter

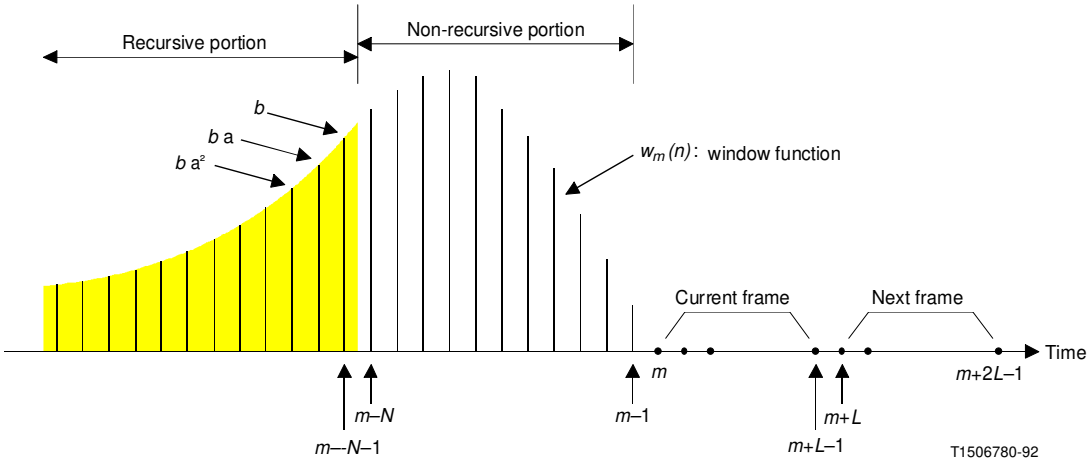


FIGURE 4b)/G.728

Illustration of a hybrid window

At time m, the hybrid window function $w_m(k)$ is defined as

$$w_m(k) = \begin{cases} f_m(k) = b\alpha^{-[k-(m-N-1)]}, & \text{if} \quad k \le m-N-1 \\ g_m(k) = -\sin[c(k-m)], & \text{if} \quad m-N \le k \le m-1 \\ 0, & \text{if} \quad k \ge m \end{cases} \tag{3-1a}$$

and the window-weighted signal is

$$s_m(k) = s_u(k)\, w_m(k) = \begin{cases} s_u(k)\, f_m(k) = s_u(k)\, b\alpha^{-[k-(m-N-1)]}, & \text{if} \quad k \le m-N-1 \\ s_u(k)\, g_m(k) = -s_u(k) \sin[c(k-m)], & \text{if} \quad m-N \le k \le m-1 \\ 0, & \text{if} \quad k \ge m \end{cases} \tag{3-1b}$$

The samples of non-recursive portion $g_m(k)$ and the initial section of the recursive portion $f_m(k)$ for different hybrid windows are specified in Annex A. For an M-th order LPC analysis, we need to calculate $M + 1$ autocorrelation coefficients $R_m(i)$ for $i = 0, 1, 2, ..., M$. The i-th autocorrelation coefficient for the current adaptation cycle can be expressed as

$$R_m(i) = \sum_{k=-\infty}^{m-1} s_m(k)\, s_m(k-i) = r_m(i) + \sum_{k=m-N}^{m-1} s_m(k)\, s_m(k-i) \tag{3-1c}$$

where

$$r_m(i) = \sum_{k=-\infty}^{m-N-1} s_m(k)\, s_m(k-i) = \sum_{k=-\infty}^{m-N-1} s_u(k)\, s_u(k-i)\, f_m(k)\, f_m(k-i) \tag{3-1d}$$

On the right-hand side of equation (3-1c), the first term $r_m(i)$ is the "recursive component" of $R_m(i)$, while the second term is the "non-recursive component". The finite summation of the non-recursive component is calculated for each adaptation cycle. On the other hand, the recursive component is calculated recursively. The following paragraphs explain how.

Suppose we have calculated and stored all $r_m(i)$s for the current adaptation cycle and want to go on to the next adaptation cycle, which starts at sample $s_u(m + L)$. After the hybrid window is shifted to the right by L samples, the new window-weighted signal for the next adaptation cycle becomes

$$s_{m+L}(k) = s_u(k)\, w_{m+L}(k) = \begin{cases} s_u(k)\, f_{m+L}(k) = s_u(k)\, f_m(k)\, \alpha^L, & \text{if} \quad k \le m+L-N-1 \\ s_u(k)\, g_{m+L}(k) = -s_u(k) \sin[c(k-m-L)], & \text{if} \quad m+L-N \le k \le m+L-1 \\ 0, & \text{if} \quad k \ge m+L \end{cases} \tag{3-1e}$$

The recursive component of $R_{m+L}(i)$ can be written as

$$\begin{aligned} r_{m+L}(i) &= \sum_{k=-\infty}^{m+L-N-1} s_{m+L}(k)\, s_{m+L}(k-i) \\ &= \sum_{k=-\infty}^{m-N-1} s_{m+L}(k)\, s_{m+L}(k-i) + \sum_{k=m-N}^{m+L-N-1} s_{m+L}(k)\, s_{m+L}(k-i) \\ &= \sum_{k=-\infty}^{m-N-1} s_u(k)\, f_m(k)\, \alpha^L\, s_u(k-i)\, f_m(k-i)\, \alpha^L + \sum_{k=m-N}^{m+L-N-1} s_{m+L}(k)\, s_{m+L}(k-i) \end{aligned} \tag{3-1f}$$

or

$$r_{m+L}(i) = \alpha^{2L}\, r_m(i) + \sum_{k=m-N}^{m+L-N-1} s_{m+L}(k)\, s_{m+L}(k-i) \tag{3-1g}$$

Therefore, $r_{m+L}(i)$ can be calculated recursively from $r_m(i)$ using equation (3-1g). This newly calculated $r_{m+L}(i)$ is stored back to memory for use in the following adaptation cycle. The autocorrelation coefficient $r_{m+L}(i)$ is then calculated as

$$R_{m+L}(i) = r_{m+L}(i) + \sum_{k=m+L-N}^{m+L-1} s_{m+L}(k)\, s_{m+L}(k-i) \tag{3-1h}$$

So far we have described in a general manner the principles of a hybrid window calculation procedure. The parameter values for the hybrid windowing module 36 in Figure 4a)/G.728 are

$$M = 10, L = 20, N = 30 \text{ and } \alpha = \left(\frac{1}{2}\right)^{\frac{1}{40}} = 0.982820598 \left(\text{ so that } \alpha^{2L} = \frac{1}{2} \right)$$

Once the 11 autocorrelation coefficients $R(i)$, $i = 0, 1, ..., 10$ are calculated by the hybrid windowing procedure described above, a "white noise correction" procedure is applied. This is done by increasing the energy $R(0)$ by a small amount:

$$R(0) \leftarrow \left(\frac{257}{256}\right) R(0) \tag{3-1i}$$

This has the effect of filling the spectral valleys with white noise so as to reduce the spectral dynamic range and alleviate ill-conditioning of the subsequent Levinson-Durbin recursion. The white noise correction factor (WNCF) of 257/256 corresponds to a white noise level about 24 dB below the average speech power.

Next, using the white noise corrected autocorrelation coefficients, the Levinson-Durbin recursion module 37 recursively computes the predictor coefficients from order 1 to order 10. Let the j-th coefficients of the i-th order predictor be $a_j^{(i)}$. Then, the recursive procedure can be specified as follows:

$$E(0) = R(0) \tag{3-2a}$$

$$k_i = -\frac{R(i) + \sum_{j=1}^{i-1} a_j^{(i-1)} R(i-j)}{E(i-1)} \tag{3-2b}$$

$$a_i^{(i)} = k_i \tag{3-2c}$$

$$a_j^{(i)} = a_j^{(i-1)} + k_i\, a_{i-j}^{(i-1)}; \qquad 1 \le j \le i-1 \tag{3-2d}$$

$$E(i) = (1 - k_i^2)\, E(i-1) \tag{3-2e}$$

Equations (3-2b) through (3-2e) are evaluated recursively for $i = 1, 2, ..., 10$, and the final solution is given by

$$q_i = a_i^{(10)}, \qquad 1 \le i \le 10 \tag{3-2f}$$

If we define q_0=1, then the 10-th order "prediction-error filter" (sometimes called "analysis filter") has the transfer function

$$\tilde{Q}(z) = \sum_{i=0}^{10} q_i z^{-i} \tag{3-3a}$$

and the corresponding 10-th order linear predictor is defined by the following transfer function

$$Q(z) = -\sum_{i=1}^{10} q_i z^{-i} \tag{3-3b}$$

The weighting filter coefficient calculator (block 38) calculates the perceptual weighting filter coefficients according to the following equations:

$$Q(z / \gamma_1) = -\sum_{i=1}^{10} (q_i \gamma_1^i) z^{-i} \tag{3-4b}$$

and

$$Q(z / \gamma_2) = -\sum_{i=1}^{10} (q_i \gamma_2^i) z^{-i} \tag{3-4c}$$

The perceptual weighting filter is a 10-th order pole-zero filter defined by the transfer function $W(z)$ in equation (3-4a). The values of γ_1 and γ_2 are 0.9 and 0.6, respectively.

Now refer to Figure 2/G.728. The perceptual weighting filter adapter (block 3) periodically updates the coefficients of $W(z)$ according to equations (3-2) through (3-4), and feeds the coefficients to the impulse response vector calculator (block 12) and the perceptual weighting filters (blocks 4 and 10).

3.4 *Perceptual weighting filter*

In Figure 2/G.728, the current input speech vector $s(n)$ is passed through the perceptual weighting filter (block 4), resulting in the weighted speech vector $v(n)$. Note that except during initialization, the filter memory (i.e. internal state variables, or the values held in the delay units of the filter) should not be reset to zero at any time. On the other hand, the memory of the perceptual weighting filter (block 10) will need special handling as described later.

3.4.1 *Non-speech operation*

For modem signals or other non-speech signals, CCITT test results indicate that it is desirable to disable the perceptual weighting filter. This is equivalent to setting $W(z)$=1. This can most easily be accomplished if γ_1 and γ_2 in equation (3-4a) are set equal to zero. The nominal values for these variables in the speech mode are 0.9 and 0.6, respectively.

3.5 *Synthesis filter*

In Figure 2/G.728, there are two synthesis filters (blocks 9 and 22) with identical coefficients. Both filters are updated by the backward synthesis filter adapter (block 23). Each synthesis filter is a 50-th order all-pole filter that consists of a feedback loop with a 50-th order LPC predictor in the feedback branch. The transfer function of the synthesis filter is $F(z) = 1/[1 - P(z)]$, where $P(z)$ is the transfer function of the 50-th order LPC predictor.

After the weighted speech vector $v(n)$ has been obtained, a zero-input response vector $r(n)$ will be generated using the synthesis filter (block 9) and the perceptual weighting filter (block 10). To accomplish this, we first open the switch 5, i.e. point it to node 6. This implies that the signal going from node 7 to the synthesis filter 9 will be zero. We then let the synthesis filter 9 and the perceptual weighting filter 10 "ring" for five samples (one vector). This means that we continue the filtering operation for five samples with a zero signal applied at node 7. The resulting output of the perceptual weighting filter 10 is the desired zero-input response vector $r(n)$.

Note that except for the vector right after initialization, the memory of the filters 9 and 10 is in general non-zero; therefore, the output vector $r(n)$ is also non-zero in general, even though the filter input from node 7 is zero. In effect, this vector $r(n)$ is the response of the two filters to previous gain-scaled excitation vectors $e(n-1)$, $e(n-2)$, ... This vector actually represents the effect due to filter memory up to time $(n-1)$.

3.6 *VQ target vector computation*

This block subtracts the zero-input response vector $r(n)$ from the weighted speech vector $v(n)$ to obtain the VQ codebook search target vector $x(n)$.

3.7 *Backward synthesis filter adapter*

This adapter 23 updates the coefficients of the synthesis filters 9 and 22. It takes the quantized (synthesized) speech as input and produces a set of synthesis filter coefficients as output. Its operation is quite similar to the perceptual weighting filter adapter 3.

A blown-up version of this adapter is shown in Figure 5/G.728. The operation of the hybrid windowing module 49 and the Levinson-Durbin recursion module 50 is exactly the same as their counterparts (36 and 37) in Figure 4a)/G.728, except for the following three differences:

a) the input signal is now the quantized speech rather than the unquantized input speech;

b) the predictor order is 50 rather than 10;

c) the hybrid window parameters are different: $N = 35$, $\alpha = \left(\frac{3}{4}\right)^{\frac{1}{40}} = 0.992833749$

Note that the update period is still $L = 20$, and the white noise correction factor is still 257/256 = 1.00390625.

Let $\hat{P}(z)$ be the transfer function of the 50-th order LPC predictor, then it has the form

$$\hat{P}(z) = -\sum_{i=1}^{50} \hat{a}_i z^{-i} \tag{3-5}$$

where $\hat{a}_i$ are the predictor coefficients. To improve robustness to channel errors, these coefficients are modified so that the peaks in the resulting LPC spectrum have slightly larger bandwidths. The bandwidth expansion module 51 performs this bandwidth expansion procedure in the following way. Given the LPC predictor coefficients $\hat{a}_i$, a new set of coefficients a_i is computed according to

$$a_i = \lambda^i \hat{a}_i, \; i = 1, 2, \ldots, 50 \tag{3-6}$$

where λ is given by

$$\lambda = \frac{253}{256} = 0.98828125 \tag{3-7}$$

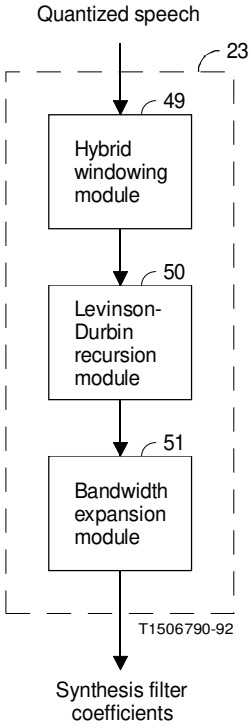


FIGURE 5/G.728

Backward synthesis filter adapter

This has the effects of moving all the poles of the synthesis filter radially toward the origin by a factor of λ. Since the poles are moved away from the unit circle, the peaks in the frequency response are widened.

After such bandwidth expansion, the modified LPC predictor has a transfer function of

$$P(z) = -\sum_{i=1}^{50} a_i z^{-i} \tag{3-8}$$

The modified coefficients are then fed to the synthesis filters 9 and 22. These coefficients are also fed to the impulse response vector calculator 12.

The synthesis filters 9 and 22 both have a transfer function of

$$F(z) = \frac{1}{1 - P(z)} \tag{3-9}$$

Similar to the perceptual weighting filter, the synthesis filters 9 and 22 are also updated once every four vectors, and the updates also occur at the third speech vector of every 4-vector adaptation cycle. However, the updates are based on the quantized speech up to the last vector of the previous adaptation cycle. In other words, a delay of two vectors is introduced before the updates take place. This is because the Levinson-Durbin recursion module 50 and the energy table calculator 15 (described later) are computationally intensive. As a result, even though the autocorrelation

of previously quantized speech is available at the first vector of each four vector cycle, computations may require more than one vector worth of time. Therefore, to maintain a basic buffer size of one vector (so as to keep the coding delay low), and to maintain real-time operation, a 2-vector delay in filter updates is introduced in order to facilitate real-time implementation.

3.8 *Backward vector gain adapter*

This adapter updates the excitation gain $\sigma(n)$ for every vector time index n. The excitation gain $\sigma(n)$ is a scaling factor used to scale the selected excitation vector $y(n)$. The adapter 20 takes the gain-scaled excitation vector $e(n)$ as its input, and produces an excitation gain $\sigma(n)$ as its output. Basically, it attempts to "predict" the gain of $e(n)$ based on the gains of $e(n-1)$, $e(n-2)$, ... by using adaptive linear prediction in the logarithmic gain domain. This backward vector gain adapter 20 is shown in more detail in Figure 6/G.728.

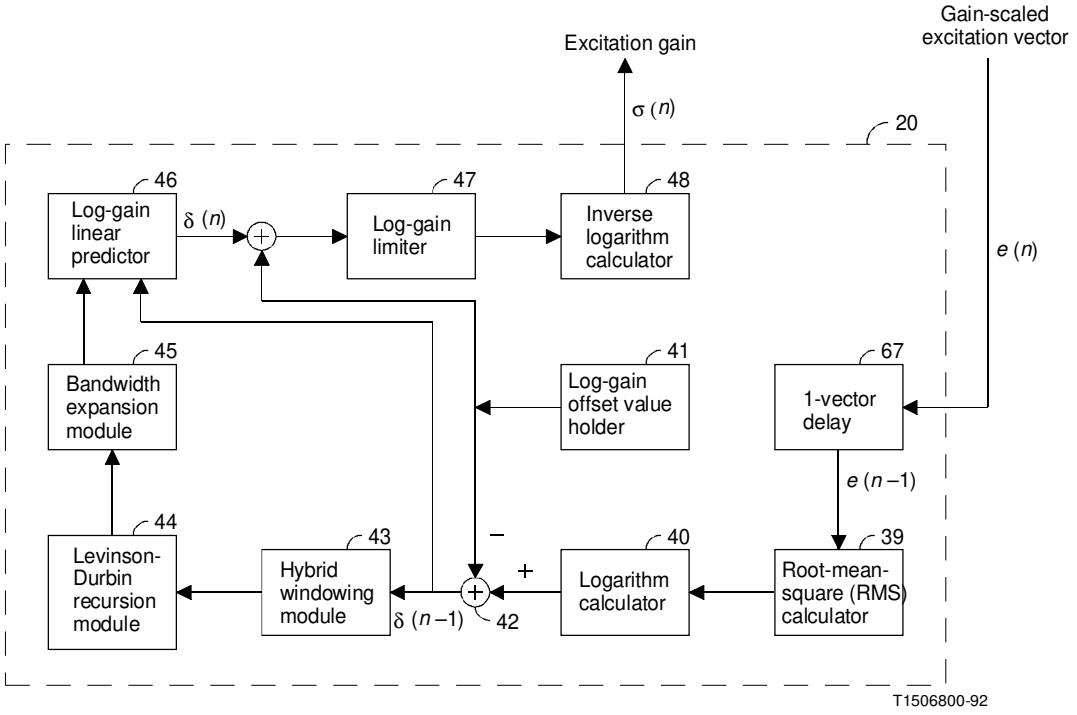


FIGURE 6/G.728

Backward vector gain adapter

Refer to Figure 6/G.728. This gain adapter operates as follows. The 1-vector delay unit 67 makes the previous gain-scaled excitation vector $e(n-1)$ available. The root-mean-square (RMS) calculator 39 then calculates the RMS value of the vector $e(n-1)$. Next, the logarithm calculator 40 calculates the dB value of the RMS of $e(n-1)$, by first computing the base 10 logarithm and then multiplying the result by 20.

In Figure 6/G.728, a log-gain offset value of 32 dB is stored in the log-gain offset value holder 41. This value is meant to be roughly equal to the average excitation gain level (in dB) during voiced speech. The adder 42 subtracts this log-gain offset value from the logarithmic gain produced by the logarithm calculator 40. The resulting offset-removed logarithmic gain $\delta(n-1)$ is then used by the hybrid windowing module 43 and the Levinson-Durbin recursion

module 44. Again, blocks 43 and 44 operate in exactly the same way as blocks 36 and 37 in the perceptual weighting filter adapter module (Figure 4a)/G.728), except that the hybrid window parameters are different and that the signal under analysis is now the offset-removed logarithmic gain rather than the input speech. (Note that only one gain value is produced for every five speech samples.) The hybrid window parameters of block 43 are:

$$M = 10, N = 20, L = 4, \alpha = \left(\frac{3}{4}\right)^{\frac{1}{8}} = 0.96467863$$

The output of the Levinson-Durbin recursion module 44 is the coefficients of a 10-th order linear predictor with a transfer function of

$$\hat{R}(z) = -\sum_{i=1}^{10} \hat{\alpha}_i z^{-i} \tag{3-10}$$

The bandwidth expansion module 45 then moves the roots of this polynomial radially toward the z-plane original in a way similar to the module 51 in Figure 5/G.728. The resulting bandwidth-expanded gain predictor has a transfer function of

$$R(z) = -\sum_{i=1}^{10} \alpha_i z^{-i} \tag{3-11}$$

where the coefficients α_i are computed as

$$\alpha_i = \left(\frac{29}{32}\right)^i \hat{\alpha}_i = (0.90625)^i \hat{\alpha}_i \tag{3-12}$$

Such bandwidth expansion makes the gain adapter (block 20 in Figure 2/G.728) more robust to channel errors. These α_i are then used as the coefficients of the log-gain linear predictor (block 46 of Figure 6/G.728).

This predictor 46 is updated once every four speech vectors, and the updates take place at the second speech vector of every 4-vector adaptation cycle. The predictor attempts to predict $\delta(n)$ based on a linear combination of $\delta(n-1)$, $\delta(n-2)$, ..., $\delta(n-10)$. The predicted version of $\delta(n)$ is denoted as $\hat{\delta}(n)$ and is given by

$$\hat{\delta}(n) = -\sum_{i=1}^{10} \alpha_i \delta(n-i) \tag{3-13}$$

After $\hat{\delta}(n)$ has been produced by the log-gain linear predictor 46, we add back the log-gain offset value of 32 dB stored in 41. The log-gain limiter 47 then checks the resulting log-gain value and clips it if the value is unreasonably large or unreasonably small. The lower and upper limits are set to 0 dB and 60 dB, respectively. The gain limiter output is then fed to the inverse logarithm calculator 48, which reverses the operation of the logarithm calculator 40 and converts the gain from the dB value to the linear domain. The gain limiter ensures that the gain in the linear domain is in between 1 and 1000.

3.9 *Codebook search module*

In Figure 2/G.728, blocks 12 through 18 constitute a codebook search module 24. This module searches through the 1024 candidate codevectors in the excitation VQ codebook 19 and identifies the index of the best codevector which gives a corresponding quantized speech vector that is closest to the input speech vector.

To reduce the codebook search complexity, the 10-bit, 1024-entry codebook is decomposed into two smaller codebooks: a 7-bit "shape codebook" containing 128 independent codevectors and a 3-bit "gain codebook" containing eight scalar values that are symmetric with respect to zero (i.e. one bit for sign, two bits for magnitude). The final output codevector is the product of the best shape codevector (from the 7-bit shape codebook) and the best gain level (from the 3-bit gain codebook). The 7-bit shape codebook table and the 3-bit gain codebook table are given in Annex B.

3.9.1 *Principle of codebook search*

In principle, the codebook search module 24 scales each of the 1024 candidate codevectors by the current excitation gain $\sigma(n)$ and then passes the resulting 1024 vectors one at a time through a cascaded filter consisting of the synthesis filter $F(z)$ and the perceptual weighting filter $W(z)$. The filter memory is initialized to zero each time the module feeds a new codevector to the cascaded filter with transfer function $H(z) = F(z)W(z)$.

The filtering of VQ codevectors can be expressed in terms of matrix-vector multiplication. Let y_j be the j-th codevector in the 7-bit shape codebook, and let g_i be the i-th level in the 3-bit gain codebook. Let $\{h(n)\}$ denote the impulse response sequence of the cascaded filter. Then, when the codevector specified by the codebook indices i and j is fed to the cascaded filter $H(z)$, the filter output can be expressed as

$$\tilde{x}_{ij} = \mathbf{H}\sigma(n) g_i y_j \tag{3-14}$$

where

$$\mathbf{H} = \begin{bmatrix} h(0) & 0 & 0 & 0 & 0 \\ h(1) & h(0) & 0 & 0 & 0 \\ h(2) & h(1) & h(0) & 0 & 0 \\ h(3) & h(2) & h(1) & h(0) & 0 \\ h(4) & h(3) & h(2) & h(1) & h(0) \end{bmatrix} \tag{3-15}$$

The codebook search module 24 searches for the best combination of indices i and j which minimizes the following mean-squared error (MSE) distortion.

$$D = \| x(n) - \tilde{x}_{ij} \|^2 = \sigma^2(n) \| \hat{x}(n) - g_i \mathbf{H} y_j \|^2 \tag{3-16}$$

where $\hat{x}(n) = x(n)/\sigma(n)$ is the gain-normalized VQ target vector. Expanding the terms gives us

$$D = \sigma^2(n) \left[\| \hat{x}(n) \|^2 - 2 g_i \hat{x}^T(n) \mathbf{H} y_j + g_i^2 \| \mathbf{H} y_j \|^2 \right] \tag{3-17}$$

Since the term $\|\hat{x}(n)\|^2$ and the value of $\sigma^2(n)$ are fixed during the codebook search, minimizing D is equivalent to minimizing

$$\hat{D} = -2 g_i p^T(n) y_j + g_i^2 E_j \tag{3-18}$$

where

$$p(n) = \mathbf{H}^T \hat{x}(n) \tag{3-19}$$

and

$$E_j = \| \mathbf{H} y_j \|^2 \tag{3-20}$$

Note that E_j is actually the energy of the j-th filtered shape codevectors and does not depend on the VQ target vector $\hat{x}(n)$. Also note that the shape codevector y_j is fixed, and the matrix **H** only depends on the synthesis filter and the weighting filter, which are fixed over a period of four speech vectors. Consequently, E_j is also fixed over a period of four speech vectors. Based on this observation, when the two filters are updated, we can compute and store the 128 possible energy terms E_j, j = 0, 1, 2, ..., 127 (corresponding to the 128 shape codevectors) and then use these energy terms repeatedly for the codebook search during the next four speech vectors. This arrangement reduces the codebook search complexity.

For further reduction in computation, we can precompute and store the two arrays

$$b_i = 2g_i \tag{3-21}$$

and

$$c_i = g_i^2 \tag{3-22}$$

for i = 0, 1, ..., 7. These two arrays are fixed since g_is are fixed. We can now express $\hat{D}$ as

$$\hat{D} = -b_i P_j + c_i E_j \tag{3-23}$$

where $P_j = p^T(n)y_j$.

Note that once the E_j, b_i, and c_i tables are precomputed and stored, the inner product term $P_j = p^T(n)y_j$, which solely depends on j, takes most of the computation in determining $\hat{D}$. Thus, the codebook search procedure steps through the shape codebook and identifies the best gain index i for each shape codevector y_j.

There are several ways to find the best gain index i for a given shape codevector y_j.

a) The first and the most obvious way is to evaluate the eight possible $\hat{D}$ values corresponding to the eight possible values of i, and then pick the index i which corresponds to the smallest $\hat{D}$. However, this requires two multiplications for each i.

b) A second way is to compute the optimal gain $\hat{g} = P_j/E_j$ first, and then quantize this gain $\hat{g}$ to one of the eight gain levels $\{g_0, ..., g_7\}$ in the 3-bit gain codebook. The best index i is the index of the gain level g_i which is closest to $\hat{g}$. However, this approach requires a division operation for each of the 128 shape codevectors, and division is typically very inefficient to implement using DSP processors.

c) A third approach, which is a slightly modified version of the second approach, is particularly efficient for DSP implementations. The quantization of $\hat{g}$ can be thought of as a series of comparisons between $\hat{g}$ and the "quantizer cell boundaries", which are the mid-points between adjacent gain levels. Let d_i be the mid-point between gain level g_i and g_{i+1} that have the same sign. Then, testing "$\hat{g} < d_i$?" is equivalent to testing "$P_j < d_i E_j$?". Therefore, by using the latter test, we can avoid the division operation and still require only one multiplication for each index i. This is the approach used in the codebook search. The gain quantizer cell boundaries d_is are fixed and can be precomputed and stored in a table. For the eight gain levels, actually only six boundary values d_0, d_1, d_2, d_4, d_5, and d_6 are used.

Once the best indices i and j are identified, they are concatenated to form the output of the codebook search module – a single 10-bit best codebook index.

3.9.2 *Operation of codebook search module*

With the codebook search principle introduced, the operation of the codebook search module 24 is now described below. Refer to Figure 2/G.728. Every time when the synthesis filter 9 and the perceptual weighting filter 10 are updated, the impulse response vector calculator 12 computes the first five samples of the impulse response of the cascaded filter $F(z)W(z)$. To compute the impulse response vector, we first set the memory of the cascaded filter to

zero, then excite the filter with an input sequence {1, 0, 0, 0, 0}. The corresponding five output samples of the filter are $h(0)$, $h(1)$, ..., $h(4)$, which constitute the desired impulse response vector. After this impulse response vector is computed, it will be held constant and used in the codebook search for the following four speech vectors, until the filters 9 and 10 are updated again.

Next, the shape codevector convolution module 14 computes the 128 vectors $\mathbf{H}y_j$, $j = 0, 1, 2, ..., 127$. In other words, it convolves each shape codevector y_j, $j = 0, 1, 2, ..., 127$ with the impulse response sequence $h(0)$, $h(1)$, ..., $h(4)$, where the convolution is only performed for the first five samples. The energies of the resulting 128 vectors are then computed and stored by the energy table calculator 15 according to equation (3-20). The energy of a vector is defined as the sum of the squared value of each vector component.

Note that the computations in blocks 12, 14, and 15 are performed only once every four speech vectors, while the other blocks in the codebook search module perform computations for each speech vector. Also note that the updates of the E_j table is synchronized with the updates of the synthesis filter coefficients. That is, the new E_j table will be used starting from the third speech vector of every adaptation cycle. (Refer to the discussion in § 3.7.)

The VQ target vector normalization module 16 calculates the gain-normalized VQ target vector $\hat{x}(n) = x(n)/\sigma(n)$. In DSP implementations, it is more efficient to first compute $1/\sigma(n)$, and then multiply each component of $x(n)$ by $1/\sigma(n)$.

Next, the time-reversed convolution module 13 computes the vector $p(n) = \mathbf{H}^T\hat{x}(n)$. This operation is equivalent to first reversing the order of the components of $\hat{x}(n)$, then convolving the resulting vector with the impulse response vector, and then reverse the component order of the output again (and hence the name "time-reversed convolution").

Once E_j, b_i, and c_i tables are precomputed and stored, and the vector $p(n)$ is also calculated, then the error calculator 17 and the best codebook index selector 18 work together to perform the following efficient codebook search algorithm:

a) Initialize $\hat{D}_{\min}$ to a number larger than the largest possible value of $\hat{D}$ (or use the largest possible number of the DSPs number representation system).

b) Set the shape codebook index $j = 0$.

c) Compute the inner product $P_j = p^t(n)y_j$.

d) If $p_j < 0$, go to step h) to search through negative gains; otherwise, proceed to step e) to search through positive gains.

e) If $P_j < d_0E_j$, set $i = 0$ and go to step k); otherwise proceed to step f).

f) If $P_j < d_1E_j$, set $i = 1$ and go to step k); otherwise proceed to step g).

g) If $P_j < d_2E_j$, set $i = 2$ and go to step k); otherwise set $i = 3$ and go to step k).

h) If $P_j > d_4E_j$, set $i = 4$ and go to step k); otherwise proceed to step i).

i) If $P_j > d_5E_j$, set $i = 5$ and go to step k); otherwise proceed to step j).

j) If $P_j > d_6E_j$, set $i = 6$; otherwise set $i = 7$.

k) Compute $\hat{D} = -b_iP_j + c_iE_j$.

l) If $\hat{D} < \hat{D}_{\min}$, then set $\hat{D}_{\min} = \hat{D}$, $i_{\min} = i$, and $j_{\min} = j$.

m) If $j < 127$, set $j = j + 1$ and go to step c); otherwise proceed to step n).

n) When the algorithm proceeds to here, all 1024 possible combinations of gains and shapes have been searched through. The resulting $i_{\min}$, and $j_{\min}$ are the desired channel indices for the gain and the shape, respectively. The output best codebook index (10-bit) is the concatenation of these two indices, and the corresponding best excitation codevector is $y(n) = g_{i_{\min}}y_{j_{\min}}$. The selected 10-bit codebook index is transmitted through the communication channel to the decoder.

3.10 *Simulated decoder*

Although the encoder has identified and transmitted the best codebook index so far, some additional tasks have to be performed in preparation for the encoding of the following speech vectors. First, the best codebook index is fed to the excitation VQ codebook to extract the corresponding best codevector $y(n) = g_{i_{\min}} y_{j_{\min}}$. This best codevector is then scaled by the current excitation gain $\sigma(n)$ in the gain stage 21. The resulting gain-scaled excitation vector is $e(n) = \sigma(n)y(n)$.

This vector $e(n)$ is then passed through the synthesis filter 22 to obtain the current quantized speech vector $s_q(n)$. Note that blocks 19 through 23 form a simulated decoder 8. Hence, the quantized speech vector $s_q(n)$ is actually the simulated decoded speech vector when there are no channel errors. In Figure 2/G.728, the backward synthesis filter adapter 23 needs this quantized speech vector $s_q(n)$ to update the synthesis filter coefficients. Similarly, the backward vector gain adapter 20 needs the gain-scaled excitation vector $e(n)$ to update the coefficients of the log-gain linear predictor.

One last task before proceeding to encode the next speech vector is to update the memory of the synthesis filter 9 and the perceptual weighting filter 10. To accomplish this, we first save the memory of filters 9 and 10 which was left over after performing the zero-input response computation described in § 3.5. We then set the memory of filters 9 and 10 to zero and close the switch 5, i.e. connect it to node 7. Then, the gain-scaled excitation vector $e(n)$ is passed through the two zero-memory filters 9 and 10. Note that since $e(n)$ is only five samples long and the filters have zero memory, the number of multiply-adds only goes up from 0 to 4 for the five-sample period. This is a significant saving in computation since there would be 70 multiply-adds per sample if the filter memory were not zero. Next, we add the saved original filter memory back to the newly established filter memory after filtering $e(n)$. This in effect adds the zero-input responses to the zero-state responses of the filters 9 and 10. This results in the desired set of filter memory which will be used to compute the zero-input response during the encoding of the next speech vector.

Note that after the filter memory update, the top five elements of the memory of the synthesis filter 9 are exactly the same as the components of the desired quantized speech vector $s_q(n)$. Therefore, we can actually omit the synthesis filter 22 and obtain $s_q(n)$ from the updated memory of the synthesis filter 9. This means an additional saving of 50 multiply-adds per sample.

The encoder operation described so far specifies the way to encode a single input speech vector. The encoding of the entire speech waveform is achieved by repeating the above operation for every speech vector.

3.11 *Synchronization and in-band signalling*

In the above description of the encoder, it is assumed that the decoder knows the boundaries of the received 10-bit codebook indices and also knows when the synthesis filter and the log-gain predictor need to be updated (recall that they are updated once every four vectors). In practice, such synchronization information can be made available to the decoder by adding extra synchronization bits on top of the transmitted 16 kbit/s bit stream. However, in many applications there is a need to insert synchronization or in-band signalling bits as part of the 16 kbit/s bit stream. This can be done in the following way. Suppose a synchronization bit is to be inserted once every N speech vectors; then, for every N-th input speech vector, we can search through only half of the shape codebook and produce a 6-bit shape codebook index. In this way, we rob one bit out of every N-th transmitted codebook index and insert a synchronization or signalling bit instead.

It is important to note that we cannot arbitrarily rob one bit out of an already selected 7-bit shape codebook index, instead, the encoder has to know which speech vectors will be robbed one bit and then search through only half of the codebook for those speech vectors. Otherwise, the decoder will not have the same decoded excitation codevectors for those speech vectors.

Since the coding algorithm has a basic adaptation cycle of four vectors, it is reasonable to let N be a multiple of 4 so that the decoder can easily determine the boundaries of the encoder adaptation cycles. For a reasonable value of N (such as 16, which corresponds to a 10 milliseconds bit robbing period), the resulting degradation in speech quality is essentially negligible. In particular, we have found that a value of $N = 16$ results in little additional distortion. The rate of this bit robbing is only 100 bits/s.

If the above procedure is followed, we recommend that when the desired bit is to be a 0, only the first half of the shape codebook be searched, i.e. those vectors with indices 0 to 63. When the desired bit is a 1, then the second half of the codebook is searched and the resulting index will be between 64 and 127. The significance of this choice is that the desired bit will be the leftmost bit in the codeword, since seven bits for the shape codevector precede the three bits for the sign and gain codebook. We further recommend that the synchronization bit be robbed from the last vector in a cycle of four vectors. Once it is detected, the next codeword received can begin the new cycle of codevectors.

Although we state that synchronization causes very little distortion, we note that no formal testing has been done on hardware which contained this synchronization strategy. Consequently, the amount of the degradation has not been measured.

However, we specifically recommend against using the synchronization bit for synchronization in systems in which the coder is turned on and off repeatedly. For example, a system might use a speech activity detector to turn off the coder when no speech were present. Each time the encoder was turned on, the decoder would need to locate the synchronization sequence. At 100 bit/s, this would probably take several hundred milliseconds. In addition, time must be allowed for the decoder state to track the encoder state. The combined result would be a phenomena known as front-end clipping in which the beginning of the speech utterance would be lost. If the encoder and decoder are both started at the same instant as the onset of speech, then no speech will be lost. This is only possible in systems using external signalling for the start-up times and external synchronization.

4 LD-CELP decoder principles

Figure 3/G.728 is a block schematic of the LD-CELP decoder. A functional description of each block is given in the following sections.

4.1 *Excitation VQ codebook*

This block contains an excitation VQ codebook (including shape and gain codebooks) identical to the codebook 19 in the LD-CELP encoder. It uses the received best codebook index to extract the best codevector $y(n)$ selected in the LD-CELP encoder.

4.2 *Gain scaling unit*

This block computes the scaled excitation vector $e(n)$ by multiplying each component of $y(n)$ by the gain $\sigma(n)$.

4.3 *Synthesis filter*

This filter has the same transfer function as the synthesis filter in the LD-CELP encoder (assuming error-free transmission). It filters the scaled excitation vector $e(n)$ to produce the decoded speech vector $s_d(n)$. Note that in order to avoid any possible accumulation of round-off errors during decoding, sometimes it is desirable to exactly duplicate the procedures used in the encoder to obtain $s_q(n)$. If this is the case, and if the encoder obtains $s_q(n)$ from the updated memory of the synthesis filter 9, then the decoder should also compute $s_d(n)$ as the sum of the zero-input response and the zero-state response of the synthesis filter 32, as is done in the encoder.

4.4 *Backward vector gain adapter*

The function of this block is described in § 3.8.

4.5 *Backward synthesis filter adapter*

The function of this block is described in § 3.7.

4.6 *Postfilter*

This block filters the decoded speech to enhance the perceptual quality. This block is further expanded in Figure 7/G.728 to show more details. Refer to Figure 7/G.728. The postfilter basically consists of three major parts: long-term postfilter 71, short-term postfilter 72, and output gain scaling unit 77. The other four blocks in Figure 7/G.728 are just to calculate the appropriate scaling factor for use in the output gain scaling unit 77.

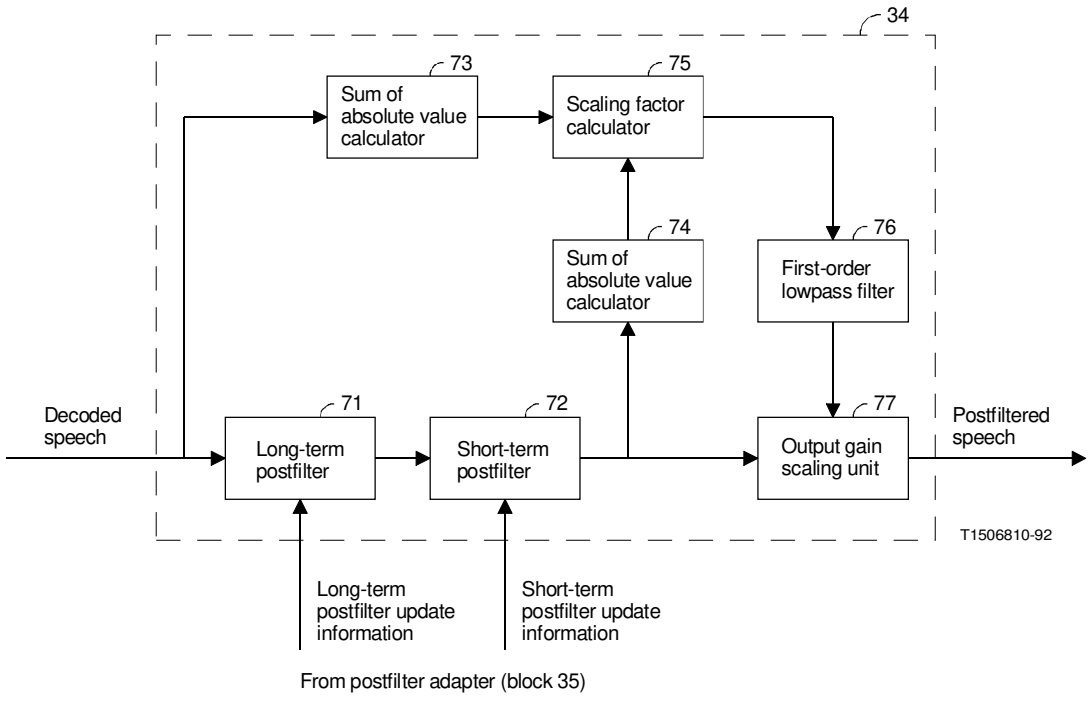


FIGURE 7/G.728

Postfilter block schematic

The *long-term postfilter* 71, sometimes called the *pitch postfilter*, is a comb filter with its spectral peaks located at multiples of the fundamental frequency (or *pitch frequency*) of the speech to be postfiltered. The reciprocal of the fundamental frequency is called the *pitch period*. The *pitch period* can be extracted from the decoded speech using a pitch detector (or pitch extractor). Let p be the fundamental pitch period (in samples) obtained by a pitch detector, then the transfer function of the long-term postfilter can be expressed as

$$H_l(z) = g_l(1 + b\,z^{-p}) \qquad (4\text{-}1)$$

where the coefficients g_l, b and the pitch period p are updated once every four speech vectors (an adaptation cycle) and the actual updates occur at the third speech vector of each adaptation cycle. For convenience, we will, from now on, call an adaptation cycle a *frame*. The derivation of g_l, b, and p will be described later in § 4.7.

The short-term postfilter 72 consists of a 10th-order pole-zero filter in cascade with a first-order all-zero filter. The 10th-order pole-zero filter attenuates the frequency components between formant peaks, while the first-order all-zero filter attempts to compensate for the spectral tilt in the frequency response of the 10th-order pole-zero filter.

Let $\tilde{a}_i$, $i = 1, 2, ..., 10$ be the coefficients of the 10th-order LPC predictor obtained by backward LPC analysis of the decoded speech, and let k_1 be the first reflection coefficient obtained by the same LPC analysis. Then, both $\tilde{a}_i$ and k_1 can be obtained as by-products of the 50th-order backward LPC analysis (block 50 in Figure 5/G.728). All we have to do is to stop the 50th-order Levinson-Durbin recursion at order 10, copy k_1 and $\tilde{a}_1$, $\tilde{a}_2$, ..., $\tilde{a}_{10}$, and then resume the Levinson-Durbin recursion from order 11 to order 50. The transfer function of the short-term postfilter is

$$H_s(z) = \frac{1 - \sum_{i=1}^{10} \bar{b}_i z^{-i}}{1 - \sum_{i=1}^{10} \bar{a}_i z^{-i}} [1 + \mu z^{-1}] \tag{4-2}$$

where

$$\bar{b}_i = \tilde{a}_i (0.65)^i; \; i = 1, 2, \ldots, 10 \tag{4-3}$$

$$\bar{a}_i = \tilde{a}_i (0.75)^i; \; i = 1, 2, \ldots, 10 \tag{4-4}$$

and

$$\mu = (0.15)\, k_1 \tag{4-5}$$

The coefficients $\bar{a}_i$; $\bar{b}_i$ and μ are also updated once a frame, but the updates take place at the first vector of each frame (i.e. as soon as $\tilde{a}_i$ becomes available).

In general, after the decoded speech is passed through the long-term postfilter and the short-term postfilter, the filtered speech will not have the same power level as the decoded (unfiltered) speech. To avoid occasional large gain excursions, it is necessary to use automatic gain control to force the postfiltered speech to have roughly the same power as the unfiltered speech. This is done by blocks 73 through 77.

The sum of absolute value calculator 73 operates vector-by-vector. It takes the current decoded speech vector $s_d(n)$ and calculates the sum of the absolute values of its five vector components. Similarly, the sum of absolute value calculator 74 performs the same type of calculation, but on the current output vector $s_f(n)$ of the short-term postfilter. The scaling factor calculator 75 then divides the output value of block 73 by the output value of block 74 to obtain a scaling factor for the current $s_f(n)$ vector. This scaling factor is then filtered by a first-order lowpass filter 76 to get a separate scaling factor for each of the five components of $s_f(n)$. The first-order lowpass filter 76 has a transfer function of $0.01/(1 - 0.99z^{-1})$. The lowpass filtered scaling factor is used by the output gain scaling unit 77 to perform sample-by-sample scaling of the short-term postfilter output. Note that since the scaling factor calculator 75 only generates one scaling factor per vector, it would have a staircase effect on the sample-by-sample scaling operation of block 77 if the lowpass filter 76 were not present. The lowpass filter 76 effectively smooths out such a staircase effect.

4.6.1 *Non-speech operation*

CCITT objective test results indicate that for some non-speech signals, the performance of the coder is improved when the adaptive postfilter is turned off. Since the input to the adaptive postfilter is the output of the synthesis filter, this signal is always available. In an actual implementation this unfiltered signal shall be output when the switch is set to disable the postfilter.

4.7 *Postfilter adapter*

This block calculates and updates the coefficients of the postfilter once a frame. This postfilter adapter is further expanded in Figure 8/G.728.

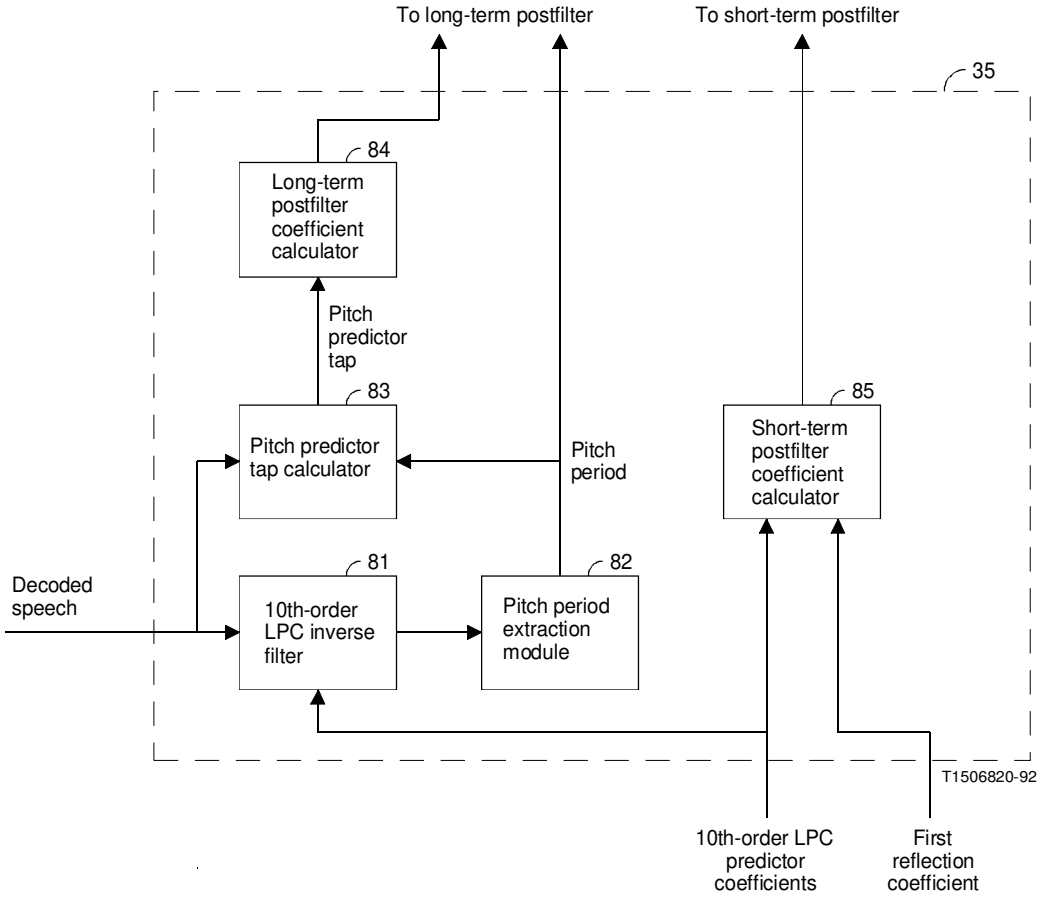


FIGURE 8/G.728

Postfilter adapter block schematic

Refer to Figure 8/G.728. The 10th-order LPC inverse filter 81 and the pitch period extraction module 82 work together to extract the pitch period from the decoded speech. In fact, any pitch extractor with reasonable performance (and without introducing additional delay) may be used here. What we described here is only one possible way of implementing a pitch extractor.

The 10th-order LPC inverse filter 81 has a transfer function of

$$\tilde{A}(z) = 1 - \sum_{i=1}^{10} \tilde{a}_i z^{-i} \qquad (4\text{-}6)$$

where the coefficients $\tilde{a}_i$ are supplied by the Levinson-Durbin recursion module (block 50 of Figure 5/G.728) and are updated at the first vector of each frame. This LPC inverse filter takes the decoded speech as its input and produces the LPC prediction residual sequence $\{d(k)\}$ as its output. We use a pitch analysis window size of 100 samples and a range

of pitch period from 20 to 140 samples. The pitch period extraction module 82 maintains a long buffer to hold the last 240 samples of the LPC prediction residual. For indexing convenience, the 240 LPC residual samples stored in the buffer are indexed as $d(-139)$, $d(-138)$, ..., $d(100)$.

The pitch period extraction module 82 extracts the pitch period once a frame, and the pitch period is extracted at the third vector of each frame. Therefore, the LPC inverse filter output vectors should be stored into the LPC residual buffer in a special order: the LPC residual vector corresponding to the fourth vector of the last frame is stored as $d(81)$, $d(82)$, ..., $d(85)$, the LPC residual of the first vector of the current frame is stored as $d(86)$, $d(87)$, ..., $d(90)$, the LPC residual of the second vector of the current frame is stored as $d(91)$, $d(92)$, ..., $d(95)$, and the LPC residual of the third vector is stored as $d(96)$, $d(97)$, ..., $d(100)$. The samples $d(-139)$, $d(-138)$, ..., $d(80)$ are simply the previous LPC residual samples arranged in the correct time order.

Once the LPC residual buffer is ready, the pitch period extraction module 82 works in the following way. First, the last 20 samples of the LPC residual buffer [$d(81)$ through $d(100)$] are lowpass filtered at 1 kHz by a third-order elliptic filter (coefficients given in Annex D) and then 4:1 decimated (i.e. down-sampled by a factor of 4). This results in five lowpass filtered and decimated LPC residual samples, denoted $\bar{d}(21)$, $\bar{d}(22)$, ..., $\bar{d}(25)$, which are stored as the last five samples in a decimated LPC residual buffer. Besides these five samples, the other 55 samples $\bar{d}(-34)$, $\bar{d}(-33)$, ..., $\bar{d}(20)$ in the decimated LPC residual buffer are obtained by shifting previous frames of decimated LPC residual samples. The i-th correlation of the decimated LPC residual samples are then computed as

$$\rho(i) = \sum_{n=1}^{25} \bar{d}(n)\,\bar{d}(n-i) \tag{4-7}$$

for time lags $i = 5, 6, 7, ..., 35$ (which correspond to pitch periods from 20 to 140 samples). The time lag τ which gives the largest of the 31 calculated correlation values is then identified. Since this time lag τ is the lag in the 4:1 decimated residual domain, the corresponding time lag which gives the maximum correlation in the original undecimated residual domain should lie between $4\tau-3$ and $4\tau+3$. To get the original time resolution, we next use the undecimated LPC residual buffer to compute the correlation of the undecimated LPC residual

$$C(i) = \sum_{k=1}^{100} d(k)\,d(k-i) \tag{4-8}$$

for seven lags $i = 4\tau-3, 4\tau-2, ..., 4\tau+3$. Out of the seven time lags, the lag p_0 that gives the largest correlation is identified.

The time lag p_0 found this way may turn out to be a multiple of the true fundamental pitch period. What we need in the long-term postfilter is the true fundamental pitch period, not any multiple of it. Therefore, we need to do more processing to find the fundamental pitch period. We make use of the fact that we estimate the pitch period quite frequency – once every 20 speech samples. Since the pitch period typically varies between 20 and 140 samples, our frequent pitch estimation means that, at the beginning of each talk spurt, we will first get the fundamental pitch period before the multiple pitch periods have a chance to show up in the correlation peak-picking process described above. From there on, we will have a chance to lock on to the fundamental pitch period by checking to see if there is any correlation peak in the neighbourhood of the pitch period of the previous frame.

Let $\hat{p}$ be the pitch period of the previous frame. If the time lag p_0 obtained above is not in the neighbourhood of $\hat{p}$, then we also evaluate equation (4-8) for $i = \hat{p}-6, \hat{p}-5, ..., \hat{p}+5, \hat{p}+6$. Out of these 13 possible time lags, the time

lag p_1 that gives the largest correlation is identified. We then test to see if this new lag p_1 should be used as the output pitch period of the current frame. First, we compute

$$\beta_0 = \frac{\sum_{k=1}^{100} d(k)\, d(k-p_0)}{\sum_{k=1}^{100} d(k-p_0)\, d(k-p_0)} \tag{4-9}$$

which is the optimal tap weight of a single-tap pitch predictor with a lag of p_0 samples. The value of β_0 is then clamped between 0 and 1. Next, we also compute

$$\beta_1 = \frac{\sum_{k=1}^{100} d(k)\, d(k-p_1)}{\sum_{k=1}^{100} d(k-p_1)\, d(k-p_1)} \tag{4-10}$$

which is the optimal tap weight of a single-tap pitch predictor with a lag of p_1 samples. The value of β_1 is then also clamped between 0 and 1. Then, the output pitch period p of block 82 is given by

$$p = \begin{cases} p_0 & \text{if} \quad \beta_1 \le 0.4\,\beta_0 \\ p_1 & \text{if} \quad \beta_1 > 0.4\,\beta_0 \end{cases} \tag{4-11}$$

After the pitch period extraction module 82 extracts the pitch period p, the pitch predictor tap calculator 83 then calculates the optimal tap weight of a single-tap pitch predictor for the decoded speech. The pitch predictor tap calculator 83 and the long-term postfilter 71 share a long buffer of decoded speech samples. This buffer contains decoded speech samples $s_d(-239)$, $s_d(-238)$, $s_d(-237)$, ..., $s_d(4)$, $s_d(5)$, where $s_d(1)$ through $s_d(5)$ correspond to the current vector of decoded speech. The long-term postfilter 71 uses this buffer as the delay unit of the filter. On the other hand, the pitch predictor tap calculator 83 uses this buffer to calculate

$$\beta = \frac{\sum_{k=-99}^{0} s_d(k)\, s_d(k-p)}{\sum_{k=-99}^{0} s_d(k-p)\, s_d(k-p)} \tag{4-12}$$

The long-term postfilter coefficient calculator 84 then takes the pitch period p and the pitch predictor tap β and calculates the long-term postfilter coefficients b and g_l as follows

$$b = \begin{cases} 0 & \text{if} \quad \beta < 0.6 \\ 0.15\,\beta & \text{if} \quad 0.6 \le \beta \le 1 \\ 0.15 & \text{if} \quad \beta > 1 \end{cases} \tag{4-13}$$

$$g_l = \frac{1}{1 + b}$$

In general, the closer β is to unity, the more periodic the speech waveform is. As can be seen in equations (4-13) and (4-14), if $\beta < 0.6$, which roughly corresponds to unvoiced or transition regions of speech, then $b = 0$ and $g_l = 1$, and the long-term postfilter transfer function becomes $H_l(z) = 1$, which means the filtering operation of the long-term postfilter is totally disabled. On the other hand, if $0.6 \leq \beta \leq 1$, the long-term postfilter is turned on, and the degree of comb filtering is determined by β. The more periodic the speech waveform, the more comb filtering is performed. Finally, if $\beta > 1$, then b is limited to 0.15; this is to avoid too much comb filtering. The coefficient g_l is a scaling factor of the long-term postfilter to ensure that the voiced regions of speech waveforms do not get amplified relative to the unvoiced or transition regions. (If g_l were held constant at unity, then after the long-term postfiltering, the voiced regions would be amplified by a factor of $1+b$ roughly. This would make some consonants, which correspond to unvoiced and transition regions, sound unclear or too soft.)

The short-term postfilter coefficient calculator 85 calculates the short-term postfilter coefficients $\bar{a}_i$, $\bar{b}_i$, and μ at the first vector of each frame according to equations (4-3), (4-4), and (4-5).

4.8 *Output PCM format conversion*

This block converts the five components of the decoded speech vector into five corresponding *A*-law or μ-law PCM samples and output these five PCM samples sequentially at 125 μs time intervals. Note that if the internal linear PCM format has been scaled as described in § 3.1.1, the inverse scaling must be performed before conversion to *A*-law or μ-law PCM.

5 Computational details

This section provides the computational details for each of the LD-CELP encoder and decoder elements. Subsections 5.1 and 5.2 list the names of coder parameters and internal processing variables which will be referred to in later sections. The detailed specification of each block in Figure 2/G.728 through Figure 6/G.728 is given in § 5.3 through the end of § 5. To encode and decode an input speech vector, the various blocks of the encoder and the decoder are executed in an order which roughly follows the sequence from § 5.3 to the end.

5.1 *Description of basic coder parameters*

The names of the basic coder parameters are defined in Table 1/G.728. In Table 1/G.728, the first column gives the names of coder parameters which will be used in later detailed description of the LD-CELP algorithm. If a parameter has been referred to in § 3 or 4 but was represented by a different symbol, that equivalent symbol will be given in the second column for easy reference. Each coder parameter has a fixed value which is determined in the coder design stage. The third column shows these fixed parameter values, and the fourth column is a brief description of the coder parameters.

5.2 *Description of internal variables*

The internal processing variables of LD-CELP are listed in Table 2/G.728, which has a layout similar to Table 1/G.728. The second column shows the range of index in each variable array. The fourth column gives the recommended initial values of the variables. The initial values of some arrays are given in Annexes A, B or C. It is recommended (although not required) that the internal variables be set to their initial values when the encoder or decoder just starts running, or whenever a reset of coder states is needed (such as in DCME applications). These initial values ensure that there will be no glitches right after start-up or resets.

Note that some variable arrays can share the same physical memory locations to save memory space, although they are given different names in the tables to enhance clarity.

TABLE 1/G.728

Basic coder parameters of LD-CELP

Name	Equivalent symbol	Value	Description
AGCFAC		0.99	AGC adaptation speed controlling factor
FAC	λ	253/256	Bandwidth expansion factor of synthesis filter
FACGP	λ_g	29/32	Bandwidth expansion factor of log-gain predictor
DIMINV		0.2	Reciprocal of vector dimension
IDIM		5	Vector dimension (excitation block size)
GOFF		32	Log-gain offset value
KPDELTA		6	Allowed deviation from previous pitch period
KPMIN		20	Minimum pitch period (samples)
KPMAX		140	Maximum pitch period (samples)
LPC		50	Synthesis filter order
LPCLG		10	Log-gain predictor order
LPCW		10	Perceptual weighting filter order
NCWD		128	Shape codebook size (number of codevectors)
NFRSZ		20	Frame size (adaptation cycle size in samples)
NG		8	Gain codebook size (number of gain levels)
NONR		35	Number of non-recursive window samples for synthesis filter
NONRLG		20	Number of non-recursive window samples for log-gain predictor
NONRW		30	Number of non-recursive window samples for weighting filter
NPWSZ		100	Pitch analysis window size (samples)
NUPDATE		4	Predictor update period (in terms of vectors)
PPFTH		0.6	Tap threshold for turning off pitch postfilter
PPFZCF		0.15	Pitch postfilter zero controlling factor
SPFPCF		0.75	Short-term postfilter pole controlling factor
SPFZCF		0.65	Short-term postfilter zero controlling factor
TAPTH		0.4	Tap threshold for fundamental pitch replacement
TILTF		0.15	Spectral tilt compensation controlling factor
WNCF		257/256	White noise correction factor
WPCF	γ_2	0.6	Pole controlling factor of perceptual weighting filter
WZCF	γ_1	0.9	Zero controlling factor of perceptual weighting filter

As mentioned in earlier sections, the processing sequence has a basic adaptation cycle of four speech vectors. The variable ICOUNT is used as the vector index. In other words, ICOUNT = n when the encoder or decoder is processing the n-th speech vector in an adaptation cycle.

It should be noted that, for the convenience of Levinson-Durbin recursion, the first element of A, ATMP, AWP, AWZ, and GP arrays are always 1 and never get changed, and, for $i \geq 2$, the i-th elements are the $(i - 1)$-th elements of the corresponding symbols in § 3.

In the following sections, the asterisk (*) denotes arithmetic multiplication.

TABLE 2/G.728

LD-CELP internal processing variables

Name	Array index range	Equivalent symbol	Initial value	Description
A	1 to LPC+1	$-a_{i-1}$	1,0,0,...	Synthesis filter coefficients
AL	1 to 3		Annex D	1 kHz lowpass filter denominator coefficients
AP	1 to 11	$-\bar{a}_{i-1}$	1,0,0,...	Short-term postfilter denominator coefficients
APF	1 to 11	$-\tilde{a}_{i-1}$	1,0,0,...	10th-order LPC filter coefficients
ATMP	1 to LPC+1	$-\mathrm{a}_{i-1}$		Temporary buffer for synthesis filter coefficients
AWP	1 to LPCW+1		1,0,0,...	Perceptual weighting filter denominator coefficients
AWZ	1 to LPCW+1		1,0,0,...	Perceptual weighting filter numerator coefficients
AWZTMP	1 to LPCW+1		1,0,0,...	Temporary buffer for weighting filter coefficients
AZ	1 to 11	$-\bar{b}_{i-1}$	1,0,0,...	Short-term postfilter numerator coefficients
B	1	b	0	Long-term postfilter coefficients
BL	1 to 4		Annex D	1 kHz lowpass filter numerator coefficients
DEC	–34 to 25	$\bar{d}(n)$	0,0,...,0	4:1 decimated LPC prediction residual
D	–139 to 100	$d(k)$	0,0,...,0	LPC prediction residual
ET	1 to IDIM	$e(n)$	0,0,...,0	Gain-scaled excitation vector
FACV	1 to LPC+1	λ^{i-1}	Annex C	Synthesis filter BW broadening vector
FACGPV	1 to LPCLG+1	$\lambda_g{}^{i-1}$	Annex C	Gain predictor BW broadening vector
G2	1 to NG	b_i	Annex B	Two times gain levels in gain codebook
GAIN	1	$\sigma(n)$		Excitation gain
GB	1 to NG–1	d_i	Annex B	Mid-point between adjacent gain levels
GL	1	g_l	1	Long-term postfilter scaling factor
GP	1 to LPCLG+1	$-\alpha_{i-1}$	1,–1,0,0,...	Log-gain linear predictor coefficients
GPTMP	1 to LPCLG+1	$-\alpha_{i-1}$		Temporary array for log-gain linear predictor coefficients
GQ	1 to NG	g_i	Annex B	Gain levels in the gain codebook
GSQ	1 to NG	c_i	Annex B	Squares of gain levels in gain codebook

TABLE 2/G.728 (*cont.*)

Name	Array index range	Equivalent symbol	Initial value	Description
GSTATE	1 to LPCLG	$\delta(n)$	–32,–32,...,–32	Memory of the log-gain linear predictor
GTMP	1 to 4		–32,–32,–32,–32	Temporary log-gain buffer
H	1 to IDIM	$h(n)$	1,0,0,0,0	Impulse response vector of $F(z)W(z)$
ICHAN	1			Best codebook index to be transmitted
ICOUNT	1			Speech vector counter (indexed from 1 to 4)
IG	1	i		Best 3-bit gain codebook index
IP	1		IPINIT [b]	Address pointer to LPC prediction residual
IS	1	j		Best 7-bit shape codebook index
KP	1	p		Pitch period of the current frame
KP1	1	$\hat{p}$	50	Pitch period of the previous frame
PN	1 to IDIM	$p(n)$		Correlation vector for codebook search
PTAP	1	β		Pitch predictor tap computed by block 83
R	1 to NR+1 [a]			Autocorrelation coefficients
RC	1 to NR [a]			Reflection coefficients, also as a scratch array
RCTMP	1 to LPC			Temporary buffer for reflection coefficients
REXP	1 to LPC+1		0,0,...,0	Recursive part of autocorrelation, synthesis filter
REXPLG	1 to LPCLG+1		0,0,...,0	Recursive part of autocorrelation, log-gain predictor
REXPW	1 to LPCW+1		0,0,...,0	Recursive part of autocorrelation, weighting filter
RTMP	1 to LPC+1			Temporary buffer for autocorrelation coefficients
S	1 to IDIM	$s(n)$	0,0,...,0	Uniform PCM input speech vector
SB	1 to 105		0,0,...,0	Buffer for previously quantized speech
SBLG	1 to 34		0,0,...,0	Buffer for previous log-gain
SBW	1 to 60		0,0,...,0	Buffer for previous input speech
SCALE	1			Unfiltered postfilter scaling factor

TABLE 2/G.728 (*Cont.*)

Name	Array index range	Equivalent symbol	Initial value	Description
SCALEFIL	1		1	Lowpass filtered postfilter scaling factor
SD	1 to IDIM	$s_d(k)$		Decoded speech buffer
SPF	1 to IDIM			Postfiltered speech vector
SPFPCFV	1 to 11	$SPFPCF^{i-1}$	Annex C	Short-term postfilter pole controlling vector
SPFZCFV	1 to 11	$SPFZCF^{i-1}$	Annex C	Short-term postfilter zero controlling vector
SO	1	$s_o(k)$		A-law or μ-law PCM input speech sample
SU	1	$s_u(k)$		Uniform PCM input speech sample
ST	–239 to IDIM	$s_q(n)$	0,0,...,0	Quantized speech vector
STATELPC	1 to LPC		0,0,...,0	Synthesis filter memory
STLPCI	1 to 10		0,0,...,0	LPC inverse filter memory
STLPF	1 to 3		0,0,0	1 kHz lowpass filter memory
STMP	1 to 4∗IDIM		0,0,...,0	Buffer for perceptually weighted filter hybrid window
STPFFIR	1 to 10		0,0,...,0	Short-term postfilter memory, all-zero section
STPFIIR	10		0,0,...,0	Short-term postfilter memory, all-pole section
SUMFIL	1			Sum of absolute value of postfiltered speech
SUMUNFIL	1			Sum of absolute value of decoded speech
SW	1 to IDIM	$v(n)$		Perceptually weighted speech vector
TARGET	1 to IDIM	$\hat{x}(n)$; $x(n)$		VQ target vector (gain-normalized)
TEMP	1 to IDIM			Scratch array for temporary working space
TILTZ	1	μ	0	Short-term postfilter tilt-compensation coefficients
WFIR	1 to LPCW		0,0,...,0	Memory of weighting filter 4, all-zero portion
WIIR	1 to LPCW		0,0,...,0	Memory of weighting filter 4, all-pole portion
WNR	1 to 105	$w_m(k)$	Annex A	Window function for synthesis filter
WNRLG	1 to 34	$w_m(k)$	Annex A	Window function for log-gain predictor
WNRW	1 to 60	$w_m(k)$	Annex A	Window function for weighting filter

TABLE G-728 (*cont.*)

Name	Array index range	Equivalent symbol	Initial value	Description
WPCFV	1 to LPCW+1	γ_2^{i-1}	Annex C	Perceptual weighting filter pole controlling vector
WS	1 to 105			Work space array for intermediate variables
WZCFV	1 to LPCW+1	γ_1^{i-1}	Annex C	Perceptual weighting filter zero controlling vector
Y	1 to IDIM∗NCWD	y_j	Annex B	Shape codebook array
Y2	1 to NCWD	E_j	Energy of y_j	Energy of convolved shape codevector
YN	1 to IDIM	$y(n)$		Quantized excitation vector
ZIRWFIR	1 to LPCW		0,0,...,0	Memory of weighting filter 10, all-zero portion
ZIRWIIR	1 to LPCW		0,0,...,0	Memory of weighting filter 10, all-pole portion

a) NR = Max(LPCW,LPCLG)>IDIM.

b) IPINIT = NPWSZ–NFRSZ+IDIM.

Note – The asterisk (∗) denotes arithmetic multiplication.

5.3 *Input PCM format conversion (block 1)*

Input: SO

Output: SU

Function: Convert A-law or μ-law or 16-bit linear input sample to uniform PCM sample.

Since the operation of this block is completely defined in Recommendations G.721 or G.711, we will not repeat it here. However, some scaling may be necessary to conform to this description's specification of an input range of –4 4095 to +4 095 (see § 3.1.1).

5.4 *Vector buffer (block 2)*

Input: SU

Output: S

Function: Buffer five consecutive uniform PCM speech samples to form a single 5-dimensional speech vector.

5.5 *Adapter for perceptual weighting filter (block 3, Figure 4a)/G.728)*

The three blocks (36, 37 and 38) in Figure 4a)/G.728 are now specified in detail below.

HYBRID WINDOWING MODULE (block 36)

Input: STMP

Output: R

Function: Apply the hybrid window to input speech and compute autocorrelation coefficients.

The operation of this module is now described below, using a "Fortran-like" style, with loop boundaries indicated by indentation and comments on the right-hand side of "|". The following algorithm is to be used once every adaptation cycle (20 samples). The STMP array holds 4-consecutive input speech vectors up to the second speech vector of the current adaptation cycle. That is, STMP(1) through STMP(5) is the third input speech vector of the previous adaptation cycle (zero initially), STMP(6) through STMP(10) is the fourth input speech vector of the previous adaptation cycle (zero initially), STMP(11) through STMP(15) is the first input speech vector of the current adaptation cycle, and STMP(16) through STMP(20) is the second input speech vector of the current adaptation cycle.

```
N1 = LPCW + NFRSZ                                  | Compute some constants (can be
N2 = LPCW + NONRW                                  | precomputed and stored in memory)
N3 = LPCW + NFRSZ + NONRW

For   N = 1,2,..,N2, do the next line
      SBW(N) = SBW(N + NFRSZ)                      | Shift the old signal buffer

For   N = 1,2,..,NFRSZ, do the next line           | Shift in the new signal
      SBW(N2 + N) = STMP(N)                        | SBW(N3) is the newest sample

K = 1
For   N = N3,N3 — 1,..,3,2,1, do the next two lines
      WS(N) = SBW(N) * WNRW(K)                     | Multiply the window function
      K = K + 1

For   I = 1,2,..,LPCW + 1, do the next four lines
      TMP = 0
      For  N = LPCW + 1,LPCW + 2,..,N1, do the next line
           TMP = TMP + WS(N) * WS(N + 1 — I)
      REXPW(I) = (1/2) * REXPW(I) + TMP            | Update the recursive component

For   I = 1,2,..,LPCW + 1, do the next three lines
      R(I) = REXPW(I)
      For  N = N1 + 1,N1 + 2,..,N3, do the next line
           R(I) = R(I) + WS(N) * WS(N + 1 — I)     | Add the non-recursive component

R(1) = R(1) * WNCF                                 | White noise correction
```

LEVINSON-DURBIN RECURSION MODULE (block 37)

Input: R (output of block 36)

Output: AWZTMP

Function: Convert autocorrelation coefficients to linear predictor coefficients.

This block is executed once every 4-vector adaptation cycle. It is done at ICOUNT = 3 after the processing of block 36 has finished. Since the Levinson-Durbin recursion is well-known prior art, the algorithm is given below without explanation.

```
If R(LPCW + 1) = 0, go to LABEL                          | Skip if zero

If R(1) ≤ 0, go to LABEL                                 | Skip if zero signal
                                                         |
RC(1) = R(2)/R(1)                                        |
AWZTMP(1) = 1                                            |
AWZTMP(2) = RC(1)                                        | First-order predictor
ALPHA = R(1) + R(2) * RC(1)                              |
If ALPHA ≤ 0, go to LABEL                                | Abort if ill-conditioned

For  MINC = 2,3,4,..,LPCW, do the following:
     SUM = 0
     For  IP = 1,2,3,..,MINC, do the next two lines
          N1 = MINC — IP + 2
          SUM = SUM + R(N1) * AWZTMP(IP)
                                                         |
     RC(MINC) = —SUM/ALPHA                               | Reflection coefficients
     MH = MINC/2 + 1                                     |
     For  IP = 2,3,4,..,MH, do the next four lines
          IB = MINC — IP + 2
          AT = AWZTMP(IP) + RC(MINC) * AWZTMP(IB)        |
          AWZTMP(IB) = AWZTMP(IB) + RC(MINC) AWZTMP(IP)  | Predictor coefficients
          AWZTMP(IP) = AT                                |

     AWZTMP(MINC + 1) = RC(MINC)                         |
     ALPHA = ALPHA + RC(MINC) * SUM                      | Prediction residual energy
     If Alpha ≤ 0, go to LABEL                           | Abort if ill-conditioned
                                                         |

Repeat the above for the next MINC

Exit this program                                        | Program terminates normally if
                                                         | execution proceeds to here
```

LABEL: If program proceeds to here, ill-conditioning had happened, then, skip block 38, do not update the weighting filter coefficients. (That is, use the weighting filter coefficients of the previous adaptation cycle.)

WEIGHTING FILTER COEFFICIENT CALCULATOR (block 38)

Input: AWZTMP

Outputs: AWZ, AWP

Function: Calculate the perceptual weighting filter coefficients from the linear predictor coefficients for input speech.

This block is executed once every adaptation cycle. It is done at ICOUNT = 3 after the processing of block 37 has finished.

```
For   I = 2,3,..,LPCW + 1, do the next line          |
      AWP(I) = WPCFV(I) * AWZTMP(I)                  | Denominator coefficients

For   I = 2,3,..,LPCW + 1, do the next line          |
      AWZ(I) = WZCFV(I) * AWZTMP(I)                  | Numerator coefficients
```

5.6 *Backward synthesis filter adapter (block 23, Figure 5/G.728)*

The three blocks (49, 50 and 51) in Figure 5/G.728 are specified below.

HYBRID WINDOWING MODULE (block 49)

Input: STTMP

Output: RTMP

Function: Apply the hybrid window to quantized speech and compute autocorrelation coefficients.

The operation of this block is essentially the same as in block 36, except for some substitutions of parameters and variables, and for the sampling instant when the autocorrelation coefficients are obtained. As described in § 3, the autocorrelation coefficients are computed based on the quantized speech vectors up to the last vector in the previous 4-vector adaptation cycle. In other words, the autocorrelation coefficients used in the current adaptation cycle are based on the information contained in the quantized speech up to the last (20-th) sample of the previous adaptation cycle. (This is in fact how we define the adaptation cycle.) The STTMP array contains the four quantized speech vectors of the previous adaptation cycle.

```
N1 = LPC + NFRSZ                                  | Compute some constants (can be
N2 = LPC + NONR                                   | precomputed and stored in memory)
N3 = LPC + NFRSZ + NONR

For   N = 1,2,..,N2, do the next line
      SB(N) = SB(N + NFRSZ)                       | Shift the old signal buffer
For   N = 1,2,..,NFRSZ, do the next line
      SB(N2 + N) = STTMP(N)                       | Shift in the new signal
                                                  | SB(N3) is the newest sample

K = 1
For   N = N3,N3 — 1,..,3,2,1, do the next two lines
      WS(N) = SB(N) * WNR(K)                      | Multiply the window function
      K = K + 1

For   I = 1,2,..,LPC + 1, do the next four lines
      TMP = 0
      For  N = LPC + 1,LPC + 2,..,N1, do the next line
           TMP = TMP + WS(N) * WS(N + 1 — I)
      REXP(I) = (3/4) * REXP(I) + TMP             | Update the recursive component

For   I = 1,2,..,LPC + 1, do the next three lines
      RTMP(I) = REXP(I)
      For  N = N1 + 1,N1 + 2,..,N3, do the next line
           RTMP(I) = RTMP(I) + WS(N) * WS(N + 1 — I)
                                                  | Add the non-recursive component

RTMP(1) = RTMP(1) * WNCF                          | White noise correction
```

LEVINSON-DURBIN RECURSION MODULE (block 50)

Input: RTMP

Output: ATMP

Function: Convert autocorrelation coefficients to synthesis filter coefficients.

The operation of this block is exactly the same as in block 37, except for some substitutions of parameters and variables. However, special care should be taken when implementing this block. As described in § 3, although the autocorrelation RTMP array is available at the first vector of each adaptation cycle, the actual updates of synthesis filter coefficients will not take place until the third vector. This intentional delay of updates allows the real-time hardware to spread the computation of this module over the first three vectors of each adaptation cycle. While this module is being executed during the first two vectors of each cycle, the old set of synthesis filter coefficients (the array "A") obtained in the previous cycle is still being used. This is why we need to keep a separate array ATMP to avoid overwriting the old "A" array. Similarly, RTMP, RCTMP, ALPHATMP, etc. are used to avoid interference to other Levinson-Durbin recursion modules (blocks 37 and 44).

```
If RTMP(LPC + 1) = 0, go to LABEL                    | Skip if zero
                                                     |
If RTMP(1) ≤ 0, go to LABEL                          | Skip if zero signal
                                                     |
RCTMP(1) = —RTMP(2)/RTMP(1)
ATMP(1) = 1                                          |
ATMP(2) = RCTMP(1)                                   | First-order predictor
ALPHATMP = RTMP(1) + RTMP(2) * RCTMP(1)              |
If ALPHATMP ≤ 0, go to LABEL                         | Abort if ill-conditioned

For   MINC = 2,3,4,..,LPC, do the following:
      SUM = 0
      For  IP = 1,2,3,..,MINC, do the next two lines
           N1 = MINC — IP + 2
           SUM = SUM + RTMP(N1) * ATMP(IP)

      RCTMP(MINC) = —SUM/ALPHATMP                    |
      MH = MINC/2 + 1                                | Reflection coefficients
      For  IP = 2,3,4,..,MH, do the next four lines  |
           IB = MINC — IP + 2
           AT = ATMP(IP) + RCTMP(MINC) * ATMP(IB)
           ATMP(IB) = ATMP(IB) + RCTMP(MINC) * ATMP(IP)
           ATMP(IP) = AT                             | Update predictor coefficients

      ATMP(MINC + 1) = RCTMP(MINC)                   |
      ALPHATMP = ALPHATMP + RCTMP(MINC) * SUM        | Predictor residual energy
      If ALPHATMP ≤ 0, go to LABEL                   | Abort if ill-conditioned
                                                     |

Repeat the above for the next MINC

Exit this program                                    | Recursion completed normally if
                                                     | execution proceeds to here
```

LABEL: If program proceeds to here, ill-conditioning had happened, then, skip block 51, do not update the synthesis filter coefficients. (That is, use the synthesis filter coefficients of the previous adaptation cycle.)

BANDWIDTH EXPANSION MODULE (block 51)

Input: ATMP

Output: A

Function: Scale synthesis filter coefficients to expand the bandwidths of spectral peaks.

This block is executed only once every adaptation cycle. It is done after the processing of block 50 has finished and before the execution of blocks 9 and 10 at ICOUNT = 3 take place. When the execution of this module is finished and ICOUNT = 3, then we copy the ATMP array to the "A" array to update the filter coefficients.

```
For   I = 2,3,..,LPC + 1, do the next line                |
      ATMP(I) = FACV(I) * ATMP(I)                         | Scale coefficients

Wait until ICOUNT = 3, then                               |
For   I = 2,3,..,LPC + 1, do the next line                | Update coefficients at the third
      A(I) = ATMP(I)                                      | vector of each cycle
```

5.7 *Backward vector gain adapter (block 20, Figure 6/G.728)*

The blocks in Figure 6/G.728 are specified below. For implementation efficiency, some blocks are described together as a single block (they are shown separately in Figure 6/G.728 just to explain the concept). All blocks in Figure 6/G.728 are executed once every speech vector, except for blocks 43, 44 and 45, which are executed only when ICOUNT = 2.

1-VECTOR DELAY, RMS CALCULATOR, AND LOGARITHM CALCULATOR
(blocks 67, 39 and 40)

Input: ET

Output: ETRMS

Function: Calculate the dB level of the root-mean square (RMS) value of the previous gain-scaled excitation vector.

When these three blocks are executed (which is before the VQ codebook search), the ET array contains the gain-scaled excitation vector determined for the previous speech vector. Therefore, the 1-vector delay unit (block 67) is automatically executed. (It appears in Figure 6/G.728 just to enhance clarity.) Since the logarithm calculator immediately follows the RMS calculator, the square root operation in the RMS calculator can be implemented as a "divide-by-two" operation to the output of the logarithm calculator. Hence, the output of the logarithm calculator (the dB value) is $10 * \log_{10}$ (energy of ET/IDIM). To avoid overflow of logarithm value when ET = 0 (after system initialization or reset), the argument of the logarithm operation is clipped to 1 if it is too small. Also, we note that ETRMS is usually kept in an accumulator, as it is a temporary value which is immediately processed in block 42.

```
ETRMS = ET(1) * ET(1)                                     |
For   K = 2,3,..,IDIM, do the next line                   | Compute energy of ET
      ETRMS = ETRMS + ET(K) * ET(K)                       |

ETRMS = ETRMS * DIMINV                                    | Divide by IDIM
If ETRMS < 1, set ETRMS = 1                               | Clip to avoid log overflow
ETRMS = 10 * log10 (ETRMS)                                | Compute dB value
```

LOG-GAIN OFFSET SUBTRACTOR (block 42)

Inputs: ETRMS, GOFF

Output: GSTATE(1)

Function: Subtract the log-gain offset value held in block 41 from the output of block 40 (dB gain level).

```
GSTATE(1) = ETRMS — GOFF
```

HYBRID WINDOWING MODULE (block 43)

Input: GTMP

Output: R

Function: Apply the hybrid window to offset-subtracted log-gain sequence and compute autocorrelation coefficients.

The operation of this block is very similar to block 36, except for some substitutions of parameters and variables, and for the sampling instant when the autocorrelation coefficients are obtained.

An important difference between block 36 and this block is that only four (rather than 20) gain samples are fed to this block each time the block is executed.

The log-gain predictor coefficients are updated at the second vector of each adaptation cycle. The GTMP array below contains four-offset-removed log-gain values, starting from the log-gain of the second vector of the previous adaptation cycle to the log-gain of the first vector of the current adaptation cycle, which is GTMP(1). GTMP(4) is the offset-removed log-gain value from the first vector of the current adaptation cycle, the newest value.

```
N1 = LPCLG + NUPDATE                                  | Compute some constants (can be
N2 = LPCLG + NONRLG                                   | precomputed and stored in memory)
N3 = LPCLG + NUPDATE + NONRLG

For   N = 1,2,..,N2, do the next line                 | Shift the old signal buffer
      SBLG(N) = SBLG(N + NUPDATE)
For   N = 1,2,..,NUPDATE, do the next line            | Shift in the new signal;
      SBLG(N2 + N) = GTMP(N)                          | SBLG(N3) is the newest sample

K = 1
For   N = N3,N3 — 1,..,3,2,1, do the next two lines
      WS(N) = SBLG(N) * WNRLG(K)                      | Multiply the window function
      K = K + 1

For   I = 1,2,..,LPCLG + 1, do the next four lines
      TMP = 0
      For  N = LPCLG + 1,LPCLG + 2,..,N1, do the next line
           TMP = TMP + WS(N) * WS(N + 1 — I)
      REXPLG(I) = (3/4) * REXPLG(I) + TMP             | Update the recursive component

For   I = 1,2,..,LPCLG + 1, do the next three lines
      R(I) = REXPLG(I)
      For  N = N1 + 1,N1 + 2,..,N3, do the next line
           R(I) = R(I) + WS(N) * WS(N + 1 — I)        | Add the non-recursive component

R(1) = R(1) * WNCF                                    | White noise correction
```

LEVINSON-DURBIN RECURSION MODULE (block 44)

Input: R (output of block 43)

Output: GPTMP

Function: Convert autocorrelation coefficients to log-gain predictor coefficients.

The operation of this block is exactly the same as in block 37, except for the substitutions of parameters and variables indicated below: replace LPCW by LPCLG and AWZ by GP. This block is executed only when ICOUNT = 2, after block 43 is executed. Note that as the first step, the value of R(LPCLG + 1) will be checked. If it is zero, we skip blocks 44 and 45 without updating the log-gain predictor coefficients. (That is, we keep using the old log-gain predictor coefficients determined in the previous adaptation cycle.) This special procedure is designed to avoid a very small glitch that would have otherwise happened right after system initialization or reset. In case the matrix is ill-conditioned, we also skip block 45 and use the old values.

BANDWIDTH EXPANSION MODULE (block 45)

Input: GPTMP

Output: GP

Function: Scale log-gain predictor coefficients to expand the bandwidths of spectral peaks.

This block is executed only when ICOUNT = 2, after block 44 is executed.

```
For   I = 2,3,..,LPCLG + 1, do the next line                    |
      GP(I) = FACGPV(I) * GPTMP(I)                              | Scale coefficients
```

LOG-GAIN LINEAR PREDICTOR (block 46)

Inputs: GP, GSTATE

Output: GAIN

Function: Predict the current value of the offset-subtracted log-gain.

```
GAIN = 0
For   I = LGLPC,LPCLG — 1,..,3,2, do the next two lines
      GAIN = GAIN — GP(I + 1) * GSTATE(I)
      GSTATE(I) = GSTATE(I — 1)
GAIN = GAIN — GP(2) * GSTATE(1)
```

LOG-GAIN OFFSET ADDER (between blocks 46 and 47)

Inputs: GAIN, GOFF

Output: GAIN

Function: Add the log-gain offset value back to the log-gain predictor output.

```
GAIN = GAIN + GOFF
```

LOG-GAIN LIMITER (block 47)

Input: GAIN

Output: GAIN

Function: Limit the range of the predicted logarithmic gain.

```
If GAIN < 0, set GAIN = 0                    | Correspond to linear gain 1
If GAIN > 60, set GAIN = 60                  | Correspond to linear gain 1000
```

INVERSE LOGARITHM CALCULATOR (block 48)

Input: GAIN

Output: GAIN

Function: Convert the predicted logarithmic gain (in dB) back to linear domain.

$GAIN = 10^{(GAIN/20)}$

5.8 *Perceptual weighting filter*

PERCEPTUAL WEIGHTING FILTER (block 4)

Inputs: S, AWZ, AWP

Output: SW

Function: Filter the input speech vector to achieve perceptual weighting.

```
For   K = 1,2,..,IDIM, do the following:
      SW(K) = S(K)
      For  J = LPCW,LPCW — 1,..,3,2, do the next two lines
           SW(K) = SW(K) + WFIR(J) * AWZ(J + 1)          |
           WFIR(J) = WFIR(J — 1)                         | All-zero part of the filter

      SW(K) = SW(K) + WFIR(1) * AWZ(2)                   |
      WFIR(1) = S(K)                                     | Handle last one differently

      For  J = LPCW,LPCW — 1,..,3,2, do the next two lines |
           SW(K) = SW(K) — WIIR(J) * AWP(J + 1)          | All-pole part of the filter
           WIIR(J) = WIIR(J — 1)

      SW(K) = SW(K) — WIIR(1) * AWP(2)                   |
      WIIR(1) = SW(K)                                    | Handle last one differently

Repeat the above for the next K
```

5.9 *Computation of zero-input response vector*

Subsection 3.5 explains how a "zero-input response vector" $r(n)$ is computed by blocks 9 and 10. Now the operation of these two blocks during this phase is specified below. Their operation during the "memory update phase" will be described later.

SYNTHESIS FILTER (block 9) DURING ZERO-INPUT RESPONSE COMPUTATION

Inputs: A, STATELPC

Output: TEMP

Function: Compute the zero-input response vector of the synthesis filter.

```
For  K = 1,2,..,IDIM, do the following:
     TEMP(K) = 0
     For  J = LPC,LPC — 1,..,3,2, do the next two lines
          TEMP(K) = TEMP(K) — STATELPC(J) * A(J + 1)      | Multiply-add
          STATELPC(J) = STATELPC(J — 1)                   | Memory shift

     TEMP(K) = TEMP(K) — STATELPC(1) * A(2)               |
     STATELPC(1) = TEMP(K)                                | Handle last one differently

Repeat the above for the next K
```

PERCEPTUAL WEIGHTING FILTER DU RING ZERO-INPUT RESPONSE COMPUTATION (block 10)

Inputs: AWZ, AWP, ZIRWFIR, ZIRWIIR, TEMP computed above

Output: ZIR

Function: Compute the zero-input response vector of the perceptual weighting filter.

```
For  K = 1,2,..,IDIM, do the following:
     TMP = TEMP(K)
          For  J = LPCW,LPCW — 1,..,3,2, do the next two lines
               TEMP(K) = TEMP(K) + ZIRWFIR(J) * AWZ(J + 1)   |
               ZIRWFIR(J) = ZIRWFIR(J — 1)                   | All-zero part of the filter

     TEMP(K) = TEMP(K) + ZIRWFIR(1) * AWZ(2)               |
     ZIRWFIR(1) = TMP                                      | Handle last one differently

     For  J = LPCW,LPCW — 1,..,3,2, do the next two lines  |
          TEMP(K) = TEMP(K) — ZIRWIIR(J) * AWP(J + 1)      | All-pole part of the filter
          ZIRWIIR(J) = ZIRWIIR(J — 1)

     ZIR(K) = TEMP(K) — ZIRWIIR(1) * AWP(2)                |
     ZIRWIIR(1) = ZIR(K)                                   | Handle last one differently

Repeat the above for the next K
```

5.10 *VQ target vector computation*

VQ TARGET VECTOR COMPUTATION (block 11)

Inputs: SW, ZIR

Output: TARGET

Function: Subtract the zero-input response vector from the weighted speech vector.

Note — *ZIR*(*K*) = *ZIRWIIR*(*IDIM* + 1 — *K*) from block 10 above. It does not require a separate storage location.

```
For   K = 1,2,..,IDIM, do the next line
      TARGET(K) = SW(K) — ZIR(K)
```

5.11 *Codebook search module (block 24)*

The seven blocks contained within the codebook search module (block 24) are specified below. Again, some blocks are described as a single block for convenience and implementation efficiency. Blocks 12, 14 and 15 are executed once every adaptation cycle when ICOUNT = 3, while the other blocks are executed once every speech vector.

IMPULSE RESPONSE VECTOR CALCULATOR (block 12)

Inputs: A, AWZ, AWP

Output: H

Function: Compute the impulse response vector of the cascaded synthesis filter and perceptual weighting filter.

This block is executed when ICOUNT = 3 and after the execution of block 23 and 3 is completed (i.e. when the new sets of A, AWZ, AWP coefficients are ready).

```
TEMP(1) = 1                                  | TEMP = synthesis filter memory
RC(1) = 1                                    | RC = W(z) all-pole part memory
For   K = 2,3,..,IDIM, do the following:
      A0 = 0
      A1 = 0
      A2 = 0
      For  I = K,K — 1,..,3,2, do the next five lines
           TEMP(I) = TEMP(I — 1)
           RC(I) = RC(I — 1)                 |
           A0 = A0 — A(I) * TEMP(I)          | Filtering
           A1 = A1 + AWZ(I) * TEMP(I)        |
           A2 = A2 — AWP(I) * RC(I)

      TEMP(1) = A0
      RC(1) = A0 + A1 + A2
Repeat the above indented section  for the next K

ITMP = IDIM + 1                              | Obtain h(n) by reversing the
For   K = 1,2,..,IDIM, do the next line      | order of the memory of all-pole
      H(K) = RC(ITMP — K)                    | section of W(z)
```

SHAPE CODEVECTOR CONVOLUTION MODULE AND ENERGY TABLE CALCULATOR (blocks 14 and 15)

Inputs: H, Y

Output: Y2

Function: Convolve each shape codevector with the impulse response obtained in block 12, then compute and store the energy of the resulting vector.

This block is also executed when ICOUNT = 3 after the execution of block 12 is completed.

```
For   J = 1,2,..,NCWD, do the following:                  | One codevector per loop
      J1 = (J — 1) * IDIM
      For  K = 1,2,..,IDIM, do the next four lines
           K1 = J1 + K + 1
           TEMP(K) = 0
           For   I = 1,2,..,K, do the next line          |
                 TEMP(K) = TEMP(K) + H(I) * Y(K1 — I)    | Convolution

Repeat the above 4 lines for the next K

Y2(J) = 0
For   K = 1,2,..,IDIM, do the next line                   |
      Y2(J) = Y2(J) + TEMP(K) * TEMP(K)                   | Compute energy

Repeat the above for the next J
```

VQ TARGET VECTOR NORMALIZATION (block 16)

Inputs: TARGET, GAIN

Output: TARGET

Function: Normalize the VQ target vector using the predicted excitation gain.

```
TMP = 1 / GAIN
For   K = 1,2,..,IDIM, do the next line
      TARGET(K) = TARGET(K) * TMP
```

TIME-REVERSED CONVOLUTION MODULE (block 13)

Inputs: H, TARGET (output from block 16)

Output: PN

Function: Perform time-reversed convolution of the impulse response vector and the normalized VQ target vector (to obtain the vector $p(n)$).

Note — The vector PN can be kept in temporary storage.

```
For   K = 1,2,..,IDIM, do the following:
      K1 = K — 1
      PN(K) = 0
      For  J = K,K + 1,..,IDIM, do the next line
           PN(K) = PN(K) + TARGET(J) * H(J — K1)

Repeat the above for the next K
```

ERROR CALCULATOR AND BEST CODEBOOK INDEX SELECTOR
(blocks 17 and 18)

Inputs: PN, Y, Y2, GB, G2, GSQ

Outputs: IG, IS, ICHAN

Function: Search through the gain codebook and the shape codebook to identify the best combination of gain codebook index and shape codebook index, and combine the two to obtain the 10-bit best codebook index.

Note — The variable COR used below is usually kept in an accumulator, rather than storing it in memory. The variables IDXG and J can be kept in temporary registers, while IG and IS can be kept in memory.

```
Initialize DISTM tothe largest number representable in the hardware
N1 = NG/2
For   J = 1,2,..,NCWD, do the following:
      J1 = (J — 1) * IDIM
      COR = 0
      For  K = 1,2,..,IDIM, do the next line                    |
           COR = COR + PN(K) * Y(J1 + K)                        | Compute inner product Pj

      If COR > 0, then do the next five lines
           IDXG = N1
           For   K = 1,2,..,N1 — 1, do the next "if" statement
                 If COR < GB(K) * Y2(J), do the next two lines
                      IDXG = K                                  | Best positive gain found
                      GO TO LABEL

      If COR ≤ 0, then do the next five lines
           IDXG = NG
           For   K = N1 + 1,N1 + 2,..,NG — 1, do the next "if" statement
                 If COR > GB(K) * Y2(J), do the next two lines
                      IDXG = K                                  | Best negative gain found
                      GO TO LABEL

LABEL: D = —G2(IDXG) * COR + GSQ(IDXG) * Y2(J)                  | Compute distortion D^

      If D < DISTM, do the next three lines
           DISTM = D                                            | Save the lowest distortion and the
           IG = IDXG                                            | best codebook indices so far
           IS = J                                               |

Repeat the above indented section for the next K

ICHAN = (IS — 1) * NG + (IG — 1)                                | Concatenate shape and gain
                                                                | codebook indices
```

Transmit ICHAN through communication channel

For serial bit stream transmission, the most significant bit of ICHAN should be transmitted first.

If ICHAN is represented by the 10 bit word $b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$, then the order of the transmitted bits should be b_9, and then b_8, and then b_7, ..., and finally b_0. (b_9 is the most significant bit.)

5.12 *Simulated decoder (block 8)*

Blocks 20 and 23 have been described earlier. Blocks 19, 21 and 22 are specified below.

EXCITATION VQ CODEBOOK (block 19)

Inputs: IG, IS

Output: YN

Function: Perform table look-up to extract the best shape codevector and the best gain, then multiply them to get the quantized excitation vector.

```
NN = (IS —1) * IDIM
For   K = 1,2,..,IDIM, do the next line
      YN(K) = GQ(IG) * Y(NN + K)
```

GAIN SCALING UNIT (block 21)

Inputs: GAIN, YN

Output: ET

Function: Multiply the quantized excitation vector by the excitation gain.

```
For   K = 1,2,..,IDIM, do the next line
      ET(K) = GAIN * YN(K)
```

SYNTHESIS FILTER (block 22)

Inputs: ET, A

Output: ST

Function: Filter the gain-scaled excitation vector to obtain the quantized speech vector.

As explained in § 3, this block can be omitted and the quantized speech vector can be obtained as a by-product of the memory update procedure to be described below. If, however, one wishes to implement this block anyway, a separate set of filter memory (rather than STATELPC) should be used for this all-pole synthesis filter.

5.13 *Filter memory update for blocks 9 and 10*

The following description of the filter memory update procedures for blocks 9 and 10 assumes that the quantized speech vector ST is obtained as a by-product of the memory updates. To safeguard possible overloading of signal levels, a magnitude limiter is built into the procedure so that the filter memory clips at MAX and MIN, where MAX and MIN are respectively the positive and negative saturation levels of A-law or μ-law PCM, depending on which law is used.

FILTER MEMORY UPDATE (blocks 9 and 10)

Inputs: ET, A, AWZ, AWP, STATELPC, ZIRWFIR, ZIRWIIR

Outputs: ST, STATELPC, ZIRWFIR, ZIRWIIR

Function: Update the filter memory of blocks 9 and 10 and also obtain the quantized speech vector.

```
ZIRWFIR(1) = ET(1)                                  | ZIRWFIR now a scratch array
TEMP(1) = ET(1)
For   K = 2,3,..,IDIM, do the following:
      A0 = ET(K)
      A1 = 0
      A2 = 0
      For  I = K,K — 1,..,2, do the next five lines
           ZIRWFIR(I) = ZIRWFIR(I — 1)
           TEMP(I) = TEMP(I — 1)
      A0 = A0 — A(I) * ZIRWFIR(I)                   |
      A1 = A1 + AWZ(I) * ZIRWFIR(I)                 | Compute zero-state responses at
      A2 = A2 — AWP(I) * TEMP(I)                    | various stages of the cascaded
                                                    | filter
      ZIRWFIR(1) = A0                               |
      TEMP(1) = A0 + A1 + A2

Repeat the above indented section for the next K

                                                    | Now update filter memory by adding
                                                    | zero-state responses to zero-input
                                                    | responses
For   K = 1,2,..,IDIM, do the next four lines
      STATELPC(K) = STATELPC(K) + ZIRWFIR(K)
      If STATELPC(K) > MAX, set STATELPC(K) = MAX   | Limit the range
      If STATELPC(K) < MIN, set STATELPC(K) = MIN   |
      ZIRWIIR(K) = ZIRWIIR(K) + TEMP(K)

For   I = 1,2,..,LPCW, do the next line             | Now set ZIRWFIR to the right
      ZIRWFIR(I) = STATELPC(I)                      | value

I = IDIM + 1
For   K = 1,2,..,IDIM, do the next line             | Obtain quantized speech by reversing
      ST(K) = STATELPC(I — K)                       | order of synthesis filter memory
```

5.14 *Decoder (Figure 3/G.728)*

The blocks in the decoder (Figure 3/G.728) are described below. Except for the output PCM format conversion block, all other blocks are exactly the same as the blocks in the simulated decoder (block 8) in Figure 2/G.728.

The decoder only uses a subset of the variables in Table 2/G.728. If a decoder and an encoder are to be implemented in a single DSP chip, then the decoder variables should be given different names to avoid overwriting the variables used in the simulated decoder block of the encoder. For example, to name the decoder variables, we can add a prefix "d" to the corresponding variable names in Table 2/G.728. If a decoder is to be implemented as a stand-alone unit independent of an encoder, then there is no need to change the variable names.

The following description assumes a stand-alone decoder. Again, the blocks are executed in the same order as they are described below.

DECODER BACKWARD SYNTHESIS FILTER ADAPTER (block 33)

Input: ST

Output: A

Function: Generate synthesis filter coefficients periodically from previously decoded speech.

The operation of this block is exactly the same as block 23 of the encoder.

DECODER BACKWARD VECTOR GAIN ADAPTER (block 30)

Input: ET

Output: GAIN

Function: Generate the excitation gain from previous gain-scaled excitation vectors.

The operation of this block is exactly the same as block 20 of the encoder.

DECODER EXCITATION VQ CODEBOOK (block 29)

Input: ICHAN

Output: YN

Function: Decode the received best codebook index (channel index) to obtain the excitation vector.

This block first extracts the 3-bit gain codebook index IG and the 7-bit shape codebook index IS from the received 10-bit channel index. Then, the rest of the operation is exactly the same as block 19 of the encoder.

```
ITMP = integer part of (ICHAN / NG)          | Decode (IS — 1)
IG = ICHAN — ITMP * NG + 1                   | Decode IG

NN = ITMP * IDIM
For   K = 1,2,..,IDIM, do the next line
      YN(K) = GQ(IG) * Y(NN + K)
```

DECODER GAIN SCALING UNIT (block 31)

Inputs: GAIN, YN

Output: ET

Function: Multiply the excitation vector by the excitation gain.

The operation of this block is exactly the same as block 21 of the encoder.

DECODER SYNTHESIS FILTER (block 32)

Inputs: ET, A, STATELPC

Output: ST

Function: Filter the gain-scaled excitation vector to obtain the decoded speech vector.

This block can be implemented as a straightforward all-pole filter. However, as mentioned in § 4.3, if the encoder obtains the quantized speech as a by-product of filter memory update (to save computation), and if potential accumulation of round-off error is a concern, then this block should compute the decoded speech in exactly the same way as in the simulated decoder block of the encoder. That is, the decoded speech vector should be computed as the sum of the zero-input response vector and the zero-state response vector of the synthesis filter. This can be done by the following procedure.

```
For     K = 1,2,..,IDIM, do the next seven lines
        TEMP(K) = 0
        For  J = LPC,LPC — 1,..,3,2, do the next two lines
             TEMP(K) = TEMP(K) — STATELPC(J) * A(J + 1)        | Zero-input response
             STATELPC(J) = STATELPC(J — 1)

        TEMP(K) = TEMP(K) — STATELPC(1) * A(2)                 |
        STATELPC(1) = TEMP(K)                                  | Handle last one differently

Repeat the above for the next K

TEMP(1) = ET(1)
For     K = 2,3,..,IDIM, do the next five lines
        A0 = ET(K)
        For  I = K,K — 1,..,2, do the next two lines
             TEMP(I) = TEMP(I — 1)
             A0 = A0 — A(I) * TEMP(I)                          | Compute zero-state response

        TEMP(1) = A0

Repeat the above for the next K

                                                               | Now update filter memory by adding
                                                               | zero-state responses to zero-input
                                                               | responses

For     K = 1,2,..,IDIM, do the next three lines
        STATELPC(K) = STATELPC(K) + TEMP(K)                    | ZIR + ZSR
        If STATELPC(K) > MAX, set STATELPC(K) = MAX            | Limit the range
        If STATELPC(K) < MIN, set STATELPC(K) = MIN            |

I = IDM + 1
For     K = 1,2,..,IDIM, do the next line                      | Obtain quantized speech by reversing
        ST(K) = STATELPC(I — K)                                | order of synthesis filter memory
```

10th-ORDER LPC INVERSE FILTER (block 81)

This block is executed once a vector, and the output vector is written sequentially into the last 20 samples of the LPC prediction residual buffer [i.e. D(81) through D(100)]. We use a pointer IP to point to the address of D(K) array samples to be written to. This pointer IP is initialized to NPWSZ — NFRSZ + IDIM before this block starts to process the first decoded speech vector of the first adaptation cycle (frame), and from there on IP is updated in the way described below. The 10th-order LPC predictor coefficients APF(I)s are obtained in the middle of Levinson-Durbin recursion by block 50, as described in § 4.6. It is assumed that before this block starts execution, the decoder synthesis filter (block 32 of Figure 3/G.728) has already written the current decoded speech vector into ST(1) through ST(IDIM).

Inputs: ST, APF

Output: D

Function: Compute the LPC prediction residual for the current decoded speech vector.

```
If IP = NPWSZ, then set IP = NPWSZ — NFRSZ                  | Check and update IP

    For K = 1,2,..,IDIM, do the next seven lines
        ITMP = IP + K
            D(ITMP) = ST(K)
            For   J = 10,9,..,3,2, do the next two lines
                D(ITMP) = D(ITMP) + STLPCI(J) * APF(J + 1)  | FIR filtering
                STLPCI(J) = STLPCI(J — 1)                   | Memory shift
            D(ITMP) = D(ITMP) + STLPCI(1) * APF(2)          | Handle last one
            STLPCI(1) = ST(K)                               | Shift in input

IP = IP + IDIM                                              | Update IP
```

PITCH PERIOD EXTRACTION MODULE (block 82)

This block is executed once a frame at the third vector of each frame, after the third decoded speech vector is generated.

Input: D

Output: KP

Function: Extract the pitch period from the LPC prediction residual.

```
If ICOUNT ≠ 3, skip the execution of this block,
otherwise, do the following:                                | Lowpass filtering & 4:1 downsampling

    For  K = NPWSZ — NFRSZ + 1,..,NPWSZ, do the next seven lines  | IIR filter
    TMP = D(K) — STLPF(1) * AL(1) — STLPF(2) *
    AL(2) — STLPF(3) * AL(3)
    If   K is divisible by 4, do the next two lines         | Do FIR filtering only if needed
        N = K/4

        DEC(N) = TMP * BL(1) + STLPF(1) * BL(2) + STLPF(2) * BL(3) + STLPF(3) * BL(4)

        STLPF(3) = STLPF(2)
        STLPF(2) = STLPF(1)                                 | Shift lowpass filter memory
        STLPF(1) = TMP

M1 = KPMIN/4                                                | Start correlation peak-picking in the
M2 = KPMAX/4                                                | decimated LPC residual domain
CORMAX = most negative number of the machine
```

```
For   J = M1,M1 + 1,..,M2, do the next six lines
      TMP = 0
      For  N = 1,2,..,NPWSZ/4, do the next line
           TMP = TMP + DEC(N) * DEC(N — J)          | TMP = correlation in decimated domain

      If   TMP > CORMAX, do the next two lines      | Find maximum correlation and the
           CORMAX = TMP                             | corresponding lag
           KMAX = J
For   N = —M2 + 1,—M2 + 2,..,(NPWSZ — NFRSZ)/4, do the next line   | Shift decimated LPC residual buffer
      DEC(N) = DEC(N + IDIM)

M1 = 4 * KMAX — 3                                   | Start correlation peak-picking in undecimated domain
M2 = 4 * KMAX + 3

If    M1 < KPMIN, set M1 = KPMIN                    | Check whether M1 out of range
If    M2 > KPMAX, set M2 = KPMAX                    | Check whether M2 out of range
CORMAX = most negative number of the machine
For   J = M1,M1 + 1,..,M2, do the next six lines
      TMP = 0
      For  K = 1,2,..,NPWSZ, do the next line
           TMP = TMP + D(K) * D(K — J)              | Correlation in undecimated domain
      If   TMP > CORMAX, do the next two lines
           CORMAX = TMP                             | Find maximum correlation and the
           KP = J                                   | corresponding lag

M1 = KP1 — KPDELTA                                  | Determine the range of search around
M2 = KP1 + KPDELTA                                  | the pitch period of previous frame

If    KP < M2 + 1, go to LABEL                      | KP can't be a multiple pitch if true

If    M1 < KPMIN, set M1 =  KPMIN                   | Check whether M1 out of range
CMAX = most negative number of the machine
      For  J = M1,M1 + 1,..,M2, do the next six lines   | Correlation in undecimated domain
           TMP = 0
           For   K = 1,2,..,NPWSZ, do the next line     | Find maximum correlation and the
                 TMP = TMP + D(K) * D(K — J)            | corresponding lag
           If    TMP > CMAX, do the next two lines
                 CMAX = TMP
                 KPTMP = J

SUM = 0
TMP = 0                                             | Start computing the tap weights

For   K = 1,2,..,NPWSZ, do the next two lines
      SUM = SUM + D(K — KP) * D(K — KP)
      TMP = TMP + D(K — KPTMP) * D(K — KPTMP)
If    SUM = 0, set TAP = 0, otherwise, set TAP = CORMAX/SUM
If    TMP = 0, set TAP1 = 0, otherwise, set TAP1 = CMAX/TMP   | Clamp TAP between 0 and 1
If    TAP > 1, set TAP = 1
If    TAP < 0, set TAP = 0                          | Clamp TAP1 between 0 and 1
If    TAP1 > 1, set TAP1 = 1
If    TAP1 < 0, set TAP1 = 0                        | Replace KP with fundamental pitch
                                                    | if TAP1 is large enough

If    TAP1 > TAPTH * TAP, then set KP = KPTMP

LABEL:  KP1 = KP                                    | Update pitch period of previous frame

      For  K = KPMAX + 1,—KPMAX + 2,..,NPWSZ — NFRSZ, do the next line

           D(K) = D(K + NFRSZ)                      | Shift the LPC residual buffer
```

PITCH PREDICTOR TAP CALCULATOR (block 83)

This block is also executed once a frame at the third vector of each frame, right after the execution of block 82. This block shares the decoded speech buffer (ST(K) array) with the long-term postfilter 71, which takes care of the shifting of the array such that ST(1) through ST(IDIM) constitute the current vector of decoded speech, and ST(—KPMAX — NPWSZ + 1) through ST(0) are previous vectors of decoded speech.

Inputs: ST, KP

Output: PTAP

Function: Calculate the optimal tap weight of the single-tap pitch predictor of the decoded speech.

```
If ICOUNT ≠ 3, skip the execution of this block,
otherwise, do the following:
      SUM = 0
      TMP = 0
      For  K = —NPWSZ + 1,— NPWSZ + 2,..,0, do the next two
lines
            SUM = SUM + ST(K — KP) * ST(K — KP)
            TMP = TMP + ST(K) * ST(K — KP)
      If    SUM = 0, set PTAP = 0, otherwise, set PTAP = TMP/SUM
```

LONG-TERM POSTFILTER COEFFICIENT CALCULATOR (block 84)

This block is also executed once a frame at the third vector of each frame, right after the execution of block 83.

Input: PTAP

Outputs: B, GL

Function: Calculate the coefficient b and the scaling factor g_l of the long-term postfilter.

```
If ICOUNT ≠ 3, skip the execution of this block,
otherwise, do the following:
      If    PTAP > 1, set PTAP = 1                  | Clamp PTAP at 1
      If    PTAP < PPFTH, set PTAP = 0              | Turn off pitch postfilter if
                                                    | PTAP smaller than threshold

B = PPFZCF * PTAP
GL = 1 / (1 + B)
```

SHORT-TERM POSTFILTER COEFFICIENT CALCULATOR (block 85)

This block is also executed once a frame, but it is executed at the first vector of each frame.

Inputs: APF, RCTMP(1)

Outputs: AP, AZ, TILTZ

Function: Calculate the coefficients of the short-term postfilter.

```
If ICOUNT ≠ 1, skip the execution of this block,
otherwise, do the following:
            For   I = 2,3,..,11, do the next two lines  |
                  AP(I) = SPFPCFV(I) * APF(I)           | Scale denominator coefficients
                  AZ(I) = SPFZCFV(I) * APF(I)           | Scale numerator coefficients
            TILTZ = TILTF * RCTMP(1)                    | Tilt compensation filter coefficients
```

LONG-TERM POSTFILTER (block 71)

This block is executed once a vector.

Inputs: ST, B, GL, KP

Output: TEMP

Function: Perform filtering operation of the long-term postfilter.

```
For   K = 1,2,..,IDIM, do the next line
      TEMP(K) = GL * (ST(K) + B * ST(K — KP))                  | Long-term postfiltering

For   K = —NPWSZ — KPMAX + 1,..,—2,—1,0, do the next line
      ST(K) = ST(K + IDIM)                                     | Shift decoded speech buffer
```

SHORT-TERM POSTFILTER (block 72)

This block is executed once a vector right after the execution of block 71.

Inputs: AP, AZ, TILTZ, STPFFIR, STPFIIR, TEMP (output of block 71)

Output: TEMP

Function: Perform filtering operation for the short-term postfilter.

```
For   K = 1,2,..,IDIM, do the following:
      TMP = TEMP(K)
          For   J = 10,9,..,3,2, do the next two lines
                TEMP(K) = TEMP(K) + STPFFIR (J) * AZ(J + 1)    |
                STPFFIR(J) = STPFFIR(J — 1)                    | All-zero part of the filter
      TEMP(K) = TEMP(K) + STPFFIR(1) * AZ(2)                   | Last multiplier
      STPFFIR(1) = TMP

      For  J = 10,9,..,3,2, do the next two lines
           TEMP(K) = TEMP(K) — STPFIIR(J) * AP(J + 1)          |
           STPFIIR(J) = STPFIIR(J — 1)                         | All-pole part of the filter
      TEMP(K) = TEMP(K) — STPFIIR(1) * AP(2)                   | Last multiplier
      STPFIIR(1) = TEMP(K)
TEMP(K) = TEMP(K) + STPFIIR(2) * TILTZ                         | Spectral tilt compensation filter
```

SUM OF ABSOLUTE VALUE CALCULATOR (block 73)

This block is executed once a vector after execution of block 32.

Input: ST

Output: SUMUNFIL

Function: Calculate the sum of absolute values of the components of the decoded speech vector.

```
SUMUNFIL = 0
For   K = 1,2,..,IDIM, do the next line
      SUMUNFIL = SUMUNFIL + absolute value of ST(K)
```

SUM OF ABSOLUTE VALUE CALCULATOR (block 74)

This block is executed once a vector after execution of block 72.

Input: TEMP (output of block 72)

Output: SUMFIL

Function: Calculate the sum of absolute values of the components of the short-term postfilter output vector.

```
SUMFIL = 0
For   K = 1,2,..,IDIM, do the next line
      SUMFIL = SUMFIL + absolute value of TEMP(K)
```

SCALING FACTOR CALCULATOR (block 75)

This block is executed once a vector after execution of blocks 73 and 74.

Inputs: SUMUNFIL, SUMFIL

Output: SCALE

Function: Calculate the overall scaling factor of the postfilter.

```
If SUMFIL > 1, set SCALE = SUMUNFIL / SUMFIL,
otherwise, set SCALE = 1
```

FIRST-ORDER LOWPASS FILTER (block 76) and
OUTPUT GAIN SCALING UNIT (block 77)

These two blocks are executed once a vector after execution of blocks 72 and 75. It is more convenient to describe the two blocks together.

Inputs: SCALE, TEMP (output of block 72)

Output: SPF

Function: Lowpass filter the once-a-vector scaling factor and use the filtered scaling factor to scale the short-term postfilter output vector.

```
For   K = 1,2,..,IDIM, do the following:
      SCALEFIL = AGCFAC * SCALEFIL +          | Lowpass filtering
      (1 — AGCFAC) * SCALE                    | Scale output
      SPF(K) = SCALEFIL * TEMP(K)
```

OUTPUT PCM FORMAT CONVERSION (block 28)

Input: SPF

Output: SD

Function: Convert the five components of the decoded speech vector into 5-corresponding A-law or μ-law PCM samples and put them out sequentially at 125 μs time intervals.

The conversion rules from uniform PCM to A-law or μ-law PCM are specified in Recommendation G.711.

ANNEX A

(to Recommendation G.728)

Hybrid window functions for various LPC analyses in LD-CELP

In the LD-CELP coder, we use three separate LPC analyses to update the coefficients of three filters: the synthesis filter, the log-gain predictor, and the perceptual weighting filter. Each of these three LPC analyses has its own hybrid window. For each hybrid window, we list the values of window function samples that are used in the hybrid windowing calculation procedure. These window functions were first designed using floating-point arithmetic and then quantized to the numbers which can be exactly represented by 16-bit representations with 15 bits of fraction. For each window, we will first give a table containing the floating-point equivalent of the 16-bit numbers and then give a table with corresponding 16-bit integer representations.

A.1 *Hybrid window for the synthesis filter*

The following table contains the first 105 samples of the window function for the synthesis filter. The first 35 samples are the non-recursive portion, and the rest are the recursive portion. The table should be read from left to right from the first row, then left to right for the second row, and so on (just like the raster scan line).

0.047760010	0.095428467	0.142852783	0.189971924	0.236663818
0.282775879	0.328277588	0.373016357	0.416900635	0.459838867
0.501739502	0.542480469	0.582000732	0.620178223	0.656921387
0.692199707	0.725891113	0.757904053	0.788208008	0.816680908
0.843322754	0.868041992	0.890747070	0.911437988	0.930053711
0.946533203	0.960876465	0.973022461	0.982910156	0.990600586
0.996002197	0.999114990	0.999969482	0.998565674	0.994842529
0.988861084	0.981781006	0.974731445	0.967742920	0.960815430
0.953948975	0.947082520	0.940307617	0.933563232	0.926879883
0.920227051	0.913635254	0.907104492	0.900604248	0.894134521
0.887725830	0.881378174	0.875061035	0.868774414	0.862548828
0.856384277	0.850250244	0.844146729	0.838104248	0.832092285
0.826141357	0.820220947	0.814331055	0.808502197	0.802703857
0.796936035	0.791229248	0.785583496	0.779937744	0.774353027
0.768798828	0.763305664	0.757812500	0.752380371	0.747009277
0.741638184	0.736328125	0.731048584	0.725830078	0.720611572
0.715454102	0.710327148	0.705230713	0.700164795	0.695159912
0.690185547	0.685241699	0.680328369	0.675445557	0.670593262
0.665802002	0.661041260	0.656280518	0.651580811	0.646911621
0.642272949	0.637695313	0.633117676	0.628570557	0.624084473
0.619598389	0.615142822	0.610748291	0.606384277	0.602020264

The next table contains the corresponding 16-bit integer representation. Dividing the table entries by $2^{15} = 32\,768$ gives the table above.

1 565	3 127	4 681	6 225	7 755
9 266	10 757	12 223	13 661	15 068
16 441	17 776	19 071	20 322	21 526
22 682	23 786	24 835	25 828	26 761
27 634	28 444	29 188	29 866	30 476
31 016	31 486	31 884	32 208	32 460
32 637	32 739	32 767	32 721	32 599
32 403	32 171	31 940	31 711	31 484
31 259	31 034	30 812	30 591	30 372
30 154	29 938	29 724	29 511	29 299
29 089	28 881	28 674	28 468	28 264
28 062	27 861	27 661	27 463	27 266
27 071	26 877	26 684	26 493	26 303
26 114	25 927	25 742	25 557	25 374
25 192	25 012	24 832	24 654	24 478
24 302	24 128	23 955	23 784	23 613
23 444	23 276	23 109	22 943	22 779
22 616	22 454	22 293	22 133	21 974
21 817	21 661	21 505	21 351	21 198
21 046	20 896	20 746	20 597	20 450
20 303	20 157	20 013	19 870	19 727

A.2 *Hybrid window for the log-gain predictor*

The following table contains the first 34 samples of the window function for the log-gain predictor. The first 20 samples are the non-recursive portion, and the rest are the recursive portion. The table should be read in the same manner as the two tables above.

0.092346191	0.183868408	0.273834229	0.361480713	0.446014404
0.526763916	0.602996826	0.674072266	0.739379883	0.798400879
0.850585938	0.895507813	0.932769775	0.962066650	0.983154297
0.995819092	0.999969482	0.995635986	0.982757568	0.961486816
0.932006836	0.899078369	0.867309570	0.836669922	0.807128906
0.778625488	0.751129150	0.724578857	0.699005127	0.674316406
0.650482178	0.627502441	0.605346680	0.583953857	

The next table contains the corresponding 16-bit integer representation. Dividing the table entries by 2^{15} = 32 768 gives the table above.

3 026	6 025	8 973	11 845	14 615
17 261	19 759	22 088	24 228	26 162
27 872	29 344	30 565	31 525	32 216
32 631	32 767	32 625	32 203	31 506
30 540	29 461	28 420	27 416	26 448
25 514	24 613	23 743	22 905	22 096
21 315	20 562	19 836	19 135	

A.3 *Hybrid window for the perceptual weighting filter*

The following table contains the first 60 samples of the window function for the perceptual weighting filter. The first 30 samples are the non-recursive portion, and the rest are the recursive portion. The table should be read in the same manner as the four tables above.

0.059722900	0.119262695	0.178375244	0.236816406	0.294433594
0.351013184	0.406311035	0.460174561	0.512390137	0.562774658
0.611145020	0.657348633	0.701171875	0.742523193	0.781219482
0.817108154	0.850097656	0.880035400	0.906829834	0.930389404
0.950622559	0.967468262	0.980865479	0.990722656	0.997070313
0.999847412	0.999084473	0.994720459	0.986816406	0.975372314
0.960449219	0.943939209	0.927734375	0.911804199	0.896148682
0.880737305	0.865600586	0.850738525	0.836120605	0.821746826
0.807647705	0.793762207	0.780120850	0.766723633	0.753570557
0.740600586	0.727874756	0.715393066	0.703094482	0.691009521
0.679138184	0.667480469	0.656005859	0.644744873	0.633666992
0.622772217	0.612091064	0.601562500	0.591217041	0.581085205

The next table contains the corresponding 16-bit integer representation. Dividing the table entries by 2^{15} = 32 768 gives the table above.

1 957	3 908	5 845	7 760	9 648
11 502	13 314	15 079	16 790	18 441
20 026	21 540	22 976	24 331	25 599
26 775	27 856	28 837	29 715	30 487
31 150	31 702	32 141	32 464	32 672
32 763	32 738	32 595	32 336	31 961
31 472	30 931	30 400	29 878	29 365
28 860	28 364	27 877	27 398	26 927
26 465	26 010	25 563	25 124	24 693
24 268	23 851	23 442	23 039	22 643
22 254	21 872	21 496	21 127	20 764
20 407	20 057	19 712	19 373	19 041

ANNEX B

(to Recommendation G.728)

Excitation shape and gain codebook tables

This annex first gives the 7-bit excitation VQ shape codebook table. Each row in the table specifies one of the 128 shape codevectors. The first column is the channel index associated with each shape codevector (obtained by a Gray-code index assignment algorithm). The second through the sixth columns are the first through the fifth components of the 128 shape codevectors as represented in the 16-bit fixed point. To obtain the floating point value from the integer value, divide the integer value by 2 048. This is equivalent to multiplication by 2^{-11} or shifting the binary point 11 bits to the left.

Channel index	Codevector components				
0	668	—2 950	—1 254	—1 790	—2 553
1	—5 032	—4 577	—1 045	2 908	3 318
2	—2 819	—2 677	—948	—2 825	—4 450
3	—6 679	—340	1 482	—1 276	1 262
4	—562	—6 757	1 281	179	—1 274
5	—2 512	—7 130	—4 925	6 913	2 411
6	—2 478	—156	4 683	—3 873	0
7	—8 208	2 140	—478	—2 785	533
8	1 889	2 759	1 381	—6 955	—5 913
9	5 082	—2 460	—5 778	1 797	568
10	—2 208	—3 309	—4 523	—6 236	—7 505
11	—2 719	4 358	—2 988	—1 149	2 664
12	1 259	995	2 711	—2 464	—10 390
13	1 722	—7 569	—2 742	2 171	—2 329
14	1 032	747	—858	—7 946	—12 843
15	3 106	4 856	—4 193	—2 541	1 035
16	1 862	—960	—6 628	410	5 882
17	—2 493	—2 628	—4 000	—60	7 202
18	—2 672	1 446	1 536	—3 831	1 233
19	—5 302	6 912	1 589	—4 187	3 665
20	—3 456	—8 170	—7 709	1 384	4 698
21	—4 699	—6 209	—11 176	8 104	16 830
22	930	7 004	1 269	—8 977	2 567
23	4 649	11 804	3 441	—5 657	1 199
24	2 542	—183	—8 859	—7 976	3 230
25	—2 872	—2 011	—9 713	—8 385	12 983
26	3 086	2 140	—3 680	—9 643	—2 896
27	—7 609	6 515	—2 283	—2 522	6 332
28	—3 333	—5 620	—9 130	—11 131	5 543
29	—407	—6 721	—17 466	—2 889	11 568
30	3 692	6 796	—262	—10 846	—1 856
31	7 275	13 404	—2 989	—10 595	4 936
32	244	—2 219	2 656	3 776	—5 412
33	—4 043	—5 934	2 131	863	—2 866
34	—3 302	1 743	—2 006	—128	—2 052
35	—6 361	3 342	—1 583	—21	1 142
36	—3 837	—1 831	6 397	2 545	—2 848

<<

Channel index	Codevector components					
37	—9 332	—6 528	5 309	1 986	—2 245	
38	—4 490	748	1 935	—3 027	—493	
39	—9 255	5 366	3 193	—4 493	1 784	
40	4 784	—370	1 866	1 057	—1 889	
41	7 342	—2 690	—2 577	676	—611	
42	—502	2 235	—1 850	—1 777	—2 049	
43	1 011	3 880	—2 465	2 209	—152	
44	2 592	2 829	5 588	2 839	—7 306	
45	—3 049	—4 918	5 955	9 201	—4 447	
46	697	3 908	5 798	—4 451	—4 644	
47	—2 121	5 444	—2 570	321	—1 202	
48	2 846	—2 086	3 532	566	—708	
49	—4 279	950	4 980	3 749	452	
50	—2 484	3 502	1 719	—170	238	
51	—3 435	263	2 114	—2 005	2 361	
52	—7 338	—1 208	9 347	—1 216	—4 013	
53	—13 498	—439	8 028	—4 232	361	
54	—3 729	5 433	2 004	—4 727	—1 259	
55	—3 986	7 743	8 429	—3 691	—987	
56	5 198	—423	1 150	—1 281	816	
57	7 409	4 109	—3 949	2 690	30	
58	1 246	3 055	—35	—1 370	—246	
59	—1 489	5 635	—678	—2 627	3 170	
60	4 830	—4 585	2 008	—1 062	799	
61	—129	717	4 594	14 937	10 706	
62	417	2 759	1 850	—5 057	—1 153	
63	—3 887	7 361	—5 768	4 285	666	
64	1 443	—938	20	—2 119	—1 697	
65	—3 712	—3 402	—2 212	110	2 136	
66	—2 952	12	—1 568	—3 500	—1 855	
67	—1 315	—1 731	1 160	—558	1 709	
68	88	—4 569	194	—454	—2 957	
69	—2 839	—1 666	—273	2 084	—155	
70	—189	—2 376	1 663	—1 040	—2 449	
71	—2 842	—1 369	636	—248	—2 677	
72	1 517	79	—3 013	—3 669	—973	
73	1 913	—2 493	—5 312	—749	1 271	
74	—2 903	—3 324	—3 756	—3 690	—1 829	
75	—2 913	—1 547	—2 760	—1 406	1 124	
76	1 844	—1 834	456	706	—4 272	
77	467	—4 256	—1 909	1 521	1 134	
78	—127	—994	—637	—1 491	—6 494	
79	873	—2 045	—3 828	—2 792	—578	
80	2 311	—1 817	2 632	—3 052	1 968	
81	641	1 194	1 893	4 107	6 342	
82	—45	1 198	2 160	—1 449	2 203	

>>

<<

Channel index	Codevector components				
83	—2 004	1 713	3 518	2 652	4 251
84	2 936	—3 968	1 280	131	—1 476
85	2 827	8	—1 928	2 658	3 513
86	3 199	—816	2 687	—1 741	—1 407
87	2 948	4 029	394	—253	1 298
88	4 286	51	—4 507	—32	—659
89	3 903	5 646	—5 588	—2 592	5 707
90	—606	1 234	—1 607	—5 187	664
91	—525	3 620	—2 192	—2 527	1 707
92	4 297	—3 251	—2 283	812	—2 264
93	5 765	528	—3 287	1 352	1 672
94	2 735	1 241	—1 103	—3 273	—3 407
95	4 033	1 648	—2 965	—1 174	1 444
96	74	918	1 999	915	—1 026
97	—2 496	—1 605	2 034	2 950	229
98	—2 168	2 037	15	—1 264	—208
99	—3 552	1 530	581	1 491	962
100	—2 613	—2 338	3 621	—1 488	—2 185
101	—1 747	81	5 538	1 432	—2 257
102	—1 019	867	214	—2 284	—1 510
103	—1 684	2 816	—229	2 551	—1 389
104	2 707	504	479	2 783	—1 009
105	2 517	—1 487	—1 596	621	1 929
106	—148	2 206	—4 288	1 292	—1 401
107	—527	1 243	—2 731	1 909	1 280
108	2 149	—1 501	3 688	610	—4 591
109	3 306	—3 369	1 875	3 636	—1 217
110	2 574	2 513	1 449	—3 074	—4 979
111	814	1 826	—2 497	4 234	—4 077
112	1 664	—220	3 418	1 002	1 115
113	781	1 658	3 919	6 130	3 140
114	1 148	4 065	1 516	815	199
115	1 191	2 489	2 561	2 421	2 443
116	770	—5 915	5 515	—368	—3 199
117	1 190	1 047	3 742	6 927	—2 089
118	292	3 099	4 308	—758	—2 455
119	523	3 921	4 044	1 386	85
120	4 367	1 006	—1 252	—1 466	—1 383
121	3 852	1 579	—77	2 064	868
122	5 109	2 919	—202	359	—509
123	3 650	3 206	2 303	1 693	1 296
124	2 905	—3 907	229	—1 196	—2 332
125	5 977	—3 585	805	3 825	—3 138
126	3 746	—606	53	—269	—3 301
127	606	2 018	—1 316	4 064	398

Next we give the values for the gain codebook. This table not only includes the values for GQ, but also the values for GB, G2 and GSQ as well. Both GQ and GB can be represented exactly in 16-bit arithmetic using Q13 format. The fixed point representation of G2 is just the same as GQ, except the format is now Q12. An approximate representation of GSQ to the nearest integer in fixed point Q12 format will suffice.

Values of gain codebook related arrays

Array index	1	2	3	4	5	6	7	8
GQ [b]	0.515625	0.90234375	1.579101563	2.763427734	— GQ(1)	— GQ(2)	— GQ(3)	— GQ(4)
GB	0.708984375	1.240722656	2.171264649	[a]	— GB(1)	— GB(2)	— GB(3)	[a]
G2	1.03125	1.8046875	3.158203126	5.526855468	— G2(1)	— G2(2)	— G2(3)	— G2(4)
GSQ	0.26586914	0.814224243	2.493561746	7.636532841	GSQ(1)	GSQ(2)	GSQ(3)	GSQ(4)

[a] Can be any arbitrary value (not used).

[b] Note that GQ(1) = 33/64, and GQ(i) = (7/4) GQ(i — 1) for i = 2, 3, 4.

ANNEX C

(to Recommendation G.728)

Values for bandwidth broadcasting

The following table gives the integer values for the pole control, zero control and bandwidth broadening vectors listed in Table 2/G.728. To obtain the floating point value, divide the integer value by 16 384. The values in this table represent these floating point values in the Q14 format, the most commonly used format to represent numbers less than 2 in 16-bit fixed point arithmetic.

i	FACV	FACGPV	WPCFV	WZCFV	SPFPCFV	SPFZCFV
1	16 384	16 384	16 384	16 384	16 384	16 384
2	16 192	14 848	9 830	14 746	12 288	10 650
3	16 002	13 456	5 898	13 271	9 216	6 922
4	15 815	12 195	3 539	11 944	6 912	4 499
5	15 629	11 051	2 123	10 750	5 184	2 925
6	15 446	10 015	1 274	9 675	3 888	1 901
7	15 265	9 076	764	8 707	2 916	1 236
8	15 086	8 225	459	7 836	2 187	803
9	14 910	7 454	275	7 053	1 640	522
10	14 735	6 755	165	6 347	1 230	339
11	14 562	6 122	99	5 713	923	221
12	14 391					
13	14 223					
14	14 056					
15	13 891					

i	FACV	FACGPV	WPCFV	WZCFV	SPFPCFV	SPFZCFV
16	13 729					
17	13 568					
18	13 409					
19	13 252					
20	13 096					
21	12 943					
22	12 791					
23	12 641					
24	12 493					
25	12 347					
26	12 202					
27	12 059					
28	11 918					
29	11 778					
30	11 640					
31	11 504					
32	11 369					
33	11 236					
34	11 104					
35	10 974					
36	10 845					
37	10 718					
38	10 593					
39	10 468					
40	10 346					
41	10 225					
42	10 105					
43	9 986					
44	9 869					
45	9 754					
46	9 639					
47	9 526					
48	9 415					
49	9 304					
50	9 195					
51	9 088					

ANNEX D

(to Recommendation G.728)

Coefficients of the 1 kHz lowpass elliptic filter used in pitch period extraction module (block 82)

The 1 kHz lowpass filter used in the pitch lag extraction and encoding module (block 82) is a third-order pole-zero filter with a transfer function of

$$L(z) = \frac{\sum_{i=0}^{3} b_i z^{-i}}{1 + \sum_{i=1}^{3} a_i z^{-i}}$$

where the coefficients a_i and b_i are given in the following tables.

i	a_i	b_i
0	—	0.0357081667
1	–2.34036589	–0.0069956244
2	2.01190019	–0.0069956244
3	–0.614109218	0.0357081667

ANNEX E

(to Recommendation G.728)

Time scheduling the sequence of computations

All of the computation in the encoder and decoder can be divided up into two classes. Included in the first class are those computations which take place once per vector. Sections 3 through § 5.14 note which computations these are. Generally they are the ones which involve or lead to the actual quantization of the excitation signal and the synthesis of the output signal. Referring specifically to the block numbers in Figure 2/G.728, this class includes blocks 1, 2, 4, 9, 10, 11, 13, 16, 17, 18, 21 and 22. In Figure 3/G.728, this class includes blocks 28, 29, 31, 32 and 34. In Figure 6/G.728, this class includes blocks 39, 40, 41, 42, 46, 47, 48 and 67. (Note that Figure 6/G.728 is applicable to both block 20 in Figure 2/G.728 and block 30 in Figure 3/G.728. Blocks 43, 44 and 45 of Figure 6/G.728 are not part of this class. Thus, blocks 20 and 30 are part of both classes.

In the other class are those computations which are only done once for every four vectors. Once more referring to Figures 2/G.728 through 8/G.728, this class includes blocks 3, 12, 14, 15, 23, 33, 35, 36, 37, 38, 43, 44, 45, 49, 50, 51, 81, 82, 83, 84 and 85. All of the computations in this second class are associated with updating one or more of the adaptive filters or predictors in the coder. In the encoder there are three such adaptive structures, the 50th order LPC synthesis filter, the vector gain predictor, and the perceptual weighting filter. In the decoder there are four such structures, the synthesis filter, the gain predictor, and the long-term and short-term adaptive postfilters. Included in the descriptions of § 3 through § 5.14 are the times and input signals for each of these five adaptive structures. Although it is redundant, this annex explicitly lists all of this timing information in one place for the convenience of the reader. Table E-1/G.728 summarizes the five adaptive structures, their input signals, their times of computation and the time at which the updated values are first used. For reference, the fourth column in Table E-1/G.728 refers to the block numbers used in the figures and in §§ 3 to 5 as a cross reference to these computations.

TABLE E-1/G.728

Timing of adapter updates			
Adapter	Input signal(s)	First use of updated parameters	Reference blocks
Backward synthesis filter adapter	Synthesis filter output speech (ST) through vector 4	Encoding/decoding vector 3	23, 33, (49,50,51)
Backward vector gain adapter	Log gains through vector 1	Encoding/decoding vector 2	20, 30 (43,44,45)
Adapter for perceptual weighting filter and fast codebook search	Input speech (S) through vector 2	Encoding vector 3	3 (36,37,38) 12, 14, 15
Adapter for long-term adaptive postfilter	Synthesis filter output speech (ST) through vector 3	Synthesizing postfiltered vector 3	35 (81 to 84)
Adapter for short-term adaptive postfilter	Synthesis filter output speech (ST) through vector 4	Synthesizing postfiltered vector 1	35 (85)

By far, the largest amount of computation is expended in updating the 50th order synthesis filter. The input signal required is the synthesis filter output speech (ST). As soon as the fourth vector in the previous cycle has been decoded, the hybrid window method for computing the autocorrelation coefficients can commence (block 49). When it is completed, Durbin's recursion to obtain the prediction coefficients can begin (block 50). In practice we found it necessary to stretch this computation over more than one vector cycle. We begin the hybrid window computation before vector 1 has been fully received. Before Durbin's recursion can be fully completed, we must interrupt it to encode vector 1. Durbin's recursion is not completed until vector 2. Finally bandwidth expansion (block 51) is applied to the predictor coefficients. The results of this calculation are not used until the encoding or decoding of vector 3 because in the encoder we need to combine these updated values with the update of the perceptual weighting filter and codevector energies. These updates are not available until vector 3.

The gain adaptation precedes in two fashions. The adaptive predictor is updated once every four vectors. However, the adaptive predictor produces a new gain value once per vector. In this section we are describing the timing of the update of the predictor. To compute this requires first performing the hybrid window method on the previous log gains (block 43), then Durbin's recursion (block 44), and bandwidth expansion (block 45). All of this can be completed during vector 2 using the log gains available up through vector 1. If the result of Durbin's recursion indicates there is no singularity, then the new gain predictor is used immediately in the encoding of vector 2.

The perceptual weighting filter update is computed during vector 3. The first part of this update is performing the LPC analysis on the input speech up through vector 2. We can begin this computation immediately after vector 2 has been encoded, not waiting for vector 3 to be fully received. This consists of performing the hybrid window method (block 36), Durbin's recursion (block 37) and the weighting filter coefficient calculations (block 38). Next we need to combine the perceptual weighting filter with the updated synthesis filter to compute the impulse response vector calculator (block 12). We also must convolve every shape codevector with this impulse response to find the codevector energies (blocks 14 and 15). As soon as these computations are completed, we can immediately use all of the updated values in the encoding of vector 3.

Note — Because the computation of codevector energies is fairly intensive, we were unable to complete the perceptual weighting filter update as part of the computation during the time of vector 2, even if the gain predictor update were moved elsewhere. This is why it was deferred to vector 3.

The long-term adaptive postfilter is updated on the basis of a fast pitch extraction algorithm which uses the synthesis filter output speech (ST) for its input. Since the postfilter is only used in the decoder, scheduling time to perform this computation was based on the other computational loads in the decoder. The decoder does not have to update the perceptual weighting filter and codevector energies, so the time slot of vector 3 is available. The codeword for vector 3 is decoded and its synthesis filter output speech is available together with all previous synthesis output vectors. These are input to the adapter which then produces the new pitch period (blocks 81 and 82) and long-term postfilter coefficient (blocks 83 and 84). These new values are immediately used in calculating the postfiltered output for vector 3.

The short-term adaptive postfilter is updated as a by-product of the synthesis filter update. Durbin's recursion is stopped at order 10 and the prediction coefficients are saved for the postfilter update. Since the Durbin computation is usually begun during vector 1, the short-term adaptive postfilter update is completed in time for the postfiltering of output vector 1.

ANNEX F

(to Recommendation G.728)

Alphabetical list of abbreviations used in this Recommendation

CELP	Code excited linear prediction
DCME	Digital circuit multiplication equipment
DSP	Digital signal processing
LD-CELP	Low-delay code excited linear prediction
LPC	Linear prediction coding
MSE	Mean-squared error
PCM	Pulse code modulation
RMS	Root-mean-square
VQ	Vector quantization
WNCF	White noise correction factor

APPENDIX I

(to Recommendation G.728)

Implementation verification

A set of verification tools have been designed in order to facilitate the compliance verification of different implementations to the algorithm defined in this Recommendation. These verification tools are available from the ITU on a set of distribution diskettes.