INTERNATIONAL TELECOMMUNICATION UNION

ITU-T G.728 – Annex G

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

(11/94)

GENERAL ASPECTS OF DIGITAL TRANSMISSION SYSTEMS

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

ANNEX G: 16 kbit/s FIXED POINT SPECIFICATION

ITU-T Recommendation G.728 – Annex G

(Previously "CCITT Recommendation")

FOREWORD

The ITU-T (Telecommunication Standardization Sector) is a permanent organ of the International Telecommunication Union (ITU). The ITU-T 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 World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1 (Helsinki, March 1-12, 1993).

ITU-T Recommendation G.728 – Annex G was prepared by ITU-T Study Group 15 (1993-1996) and was approved under the WTSC Resolution No. 1 procedure on the 1st of November 1994.

NOTE

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

CONTENTS

Page

Annex G – 16 kbit/s fixed point specification 1
G.1 Introduction 1
G.1.1 General philosophy 1
G.1.2 Numerical representation 2
G.1.3 Arithmetic operations 3
G.2 Algorithmic changes 8
G.2.1 Changes in the backward vector gain adapter (block 20) 8
G.2.2 Changes in the Levinson-Durbin recursion modules 12
G.3 Pseudo-code for other modules of Recommendation G.728 18
G.3.1 Block 4 – Pseudo-code for weighting filter 20
G.3.2 Blockzir – Pseudo-code for synthesis and perceptual weighting filters during zero-input response computation 21
G.3.3 Blocks 9 and 10 – Pseudo-code for synthesis and perceptual weighting filter memory updates 23
G.3.4 Block 11 – VQ target vector computation 26
G.3.5 Block 12 – Impulse response vector calculation 26
G.3.6 Block 13 – Time-reversed convolution 27
G.3.7 Block 14 – Shape codevector convolution and energy calculation 27
G.3.8 Block 16 – VQ target vector normalization 28
G.3.9 Block 17 – VQ search error calculator and best codebook index selector 29
G.3.10 Block 19 – Excitation VQ codebook and block 21 – Gain scaling unit 30
G.3.11 Block 32 – Decoder synthesis filter 31
G.3.12 Block 36 – Pseudo-code for hybrid windowing module 33
G.3.13 Block 38 – Weighting filter coefficient calculator 35
G.3.14 Block 43 – Hybrid windowing module 36
G.3.15 Block 45 – Bandwidth expansion module 38
G.3.16 Block 46 – Log-gain linear prediction 39
G.3.17 Block 49 – Hybrid window module for synthesis filter 41
G.3.18 HWMCORE – Core of hybrid window module 43
G.3.19 Block 51 – Bandwidth expansion module 47
G.3.20 Blocks 71 and 72 – Long-term and short-term postfilters 48
G.3.21 Blocks 73 and 74 – Sum of absolute value calculators 49
G.3.22 Block 75 – Scaling factor calculator 50
G.3.23 Block 76 – First-order lowpass filter and block 77 – Output gain scaling unit 50
G.3.24 Block 81 – 10th order LPC inverse filter 51
G.3.25 Block 82 – Pitch period extraction module 51
G.3.26 Block 83 – Pitch predictor tap calculator 54
G.3.27 Block 84 – Long-term postfilter coefficient calculator 55
G.3.28 Block 85 – Short-term postfilter coefficient calculator 56
G.4 LD-CELP internal variable representations 57
G.5 Log-gain tables for gain and shape codebook vectors 60
G.6 Integer values of gain codebook related arrays 62
G.7 Encoder and decoder main program pseudo-codes 62

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

(Geneva, 1992)

Annex G

16 kbit/s fixed point specification

(Geneva, 1994)

(This annex forms an integral part of this Recommendation)

G.1 Introduction

The purpose of this annex is to describe in sufficient detail how ITU-T Recommendation G.728 for 16 kbit/s LD-CELP can be implemented on a fixed point arithmetic device. A fixed point implementation based on this description should be capable of fully interworking with a floating point version of Recommendation G.728 and producing an output signal of equivalent quality, whether that signal is speech or an in-band data signal. By fixed point arithmetic we mean a 16-bit word size. Most 16-bit devices have other word sizes as well. For example, the product of two 16-bit words is a 32-bit word. So, the product register of such a device is typically 32 bits wide. The accumulator stores the sum of products, so it must also be at least 32 bits wide. Thus, although we are describing a "16-bit implementation," some internal state variables have other than 16-bit precision.

It is the intent of this annex to provide a complete bit exact description of all operations necessary for the implementation of Recommendation G.728 on a 16-bit fixed point digital signal processor having a 32-bit product register and at least two 32-bit (or greater) accumulators. In numerous instances throughout the annex there are possible alternative methods to perform operations such that the exact same result is obtained. In such instances the alternate method may be substituted. However, if the exact same result is not obtained for all possible inputs, then the substitution should not be made. Since the number of possible alternatives is very large, no attempt has been made to point out the great majority of them.

This annex is divided into seven subclauses. The first subclause is an introduction and contains further information about fixed point signal processing and the conventions used throughout this annex. The second subclause contains information about algorithmic changes which were made especially for fixed point implementation of Recommendation G.728. The third subclause gives fixed point pseudo-code for the remaining modules of the coder. The fourth subclause provides an overall summary of state variable representations for the fixed point coder. The last subclauses contain tables pertaining to the backward vector gain adapter.

G.1.1 General philosophy

This annex is an annex to ITU-T Recommendation G.728. It is therefore unnecessary to repeat all of the details and discussions in that Recommendation. Where it is helpful, some of the details will be reviewed. In that Recommendation, complete computational details were given for a floating point implementation. Where the computational details are unchanged except for the substitution of fixed point arithmetic operations for floating point, no computational details will be given in this annex.

The greatest changes from the floating point version of the coder to this one are:

1) the introduction of different types of arithmetic operations and precisions for the state variables;

2) a changed, but mathematically equivalent method for the backward vector gain adaptation; and

3) the introduction of variable precision in the calculation of the predictor coefficients in the Levinson-Durbin recursion.

The remainder of this first subclause of the annex gives details on the different numerical representations and fixed point arithmetic. The second subclause of the annex gives details on the two major algorithmic changes mentioned above, the backward vector gain adaptation and the Levinson-Durbin recursion. The third subclause of the annex gives pseudo-code for the hybrid windowing module, block 49 in Recommendation G.728. The algorithm for this module is unchanged, but the implementation is complicated by the use of fixed point arithmetic. The pseudo-code for this module is a good example of the types of changes which must be made throughout the other modules in the coder. The fourth subclause of the annex contains a table corresponding to Table 2/G.728 giving the numerical representation of all state variables used in the encoder and decoder.

For consistency in this annex, all representations assume that 2's complement arithmetic is used throughout. Alternative representations which can produce mathematically equivalent results can be used to implement the coder.

G.1.2 Numerical representation

The basic unit of a 16-bit fixed point implementation is the 16-bit word. When representing pure integers, it has a range of –32768 to +32767. The representation for 1 is given by 0000000000000001 and the representation for –32768 is given by 1000000000000000. Here the right-most bit represents the least significant bit (LSB) and the left-most bit represents the most significant bit (MSB). For 2's complement arithmetic, if the MSB is 0, the number is positive, while if the MSB is 1, the number is negative. We can number the bits from 0 to 15, with bit 0 being the LSB and bit 15 the MSB.

To represent numbers with fractional parts, a decimal point must be assigned between two of the bits. For example, to represent numbers between –1.0 and +1.0, we would assign the decimal point between bits 14 and 15. This particular format is called Q15 because there are 15 bits to the right of the decimal point. Qn format is defined to have n bits to the right of the decimal point. Purely integer data would be represented by Q0 format.

Some data requires a greater precision than representation by a 16-bit word. To accommodate such data, double precision format is defined. This means that there are 32 bits of information. Whereas 16-bit words are capable of representing data with a precision of 1 in 2^{15}, 32-bit registers such as the product register or the accumulator on most commercially available DSP chips can represent data with a precision of 1 in 2^{31}. Such words are referred to as double precision. Once again, there must be a decimal point to indicate the dynamic range of the word as well.

Some data has a greater range than can be represented by any fixed 16-bit format. Perhaps 16 bits of precision is adequate, but the scaling of the value must be dynamic. Such data can be represented by single precision floating point. This means that the data is represented by two words. The first 16-bit word contains a number whose magnitude falls between 16384 and 32767. This is the mantissa of the value and we say that its value is represented in normalized format because of the range of its magnitude. If the value is positive, then bit 14 of the mantissa is a 1. The second word contains the number of left shifts (NLS) used to put the value in normalized format. Thus, the second word specifies the Q format of the mantissa. If this format is used for a single value, it is called scalar floating point.

It is also possible to represent an array of n values with n + 1 words using block floating point. Using this format, the largest magnitude value in the array would be represented the same way as just described for scalar floating point. All other values in the array would share the same NLS. Their mantissas would not necessarily be in normalized format. An extension of this representation is segmented block floating point. In this case an array of mn values is represented by m(n + 1) words. The array is subdivided into m sub-arrays of size n and each sub-array is represented in block floating point with n words representing the magnitudes and 1 word representing the NLS.

The other type of representation used is double precision floating point. In this instance double precision integers are used for the mantissas and one single precision word is used to represent the NLS. In summary, the different types of representations used are single precision fixed point, double precision fixed point for the accumulators and product register, scalar single precision floating point, and single and double precision block floating point formats.

G.1.3 Arithmetic operations

In multiplying two 16-bit words, the result is a 32-bit number. This is the reason that product registers are customarily double precision. Since product registers can be added the accumulators, the accumulators must also be at least 32 bits wide. For a sum of products type of computation, as in convolution or FIR filtering, the accumulator could overflow. This problem of overflow is addressed differently in commercially available DSP chips.

In IIR filtering, the sum of products, or the result of the multiply-accumulate operations, becomes part of the memory for the filter and is used again the next time the filtering operation is performed. Specifically, the 16 bits in the high word of the output will be used as an input to the multiplier. An overflow which converts a large positive value to a large negative value or vice versa is known as wrap around and will cause a big difference in the output of the filter. To guard against this, we use saturation mode arithmetic for all IIR filters and anywhere else that a sum of products will later be used as an input for a multiplier. Saturation mode means that if the high word becomes greater than 32767 or less than –32768, it will be clipped to these values in order to prevent wrap around.

G.1.3.1 Shifting and rounding

In discussing arithmetic operations, we begin with shifting and rounding. If we multiply a Qn format value by a Qm format value number, the result in the product register will have double precision format Q(n + m). If the result needs to be stored or added at a different precision, then the result must be shifted and/or rounded to the correct precision.

Two types of shifts are possible, left shifts and right shifts. On commercially available DSP chips, shifts can usually be done in the accumulator. Also, it is usually possible to shift the result in a product register before adding it to or storing it in the accumulator. As their names imply, in a left shift, the bits are moved to the left and in a right shift they are moved to the right. If we shift a value k bits to the right, then the least significant k bits of the old value are lost. If we shift a value to the left, we need to check for possible overflows. The expression to indicate a right shift of k bits for a variable TMP is

$$TMP = TMP >> k$$

and the expression for a left shift of k bits is given by

$$TMP = TMP << k$$

In some cases k is a variable and can even be negative. In those instances when k is negative, a left shift by k bits is defined to be a right shift by –k bits. Similarly, a right shift by k bits when k is negative is equivalent to a left shift by –k bits. Where the possibility of k being negative exists, the pseudo-code includes a test for this possibility followed by the reverse shift by –k bits if k is negative. While negative shifts have been defined above mathematically, they cannot be implemented on most devices or in some computer languages.

It is worth noting one particular anomaly of right shifts for 2's complement arithmetic. Suppose that the value to be right shifted is 3 and the shift is 1 bit. The 16-bit representation of 3 is given by 0000000000000011. If we right shift this by one bit, we get 0000000000000001 = 1. If the value to be right shifted is –3, then the representation is 1111111111111101. After a right shift, the result is 1111111111111110 = –2. The first item to note is that for right shifting, the sign bit is extended. The anomaly is that the magnitude of the answers for these two examples do not agree. If a sign-magnitude representation were used, they would agree. Implementers should be aware of this difference.

In simulating the coder an additional, more subtle difference was found which is compiler dependent. It is possible that in the algorithm an instruction is generated to right shift a word by greater than the size of that word. For example, it could be to shift a 16-bit word by 18 bits. If the operation were implemented by doing 18 individual 1-bit right shifts, the result of such an operation should be 0 or –1, depending on the sign of the original data. However, it was found that some compilers consider an 18-bit shift to be an illegal instruction and produce spurious results. Implementers should verify how their target hardware and language compiler would handle such a case.

Rounding is the process of converting from double precision to single precision in the accumulator. Usually rounding is performed immediately preceding the storage of the value to a 16-bit word in memory. An accumulator consists of a high word and a low word (and possibly the additional bits to the left of the high word). Usually, either the high word or the low word can be stored to memory, or both on two successive instructions. If we consider the accumulator to have a decimal point placed between the high word and the low word, then rounding is the operation of converting the accumulator to the integer value closest to the non-integer value stored in the original two words. The usual convention for 2's complement numbers is to test the MSB in the low word. If it is 1, add 1 to the value in the high word. Then zero out the low word. For example, if the value in the accumulator is 1.5, the high word is given by 0000000000000001 and the low word is given by 1000000000000000. Since the MSB of the low word is 1, add 1 to the high word and zero out the low word. The result is 0000000000000010 for the high word, or 2. If the value in the accumulator is –1.5, then the high word is given by 1111111111111110 and the low word is given by 1000000000000000. Since the MSB of the low word is 1, add 1 to the high word and then zero the low word. The result is 1111111111111111 = –1. This is similar to the anomaly for right shifts.

In performing the rounding function it is necessary to be aware of the possibility of overflow. For example, if the high word value is 0111111111111111 (= 32767) and the low word has a 1 in the MSB, then following the usual convention results in an overflow. Depending on the processor, the output word could become 1000000000000000 which represents –32768. In such a case, the usual convention is not followed. Instead the value is saturated to avoid an unrepresentable value.

In the pseudo-code examples, the rounding function described above is represented as RND (.).

Pseudo-code for VSCALE

One new module of pseudo-code which needs to be introduced at this point performs vector scaling for block floating point representation. The name given to this module is VSCALE. Its purpose is to scale a vector so that the largest magnitude of its elements is left justified as desired, i.e. represented in normalized format. This module can be used for vectors where the first element is known to have the largest element or for vectors where the location of the largest element is unknown. The inputs to VSCALE are IN, the input vector to be scaled, LEN, the length of the input vector, SLEN, the search length for finding the maximum value, and MLS, the maximum number of left shifts permitted. The outputs of VSCALE are OUT, the output vector, and NLS, the number of left shifts used for scaling the input vector. The input and output vectors are assumed to be of the same type and can be either single precision block floating point (16-bit integers) or double precision block floating point (32-bit integers). In the case of single precision vectors, MLS = 14, while for double precision vectors, MLS = 30. Sometimes, it is desired to use less than 16 bits or 32 bits to represent a variable. For example, there are several variables which are specified to have either 14 or 15 bits of precision. In these cases, set MLS = 12 or 13, respectively. Because of this possibility, there is also a possibility that rather than left shifts to normalize the variable, it will require right shifts. In those instances, the NLS value returned will be negative. For example, if NLS = –1 is returned, this indicates that a right shift of 1 bit was necessary. The module assumes that there is an accumulator (AA0) available for shifting and that it has at least 32 bits of precision. If the maximum element is known to be the first, set SLEN = 1. Otherwise, set SLEN = LEN and the entire vector will be searched for the maximum value.

The following code follows the convention that data is represented in 2's complement form. It treats the cases where largest magnitude values are positive or negative, separately.

```
SUBROUTINE VSCALE(IN, LEN, SLEN, MLS, OUT, NLS)
AA0 = IN(1)                                          | Find maximum positive value of input
AA1 = IN(1)                                          | Find maximum negative value of input
If SLEN = 1, skip the next 3 lines
    For I = 2, 3, ..., SLEN, do the next two lines
        If IN(I) > AA0, set AA0 = IN(I)
        If IN(I) < AA1, set AA1 = IN(I)

                                                     | Case 1: zero input vector
If AA0 = 0 and AA1 = 0, do the next 3 lines
    For I = 1, 2, ..., LEN, set OUT(I) = 0
    NLS = MLS + 1                                    | Let 0 have one more bit of
    Exit this subroutine                             | left shift than 1

NLS = 0                                              | Initialize NLS

                                                     | Determine Case 2 or Case 3
If AA0 < 0 or AA1 < –AA0, then do the following indented lines
                                                     | Case 2, negative is larger
    MAXI = –2^MLS                                    | Mantissa lower bound after shift
    MINI = 2 * MAXI
    If AA1 < MINI, then do the following doubly indented lines to find the number of right shifts needed and then scale the
    elements

LOOP1R:  AA1 = AA1 >> 1
         NLS = NLS – 1                               | Negative NLS = => right shifts
         If AA1 < MINI, go to LOOP1R
         For I = 1, 2, 3, ..., LEN, do the next line
             OUT(I) = IN(I) >> –NLS
         Exit this subroutine

LOOP1L:  If AA1 < MAXI, go to SCALE1                 | Find number of left shifts
         AA1 = AA1 << 1
         NLS = NLS + 1
         Go to LOOP1L

SCALE1:  For I = 1, 2, 3, ..., LEN, do the next line
             OUT(I) = IN(I) << NLS
         Exit this subroutine

    Else, do the following indented lines
                                                     | Case 3, positive number is larger

    MINI = 2^MLS                                     | Mantissa lower bound after shift
    MAXI = MINI – 1                                  | 2 * MIN will overflow if MLS = 30
    MAXI = MAXI + MINI                               | Mantissa upper bound
    If AA0 > MAXI, then do the following doubly indented lines to find the number of right shifts needed and then scale the
    elements

LOOP2R:  AA0 = AA0 >> 1
         NLS = NLS – 1
         If AA0 > MAXI, go to LOOP2R
         For I = 1, 2, 3, ..., LEN, do the next line
             OUT(I) = IN(I) >> –NLS
         Exit this subroutine

LOOP2L:  If AA0 ≥ MINI, go to SCALE2
         AA0 = AA0 << 1
         NLS = NLS + 1
         Go to LOOP2L

SCALE2:  For I = 1, 2, 3, ..., LEN, do the next line
             OUT(I) = IN(I) << NLS
         Exit this subroutine
```

In some instances we find that it is not actually desired to re-scale the data, but merely to find the number of left shifts required if one wanted to re-scale the data. The following routine uses the same inputs as VSCALE but provides only NLS as an output. It omits the scaling of the input vector, but is otherwise the same as VSCALE.

```
SUBROUTINE FINDNLS(IN, SLEN, MLS, NLS)
AA0 = IN(1)                                   | Find maximum positive value of input
AA1 = IN(1)                                   | Find maximum negative value of input
If SLEN = 1, skip the next 3 lines
    For I = 2, 3, ..., SLEN, do the next two lines
        If IN(I) > AA0, set AA0 = IN(I)
        If IN(I) < AA1, set AA1 = IN(I)

                                              | Case 1: zero input vector
If AA0 = 0 and AA1 = 0, do the next 2 lines
    NLS = MLS + 1                             | Let 0 have one more bit of
    Exit this subroutine                      | left shift than 1

NLS = 0                                       | Initialize NLS

                                              | Determine Case 2 or Case 3

If AA0 < 0 or AA1 < –AA0, then do the following indented lines
                                              | Case 2, negative is larger
    MAXI = –2^MLS                             | Mantissa lower bound after shift
    MINI = 2 * MAXI
    If AA1 < MINI, then do the following doubly indented lines to find the number of right shifts needed

LOOP1R:  AA1 = AA1 >> 1
         NLS = NLS – 1                        | Negative NLS = => right shifts
         If AA1 < MINI, go to LOOP1R
         Exit this subroutine

LOOP1L:  If AA1 < MAXI, exit this subroutine  | Find NLS
         AA1 = AA1 << 1
         NLS = NLS + 1
         Go to LOOP1L

    Else, do the following indented lines
                                              | Case 3, positive number is larger

    MINI = 2^MLS                              | Mantissa lower bound after shift
    MAXI = MINI – 1                           | 2 * MIN will overflow if MLS = 30
    MAXI = MAXI + MINI                        | Mantissa upper bound
    If AA0 > MAXI, then do the following doubly indented lines to find the number of right shifts needed

LOOP2R:  AA0 = AA0 >> 1
         NLS = NLS – 1
         If AA0 > MAXI, go to LOOP2R
         Exit this subroutine

LOOP2L:  If AA0 ≥ MINI, exit this subroutine  | Find NLS
         AA0 = AA0 << 1
         NLS = NLS + 1
         Go to LOOP2L
```

G.1.3.2 Multiplication

Multiplication of two fixed point numbers results in a 32-bit number, usually stored in a product register in a DSP. If the two fixed point numbers were in Qn and Qm formats, the result in the product register is in Q(n + m) format. Before adding it to an accumulator, it may be necessary to shift the result as explained in the preceding subclause.

Multiplication of two floating point words is accomplished by fixed point multiplication of the two mantissas and addition of the two NLS. As described above, the product is a 32-bit word with Q(n + m) format. If the product must be converted back to floating point, then the product may need to be renormalized. For example, in the case of multiplying two positive floating point words, the product must have a 1 in either bit 30 or 29. Renormalization is necessary if bit 30 is 0. This means one additional left shift is necessary. After the left shift, the product is represented in Q(n + m + 1) format. If the product is to be stored in scalar floating point, it must be rounded before storing. If the product needs to be in block floating point, the entire array needs to be renormalized, according to which value is now the largest in magnitude.

Although double precision variables are used in parts of this coder, there are no double precision multiplications. In some instances, double precision variables are multiplied, but in those cases, only the 16 most significant bits are used. These instances are noted in the pseudo-code.

G.1.3.3 Addition

Addition of fixed point numbers requires that both be stored in the same Q format. Generally, the value which is stored with the greater dynamic range determines which value must be changed to the appropriate Q format. For example, if adding values stored in Q9 and Q11 formats, the value in the Q11 format must be right shifted by 2 bits before adding it to the value stored in Q9 format.

Addition of scalar floating point numbers is similar. The two values must both have the same NLS. Again, the value with the higher NLS needs to be right shifted to match the other value's NLS. If the sum requires 17 bits for representation, the sum in the accumulator can be right shifted by 1 bit and then rounded back to 16 bits and the new NLS will be one less than the previous format. As an example, consider the case of adding two values whose NLS are 5 and 7. The value whose NLS is 7 must be right shifted by 2 bits before it can be added to the other value. If both values have the same sign, the sum of the two mantissas may have a magnitude greater than 32767. In this case, the value in the accumulator must be shifted by one bit and then rounded. The NLS of the sum will be 4. If the two values are of opposite sign, the result in the accumulator may have a mantissa whose magnitude is less than 16384. In this case, the result should be renormalized by left shifting until the magnitude is greater than or equal to 16384 and the NLS increased by the number of left shifts. For our example with the NLS being 5 and 7 initially, the final NLS can be no greater than 6 and no less than 4.

Addition of block floating point numbers is complicated by the fact that the constraints are based on the largest magnitude value. In this case, if two vectors have NLS of 5 and 7, the one with NLS of 7 must be right shifted by 2 bits. Each of the pairs is summed. The largest of the resulting sums will determine whether renormalization is necessary.

G.1.3.4 Division

Division is not used nearly as frequently as addition or multiplication. The only divisions used are scalar floating point divisions. The numerator and denominator are represented in normalized format, as is the quotient. The quotient's NLS is calculated by subtracting the NLS of the denominator from that of the numerator and adding 14. To explain this 14, consider the case where the numerator was slightly larger than the denominator and both had NLS equal 0. The quotient would have NLS equal 14 in this case and would be properly normalized. If the numerator's mantissa is less than the denominator's, then the numerator should be left shifted by 1 bit and its NLS increased by 1 in order to compute the NLS of the quotient. This guarantees that the mantissa of the quotient will be in normalized format.

Division occurs within Durbin's recursion, a routine requiring full 16-bit precision in the result. Therefore, approximate division routines are not sufficient. The mantissa of the result must have full 16-bit precision including rounding of the 17-bit result. Pseudo-code for such a division is given below.

If either the numerator or denominator is not initially stored in scalar floating point, it must first be converted to this format. The function FLOAT(.) is used in the pseudo-code to represent such conversions. The argument could be either single precision or double precision fixed point.

Pseudo-code for Floating Point Division

This routine is used for computing floating point division on a 16-bit fixed point device. It is assumed that there is at least one 32-bit accumulator available. All inputs and outputs are 16-bit words.

Input: NUM, NUMNLS, DEN, DENNLS

Output: QUO, QUONLS

Function: Compute the quotient. NUM and NUMNLS are the mantissa and Q format for the numerator. DEN and DENNLS are the mantissa and Q format for the denominator. QUO and QUONLS are the mantissa and Q format for the quotient. All are assumed to be in normalized format. There is no test for DEN being zero – it is assumed that it is not zero.

```
SUBROUTINE DIVIDE(NUM, MUMNLS, DEN, DENNLS, QUO, QUONLS)
SIGN = 1                                  | First determine the
P = NUM * DEN                             | sign bit of the
If P < 0, set SIGN = –1                   | quotient

QUONLS = NUMNLS – DENNLS + 14             | Next compute QUONLS
A0 = | NUM |                              | A0 is 32-bit accumulator
                                          | | NUM | is in lower 16 bits
A1 = | DEN |                              | A1 can be 16 or 32-bit register
                                          | if 32-bit, | DEN | is lower
                                          | 16 bits
If A0 < A1, do the next 2 lines
     QUONLS = QUONLS + 1
     A0 = A0 << 1

QUO = 0                                   | Quotient initialization
I = 0                                     | Loop counter initialization

LOOP: QUO = QUO << 1                      | Long division loop
If A0 ≥ A1, do the next 2 lines
     QUO = QUO + 1
     A0 = A0 – A1
A0 = A0 << 1
I = I + 1
If I < 15, GO TO LOOP

If A0 ≥ A1, set QUO = QUO + 1             | Take care of rounding

If SIGN < 0, set QUO = –QUO               | Take care of the sign
```

G.2 Algorithmic changes

G.2.1 Changes in the backward vector gain adapter (block 20)

NOTE – This subclause refers to 3.8/G.728. Readers should familiarize themselves with 3.8/G.728 before attempting to understand this subclause. The changes outlined in this subclause pertain to the once-per-vector computations for the backward vector gain adpater. Wherever possible, the same notation used in Recommendation G.728 has been used here.

In this subclause we briefly describe the once-per-vector backward vector gain adapter operations in Recommendation G.728 as implemented in floating point. We then describe a mathematically equivalent method which can be more easily and accurately implemented on fixed point processors. Tables for values required by this alternate method are given in the addendum to this annex.

The floating point operations can be described briefly as follows. The internal state variable array GSTATE, represented by the symbol δ, contains the previous 10 offset-removed logarithmic gains. The symbol $\delta(n)$ denotes the offset-removed logarithmic gain for vector n. The log-gain predictor output [the predicted version of $\delta(n)$] for vector n is given by:

$$\delta\$(n) = -\sum_{i=1}^{10} \alpha_i \, \delta(n-i) \qquad \text{(G-1)}$$

As shown in Figure 6/G.728, before converting $\hat{\delta}(n)$ to the linear domain, a gain offset of 32 dB must be added and the result checked to make sure that:

$$0 \leq \hat{\delta}(n) + 32 \leq 60 \tag{G-2}$$

Equivalently, we can say the allowed range for $\hat{\delta}(n)$ is:

$$-32 \leq \hat{\delta}(n) \leq 28 \tag{G-3}$$

The estimated gain in the linear domain is given by:

$$\sigma(n) = 10^{(\hat{\delta}(n) + 32)/20} \tag{G-4}$$

The value of $\sigma(n)$ is first used to normalize the excitation VQ target vector. After the codebook search is completed, $\sigma(n)$ is then used to scale the best codevector selected. If we assume that gain codebook index i and shape codebook index j were chosen for vector n, then the excitation vector $e(n)$ is given by:

$$e(n) = \sigma(n)\, g_i\, y_j \tag{G-5}$$

where y_j is the j-th shape codevector and g_i is the i-th gain level in the gain level in the gain codebook. The excitation vector $e(n)$ is then used to compute $\delta(n)$. First, we compute the squared RMS value of $e(n)$ [or the "power" of $e(n)$], which is given by:

$$P[e(n)] = \frac{1}{5} \sum_{k=1}^{5} e_k^2(n) \tag{G-6}$$

For any given vector x, we use the symbol $P[x]$ to represent the power of x, which is defined as the energy of x divided by the vector dimension of x. Before converting $P[e(n)]$ to the dB value in the logarithmic domain, we clip $P[e(n)]$ to 1 if it is less than 1. Thus, the allowed range for $P[e(n)]$ is:

$$P[e(n)] \geq 1 \tag{G-7}$$

This is to avoid overflow in the logarithm conversion or exceedingly small dB value. Note that although this range-limiting action is not explicitly shown in Figure 6/G.728, it is implemented in the "pseudo-code" in 5.7/G.728. The offset-removed logarithmic gain (in dB) for vector n is then obtained as:

$$\delta(n) = 10 \log_{10} P[e(n)] - 32 \tag{G-8}$$

Note that equation (G-7) implies that:

$$\delta(n) \geq -32 \tag{G-9}$$

Next, the $\delta(n)$ calculated in equation (G-8) is used to predict the following excitation gains and to update the log-gain predictor coefficients. This completes our brief review of the floating point operation for the backward vector gain adapter.

We now describe the mathematically equivalent method for fixed point implementation. Let y_{jk} be the k-th element of the j-th codevector in the shape codebook. Then, combining equations (G-5) and (G-6), we have:

$$P[e(n) = \sum_{k=1}^{5} \left(\sigma(n)\, g_i\, y_{jk}\right)^2 \tag{G-10}$$

$$= \sigma^2(n)\, g_i^2 \left[\sum_{k=1}^{5} y_{jk}^2\right] \tag{G-11}$$

$$= \sigma^2(n)\, g_i^2\, P[y_j] \tag{G-12}$$

Substituting equation (G-12) for equation (G-8) yields:

$$\delta(n) = 20 \log_{10} \sigma(n) - 32 + 20 \log_{10} |\, g_i\,| + 10 \log_{10} P[y_j] \tag{G-13}$$

Now, using equation (G-4), we can express $\delta(n)$ as:

$$\delta(n) = \hat{\delta}(n) + 20 \log_{10} |\, g_i\,| + \log_{10} P[y_j] \tag{G-14}$$

In other words, $\delta(n)$ is simply the predicted log-gain $\hat{\delta}(n)$ plus two "correction terms":

1) $20 \log_{10} |\, g_i\,|$, the dB value of the best gain level selected from the gain codebook; and

2) $10 \log_{10} P[y_j]$, the dB value of the power of the best shape codevector selected from the shape codebook. (In a sense, this is like a conventional predictive coder for the gain, but operated in the logarithmic domain.)

Figure G.1 shows the block schematic of this mathematically equivalent method. Since there are only 4 possible $|\, g_i\,|$ and 128 possible $P[y_j]$ values, we can precompute their dB values and store them in two log-gain tables (blocks 93 and 94 in Figure G.1).

The delay units 91 and 92 make available the best gain and shape codebook indices chosen in the excitation codebook search of the previous vector. These two indices are used to look up the values of $20 \log_{10} |\, g_i\,|$ and $10 \log_{10} P[y_j]$ from the log-gain tables in blocks 93 and 94. The 1-sample delay unit 95 holds the previous predicted (and possibly range limited) log-gain $\hat{\delta}(n-1)$. The adder 96 adds the outputs of blocks 93, 94, and 95 to produce an unclipped $\tilde{\delta}(n-1)$ according to equation (G-14). Then, the limiter 97 enforces the inequality in equation (G-9) by clipping the output of the adder 96 at –32 dB if it is less than –32 dB.

The ouput of the limiter 97 is mathematically equivalent to the output of the adder 42 in Figure 6/G.728. Therefore, blocks 43 through 46 in Figure G.1 are identical to their counterparts in Figure 6/G.728. The operation of the log-gain limiter 98 is similar to the limiter 47 in Figure 6/G.728, except that the allowed range has been shifted down by 32 dB. The adder 99 adds the log-gain offset value of 32 dB, stored in block 41, to the output of the log-gain limiter 98. The resulting log-gain value is then converted to the linear domain by the inverse logarithm calculator 48, which is identical to its counterparts in Figure 6/G.728. This completes the descriptions of the mathematically equivalent method for fixed point implementation.

The equivalent method shown in Figure G.1 has two important advantages over the original method in Figure 6/G.728.

a) It eliminates the need to calculate the logarithm function (block 40 in Figure 6/G.728). In DSP implementations, the logarithm function is usually calculated using a power series expansion and typically takes a large number of instruction cycles to calculate. Thus, replacing the logarithm calculation by a table look-up could mean a considerable saving in DSP cycles. Also, the table entries can be pre-computed to the maximum desired.

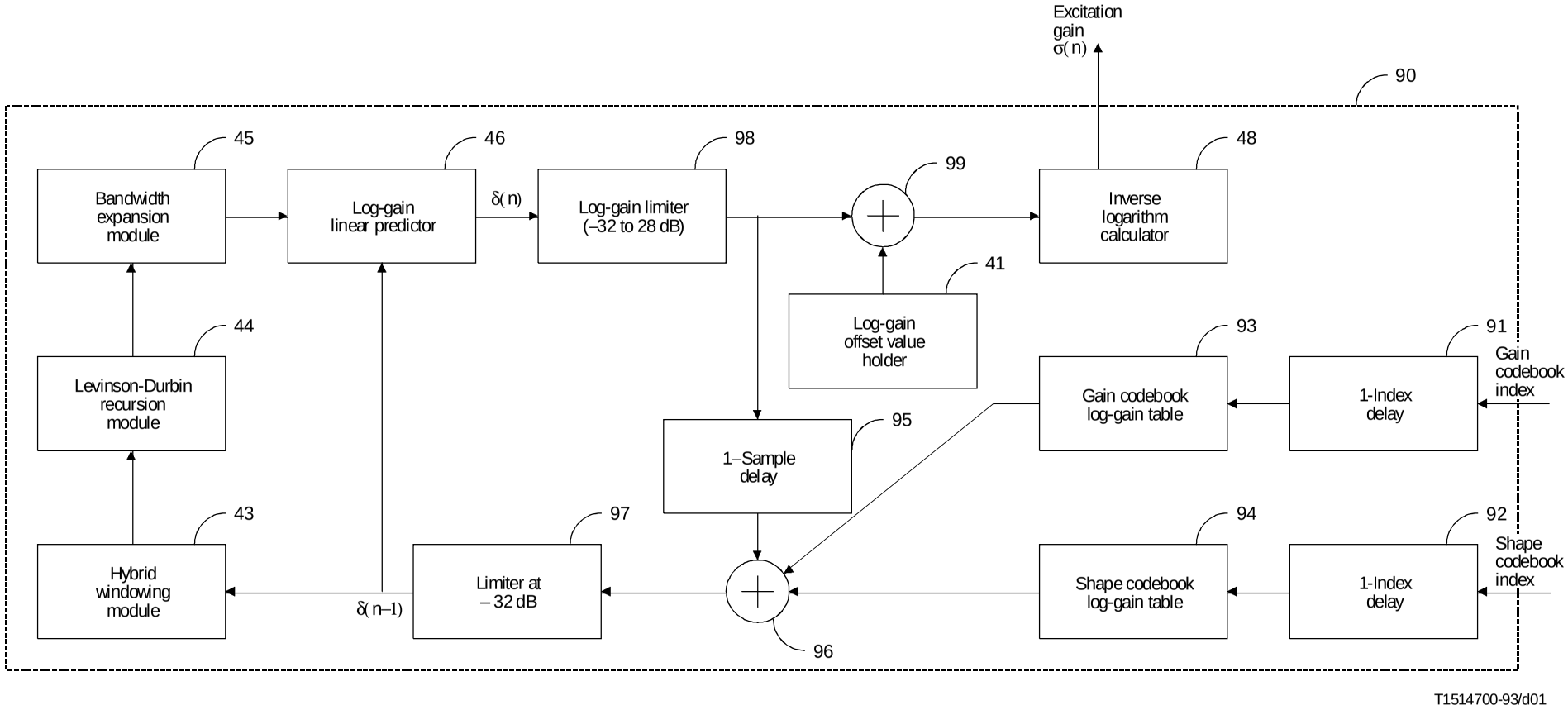

FIGURE G.1/G.728

Backward vector gain adapter for fixed point implementation

b) It is likely to give more accurate numerical results than the original method when a fixed point processor is used. Due to backward adaptation, there is a feedback loop in the gain adaptation process. In Figure 6/G.728, this feedback loop is very long. It goes from the inverse logarithm calculator 48 to the gain scaling unit 21 (in Figure 2/G.728), and then back to blocks 67, 39, 40 and 42 through 48. The more computations done in this loop, the more likely that numerical errors due to finite precision may accumulate in the feedback loop. This is especially true if the fixed point processor does not always achieve the maximum possible accuracy for the logarithm function. In contrast, the feedback loop in Figure G.1 is as tight as it can be. Note that the gain scaling unit, the energy and power calculation for $e(n)$, the logarithm calculator, and even the adder for restoring the log-gain offset are now all out of the feedback loop. Except for blocks 43 through 46 that are common in both methods, the feedback loop only involves two limiters and two additions, which can be implemented with very high precision by fixed point processors.

As a result of this change, interoperability between fixed and floating point implementations of Recommendation G.728 is enhanced. The main disadvantage of this new method is that it requires additional words of ROM memory. There are 128 shape vectors and 4 possible gain vectors. The additional memory required is 128 + 4 = 132 words. This is only a very small fraction of the ROM space already needed in Recommendation G.728.

This new method has changed the input to the backward vector gain adapter (blocks 20 and 30) from $e(n)$ to the gain and shape codebook indices i and j. To reflect this fact, Figures 1/G.728 through 3/G.728 should have been re-drawn here so that the backward vector gain adapter gets its input from the excitation VQ codebook block. However, such modified figures are omitted here, since the necessary change is trivial and it should be very clear from the description above.

G.2.2 Changes in the Levinson-Durbin recursion modules

This subclause is about changes in the Levinson-Durbin recursion modules used in Recommendation G.728. There are three such modules, designated as blocks 37, 44 and 50, and used for the perceptual weighting filter, the log-gain linear predictor, and the synthesis filter, respectively. Readers should refer to 5.5/G.728 and 5.6/G.728 for more details. In this section we will use the pseudo-code for block 50 from 5.6/G.728 as an example and show how it must be modified for fixed point implementation. Similar changes need to be made for blocks 37 (perceptual weighting filter) and 44 (log-gain predictor). We begin with a listing of the floating point pseudo-code.

```
If RTMP (LPC + 1) = 0, go to LABEL                | Skip if zero
                                                  |
If RTMP(1) ≤ 0, go to LABEL                       | Skip if zero signal
                                                  |


RC1 = –RTMP(2)/RTMP(1)
ATMP(1) = 1                                       |
ATMP(2) = RC1                                     | First-order predictor
ALPHATMP = RTMP(1) + RTMP(2) * RC1                |
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 2 lines
        N1 = MINC – IP + 2
        SUM = SUM + RTMP(N1) * ATMP(IP)


                                                  |
    RC = –SUM/ALPHATMP                            | Reflection coefficient
    MH = MINC/2 + 1                               |
    For IP = 2, 3, 4, ..., MH, do the next 4 lines
        IB = MINC – IP + 2
        AT = ATMP(IP) + RC * ATMP(IB)             |
        ATMP(IB) = ATMP(IB) + RC * ATMP(IP)       | Update predictor coefficient
        ATMP(IP) = AT                             |
```

```
        ATMP(MINC + 1) = RC                          |
        ALPHATMP = ALPHATMP + RC * SUM               | Prediction residual energy
        If ALPHATMP ≤ 0, go to LABEL                 | Abort if ill-conditioned
    Repeat the above for the next MINC

                                                     | Recursion completed normally
    Exit this program                                | 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.)

The best way to begin is to consider the floating point variables referred to in this pseudo-code. These are RC, RC1, RTMP, SUM, ALPHATMP and ATMP. (The other variables in the code, MINC, IP, IB and N1 are all indices which are integers.)

RC refers to the reflection coefficients, which are computed as an intermediate variable in this module. Reflection coefficients have the property that they have a magnitude which is always less than unity for a stable LPC filter. As such, RC can be represented with a Q15 format, meaning that one bit is used for the sign bit and the other 15 bits are used to represent the fractional part of the value.

We note that at each iteration we compute RC, use it for that iteration, and then never use it again. The only exception is that for the synthesis filter LPC analysis in the decoder. RC1 is saved for later use in the postfilter. In order to save memory, only the value of RC1 is saved above. All other values of RC are written to a single location which is overwritten at the next iteration. This represents a change from the original floating point pseudo-code, but has no effect on the output results and can be used for floating point implementations as well.

RTMP refers to the autocorrelation function values. These values have a tremendous dynamic range. By necessity RTMP must be kept in block floating point. This means that all values are normalized by the same power of 2. Theoretically, RTMP(1) should have the largest value. We also know that it must be positive. The representation used will be such that the largest magnitude of RTMP is between 0.5 and 1. This being the case, all of RTMP can be represented in Q15 format. All of RTMP is represented in block floating point in hybrid window, but only the mantissas are needed in Durbin's recursion.

One other note concerns RTMP(LPC + 1). As indicated on the first line, if this variable has a value of zero, this module should be terminated. If RTMP(LPC + 1) is represented by a 16-bit integer, this condition is much more likely to happen than for the case when the same RTMP (LPC + 1) is represented by a 32-bit floating point number in a floating point implementation. This causes interoperability problems. In computing RTMP(LPC + 1) in the previous module (hybrid window, block 49), the value is accumulated in the accumulator, which is at least 32 bits in all fixed point DSPs. It is proposed that this 32-bit value of the accumulator when the computation is completed be tested to check for zero. By making this change, premature termination (and thus interoperability problems) can be avoided. In the new code a logical variable named ILLCOND is tested to see whether it is true or false. Its value depends on the results of the test of RTMP (LPC + 1) in the hybrid window module. We later use ILLCOND as an output variable for this block to indicate whether the output values should be used or ignored.

For the postfilter, there is the possibility that the ill-conditioning occurred after the 10th iteration. In that case, a new set of short-term adaptive postfilter prediction coefficients have been determined and are valid, but the 50th order synthesis filter coefficients are not valid. A second logical variable, ILLCONDP indicates the status of the postfilter coefficients.

SUM and ALPHATMP are the next two variables. Both are values which are accumulated, but are never multiplied. The value held in an accumulator is 32 bits. However, these two variables are divided to compute RC. Both SUM and ALPHATMP are converted to 16-bit fixed point for the division. The result of the division is represented in Q15 fixed point format and assigned to RC. The variable SUM does not appear explicitly in the fixed point pseudo-code. In 32-bit format, it is the accumulator AA0 and in 16-bit format it is the variable SIGN.

Note that ALPHATMP is naturally a 32-bit number which is accumulated in an accumulator. However, in actual DSP implementations, it is necessary to save ALPHATMP in memory for the next higher order of recursion, because the accumulator will be needed for other computations before ALPHATMP is updated again. Therefore, some DSP cycles can be saved if we save and load only the rounded 16-bit high word of ALPHATMP rather than the entire 32-bit word. In practice, it is found that saving only the high word of ALPHATMP after each update did not degrade the coder's performance. Therefore, only the high word of ALPHATMP is saved after each update of ALPHATMP. In order to make clear when ALPHATMP is represented by 16 bits and when by 32, the name ALPHATMP is used in the pseudo-code only when it is a 16-bit number. An accumulator is referenced when it is a 32-bit number.

The remaining vector of variables is ATMP which represents the predictor coefficients. In all simulations, the maximum value observed for ATMP has been less than 4. This would suggest using Q13 format throughout. However, the same simulations also showed that using Q13 format within Durbin did not provide sufficient interoperability with floating point implementations. To achieve greater interoperability between fixed and floating point implementations, it was found to be better to use Q15 format except when this caused overflows.

On a DSP chip, the results of the computation of ATMP(IB) or ATMP(IP) are initially in the accumulator. The accumulators of most 16-bit fixed point DSP chips are at least 32 bits wide. The values of ATMP(IB) and ATMP(IP) must be rounded to 16-bit precision. In the fixed point code, it is important that the accumulator contains guard bits or provides an overflow flag, so that when computing ATMP(IB) or ATMP(IP) if overflow occurs, it will be detected.

The following strategy for choosing the format of ATMP has been adopted. The iterations can be numbered according to the value of MINC. We begin with MINC = 2 and Q15 format for ATMP. If overflow never occurs, i.e. for all IP, $|\ ATMP(IP)\ | < 1$, then the final representation for ATMP is Q15. The other possible case in that an overflow occurs during one of the iterations. Suppose that it is iteration K. In this case, all of the values of ATMP computed during iterations K and K – 1 must be converted to Q14 format by right shifting. Iteration K is then restarted using Q14 format. Subsequent iterations from K + 1 on are also computed using Q14 format. Overflows may also occur while using Q14 format. In that case, the same procedure is followed and the computation continues in Q13 format. Empirically it was observed that such overflows never occurred in Q13 format. The reason for using the other formats is that if the overflows are avoided, the result is more accurate. The final representation of ATMP before exiting this block is either Q13, Q14 or Q15.

It was also observed that after the bandwidth expansion operations, the filter coefficients at the output of the bandwidth expansion modules (blocks 38, 45, 51 and 85) are always representable in Q14 format. It was further observed that representation in Q15 format whenever possible did not improve cross-decoding SNR above that observed for using Q14 format. Therefore, those three bandwidth expansion modules always convert their output coefficient arrays to Q14, regardless of whether their input coefficient arrays (i.e. the Levinson-Durbin recursion modules output) are in Q13, Q14 or Q15. This means that when a Levinson-Durbin recursion module produces a Q13 or Q15 output coefficient array, it must signal the corresponding bandwidth expansion module so that an additional shift can be performed to convert the array to Q14. For this reason, in the fixed point Levinson-Durbin recursion module given below, we have added an additional flag NLSATMP as one of the outputs of this module.

In the decoder the Levinson-Durbin recursion is interrupted after the 10th order prediction coefficients are derived. These values are saved for the adaptive postfilter. Consequently, there are two possible starting conditions. In the ordinary case, the recursion is begun with MINC0 = 1. In the decoder, MINC0 = 10 is another possibility. In the event of this latter case, the values of NRS and ALPHATMP must be saved. Also we note that the value of NLSATMP must be saved until ICOUNT = 3 when the bandwidth expansion module is executed.

Finally, we note that this routine uses three accumulators. The third accumulator, AA2, is used to hold the 17-bit precision value of RC for updating the newest prediction coefficient.

The following pseudo-code describes the fixed point version of the Levinson-Durbin recursion modules.

```
If MINC0 > 1, go to RECURSION
MINC0 = 1                                   | Initializations for
ILLCONDP = .FALSE.                          | decoder only

If ILLCOND = .TRUE., go to FAILED           | Skip if RTMP(LPC + 1) is zero
If RTMP(1) ≤ 0, go to FAILED                | Skip if zero signal
NRS = 0                                     | Q15 format initially

DEN = RTMP(1)                               | Calculate first order predictor
NUM = RTMP(2)
If NUM < 0, set NUM = –NUM
Call SIMPDIV(NUM, DEN, AA0)                 | | RTMP(2) | /RTMP(1)

AA0 = AA0 << 15
RC1 = RND(AA0)
If RTMP(2) > 0, set RC1 = –RC1              | Add sign information
RC = RC1                                    | First order predictor coefficient
ATMP(2) = RC1

AA0 = RTMP(1) << 16                         |
P = RTMP(2) * RC                            |
AA0 = AA0 + (P << 1)                        |
ALPHATMP = RND(AA0)                         | Save DSP accumulator high
                                            | word to memory

RECURSION:

For MINC = MINC0 + 1, MINC0 + 2, ..., LPC, do the following indented lines
    AA0 = 0
    For IP = 2, 3, ..., MINC, do the next 3 lines
        N1 = MINC – IP + 2
        P = RTMP(N1) * ATMP(IP)
        AA0 = AA0 + P                       | 32 bits for SUM
    AA0 = AA0 << 1

    AA0 = AA0 << NRS
    AA1 = RTMP(MINC + 1) << 16
    AA0 = AA0 + AA1                         |
    SIGN = RND(AA0)                         | Save high word sign
    NUM = SIGN
    If NUM < 0, set NUM = –NUM
    If NUM ≥ ALPHATMP, go to FAILED         |
    Call SIMPDIV(NUM, ALPHATMP, AA0)        | Divide to get RC
    AA2 = AA0 << 15                         | AA2 stores 17-bit RC
    RC = RND(AA2)
    If SIGN > 0, set RC = –RC

                                            | Now update ALPHATMP

    AA1 = ALPHATMP << 16
    P = RC * SIGN
    AA1 = AA1 + (P << 1)
    If AA1 ≤ 0, go to FAILED
    ALPHATMP = RND(AA1)

    MH = MINC/2 + 1                         | Fractional part of MINC/2 truncated;
                                            | MH = integer
                                            | Begin to update predictor
                                            | coefficients
```

```
For IP = 2, 3, 4, ..., MH, do the following doubly indented lines
    IB = MINC – IP + 2
    AA0 = ATMP(IP) << 16                          | Load AA0 high word
    P = RC * ATMP(IB)                             | Q15 RC, so << 1
    AA0 = AA0 + (P << 1)
    If AA0 overflowed, then do the following triply indented lines
        NRS = NRS + 1
        For LP = 2, 3, ..., MINC, set ATMP(LP) = ATMP(LP) >> 1
        AA0 = ATMP(IP) << 16                      | First re-scale ATMP
        P = RC * ATMP(IB)                         | Next re-calculate
        AA0 = AA0 + (P << 1)                      | overflowed AA0
    AA1 = ATMP(IB) << 16


        P = RC * ATMP(IP)
        AA1 = AA1 + (P << 1)
        If AA1 overflowed, then do the following triply indented lines


            NRS = NRS + 1
            For LP = 2, 3, ..., MINC, set ATMP(LP) = ATMP(LP) >> 1
            AA0 = ATMP(IP) << 16                  | First re-scale ATMP(IP)
            P = RC * ATMP(IB)                     | Next re-calculate AA0
            AA0 = AA0 + (P << 1)                  |
            AA1 = ATMP(IB) << 16                  | Next re-scale ATMP(IB)
            P = RC * ATMP(IP)                     | Next re-calculate
            AA1 = AA1 + (P << 1)                  | overflowed AA1
        ATMP(IP) = RND(AA0)
        ATMP(IB) = RND(AA1)



                                                  | Update ATMP(MINC + 1)
    AA0 = AA2 >> NRS                              | AA2 contains 17-bit RC


    AA0 = RND(AA0)                                | Output in low word of AA0
    If SIGN > 0, set AA0 = –AA0
    ATMP(MINC + 1) = AA0                          | Low word stored in ATMP

Repeat the above indented lines for the next MINC


NLSATMP = 15 – NRS
If NLSATMP < 13, go to FAILED
Exit this program                                 | Recursion completed normally
                                                  | if execution proceeds to here


FAILED:  Set ILLCOND = .TRUE.
         If MINC ≤ 10, set ILLCONDP = .TRUE.
```

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

The following table lists all variables in the above pseudo-code with their representation format for easy reference.

Variable	Format	Size	Temp/perm	Old/new
AA0, AA1, AA2	DP-integer	1	temp	new
ALPHATMP	SFL	1	temp	old
ATMP	Q13/Q14/Q15	51	perm	old
IB	integer	1	temp	old
ILLCOND	logical	1	perm	new
ILLCONDP	logical	1	perm	new
IP	integer	1	temp	old
LP	integer	1	temp	new
MH	integer	1	temp	old
MINC	integer	1	temp	old
NLSATMP	integer	1	temp	new
NRS	integer	1	temp	new
NUM	integer	1	temp	new
RC	Q15	1	temp	new
RC1	Q15	1	temp	new
RTMP	Q15	51	perm	old
SIGN	integer	1	temp	new

SFL	16-bit scalar floating point
DP-integer	32-bit register such as accumulator or product registers (AA1, AA2 & P)
Integer	16-bit integer
Q13/Q14/Q15	16-bit integer with one of these representations

The above code was written for block 50 and used variable names associated with block 50. However, it can be used for blocks 37 and 44. The following table translates the variable names which are specific for block 50 to those which are specific for one of the other blocks.

Block 50	Block 37	Block 44
ATMP	AWZTMP	GPTMP
ILLCOND	ILLCONDW	ILLCONDG
NLSATMP	NLSAWZTMP	NLSGPTMP
RTMP	R	R

The above code uses a different and simpler division algorithm than that used throughout the rest of the algorithm. It is referred to above as SIMPDIV. The pseudo-code for SIMPDIV is given below. The inputs are NUM and DEN, both 16-bit integers. The output is AA0 with results in lower 17 bits.

```
        Subroutine SIMPDIV(NUM, DEN, AA0)
        AA0 = 0
        AA1 = NUM
        K = 0
LOOP:   AA0 = AA0 << 1
        AA1 = AA1 << 1
        If AA1 ≥ DEN, then set AA1 = AA1 – DEN and AA0 = AA0 + 1
        K = K + 1
        If K < 16, go to LOOP
```

G.3 Pseudo-code for other modules of Recommendation G.728

In this subclause pseudo-code for other modules of Recommendation G.728 is presented. The pseudo-code for the Levinson-Durbin recursion was contained in the previous section, together with the algorithmic changes for the backward vector gain adapter. For each module the floating point pseudo-code is presented first and is then followed by commentary and the fixed point pseudo-code. The following table can be used as a reference for finding the pseudo-code for a particular module of the coder.

The pseudo-codes for all blocks provide a bit-exact specification and are the ultimate definition of the fixed point G.728 coder. Any deviation from these pseudo-codes may result in an incorrect simulation or implementation.

Fixed-point G.728 block number, description and pseudo-code name

Block	Description	Pseudo-Code
1	Input PCM format conversion	Not needed
2	Vector buffer	Not needed
3	Adapter for weighting filter	Use block 45
4	Weighting filter	Block 4
5-7	Switch for ZIR/memory update	Not needed
8	Simulated decoder	See detailed blocks below
9	Synthesis filter for ZIR	Blockzir
10	Weighting filter for ZIR	Blockzir
9, 10	Blocks 9 & 10 for memory update	Block 9
11	VQ target vector computation	Block 11
12	Impulse response vector calc.	Block 12
13	Time-reversed convolution	Block 13
14	Shape codevector convolution	Block 14
15	Codebook energy table calc.	Block 14
16	VQ target vector normalization	Block 16
17	VQ search error calculator	Block 17
18	Best codebook index selector	Block 17
19	Excitation VQ codebook	Block 19
20	Backward vector gain adapter	See detailed blocks below
21	Gain scaling unit	Block 19
22	Synthesis filter	Use block 9
23	Synthesis filter adapter	See blocks 49-51

Fixed-point G.728 block number, description and pseudo-code name *(end)*

Block	Description	Pseudo-Code
24	Codebook search module	See blocks 12-18
28	Output PCM format conversion	Not needed
29	Decoder excitation codebook	Use block 19
30	Decoder backward gain adapter	Same as block 20
31	Decoder gain scaling unit	Use block 19
32	Decoder synthesis filter	Block 32
33	Decoder syn. filter adapter	Same as block 23
34	Postfilter	See blocks 71-77
35	Postfilter adapter	See blocks 81-85
36	Hybrid window for W(z)	Block 36
37	Durbin's recursion for W(z)	See G.2
38	W(z) coefficient calculator	Block 38
43	Hybrid window for GP(z)	Block 43
44	Durbin's recursion for GP(z)	See G.2
45	GP(z) bandwidth expansion	Block 45
46	Log-gain linear predictor	Block 46
48	Inverse logarithm calculator	Block 46
49	Hybrid window for A(z)	Block 49
50	Durbin's recursion for A(z)	See G.2
51	A(z) coefficient calculator	Block 51
71-77	Blocks inside postfilter	Corresponding blocks
81-85	Blocks in postfilter adapter	Corresponding blocks
91	Gain codebook index delay unit	Not needed
92	Shape codebook index delay unit	Not needed
93	Gain codebook log-gain table	Table in G.5
94	Shape codebook log-gain table	Table in G.5
95	1-sample delay for log-gain	Not needed
96	adder to update log-gain	Block 46
97	Log-gain limiter at –32 dB	Block 46
98	Log-gain limiter: –32 to 28 dB	Block 46
99	Adder to restore gain offset	Block 46

G.3.1 Block 4 – Pseudo-code for weighting filter

This is the floating point pseudo-code for block 4, the filtering of the input speech by the perceptual weighting filter.

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

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

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

      SW(K) = SW(K) – WIIR(1) * AWP(2)                  | Handle the last
      WIIR(1) = SW(K)                                   | one differently
```

Repeat the above for the next K

For the fixed point version of this pseudo-code there are the NLS values associated with WFIR and WIIR. The computation must be so that the input speech has the same NLS as WFIR and the result from this calculation has the same NLS as WIIR. In this instance, the NLS values for WIIR and WFIR are fixed at the same value as the input speech. AWZ and AWP are Q14. The value for the input speech, NLSS, is 2. Different input formats (16-bit linear, μ-law, A-law, etc.) are supposed to convert to the range of [–4096, +4095.75] represented in a Q2 format.

For K bit linear input, it is assumed that the data occupies the K least significant bits of a 16-bit word, K_BIT_SAMPLE. The proper representation is given by:

$$NLS = 15 - K$$

$$S = K\_BIT\_SAMPLE << NLS$$

For 16-bit linear input signals (16_BIT_SAMPLE), a right shift of 1 bit is required:

$$S = 16\_BIT\_SAMPLE >> 1$$

For μ-law PCM (MULAW_SAMPLE), the largest magnitude sample value is 4015.5 and it is assumed that this would be represented in Q1 format as 8031. To convert to Q2 format, a left shift of 1 bit is needed:

$$S = MULAW\_SAMPLE << 1$$

For A-law PCM (ALAW_SAMPLE), the largest magnitude sample is 2016, but some sample values have a fractional part of 0.5. Consequently, 2016 would be represented as 4032 in a 16-bit word. To put this value in the proper range, a left shift of 2 bits is needed:

$$S = ALAW\_SAMPLE << 2$$

This is the fixed point pseudo-code for block 4, the filtering of the input speech by the perceptual weighting filter.

```
For   K = 1, 2, ..., IDIM, do the following
      AA0 = S(K)
      AA0 = AA0 << 14
      For  J = LPCW, LPCW – 1, ..., 3, 2, do the next 2 lines
           AA0 = AA0 + WFIR(J) * AWZ(J + 1)             | All-zero part
           WFIR(J) = WFIR(J – 1)                        | of the filter

      AA0 = AA0 + WFIR(1) * AWZ(2)                      | Handle the last
      WFIR(1) = S(K)                                    | one differently
```

```
For  J = LPCW, LPCW – 1, ..., 3, 2, do the next 2 lines
     AA0 = AA0 – WIIR(J) * AWP(J + 1)          | All-pole part
     WIIR(J) = WIIR(J – 1)                     | of the filter

AA0 = AA0 – WIIR(1) * AWP(2)                   | Handle the last
AA0 = AA0 >> 14                                | one differently

If AA0 > 32767, set AA0 = 32767                | Saturation mode for
If AA0 < –32768, set AA0 = –32768              | multiplier input later

WIIR(1) = AA0                                  | 16-bit lower word saved
SW(K) = AA0                                    | SW is Q2
```

Repeat the above for the next K

G.3.2 Blockzir – Pseudo-code for synthesis and perceptual weighting filters during zero-input response computation

This is the floating point pseudo-code for block 9 (the synthesis filter) during zero-input response computation.

```
For  K = 1, 2, ..., IDIM, do the following
     TEMP(K) = 0.
     For  J = LPC, LPC – 1, ..., 3, 2, do the next 2 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)             | Handle last one
     STATELPC(1) = TEMP(K)                              | differently
```

Repeat the above for the next K

This is the floating point pseudo-code for block 10 (the perceptual weighting filter) during zero-input response computation.

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

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

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

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

Repeat the above for the next K

In the fixed point code, we note that STATELPC is segmented block floating point and has associated with it NLSSTATE. Since there is zero-input, we do not need to match NLSSTATE with the NLS of the input. The A( ), AWZ( ), and AWP( ) values are always represented in Q14 format.

This is the fixed point pseudo-code for block 9 (the synthesis filter) during zero-input response computation.

```
NLSSTATE(11) = NLSSTATE(1)
For  K = 2, 3, 4, ..., 10, do the next line                | Find minimum NLSSTATE
     If NLSSTATE(K) < NLSSTATE(11), set NLSSTATE(11) = NLSSTATE(K)

For  K = 1, 2, ..., IDIM, do the following
     I = 1
```

```
    L = 6 – K
    J = LPC
    AA0 = 0
    For  LL = 1, ..., L, do the next 3 lines
        AA0 = AA0 – STATELPC(J) * A(J + 1)          | Multiply – add
        STATELPC(J) = STATELPC(J – 1)               | Memory shift
        J = J – 1
    NLS = NLSSTATE(I) – NLSSTATE(11)
    AA1 = AA0 >> NLS


    For  I = 2, ..., 10, do the next 8 lines
        AA0 = 0
        For   LL = 1, 2, ..., IDIM, do the next 3 lines
            AA0 = 0 – STATELPC(J) * A(J + 1)
            STATELPC(J) = STATELPC(J – 1)           | STATELPC(0) = garbage if J = 1; it is OK
            J = J – 1
        NLS = NLSSTATE(I) – NLSSTATE(11)
        AA0 = AA0 >> NLS                            | Shift to align
        AA1 = AA1 + AA0


    If K = 1, go to SHIFT2
    L = K – 1
    AA0 = 0
    For  LL = 1, 2, ..., L, do the next 3 lines
        AA0 = AA0 – STATELPC(J) * A(J + 1)
        STATELPC(J) = STATELPC(J – 1)               | STATELPC(0) = garbage if  J = 1; it is OK
        J = J – 1
    AA1 = AA1 + AA0                                 | No shift necessary for this time


SHIFT2: AA1 = AA1 >> 14                             | A( ) was Q14, NLS of AA1
                                                    | is now NLSSTATE(11)
    If AA1 > 32767, set AA1 = 32767                 | Clip to 16 bits if necessary since
    If AA1 < –32768, set AA1 = –32768               | STATELPC(1) will be multiplier input


    STATELPC(1) = AA1                               | Save lower 16-bit word for
                                                    | STATELPC
    IR = NLSSTATE(11) – 2                           | Make TEMP Q2 format
    If IR > 0, set AA1 = AA1 >> IR                  |
    If IR < 0, set AA1 = AA1 << –IR                 |
    TEMP(K) = AA1

Repeat the above for the next K


Call VSCALE(STATELPC, IDIM, IDIM, 13, STATELPC, NLS)
NLSSTATE(11) = NLSSTATE(11) + NLS                   | Re-normalize new STATELPC to 15 bits


For     L = 1, 2, ..., 10, do the next line         | Update NLSSTATE
        NLSSTATE(L) = NLSSTATE(L + 1)
```

In the fixed point pseudo-code for block 10, TEMP, ZIRWFIR and ZIRWIIR are Q2. In the previous block TEMP was explicitly created with this value. Thus, we do not need to normalize to add them together. This is the fixed point pseudo-code for block 10 (the perceptual weighting filter) during zero-input response computation.

```
For K = 1, 2, ..., IDIM, do the following
    AA0 = TEMP(K) << 14                             | Because AWZ is Q14
    For  J = LPCW, LPCW – 1, ..., 3, 2, do the next 2 lines
        AA0 = AA0 + ZIRWFIR(J) * AWZ(J + 1)         | All-zero part
        ZIRWFIR(J) = ZIRWFIR(J – 1)                 | of the filter


    AA0 = AA0 + ZIRWFIR(1) * AWZ(2)                 | Handle last one
    ZIRWFIR(1) = TEMP(K)                            | differently
```

```
For  J = LPCW, LPCW – 1, ..., 3, 2, do the next 2 lines
     AA0 = AA0 – ZIRWIIR(J) * AWP(J + 1)        | All-pole part
     ZIRWIIR(J) = ZIRWIIR(J – 1)                | of the filter

AA0 = AA0 – ZIRWIIR(1) * AWP(2)                 | Handle last one
AA0 = AA0 >> 14                                 | differently

If AA0 > 32767, set AA0 = 32767                 | Clip since ZIR & ZIRWIIR
If AA0 < –32768, set AA0 = –32768               | will be multiplier input

ZIR(K) = AA0                                    | Save lower 16-bit word
ZIRWIIR(1) = AA0                                | for ZIR and ZIRWIIR
```

Repeat the above for the next K

G.3.3 Blocks 9 and 10 – Pseudo-code for synthesis and perceptual weighting filter memory updates

This is the floating point pseudo-code for blocks 9 and 10, the filter memory update.

```
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 5 lines
          ZIRWFIR(I) = ZIRWFIR(I – 1)
          TEMP(I) = TEMP(I – 1)
          A0 = A0 – A(I) * ZIRWFIR(I)           | Compute zero-state responses
          A1 = A1 + AWZ(I) * ZIRWFIR(I)         | at various stages of
          A2 = A2 – AWP(I) * TEMP(I)            | 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 4 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
     ZIRWFIR(I) = STATELPC(I)                   | the right value

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

The following is the fixed point pseudo-code for the same blocks. STATELPC has 10 exponents stored in NLSSTATE(1), ..., NLSSTATE(10). Associated with the array ET is NLSET. ZIRWIIR and ZIRWFIR are Q2 after the update. ZIRWFIR is initially used as a scratch array. Upon entry into this code, both ET and the top 5 elements of STATELPC [STATELPC(1) through STATELPC(5)] are 15-bit block floating point arrays. When ET is filtered by the LPC synthesis filter without memory, the output (i.e. zero-state response of the LPC filter) may exceed the 15-bit range. When this happens, we right shift ET by 1 bit and repeat the calculation until the output fits into 15 bits. Empirically the

process repeats at most 3 times (or 4 times if the first time through is counted). Note that there are only 10 multiply-adds for each repetition of the calculation, because the calculation of the zero-state response of the weighting filter has been moved to a separate loop. The zero-state response of the LPC filter calculated this way is always representable in 15 bits or less. When this response is then added to the 15-bit STATELPC to update STATELPC, the result of the addition is guaranteed to be representable by 16 bits. Before exiting this code, STATELPC is scaled to 14 bits to avoid overflows in the zero-input response calculation later.

```
                                                    | First calculate zero-state response
LABEL1: ZIRWFIR(1) = ET(1)                          | of the LPC synthesis filter
For   K = 2, 3, ..., IDIM, do the following indented lines

      AA0 = ET(K) << 14                             | Because A(1) = 1 in Q14 = 16384
      For  I = K, K – 1, ..., 2, do the next 3 lines
           ZIRWFIR(I) = ZIRWFIR(I – 1)
           P = A(I) * ZIRWFIR(I)                    | Q14 multiplication
           AA0 = AA0 – P                            | Compute zero-state responses


      AA1 = AA0 << 3
      If AA1 overflowed above, do the next 4 lines  | Make sure after AA0 >> 14 later,
           For  I = 1, 2, ..., IDIM, do the next line | the result does not exceed 15 bits.
                ET(I) = ET(I) >> 1                  | If it does, then ET >> 1
                NLSET = NLSET – 1                   | and repeat the
                GO TO LABEL1                        | calculation until it fits


      AA0 = AA0 >> 14                               | Compensate for A( ) being Q14
      ZIRWFIR(1) = AA0                              | Keep lowest 16 bits

Repeat the above indented section for the next K


N = IDIM + 1                                        | Now calculate the zero-state response
TEMP(1) = ZIRWFIR(IDIM)                             | of the weighting filter
For   K = 2, 3, ..., IDIM, do the following indented lines
      AA1 = ZIRWFIR(N – K) << 14                    | Because AWZ(1) = 1 (in Q14 = 16384)
      M = IDIM – K
      For  I = K, K – 1, ..., 2, do the next 5 lines
           TEMP(I) = TEMP(I – 1)                    | Shift all-pole part of filter memory
           P = AWZ(I) * ZIRWFIR(I + M)              | all-zero part of the weighting filter
           AA1 = AA1 + P
           P = AWP(I) * TEMP(I)                     | All-pole part of the weighting filter


           AA1 = AA1 – P


      AA1 = AA1 >> 14
      If AA1 > 32767, set AA1 = 32767               | Clip if necessary, since TEMP(1)
      If AA1 < –32768, set AA1 = –32768             | will be 16-bit input to multiplier
      TEMP(1) = AA1                                 | Keep lowest 16 bits

Repeat the above indented section for the next K


IR = NLSET – 2
For   K = 1, ..., IDIM, do the next 2 lines         | Now shift TEMP to Q2 like
      If IR > 0, set TEMP(K) = TEMP(K) >> IR        | ZIRWIIR
      If IR < 0, set TEMP(K) = TEMP(K) << –IR

                                                    | Now update filter memory by adding
                                                    | zero-state responses to zero-input
                                                    | responses. First we must match the
                                                    | NLS of ZIRWFIR and STATELPC

If NLSET = NLSSTATE(10), go to LABEL2               | No changes necessary
```

```
If NLSET < NLSSTATE(10), do the next 5 lines          | Lose precision in STATELPC
      NLSD = NLSSTATE(10) – NLSET                     | by NLSD bits
      For  K = 1, 2, ..., IDIM, do the next line
           STATELPC(K) = STATELPC(K) >> NLSD
      NLSSTATE(10) = NLSET
      go to LABEL2                                    | Only case left is:
                                                      | NLSET > NLSSTATE
NLSD = NLSET – NLSSTATE(10)                           | Lose precision in ZIRWFIR
For   K = 1, 2, ..., IDIM, do the next line           | by NLSD bits
      ZIRWFIR(K) = ZIRWFIR(K) >> NLSD

LABEL2:                                               | Now we are ready
AA1 = 4095                                            | 4095 = STATELPC clipping level
If NLSSTATE(10) ≥ 0, set AA1 = AA1 << NLSSTATE(10)    | Shift clipping level to
If NLSSTATE(10) < 0, set AA1 = AA1 >> –NLSSTATE(10)   | align with STATELPC

For   K = 1, 2, ..., IDIM, do the following indented lines
      AA0 = STATELPC(K) + ZIRWFIR(K)                  | Update LPC filter memory.
      If AA0 > AA1, set AA0 = AA1                     | If necessary, perform the clipping as specified
      If AA0 < –AA1, set AA0 = –AA1                   | in floating point in Recommendation G.728.
                                                      | Note that these values were scaled.
                                                      | So, if 32767 < | AA0 | < AA1, we need to clip
      If AA0 > 32767, set AA0 = 32767                 | AA0 to 16 bits since STATELPC(K)
      If AA0 < –32768, set AA0 = –32768               | will later be a 16-bit
      STATELPC(K) = AA0                               | input to the multiplier

      AA0 = ZIRWIIR(K) + TEMP(K)                      | Update all-pole part of W(z) memory
      If AA0 > 32767, set AA0 = 32767                 | Again, clip to 16 bits if necessary
      If AA0 < –32768, set AA0 = –32768               | since ZIRWIIR(K) will later be
      ZIRWIIR(K) = AA0                                | a 16-bit input to the multiplier

Repeat the above indented section for the next K

Call VSCALE(STATELPC, IDIM, IDIM, 12, STATELPC, NLS)  | Scale STATELPC to 14 bits
NLSSTATE(10) = NLSSTATE(10) + NLS                     | to avoid overflow in
                                                      | zero-input response calculation later

IR = NLSSTATE(10) – 2

For   I = 1, 2, ..., 5, do the next 4 lines           | Now set ZIRWFIR, the all zero
      AA0 = STATELPC(I)                               | part of W(z) memory, to the
      If IR > 0, set AA0 = AA0 >> IR                  | right values in Q2 format
      If IR < 0, set AA0 = AA0 << –IR                 |
      ZIRWFIR(I) = AA0                                |
IR = NLSSTATE(9) – 2                                  |
For   I = 6, 7, ..., 10, do the next 4 lines          |
      AA0 = STATELPC(I)                               |
      If IR > 0, set AA0 = AA0 >> IR                  |
      If IR < 0, set AA0 = AA0 << –IR                 |
      ZIRWFIR(I) = AA0                                |

I = IDIM + 1
For   K = 1, 2, ..., IDIM, do the next line           | Obtain quantized speech by
      ST(K) = STATELPC(I – K)                         | reversing the order of the top 5
NLSST = NLSSTATE(10)                                  | synthesis filter memory locations
                                                      | NLSST is only used in decoder
```

G.3.4 Block 11 – VQ target vector computation

This is the floating point pseudo-code for block 11, the VQ target vector computation.

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

For the fixed point code, SW and ZIR are both in Q2 format, the same as the input speech. Thus, NLSTARGET = 2. Here is the fixed point pseudo-code.

```
set NLSTARGET = 2

For   K = 1, 2, ..., IDIM, do the next 6 lines
      AA0 = SW(K)
      AA1 = ZIR(K)
      AA0 = AA0 – AA1
      If AA0 > 32767, set AA0 = 32767                  | Clip if necessary
      If AA0 < –32768, set AA0 = –32768
      TARGET(K) = AA0
```

G.3.5 Block 12 – Impulse response vector calculation

The following is the floating pseudo-code for block 12.

```
TEMP(1) = 1                                            | TEMP = synthesis filter memory
WS(1) = 1                                              | WS = 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 5 lines
           TEMP(I) = TEMP(I – 1)
           WS(I) = WS(I – 1)                           |
           A0 = A0 – A(I) * TEMP(I)                    | Filtering
           A1 = A1 + AWZ(I) * TEMP(I)                  |
           A2 = A2 – AWP(I) * WS(I)

      TEMP(1) = A0
      WS(1) = A0 + A1 + A2
```

Repeat the above indented section for the next K

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

The values for the predictor coefficients, A( ), AWZ( ) and AWP( ) are all stored in Q14 format. In the fixed point pseudo-code to follow, only two 32-bit accumulators are indicated, AA0 and AA1. Accumulators A1 and A2 in the floating point pseudo-code have been combined. Guard bits are not required. The output array, H( ) is stored in Q13 format. The following is the fixed point pseudo-code.

```
TEMP(1) = 8192                                         | TEMP = synthesis filter memory
WS(1) = 8192                                           | WS = W(z) all-pole part memory
                                                       | WS & TEMP are Q13 16-bit words

For   K = 2, 3, ..., IDIM, do the following
      AA0 = 0
      AA1 = 0
      For  I = K, K – 1, ..., 3, 2, do the next 5 lines
           TEMP(I) = TEMP(I – 1)
           WS(I) = WS(I – 1)                           |
           AA0 = AA0 – A(I) * TEMP(I)                  | Filtering
           AA1 = AA1 + AWZ(I) * TEMP(I)                |
           AA1 = AA1 – AWP(I) * WS(I)
```

```
    AA1 = AA0 + AA1
    AA0 = AA0 >> 14                          | >> 14 because A( ), AWZ( ) and
    AA1 = AA1 >> 14                          | AWP( ) were inQ14 format
    TEMP(1) = AA0
    WS(1) = AA1
```

Repeat the above indented section for the next K

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

G.3.6 Block 13 – Time-reversed convolution

This module performs time-reversed convolution in preparation for the codebook search. The original floating point pseudo-code was

```
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

In the fixed point version, H( ) is represented in Q13 format and TARGET is represented in block floating point. NLSTARGET is determined in block 16. NLSPN is fixed at 7.

```
For  K = 1, 2, ..., IDIM, do the following
     K1 = K – 1
     AA0 = 0                                 | Accumulator zeroed
     For  J = K, K + 1, ..., IDIM, do the next 2 lines
          P = TARGET(J) * H(J – K1)
          AA0 = AA0 + P
     AA0 = AA0 >> 13 + (NLSTARGET – 7)       | Right shift to make Q7
     If AA0 > 32767, set AA0 = 32767         | Clip AA0 to 16 bits since
     If AA0 < –32768, set AA0 = –32768       | PN will be multiplier input
     PN(K) = AA0                             | AA0 in saturation mode
```

Repeat the above for the next K

G.3.7 Block 14 – Shape codevector convolution and energy calculation

This is the pseudo-code for the codevector energy calculation, blocks 14 and 15.

```
For  J = 1, 2, ..., NCWD, do the following   | One codevector per loop
     J1 = (J – 1) * IDIM
     For  K = 1, 2, ..., IDIM, do the next 4 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

In the fixed point pseudo-code, H( ) is represented in Q13 format and Y( ) in Q11 format. It was found empirically that after convolution of H( ) and Y( ), overflows in the accumulator, i.e. a result with magnitude larger than 2 ** 31, did not occur. Thus, the following pseudo-code does not test for overflow in the accumulator. This makes the corresponding DSP code run faster. The subsequent shift by 14 bits is necessary to allow representation of TEMP( ) in Q10 format. The representations used were found to work for a variety of files. NLSY2 = (NLSH + NLSY – 14) * 2 – 15. Since NLSY = 11 and NLSH = 13 then NLSY2 = 5.

```
For   J = 1, 2, ..., NCWD, do the following                | One codevector per loop
      J1 = (J – 1) * IDIM
      For  K = 1, 2, ..., IDIM, do the next 7 lines
           K1 = J1 + K + 1
           AA0 = 0
           For I = 1, 2, ..., K, do the next 2 lines       |
                P = H(I) * Y(K1 – I)                       | Convolution
                AA0 = AA0 + P                              |


                AA0 = AA0 >> 14
                TEMP(K) = AA0                              | Lowest 16 bits only

      Repeat the above 7 lines for the next K


      AA0 = 0
      For  K = 1, 2, ..., IDIM, do the next 2 lines
           P = TEMP(K) * TEMP(K)                           | Compute energy
           AA0 = AA0 + P
      AA0 = AA0 >> 15
      Y2(J) = AA0                                          | Lowest 16 bits only

Repeat the above for the next J
```

G.3.8 Block 16 – VQ target vector normalization

The floating point pseudo-code for this module is given first.

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

For the fixed point pseudo-code, we need to consider the NLS of the gain and the NLS of TARGET. At entry NLSTARGET = 2. In the process, we will create the NLS for TMP.

```
                                                           | Numerator for division = 16384
                                                           | NLS = 14 for numerator
                                                           | NLS for denominator is NLSGAIN
                                                           | NLSGAIN determined in block 46

Call DIVIDE(16384, 14, GAIN, NLSGAIN, TMP, NLSTMP)

For   K = 1, 2, ..., IDIM, do the next 2 lines
      AA0 = TMP * TARGET(K)                                | AA0 is 32 bits
      TARGET(K) = AA0 >> 15                                | Keep only the lower 16 bits
                                                           | TARGET is Q2 at this point

NLSTARGET = 2 + NLSTMP – 15                                | Make TARGET block
                                                           | floating point
Call VSCALE(TARGET, IDIM, IDIM, 14, TARGET, NLS)
NLSTARGET = NLSTARGET + NLS                                | NLS was change in VSCALE
```

G.3.9 Block 17 – VQ search error calculator and best codebook index selector

The following is the floating point pseudo-code for the error calculator and best codebook index selector (blocks 17 and 18).

```
Initialize DISTM to the 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 5 lines
           IDXG = N1
           For   K = 1, 2, ..., N1 – 1, do the next "if" statement
                 If    COR < GB(K) * Y2(J), do the next 2 lines
                       IDXG = K                             | Best positive gain found
                       GO TO LABEL

      If COR ≤ 0, then do the next 5 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 2 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 3 lines
           DISTM = D                                        | Save the lowest distortion
           IG = IDXG                                        | and the best codebook
           IS = J                                           | indices so far

Repeat the above inserted section for the next J

ICHAN = (IS – 1) * NG + (IG – 1)
```

The following is the fixed point pseudo-code for the error calculator and best codebook index selector (blocks 17 and 18). The code has been written such that the absolute value of the correlation with the target value, AA0 below, is used for the gain search and then later the sign of the correlation is re-determined. While this may seem like more work, it avoids having a branch in the middle of the search loop. For most DSP implementations avoiding a branch saves instructions. Alternatively, the sign can be saved and the recomputation avoided. However, this usually costs an extra instruction in the search loop as well.

```
DISTM = 2147483647
For   J = 1, 2, ..., NCWD, do the following
      J1 = (J – 1) * IDIM
      AA0 = 0
      For  K = 1, 2, ..., IDIM, do the next 2 lines
           P = PN(K) * Y(J1 + K)                            | Compute inner product Pj
           AA0 = AA0 + P                                    | NLS for AA0 is 7 + 11 = 18
      If AA0 < 0, set AA0 = – AA0                           | Take absolute value

      IDXG = 1
      P = GB(1) * Y2(J)                                     | NLS for P is 13 + 5 = 18
      If AA0 ≥ P, set IDXG = IDXG + 1
      P = GB(2) * Y2(J)
      If AA0 ≥ P, set IDXG = IDXG + 1
      P = GB(3) * Y2(J)
      If AA0 ≥ P, set IDXG = IDXG + 1
```

```
    AA0 = AA0 >> 14                                  | NLS for AA0 = 4
    If AA0 > 32767, set AA0 = 32767                  | Clip AA0; AA0 in saturation mode
    AA1 = GSQ(IDXG) * Y2(J)                          | NLSGSQ = 11, NLSY2 = 5, so NLSAA1 = 16
    P = G2(IDXG) * AA0                               | NLSG2 = 12, NLSAA0 = 4, so NLSP = 16

    AA1 = AA1 – P
    If AA1 < DISTM, do the next 3 lines
        DISTM = AA1                                  | Double precision DISTM
        IG = IDXG
        IS = J
```

Repeat the above inserted section for the next J

```
AA0 = 0                                              | Now find the sign bit
J1 = (IS – 1) * IDIM
For   K = 1, 2, ..., IDIM, do the next 2 lines
      P = PN(K) * Y(J1 + K)                          | Compute inner product
      AA0 = AA0 + P
If AA0 ≤ 0, set IG = IG + 4

ICHAN = (IS – 1) * NG + (IG – 1)
```

In the above code, we used the following four lines

```
    AA0 = AA0 >> 14                                  | NLS for AA0 = 4
    If AA0 > 32 767, set AA0 = 32 767                | Clip AA0
    AA1 = GSQ(IDXG) * Y2(J)                          | NLSGSQ = 11, NLSY2 = 5, so NLSAA1 = 16
    P = G2(IDXG) * AA0                               | NLSG2 = 12, NLSAA0 = 4, so NLSP = 16
```

In DSP chips which have a "clipping" function, these lines can be replaced by the following code to give the exact same results.

```
    AA0 = AA0 << 2                                   | NLS for AA0 = 20
    AA0 = CLIP(AA0)                                  | AA0 is in saturation mode
    AA0 = AA0 >> 16                                  | Take high word; NLS for AA0 = 4
    AA1 = GSQ(IDXG) * Y2(J)                          | NLSGSQ = 11, NLSY2 = 5, so NLSAA1 = 16
    P = G2(IDXG) * AA0                               | NLSG2 = 12, NLSAA0 = 4, so NLSP = 16
```

The CLIP function and saturation mode refer to the concept of not allowing AA0 to overflow when the << 2 operation is performed. Instead of overflow, AA0 is set to the maximum positive or negative number, depending on its original sign. In this case, AA0 is always positive. This alternative is DSP dependent and may require more than a 32 bit accumulator. The alternative in the main pseudo-code can always be implemented.

G.3.10 Block 19 – Excitation VQ codebook and block 21 – Gain scaling unit

This is the floating point version of the pseudo-code for block 19, the excitation VQ codebook.

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

The floating point version of the pseudo-code for block 21, the gain scaling unit is given below.

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

For the fixed point pseudo-code, we combine both blocks 19 and 21 into a single module. Both Y and GQ have fixed Q formats, Q11 and Q13, respectively. The value of GAIN has associated with it NLSGAIN. To get the maximum accuracy, the product GQ(IG) * GAIN is normalized to 32 bits before rounding to the upper 16 bits is performed. Let NNGQ(I) be [1 + the number of left shifts needed to normalize the Q13 GQ(I)]. So, NNGQ(I) = 3 for I = 1, 2, 5, 6, NNGQ(I) = 2 for I = 3, 7, and NNGQ(I) = 1 for I = 4, 8. Then the pseudo-code can be written as follows.

```
AA0 = GQ(IG) * GAIN                                  | AA0 has NNGQ(IG) leading zeros
AA0 = AA0 << NNGQ(IG)                                | Left shift NNGQ(IG) bits to
                                                     | normalize AA0
TMP = RND(AA0)                                       | Round to upper 16 bits and assign to TMP

NLSAA0 = 13 + NLSGAIN                                | Q format of the product GQ(IG) * GAIN
NLSTMP = NLSAA0 + NNGQ(IG) – 16                      | Q format of TMP, because
                                                     | AA0 left shift by NNGQ(IG) bits
                                                     | then round and take upper 16 bits

NN = (IS – 1) * IDIM                                 | Normalize selected shape
Call VSCALE(Y(NN + 1), IDIM, IDIM, 14, TEMP, NLS)    | codevector to 16 bits; put in TEMP

For  K = 1, 2, ..., IDIM, do the next 2 lines        | TMP and TEMP both normalized to 16 bits,
     AA0 = TMP * TEMP(K)                             | so the product has 1 leading zero.
     ET(K) = RND(AA0)                                | Directly rounding to high work
                                                     | gives us a 15-bit ET array

NLSET = NLSTMP + 11 + NLS – 16                       | Calculate the NLS for ET
```

G.3.11 Block 32 – Decoder synthesis filter

This is the floating point pseudo-code for block 32, the decoder synthesis filter.

```
For  K = 1, 2, ..., IDIM, do the next 6 lines
     TEMP(K) = 0
     For  J = LPC, LPC – 1, ..., 3, 2, do the next 2 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)          | Handle last one differently
     STATELPC(1) = TEMP(K)                           |

Repeat the above for the next K

TEMP(1) = ET(1)
For  K = 2, 3, ..., IDIM, do the next 5 lines
     A0 = ET(K)
     For  I = K, K – 1, ..., 2, do the next 2 lines
          TEMP(I) = TEMP(I – 1)
          A0 = A0 – A(I) * TEMP(I)                   | Compute zero-state response

     TEMP(1) = A0

Repeat the above 5 lines 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 3 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 = IDIM + 1
For  K = 1, 2, ..., IDIM, do the next line           | Obtain quantized speech by
     ST(K) = STATELPC(I – K)                         | reversing order of synthesis
                                                     | filter memory
```

The fixed point pseudo-code for block 32 follows the same methodology used in block 9 except that there is no memory update for the perceptual weighting filter.

```
NLSSTATE(11) = NLSSTATE(1)
For   K = 2, 3, 4, ..., 10, do the next line                    | Find minimum NLSSTATE
      If NLSSTATE(K) < NLSSTATE(11), set NLSSTATE(11) = NLSSTATE(K)

For   K = 1, 2, ..., IDIM, do the following
      I = 1
      L = 6 – K
      J = LPC
      AA0 = 0
      For  LL = 1, ..., L, do the next 3 lines
           AA0 = AA0 – STATELPC(J) * A(J + 1)                   | Multiply – add
           STATELPC(J) = STATELPC(J – 1)                        | Memory shift
           J = J – 1
      NLS = NLSSTATE(I) – NLSSTATE(11)
      AA1 = AA0 >> NLS

      For  I = 2, ..., 10, do the next 8 lines
           AA0 = 0
           For LL = 1, 2, ..., IDIM, do the next 3 lines
                AA0 = AA0 – STATELPC(J) * A(J + 1)
                STATELPC(J) = STATELPC(J – 1)                   | STATELPC(0) = garbage if J = 1; it is OK
                J = J – 1
           NLS = NLSSTATE(I) – NLSSTATE(11)
           AA0 = AA0 >> NLS                                     | Shift to align
           AA1 = AA1 + AA0

      If K = 1, go to SHIFT2
      L = K – 1
      AA0 = 0
      For  LL = 1, 2, ..., L, do the next 3 lines
           AA0 = AA0 – STATELPC(J) * A(J + 1)
           STATELPC(J) = STATELPC(J – 1)                        | STATELPC(0) = garbage if J = 1; it is OK
           J = J – 1
      AA1 = AA1 + AA0                                           | No shift necessary for this time

SHIFT2: AA1 = AA1 >> 14                                         | A( ) was Q14, NLS of AA1
                                                                | is now NLSSTATE(11)
      If AA1 > 32767, set AA1 = 32767                           | Clip to 16 bits if necessary since
      If AA1 < –32768, set AA1 = –32768                         | STATELPC(1) will be multiplier input

      STATELPC(1) = AA1                                         | Save lower 16-bit word
      IR = NLSSTATE(11) – 2                                     | for STATELPC
      If IR > 0, set AA1 = AA1 >> IR                            | Make TEMP Q2 format
      If IR < 0, set AA1 = AA1 << – IR                          |
      TEMP(K) = AA1

Repeat the above for the next K

Call VSCALE(STATELPC, IDIM, IDIM, 13, STATELPC, NLS)
NLSSTATE(11) = NLSSTATE(11) + NLS                               | Re-normalize new STATELPC to 15 bits

For   L = 1, 2, ..., 10, do the next line                       | Update NLSSTATE
      NLSSTATE(L) = NLSSTATE(L + 1)


                                                                | Frst calculate zero-state response
LABEL1: TEMP(1) = ET(1)                                         | of the LPC synthesis filter
For   K = 2, 3, ..., IDIM, do the following indented lines
      AA0 = ET(K) << 14                                         | Because A(1) = 1 in Q14 = 16384
      For  I = K, K – 1, ..., 2, do the next 3 lines
           TEMP(I) = TEMP(I – 1)
           P = A(I) * TEMP(I)                                   | Q14 multiplication
           AA0 = AA0 – P                                        | Compute zero-state responses
```

```
    AA1 = AA0 << 3
    If AA1 overflowed above, do the next 4 lines          | Make sure after AA0 >> 14 later,
        For  I = 1, 2, ..., IDIM, do the next line        |  the result does not exceed 15 bits.
            ET(I) = ET(I) >> 1                            | If it does, then ET >> 1
        NLSET = NLSET – 1                                 | and repeat
        GO TO LABEL1                                      | the calculation until it fits

    AA0 = AA0 >> 14                                       | Compensate for A( ) being Q14
    TEMP(1) = AA0                                         | Keep lowest 16 bits

Repeat the above indented section for the next K

If NLSET = NLSSTATE(10), go to LABEL2                     | No changes necessary

If NLSET < NLSSTATE(10), do the next 5 lines              | Lose precision in STATELPC
    NLSD = NLSSTATE(10) – NLSET                           | by NLSD bits
    For  K = 1, 2, ..., IDIM, do the next line
        STATELPC(K) = STATELPC(K) >> NLSD
    NLSSTATE(10) = NLSET
    go to LABEL2                                          | Only case left is NLSET > NLSSTATE
                                                          | Lose precision in TEMP
NLSD = NLSET – NLSSTATE(10)                               | by NLSD bits
For  K = 1, 2, ..., IDIM, do the next line                |
    TEMP(K) = TEMP(K) >> NLSD

LABEL2:                                                   | Now we are ready
AA1 = 4095                                                | 4095 = STATELPC clipping level
If NLSSTATE(10) ≥ 0, set AA1 = AA1 << NLSSTATE(10)        | Shift clipping level to
If NLSSTATE(10) < 0, set AA1 = AA1 >> –NLSSTATE(10)       | align with STATELPC

For  K = 1, 2, ..., IDIM, do the following indented lines
    AA0 = STATELPC(K) + TEMP(K)                           | Update LPC filter memory
    If AA0 > AA1, set AA0 = AA1                           | If necessary, perform the clipping as specified in
    If AA0 < –AA1, set AA0 = –AA1                         | floating point in Recommendation G.728
                                                          | Note that these values were scaled
                                                          | So, if 32767 > | AA0 | < AA1, we need
    If AA0 > 32767, set AA0 = 32767                       | to clip AA0 to 16 bits since STATELPC(K)
    If AA0 < –32768, set AA0 = –32768                     | will later be a 16-bit input to
    STATELPC(K) = AA0                                     | the multiplier

Repeat the above indented section for the next K

Call VSCALE(STATELPC, IDIM, IDIM, 12, STATELPC, NLS)      | Scale STATELPC to 14 bits
NLSSTATE(10) = NLSSTATE(10) + NLS                         | to avoid overflow in
                                                          | zero-input response calculation later

I = IDIM + 1
For  K = 1, 2, ..., IDIM, do the next line                | Obtain quantized speech by
    ST(K) = STATELPC(I – K)                               | reversing the order of the top 5
NLSST = NLSSTATE(10)                                      | synthesis filter memory locations
                                                          | NLSST is used later in decoder
```

G.3.12 Block 36 – Pseudo-code for hybrid windowing module

In this subclause both the floating point and fixed point pseudo-code for block 36 are given. First, the floating point pseudo-code is presented.

```
    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
        SBW(N2 + N) = STMP(N)                             | Shift in the new signal
                                                          | SBW(N3) is the newest sample
```

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

For  I = 1, 2, ..., LPCW + 1, do the next 4 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 3 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
```

Now we give the fixed point version of the same module. In this code we have added several new variables. NLSREXPW is a global variable holding the number of left shifts for normalizing REXPW. This variable is initialized with a value of 31.

```
N1 = LPCW + NFRSZ (= 10 + 20)                    | Compute some constants (can be
N2 = LPCW + NONRW (= 10 + 30)                    | precomputed and stored in memory)
N3 = LPCW + NFRSZ + NONRW (= 10 + 20 + 30)

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
     SBW(N2 + N) = STMP(N)                       | SBW(N3) is the newest sample
                                                 | All SBW are Q2 and represented
                                                 | in 15 bits precision

Call FINDNLS(SBW, N3, N3, 14, NLS)               | Find the amount of left shifts
                                                 | needed in the next loop to get
                                                 | 2 bits of headroom. We do not
                                                 | really need to do the scaling
                                                 | We just use NLS

NLSTMP = NLS – 1
K = 1
For  N = 60, 59, ..., 1, do the next 4 lines
     P = SBW(N) * WNRW(K)                        | WNRW is Q15, left shift by
     AA0 = P << NLSWS                            | NLSWS bits will make
     WS(N) = RND(AA0)                            | the largest WS(N) element
     K = K + 1                                   | a 14-bit number (2 bits of headroom for
                                                 | later acumulation)

NLSATTW = 15
Call
HWMCORE(LPCW, N1, N3, NLSATTW, WS, NLSTMP, REXPW, NLSREXPW, R, ILLCONDW)

If NLSREXPW > 41, set NLSREXPW = 41              | To avoid reduced accuracy in
                                                 | REXPW( ) and R( ) during long periods
                                                 | of zero input signal
```

The subroutine HWMCORE can be found in G.3.18.

In the above code a call to FINDNLS searches the entire SBW buffer of 60 samples. However, a bit-exact substitute which uses 2 more words of memory can be used to reduce that computation. SBW will always contain 40 old samples and 20 new ones. We can divide this into three vectors of 20 samples each. We keep track of the NLS for each of the three vectors and then choose the minimum value one for use in applying the hybrid window. Since two of the vectors are composed of old samples, we will already know their respective NLS. We need only check the newest vector to find its NLS. We then need to store the NLS for the newest vector and the newer of the two old vectors for the next computation. This method will result in the selection of exactly the same NLS as the procedure shown in the above pseudo-code.

The following table lists all variables in this pseudo-code with their representation format and size for easy reference. The table notes whether each variable is temporary (temp), meaning that it need not be stored after the module is completed, or permanent (perm), meaning that the value will be needed after the current calculation as well. The table also notes which variables were not included in the previous floating point pseudo-code (old/new).

Variable	Format	Size	Temp/perm	Old/new
NLS	integer	1	temp	new
NLSREXPW	integer	1	perm	new
NLSTMP	integer	1	temp	new
REXPW	BFL	11	perm	old
R	BFL	1	perm	old
SBW	Q2	60	perm	old
STMP	Q2	20	perm	old
WS	BFL	60	temp	old
BFL Block floating point Integer 16-bit integer				

G.3.13 Block 38 – Weighting filter coefficient calculator

We begin with the floating point pseudo-code for this block.

```
If ICOUNT ≠ 3, skip the execution of this block
Otherwise, do the following

For  I = 2, 3, ..., 11, do the next line
     AWP(I) = WPCFV(I) * AWZTMP(I)          | Scale denominator coefficients

For  I = 2, 3, ..., 11, do the next line
     AWZ(I) = WZCFV(I) * AWZTMP(I)          | Scale numerator coefficients
```

In the fixed point pseudo-code, we must consider the possibility that there was ill-conditioning in Durbin's recursion or that AWZTMP could not even be expressed in Q13. (It has never been observed that Q13 was not sufficient, but this possibility must still be considered.) The variable ILLCONDW is a flag from block 37 which indicates whether the results of block 37 are valid or not. In Recommendation G.728, there is an implicit assumption that the results of Durbin will not be used if ILLCONDW is true. That is, AWZ and AWP will not be updated from AWZTMP. The same assumption is repeated here. If ILLCONDW is true, then we do not update AWP or AWZ. It is unnecessary to do so because we will continue to use the previous values.

Next, we must consider the possibility that the coefficients AWZTMP( ) from Durbin's recursion may be in Q13, Q14 or Q15. NLSAWZTMP is the number of left shifts of AWZTMP. We want the numerator and denominator coefficients, AWZ and AWP to be in Q14 for the output. It may be the case that AWZ cannot be represented in Q14. When this is the case, do not update AWZ and AWP. The fixed point pseudo-code is given by the following.

```
If ICOUNT ≠ 3, skip the execution of this block
Otherwise, do the following

                                        | First check to see if ILLCONDW is true
```

```
If ILLCONDW = .TRUE., skip the execution of this block
Otherwise, do the following

                                          | Next do the numerator coefficients
                                          | If they overflow for Q14,
                                          | do not update AWZ or AWP
                                          | Temporary array WS is used in case
                                          | of overflow, so that AWZ is preserved

    For  I = 2, 3, ..., 7, do the next 6 lines
         AA0 = WZCFV(I) * AWZTMP(I)       | WZCFV is Q14,
         If NLSAWZTMP = 13, AA0 = AA0 << 3| AA0 is 14 + NLSAWZTMP
         If NLSAWZTMP = 14, AA0 = AA0 << 2| Make AA0 Q30 for all 3 cases by
         If NLSAWZTMP = 15, AA0 = AA0 << 1| appropriate number of shifts
         If AA0 overflowed above, go to LABEL | If true, Q14 will overflow
         WS(I) = RND(AA0)                 | Round to high word for WS
                                          | Overflow cannot occur in remaining cases
                                          | If you reach here then
                                          | continue without the checks
                                          | then copy WS to AWZ

    For  I = 8, 9, 10, 11, do the next 5 lines
         AA0 = WZCFV(I) * AWZTMP(I)
         If NLSAWZTMP = 13, AA0 = AA0 << 3
         If NLSAWZTMP = 14, AA0 = AA0 << 2
         If NLSAWZTMP = 15, AA0 = AA0 << 1
         WS(I) = RND(AA0)
    For  I = 2, 3, ..., 11, do the next line | No overflows, so copy
         AWZ(I) = WS(I)                   | WS to AWZ

                                          | Now do the denominator
                                          | coefficients
                                          | If the numerator did not overflow,
                                          | then the denominator cannot, either

    For  I = 2, 3, ..., 11, do the next 5 lines
         AA0 = WPCFV(I) * AWZTMP(I)       | WPCFV is Q14; AA0 is 14 + NLSAWZTMP
         If NLSAWZTMP = 13, AA0 = AA0 << 3| Make AA0 Q30 for all 3 cases
         If NLSAWZTMP = 14, AA0 = AA0 << 2| appropriate number of shifts
         If NLSAWZTMP = 15, AA0 = AA0 << 1|
         AWP(I) = RND(AA0)                | Round to high word for AWP

    Exit this subroutine

LABEL:                                    | If program proceeds to here, we will have an overflow
                                          | if we try to represent AWZ in Q14. In this case,
                                          |  do not update the weighting filter coefficients
                                          | (i.e. keep using the filter coefficients from the
                                          | previous adaptation cycle).
```

G.3.14 Block 43 – Hybrid windowing module

In this subclause both the floating point and fixed point pseudo-code for block 43 are given. First, the floating point pseudo-code is presented.

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

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

For  I = 1, 2, ..., LPCLG + 1, do the next 4 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 3 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
```

Note that before this routine is called, GTMP( ) is assigned as

```
GTMP(1) = GSTATE(4)
GTMP(2) = GSTATE(3)
GTMP(3) = GSTATE(2)
GTMP(4) = GSTATE(1)
```

and the initial values of GSTATE( ) are –32 in floating point, which is –16384 in Q9 fixed point. Now we give the fixed point version of the same module. In this code we have added several new variables. NLSREXPLG is a global variable holding the number of left shifts for normalizing REXPLG. This variable is initialized with a value of 31.

```
N1 = LPCLG + NUPDATE (= 10 + 4)                    | Compute some constants (can be
N2 = LPCLG + NONRLG (= 10 + 20)                    | precomputed and stored in memory)
N3 = LPCLG + NUPDATE + NONRLG (= 10 + 4 + 20)

For  N = 1, 2, ..., N2, do the next line
     SBLG(N) = SBLG(N + NUPDATE)                   | Shift the old signal buffer
For  N = 1, 2, ..., NUPDATE, do the next line
     SBLG(N2 + N) = GTMP(N)                        | SBLG(N3) is the newest sample
                                                   | All SBLG are Q9 and represented
                                                   | in 16-bits precision
Call FINDNLS(SBLG, N3, N3, 14, NLS)                | Find the amount of left shifts
NLSTMP = NLS – 1                                   | needed in the next loop for 2 bits
                                                   | of headroom later

K = 1
For  N = 34, 33, ..., 1, do the next 5 lines
     P = SBLG(N) * WNRLG(K)                        | WNRLG is Q15
     If NLSTMP = –1, set AA0 = P >> 1
     If NLSTMP > –1, set AA0 = P << NLSTMP
     WS(N) = RND(AA0)                              | WS(N) is 14 bits or less
     K = K + 1

NLSATTLG = 14
Call HWMCORE(LPCLG, N1, N3, NLSATTLG, WS, NLSTMP, REXPLG, NLSREXPLG, R, ILLCONDG)
```

The subroutine HWMCORE can be found in G.3.18.

The following table lists all variables in this pseudo-code with their representation format and size for easy reference. The table notes whether each variable is temporary (temp), meaning that it need not be stored after the module is completed, or permanent (perm), meaning that the value will be needed after the current calculation as well. The table also notes which variables were not included in the previous floating point pseudo-code (old/new).

Variable	Format	Size	Temp/perm	Old/new
GTMP	Q9	4	perm	old
NLS	integer	1	temp	new
NLSREXPLG	integer	1	perm	new
NLSTMP	integer	1	temp	new
REXPLG	BFL	11	perm	old
R	BFL	11	perm	old
SBLG	Q9	34	perm	old
WS	BFL	34	temp	old
BFL Block floating point Integer 16-bit integer				

G.3.15 Block 45 – Bandwidth expansion module

This is the floating point pseudo-code for block 45, the bandwidth expansion module.

```
If ICOUNT ≠ 2, skip the execution of this block
For  I = 2, 3, ..., LPCLG + 1, do the next line
     GP(I) = FACGPV(I) * GPTMP(I)                    | Scale coefficients
```

The tables for FACGPV are given in Q14 format, as are the tables for the other bandwidth expansion coefficients. The values for the input GPTMP array are in Q13, Q14 or Q15 format. As discussed in the earlier description of the fixed point Levinson-Durbin recursion module, NLSGPTMP is given by the Levinson-Durbin recursion module to indicate which format is used for GPTMP. After the multiplication FACGPV(I)*GPTMP(I) the corresponding amount of left shifts is required.

The final values for GP are always represented in Q14 format. Empirically, the output coefficient arrays of block 45 have never been too large to be represented in Q14 (i.e. requiring Q13 format or lower). However, to be safe, we have to be prepared to handle the unlikely event of Q14 overflow at the output of the bandwidth expansion blocks. In the pseudo-code below, we check for the possibility of Q14 overflow. If such a case is detected, we do something similar to the Levinson-Durbin recursion modules - we do not update the predictor coefficients and keep using the old coefficients of the previous adaptation cycle. Potentially, we could use a switchable Q14/Q13 format, with a flag to signal the filtering modules which of the two possible Q formats are used. However, this will unnecessarily increase the complexity of the DSP code and the execution time. Since Q14 overflow was never observed at the output of bandwidth expansion modules, a simple safety check as implemented below suffices.

This is the fixed point pseudo-code for block 45.

```
If ICOUNT ≠ 2, skip the execution of this block
Otherwise, do the following

                                                     | First check to see if ILLCONDG is true

If ILCONDG = .TRUE., skip the execution of this block
Otherwise, do the following
     GPTMP(1) = 16 384
     For  I = 2, 3, 4, ..., LPCLG + 1, do the next 6 lines
          AA0 = FACGPV(I) * GPTMP(I)                 | AA0 is Q27, Q28 or Q29
          If NLSGPTMP = 13, AA0 = AA0 << 3           | Make AA0 Q30 for all 3 cases by
          If NLSGPTMP = 14, AA0 = AA0 << 2           | appropriate number of shifts
          If NLSGPTMP = 15, AA0 = AA0 << 1           |
          If AA0 overflowed above, go to LABEL       | If not true,
          GPTMP(I) = RND(AA0)                        | round to high word for GP
```

```
    For  I = 2, 3, 4, ..., LPCLG + 1, do the next line      | Everything is normal, copy GPTMP
         GP(I) = GPTMP(I)                                   | to GP and then exit
    Exit this program

LABEL:                                                      | If program proceeds to here, we will have an
                                                            | overflow if we try
                                                            | to represent GP in Q14. In this case, do not update
                                                            | the log-gain predictor coefficients (i.e. keep using
                                                            | the log-gain predictor coefficients of the previous
                                                            | adaptation cycle).
```

G.3.16 Block 46 – Log-gain linear prediction

This is the floating point pseudo-code for the log-gain linear predictor, block 46.

```
LOGGAIN = 0
For   I = LPCLG, LPCLG – 1, ..., 3, 2, do the next 2 lines
      LOGGAIN = LOGGAIN – GP(I + 1) * GSTATE(I)
      GSTATE(I) = GSTATE(I – 1)

LOGGAIN = LOGGAIN – GP(2) * GSTATE(1)
```

LOGGAIN and GSTATE are represented in Q9 format throughout the coder. GP is represented in Q14 format. Here is the fixed point pseudo-code.

```
AA0 = 0
For   I = LPCLG, LPCLG – 1, ..., 3, 2, do the next 3 lines
      P = GP(I + 1) * GSTATE(I)
      AA0 = AA0 – P
      GSTATE(I) = GSTATE(I – 1)

P = GP(2) * GSTATE(1)
AA0 = AA0 – P
AA0 = AA0 >> 14
LOGGAIN = AA0
```

This is the floating point pseudo-code for block 98, the log-gain limiter. Since this code is based on modifications made for fixed point, it does not appear in Recommendation G.728. We include it here in order to have it for comparison purposes with the fixed point pseudo-code to follow.

```
If LOGGAIN > 28., set LOGGAIN = 28
If LOGGAIN < –32., set LOGGAIN = –32
```

Since LOGGAIN is represented in Q9 format, the maximum and minimum thresholds are multiplied by 512. These values are used in the fixed point pseudo-code given below.

```
If LOGGAIN > 14336, set LOGGAIN = 14336
If LOGGAIN < –16384, set LOGGAIN = –16384
```

This is the floating point pseudo-code for the Log-Gain Offset Adder which is block 99.

```
Z = LOGGAIN + GOFF
```

The floating point value of GOFF is 32 and its fixed point value is 16384, which corresponds to 512 * 32. Since LOGGAIN has a range between –32 and +28, Z has a range of 0 to 60. The fixed point code is identical to the floating point code.

This is the floating point code for block 48, the Inverse Logarithm Calculator.

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

The complete value we wish can be expressed in terms of the antilog of 2. It is

$$10^{0.05\,Z} = 2^{0.05\,\log_2(10)\,Z} = 2^{0.1660964\,Z}$$

We let X = 0.1660964 Z, which will have a range from 0 to 9.97. Finally, we let X = [X] + x, where [X] is the greatest integer less than or equal to X and x is the fractional part. The value of $2^{[X]}$ is exact and only needs to be represented by its exponent. What remains is the problem of computing the value for the fractional part.

In computing X, we let 0.1660964 be represented in Q21 format. This corresponds to a number that can be represented as 10 in the upper 16 bits and 20649 in the lower 15 bits. We multiply Z by both parts separately in order to get good precision for X. We then separate [X] and x. In computing the exponential for the fractional part we know $0 < x < 1$, so $1 < 2^x < 2$. Therefore, we can use the following fixed representations: x is Q15 and 2^x is Q14. We use a Taylor series expansion to compute 2^x:

$$2^x = \Big(((c4\,x + c3)\,x + c2)\;x + c1\Big)x + c0$$
$$= c4\,x^4 + c3\,x^3 + c2\,x^2 + c1\,x + c0$$

The c values are stored in Q14 and Q15 and are given by

c4 = 323 = 0.0098571 in Q15
c3 = 1874 = 0.0571899 in Q15
c2 = 7866 = 0.2400512 in Q15
c1 = 22702 = 0.6928100 in Q15
c0 = 16384 = 1.0 in Q14

Here is the pseudo-code for computing $10^{0.05\ GAIN}$ on a 16-bit DSP with two 32-bit accumulators. It is assumed that GAIN is in Q9 format and the offset of 32 dB has already been added to it.

```
AA0 = 10 * Z                | Z is Q9, 10 is Q6, so AA0 is Q15
AA1 = 20649 * Z             | 20649 is Q21, so AA1 is Q30
AA1 = AA1 << 1              | Make AA1 Q31
AA1 = RND(AA1)              | Round AA1 to make it Q15 in
                            | low word
AA0 = AA0 + AA1             | AA0 = [X] + x in Q15
AA1 = AA0 >> 15             | Want [X] in Q0 and x in Q15
NLS = AA1                   | NLS = [X]
AA1 = AA1 << 15
x = AA0 – AA1               | x is the fractional part in Q15

                            | Compute 2 ** x
AA0 = c4 * x                | Q15 * Q15 = => AA0 is Q30
AA0 = AA0 << 1              | AA0 now Q31
AA1 = c << 16               | AA1 is Q31
AA0 = AA0 + AA1
TMP = RND(AA0)              | TMP is Q15, use routing
AA0 = TMP * x               | Q15 * Q15 = => AA0 is Q30
AA0 = AA0 << 1              | AA0 now Q31
AA1 = c2 << 16              | AA1 is Q31
AA0 = AA0 + AA1
TMP = RND(AA0)              | TMP is Q15, use routing
AA0 = TMP * x               | Q15 * Q15 = => AA0 is Q30
AA0 = AA0 << 1              | AA0 now Q31
AA1 = c1 << 16              | AA1 is Q31
AA0 = AA0 + AA1
TMP = RND(AA0)              | TMP is Q15, use routing
AA0 = TMP * x               | Q15 * Q15 = => AA0 is Q30
                            | No left shift this time!!
AA1 = c0 = << 16            | AA1 is Q30
AA0 = AA0 + AA1
GAIN = RND(AA0)             | GAIN is Q14 and contains 2 ** x
                            | NLSGAIN is 14 – NLS for 2 ** X
NLSGAIN = 14 – NLS          | Q factor for result
```

The following is the floating point pseudo-code for blocks 96 and 97, the calculation of the new value for GSTATE(1). This block does not appear in Recommendation G.728 and is given here for comparison with the fixed point pseudo-code to be given immediately following. The input variables are the GAIN value output from block 98, GCBLG, the dB value of the gain codebook entry selected for the previous excitation vector, and SHAPELG, the dB value of the shape codevector selected for the previous excitation vector. These values are given in Tables G.3 and G.4, respectively. Those tables give both the floating point and fixed point representations for the values. The fixed point representations are in Q11 format. The floating point pseudo-code is:

```
GSTATE(1) = LOGGAIN + GCBLG(IG) + SHAPELG(IS)
If GSTATE(1) < –32., set GSTATE(1) = –32.
```

The fixed point pseudo-code follows.

```
AA0 = LOGGAIN << 7                          | Align decimal points at
AA0 = AA0 + (GCBLG(IG) << 5)                | boundary between the high and
AA0 = AA0 + (SHAPELG(IS) << 5)              | low words of the accumulator

AA0 = AA0 >> 7                              | Right shift back to Q9 format

IF AA0 < –16384, set AA0 = –16384           | Check lower limit

GSTATE(1) = AA0                             | Lower 16-bit word saved
```

G.3.17 Block 49 – Hybrid window module for synthesis filter

We begin with the floating point pseudo-code for the hybrid windowing module.

```
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 2 lines
      WS(N) = SB(N) * WNR(K)                | Multiply the window function
      K = K + 1

For   I = 1, 2, ..., LPC + 1, do the next 4 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 3 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
```

The fixed point pseudo-code for the hybrid windowing module (block 49) is much more complicated than the floating point version. This is due to the special handling of Segmental Block Floating Point (SBFL) format which is needed to retain sufficient numerical precision.

The STTMP array contains 4 quantized speech vectors of the previous adaptation cycle. When each of these 4 quantized speech vector (the ST array) was computed, it was represented in 14-bit precision BFL format. The number of left shifts (NLS) for the 4 quantized vectors will, in general, be different. For this reason, the STTMP array is said to be stored in SBFL format since it is the concatenation of 4 BFL ST vectors. The SB array is the concatenation of 21 BFL ST vectors. For this reason the SB array is stored in the same 14-bit precision SBFL format. For each of the 4 vectors composing STTMP, there is an associated NLS value. These are stored in the array NLSSTTMP( ). For the 21 vectors composing SB, the NLS values are stored in the array NLSSB().

Next, we give the fixed point pseudo-code for the hybrid windowing module.

```
N1 = LPC + NFRSZ (= 70)                          | Compute some constants (can be
N2 = LPC + NONR (= 85)                           | precomputed and stored in memory)
N3 = LPC + NFRSZ + NONR (= 105)
N4 = N3/IDIM (= 21)
N5 = NFRSZ/IDIM (= 4)
N6 = N4 – N5 (= 17)

For   N = 1, 2, ..., N2, do the next line
      SB(N) = SB(N + NFRSZ)                      | Shift old part of buffer SB
For   N = 1, 2, ..., N6, do the next line
      NLSSB(N) = NLSSB(N + N5)                   | Shift old NLSSB
For   N = 1, 2, ..., NFRSZ, do the next line
      SB(N2 + N) = STTMP(N)                      | Shift in new part of SB
For   N = 1, 2, ..., N5, do the next line
      NLSSB(N6 + N) = NLSSTTMP(N)                | Shift in new NLSSB

                                                 | Now find the minimum NLSSB,
                                                 | this determines NLSWS
NLSTMP = Min{NLSSB(1), NLSSB(2), ..., NLSSB(N4)}

K = 1                                            | Now multiply SB by
N = N3                                           | the hybrid window function
For   J = 1, 2, ..., N4, do the next 8 lines
      NRSH = NLSSB(J) – NLSTMP – 1               | –1 to compensate for Q15 multiplication
      For  M = 1, 2, ..., IDIM, do the next 6 lines
           P = SB(K) * WNR(N)                    | WNR is Q15 multiplication
           If NRSH = –1, set AA0 = P << 1
           If NRSH > –1, set AA0 = P >> NRSH
           WS(K) = RND(AA0)                      | Round upper word and store in WS
           N = N – 1
           K = K + 1

NLSATT50 = 14
Call HWMCORE(LPC, N1, N3, NLSATT50, WS, NLSTMP, REXP, NLSREXP, RTMP, ILLCOND)
```

NOTE – The following table lists all variables in this pseudo-code with their representation format and size for easy reference. The table notes whether each variable is temporary (temp), meaning that it need not be stored after the module is completed, or permanent (perm), meaning that the value will be needed after the current calculation as well. The table also notes which variables were not included in the previous floating point pseudo-code (old/new).

Variable	Format	Size	Temp/perm	Old/new
NLSSB	integer	21	perm	new
NLSREXP	integer	1	perm	new
NLSSTTMP	integer	4	perm	new
NLSTMP	integer	1	temp	new
NRSH	integer	1	temp	new
REXP	BFL	51	perm	old
RTMP	BFL	51	perm	old
SB	SBFL	105	perm	old
STTMP	SBFL	20	perm	old
WS	BFL	105	temp	old

BFL Block floating point
Integer 16-bit integer
SBFL 14-bit precision segmented block floating point
WS 14-bit precision BFL
REXP and RTMP 16-bit precision

G.3.18 HWMCORE – Core of hybrid window module

This module is used to complete the hybrid window calculation for blocks 36, 43 and 49. Each of those blocks has its own initial portion. Variables are passed along from those blocks to this module. In order to avoid confusion, we have renamed certain variables so that this pseudo-code does not use names associated with any one of those three blocks. The following table matches the names used in this module with the names used in blocks 36, 43 and 49.

Variable	Block 36	Block 43	Block 49
LPO	LPCW (=10)	LPCLG (=10)	LPC (=50)
NLSATT	NLSATTW	NLSATTLG	NLSATT50
NLSRREC	NLSREXPW	NLSREXPLG	NLSREXP
N1	30	14	70
N3	60	34	105
R	R	R	RTMP
RREC	REXPW	REXPLG	REXP

In addition to these variables, the scratch array WS and the corresponding number of shifts, NLSTMP, are also passed from those blocks to this module.

The following is the fixed point pseudo-code for this module.

```
SUBROUTINE HWMCORE(LPO, N1, N3, NLSATT, WS, NLSTMP, RREC, NLSRREC, R, ILLCOND)

NLSAA0 = 2 * NLSTMP
AA0 = 0                                          | Compute recursive part of RREC(1)
For N = LPO + 1, ..., N1, do the next 2 lines
    P = WS(N) * WS(N)                            | WS has 2 bits of headroom
    AA0 = AA0 + P                                | AA0 will have 5 bits of headroom
                                                 | for energy calculation


                                                 | Case 1: NLSRREC > NLSAA0
If NLSRREC > NLSAA0, do the next 22 lines
    AA0 = AA0 >> 1
    IR = NLSRREC – NLSAA0 + 1
    AA1 = RREC(1) << NLSATT                      | This can be done by multiplication
    AA1 = –AA1 + (RREC(1) << 16)                 | Scale RREC by attenuation factor
    AA1 = AA1 >> IR                              | Align AA0 and AA1
    AA0 = AA0 + AA1
    Call VSCALE(AA0, 1, 1, 30, AA0, NLSRE)       | Find NLS for RREC
    RREC(1) = RND(AA0)                           | Upper 16 bits of AA1 saved
    NLSRREC = NLSAA0 – 1 + NLSRE
    For I = 1, 2, ..., LPO, do the next 11 lines
        AA0 = 0                                  | Compute recursive part of RREC(I + 1)
        For N = LPO + 1, ..., N1, do the next 2 lines
            P = WS(N) * WS(N – I)
            AA0 = AA0 + P
        AA0 = AA0 >> 1
        AA1 = RREC(I + 1) << NLSATT              | Scale RREC by 3/4 or 1/2
        AA1 = AA1 + (RREC(I + 1) << 16)          |
        AA1 = AA1 >> IR
        AA0 = AA0 + AA1
        AA0 = AA0 << NLSRE
        RREC(I + 1) = RND(AA0)                   | Upper 16 bits of AA0 saved
    Go to FIN_RECUR


                                                 | Case 2: NLSRREC = NLSAA0
If NLSRREC = NLSAA0, do the next 21 lines
    AA1 = RREC(1) << NLSATT                      | Scale RREC by 3/4 or 1/2
    AA1 = –AA1 + (RREC(1) << 16)                 |
    AA0 = AA0 >> 1
    AA1 = AA1 >> 1
    AA0 = AA0 + AA1
    Call VSCALE(AA0, 1, 1, 30, AA0, NLSRE)       | Find NLS for RREC
    RREC(1) = RND(AA0)                           | Upper 16 bits of AA1 saved
    NLSRREC = NLSRREC – 1 + NLSRE
    For I = 1, 2, ..., LPO, do the next 11 lines
        AA0 = 0                                  | Compute recursive part of RREC(I + 1)
        For N = LPO + 1, ..., N1, do the next 2 lines
            P = WS(N) * WS(N – I)
            AA0 = AA0 + P
        AA0 = AA0 >> 1
        AA1 = RREC(I + 1) << NLSATT              | Scale RREC by 3/4 or 1/2
        AA1 = –AA1 + (RREC(I + 1) << 16)         |
        AA1 = AA1 >> 1
        AA0 = AA0 + AA1
        AA0 = AA0 << NLSRE
        RREC(I + 1) = RND(AA0)                   | Upper 16 bits of AA0 saved
    Go to FIN_RECUR
```

```
                                                    | Case 3: NLSRREC < NLSAA0
If NLSRREC < NLSAA0, do the next 21 lines
    IR = NLSAA0 – NLSRREC + 1
    AA0 = AA0 >> IR
    AA1 = RREC(1) << NLSATT                         | Scale RREC by 3 /4 or 1/2
    AA1 = –AA1 + (RREC(1) << 16)                    |
    AA1 = AA1 >> 1
    AA0 = AA0 + AA1
    Call VSCALE(AA0, 1, 1, 30, AA0, NLSRE)
    RREC(1) = RND(AA0)                              | Upper 16 bits of AA1 saved
    NLSRREC = NLSRREC – 1 + NLSRE
    For I = 1, 2, ..., LPO, do the next 11 lines
        AA0 = 0                                     | Compute recursive part of RREC(I + 1)
        For N = LPO + 1, ..., N1, do the next 2 lines
            P = WS(N) * WS(N – I)
            AA0 = AA0 + P
        AA0 = AA0 >> IR
        AA1 = RREC(I + 1) << NLSATT                 | Scale RREC by 3/4 or 1/2
        AA1 = –AA1 + (RREC(I + 1) << 16)            |
        AA1 = AA1 >> 1
        AA0 = AA0 + AA1
        AA0 = AA0 << NLSRE
        RREC(I + 1) = RND(AA0)                      | Upper 16 bits of AA0 saved

FIN_RECUR:                                          | When you reach this point the
                                                    | recursive component has been computed

    AA0 = 0                                         | Compute non-recursive part of R(1)
    For N = N1 + 1, ..., N3, do the next 2 lines
        P = WS(N) * WS(N)
        AA0 = AA0 + P

                                                    | Case 1: NLSRREC > NLSAA0
    If NLSRREC > NLSAA0, do the next 21 lines
        IR = NLSRREC – NLSAA0 + 1
        AA1 = RREC(1) << 16
        AA1 = AA1 >> IR
        AA0 = AA0 >> 1
        AA1 = AA0 + AA1
        AA0 = AA1 >> 8                              | Apply white noise correction factor
        AA1 = AA1 + AA0
        Call VSCALE(AA1, 1, 1, 30, AA1, NLSRR)
        R(1) = RND(AA1)                             | Upper 16 bits of AA1 saved

    For I = 1, 2, ..., LPO, do the next 10 lines
        AA0 = 0                                     | Compute non-recursive part of R(I + 1)
        For N = N1 + 1, ..., N3, do the next 2 lines
            P = WS(N) * WS(N – I)
            AA0 = AA0 + P
        AA0 = AA0 >> 1
        AA1 = RREC(I + 1) << 16
        AA1 = AA1 >> IR
        AA1 = AA0 + AA1
        AA1 = AA1 << NLSRR
        R(I + 1) = RND(AA1)                         | Save upper 16 bits
    Go to END
```

```
                                                    | Case 2: NLSRREC = NLSAA0
    If NLSRREC = NLSAA0, do the next 18 lines
        AA0 = AA0 >> 1
        AA1 = RREC(1) << 15                         | This can be done by multiplication
        AA1 = AA0 + AA1
        AA0 = AA1 >> 8                              | Apply white noise correction factor
        AA1 = AA1 + AA0
        Call VSCALE(AA1, 1, 1, 30, AA1, NLSRR)
        R(1) = RND(AA1)                             | Upper 16 bits of AA1 saved
    For I = 1, 2, ..., LPO, do the next 9 lines
        AA0 = 0                                     | Compute non-recursive part of R(I + 1)
        For N = N1 + 1, ..., N3, do the next 2 lines
            P = WS(N) * WS(N – I)
            AA0 = AA0 + P
        AA0 = AA0 >> 1
        AA1 = RREC(I + 1) << 15
        AA1 = AA0 + AA1
        AA1 = AA1 << NLSRR
        R(I + 1) = RND(AA1)                         | Save upper 16 bits
    Go to END



                                                    | Case 3: NLSRREC < NLSAA0
    If NLSRREC < NLSAA0, do the next 18 lines
        IR = NLSAA0 – NLSRREC + 1
        AA0 = AA0 >> IR
        AA1 = RREC(1) << 15                         | This can be done by multiplication
        AA1 = AA0 + AA1
        AA0 = AA1 >> 8                              | Apply white noise correction factor
        AA1 = AA1 + AA0
        Call VSCALE(AA1, 1, 1, 30, AA1, NLSRR)
        R(1) = RND(AA1)                             | Upper 16 bits of AA1 saved
        For I = 1, 2, ..., LPO, do the next 9 lines
        AA0 = 0                                     | Compute non-recursive part of R(I + 1)
        For N = N1 + 1, ..., N3, do the next 2 lines
            P = WS(N) * WS(N – I)
            AA0 = AA0 + P
        AA0 = AA0 >> IR
        AA1 = RREC(I + 1) << 15
        AA1 = AA0 + AA1
        AA1 = AA1 << NLSRR
        R(I + 1) = RND(AA1)                         | Save upper 16 bits



END:                                                | One last job, check for ill-conditioning
    ILLCOND = .FALSE.
    If AA1 = 0, set ILLCOND = .TRUE                 | AA1 still contains 32 bit R(LPO + 1)
```

NOTE – The following table lists all variables in this pseudo-code with their representation format and size for easy reference. The table notes whether each variable is temporary (temp), meaning that it need not be stored after the module is completed, or permanent (perm), meaning that the value will be needed after the current calculation as well. The table also notes which variables were not included in the previous floating point pseudo-code (old/new).

Variable	Format	Size	Temp/perm	Old/new
NLSRE	integer	1	temp	new
NLSRR	integer	1	temp	new
NLSRREC	integer	1	perm	new
NLSTMP	integer	1	temp	new
RREC	BFL	51	perm	old
R	BFL	51	perm	old
WS	BFL	105	temp	old

BFL	Block floating point
Integer	16-bit integer
SBFL	14-bit precision segmented block floating point
WS	14-bit precision BFL
RREC and R	16-bit precision
RREC	Represents either REXP, REXPW or REXPLG, depending on whether this module is called from block 49, 36 or 43

G.3.19 Block 51 – Bandwidth expansion module

This is the floating point pseudo-code for block 51, the bandwidth expansion module. A similar code also exists for block 45 for the log gain linear predictor bandwidth expansion module. In that instance a different table is used and the number of filter coefficients is greater.

```
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
    A(I) = ATMP(I)
```

The tables for FACV are given in Q14 format for the other bandwidth expansion coefficients. The values for the input ATMP array are in Q13, Q14 or Q15 format. As discussed in the earlier description of the fixed point Levinson-Durbin recursion module, NLSATMP is given by the Levinson-Durbin recursion module to indicate which format is used for ATMP. After the multiplication FACV(I) * ATMP(I) the corresponding amount of left shifts are required.

The final values for ATMP are always represented in Q14 format. Empirically, the values of ATMP have never been too large to be represented in Q14 (i.e. requiring Q13 format or lower). However, to be safe, we have to be prepared to handle the unlikely event of Q14 overflow at the output of the bandwidth expansion module. In the pseudo-code below, we check for the possibility of Q14 overflow. If such a case is detected, we do something similar to the Levinson-Durbin recursion modules - we do not update the predictor coefficients and keep using the old coefficients of the previous adaptation cycle. Potentially, we could use a switchable Q14/Q13 format, with a flag to signal the filtering modules which of the two possible Q formats are used. However, this will unnecessarily increase the complexity of the DSP code and the execution time. Since Q14 overflow was never observed at the output of bandwidth expansion modules, a simple safety check as implemented below suffices.

This is the fixed point pseudo-code for block 51.

```
If ICOUNT ≠ 3, skip the following
Otherwise, do the following.
                                                    | First check to see if ILLCOND is true
If ILLCOND = .TRUE., skip the execution of this block
Otherwise, do the following
        ATMP(1) = 16384
        For I = 2, 3, 4, ..., LPC + 1, do the next 6 lines
            AA0 = FACV(I) * ATMP(I)                 | AA0 is Q27, Q28 or Q29
            If NLSATMP = 13, AA0 = AA0 << 3         | Make AA0 Q30 for all 3 cases by
            If NLSATMP = 14, AA0 = AA0 << 2         | appropriate number of shifts
            If NLSATMP = 15, AA0 = AA0 << 1
            If AA0 overflowed above, go to LABEL    | If not true,
            ATMP(I) = RND(AA0)                      | round to high word for ATMP

        For I = 2, 3, ..., LPC + 1, do the next line
            A(I) = ATMP(I)

        Exit this module

LABEL:                                              | If program proceeds to here, we will have an
                                                    | overflow if we try to represent A in Q14.
                                                    | In this case, do not update the synthesis filter
                                                    | coefficients (i.e. keep using
                                                    | the synthesis filter coefficients from the previous
                                                    | adaptation cycle).
```

G.3.20 Blocks 71 and 72 – Long-term and short-term postfilters

Blocks 71 and 72 are combined in order to preserve the precision of the intermediate variable TEMP which was passed between them in the floating point pseudo-code. The floating point pseudo-code for both of these blocks is given first.

```
        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

        For K = 1, 2, ..., IDIM, do the following
            TEMP = TEMP(K)
            For J = 10, 9, ..., 3, 2, do the next 2 lines
                TEMP(K) = TEMP(K) + STPFFIR(J) * AZ(J + 1)   | All-zero part
                STPFFIR(J) = STPFFIR(J – 1)                 | 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 2 lines
                TEMP(K) =  TEMP(K) – STPFIIR(J) * AP(J + 1)  | All-pole part
                STPFIIR(J) = STPFIIR(J – 1)                 | 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
```

The fixed point pseudo-code is given by the following. The variables STPFFIR and STPFIIR are in Q2 throughout.

```
For K = 1, 2, ..., IDIM, do the following indented lines
                                                    | First do long-term postfilter
    AA0 = GL * SST(K)                               | GL is Q14, SST(1:5) is Q2
    AA0 = AA0 + GLB * SST(K – KP)                   | GLB is Q16, SST(–239:0) is Q0
```

```
AA1 = AA0                                          | Next do short-term postfilter
                                                   | Perform FIR part of filter
For J = 10, 9, ..., 3, 2, do the next 2 lines
     AA1 = AA1 + STPFFIR(J) * AZ(J + 1)            | AZ is Q14, STPFFIR(J) is Q2
     STPFFIR(J) = STPFFIR(J – 1)
AA1 = AA1 + STPFFIR(1) * AZ(2)
AA0 = AA0 << 2
STPFFIR(1) = RND(AA0)                              | Q2 for STPFFIR

                                                   | Now perform IIR part of filter
For J = 10, 9, ..., 3, 2, do the next 2 lines
     AA1 = AA1 – STPFIIR(J) * AP(J + 1)            | AP is Q14, STPFIIR(J) is Q2
     STPFIIR(J) = STPFIIR(J – 1)
AA1 = AA1 – STPFIIR(1) * AP(2)

AA0 = AA0 >> 14
                                                   | Now check for saturation
If AA0 > 32767, set AA0 = 32767
If AA0 < –32768, set AA0 = –32768
STPFIIR(1) = AA0

                                                   | Now do spectral compensation
                                                   | tilt filter

AA1 = AA1 + STPFIIR(2) * TILTZ                     | TILTZ is Q14
AA1 = AA1 >> 14
If AA1 > 32767, set AA1 = 32767
If AA1 < –32768, set AA1 = –32768
TEMP(K) = AA1

                                                   | Now shift the long-term postfilter
                                                   | memory buffer
For K = –NPWSZ – KPMAX + 1, ..., –7, –6, –5, do the next line
     SST(K) = SST(K + IDIM)                        | Shift decoded
                                                   | speech buffer
For K = –4, –3, ..., 0, do the next line
     SST(K) = SST(K + IDIM) >> 2                   | Shift decoded speech buffer
                                                   | and change from Q2 to Q0
```

G.3.21 Blocks 73 and 74 – Sum of absolute value calculators

Blocks 73 and 74 are quite similar. Their results are kept in double precision. As indicated here, these results need not be stored before block 75. The floating point pseudo-code for block 73 is given by the following. Note that we have substituted the name SST for the variable ST in floating point code here. This is to keep consistency between this code and the fixed point code presented below.

Recall that SST(1:5) is represented in Q2.

```
SUMUNFIL = 0
For K = 1, 2, ..., IDIM, do the next line
     SUMUNFIL = SUMUNFIL + | SST(K) |
```

The pseudo-code for block 74 is given by the following.

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

The fixed point pseudo-code for these two blocks is given by the following.

```
AA1 = 0
AA0 = 0
For K = 1, 2, ..., IDIM, do the next 2 lines
        AA0 = AA0 + | SST(K) |                  | Add absolute value of SST(K)
        AA1 = AA1 + | TEMP(K) |                 | Add absolute value of TEMP(K)
                                                 | AA0 = SUMUNFIL
                                                 | AA1 = SUMFIL
                                                 | SST and TEMP are Q2, so
                                                 | AA0 and AA1 are also Q2
                                                 | AA0 and AA1 will be used in block 75
```

G.3.22 Block 75 – Scaling factor calculator

Block 75 calculates the ratio of SUMUNFIL/SUMFIL and the result is stored in SCALE in NLSSCALE precision. SUMUNFIL(AA0) and SUMFIL(AA1) come from blocks 73 and 74, respectively. The floating point-pseudo-code is given by the following.

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

The fixed point pseudo-code is given by the following.

```
If AA1 > 4, do the following indented lines
    Call VSCALE(AA1, 1, 1, 30, AA1, NLSDEN)
    DEN = RND(AA1)
    Call VSCALE(AA0, 1, 1, 30, AA0, NLSNUM)
    NUM = RND(AA0)                               | NLSNUM and NLSDEN are both off by
                                                 | 16 which cancels out
    Call DIVIDE(NUM, NLSNUM, DEN, NLSDEN, SCALE, NLSSCALE)
    Otherwise, set SCALE = 16384 and NLSSCALE = 14
```

G.3.23 Block 76 – First-order lowpass filter and block 77 – Output gain scaling unit

The floating point pseudo-code for these two blocks is given by the following.

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

In the fixed point pseudo-code, the second term is computed once and then added in each iteration in order to save both the subtraction and the multiplication inside the loop. The fixed point pseudo-code is given by the following.

```
AA1 = AGCFAC1 * SCALE                            | AGCFAC1 = 20972 in Q21 = 0.010000228
NRS = NLSSCALE – 14 + (21 – 14)                  | Compute right shift
If NRS ≥ 0, AA1 = AA1 >> NRS                     | Want AA1 to be Q28
If NRS < 0, AA1 = AA1 << –NRS                    | Left shift if NRS is negative

For K = 1, 2, ..., IDIM, do the following
                                                 | Lowpass filtering
    AA0 = AA1 + AGCFAC * SCALEFIL                | AGCFAC = 16220 in Q14 and SCALEFIL
                                                 | is Q14
    AA0 = AA0 << 2                               | Make SCALEFIL Q14
    SCALEFIL = RND(AA0)
                                                 | Scale output
    AA0 = SCALEFIL * TEMP(K)                     | TEMP(K) is Q2
    AA0 = AA0 << 2
    SPF(K) = RND(AA0)                            | SPF(K) is Q2
```

G.3.24 Block 81 – 10th order LPC inverse filter

This is the floating point version of the pseudo-code for block 81, the 10th order LPC inverse filter.

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

For   K = 1, 2, ..., IDIM, do the next 7 lines
      ITMP = IP + K
      D(ITMP) = ST(K)
      For J = 10, 9, ..., 3, 2, do the next 2 lines
            D(ITMP) = D(ITMP) + STLPCI(J) * APF(J + 1)     | FIR filtering
            STLPCI(J) = STLPCI(J – 1)                     | Shift in input
      D(ITMP) = D(ITMP) + STLPCI(1) * APF(2)              | Handle last one
      STLPCI(1) =  ST(K)                                  | Shift in input

IP = IP + IDIM                                            | Update IP
```

In the fixed code, we first need to convert ST from block floating point to fixed Q2 format, then write the Q2 version of ST to the long-term postfilter memory buffer, SST, for use later by the long-term postfilter. Note that this buffer was previously labelled ST in the floating point pseudo-code. ST is block floating point and the memory buffer is Q2. In order to avoid confusion, it was necessary to rename the memory buffer SST. We then compute the LPC inverse filter. Note that the coefficients for the LPC filter, APF, are represented in Q13.

```
NLS = 16 – NLSST + 2                                      | Compute left shift amount for Q2
For K = 1, 2, ..., IDIM, do the next 2 lines
      AA0 = ST(K) << NLS
      SST(K) = RND(AA0)                                   | SST is new long-term
                                                          | postfilter buffer
If IP = NPWSZ, then set IP = NFRSZ                        | Check and update IP

                                                          | Start LPC inverse filtering
For K = 1, 2, ..., IDIM, do the next 10 lines
      AA0 = SST(K)
      AA0 = AA0 << 13
      For J = 10, 9, ..., 3, 2, do the next 2 lines
            AA0 = AA0 + STLPCI(J) * APF(J + 1)
            STLPCI(J) = STLPCI(J – 1)
      AA0 = AA0 + STLPCI(1) * APF(2)
      STLPCI(1) = SST(K)
      ITMP = IP + K
      AA0 = AA0 << 2
      D(ITMP) = RND(AA0)                                  | D(ITMP) is in Q1

IP = IP + IDIM
```

G.3.25 Block 82 – Pitch period extraction module

We begin with the floating point version of the pseudo-code for block 82, the pitch period extraction module.

```
IF ICOUNT ≠ 3, skip the execution of this block
Otherwise, do the following
                                                          | Lowpass filtering and 4:1 downsampling

For K = NPWSZ – NFRSZ + 1, ..., NPWSZ, do the next 7 lines
      TMP = D(K) – STLPF(1) * AL(1) – STLPF(2) * AL(2) – STLPF(3) * AL(3)     | IIR filter
      If K is divisible by 4, do the next 2 lines
            N = K/4                                       | Do FIR filtering only if needed
            DEC(N) = TEMP * 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
M2 = KPMAX/4                                          | in the decimated LPC residual domain
CORMAX = most negative number of the machine
For J = M1, M1 + 1, ..., M2, do the next 6 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 2 lines
            CORMAX = TMP                              | Find maximun correlation
            KMAX = J                                  | and the corresponding lag
For N = –M2 + 1, –M2 + 2, ..., (NPWSZ – NFRSZ)/4, do the next line
      DEC(N) = DEC(N + IDIM)                          | Shift decimated LPC residual buffer


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 6 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 2 lines
            CORMAX = TMP                              | Find maximum correlation
            KP = J                                    | and the 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 cannot 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 6 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 > CMAX, do the next 2 lines
            CMAX = TMP                                | Find maximum correlation
            KPTMP = J                                 | and the corresponding lag
SUM = 0
TMP = 0                                               | Start computing the tap weights
For K = 1, 2, ..., NPWSZ, do the next 2 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
If TAP > 1, set TAP = 1                               | Clamp TAP between 0 and 1
If TAP < 0, set TAP = 0
If TAP1 > 1, set TAP1 = 1                             | Clamp TAP1 between 0 and 1
If TAP1 < 0, set TAP1 = 0
                                                      | Replace KP with fundamental pitch
                                                      | if si TAP1 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
```

In the fixed point version of this block, the D array and the state variables of the lowpass filter are represented in Q1. This is to avoid overflow in correlation and energy computations. The fixed point pseudo-code is given by the following.

```
If ICOUNT ≠ 3, skip the execution of this block
Otherwise, do the following

For K = NPWSZ – NFRSZ + 1, ..., NPWSZ, do the next 17 lines
    AA0 = D(K) * BL(0)                          | First do the FIR part
    AA0 = AA0 + LPFFIR(1) * BL(1)               | D(K) is Q1, BL( ) is Q19
    AA0 = AA0 + LPFFIR(2) * BL(2)               | BL(0) = 18721, BL(1) = –3668
    AA0 = AA0 + LPFFIR(3) * BL(3)               | BL(2) = –3668, BL(3) = 18721
    LPFFIR(3) = LPFFIR(2)
    LPFFIR(2) = LPFFIR(1)
    LPFFIR(1) = D(K)                            | LPFFIR is Q1
    AA0 = AA0 >> 6                              | Now do the IIR part
    AA0 = AA0 – LPFIIR(1) * AL(1)               | AL( ) are Q13, LPFIIR( ) are Q1
    AA0 = AA0 – LPFIIR(2) * AL(2)               | AL(1) = –19172, AL(2) = 16481
    AA0 = AA0 – LPFIIR(3) * AL(3)               | AL(3) = –5031
    LPFIIR(3) = LPFIIR(2)
    LPFIIR(2) = LPFIIR(1)
    AA0 = AA0 << 3
    LPFIIR(1) = RND(AA0)                        | LPFIIR is Q1
    N = (K >> 2)
    If K = (N << 2), set DEC(N) = LPFIIR(1)     | DEC(N) is Q1

M1 = KPMIN/4                                    | Start correlation peak-picking
M2 = KPMAX/4                                    | in the decimated LPC residual domain
AA1 = –2147483648                               | = –2^31

For J = M1, M1 + 1, ..., M2, do the next 6 lines
    AA0 = 0
    For N = 1, 2, ..., NPWSZ/4, do the next line
        AA0 = AA0 + DEC(N) * DEC(N – J)
    If AA0 > AA1, do the next 2 lines
        AA1 = AA0                               | Find maximum correlation and
        KMAX = J                                | the corresponding lag

For N = –M2 + 1, –M2 + 2, ..., (NPWSZ – NFRSZ)/4, do the next line
    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
AA1 = –2147483648                               | = –2^31

For J = M1, M1 + 1, ..., M2, do the next 6 lines
    AA0 = 0
    For K = 1, 2, ..., NPWSZ, do the next line
        AA0 = AA0 + D(K) * DEC(K – J)           | Correlation in undecimated domain
    If AA0 > AA1, do the next 2 lines
        AA1 = AA0
        KP = J
CORMAX = AA1                                    | Double precision save to CORMAX

M1 = KP1 – KPDELTA                              | Determine the range of search around
M2 = KP1 + KPDELTA                              | the pitch period of the previous frame
If KP < M2 + 1, go to LABEL                     | KP cannot be a multiple pitch if true
If M1 < KPMIN, set M1 = KPMIN                   | Check whether M1 out of range
If M2 > KPMAX, set M2 = KPMAX                   | Check whether M2 out of range
                                                | This last statement is not
                                                | in floating point
AA1 = –2147483648                               | = –2^31
```

```
        For J = M1, M1 + 1, ..., M2, do the next 6 lines
            AA0 = 0
            For K = 1, 2, ..., NPWSZ, do the next line
                AA0 = AA0 + D(K) * D(K – J)          | Correlation in undecimated domain
            If AA0 > AA1, do the next 2 lines
                AA1 = AA0                            | Find maximum correlation and
                KPTMP = J                            | the corresponding lag
        CMAX = AA1                                   | Double precision save to CMAX

    AA0 = 0
    AA1 = 0
    For K = 1, 2, ..., NPWSZ, do the next 2 lines
        AA0 = AA0 + D(K – KP) * D(K – KP)
        AA1 = AA1 + D(K – KPTMP) * D(K – KPTMP)

                                                     | Find TAP
                                                     | Clip TAP weights if necessary
    If AA0 = 0, set CORMAX = 0
    If AA1 = 0, set CMAX = 0
    If CORMAX > AA0, set CORMAX = AA0
    If CORMAX < 0, set CORMAX = 0
    If CMAX > AA1, set CMAX = AA1
    If CMAX < 0, set CMAX = 0

    If AA0 > AA1, do the next 2 lines
        call VSCALE(AA0, 1, 1, 30, AA0, NLS)
        AA1 = AA1 << NLS
    otherwise do the next 2 lines
        call VSCALE(AA1, 1, 1, 30, AA1, NLS)
        AA0 = AA0 << NLS

    SUM = AA0 >> 16
    TMP = AA1 >> 16
    AA0 = CORMAX << NLS
    CORMAX = AA0 >> 16
    AA0 = CMAX << NLS
    CMAX = AA0 >> 16
    AA1 = CORMAX * TMP
    AA1 = AA1 >> 16
    AA1 = AA1 * ITAPTH                               | ITAPTH = 26214 in Q16
    AA0 = CMAX * SUM
    If AA0 > AA1, set KP – KPTMP

LABEL:  KP1 = KP                                     | Update KP1 and shift LPC residual
    For  K = –KPMAX + 1, –KPMAX + 2, ..., NPWSZ – NFRSZ, do the next line
        D(K) = D(K + NFRSZ)                          | Shift the LPC residual buffer
```

G.3.26 Block 83 – Pitch predictor tap calculator

We begin with the floating point version of the pseudo-code. Here we have used SST rather than ST for the name of the long-term postfilter memory buffer.

```
    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 2 lines
        SUM = SUM + SST(K – KP) * SST(K – KP)
        TMP = TMP + SST(K) * SST(K – KP)
    If SUM = 0, set PTAP = 0; otherwise, set PTAP = TMP/SUM
```

The fixed point pseudo-code is given by the following. Note that SST( ) is 13 bit Q0. In performing the correlations, the mulplication of SST by either itself or a delayed sample gives a result which is 25 bit Q0.

```
If ICOUNT ≠ 3, skip the execution of this block
Otherwise, do the following
    AA0 = 0
    AA1 = 0
    For K = –NPWSZ + 1, –NPWSZ + 2, ..., 0, do the next 4 lines
        P = SST(K – KP) * SST(K – KP)
        AA0 = AA0 + P
        P = SST(K) * SST(K – KP)
        AA1 = AA1 + P


    If AA0 = 0, set PTAP = 0 and return to calling program
    If AA1 ≤ 0, set PTAP = 0 and return to calling program
    If AA1 ≥ AA0, set PTAP = 16384                          | NLSPTAP = 14
    Otherwise, do the following
        Call VSCALE(AA0, 1, 1, 30, AA0, NLSDEN)
        Call VSCALE(AA1, 1, 1, 30, AA1, NLSNUM)
        NUM = RND(AA1)
        DEN = RND(AA0)
        Call DIVIDE(NUM, NLSNUM, DEN, NLSDEN, PTAP, NLSPTAP)
        NRS = NLSPTAP – 14
        PTAP = PTAP >> NRS                                  | NLSPTAP = 14
```

G.3.27 Block 84 – Long-term postfilter coefficient calculator

We begin with the floating point pseudo-code for block 84.

```
    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)
```

This fixed point pseudo-code is given by the following. We define an additional variable GLB which is the product of GL and B. This saves us later multiplications. B and GLB are output in Q16 and GL is output in Q14.

```
                                                            | Note that PTAP is < 16385 from block 83
If ICOUNT ≠ 3, skip the execution of this block
Otherwise, do the following
    If PTAP < PPFTH, set PTAP = 0                           | PPFTH = 9830 in Q14
    AA0 = PPFZCF * PTAP                                     | PPFZCF = 9830 in Q16, PTAP is in Q14
    B = AA0 >> 14                                           | Save B in Q16
    AA0 = AA0 >> 16                                         | AA0 = B in Q14
    AA0 = AA0 + 16384
    DEN = AA0                                               | DEN is in Q14
    Call DIVIDE(16384, 14, DEN, 14, GL, NLS)
    AA0 = GL * B                                            | NLS = 14 or 15, NLS of B = 16
    GLB = AA0 >> NLS                                        | GLB is GL * B and is precomputed here
                                                            | in Q16 for block 71
    NRS = NLS – 14
    If NRS > 0, SET GL = GL >> NRS                          | Make GL Q14
```

G.3.28 Block 85 – Short-term postfilter coefficient calculator

We begin with the floating point pseudo-code for this block.

```
If ICOUNT ≠ 1, skip the execution of this block
Otherwise, do the following
For I = 2, 3, ..., 11, do the next 2 lines
     AP(I) = SPFPCFV(I) * APF(I)          | Scale denominator coefficients
     AZ(I) = SPFZCFV(I) * APF(I)          | Scale numerator coefficients
TILTZ = TILTF * RC1                       | Tilt compensation filter coefficients
```

In the fixed point pseudo-code, we must consider the possibility that there was ill-conditioning in Durbin's recursion or that the prediction coefficients could not even be expressed in Q13. (It has never been observed that Q13 was not sufficient, but this possibility must still be considered.) The variable ILLCONDP is a flag from block 50 which indicates whether the results of block 50 are valid or not. In Recommendation G.728, there is an implicit assumption that the results of Durbin will not be used if ILLCONDP is true. That is, ATMP will not be copied to APF after the 10th order recursion is completed. The same assumption is repeated here. If ILLCONDP is true, then we do not update AP, AZ or TILTZ.

Next, we must deal with the fact that the coefficients APF( ) from Durbin's recursion may be in Q13, Q14 or Q15. NLSAPF is the number of left shifts of APF. At the output, we also wish to save APF( ) in Q13 for later use in the LPC inverse filtering operation. We want the numerator and denominator coefficients, AP( ) and AZ( ) to be in Q14 for the output. TILTZ is output in Q14. It may be the case that AP cannot be represented in Q14. When this is the case, do not update AP, AZ or TILTZ, but the new values for APF can be used. They should already be in Q13 format. The fixed point pseudo-code is given by the following.

```
If ICOUNT ≠ 1, skip the execution of this block
Otherwise, do the following

                                          | First check to see if ILLCONDP is true

If ILLCONDP = .TRUE., skip the execution of this block
otherwise, do the following

                                          | Next do the denominator coefficients
                                          | If they overflow for Q14, do not
                                          | update AP, AZ or TILTZ
                                          | Temporary array WS is used in case
                                          | of overflow, so that AP is preserved

For I = 2 and 3, do the next 6 lines
     AA0 = SPFPCFV(I) * APF(I)            | SPFPCFV is Q14, AA0 is 14 + NLSAPF
     If NLSAPF = 13, AA0 = AA0 << 3       | Make AA0 Q30 for all 3 cases by
     If NLSAPF = 14, AA0 = AA0 << 2       | appropriate number of shifts
     If NLSAPF = 15, AA0 = AA0 << 1
     If AA0 overflowed above, go to LABEL
     WS(I) = RND(AA0)                     | Round to high word for WS
                                          | Overflow can only occur for 2 and 3,
                                          | so copy these to AP and continue
For I = 2 and 3, do the next line
     AP(I) = WS(I)

                                          | Now do the rest

For I = 4, 5, ..., 11, do the next 5 lines
     AA0 = SPFPCFV(I) * APF(I)            | SPFPCFV is Q14, AA0 is 14 + NLSAPF
     If NLSAPF = 13, AA0 = AA0 << 3       | Make AA0 Q30 for all 3 cases by
     If NLSAPF = 14, AA0 = AA0 << 2       | appropriate number of shifts
     If NLSAPF = 15, AA0 = AA0 << 1
     AP(I) = RND(AA0)                     | Round to high word for AP
                                          | Now do the numerator coefficients
                                          | If the denominator did not overflow,
                                          | then the numerator cannot, either
```

```
For I = 2, 3, ..., 11, do the next 5 lines
     AA0 = SPFZCFV(I) * APF(I)          | SPFZCFV is Q14, AA0 is 14 + NLSAPF
     If NLSAPF = 13, AA0 = AA0 << 3     | Make AA0 Q30 for all 3 cases by
     If NLSAPF = 14, AA0 = AA0 << 2     | appropriate number of shifts
     If NLSAPF = 15, AA0 = AA0 << 1
     AZ(I) = RND(AA0)                   | Round to high word for AZ

                                        | Now update TILTZ
AA0 = TILTF * RC1|  TILTZ = 4915 in Q15
TILTZ = RND(AA0)                        | RC1 is Q15
                                        | TILTZ is Q14

LABEL:                                  | Save APF( ) in Q13 for LPC inverse
                                        | filtering later
                                        | Case 1: NLSAPF = 13, do nothing

If NLSAPF = 14, do the next 3 lines     | Case 2: NLSAPF = 14, shift 15, round
     For I = 2, 3, 4, ..., 11, do the next 2 lines
          AA0 = APF(I) << 15
          APF(I) = RND(AA0)

If NLSAPF = 15, do the next 3 lines     | Case 3: NLSAPF = 15, shift 14, round
     For I = 2, 3, 4, ..., 11, do the next 2 lines
          AA0 = APF(I) << 14
          APF(I) = RND(AA0)
```

Note that in the above code, the "For" loops containing three "If NLSAPF = ..." statements can be eliminated if the entire code is re-written for each of the three possible values of NLSAPF. This longer code will produce exactly the same results, but will execute more quickly on most programmable devices.

G.4 LD-CELP internal variable representations

In this subclause updated versions of Tables 1/G.728 and 2/G.728 are presented. Table G.1 is a shortened version of Table 1/G.728. It lists only constants which are not inherently integers and are not given elsewhere in the Recommendation. The Equivalent Symbol and Initial Value entries in Table 1/G.728 have been deleted in order to leave space for the Fixed Point Format and representation required for each variable. Table G.2 is the integer version of Table 2/G.728. The same column has also been deleted from Table 2/G.728 in order to present the fixed point format. Several new variables are listed which relate only to the fixed point specification.

TABLE G.1/G.728

Basic coder parameters that are not inherently integers and not given elsewhere

Name	Floating-Point Value	Fixed-Point Value	Q format	Description
AGCFAC	0.99	16220	Q14	AGC adaptation speed controlling factor
AGCFAC1	0.01	20972	Q21	The value of (1 – AGCFAC)
GOFF	32	16384	Q9	Log-gain offset value
PPFTH	0.6	9830	Q14	Tap threshold for turning off pitch postfilter
PPFZCF	0.15	9830	Q16	Pitch postfilter zero controlling factor
TAPTH	0.4	26214	Q16	Tap threshold for fundamental pitch replacement
TILTF	0.15	4915	Q15	Spectral tilt compensation controlling factor

TABLE G.2/G.728

LD-CELP internal processing variables

Name	Array Index Range	Fixed Point Format	Description
A	1 to LPC + 1	Q14	Synthesis filter coefficients
AL	1 to 3	Q13	1 kHz lowpass filter denominator coefficients
AP	1 to 11	Q14	Short-term postfilter denominator coefficients
APF	1 to 11	Q13	10th-order LPC filter coefficients
ATMP	1 to LPC + 1	Q13/Q14/Q15	Temporary buffer for synthesis filter coefficients
AWP	1 to LPCW + 1	Q14	Perceptual weighting filter denominator coefficients
AWZ	1 to LPCW + 1	Q14	Perceptual weighting filter numerator coefficients
AWZTMP	1 to LPCW + 1	Q13/Q14/Q15	Temporary buffer for weighting filter coefficients
AZ	1 to 11	Q14	Short-term postfilter numerator coefficients
B	1	Q16	Long-term postfilter coefficients
BL	1 to 4	Q19	1 kHz lowpass filter numerator coefficients
D	–139 to 100	Q1	LPC prediction residual
DEC	–34 to 25	Q1	4:1 decimated LPC prediction residual
ET	1 to IDIM	15b BFL	Gain-scaled excitation vector
FACV	1 to LPC + 1	Q14	Synthesis filter BW broadening vector
FACGPV	1 to LPCLG + 1	Q14	Gain predictor BW broadening vector
G2	1 to NG	Q12	2 times gain levels in gain codebook
GAIN	1	SFL	Linear excitation gain
GB	1 to NG – 1	Q13	Mid-point between adjacent gain levels
GL	1	Q14	Long-term postfilter scaling factor
GLB	1	Q16	Long-term postfilter product term
GP	1 to LPCLG + 1	Q14	Log-gain predictor coefficients, initial value = 16384, –16384, 0, ..., 0
GPTMP	1 to LPCLG + 1	Q13/Q14/Q15	Temporary array for log-gain linear predictor coefficients
GQ	1 to NG	Q13	Gain levels in the gain codebook
GSQ	1 to NG	Q11	Squares of gain levels in gain codebook
GSTATE	1 to LPCLG	Q9	Log-gain predictor memory, initial value = –16384
GTMP	1 to 4	Q9	Temporary log-gain buffer, initial value = –16384
H	1 to IDIM	Q13	Impulse response vector of F(z) W(z)
ICHAN	1	Q0	Best codebook index to be transmitted
ICOUNT	1	Q0	Speech vector counter (indexed from 1 to 4)
IG	1	Q0	Best 3-bit gain codebook index
ILLCOND	1	Q0	III-conditioning flag for synthesis filter
ILLCONDG	1	Q0	III-conditioning flag for log-gain predictor
ILLCONDP	1	Q0	Ill-conditioning flag for postfilter
ILLCONDW	1	Q0	III-conditioning flag for weighting filter
IP	1	Q0	Address pointer to LPC prediction residual
IS	1	Q0	Best 7-bit shape codebook index
KP	1	Q0	Pitch period of the current frame
KP1	1	Q0	Pitch period of the previous frame

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

LD-CELP internal processing variables

Name	Array Index Range	Fixed Point Format	Description
LOGGAIN	1	Q9	Log of excitation gain
LPFFIR	3	Q1	Lowpass filter FIR memory
LPFIIR	3	Q1	Lowpass filter IIR memory
NLSATMP	1	Q0	Durbin's recursion precision flag for ATMP
NLSAWZTMP	1	Q0	Durbin's recursion precision flag for AWZTMP
NLSGPTMP	1	Q0	Durbin's recursion precision flag for GPTMP
NLSET	1	Q0	NLS for ET
NLSGAIN	1	Q0	NLS for linear GAIN
NLSREXP	1	Q0	NLS for REXP, initial value = 31
NLSREXPLG	1	Q0	NLS for REXPLG, initial value = 31
NLSREXPW	1	Q0	NLS for REXPW, initial value = 31
NLSSB	21	Q0	NLS for SB, initial value = 16
NLSST	1	Q0	NLS for ST in decoder
NLSSTATE	11	Q0	NLS for STATELPC, initial value = 16
NLSSTTMP	4	Q0	NLS for STTMP, initial value = 16
PN	1 to IDIM	Q7	Correlation vector for codebook search
PTAP	1	Q14	Pitch predictor tap computed by block 83
R	1 to 11	BFL	Autocorrelation coefficients
RC	1	Q15	Reflection coefficients
RC1	1	Q15	Temporary buffer for first reflection coefficients
REXP	1 to LPC + 1	BFL	Recursive part of autocorrelation, synthesis filter
REXPLG	1 to LPCLG + 1	BFL	Recursive part of autocorrelation, log-gain predictor
REXPW	1 to LPCW + 1	BFL	Recursive part of autocorrelation, weighting filter
RTMP	1 to LPC + 1	BFL	Temporary buffer for autocorrelation coefficients
S	1 to IDIM	15b Q2	Uniform PCM input speech vector
SB	1 to 105	14b BFL	Buffer for previously quantized speech
SBLG	1 to 34	Q9	Buffer for previous log-gain
SBW	1 to 60	Q2	Buffer for previous input speech
SCALE	1	SFL	Unfiltered postfilter scaling factor
SCALEFIL	1	Q14	Lowpass filtered postfilter scaling factor, initial value = 16384
SD	1 to IDIM	Q0	Decoded speech buffer
SPF	1 to IDIM	Q2	Postfiltered speech vector
SPFPCFV	1 to 11	Q14	Short-term postfilter pole controlling vector
SPFZCFV	1 to 11	Q14	Short-term postfilter zero controlling vector
SO	1	byte	A-law or μ-law PCM input speech sample
SST (past)	–239 to 0	13b Q0	Quantized speech buffer
SST (current)	1 to IDIM	15b Q2	Quantized speech buffer
ST	1 to IDIM	14b BFL	Quantized speech vector
STATELPC	1 to LPC	14b SBFL	Synthesis filter memory
STLPCI	1 to 10	Q2	LPC inverse filter memory

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

LD-CELP internal processing variables

Name	Array Index Range	Fixed Point Format	Description
STMP	1 to 4 ∗ IDIM	15b Q2	Buffer for perceptual weighting filter hybrid window
STTMP	1 to 4 ∗ IDIM	14b SBFL	Buffer for synthesis filter hybrid window
STPFFIR	1 to 10	Q2	Short-term postfilter memory, all-zero section
STPFIIR	1 to 10	Q2	Short-term postfilter memory, all-pole section
SU	1	Q2	Uniform PCM input speech sample
SUMFIL	1	Q2	Sum of absolute value of postfiltered speech
SUMUNFIL	1	Q2	Sum of absolute value of decoded speech
SW	1 to IDIM	Q2	Perceptually weighted speech vector
TARGET	1 to IDIM	BFL	(Gain-normalized) VQ target vector
TEMP	1 to IDIM	∗	Scratch array for temporary working space
TILTZ	1	Q14	Short-term postfilter tilt-compensation coefficient
WFIR	1 to LPCW	Q2	Memory of weighting filter 4, all-zero portion
WIIR	1 to LPCW	Q2	Memory of weighting filter 4, all-pole portion
WNR	1 to 105	Q15	Window function for synthesis filter
WNRLG	1 to 34	Q15	Window function for log-gain predictor
WNRW	1 to 60	Q15	Window function for weighting filter
WPCFV	1 to LPCW + 1	Q14	Perceptual weighting filter pole controlling vector
WS	1 to 105	#	Work Space array for intermediate variables
WZCFV	1 to LPCW + 1	Q14	Perceptual weighting filter zero controlling vector
Y	1 to IDIM ∗ NCWD	Q11	Shape codebook array
Y2	1 to NCWD	Q5	Energy of convolved shape codevector
ZIR	1 to IDIM	15b Q2	Zero input response
ZIRWFIR	1 to LPCW	15b Q2	Memory of weighting filter 10, all-zero portion
ZIRWIIR	1 to LPCW	15b Q2	Memory of weighting filter 10, all-pole portion

SFL Scalar floating point
BFL Block floating point
SBFL Segmented block floating point
Qx Qx format
14b or15b Indicates 14- or 15-bit precision, all others are assumed full 16-bit precision
Defined by use, can be BFL or fixed Q, since it is scratch memory

* TEMP is a temporary working array and is used in several blocks; its Q format may change from block to block.

G.5 Log-gain tables for gain and shape codebook vectors

See Tables G.3 and G.4.

TABLE G.3/G.728

Floating point gain in dB and Q11 fixed point representation for gain codebook vectors

Index		dB	Fixed point
0	4	–5.7534180	–11783
1	5	–0.8925781	–1828
2	6	3.9682620	8127
3	7	8.8291020	18082

NOTE – To obtain the fixed point value, multiply the floating point value by $2048 = 2^{11}$.

TABLE G.4/G.728

Floating point gain in dB and Q11 fixed point representation for shape codebook vectors

Index	dB	Fixed point	Index	dB	Fixed point	Index	dB	Fixed point
0	–0.1108398	–227	43	1.1064450	2266	85	1.7133790	3509
1	5.0332030	10308	44	7.0932620	14527	86	0.4252930	871
2	3.1977540	6549	45	9.1738280	18788	87	1.0693360	2190
3	3.7856450	7753	46	6.3623050	13030	88	2.7080080	5546
4	3.7094730	7597	47	3.0458980	6238	89	7.4887700	15337
5	8.0874020	16563	48	0.8911133	1825	90	1.8105470	3708
6	3.1279300	6406	49	4.4384770	9090	91	1.1748050	2406
7	5.8266600	11933	50	0.1030273	211	92	2.8076170	5750
8	6.6254880	13569	51	0.9218750	1888	93	3.6806640	7538
9	5.1606450	10569	52	8.8320310	18088	94	1.9101560	3912
10	7.9726560	16328	53	11.0141600	22557	95	1.7299800	3543
11	3.1914060	6536	54	5.3188480	10893	96	–4.9335940	–10104
12	7.7163090	15803	55	8.8652340	18156	97	0.1479492	303
13	5.6997070	11673	56	1.6728520	3426	98	–3.0083010	–6161
14	10.4091800	21318	57	6.5429690	13400	99	–0.5576172	–1142
15	4.4433590	9100	58	–2.1362300	–4375	100	1.8881840	3867
16	5.9790040	12245	59	3.8916020	7970	101	2.8979490	5935
17	5.8681640	12018	60	3.7861330	7754	102	–3.5161130	–7201
18	1.2221680	2503	61	12.3388700	25270	103	–0.3706055	–759
19	7.1728520	14690	62	2.5942380	5313	104	–1.0219730	–2093
20	8.8818360	18190	63	7.6245120	15615	105	–1.3979490	–2863
21	14.0629900	28801	64	–3.0742190	–6296	106	1.0825200	2217
22	8.2045900	16803	65	2.2021480	4510	107	–1.5834960	–3243
23	9.9272460	20331	66	1.0751950	2202	108	3.0083010	6161
24	8.7983400	18019	67	–3.5297850	–7229	109	2.8579100	5853
25	12.1679700	24920	68	1.5361330	3146	110	3.7104490	7599
26	7.8901370	16159	69	–1.3759770	–2818	111	3.2944340	6747
27	8.6025390	17618	70	–1.3056640	–2674	112	–0.9770508	–2001
28	11.2656300	23072	71	–0.7651367	–1567	113	4.9892580	10218
29	13.7085000	28075	72	0.8989258	1841	114	–0.0263672	–54
30	9.3598630	19169	73	2.8334960	5803	115	0.9335938	1912
31	12.5600600	25723	74	3.8203130	7824	116	5.6127930	11495
32	4.2333980	8670	75	0.1557617	319	117	5.1635740	10575
33	4.9165040	10069	76	0.8862305	1815	118	2.2055660	4517
34	0.2456055	503	77	0.8618164	1765	119	2.0893550	4279
35	4.2221680	8647	78	3.3930660	6949	120	0.8852539	1813
36	5.4516600	11165	79	1.2128910	2484	121	0.2763672	566
37	9.0073240	18447	80	1.3710940	2808	122	2.2309570	4569
38	2.0820310	4264	81	4.7431640	9714	123	2.0278320	4153
39	8.4868160	17381	82	–2.0581050	–4215	124	1.6445310	3368
40	1.7241210	3531	83	3.2607420	6678	125	5.4584960	11179
41	5.1479490	10543	84	1.2861330	2634	126	0.8271484	1694
42	–1.1679690	–2392				127	0.3715820	761

NOTE – To obtain the fixed point value, multiply the floating point value by $2048 = 2^{11}$.

G.6 Integer values of gain codebook related arrays

This subclause includes the equivalent integer values for the floating point table given in Annex B/G.728 (see Table G.5).

TABLE G.5/G.728

Integer values of gain codebook related arrays

Array index	1	2	3	4	5	6	7	8
GQ (Q13)	4224	7392	12936	22638	–4224	–7392	–12936	–22638
GB (Q13)	5808	10164	17787	*	–5808	–10164	–17787	*
G2 (Q12)	4224	7392	12936	22638	–4224	–7392	–12936	–22638
GSQ (Q11)	545	1668	5107	15640	545	1668	5107	15640
* Can be any arbitrary value (not used).								

G.7 Encoder and decoder main program pseudo-codes

This subclause gives the pseudo-codes for the encoder main program and the decoder main program. The main purpose is to show the order in which various blocks are executed. Therefore, only the block execution sequence is shown and no low-level detail of parameter passing is described. Note that the allowable sequence of execution is not unique. There are many different orders of execution which all achieve bit-exact result. The pseudo-codes shown below are just two examples. However, if a different order of block execution is used, the implementer should make sure it gets bit-exact results.

The pseudo-code for the encoder main program is now given below.

```
Initialize all encoder variables to their initial values.

Initialize y2( ) by executing blocks 14 and 15 with h = [8192, 0, 0, 0, 0]

ILLCOND = .FALSE.
ILLCONDW = .FALSE.
ILLCONDG = .FALSE.
ICOUNT = 0

VEC_LOOP:
        If ICOUNT = 4, set ICOUNT = 0                          | Reset vector counter
        ICOUNT = ICOUNT + 1                                    | Update vector counter
        Get one vector of input speech from the input buffer
        Convert input speech vector to the range [–16384, +16383],
            then assign to S( )                                | Q2 representation of [–4096, +4095.75]

                                                               | Check whether to update
                                                               | filter coefficients
        If ICOUNT = 3, do the next 4 lines
            If ILLCOND = .FALSE., do block 51
            If ILLCONDW = .FALSE., do block 38
            do block 12
            do blocks 14 and 15

        If ICOUNT = 2 and ILLCONDG = .FALSE., then do block 45

                                                               | Start once-per-vector processing
        do blocks 46, 98, 99, and 48                           | Get backward-adapted gain
                                                               | GSTATE(1:9) shifted down 1 position
        do "blockzir" (blocks 9 and 10 during zero-input response calculation)
        do block 4                                             | Perceptual weighting filter
        do block 11                                            | VQ target vector computation
```

```
    do block 16                                          | VQ target vector normalization
    do block 13                                          | Time-reversed convolution
    do blocks 17 and 18                                  | Excitation codebook search
    put out ICHAN to the communication channel
    do blocks 19 and 21                                  | Scale selected excitation codevector
    do blocks 9 and 10 during filter memory update       | Get ( )
    do blocks 93, 94, 96, and 97                         | Update log-gain; note that the
                                                         | 3 delay units in the gain adapter just
                                                         | happen naturally in the looping
                                                         | process and need not be
                                                         | implemented explicitly
    GSTATE(1) =  output of block 97                      | Update gain predictor memory
    I = (ICOUNT – 1) * IDIM                              | I = starting address of STTMP( )
    copy ST(1:5) to STTMP(I + 1:I + 5)                   | Update STTMP( )
    NLSSTTMP(ICOUNT) = NLSST
    I = (ICOUNT – 3) * IDIM
    If ICOUNT < 3, set I = I + 20                        | I = starting address of STMP( )
    copy S(1:5) to STMP(I + 1:I + 5)                     | Update STMP( )
                                                         | End of once-per-vector processing

                                                         | Start once-per-frame processing
    If ICOUNT = 4, do the next 2 lines
        do block 49                                      | Output ill-condition flag = ILLCOND
        do block 50                                      | Output predictor coefficients = ATMP( )
                                                         | Output ill-condition flag = ILLCOND
    If ICOUNT = 2, do the next 2 lines
        do block 36                                      | Output ill-condition flag = ILLCONDW
        do block 37                                      | Output predictor coefficients = ATMP( )
                                                         | Output ill-condition flag = ILLCONDW
    If ICOUNT = 1, do the next 6 lines
        GTMP(1) = GSTATE(4)                              | Update GTMP( )in the one shot
        GTMP(2) = GSTATE(3)
        GTMP(3) = GSTATE(2)
        GTMP(4) = GSTATE(1)
        do block 43                                      | Output ill-condition flag = ILLCONDG
        do block 44                                      | Output predictor coefficients = GPTMP( )
                                                         | Output ill-condition flag = ILLCONDG
                                                         | End of once-per-frame processing

Go to VEC_LOOP
```

Next, the pseudo-code for the decoder main program is given below. Again, only the block execution sequence is shown and no low-level detail of parameter passing is described.

Initialize all decoder variables to their initial values.

```
ILLCOND = .FALSE.
ILLCONDG = .FALSE.
ILLCONDP = .FALSE.
ICOUNT = 0

VEC_LOOP:
        If ICOUNT = 4, set ICOUNT = 0                    | Reset vector counter
        ICOUNT = ICOUNT + 1                              | Udate vector counter
        Get ICHAN of the current vector from the input buffer
        Obtain the shape index IS and gain index IG from ICHAN

                                                         | Check whether to update
                                                         | filter coefficients
```

```
If ICOUNT = 3, do the next line
      If ILLCOND = .FALSE., do block 51


If ICOUNT = 2 and ILLCONDG = .FALSE., then do block 45



                                                  | Start once-per-vector processing
do blocks 46, 98, 99, and 48                      | Get backward-adapted gain
                                                  | GSTATE(1:9) shifted down 1 position
do blocks 19 and 21                               | Scale selected excitation codevector
do block 32
If ICOUNT = 1, do block 85                        | Update short-term postfilter coefficients
do block 81
If ICOUNT = 3, do the next 3 lines
      do block 82                                 | Pitch period extraction
      do block 83                                 | Compute pitch predictor tap
      do block 84                                 | Update long-term postfilter coefficients
do block 71                                       | Long-term postfilter
do block 72                                       | Short-term postfilter
do blocks 73 and 74                               | Calculate sums of absolute values
do block 75                                       | Ratio of sums of absolute values
do block 76                                       | Low-pass filter of scaling factor
do block 77                                       | Gain control of postfilter output


do blocks 93, 94, 96, and 97                      | Update log-gain; note that the
                                                  | 3 delay units in the gain adapter just
                                                  | happen naturally in the looping
                                                  | process and need not be
                                                  | implemented explicitly
GSTATE(1) =  output of block 97                   | Update gain predictor memory
I = (ICOUNT – 1) * IDIM                           | I = starting address of STTMP( )
copy ST(1:5) to STTMP(I + 1:I + 5)                | Update STTMP( )
NLSSTTMP(ICOUNT) = NLSST                          | End of once-per-vector processing


                                                  | Start once-per-frame processing
If ICOUNT = 4, do the next 5 lines
      do block 49                                 | Output ill-condition flag = ILLCOND


      do block 50, order 1 to 10                  | Output predictor coefficients = ATMP( )
                                                  | with NLSATMP
                                                  | Output ill-condition flag = ILLCONDP
      NLSAPF = NLSATMP                            | Save the 10th-order predictor
      copy ATMP(2:11) to APF(2:11)                | for postfilter use later
      continue block 50, order 11 to 50           | Continue to finish block 50
                                                  | Output predictor coefficients = ATMP( )
                                                  | with NLSATMP
                                                  | Output ill-condition flag = ILLCOND


If ICOUNT = 1, do the next 6 lines
      GTMP(1) = GSTATE(4)                         | Update GTMP( ) in one shot
      GTMP(2) = GSTATE(3)
      GTMP(3) = GSTATE(2)
      GTMP(4) = GSTATE(1)
      do block 43                                 | Output ill-condition flag = ILLCONDG
      do block 44                                 | Output predictor coefficients = GPTMP( )
                                                  | Output ill-condition flag = ILLCONDG
                                                  | End of once-per-frame processing
Go to VEC_LOOP
```