

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

G.728

(06/2012)

SERIES G: TRANSMISSION SYSTEMS AND MEDIA,
DIGITAL SYSTEMS AND NETWORKS

Digital terminal equipments – Coding of voice and audio
signals

Coding of speech at 16 kbit/s using low-delay code excited linear prediction

Recommendation ITU-T G.728

ITU-T G-SERIES RECOMMENDATIONS

TRANSMISSION SYSTEMS AND MEDIA, DIGITAL SYSTEMS AND NETWORKS

INTERNATIONAL TELEPHONE CONNECTIONS AND CIRCUITS	G.100–G.199
GENERAL CHARACTERISTICS COMMON TO ALL ANALOGUE CARRIER-TRANSMISSION SYSTEMS	G.200–G.299
INDIVIDUAL CHARACTERISTICS OF INTERNATIONAL CARRIER TELEPHONE SYSTEMS ON METALLIC LINES	G.300–G.399
GENERAL CHARACTERISTICS OF INTERNATIONAL CARRIER TELEPHONE SYSTEMS ON RADIO-RELAY OR SATELLITE LINKS AND INTERCONNECTION WITH METALLIC LINES	G.400–G.449
COORDINATION OF RADIOTELEPHONY AND LINE TELEPHONY	G.450–G.499
TRANSMISSION MEDIA AND OPTICAL SYSTEMS CHARACTERISTICS	G.600–G.699
DIGITAL TERMINAL EQUIPMENTS	G.700–G.799
General	G.700–G.709
Coding of voice and audio signals	G.710–G.729
Principal characteristics of primary multiplex equipment	G.730–G.739
Principal characteristics of second order multiplex equipment	G.740–G.749
Principal characteristics of higher order multiplex equipment	G.750–G.759
Principal characteristics of transcoder and digital multiplication equipment	G.760–G.769
Operations, administration and maintenance features of transmission equipment	G.770–G.779
Principal characteristics of multiplexing equipment for the synchronous digital hierarchy	G.780–G.789
Other terminal equipment	G.790–G.799
DIGITAL NETWORKS	G.800–G.899
DIGITAL SECTIONS AND DIGITAL LINE SYSTEM	G.900–G.999
MULTIMEDIA QUALITY OF SERVICE AND PERFORMANCE – GENERIC AND USER-RELATED ASPECTS	G.1000–G.1999
TRANSMISSION MEDIA CHARACTERISTICS	G.6000–G.6999
DATA OVER TRANSPORT – GENERIC ASPECTS	G.7000–G.7999
PACKET OVER TRANSPORT ASPECTS	G.8000–G.8999
ACCESS NETWORKS	G.9000–G.9999

For further details, please refer to the list of ITU-T Recommendations.

Recommendation ITU-T G.728

Coding of speech at 16 kbit/s using low-delay code excited linear prediction

Summary

Recommendation ITU-T G.728 contains the description of an algorithm for the coding of speech signals at 16 kbit/s using low-delay code excited linear prediction (LD-CELP). The main description is found in the main body of the Recommendation, where the floating-point implementation of LD-CELP is described. Annexes A, B, C and D contain tables of constants used by the LD-CELP algorithm. In Annex E, the sequencing of variable adaptation and its use is given and Annex F lists the abbreviations used in this Recommendation.

Annex G contains the alternative 16 kbit/s (16-bit) fixed-point specification of ITU-T G.728. The purpose of this Annex is to describe in sufficient detail how ITU-T G.728 operating at 16 kbit/s 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 ITU-T G.728 and producing an output signal of equivalent quality, whether that signal is speech or an in-band data signal.

ITU-T G.728 LD-CELP can operate at three bitrates other than 16 kbit/s, as defined in Annexes H and J. These additional operating modes are particularly useful for applications such as digital circuit multiplication equipment (DCME).

Annex H contains the modifications to ITU-T G.728 LD-CELP speech coding algorithm needed to reduce the coding bit rate down to 12.8 and 9.6 kbit/s. These modifications include the modifications to the shape and gain codebooks. Operation at lower rates can assist DCME in handling increased voice traffic while showing a graceful quality degradation.

Annex I is concerned with how to conceal the loss of bitstream information due to frame erasure or packet loss in the communications channel. During normal operation, the decoder performs in a manner identical with the ITU-T G.728 main body or its Annex G when operating at 16 kbit/s, or with Annex H when operating at 12.8 or 9.6 kbit/s. The modification described in Annex I only involves changing the decoder during times when the bitstream is unavailable. It is presumed that this loss of the bitstream is indicated to the decoder by some external means. Annex I is not essential for normal operation of ITU-T G.728.

Annex J defines a 40 kbit/s extension that is optimized for voiceband data (VBD). The algorithmic delay is five samples long (0.625 ms), which is the same as the other LD-CELP modes of operation described in this Recommendation. The 40 kbit/s VBD algorithm in Annex J is intended for VBD signal compression transmission in applications such as DCME. The algorithm allows soft transition to and from the other ITU-T G.728 LD-CELP operating modes, and is also designed to maintain toll-quality speech. Annex J is an implementation-efficient alternative to the 40 kbit/s ADPCM mode (ITU-T G.726) in DCME systems that incorporate LD-CELP (ITU-T G.728).

Appendix I contains information on the implementation verification procedures for LD-CELP operating at 16 kbit/s (both floating- and fixed-point operation). This Appendix describes the digital test sequences and the measurement software to be used for implementation verification of ITU-T G.728. Provision is included for both floating-point implementations, based on the ITU-T G.728 main body, and bit-exact fixed-point implementations, based on Annex G.

Appendix II gives a broad outline of the speech performance of the 16 kbit/s LD-CELP algorithm when interacting with other parts of the network. Some general guidance is also offered on voice-like and non-voice signals. The performance of ITU-T G.728 is compared to that of 32 kbit/s operating mode of the ITU-T G.726 ADPCM) and of ITU-T G.711 (64 kbit/s PCM).



Documents

Document Sharing Hub

Rec. ITU-T G.728 (06/2012)

1

Appendix III provides information on the operation of the algorithm defined in Annex I when frame erasures occur.

This Recommendation includes an electronic attachment containing programs and test sequences for verification of the floating point and fixed point implementations of the ITU-T G.728 LD-CELP algorithm.

History

Edition	Recommendation	Approval	Study Group
1.0	ITU-T G.728	1992-09-01	
1.1	ITU-T G.728 Annex G	1994-11-01	15
1.2	ITU-T G.728 App. I Software	1995-07-01	
1.3	ITU-T G.728 App. II	1995-11-13	
1.4	ITU-T G.728 Annex H	1997-07-11	16
1.5	ITU-T G.728 Annex H	1999-05-27	16
1.6	ITU-T G.728 Annex I	1999-05-27	16
1.7	ITU-T G.728 Annex J	1999-09-30	16
1.8	ITU-T G.728 Annex G (1994) Cor. 1	2000-02-17	16
1.9	ITU-T G.728 (1992) Amd. 1	2006-05-29	16
2.0	ITU-T G.728	2012-06-29	16

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. 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 Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

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

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at <http://www.itu.int/ITU-T/ipr/>.

© ITU 2013

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

Table of Contents

		Page
1	Introduction and references	1
1.1	Introduction	1
1.2	References	1
2	Outline of LD-CELP	1
2.1	LD-CELP encoder	2
2.2	LD-CELP decoder	2
3	LD-CELP (encoder principles).....	4
3.1	Input PCM format conversion	4
3.2	Vector buffer	5
3.3	Adapter for perceptual weighting filter	5
3.4	Perceptual weighting filter	9
3.5	Synthesis filter	9
3.6	VQ target vector computation	9
3.7	Backward synthesis filter adapter.....	9
3.8	Backward vector gain adapter	11
3.9	Codebook search module	13
3.10	Simulated decoder	16
3.11	Synchronization and in-band signalling	16
4	LD-CELP decoder principles	17
4.1	Excitation VQ codebook	17
4.2	Gain scaling unit.....	18
4.3	Synthesis filter	18
4.4	Backward vector gain adapter	18
4.5	Backward synthesis filter adapter.....	18
4.6	Postfilter	18
4.7	Postfilter adapter.....	20
4.8	Output PCM format conversion	23
5	Computational details	23
5.1	Description of basic coder parameters	23
5.2	Description of internal variables	23
5.3	Input PCM format conversion (block 1)	28
5.4	Vector buffer (block 2).....	28
5.5	Adapter for perceptual weighting filter (block 3, Figure 4-a)	28
5.6	Backward synthesis filter adapter (block 23, Figure 5).....	30
5.7	Backward vector gain adapter (block 20, Figure 6)	32
5.8	Perceptual weighting filter	35
5.9	Computation of zero-input response vector	36
5.10	VQ target vector computation	37

	Page
5.11 Codebook search module (block 24)	37
5.12 Simulated decoder (block 8)	40
5.13 Filter memory update for blocks 9 and 10	41
5.14 Decoder (Figure 3)	42
Annex A – Hybrid window functions for various LPC analyses in LD-CELP	51
A.1 Hybrid window for the synthesis filter	51
A.2 Hybrid window for the log-gain predictor	52
A.3 Hybrid window for the perceptual weighting filter	53
Annex B – Excitation shape and gain codebook tables	55
Annex C – Values for bandwidth broadcasting	59
Annex D – Coefficients of the 1 kHz lowpass elliptic filter used in pitch period extraction module (block 82)	61
Annex E – Time scheduling the sequence of computations	62
Annex F – Alphabetical list of abbreviations used in this Recommendation	64
Annex G – 16 kbit/s fixed point specification	65
G.1 Introduction	65
G.2 Algorithmic changes	72
G.3 Pseudo-code for other modules of ITU-T G.728	81
G.4 LD-CELP internal variable representations	117
G.5 Log-gain tables for gain and shape codebook vectors	121
G.6 Integer values of gain codebook related arrays	122
G.7 Encoder and decoder main program pseudo-codes	123
Annex H – Variable bit rate LD-CELP operation mainly for DCME at rates less than 16 kbit/s	126
H.1 Introduction	126
H.2 Principles of operation	126
H.3 Modifications for 12.8 kbit/s operation	127
H.4 Modifications for 9.6 kbit/s operation	131
Annex I – Frame or packet loss concealment for the LD-CELP decoder	137
I.1 Scope	137
I.2 Normative references	137
I.3 Introduction	137
I.4 Principles of operation	138
I.5 Pseudo-code for frame erasure concealment	141
I.6 Additional coder parameters and variables	153
Annex J – Variable bit-rate operation of LD-CELP mainly for voiceband-data applications in DCME	155
J.1 Scope	155
J.2 Normative references	155

	Page
J.3 Overview	155
J.4 Algorithm description.....	156
Appendix I – Programs and test sequences for implementation verification of the algorithm of the ITU-T G.728 16 kbit/s LD-CELP speech coder	186
I.1 General	186
I.2 Test configurations	186
I.3 Verification programs.....	189
I.4 Test sequences	190
I.5 Verification tools distribution.....	192
Appendix II – Speech performance	197
II.1 Introduction	197
II.2 Speech performance	197
II.3 Performance with non-speech signals	198
II.4 Artificial voice signals.....	198
Appendix III – Values used for scaling ETPAST during frame erasures.....	200
Bibliography.....	201

Electronic attachment: Programs and test sequences for verification of the floating point and fixed point implementations of the ITU-T G.728 LD-CELP algorithm

Recommendation ITU-T G.728

Coding of speech at 16 kbit/s using low-delay code excited linear prediction

1 Introduction and references

1.1 Introduction

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

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

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

1.2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

- [ITU-T G.711] Recommendation ITU-T G.711 (1988), *Pulse code modulation (PCM) of voice frequencies*.
- [ITU-T G.726] Recommendation ITU-T G.726 (1990), *40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (ADPCM)*.

2 Outline of LD-CELP

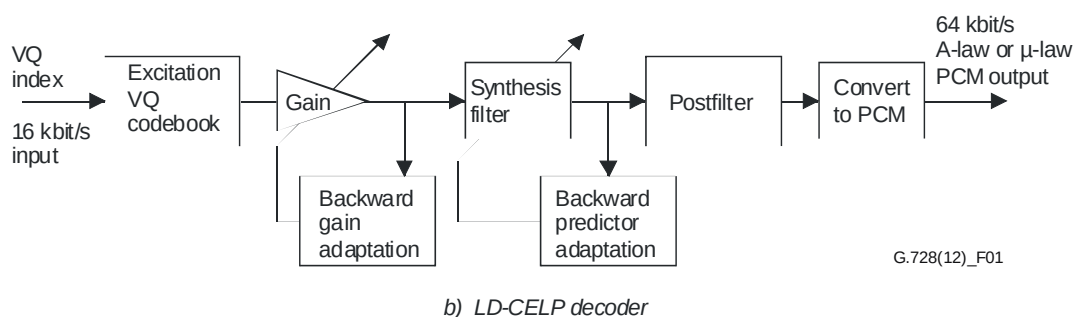
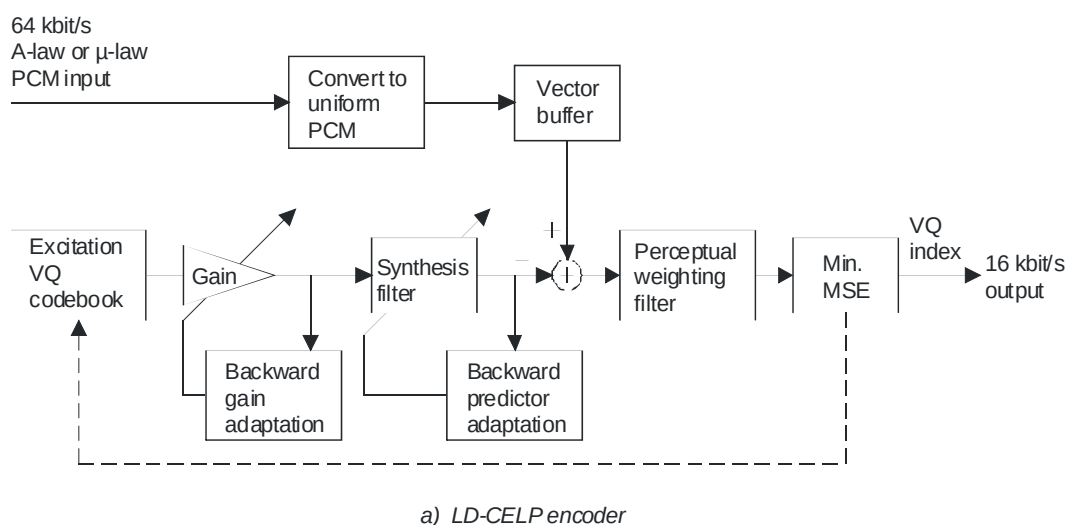
The LD-CELP algorithm consists of an encoder and a decoder described in clauses 2.1 and 2.2 respectively, and illustrated in Figure 1.

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

2.1 LD-CELP encoder

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



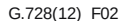
G.728(12)_F01

Figure 1 – Simplified block diagram of LD-CELP coder

2.2 LD-CELP decoder

The decoding operation is also performed on a block-by-block basis. Upon receiving each 10-bit index, the decoder performs a table look-up to extract the corresponding codevector from the excitation codebook. The extracted codevector is then passed through a gain scaling unit and a synthesis filter to produce the current decoded signal vector. The synthesis filter coefficients and the gain are then updated in the same way as in the encoder. The decoded signal vector is then passed through an adaptive postfilter to enhance the perceptual quality. The postfilter coefficients are

0 100 200 300 400 500 600 700 800 900 1000



S

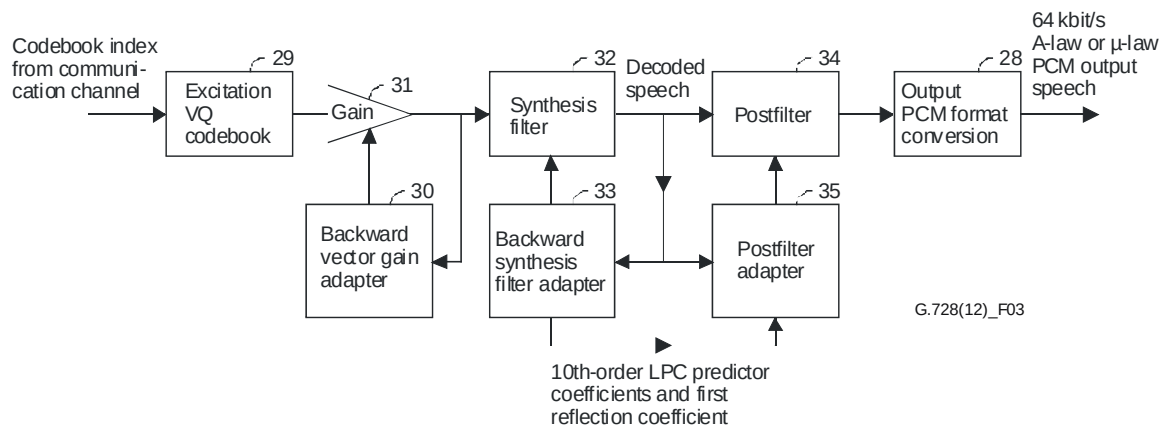


Figure 3 – LD-CELP decoder block schematic

3 LD-CELP (encoder principles)

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

In the following description:

- for each variable to be described, k is the sampling index and samples are taken at 125 μ s intervals;
- a group of five consecutive samples in a given signal is called a *vector* of that signal. For example, five consecutive speech samples form a speech vector, five excitation samples form an excitation vector, and so on;
- we use n to denote the vector index, which is different from the sample index k ;
- four consecutive vectors build one *adaptation cycle*. In later clauses, we also refer to adaptation cycles as *frames*. The two terms are used interchangeably.

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

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

3.1 Input PCM format conversion

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

3.1.1 Internal linear PCM levels

In converting from A-law or μ -law to linear PCM, different internal representations are possible, depending on the device. For example, standard tables for μ -law PCM define a linear range of

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

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

3.2 Vector buffer

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

3.3 Adapter for perceptual weighting filter

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

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

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

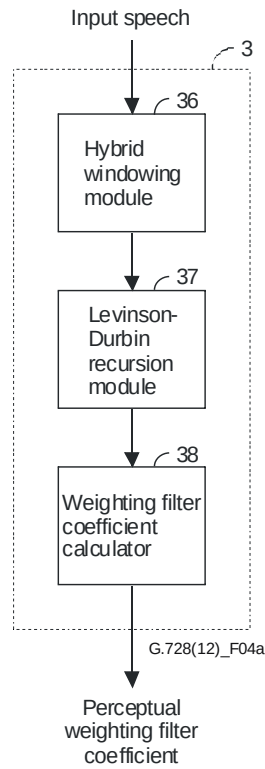


Figure 4-a – Perceptual weighting filter adapter

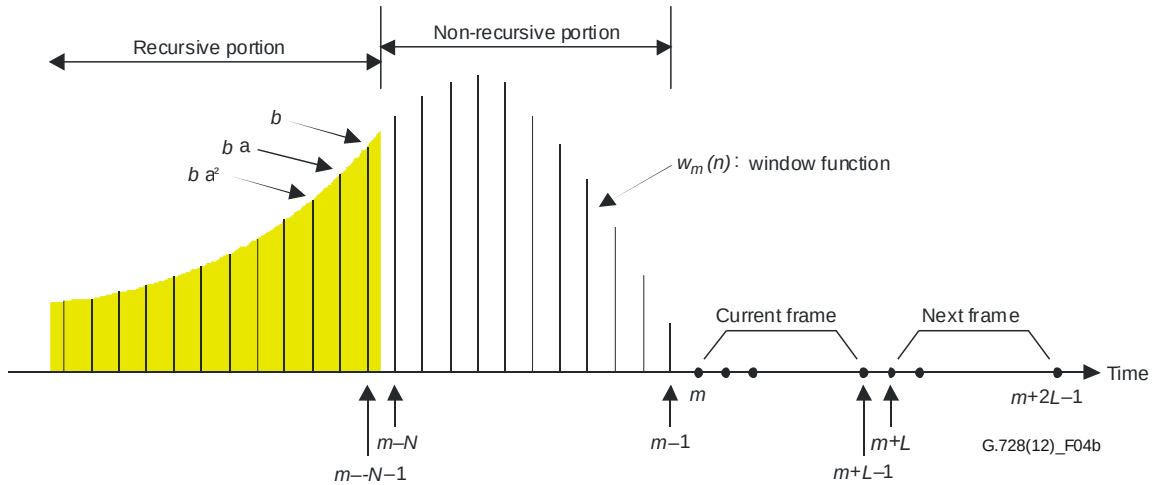


Figure 4-b – Illustration of a hybrid window

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

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

and the window-weighted signal is:

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

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

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

where:

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

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

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

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

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

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

OR

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

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

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

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

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

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

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

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

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

$$E(0) = R(0) \quad (3-2a)$$

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

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

$$a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{(i-1)}, \quad 1 \leq j \leq i-1 \quad (3-2d)$$

$$E(i) = (1 - k_i^2) E(i-1) \quad (3-2e)$$

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

$$q_i = a_i^{(10)}, \quad 1 \leq i \leq 10 \quad (3-2f)$$

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

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

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

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

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

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

and

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

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

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

3.4 Perceptual weighting filter

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

3.4.1 Non-speech operation

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

3.5 Synthesis filter

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

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

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

3.6 VQ target vector computation

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

3.7 Backward synthesis filter adapter

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

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

- a) the input signal is now the quantized speech rather than the unquantized input speech;
- b) the predictor order is 50 rather than 10;
- c) the hybrid window parameters are different: $N = 35$, $\alpha = \left(\frac{3}{4}\right)^{\frac{1}{40}} = 0.992833749$.

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

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

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

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

$$a_i = \lambda^i \hat{a}_i, \quad i = 1, 2, \dots, 50 \quad (3-6)$$

where λ is given by:

$$\lambda = \frac{253}{256} = 0.98828125 \quad (3-7)$$

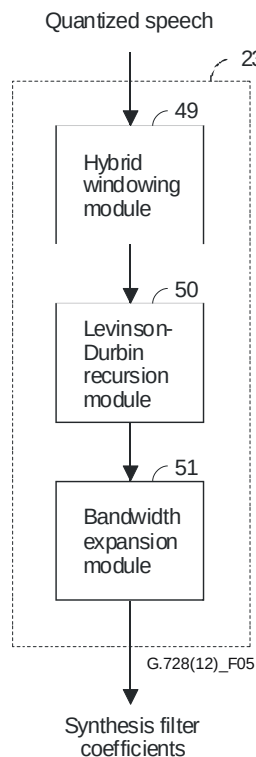


Figure 5 – Backward synthesis filter adapter

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

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

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

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

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

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

Similar to the perceptual weighting filter, the synthesis filters 9 and 22 are also updated once every four vectors, and the updates also occur at the third speech vector of every 4-vector adaptation cycle. However, the updates are based on the quantized speech up to the last vector of the previous adaptation cycle. In other words, a delay of two vectors is introduced before the updates take place. This is because the Levinson-Durbin recursion module 50 and the energy table calculator 15 (described later) are computationally intensive. As a result, even though the autocorrelation of previously quantized speech is available at the first vector of each four vector cycle, computations may require more than one vector worth of time. Therefore, to maintain a basic buffer size of one vector (so as to keep the coding delay low), and to maintain real-time operation, a 2-vector delay in filter updates is introduced in order to facilitate real-time implementation.

3.8 Backward vector gain adapter

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

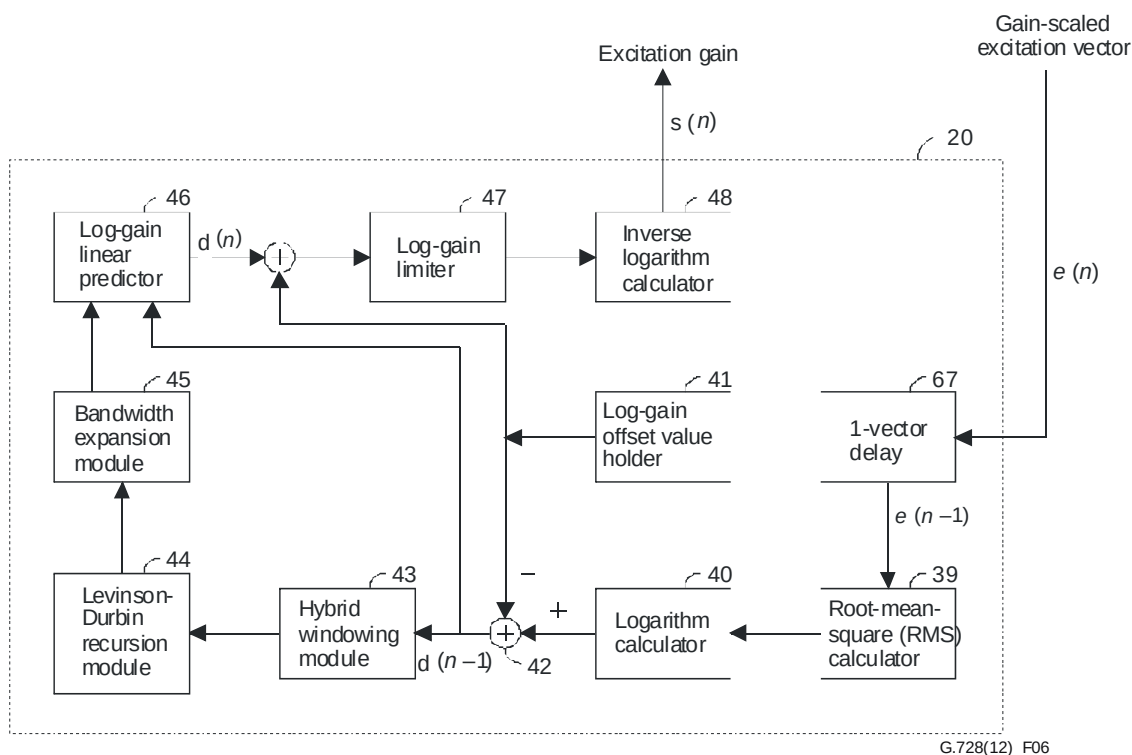


Figure 6 – Backward vector gain adapter

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

In Figure 6, a log-gain offset value of 32 dB is stored in the log-gain offset value holder 41. This value is meant to be roughly equal to the average excitation gain level (in dB) during voiced speech. The adder 42 subtracts this log-gain offset value from the logarithmic gain produced by the logarithm calculator 40. The resulting offset-removed logarithmic gain $\delta(n-1)$ is then used by the hybrid windowing module 43 and the Levinson-Durbin recursion module 44. Again, blocks 43 and 44 operate in exactly the same way as blocks 36 and 37 in the perceptual weighting filter adapter module (Figure 4-a), except that the hybrid window parameters are different and that the signal under analysis is now the offset-removed logarithmic gain rather than the input speech. (Note that only one gain value is produced for every five speech samples.) The hybrid window parameters of block 43 are:

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

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

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

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

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

where the coefficients α_i are computed as:

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

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

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

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

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

from the dB value to the linear domain. The gain limiter ensures that the gain in the linear domain is in between 1 and 1000.

3.9 Codebook search module

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

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

3.9.1 Principle of codebook search

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

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

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

where:

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

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

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

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

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

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

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

where:

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

and

$$E_j = \|\mathbf{H}y_j\|^2 \quad (3-20)$$

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

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

$$b_i = 2g_i \quad (3-21)$$

and

$$c_i = g_i^2 \quad (3-22)$$

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

$$\hat{D} = -b_i P_j + c_i E_j \quad (3-23)$$

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

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

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

- a) The first and the most obvious way is to evaluate the eight possible $\hat{D}$ values corresponding to the eight possible values of i , and then pick the index i which corresponds to the smallest $\hat{D}$. However, this requires two multiplications for each i .
- b) A second way is to compute the optimal gain $\hat{g} = P_j/E_j$ first, and then quantize this gain $\hat{g}$ to one of the eight gain levels $\{g_0, \dots, g_7\}$ in the 3-bit gain codebook. The best index i is the index of the gain level g_i which is closest to $\hat{g}$. However, this approach requires a division operation for each of the 128 shape codevectors, and division is typically very inefficient to implement using DSP processors.
- c) A third approach, which is a slightly modified version of the second approach, is particularly efficient for DSP implementations. The quantization of $\hat{g}$ can be thought of as a series of comparisons between $\hat{g}$ and the "quantizer cell boundaries", which are the mid-points between adjacent gain levels. Let d_i be the mid-point between gain level g_i and g_{i+1} that have the same sign. Then, testing " $\hat{g} < d_i$?" is equivalent to testing " $P_j < d_i E_j$?". Therefore, by using the latter test, we can avoid the division operation and still require only one multiplication for each index i . This is the approach used in the codebook search. The gain quantizer cell boundaries d_i s are fixed and can be precomputed and stored in a table. For the eight gain levels, actually only six boundary values d_0, d_1, d_2, d_4, d_5 , and d_6 are used.

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

3.9.2 Operation of codebook search module

With the codebook search principle introduced, the operation of the codebook search module 24 is now described below. Refer to Figure 2. Every time when the synthesis filter 9 and the perceptual weighting filter 10 are updated, the impulse response vector calculator 12 computes the first five samples of the impulse response of the cascaded filter $F(z)W(z)$. To compute the impulse response vector, we first set the memory of the cascaded filter to zero, then excite the filter with an input sequence $\{1, 0, 0, 0, 0\}$. The corresponding five output samples of the filter are $h(0), h(1), \dots, h(4)$, which constitute the desired impulse response vector. After this impulse response vector is computed, it will be held constant and used in the codebook search for the following four speech vectors, until the filters 9 and 10 are updated again.

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

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

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

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

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

- a) Initialize $\hat{D}_{\min}$ to a number larger than the largest possible value of $\hat{D}$ (or use the largest possible number of the DSPs number representation system).
- b) Set the shape codebook index $j = 0$.
- c) Compute the inner product $P_j = p^T(n)y_j$.
- d) If $p_j < 0$, go to step h) to search through negative gains; otherwise, proceed to step e) to search through positive gains.
- e) If $P_j < d_0 E_j$, set $i = 0$ and go to step k); otherwise proceed to step f).
- f) If $P_j < d_1 E_j$, set $i = 1$ and go to step k); otherwise proceed to step g).
- g) If $P_j < d_2 E_j$, set $i = 2$ and go to step k); otherwise set $i = 3$ and go to step k).
- h) If $P_j > d_4 E_j$, set $i = 4$ and go to step k); otherwise proceed to step i).
- i) If $P_j > d_5 E_j$, set $i = 5$ and go to step k); otherwise proceed to step j).
- j) If $P_j > d_6 E_j$, set $i = 6$; otherwise set $i = 7$.

- k) Compute $\hat{D} = -b_i P_j + c_i E_j$.
- l) If $\hat{D} < \hat{D}_{\min}$, then set $\hat{D}_{\min} = \hat{D}$, $i_{\min} = i$, and $j_{\min} = j$.
- m) If $j < 127$, set $j = j + 1$ and go to step c); otherwise proceed to step n).
- n) When the algorithm proceeds to here, all 1024 possible combinations of gains and shapes have been searched through. The resulting $i_{\min}$ and $j_{\min}$ are the desired channel indices for the gain and the shape, respectively. The output best codebook index (10-bit) is the concatenation of these two indices, and the corresponding best excitation codevector is $y(n) = g_{i_{\min}} y_{j_{\min}}$. The selected 10-bit codebook index is transmitted through the communication channel to the decoder.

3.10 Simulated decoder

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

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

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

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

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

3.11 Synchronization and in-band signalling

In the above description of the encoder, it is assumed that the decoder knows the boundaries of the received 10-bit codebook indices and also knows when the synthesis filter and the log-gain predictor need to be updated (recall that they are updated once every four vectors). In practice, such synchronization information can be made available to the decoder by adding extra synchronization

bits on top of the transmitted 16 kbit/s bitstream. However, in many applications there is a need to insert synchronization or in-band signalling bits as part of the 16 kbit/s bitstream. This can be done in the following way. Suppose a synchronization bit is to be inserted once every N speech vectors; then, for every N -th input speech vector, we can search through only half of the shape codebook and produce a 6-bit shape codebook index. In this way, we rob one bit out of every N -th transmitted codebook index and insert a synchronization or signalling bit instead.

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

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

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

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

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

4 LD-CELP decoder principles

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

4.1 Excitation VQ codebook

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

4.2 Gain scaling unit

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

4.3 Synthesis filter

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

4.4 Backward vector gain adapter

The function of this block is described in clause 3.8.

4.5 Backward synthesis filter adapter

The function of this block is described in clause 3.7.

4.6 Postfilter

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

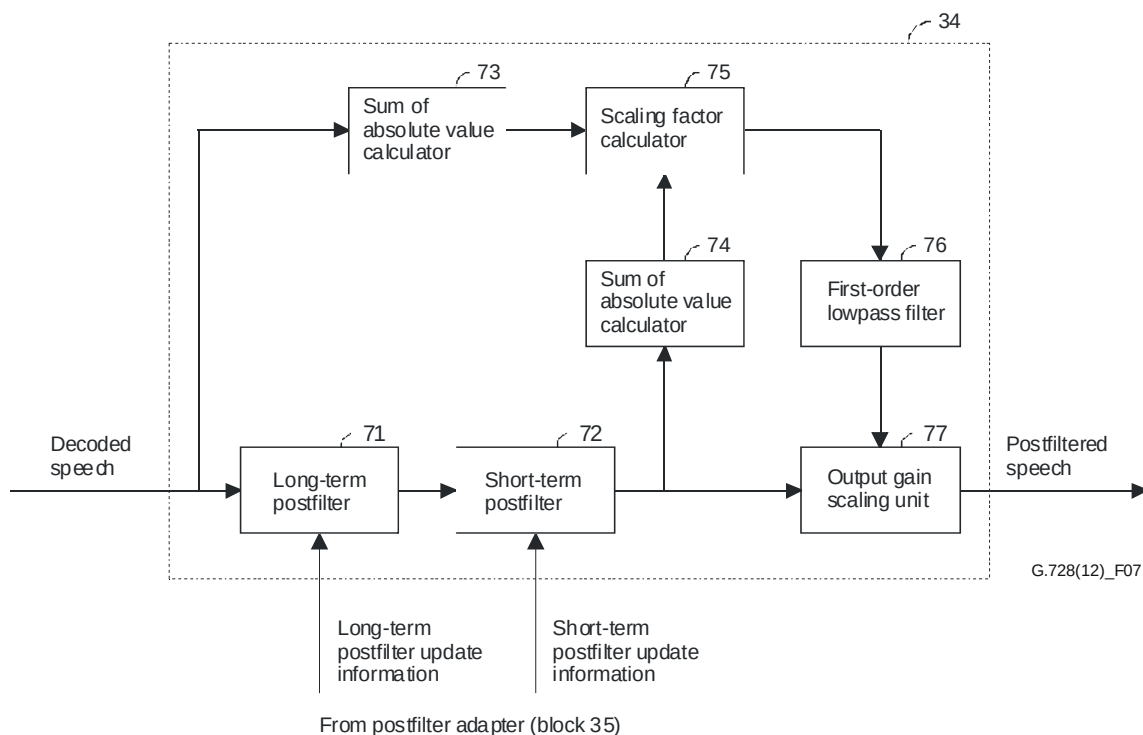


Figure 7 – Postfilter block schematic

The *long-term postfilter* 71, sometimes called the *pitch postfilter*, is a comb filter with its spectral peaks located at multiples of the fundamental frequency (or *pitch frequency*) of the speech to be

postfiltered. The reciprocal of the fundamental frequency is called the *pitch period*. The *pitch period* can be extracted from the decoded speech using a pitch detector (or pitch extractor). Let p be the fundamental pitch period (in samples) obtained by a pitch detector, then the transfer function of the long-term postfilter can be expressed as:

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

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

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

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

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

where:

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

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

and

$$\mu = (0.15) k_1 \quad (4-5)$$

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

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

The sum of absolute value calculator 73 operates vector-by-vector. It takes the current decoded speech vector $s_d(n)$ and calculates the sum of the absolute values of its five vector components. Similarly, the sum of absolute value calculator 74 performs the same type of calculation, but on the current output vector $s_f(n)$ of the short-term postfilter. The scaling factor calculator 75 then divides the output value of block 73 by the output value of block 74 to obtain a scaling factor for the current $s_f(n)$ vector. This scaling factor is then filtered by a first-order lowpass filter 76 to get a separate scaling factor for each of the five components of $s_f(n)$. The first-order lowpass filter 76 has a transfer function of $0.01/(1 - 0.99z^{-1})$. The lowpass filtered scaling factor is used by the output gain scaling unit 77 to perform sample-by-sample scaling of the short-term postfilter output. Note that

since the scaling factor calculator 75 only generates one scaling factor per vector, it would have a staircase effect on the sample-by-sample scaling operation of block 77 if the lowpass filter 76 were not present. The lowpass filter 76 effectively smooths out such a staircase effect.

4.6.1 Non-speech operation

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

4.7 Postfilter adapter

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

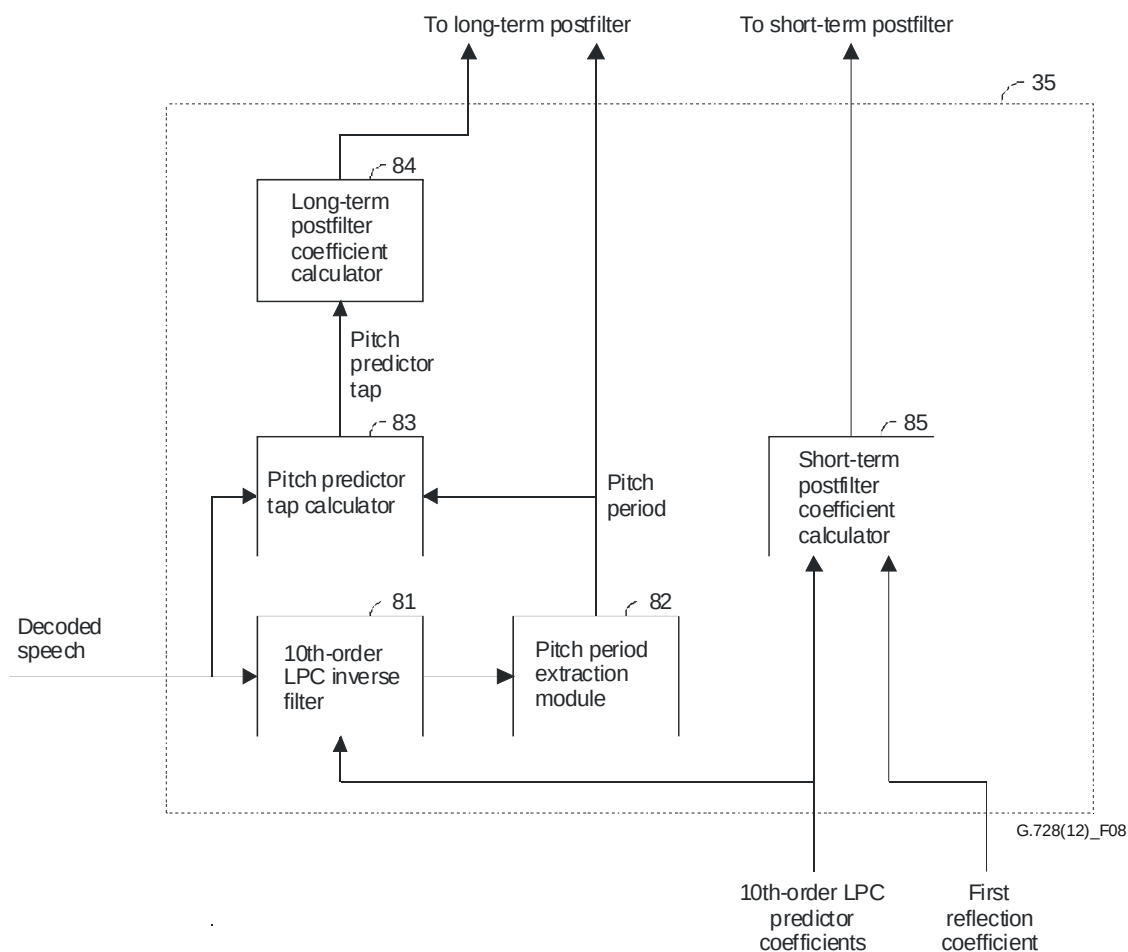


Figure 8 – Postfilter adapter block schematic

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

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

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

where the coefficients $\tilde{a}_i$ are supplied by the Levinson-Durbin recursion module (block 50 of Figure 5) and are updated at the first vector of each frame. This LPC inverse filter takes the decoded speech as its input and produces the LPC prediction residual sequence $\{d(k)\}$ as its output. We use a pitch analysis window size of 100 samples and a range of pitch period from 20 to 140 samples. The pitch period extraction module 82 maintains a long buffer to hold the last 240 samples of the LPC prediction residual. For indexing convenience, the 240 LPC residual samples stored in the buffer are indexed as $d(-139), d(-138), \dots, d(100)$.

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

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

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

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

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

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

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

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

these 13 possible time lags, the time lag p_1 that gives the largest correlation is identified. We then test to see if this new lag p_1 should be used as the output pitch period of the current frame. First, we compute:

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

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

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

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

$$p = \begin{cases} p_0 & \text{if } \beta_1 \leq 0.4 \beta_0 \\ p_1 & \text{if } \beta_1 > 0.4 \beta_0 \end{cases} \quad (4-11)$$

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

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

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

$$b = \begin{cases} 0 & \text{if } \beta < 0.6 \\ 0.15 \beta & \text{if } 0.6 \leq \beta \leq 1 \\ 0.15 & \text{if } \beta > 1 \end{cases} \quad (4-13)$$

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

In general, the closer β is to unity, the more periodic the speech waveform is. As can be seen in equations (4-13) and (4-14), if $\beta < 0.6$, which roughly corresponds to unvoiced or transition regions of speech, then $b = 0$ and $g_l = 1$, and the long-term postfilter transfer function becomes $H_l(z) = 1$, which means the filtering operation of the long-term postfilter is totally disabled. On the other hand, if $0.6 \leq \beta \leq 1$, the long-term postfilter is turned on, and the degree of comb filtering is determined by β . The more periodic the speech waveform, the more comb filtering is performed. Finally, if

$\beta > 1$, then b is limited to 0.15; this is to avoid too much comb filtering. The coefficient g_1 is a scaling factor of the long-term postfilter to ensure that the voiced regions of speech waveforms do not get amplified relative to the unvoiced or transition regions. (If g_1 were held constant at unity, then after the long-term postfiltering, the voiced regions would be amplified by a factor of $1+b$ roughly. This would make some consonants, which correspond to unvoiced and transition regions, sound unclear or too soft.)

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

4.8 Output PCM format conversion

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

5 Computational details

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

5.1 Description of basic coder parameters

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

5.2 Description of internal variables

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

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

Table 1 – Basic coder parameters of LD-CELP

Name	Equivalent symbol	Value	Description
AGCFAC		0.99	AGC adaptation speed controlling factor
FAC	λ	253/256	Bandwidth expansion factor of synthesis filter
FACGP	λ_g	29/32	Bandwidth expansion factor of log-gain predictor

Table 1 – Basic coder parameters of LD-CELP

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

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

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

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

Table 2 – LD-CELP internal processing variables

Name	Array index range	Equivalent symbol	Initial value	Description
A	1 to LPC+1	$-a_{i-1}$	1,0,0,...	Synthesis filter coefficients
AL	1 to 3		Annex D	1 kHz lowpass filter denominator coefficients
AP	1 to 11	$-\bar{a}_{i-1}$	1,0,0,...	Short-term postfilter denominator coefficients
APF	1 to 11	$-\bar{a}_{i-1}$	1,0,0,...	10th-order LPC filter coefficients
ATMP	1 to LPC+1	$-a_{i-1}$		Temporary buffer for synthesis filter coefficients
AWP	1 to LPCW+1		1,0,0,...	Perceptual weighting filter denominator coefficients
AWZ	1 to LPCW+1		1,0,0,...	Perceptual weighting filter numerator coefficients
AWZTMP	1 to LPCW+1		1,0,0,...	Temporary buffer for weighting filter coefficients
AZ	1 to 11	$-\bar{b}_{i-1}$	1,0,0,...	Short-term postfilter numerator coefficients
B	1	b	0	Long-term postfilter coefficients
BL	1 to 4		Annex D	1 kHz lowpass filter numerator coefficients
DEC	-34 to 25	$\bar{d}(n)$	0,0,...,0	4:1 decimated LPC prediction residual
D	-139 to 100	$d(k)$	0,0,...,0	LPC prediction residual
ET	1 to IDIM	$e(n)$	0,0,...,0	Gain-scaled excitation vector
FACV	1 to LPC+1	λ^{i-1}	Annex C	Synthesis filter BW broadening vector
FACGPV	1 to LPCLG+1	λ_g^{i-1}	Annex C	Gain predictor BW broadening vector
G2	1 to NG	b_i	Annex B	Two times gain levels in gain codebook
GAIN	1	$\sigma(n)$		Excitation gain
GB	1 to NG-1	d_i	Annex B	Mid-point between adjacent gain levels
GL	1	g_l	1	Long-term postfilter scaling factor
GP	1 to LPCLG+1	$-\alpha_{i-1}$	1,-1,0,0,...	Log-gain linear predictor coefficients
GPTMP	1 to LPCLG+1	$-\alpha_{i-1}$		Temporary array for log-gain linear predictor coefficients
GQ	1 to NG	g_i	Annex B	Gain levels in the gain codebook
GSQ	1 to NG	c_i	Annex B	Squares of gain levels in gain codebook
GSTATE	1 to LPCLG	$\delta(n)$	-32,-32,...,-32	Memory of the log-gain linear predictor
GTMP	1 to 4		-32,-32,-32,-32	Temporary log-gain buffer
H	1 to IDIM	$h(n)$	1,0,0,0,0	Impulse response vector of $F(z)W(z)$
ICHAN	1			Best codebook index to be transmitted
ICOUNT	1			Speech vector counter (indexed from 1 to 4)
IG	1	i		Best 3-bit gain codebook index

Table 2 – LD-CELP internal processing variables

Name	Array index range	Equivalent symbol	Initial value	Description
IP	1		IPINIT ^{b)}	Address pointer to LPC prediction residual
IS	1	j		Best 7-bit shape codebook index
KP	1	p		Pitch period of the current frame
KP1	1	$\hat{p}$	50	Pitch period of the previous frame
PN	1 to IDIM	$p(n)$		Correlation vector for codebook search
PTAP	1	β		Pitch predictor tap computed by block 83
R	1 to NR+1 ^{a)}			Autocorrelation coefficients
RC	1 to NR ^{a)}			Reflection coefficients, also as a scratch array
RCTMP	1 to LPC			Temporary buffer for reflection coefficients
REXP	1 to LPC+1		0,0,...,0	Recursive part of autocorrelation, synthesis filter
REXPLOG	1 to LPCLG+1		0,0,...,0	Recursive part of autocorrelation, log-gain predictor
REXPW	1 to LPCW+1		0,0,...,0	Recursive part of autocorrelation, weighting filter
RTMP	1 to LPC+1			Temporary buffer for autocorrelation coefficients
S	1 to IDIM	$s(n)$	0,0,...,0	Uniform PCM input speech vector
SB	1 to 105		0,0,...,0	Buffer for previously quantized speech
SBLG	1 to 34		0,0,...,0	Buffer for previous log-gain
SBW	1 to 60		0,0,...,0	Buffer for previous input speech
SCALE	1			Unfiltered postfilter scaling factor
SCALEFIL	1		1	Lowpass filtered postfilter scaling factor
SD	1 to IDIM	$s_d(k)$		Decoded speech buffer
SPF	1 to IDIM			Postfiltered speech vector
SPFPCFV	1 to 11	$SPFPCF^{i-1}$	Annex C	Short-term postfilter pole controlling vector
SPFZCFV	1 to 11	$SPFZCF^{i-1}$	Annex C	Short-term postfilter zero controlling vector
SO	1	$s_o(k)$		A-law or μ -law PCM input speech sample
SU	1	$s_u(k)$		Uniform PCM input speech sample
ST	–239 to IDIM	$s_q(n)$	0,0,...,0	Quantized speech vector
STATELPC	1 to LPC		0,0,...,0	Synthesis filter memory
STLPCI	1 to 10		0,0,...,0	LPC inverse filter memory
STLPLF	1 to 3		0,0,0	1 kHz lowpass filter memory

Table 2 – LD-CELP internal processing variables

Name	Array index range	Equivalent symbol	Initial value	Description
STMP	1 to 4*IDIM	$v(n)$ $\hat{x}(n); x(n)$	0,0,...,0	Buffer for perceptually weighted filter hybrid window
STPFFIR	1 to 10		0,0,...,0	Short-term postfilter memory, all-zero section
STPFIIR	10		0,0,...,0	Short-term postfilter memory, all-pole section
SUMFIL	1			Sum of absolute value of postfiltered speech
SUMUNFIL	1			Sum of absolute value of decoded speech
SW	1 to IDIM			Perceptually weighted speech vector
TARGET	1 to IDIM			VQ target vector (gain-normalized)
TEMP	1 to IDIM	μ		Scratch array for temporary working space
TILTZ	1		0	Short-term postfilter tilt-compensation coefficients
WFIR	1 to LPCW		0,0,...,0	Memory of weighting filter 4, all-zero portion
WIIR	1 to LPCW	$w_m(k)$	0,0,...,0	Memory of weighting filter 4, all-pole portion
WNR	1 to 105		Annex A	Window function for synthesis filter
WNRLG	1 to 34		Annex A	Window function for log-gain predictor
WNRW	1 to 60		Annex A	Window function for weighting filter
WPCFV	1 to LPCW+1	γ_2^{i-1}	Annex C	Perceptual weighting filter pole controlling vector
WS	1 to 105	γ_1^{i-1}		Work space array for intermediate variables
WZCFV	1 to LPCW+1		Annex C	Perceptual weighting filter zero controlling vector
Y	1 to IDIM*NCWD	y_j	Annex B	Shape codebook array
Y2	1 to NCWD	E_j	Energy of y_j	Energy of convolved shape codevector
YN	1 to IDIM	$y(n)$		Quantized excitation vector
ZIRWFIR	1 to LPCW		0,0,...,0	Memory of weighting filter 10, all-zero portion
ZIRWIIR	1 to LPCW		0,0,...,0	Memory of weighting filter 10, all-pole portion
a) NR = Max(LPCW,LPCLG)>IDIM. b) IPINIT = NPWSZ–NFRSZ+IDIM. NOTE – The asterisk (*) denotes arithmetic multiplication.				

5.3 Input PCM format conversion (block 1)

Input: SO

Output: SU

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

Since the operation of this block is completely defined in [ITU-T G.726] or [ITU-T G.711], we will not repeat it here. However, some scaling may be necessary to conform to this description's specification of an input range of $-4\,095$ to $+4\,095$ (see clause 3.1.1).

5.4 Vector buffer (block 2)

Input: SU

Output: S

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

5.5 Adapter for perceptual weighting filter (block 3, Figure 4-a)

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

HYBRID WINDOWING MODULE (block 36)

Input: STMP

Output: R

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

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

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

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
two lines                   | Multiply the window function
WS(N) = SBW(N) * WNRW(K)
K = K + 1
```

```

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

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

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

```

LEVINSON-DURBIN RECURSION MODULE (block 37)

Input: R (output of block 36)

Output: AWZTMP

Function: Convert autocorrelation coefficients to linear predictor coefficients.

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

```

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

If R(1) ≤ 0, go to LABEL
| Skip if zero signal

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

For MINC = 2,3,4,...,LPCW, do the following:
SUM = 0
For IP = 1,2,3,...,MINC, do the next two lines
    N1 = MINC - IP + 2
    SUM = SUM + R(N1) * AWZTMP(IP)

RC(MINC) = -SUM/ALPHA
MH = MINC/2 + 1
| Reflection coefficients
For IP = 2,3,4,...,MH, do the next four lines
    IB = MINC - IP + 2
    AT = AWZTMP(IP) + RC(MINC) * AWZTMP(IB)
    AWZTMP(IB) = AWZTMP(IB) + RC(MINC) AWZTMP(IP)
    AWZTMP(IP) = AT
| Predictor coefficients

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

Repeat the above for the next MINC

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

```

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

WEIGHTING FILTER COEFFICIENT CALCULATOR (block 38)

Input: AWZTMP

Outputs: AWZ, AWP

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

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

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

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

5.6 Backward synthesis filter adapter (block 23, Figure 5)

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

HYBRID WINDOWING MODULE (block 49)

Input: STTMP

Output: RTMP

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

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

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

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

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

```

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

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

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

```

LEVINSON-DURBIN RECURSION MODULE (block 50)

Input: RTMP

Output: ATMP

Function: Convert autocorrelation coefficients to synthesis filter coefficients.

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

```

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

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

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

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

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

```

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

Repeat the above for the next MINC

Exit this program

| Recursion completed normally if
| execution proceeds to here

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

BANDWIDTH EXPANSION MODULE (block 51)

Input: ATMP

Output: A

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

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

<pre> For I = 2,3,...,LPC + 1, do the next line ATMP(I) = FACV(I) * ATMP(I) </pre>	<pre> Scale coefficients </pre>
<pre> Wait until ICOUNT = 3, then For I = 2,3,...,LPC + 1, do the next line A(I) = ATMP(I) </pre>	<pre> Update coefficients at the third vector of each cycle </pre>

5.7 Backward vector gain adapter (block 20, Figure 6)

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

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

Input: ET

Output: ETRMS

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

When these three blocks are executed (which is before the VQ codebook search), the ET array contains the gain-scaled excitation vector determined for the previous speech vector. Therefore, the 1-vector delay unit (block 67) is automatically executed. (It appears in Figure 6 just to enhance clarity.) Since the logarithm calculator immediately follows the RMS calculator, the square root operation in the RMS calculator can be implemented as a "divide-by-two" operation to the output of the logarithm calculator. Hence, the output of the logarithm calculator (the dB value) is $10 * \log_{10}$

(energy of ET/IDIM). To avoid overflow of logarithm value when $ET=0$ (after system initialization or reset), the argument of the logarithm operation is clipped to 1 if it is too small. Also, we note that ETRMS is usually kept in an accumulator, as it is a temporary value which is immediately processed in block 42.

ETRMS = ET(1) * ET(1)		
For K = 2,3,...,IDIM, do the next line		Compute energy of ET
ETRMS = ETRMS + ET(K) * ET(K)		
ETRMS = ETRMS * DIMINV		Divide by IDIM
If ETRMS < 1, set ETRMS = 1		Clip to avoid log overflow
ETRMS = 10 * log ₁₀ (ETRMS)		Compute dB value

LOG-GAIN OFFSET SUBTRACTOR (block 42)

Inputs: ETRMS, GOFF

Output: GSTATE(1)

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

$GSTATE(1) = ETRMS - GOFF$

HYBRID WINDOWING MODULE (block 43)

Input: GTMP

Output: R

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

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

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

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

N1 = LPCLG + NUPDATE		Compute some constants (can be
N2 = LPCLG + NONRLG		precomputed and stored in
N3 = LPCLG + NUPDATE + NONRLG		memory)
For N = 1,2,...,N2, do the next line		Shift the old signal buffer
SBLG(N) = SBLG(N + NUPDATE)		
For N = 1,2,...,NUPDATE, do the next line		Shift in the new signal;
SBLG(N2 + N) = GTMP(N)		SBLG(N3) is the newest sample
K = 1		
For N = N3,N3 - 1,...,3,2,1, do the next two lines		Multiply the window function
WS(N) = SBLG(N) * WNRLG(K)		
K = K + 1		

```

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

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

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

```

LEVINSON-DURBIN RECURSION MODULE (block 44)

Input: R (output of block 43)

Output: GPTMP

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

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

BANDWIDTH EXPANSION MODULE (block 45)

Input: GPTMP

Output: GP

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

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

```

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

```

LOG-GAIN LINEAR PREDICTOR (block 46)

Inputs: GP, GSTATE

Output: GAIN

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

```

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

```

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

Inputs: GAIN, GOFF

Output: GAIN

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

```
GAIN = GAIN + GOFF
```

LOG-GAIN LIMITER (block 47)

Input: GAIN

Output: GAIN

Function: Limit the range of the predicted logarithmic gain.

```

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

```

INVERSE LOGARITHM CALCULATOR (block 48)

Input: GAIN

Output: GAIN

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

```
GAIN = 10(GAIN/20)
```

5.8 Perceptual weighting filter

PERCEPTUAL WEIGHTING FILTER (block 4)

Inputs: S, AWZ, AWP

Output: SW

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

```

For K = 1,2,...,IDIM, do the following:
SW(K) = S(K)
For J = LPCW,LPCW - 1,...,3,2, do the next
two lines
    SW(K) = SW(K) + WFIR(J) * AWZ(J + 1)
    WFIR(J) = WFIR(J - 1)
    SW(K) = SW(K) + WFIR(1) * AWZ(2)
WFIR(1) = S(K)
    ForJ = LPCW,LPCW - 1,...,3,2, do the
next two lines
    SW(K) = SW(K) - WIIR(J) * AWP(J + 1)
    WIIR(J) = WIIR(J - 1)
    SW(K) = SW(K) - WIIR(1) * AWP(2)
WIIR(1) = SW(K)
Repeat the above for the next K

```

| All-zero part of the filter
| Handle last one differently
| All-pole part of the filter
| Handle last one differently

5.9 Computation of zero-input response vector

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

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

Inputs: A, STATELPC

Output: TEMP

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

```

For K = 1,2,...,IDIM, do the following:
TEMP(K) = 0
For J = LPC,LPC - 1,...,3,2, do the next
two lines
    TEMP(K) = TEMP(K) - STATELPC(J) * A(J
+ 1)
    STATELPC(J) = STATELPC(J - 1)
    TEMP(K) = TEMP(K) - STATELPC(1) * A(2)
STATELPC(1) = TEMP(K)
Repeat the above for the next K

```

| Multiply-add
| Memory shift
| Handle last one differently

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

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

Output: ZIR

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

```

For K = 1,2,...,IDIM, do the following:
TMP = TEMP(K)
  For J = LPCW,LPCW - 1,...,3,2, do the next
two lines
    TEMP(K) = TEMP(K) + ZIRWFIR(J) * AWZ(J + 1)
    ZIRWFIR(J) = ZIRWFIR(J - 1)
    TEMP(K) = TEMP(K) + ZIRWFIR(1) * AWZ(2)
ZIRWFIR(1) = TMP
    ForJ = LPCW,LPCW - 1,...,3,2, do the next
two lines
    TEMP(K) = TEMP(K) - ZIRWIIR(J) * AWP(J + 1)
    ZIRWIIR(J) = ZIRWIIR(J - 1)
    ZIR(K) = TEMP(K) - ZIRWIIR(1) * AWP(2)
ZIRWIIR(1) = ZIR(K)
Repeat the above for the next K

```

| All-zero part of the filter
|
| Handle last one differently
| All-pole part of the filter
| Handle last one differently

5.10 VQ target vector computation

VQ TARGET VECTOR COMPUTATION (block 11)

Inputs: SW, ZIR

Output: TARGET

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

NOTE – $ZIR(K) = ZIRWIIR(IDIM + 1 - K)$ from block 10 above. It does not require a separate storage location.

```

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

```

5.11 Codebook search module (block 24)

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

IMPULSE RESPONSE VECTOR CALCULATOR (block 12)

Inputs: A, AWZ, AWP

Output: H

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

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

TEMP(1) = 1		TEMP = synthesis filter memory
RC(1) = 1		RC = $W(z)$ all-pole part memory
For K = 2,3,...,IDIM, do the following:		
A0 = 0		
A1 = 0		
A2 = 0		
For I = K,K - 1,...,3,2, do the next five lines		
TEMP(I) = TEMP(I - 1)		
RC(I) = RC(I - 1)		Filtering
A0 = A0 - A(I) * TEMP(I)		
A1 = A1 + AWZ(I) * TEMP(I)		
A2 = A2 - AWP(I) * RC(I)		
TEMP(1) = A0		
RC(1) = A0 + A1 + A2		
Repeat the above indented section for the next K		
ITMP = IDIM + 1		
For K = 1,2,...,IDIM, do the next line		
H(K) = RC(ITMP - K)		Obtain $h(n)$ by reversing the order of the memory of all-pole section of $W(z)$

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

Inputs: H, Y

Output: Y2

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

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

For J = 1,2,...,NCWD, do the following:		One codevector per loop
J1 = (J - 1) * IDIM		
For K = 1,2,...,IDIM, do the next four lines		
K1 = J1 + K + 1		
TEMP(K) = 0		
For I = 1,2,...,K, do the next line		Convolution
TEMP(K) = TEMP(K) + H(I) * Y(K1 - I)		
Repeat the above 4 lines for the next K		
Y2(J) = 0		
For K = 1,2,...,IDIM, do the next line		
Y2(J) = Y2(J) + TEMP(K) * TEMP(K)		Compute energy
Repeat the above for the next J		

VQ TARGET VECTOR NORMALIZATION (block 16)

Inputs: TARGET, GAIN

Output: TARGET

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

```

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

```

TIME-REVERSED CONVOLUTION MODULE (block 13)

Inputs: H, TARGET (output from block 16)

Output: PN

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

NOTE – The vector PN can be kept in temporary storage.

```

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

```

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

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

Outputs: IG, IS, ICHAN

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

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

```

Initialize DISTM 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          | Compute inner product Pj
    COR = COR + PN(K) * Y(J1 + K)

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

```

```

    If  $COR \leq 0$ , then do the next five lines
     $IDXG = NG$ 
    For  $K = N1 + 1, N1 + 2, \dots, NG - 1$ , do the next
    "if" statement
        If  $COR > GB(K) * Y2(J)$ , do the next two
lines
         $IDXG = K$ 
        GO TO LABEL
LABEL:  $D = -G2(IDXG) * COR + GSQ(IDXG) * Y2(J)$ 
        If  $D < DISTM$ , do the next three lines
         $DISTM = D$ 
         $IG = IDXG$ 
         $IS = J$ 
Repeat the above indented section for the next  $K$ 
 $ICHAN = (IS - 1) * NG + (IG - 1)$ 

```

| Best negative gain found

| Compute distortion $\hat{D}$

| Save the lowest distortion and
the
| best codebook indices so far
|

| Concatenate shape and gain
| codebook indices

Transmit ICHAN through communication channel

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

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

5.12 Simulated decoder (block 8)

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

EXCITATION VQ CODEBOOK (block 19)

Inputs: IG, IS

Output: YN

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

```

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

```

GAIN SCALING UNIT (block 21)

Inputs: GAIN, YN

Output: ET

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

```

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

```

SYNTHESIS FILTER (block 22)

Inputs: ET, A

Output: ST

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

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

5.13 Filter memory update for blocks 9 and 10

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

FILTER MEMORY UPDATE (blocks 9 and 10)

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

Outputs: ST, STATELPC, ZIRWFIR, ZIRWIIR

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

```
ZIRWFIR(1) = ET(1)
TEMP(1) = ET(1)
For K = 2,3,...,IDIM, do the following:
A0 = ET(K)
A1 = 0
A2 = 0
For I = K,K - 1,...,2, do the next five
lines
    ZIRWFIR(I) = ZIRWFIR(I - 1)
    TEMP(I) = TEMP(I - 1)
A0 = A0 - A(I) * ZIRWFIR(I)
A1 = A1 + AWZ(I) * ZIRWFIR(I)
A2 = A2 - AWP(I) * TEMP(I)

ZIRWFIR(1) = A0
TEMP(1) = A0 + A1 + A2

Repeat the above indented section for the
next K
```

| ZIRWFIR now a scratch array

| Compute zero-state responses at
| various stages of the cascaded
| filter

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

For K = 1,2,...,IDIM, do the next four lines	
STATELPC(K) = STATELPC(K) + ZIRWFIR(K)	Limit the range
If STATELPC(K) > MAX, set STATELPC(K) = MAX	
If STATELPC(K) < MIN, set STATELPC(K) = MIN	
ZIRWIIR(K) = ZIRWIIR(K) + TEMP(K)	
For I = 1,2,...,LPCW, do the next line	Now set ZIRWFIR to the right
ZIRWFIR(I) = STATELPC(I)	value
I = IDIM + 1	
For K = 1,2,...,IDIM, do the next line	Obtain quantized speech by
ST(K) = STATELPC(I - K)	reversing
	order of synthesis filter memory

5.14 Decoder (Figure 3)

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

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

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

DECODER BACKWARD SYNTHESIS FILTER ADAPTER (block 33)

Input: ST

Output: A

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

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

DECODER BACKWARD VECTOR GAIN ADAPTER (block 30)

Input: ET

Output: GAIN

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

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

DECODER EXCITATION VQ CODEBOOK (block 29)

Input: ICHAN

Output: YN