ETSI/TC SMG
Released by : ETSI/PT 12
Release date: February 1992

RELEASE NOTE

Recommendation GSM 06.32

Voice Activity Detection

Previously distributed version : 3.0.0 (Release 1/90)
New Released version February 92 : 3.0.0 (Release 92, Phase 1)

1. Reason for changes

No changes since the previously distributed version.

ETSI/GSM

Version 3.0.0

Recommendation: GSM 06.32

Title: Voice Activity Detection

Date: February, 1992

List of contents:

Language of original: English

Number of pages: 37

Detailed list of contents
****

0. SCOPE

This recommendation specifies the voice activity detector (VAD) to be used in the Discontinuous Transmission (DTX) as described in Recommendation GSM 06.31. It also specifies the test methods to be used to verify that a VAD complies with the recommendation.

The requirements are mandatory on any VAD to be used either in the GSM Mobile Stations or Base Station Systems.

1. GENERAL

The function of the VAD is to indicate whether each 20ms frame produced by the speech encoder contains speech or not. The output is a binary flag which is used by the TX DTX handler defined in recommendation GSM 06.31.

The recommendation is organised as follows:

Section 2 describes the principles of operation of the VAD.

In section 3, the computational details necessary for the fixed point implementation of the VAD algorithm are given. This section uses the same notation as used for computational details in GSM 06.10.

The verification of the VAD is based on the use of digital test sequences. Section 4 defines the input and output signals and the test configuration, whereas the detailed description of the test sequences is contained in annex 2.

The performance of the VAD algorithm is characterised by the amount of audible speech clipping it introduces and the percentage activity it indicates. These characteristics for the VAD defined in this recommendation have been established by extensive testing under a wide range of operating conditions. The results are summarised in annex 3.

2. FUNCTIONAL DESCRIPTION

The purpose of this section is to give the reader an understanding of the principles of operation of the VAD, whereas the detailed description is given in section 3. In case of discrepancy between the two descriptions, the detailed description of section 3 shall prevail.

In the following subsections of section 2, a Pascal programming type of notation has been used to describe the algorithm.

2.1. OVERVIEW AND PRINCIPLES OF OPERATION

The function of the VAD is to distinguish between noise with speech present and noise without speech present. The biggest difficulty for detecting speech in a mobile environment is the very low speech/noise ratios which are often encountered. The accuracy of the VAD is improved by using filtering to increase the speech/noise ratio before the decision is made.

For a mobile environment, the worst speech/noise ratios are encountered in moving vehicles. It has been found that the noise is relatively stationary for quite long periods in a mobile environment. It is therefore possible to use an adaptive filter with coefficients obtained during noise, to remove much of the vehicle noise.

The VAD is basically an energy detector. The energy of the filtered signal is compared with a threshold; speech is indicated whenever the threshold is exceeded.

The noise encountered in mobile environments may be constantly changing in level. The spectrum of the noise can also change, and varies greatly over different vehicles. Because of these changes the VAD threshold and adaptive filter coefficients must be constantly adapted. To give reliable detection the threshold must be sufficiently above the noise level to avoid noise being identified as speech but not so far above it that low level parts of speech are identified as noise. The threshold and the adaptive filter coefficients are only updated when speech is not present. It is, of course, potentially dangerous for a VAD to update these values on the basis of its own decision. This adaptation therefore only occurs when the signal seems stationary in the frequency domain but does not have the pitch component inherent in voiced speech and information tones.

A further mechanism is used to ensure that low level noise (which is often not stationary over long periods) is not detected as speech. Here, an additional fixed threshold is used.

A VAD hangover period is used to eliminate mid-burst clipping of low level speech. Hangover is only added to speech-bursts which exceed a certain duration to avoid extending noise spikes.

2.2. ALGORITHM DESCRIPTION

The block diagram of the VAD algorithm is shown in Figure 2/1. The individual blocks are described in the following sections. ACF and N are calculated in the speech encoder.

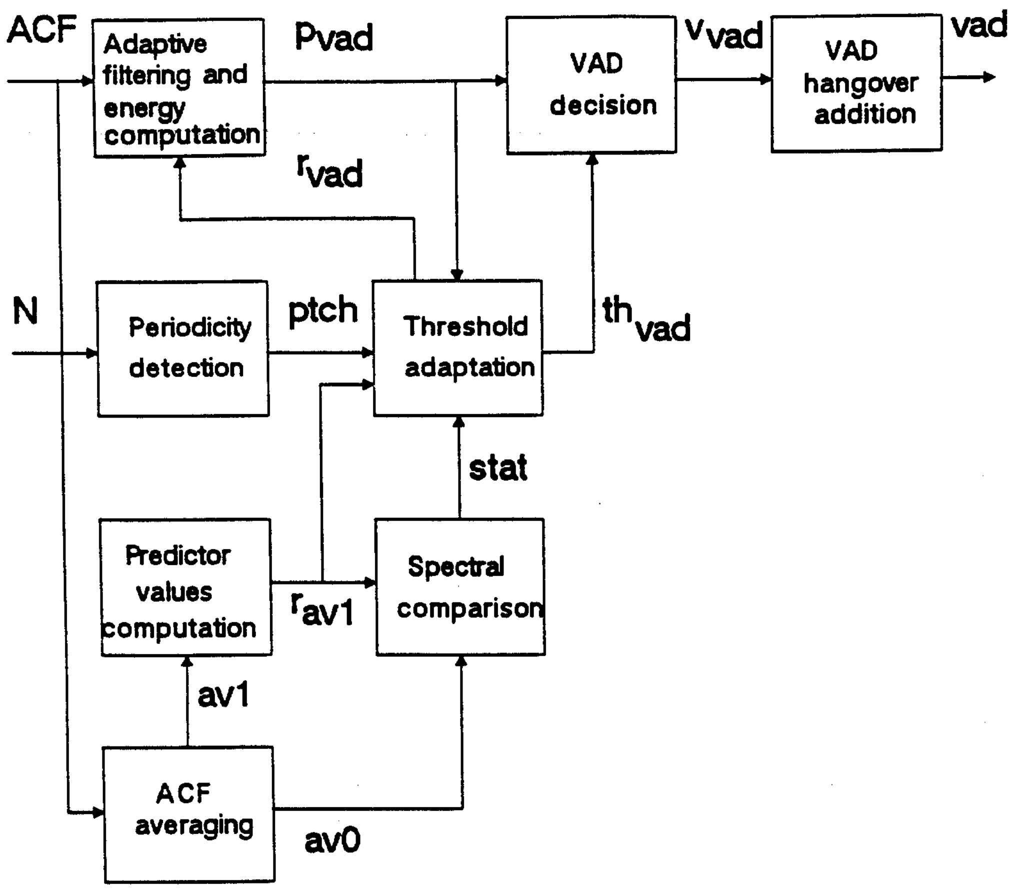

Fig 2 – 1. Functional block diagram of the VAD

The global variables shown in the block diagram are described as follows:

- ACF are autocorrelation coefficients which are calculated in the speech encoder defined in recommendation GSM 06.10 (section 3.1.4, see also Annex 1). The inputs to the speech encoder are 16 bit 2's complement numbers, as described in rec. GSM 06.10, section 4.2.0.

- av0 and av1 are averaged ACF vectors.

- rav1 are autocorrelated predictor values obtained from av1.

- rvad are the autocorrelated predictor values of the adaptive filter.

- N is the long term predictor lag value which is obtained every subsegment in the speech coder defined in recommendation GSM 06.10.
- ptch indicates whether the signal has a steady periodic component.
- pvad is the energy in the current frame of the input signal after filtering.
- thvad is an adaptive threshold.
- stat indicates spectral stationarity.
- vvad indicates the VAD decision before hangover is added.
- vad is the final VAD decision with hangover included.

2.2.1. Adaptive filtering and energy computation

Pvad is computed as follows:

$$\text{Pvad} := \text{rvad}[0]\ \text{ACF}[0] + 2\sum_{i=1}^{8} \text{rvad}[i]\ \text{ACF}[i]$$

This corresponds to performing an 8th order block filtering on the input samples to the speech encoder, after zero offset compensation and pre-emphasis. This is explained in annex 1.

2.2.2. ACF averaging

Spectral characteristics of the input signal have to be obtained using blocks that are larger than one 20ms frame. This is done by averaging the autocorrelation values for several consecutive frames. This averaging is given by the following equations:

$$\text{av0}\{n\}[i] := \sum_{j=0}^{\text{frames}-1} \text{ACF}\{n-j\}[i] \qquad ;\ i = 0..8$$

$$\text{av1}\{n\}[i] := \text{av0}\{n-\text{frames}\}[i] \qquad ;\ i = 0..8$$

Where n represents the current frame, n-1 represents the previous frame etc. The values of constants are given in table 2-1.

Constant	Value	Variable	Initial value
frames	4	previous ACF's av0 & av1	All set to 0

Table 2-1. Constants and variables for ACF averaging

2.2.3. Predictor values computation

The filter predictor values aav1 are obtained from the autocorrelation values av1 according to the equation:

$$\underline{a} := \underline{R}^{-1}\,\underline{p}$$

where:

$$\underline{R} := \begin{bmatrix} av1[0], & av1[1], & av1[2], & av1[3], & av1[4], & av1[5], & av1[6], & av1[7] \\ av1[1], & av1[0], & av1[1], & av1[2], & av1[3], & av1[4], & av1[5], & av1[6] \\ av1[2], & av1[1], & av1[0], & av1[1], & av1[2], & av1[3], & av1[4], & av1[5] \\ av1[3], & av1[2], & av1[1], & av1[0], & av1[1], & av1[2], & av1[3], & av1[4] \\ av1[4], & av1[3], & av1[2], & av1[1], & av1[0], & av1[1], & av1[2], & av1[3] \\ av1[5], & av1[4], & av1[3], & av1[2], & av1[1], & av1[0], & av1[1], & av1[2] \\ av1[6], & av1[5], & av1[4], & av1[3], & av1[2], & av1[1], & av1[0], & av1[1] \\ av1[7], & av1[6], & av1[5], & av1[4], & av1[3], & av1[2], & av1[1], & av1[0] \end{bmatrix}$$

and:

$$\underline{p} := \begin{bmatrix} av1[1] \\ av1[2] \\ av1[3] \\ av1[4] \\ av1[5] \\ av1[6] \\ av1[7] \\ av1[8] \end{bmatrix} \qquad \underline{a} := \begin{bmatrix} aav1[1] \\ aav1[2] \\ aav1[3] \\ aav1[4] \\ aav1[5] \\ aav1[6] \\ aav1[7] \\ aav1[8] \end{bmatrix}$$

aav1[0] := -1

av1 is used in preference to av0 as av0 may contain speech.

The autocorrelated predictor values ravl are then obtained:

$$\mathrm{ravl}[i] := \sum_{k=0}^{8-i} \mathrm{aavl}[k]\ \mathrm{aavl}[k+i] \qquad ;\ i = 0..8$$

2.2.4. Spectral comparison

The spectra represented by the autocorrelated predictor values ravl and the averaged autocorrelation values av0 are compared using the distortion measure dm defined below. This measure is used to produce a boolean value stat every 20ms, as given by these equations:

$$\mathrm{dm} := \left(\mathrm{ravl}[0]\mathrm{av0}[0] + 2\sum_{i=1}^{8} \mathrm{ravl}[i]\mathrm{av0}[i] \right) / \mathrm{av0}[0]$$

$$\mathrm{difference} := |\mathrm{dm} - \mathrm{lastdm}|$$

$$\mathrm{lastdm} := \mathrm{dm}$$

$$\mathrm{stat} := \mathrm{difference} < \mathrm{thresh}$$

The values of constants and initial values are given in table 2-2.

Constant:	Value:	Variable:	Initial value:
thresh	0.05	lastdm	0

Table 2-2. Constants and variables for spectral comparison

2.2.5. Periodicity detection

The frequency spectrum of mobile noise is relatively stationary over quite long periods. The Inverse Filter Autocorrelated Predictor coefficients of the adaptive filter rvad are only updated when this stationarity is detected. Vowel sounds and Information tones however, also have this stationarity, but can be excluded by detecting the periodicity of these sounds using the long term predictor lag values (Nj) which are obtained every subsegment from the speech codec defined in rec. GSM 06.10. Consecutive lag values are compared. Cases in which one lag value is a factor of the other are catered for, however cases in which both lag values have a common factor, are not. This case is not important for speech input but this method of periodicity detection may fail for some sine waves. The boolean variable ptch is updated every 20ms and is true when periodicity is detected. It is calculated according to the following equation:

```
ptch := oldlagcount + veryoldlagcount >= nthresh
```

The following operations are done after the VAD decision and when the current LTP lag values (N0 .. N3) are available, this reduces the delay of the VAD decision. (N{-1} = N3 of previous segment.)

```
lagcount := 0

for j := 0 to 3 do
begin
  smallag := maximum(Nj,N{j-1}) mod minimum(Nj,N{j-1})
  if minimum(smallag,minimum(Nj,N{j-1})-smallag) < lthresh
    then increment(lagcount)
end

veryoldlagcount := oldlagcount

oldlagcount := lagcount
```

The values of constants and initial values are given in table 2-3.

Constant:	Value:	Variable:	Initial value:
lthresh	2	oldlagcount	0
nthresh	4	veryoldlagcount	0
		N3	40

Table 2-3. Constants and variables for periodicity detection

2.2.6. Threshold adaptation

A check is made every 20ms to determine whether the VAD decision threshold (thvad) should be changed. This adaptation is carried out according to the flowchart shown in fig 2-2. The constants used are given in table 2-4.

Adaptation takes place in two different situations: firstly whenever ACF[0] is very low and secondly whenever there is a very high probability that speech is not present.

In the first case, the threshold is adapted if the energy of the input signal is less than pth. The threshold is set to plev without carrying out any further tests because at these very low levels the effect of the signal quantization makes it impossible to obtain reliable results from these tests.

In the second case, the decision threshold (thvad) and the adaptive filter coefficients (rvad) are only updated with the ravl values when the signal is stationary and has no periodic component. In this situation there is a very high probability that speech is not present. The stationarity is detected in the frequency domain, by calculating the spectral difference using consecutive averaged ACF values. If this spectral difference changes very little over a certain number of frames (adp), and the signal does not have a periodic component inherent in voiced speech and information tones, then adaptation occurs.

The step-size by which the threshold is adapted is not constant but a proportion of the current value (determined by constants dec and inc). The adaptation begins by experimentally multiplying the threshold by a factor of (1-1/dec). If the new threshold is now higher than or equal to Pvad times fac then the threshold needed to be decreased and it is left at this new lower level. If, on the other hand, the new threshold level is less than Pvad times fac then the threshold either needed to be increased or kept constant. In this case it is set to Pvad times fac unless this would mean multiplying it by more than a factor of (1+1/inc) (in which case it is multiplied by a factor of (1+1/inc)). The threshold is never allowed to be greater than Pvad+margin.

Constant:	Value:	Variable:	Initial value:
pth	300000	adaptcount	0
plev	800000	thvad	1000000
fac	3.0	rvad[0]	6
adp	8	rvad[1]	-4
inc	16	rvad[2]	1
dec	32	rvad[3] to	
margin	80000000	rvad[8]	All 0

Table 2-4. Constants and variables for threshold adaptation

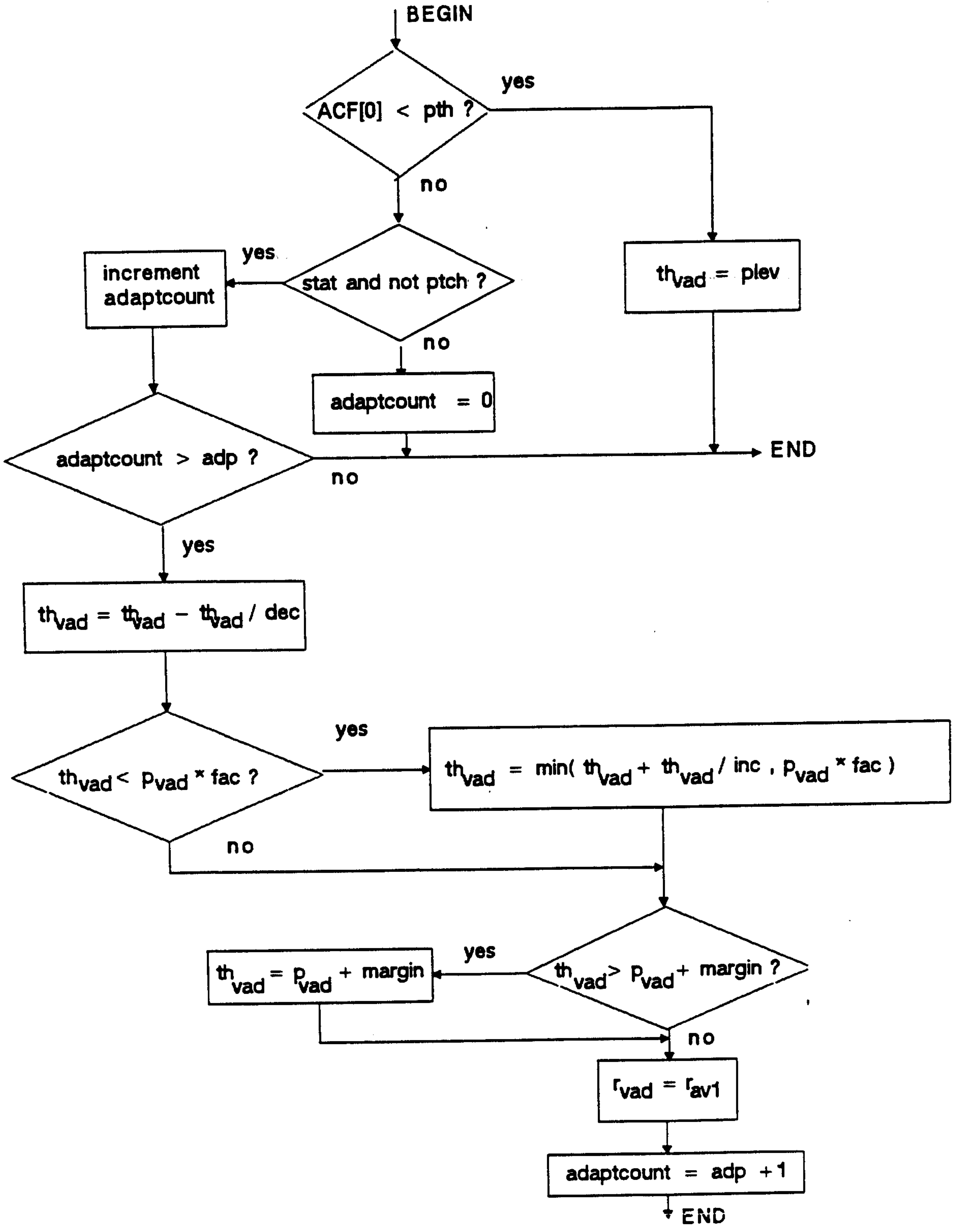
BEGIN
ACF[0] < pth ?
yes
no
th_{vad} = plev
stat and not ptch ?
yes
increment adaptcount
no
adaptcount = 0
adaptcount > adp ?
no
END
yes
$th_{vad} = th_{vad} - th_{vad}$ / dec
$th_{vad} < p_{vad} * fac$?
yes
th_{vad} = min($th_{vad} + th_{vad}$ / inc , $p_{vad} * fac$)
no
$th_{vad} > p_{vad}$ + margin ?
yes
$th_{vad} = p_{vad}$ + margin
no
$r_{vad} = r_{av1}$
adaptcount = adp + 1
END

Fig 2 – 2. Flow diagram for threshold adaptation

2.2.7. VAD decision

Prior to hangover the VAD decision condition is:

```
  vvad := pvad > thvad
```

2.2.8. VAD hangover addition

VAD hangover is only added to bursts of speech greater than or equal to burstconst blocks. The boolean variable vad indicates the decision of the VAD with hangover included. The values of the constants are given in table 2-5. The hangover algorithm is as follows:

```
  if vvad then increment(burstcount) else burstcount := 0

  if burstcount >= burstconst then
  begin
    hangcount := hangconst;
    burstcount := burstconst
  end

  vad := vvad or (hangcount >= 0)

  if hangcount >= 0 then decrement(hangcount)
```

Constant:	Value:	Variable:	Initial value:
burstconst	3	burstcount	0
hangconst	5	hangcount	-1

Table 2-5. Constants and variables for VAD hangover addition

3. COMPUTATIONAL DETAILS

In the next paragraphs, the detailed description of the VAD algorithm follows the preceeding high level description. This detailed description is divided in nine sections related to the blocks of figure 2-1 (except the last one) in the high level description of the VAD algorithm.

Those sections are:

1. Adaptive filtering and energy computation
2. ACF averaging
3. Predictor values computation
4. Spectral comparison
5. Periodicity detection
6. Threshold adaptation
7. VAD decision
8. VAD hangover addition
9. Periodicity updating

The VAD algorithm takes as input the following variables of the RPE-LTP encoder (see the detailed description of the RPE-LTP encoder rec GSM 06.10):

- L_ACF[0..8], autocorrelation function (rec GSM 06.10/4.2.4);
- scalauto, scaling factor to compute the L_ACF[0..8] (rec GSM 06.10/4.2.4);
- Nc, LTP lag (one for each sub-segment, rec GSM 06.10/4.2.11);

So four Nc values are needed for the VAD algorithm.

The VAD computation can start as soon as the L_ACF[0..8] and scalauto variables are known. This means that the VAD computation can take place after part 4.2.4 of rec GSM 06.10 (Autocorrelation) of the LPC analysis section of the RPE-LTP encoder. This scheme will reduce the delay to yield the VAD information. The periodicity updating which is included in section 2.2.5, is done after the processing of the current speech encoder frame.

All the arithmetic operations and names of the variables follow the RPE-LTP detailed description. To increase the precision within the fixed point implementation, a pseudo-floating point representation of some variables is used. This stands for the following variables (and related constants) of the VAD algorithm:

pvad:	Energy of filtered signal
thvad:	Threshold of the VAD decision.
acf0:	Energy of input signal.

For the representation of these variables, two integers (16 bits) are needed:

- one for the exponent (e_pvad, e_thvad, e_acf0)
- one for the mantissa (m_pvad, m_thvad, m_acf0).

The value e_pvad represents the lowest power of 2 just greater or equal to the actual value of pvad and the m_pvad value represents a integer which is always greater or equal to 16384 (normalized mantissa). It means that the pvad value is equal to:

$$pvad = 2^{(e\_pvad)} \times (m\_pvad/32768).$$

This scheme guarantees a large dynamic range for the pvad value and always keeps a precision of 16 bits. All the comparisons are easy to make by comparing the exponents of two variables and the VAD algorithm needs only one pseudo-floating point addition. All the computations related to the pseudo-floating point variables require very simple 16 or 32 bits arithmetic operations defined in the detailed description of the RPE-LTP encoder. This pseudo-floating point arithmetic is only used in section 3.1 and 3.6.

Table 3-1 gives a list of all the variables of the VAD algorithm that must be initialized in the reset procedure and kept in memory for processing the subsequent frame of the RPE- LTP encoder. The types (16 or 32 bits) and initial values of all these variables are clearly indicated and their related sub-section is also mentionned. The bit exact implementation uses other temporary variables that are introduced in the detailed description whenever it is needed.

Names of variables:	type (# of bits):	Initialization:	Sub-section:
Adaptive filter coefficients:			
rvad[0]	16	24576	3.1, 3.6
rvad[1]	16	-16384	3.1, 3.6
rvad[2]	16	4096	3.1, 3.6
rvad[3..8]	16	0	3.1, 3.6
Scaling factor of ravd[0..8]:			
normrvad	16	7	3.1, 3.6
Delay line of the autocorrelation coefficients:			
L_sacf[0..26]	32	0	3.2
L_sav0[0..35]	32	0	3.2
Pointers on the delay lines:			
pt_sacf	16	0	3.2
pt_sav0	16	0	3.2
Distance measure:			
L_lastdm	32	0	3.4
Periodicity counters:			
oldlagcount	16	0	3.5, 3.9
veryoldlagcount	16	0	3.5, 3.9
Adaptive threshold:			
e_thvad (exponent)	16	20	3.6
m_thvad (mantissa)	16	31250	3.6
Counter for adaptation:			
adaptcount	16	0	3.6
Hangover flags:			
burstcount	16	0	3.8
hangcount	16	-1	3.8
LTP lag memory:			
oldlag	16	40	3.9

Table 3-1. Initial values for variables to be stored in memory

3.1. ADAPTIVE FILTERING AND ENERGY COMPUTATION

This section computes the e_pvad and m_pvad variables which represent the pvad value. It needs the L_ACF[0..8] and scalauto variables of the RPE-LTP algorithm and the rvad[0..8] and normrvad variables produced by section 3.6 of the VAD algorithm. It also computes a floating point representation of L_ACF[0] (e_acf0 and m_acf0) used in section 3.6.

Test if L_ACF[0] is equal to 0:

```
IF ( scalauto < 0 ) THEN scalvad = 0;
ELSE scalvad = scalauto;  / keep scalvad for use in section 3.2 /

IF ( L_ACF[0] == 0 ) THEN
                           | e_pvad = -32768;
                           | m_pvad = 0;
                           | e_acf0 = -32768;
                           | m_acf0 = 0;
                           | EXIT  /continue with section 3.2/
```

Re-normalization of the L_acf[0..8]:

```
normacf = norm( L_ACF[0] );

| FOR i = 0 to 8:
|  sacf[i] = ( L_ACF[i] << normacf ) >> 19;
| NEXT i:
```

Computation of e_acf0 and m_acf0:

```
e_acf0 = add( 32, (scalvad << 1 ) );
e_acf0 = sub( e_acf0, normacf);
m_acf0 = sacf[0] << 3;
```

Computation of e_pvad and m_pvad:

```
e_pvad = add( e_acf0, 14 );
e_pvad = sub( e_pvad, normrvad );

L_temp = 0;

| FOR i = 1 to 8:
|  L_temp = L_add( L_temp, L_mult( sacf[i], rvad[i] ) );
| NEXT i:

L_temp = L_add( L_temp, L_mult( sacf[0], rvad[0] ) >> 1 );
```

```
IF ( L_temp <= 0 ) THEN L_temp = 1;

normprod = norm( L_temp );
e_pvad = sub( e_pvad, normprod );
m_pvad = ( L_temp << normprod ) >> 16;
```

3.2. ACF AVERAGING

This section uses the L_ACF[0..8] and the scalvad variables to compute the array L_av0[0..8] and L_av1[0..8] used in section 3.3 and 3.4.

<u>Computation of the scaling factor</u>:

```
scal = sub( 10, (scalvad << 1) );
```

<u>Computation of the arrays L_av0[0..8] and L_av1[0..8]</u>:

```
| FOR i = 0 to 8:
|  L_temp = L_ACF[i] >> scal;
|  L_av0[i] = L_add( L_sacf[i], L_temp );
|  L_av0[i] = L_add( L_sacf[i+9], L_av0[i] );
|  L_av0[i] = L_add( L_sacf[i+18], L_av0[i] );
|  L_sacf[ pt_sacf + i ] = L_temp;
|  L_av1[i] = L_sav0[ pt_sav0 + i ];
|  L_sav0[ pt_sav0 + i] = L_av0[i];
| NEXT i:
```

<u>Update of the array pointers</u>:

```
IF ( pt_sacf == 18 ) THEN pt_sacf = 0;
ELSE pt_sacf = add( pt_sacf, 9);

IF ( pt_sav0 == 27 ) THEN pt_sav0 = 0;
ELSE pt_sav0 = add( pt_sav0, 9);
```

3.3. PREDICTOR VALUES COMPUTATION

This section computes the array rav1[0..8] needed for the spectral comparison and the threshold adaptation. It uses the L_av1[0..8] computed in section 3.2, and is divided in the three following sub-sections:

- Schur recursion to compute reflection coefficients.
- Step up procedure to obtain the aav1[0..8].
- Computation of the rav1[0..8].

3.3.1. Schur recursion to compute reflection coefficients

This sub-section is identical to the one used in the RPE-LTP algorithm. The array vpar[1..8] is computed with the array L_avl[0..8] as an input.

Schur recursion with 16 bits arithmetic:

```
IF( L_avl[0] == 0 ) THEN
                                |== FOR i = 1 to 8:
                                |     vpar[i] = 0;
                                |== NEXT i:
                                |     EXIT; /continue with section 3.3.2/
temp = norm( L_avl[0] );
|== FOR k=0 to 8:
|     sacf[k] = ( L_avl[k] << temp ) >> 16;
|== NEXT k:
```

Initialize array P[..] and K[..] for the recursion:

```
|== FOR i=1 to 7:
|     K[9-i] = sacf[i];
|== NEXT i:

|== FOR i=0 to 8:
|     P[i] = sacf[i];
|== NEXT i:
```

Compute reflection coefficients:

```
|== FOR n=1 to 8:
|     IF( P[0] < abs( P[1] ) ) THEN
|                                             |== FOR i = n to 8:
|                                             |     vpar[i] = 0;
|                                             |== NEXT i:
|                                             | EXIT; /continue with
|                                             |        section 3.3.2/
|     vpar[n] = div( abs( P[1] ), P[0] );
|     IF ( P[1] > 0 ) THEN vpar[n] = sub( 0, vpar[n] );
|     IF ( n == 8 ) THEN EXIT; /continue with section 3.3.2/
|
|   Schur recursion:
|
|     P[0] = add( P[0], mult_r( P[1], vpar[n] ) );
|==== FOR m=1 to 8-n:
|       P[m] = add( P[m+1], mult_r( K[9-m], vpar[n] ) );
|       K[9-m] = add( K[9-m], mult_r( P[m+1], vpar[n] ) );
|==== NEXT m:
|
|== NEXT n:
```

3.3.2. Step-up procedure to obtain the aavl[0..8]

Initialization of the step-up recursion:

```
L_coef[0] = 16384 << 15;
L_coef[1] = vpar[1] << 14;
```

Loop on the LPC analysis order:

```
|= FOR m = 2 to 8:
|== FOR i = 1 to m-1:
|==  temp = L_coef[m-i] >> 16;  / takes the msb /
|==  L_work[i] = L_add( L_coef[i], L_mult( vpar[m], temp ) );
|== NEXT i
|=
|== FOR i = 1 to m-1:
|==  L_coef[i] = L_work[i];
|== NEXT i
|=
|= L_coef[m] = vpar[m] << 14;
|= NEXT m:
```

Keep the aavl[0..8] on 13 bits for next section:

```
| FOR i = 0 to 8:
|  aavl[i] = L_coef[i] >> 19;
| NEXT i:
```

3.3.3. Computation of the ravl[0..8]

```
|= FOR i= 0 to 8:
|= L_work[i] = 0;
|== FOR k = 0 to 8-i:
|==  L_work[i] = L_add( L_work[i], L_mult( aavl[k], aavl[k+i] ) );
|== NEXT k:
|= NEXT i:

IF ( L_work[0] == 0 ) THEN normravl =0;
ELSE normravl = norm( L_work[0] );

|= FOR i= 0 to 8:
|= ravl[i] = ( L_work[i] << normravl ) >> 16;
|= NEXT i:
```

Keep the normravl for use in section 3.4 and 3.6.

3.4. SPECTRAL COMPARISON

This section computes the variable stat needed for the threshold adaptation. It uses the array L_av0[0..8] computed in section 3.2 and the array rav1[0..8] computed in section 3.3.3.

Re-normalize L_av0[0..8]:

```
IF ( L_av0[0] == 0 ) THEN
                            ¦ FOR i = 0 to 8:
                            ¦  sav0[i] = 4095;
                            ¦ NEXT i:
ELSE
    ¦ shift = norm( L_av0[0] );
    ¦= FOR i = 0 to 8:
----¦---~~sav0[i] = ( L_av0[i] << shift ) >> 19;~~
    ¦=  sav0[i] = ( L_av0[i] << shift-3 ) >> 16;
    ¦= NEXT i:
```

Compute partial sum of dm:

```
L_sump = 0;
¦= FOR i = 1 to 8:
¦= L_sump = L_add( L_sump, L_mult( rav1[i], sav0[i] ) );
¦= NEXT i:
```

Compute the division of partial sum by sav0[0]:

```
IF ( L_sump < 0 ) THEN L_temp = L_sub( 0, L_sump );
ELSE L_temp = L_sump;

IF ( L_temp == 0 ) THEN
                         ¦ L_dm  = 0;
                         ¦ shift = 0;
ELSE
    ¦ sav0[0] = sav0[0] << 3;
    ¦ shift = norm( L_temp );
    ¦ temp  = ( L_temp << shift ) >> 16;
    ¦ IF ( sav0[0] >= temp ) THEN
    ¦                              ¦ divshift = 0;
    ¦                              ¦ temp = div( temp, sav0[0] );
    ¦ ELSE
    ¦     ¦ divshift = 1;
    ¦     ¦ temp = sub( temp, sav0[0] );
    ¦     ¦ temp = div( temp, sav0[0] );
    ¦
    ¦ IF( divshift == 1 ) THEN L_dm = 32768;
    ¦ ELSE L_dm = 0;
    ¦
    ¦ L_dm = L_add( L_dm, temp) << 1;
    ¦ IF( L_sump < 0 ) THEN L_dm = L_sub( 0,  L_dm);
```

Re-normalization and final computation of L_dm:

```
L_dm = ( L_dm << 14 );
L_dm = L_dm >> shift;
L_dm = L_add( L_dm, ( ravl[0] << 11 ) );
L_dm = L_dm >> normravl;
```

Compute the difference and save L_dm:

```
L_temp   = L_sub( L_dm, L_lastdm );
L_lastdm = L_dm;
IF ( L_temp < 0 ) THEN L_temp = L_sub( 0, L_temp );
L_temp = L_sub( L_temp, 3277 );
```

Evaluation of the stat flag:

```
IF ( L_temp < 0 ) THEN stat = 1;
ELSE stat = 0;
```

3.5. PERIODICITY DETECTION

This section just sets the ptch flag needed for the threshold adaptation.

```
temp = add( oldlagcount, veryoldlagcount );
IF ( temp >= 4 ) THEN ptch = 1;
ELSE ptch = 0;
```

3.6. THRESHOLD ADAPTATION

This section uses the variables e_pvad, m_pvad, e_acf0 and m_acf0 computed in section 3.1. It also uses the flags stat (see section 3.4) and ptch (see section 3.5). It follows the flowchart represented on figure 2.2.

Some constants, represented by a floating point format, are needed and a symbolic name (in capital letter) for their exponent and mantissa is used; table 3-2 lists all these constants with the symbolic names associated and their numerical constant values.

Constant	Exponent	Mantissa
pth	E_PTH = 19	M_PTH = 18750
margin	E_MARGIN = 27	M_MARGIN = 19531
plev	E_PLEV = 20	M_PLEV = 25000

Table 3-2. List of constants

NOTE: Floating point representation of constants used in section 3.6:

```
pth    = 2(E_PTH)x(M_PTH/32768).
margin= 2(E_MARGIN)x(M_MARGIN/32768).
plev   = 2(E_PLEV)x(M_PLEV/32768).
```

Test if acf0 < pth; if yes set thvad to plev:

```
comp = 0;
IF ( e_acf0 < E_PTH ) THEN  comp = 1;
IF ( e_acf0 == E_PTH ) THEN  IF ( m_acf0 < M_PTH ) THEN comp =1;
IF ( comp == 1 ) THEN
                          | e_thvad = E_PLEV;
                          | m_thvad = M_PLEV;
                          | EXIT; /continue with section 3.7/
```

Test if an adaptation is needed:

```
comp = 0;
IF ( ptch == 1 ) THEN comp = 1;
IF ( stat == 0 ) THEN comp = 1;
IF ( comp == 1 ) THEN
                          | adaptcount = 0;
                          | EXIT; /continue with section 3.7/
```

Incrementation of adaptcount:

```
adaptcount = add( adaptcount, 1 );
IF ( adaptcount <= 8 ) THEN EXIT; /continue with section 3.7/
```

Computation of thvad-(thvad/dec):

```
m_thvad = sub( m_thvad, (m_thvad >> 5 ) );
IF ( m_thvad < 16384) THEN
                              | m_thvad = m_thvad << 1;
                              | e_thvad = sub( e_thvad, 1 );
```

Computation of pvad*fac:

```
L_temp = L_add( m_pvad, m_pvad );
L_temp = L_add( L_temp, m_pvad );
L_temp = L_temp >> 1;
e_temp = add( e_pvad, 1 );
IF ( L_temp > 32767 ) THEN
                                  ¦ L_temp = L_temp >> 1;
                                  ¦ e_temp = add( e_temp, 1 );
m_temp = L_temp;
```

Test if thvad < pvad*fac:

```
comp = 0;
IF ( e_thvad < e_temp) THEN comp = 1;
IF (e_thvad == e_temp) THEN  IF (m_thvad < m_temp) THEN comp =1;
```

Computation of minimum (thvad+(thvad/inc), pvad*fac) if comp = 1:

```
IF ( comp == 1 ) THEN
¦  Compute thvad +(thvad/inc).
¦ L_temp = L_add( m_thvad, (m_thvad >> 4 ) );
¦ IF ( L_temp > 32767 ) THEN
¦                                  ¦ m_thvad = L_temp >> 1;
¦                                  ¦ e_thvad = add( e_thvad,1 );
¦ ELSE m_thvad = L_temp;
¦ comp2 = 0;
¦ IF ( e_temp < e_thvad) THEN comp2 = 1;
¦ IF (e_temp == e_thvad) THEN IF (m_temp<m_thvad) THEN comp2 = 1;
¦ IF ( comp2 == 1 ) THEN
¦                                ¦ e_thvad = e_temp;
¦                                ¦ m_thvad = m_temp;
```

Computation of pvad + margin:

```
IF ( e_pvad == E_MARGIN ) THEN
                                 | L_temp = L_add(m_pvad, M_MARGIN);
                                 | m_temp = L_temp >> 1;
                                 | e_temp = add( e_pvad, 1 );
ELSE
    | IF ( e_pvad > E_MARGIN ) THEN
    |      | temp = sub( e_pvad, E_MARGIN );
    |      | temp = M_MARGIN >> temp;
    |      | L_temp = L_add( m_pvad, temp );
    |      | IF ( L_temp > 32767) THEN
    |      |                             | e_temp = add( e_pvad, 1 );
    |      |                             | m_temp = L_temp >> 1;
    |      | ELSE
    |      |     | e_temp = e_pvad;
    |      |     | m_temp = L_temp;
    | ELSE
    |      | temp = sub( E_MARGIN, e_pvad );
    |      | temp = m_pvad >> temp;
    |      | L_temp = L_add( M_MARGIN, temp );
    |      | IF (L_temp > 32767) THEN
    |      |                            | e_temp = add( E_MARGIN, 1);
    |      |                            | m_temp = L_temp >> 1;
    |      | ELSE
    |      |     | e_temp = E_MARGIN;
    |      |     | m_temp = L_temp;
```

Test if thvad > pvad + margin:

```
comp = 0;
IF ( e_thvad > e_temp) THEN comp = 1;
IF (e_thvad == e_temp) THEN  IF (m_thvad > m_temp) THEN comp =1;

IF ( comp == 1 ) THEN
                       | e_thvad = e_temp;
                       | m_thvad = m_temp;
```

Initialize new rvad[0..8] in memory:

```
normrvad  = normravl;

|= FOR i = 0 to 8:
|= rvad[i] = ravl[i];
|= NEXT i:
```

Set adaptcount to adp + 1:

```
adaptcount = 9;
```

3.7. VAD DECISION

This section only outputs the result of the comparison between pvad and thvad using the pseudo-floating point representation of thvad and pvad. The values e_pvad and m_pvad are computed in section 3.1 and the values e_thvad and m_thvad are computed in section 3.6.

```
vvad = 0;
IF (e_pvad >  e_thvad) THEN vvad = 1;
IF (e_pvad == e_thvad) THEN IF (m_pvad > m_thvad) THEN vvad =1;
```

3.8. VAD HANGOVER ADDITION

This section finally sets the vad decision for the current frame to be processed.

```
IF ( vvad == 1 ) THEN burstcount = add( burstcount, 1 );
ELSE burstcount = 0;

IF ( burstcount >= 3 ) THEN
                              | hangcount =  5;
                              | burstcount = 3;

vad = vvad;
IF ( hangcount >= 0 ) THEN
                             | vad = 1;
                             | hangcount = sub( hangcount, 1 );
```

3.9. PERIODICITY UPDATING

This section must be delayed until the LTP lags are computed by the RPE-LTP algorithm. The LTP lags called Nc in the speech encoder are renamed lags[0..3] (index 0 for the first sub- segment of the frame, 1 for the second and so on).

Loop on sub-segments for the frame:

```
lagcount = 0;

¦= FOR i = 0 to 3:
¦=  Search the maximum and minimum of consecutive lags.
¦= IF ( oldlag > lags[i] ) THEN
¦=                                     ¦ minlag = lags[i];
¦=                                     ¦ maxlag = oldlag;
¦= ELSE
¦=       ¦ minlag = oldlag;
¦=       ¦ maxlag = lags[i] ;
¦=
¦= Compute smallag (modulo operation not defined ):
¦=
¦= smallag = maxlag;
¦== ¦ FOR j = 0 to 2:
¦== ¦  IF (smallag >= minlag) THEN smallag =sub( smallag, minlag);
¦== ¦ NEXT j;
¦=
¦= Minimum of smallag and minlag - smallag:
¦=
¦= temp = sub( minlag, smallag );
¦= IF ( temp < smallag ) THEN smallag = temp;
¦= IF ( smallag < 2 ) THEN lagcount = add( lagcount, 1 );
¦= Save the current LTP lag.
¦= oldlag = lags[i];
¦= NEXT i:
```

Update the veryoldlagcount and oldlagcount:

```
veryoldlagcount = oldlagcount;
oldlagcount     = lagcount;
```

4. DIGITAL TEST SEQUENCES

This chapter provides information on the digital test sequences that have been designed to help the verification of implementations of the Voice Activity Detector. Copies of these sequences are available (see Annex 2).

4.1. TEST CONFIGURATION

The VAD must be tested in conjunction with the speech encoder defined in recommendation GSM 06.10. The test configuration is shown in fig 4-1. The input signal to the speech encoder is the sop[...] signal as defined in recommendation GSM 06.10 table 5.1. The relevant parameters produced by the speech encoder are input to the VAD algorithm to produce the VAD output. This output has to be checked against some reference files.

The file format of the encoder output parameters given in recommendation GSM 06.10 table 5.1 is extended to carry the VAD information.

The VAD information is placed in the unused bit 15 (MSB) of the first encoded parameter:

```
    LAR(1): bit 15 := 1 if VAD on

            bit 15 := 0 if VAD Off
```

Furthermore, in order to facilitate approval testing over the air interface, the SP flag generated by the TX DTX handler (see recommendation GSM 06.31) on the basis of the VAD flag is placed in the MSB position of the second encoded parameter:

```
    LAR(2): bit 15 := 1 if SP on

            bit 15 := 0 if SP off
```

The output file will also contain the SID codeword and the comfort noise parameters as described in recommendations GSM 06.12 and GSM 06.31.

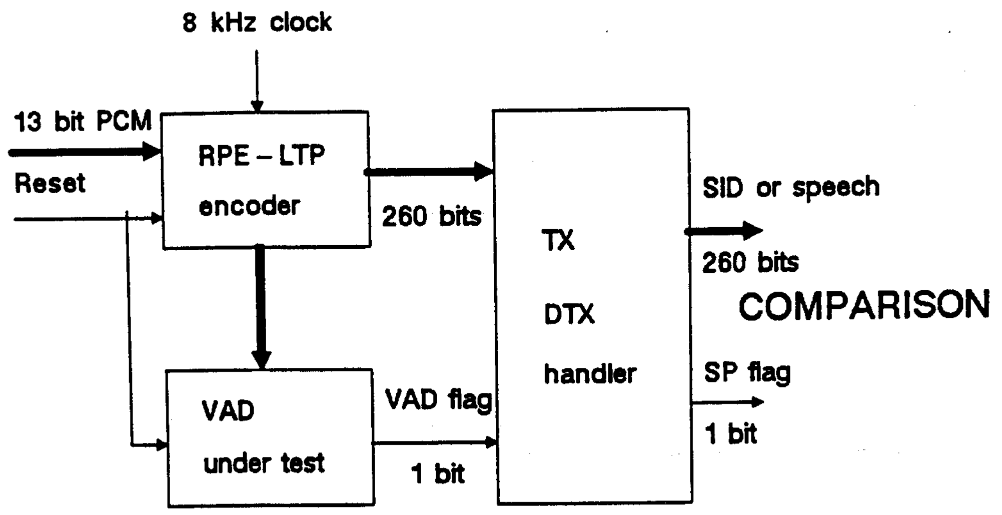

Fig 4 - 1. VAD test configuration

4.2. TEST SEQUENCES

The test sequences are described in detail in Annex 2.

ANNEX 1. SIMPLIFIED BLOCK FILTERING OPERATION

Consider an 8th order transversal filter with filter coefficients a0..a8, through which a signal is being passed, the output of the filter being:

$$s'_n := -\sum_{i=0}^{8} a_i s_{n-i} \qquad [1]$$

If we apply block filtering over 20ms segments, then this equation becomes:

$$s'_n := -\sum_{i=0}^{8} a_i s_{n-i} \qquad ;\ n = 0..167 \quad ;\ 0 <= n-i <= 159 \qquad [2]$$

If the energy of the filtered signal is then obtained for every 20ms segment, the equation for this is:

$$Pvad := \sum_{n=0}^{167} \left(-\sum_{i=0}^{8} a_i s_{n-i} \right)^2 \qquad ;\ 0 <= n-i <= 159 \qquad [3]$$

We know that (see recommendation GSM 06.10, section 3.1.4):

$$ACF[i] := \sum_{n=0}^{159} s_n s_{n-i} \qquad ;\ i = 0..8 \quad ;\ 0 <= n-i <= 159 \qquad [4]$$

If equation [3] is expanded and ACF[0]..ACF[8] are substituted for sn then we arrive at the equations:

$$Pvad := r[0]ACF[0] + 2\sum_{i=1}^{8} r[i]ACF[i] \qquad [5]$$

Where:

$$r[i] := \sum_{k=0}^{8-i} a_k a_{k+i} \qquad ;\ i = 0..8 \qquad [6]$$

ANNEX 2. DESCRIPTION OF DIGITAL TEST SEQUENCES

A2.1. TEST SEQUENCES

The VAD algorithm uses results from the full rate speech encoder defined in recommendation GSM 06.10. In the testing of the VAD, it is assumed that the relevant speech encoder functions have been verified by the test sequences defined in recommendation GSM 06.10.

The five types of input sequences are briefly described below.

Spectral comparison

The two kinds of statements of the spectral comparison algorithm (section 3.4), arithmetic statements and control statements, are tested by separate test sequences.

Arithmetic statements:

```
spect_a1.*
spect_a2.*
```

Control statements

```
spect_c1.*
spect_c2.*
spect_c3.*
spect_c4.*
```

Threshold adaptation

There are two types of tests to verify the threshold adaptation described in section 3.6:

```
adapt_i1.*
adapt_i2.*
```

The initial test sequences test the acf0 and VAD decision. A fault in the VAD decision will cause all the other sequences to fail, so it is recommended that this test is run before all other tests.

```
adapt_m1.*
adapt_m2.*
```

The main test sequences will check the basic threshold adaptation mechanism.

Periodicity detection

pitch1.*
pitch2.*

These sequences check the periodicity detection algorithm described in section 3.5.

"Safety" and initialisation

safety.*

This sequence checks that safety tests have been implemented to prevent zero values being passed to the *norm* function. It checks the functions described in the Adaptive Filtering and Energy Computation section (section 3.1), and the Predictor Values Computation (section 3.3). This sequence also checks the initialization of th_{vad} and the r_{vad} array.

Real speech

good_sp.*
bad_sp.*

Because the test sequences cannot be guaranteed to find every possible error, there is a small possibility that an implementation of the correct output for test sequences, but fail with real speeech. Because of this, an extra set of sequences are included that consist of barely detectable speech and very clean speech.

There are 3 different file extensions:

*.INP:	speech encoder input sequences, binary files
*.VAD:	output flag of the VAD algorithm, ASCII files
*.COD:	TX DTX handler output sequences, binary files for comparison with VAD/DTX handler output.

The *.COD files contain speech coder output information in the format described in section 4.

It should be noted that there is no requirement in recommendation GSM 06.12 for a bit exact implementation of the averaging procedure to calculate the "LAR" and "xmax" parameters in the SID frames. Different implementations are allowed.

The algorithms used for the calculation of the LAR and xmax parameters of the SID frames are therefore reproduced below:

LAR averaging:

```
¦ FOR i = 1 to 8:
¦  L_Temp = 2;                    /* const. for rounding*/
¦   ¦ FOR n = 1 to 4:
¦   ¦  L_Temp1 = LAR[j-n](i);     /*conversion 16 --> 32 bit*/
¦   ¦  L_Temp  = L_Add( L_Temp , L_Temp1 );
¦   ¦ NEXT n
¦  L_Temp = L_temp >> 2;
¦  mean (LAR(i)) = L_Temp;        /*conversion 32 --> 16 bit*/
¦ NEXT i;
```

xmax averaging

```
L_Temp = 8;                       /* const. for rounding*/

¦ FOR n = 1 to 4:
¦   ¦ FOR i = 1 to 4:
¦   ¦  L_Temp1 = xmax[j-n](i);    /*conversion 16 --> 32 bit*/
¦   ¦  L_Temp  = L_Add( L_Temp , L_Temp1 );
¦   ¦ NEXT i
¦ NEXT n

L_Temp = L_Temp >> 4;

mean (xmax) = L_Temp;             /*conversion 32 --> 16 bit*/
```

A2.2. FILE FORMAT DESCRIPTION

All the *.INP and *.COD files are written in binary using 16 bit words, while all *.VAD files are written in ASCII format. The sizes of the files are shown in Table A2-1, A2-2 and A2-3. The detailed format of the *.INP and *.COD files is in accordance with the descriptions given in recommendation GSM 06.10 section 5.

The diskette is formatted according to the high capacity (1.2 Mb) specifications for MS-DOS PC-AT compatible computers.

File:	Frames:	Size in bytes:
spect_a1.inp	22	7040
spect_a2.inp	22	7040
spect_c1.inp	48	15360
spect_c2.inp	48	15360
spect_c3.inp	48	15360
spect_c4.inp	48	15360
adapt_i1.inp	67	21440
adapt_i2.inp	48	15360
adapt_m1.inp	403	128960
adapt_m2.inp	376	120320
pitch1.inp	35	11200
pitch2.inp	35	11200
safety.inp	5	1600
good_sp.inp	312	99840
bad_sp.inp	312	99840

Table A2-1. File sizes for *.INP extension files

File:	Frames:	Size in bytes:
spect_a1.cod	22	3344
spect_a2.cod	22	3344
spect_c1.cod	48	7296
spect_c2.cod	48	7296
spect_c3.cod	48	7296
spect_c4.cod	48	7296
adapt_i1.cod	67	10184
adapt_i2.cod	48	7296
adapt_m1.cod	403	61256
adapt_m2.cod	376	57152
pitch1.cod	35	5320
pitch2.cod	35	5320
safety.cod	5	760
good_sp.cod	312	47424
bad_sp.cod	312	47424

Table A2-2. File sizes for *.COD extension files

File:	Frames:	Size in bytes:
spect_a1.cod	22	88
spect_a2.cod	22	88
spect_c1.cod	48	192
spect_c2.cod	48	192
spect_c3.cod	48	192
spect_c4.cod	48	192
adapt_i1.cod	67	268
adapt_i2.cod	48	192
adapt_m1.cod	403	1612
adapt_m2.cod	376	1504
pitch1.cod	35	140
pitch2.cod	35	140
safety.cod	5	20
good_sp.cod	312	1248
bad_sp.cod	312	1248

Table A2-3. File sizes for *.VAD extension files

ANNEX 3. VAD PERFORMANCE

In optimising a VAD a difficult trade-off has to be made between speech clipping which reduces the subjective performance of the system, and the average activity factor. The benefit of DTX is increased as the average activity factor is reduced. However, in general, a reduction of the activity will be associated with a greater risk for audible speech clipping.

In the optimisation process, great emphasis has been placed on avoiding unnecessary speech clipping. However, it has been found that a VAD with virtually no audible clipping would result in a very high activity and very little DTX advantage.

The VAD specified in this recommendation introduces audible and possibly objectionable clipping in certain cases, mainly with low input levels. However, a comprehensive evaluation programme consisting of about 600 individual conversations conducted in a wide range of realistic conditions, it was found that about 90% of the conversations were free from objectionable clipping.

The voice activity performance of the VAD is summarised in table A3-1. The activity figures are averages of a large number of conversations covering factors like different talkers, noise characteristics and locations. It should be noted that the actual activity of a particular talker in a specific conversation may vary considerably relative to the averages given. This is due both to the variation in talker behaviour as well as to the level dependency of the VAD (the channel activity has been found to decrease by about 0.5 points of percentage per dB level reduction). However, as mentioned above, a decreased speech input level increases the risk of objectionable speech clipping.

All the values given are activity figures, i.e. the % of time the radio channel has to be on.

Telephone instrument:	Situation:	Typical channel activity factor:
Handset	Quiet location	55 %
Handset	Moderate office noise with voice interference	60 %
Handset	Strong voice interference (eg airport/railway station)	65-70 %
Handsfree/ handset	Variable vehicle noise	60 %

Table A3-1. Summary of channel activity