**ETSI SMG11**

**Title: AMR permanent document (AMR-9):**

**Complexity and delay assessment**

**Version: 1.3**

**Editor: Frédéric Lejay (Alcatel)**

1. Introduction

This document contains the complexity and delay assessment methodology to be applied for the AMR Codec qualification, selection and verification/characterisation phases.

Since this document deals only with methodology, the complexity and delay requirements are not given here but in the Design Constraints permanent AMR document (AMR-5).

Note that the assessment methodology is basically the same as for the GSM Half-Rate and Enhanced Full Rate Codec standardisation.

2. Source Code

* **For the qualification phase**, proponents are allowed to use a floating point ANSI C source code. In that case, it is the candidate responsibility to correctly assess the complexity figures of an equivalent fixed point implementation such as defined below.
* **For the selection and verification/characterisation phases**, the complexity assessment will be based on fixed point ANSI C code using the arithmetic operations defined in [2] and 16 or/and 32 bits data. Counters have to be included for wMOPS computation.

3. Assessment methodology

## 3.1. Complexity

### 3.1.1. General rules

The complexity of the channel or speech codec is characterised by the 3 following figures:

* wMOPS (weighted MOPS)
* RAM size
* ROM size (data only)

The Figure Of Merit formula to be used is:

FOM = wMOPS + RAM/5 + ROM/20

where RAM and ROM are evaluated in Kbytes (1Kbyte = 500 words of 16 bits)

For the computation of the Figure of Merit, the worst case wMOPS and RAM for any codec mode, and the total ROM for all codec modes (Half Rate or full Rate) must be used.

For every proponent, complexity figures are computed independently for:

* Half-rate channel coder/decoder, including control loop
* Full-rate channel coder/decoder, including control loop
* Speech coder/decoder, excluding VAD/DTX
* VAD/DTX

The detailed description of the assessment methodology can be found in the reference document of the GSM Half-Rate standardisation [1]. We summarise in the following sections the main points for each of the complexity component.

### 3.1.2. weighted MOPS

There are two different approaches for computing wMOPS:

* Worst Observed Frame (WOF) (use of counters)
* Theoretical Worst Case (TWC)

Both are based on the use of the set of basic ETSI fixed point operators with associated weight. Those operators and their weights have been defined in [2]. They can also be found in Annexe C of present document.

The Worst Observed Frame wMOPS will be computed for the selection and verification phases.

For the verification phase, the Theoretical Worst Case wMOPS will also be computed.

We remind that the coder and decoder may be in a different codec mode, but in same rate (half-rate or full-rate). It means that wMOPS have to be computed independently for the coder and decoder, taking for each module the maximum for all codec modes.

#### 3.1.2.1. Worst Observed Frame (WOF)

For the Worst Observed Frame approach, weighted MOPS is usually computed automatically using operation counters introduced in the C source code.

A speech input sample is processed through the codec, and the wMOPS for each frame are computed independently for the four modules: channel encoder, channel decoder, speech encoder and speech decoder. The WOF wMOPS for each of the four modules is the maximum wMOPS value obtained over the processed speech frames. This has to be done for any mode in order to find the worst case for each of the modules.

WMOPS have also to be computed for control loop algorithm when adaptation is active.

For the qualification, it is not required to use the counters to compute the Worst Observed Frame. Since floating point implementation might be used, the methodology to use is left open.

For the selection phase, wMOPS are computed running :

* the whole selection test plan with its speech material
* an additional sample provided on purpose by BT at all modes and rates, no adaptation. This 28 seconds sample is build of following segments :
* digital silence (0x00) (2 secs)
* A-law silence (DC offset of 0x08) (2 secs)
* swept sine from 10 to 3900 Hz (4 secs)
* 200ms tone bursts from 100 to 3900 Hz in 200Hz steps (4 secs)
* speech normalized to -26dBov and then scaled up by 20dB (8 secs)
* classical music (8 secs)

#### 3.1.2.2. Theoretical Worst Case

The Theoretical Worst Case methodology is based on the evaluation of the Theoretical Worst possible Case.

The final result is the sum of the TWC complexity for the coder and the TWC complexity for the decoder computed independently.

The calculation of the TWC complexity is done in two steps:

1) - Pre-selection of the flow paths of the program;

2) - Calculation and selection of the program flow path TWC complexity.

The pre-selection of the flow paths of the program is done by travelling through the listing of the program from its beginning to its end, in order to represent the program as a tree where:

* a root corresponds to the beginning of the program;
* a node corresponds to the encounter of a choice;
* a branch corresponds to the list of the consecutive executed functions where no choice has been encountered;
* a leaf corresponds to the end of a possible flow path.

All the considered flow paths must be realistic, it means that two consecutive choices (even in different procedures) must be consistent : i.e. it at the beginning of the coder the speech is considered as unvoiced, this will be true until the end of the program.

When the Theoretical Worst Case Path is identified, the complexity is evaluated using the weight of the defined basic operators.

### 3.1.3. RAM

The amount of RAM is the sum of the static RAM and scratch RAM.

The determination of static RAM takes into account only C declarations. It means that only the variables declared in C as ‘static’ are counted, with the exception of static RAM arrays that are used like constant tables and counted as ROM.

Scratch memory which can be shared by different routines is counted only once. Scratch RAM will be the maximum value of the encoder value and the decoder value (but independently for channel and speech codec).

The amount of RAM is expressed in 16 bit words.

### 3.1.4. ROM (Tables)

The total data ROM is computed counting all the different tables only once (but independently for channel and speech codec).

The amount of ROM is expressed in 16 bit words.

### 3.1.5. Program ROM (Source code size)

Code size will be estimated using the same method as for the Half-Rate standardisation.

It is estimated computing the ratio between the AMR candidate and the GSM FR number of ‘pure-instruction’ C-lines of the source. When a floating point C source code is used, the number of lines has to be estimated for an ‘equivalent fixed-point implementation’.

A procedure for estimating the number of « pure instructions » can be :

* edit the C source files to remove data memory declarations ;
* use the C pre-processor to expand macros and remove comments ;
* filter the result by removing blank lines and function declarations ;
* count the number of remaining lines ;
* compute the ratio with HR channel codec and EFR speech codec results (see AnnexeA, number of lines) ;

## 3.2.

## Algorithmic round-trip delay

The MS-to-MS algorithmic round-trip delay evaluation of a codec mode is based, as for the HR and EFR standardisation, on four codec dependent algorithmic delay contributors :

* analysis frame length delay (*Tsample*): duration of the segment of PCM speech operated on by the speech transcoder.
* interleaving and de-interleaving delay (*Trftx*): time required for transmission of a speech frame over the air interface due to interleaving and de-interleaving.
* uplink Abis delay (*TAbisu*): time needed to transmit the minimum amount of bits over the Abis interface that are required at the speech decoder to synthesise the first output sample.
* downlink Abis delay (*TAbisd*): time required to transmit all the speech frame data bits over the Abis interface in the downlink direction that are required to encode one speech frame.

The formula used for round-trip delay evaluation is the following:

***Dround-trip* = 2(*Tsample* + *Trftx*)+ *TAbisu* + *TAbisd***

The proponents must compute and provide figures for each component of the round trip delay for following configurations :

* highest delay of full-rate modes with 16kbps sub-multiplexing scheme ;
* highest delay of half-rate modes suitable to 8kbps sub-multiplexing scheme ;
* highest delay of half-rate modes with 16kbps sub-multiplexing ;

The Abis delays (uplink and downlink) should be computed with a similar methodology as the one used in the GSM Recommendations 03.05 for the GSM FR, and 06.55 for the GSM EFR.

For the qualification and the selection phases the proponents must justify how the delay figures were computed. To that purpose, they can base the Abis delays on either type of TRAU frames (existing FR, and HR TRAU frames, or new TRAU frames format proposed for AMR).

**REFERENCES**

[1]: ETSI/TM/TM5/TCH-HS - TD93/95. Expert Group Traffic Channel Half Rate Speech. *‘Reference document of GSM full and half-rate complexity evaluation rules for the half-rate selection’*. Version 2.2.0.

Copy: TD SMG11 AMR 3/98

[2] Annex 0 of [1], also copied in TD SMG11 AMR 155/97

**Annexe A: Complexity and Delay for current GSM Codecs**

1. Complexity

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Codec | wMOPS | data RAM  (16 bits words) | data ROM  (16 bits words) | program ROM  (C lines / assembly instruction) | FOM |
| FR channel codec | 1.72 | 1690 | 824 | 342 lines | 2.5 |
| FR speech codec | 2.95 | 1201 | 80 | 934 lines | 3.4 |
| FR speech + channel | 4.67 | 2053 | 904 | 1276 lines /2000-3000 | 5.6 |
| HR channel codec | 2.69 | 3154 | 900 | 1328 lines | 4.0 |
| HR speech codec | 18.47 | 4363 | 7881 | 4128 lines | 21.0 |
| HR speech + channel | 21.16 | 5002 | 8781 | 5456 lines / 8000-12000 | 24.0 |
| EFR channel codec | 2.69 | 2364 | 96 | ? | 3.7 |
| EFR speech codec | 15.21 | 4711 | 5267 | 1832 | 17.6 |
| EFR speech + channel | 17.9 | 5167 | 5363 | 6000-9000 | 20.5 |

**References:**

* FR codec complexities are taken from :

*Complexity evaluation of the full-rate, ANT and Motorola codecs for the Phase III selection of the half-rate codec, Tdoc TCH-HS 93/95, Source: TCH-HS complexity evaluation sub-group (Annexes)*

FR speech codec complexity do not include VAD/DTX

* HR codec complexities are taken from :

*Complexity evaluation of the full-rate, ANT and Motorola codecs for the Phase III selection of the half-rate codec, Tdoc TCH-HS 93/95, Source: TCH-HS complexity evaluation sub-group (Annexes),*

and *Complexity Evaluation of the Optimised Versions (3.2, 3.3 and 3.4) of the Motorola codec v1.0, TCH-HS 36/94, Source Alcatel Mobile Communication*

HR speech codec complexity do not include VAD/DTX. The HR codec is taken as the optimised version Motorola+ with delta added to v3.2 and v3.3:

|  |  |  |  |
| --- | --- | --- | --- |
| HR ChC | 2.67 + 0.020 (v3.3) = 2.69 | 3154 | 900 |
| HR SpC | 17.66 + 0.563(v3.2) + 0.253(v3.3) = 18.47 | 3617 + 612(v3.2) + 134(v3.3) = 4363 | 7786 + 95(v3.3) = 7881 |
| HR SpC+ChC | 21.16 | 5002 | 8781 |

* EFR codec complexities are taken from:

*«Enhanced Full Rate Complexity Evaluation», Tdoc SMG2-SEG 128/95, Source: Alcatel Mobile Phones, PA Consulting Group, Texas Instruments*

The EFR speech codec complexity includes the contribution from VAD/DTX.

2. Delay

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Coder | Tsample | Trftx | Tabisu | Tabisd | Total |
| Full Rate | 20.0 ms | 37.5 ms | 4.0 ms | 17.375 ms | 136.4 ms |
| Half-Rate, Abis=8 kb/s | 24.4 ms | 32.9 ms | 9.75 | 17.5 | 141.85 |
| Half-Rate, Abis=16 kb/s | 24.4 ms | 32.9 ms | 4.8125 | 8.375 | 127.7875 |
| Half-Rate, DTX, Abis=16 kb/s | 24.4 ms | 32.9 ms | 4.8125 | 18.06 | 137.4725 |
| Enhanced Full-Rate | 20.0 ms | 37.5 ms | 6.4375ms | 17.375 ms | 138.8 ms |

**References:**

* FR and EFR codec delay contributors are taken from :

*GSM 06.55*

* HR codec delay contributors are taken from:

*ETSI SMG2 SEG Tdoc 21/96 rev 1, A proposal for GSM EFR and GSM HR delay using a common approach, Texas Instrument, Matra, AEG.*

**Annexe B: Selection rules for HR/EFR selection**

**HR selection:**

The complexity figure obtained using the Figure Of Merit for the HR had to be less than or equal to 4 times the FR figure.

The Theoretical Worst Case was used.

**EFR selection:**

The requirements agreed by the Speech Expert Group were (SEG TD 82/95)

wMOPS of EFR lower or equal than Half Rate wMOPS (21.16 wMOPS)

Data RAM lower or equal than Half Rate wMOPS (5002 words)

Data ROM lower or equal than Half Rate wMOPS (8781 words)

Program ROM lower or equal than Half Rate wMOPS

The 4 parameters are the total contribution of the Speech Coder, the Speech Decoder, the Channel Coder, the Channel Decoder and the VAD/DTX algorithm.

wMOPS are computed following the Theoretical Worst Case methodology

**Annexe C: ETSI Basic Operators as defined in [2]**

1. Variable definitions

The variables used in the operators are signed integers in 2’s complements representation, defined by:

var1, var2 : 16 bit variables

L\_var1, L\_var2, L\_var3 : 32 bit variables

2. Arithmetic operators with complexity weight of 1

|  |  |
| --- | --- |
| add (var1, var2) | Performs the addition (var1+var2) with overflow control and saturation; the 16 bit result is set at +32767 when overflow occurs or at ‑32768 when underflow occurs. |
| sub (var1, var2) | Performs the subtraction (var1‑var2) with overflow control and saturation; the 16 bit result is set at +32767 when overflow occurs or at ‑32768 when underflow occurs. |
| abs\_s (var1) | Absolute value of var1;  abs\_s (‑32768) = 32767. |
| shl (var1, var2) | Arithmetically shift the 16 bit input var1 left var2 positions. Zero fill the var2 LSB of the result. If var2 is negative, arithmetically shift var1 right by ‑var2 with sign extension. Saturate the result in case of underflows or overflows. |
| shr (var1, var2) | Arithmetically shift the 16 bit input var1 right var2 positions with sign extension. If var2 is negative, arithmetically shift var1 left by ‑var2 and zero fill the ‑var2 LSB of the result:  shr (var1, var2) = shl (var1, ‑var2).  Saturate the result in case of underflows or overflows. |
| extract\_h (L\_var1) | Return the 16 MSB of L\_var1. |
| extract\_l (L\_var1) | Return the 16 LSB of L\_var1. |
| mult (var1, var2) | Performs the multiplication of var1 by var2 and gives a 16 bit result which is scaled, i.e.  mult (var1, var2) = extract\_l (L\_shr ((var1 times var2), 15))  and mult (‑32768, ‑32768) = 32767. |
| L\_mult (var1, var2) | L\_mult is the 32 bit result of the multiplication of var1 times var2 with one shift left, i.e.  L\_mult (var1, var2) = L\_shl (var1 times var2), 1)  and L\_mult (‑32768, ‑32768) = 2147483647. |
| negate (var1) | Negate var1 with saturation, saturate in the case when input is ‑32768:  negate (var1) = sub (0, var1). |
| round (L\_var1) | Round the lower 16 bits of the 32 bit input number into the MS 16 bits with saturation. Shift the resulting bits right by 16 and return the 16 bit number:  round (L\_var1) = extract\_h (L\_add (L\_var1, 32768)). |
| L\_mac (L\_var3, var1, var2) | Multiply var1 by var2 and shift the result left by 1. Add the 32 bit result to L\_var3 with saturation, return a 32 bit result:  L\_mac (L\_var3, var1, var2) = L\_add (L\_var3, L\_mult (var1, var2)). |
| L\_msu (L\_var3, var1, var2) | Multiply var1 by var2 and shift the result left by 1. Subtract the 32 bit result from L\_var3 with saturation, return a 32 bit result:  L\_msu (L\_var3, var1, var2) = L\_sub (L\_var3, L\_mult (var1, var2)). |
| L\_macNs (L\_var3, var1, var2) | Multiply var1 by var2 and shift the result left by 1. Add the 32 bit result to L\_var3 without saturation, return a 32 bit result. Generates carry and overflow values:  L\_macNs (L\_var3, var1, var2) =  L\_add\_c (L\_var3, L\_mult (var1, var2)). |
| L\_msuNs (L\_var3, var1, var2) | Multiply var1 by var2 and shift the result left by 1. Subtract the 32 bit result from L\_var3 without saturation, return a 32 bit result. Generates carry and overflow values:  L\_msuNs (L\_var3, var1, var2) =  L\_sub\_c (L\_var3, L\_mult (var1, var2)). |

3. Arithmetic operations with complexity weight of 2

|  |  |
| --- | --- |
| L\_add (L\_var1, L\_var2) | 32 bit addition of the two 32 bit variables (L\_var1+L\_var2) with overflow control and saturation; the result is set at +2147483647 when overflow occurs or at ‑2147483648 when underflow occurs. |
| L\_sub (L\_var1, L\_var2) | 32 bit subtraction of the two 32 bit variables (L\_var1‑L\_var2) with overflow control and saturation; the result is set at +2147483647 when overflow occurs or at ‑2147483648 when underflow occurs. |
| L\_add\_c (L\_var1, L\_var2) | Performs the 32 bit addition with carry. No saturation. Generates carry and overflow values. The carry and overflow values are binary variables which can be tested and assigned values. |
| L\_sub\_c (L\_var1, L\_var2) | Performs the 32 bit subtraction with carry (borrow). Generates carry (borrow) and overflow values. No saturation. The carry and overflow values are binary variables which can be tested and assigned values. |
| L\_negate (L\_var1) | Negate the 32 bit L\_var1 with saturation, saturate in the case where input is ‑2147483648. |
| L\_shl (L\_var1, var2) | Arithmetically shift the 32 bit input L\_var1 left var2 positions. Zero fill the var2 LSB of the result. If var2 is negative, arithmetically shift L\_var1 right by ‑var2 with sign extension. Saturate the result in case of underflows or overflows. |
| L\_shr (L\_var1, var2) | Arithmetically shift the 32 bit input L\_var1 right var2 positions with sign extension. If var2 is negative, arithmetically shift L\_var1 left by ‑var2 and zero fill the ‑var2 LSB of the result. Saturate the result in case of underflows or overflows. |
| mult\_r (var1, var2) | Same as mult but with rounding, i.e.  mult\_r (var1, var2) =  extract\_l (L\_shr (((var1 times var2)+16384), 15))  and mult\_r (‑32768, ‑32768) = 32767. |
| shr\_r (var1, var2) | Same as shr (var1, var2) but with rounding. Saturate the result in case of underflows or overflows:  If var2 is greater than zero then:  if (sub(shl(shr(var1,var2),1), shr(var1,sub(var2,1))) is equal to zero  then  shr\_r (var1, var2) = shr (var1, var2)  else  shr\_r (var1, var2) = add (shr (var1, var2), 1)  If var2 is less than or equal to zero then:  shr\_r (var1, var2) = shr (var1, var2). |
| shift\_r (var1, var2) | Same as shl (var1, var2) but with rounding. Saturate the result in case of underflows or overflows:  shift\_r (var1, var2) = shr\_r (var1, ‑var2). |
| mac\_r (L\_var3, var1, var2) | Multiply var1 by var2 and shift the result left by 1. Add the 32 bit result to L\_var3 with saturation. Round the LS 16 bits of the result into the MS 16 bits with saturation and shift the result right by 16. Return a 16 bit result.  mac\_r (L\_var3, var1, var2)  = round (L\_mac (L\_var3, var1, var2))  = extract\_h (L\_add (L\_add (L\_var3, L\_mult (var1, var2)), 32768)). |
| msu\_r (L\_var3, var1, var2) | Multiply var1 by var2 and shift the result left by 1. Subtract the 32 bit result from L\_var3 with saturation. Round the LS 16 bits of the result into the MS 16 bits with saturation and shift the result right by 16. Return a 16 bit result.  msu\_r (L\_var3, var1, var2)  = round (L\_msu (L\_var3, var1, var2))  = extract\_h (L\_add (L\_sub (L\_var3, L\_mult (var1, var2)), 32768)). |
| L\_deposit\_h (var1) | Deposit the 16 bit var1 into the 16 bit MS bit of the 32 bit output. The 16 LS bits of the output are zeroed. |
| L\_deposit\_l (var1) | Deposit the 16 bit var1 into the 16 bit LS bit of the 32 bit output. The 16 MS bits of the output are sign extended. |

4. Arithmetic operations with complexity weight more than 2

## 4.1. Weight of 3:

|  |  |
| --- | --- |
| L\_shr\_r (L\_var1, var2) | Same as L\_shr (L\_var1, var2) but with rounding. Saturate the result in case of underflows or overflows:  - If var2 is greater than zero then:  if (L\_sub(L\_shl(L\_shr(L\_var1,var2),1), L\_shr(L\_var1,sub(var2,1)))  is equal to zero then  L\_shr\_r (L\_var1, var2) = L\_shr (L\_var1, var2)  else  L\_shr\_r (L\_var1, var2) = L\_add (L\_shr (L\_var1, var2), 1)  - If var2 is less than or equal to zero then:  L\_shr\_r (L\_var1, var2) = L\_shr (L\_var1, var2). |
| L\_shift\_r (L\_var1, var2) | Same as L\_shl (L\_var1, var2) but with rounding. Saturate the result in case of underflows or overflows.  L\_shift\_r (L\_var1, var2) = L\_shr\_r (L\_var1, ‑var2). |
| L\_abs (L\_var1) | Absolute value of L\_var1:  L\_abs(‑2147483648) = 2147483647. |

## 4.2. Weight of 4:

|  |  |
| --- | --- |
| L\_sat (L\_var1) | 32 bit L\_var1 is set to 2147483647 if an overflow occurred, or ‑2147483648 if an underflow occurred, on the most recent L\_add\_c, L\_sub\_c, L\_macNs or L\_msuNs operations. The carry and overflow values are binary variables which can be tested and assigned values. |

## 4.3. Weight of 15:

|  |  |
| --- | --- |
| norm\_s (var1) | Produces the number of left shifts needed to normalise the 16 bit variable var1 for positive values on the interval with minimum of 16384 and maximum 32767, and for negative values on the interval with minimum of ‑32768 and maximum of ‑16384; in order to normalise the result, the following operation must be done: norm\_var1 = shl (var1, norm\_s (var1)). |

## 4.4. Weight of 18:

|  |  |
| --- | --- |
| div\_s (var1, var2) | Produces a result which is the fractional integer division of var1 by var2; var1 and var2 must be positive and var2 must be greater than or equal to var1. The result is positive (leading bit equal to 0) and truncated to 16 bits. If var1=var2 then div (var1, var2) = 32767. |

## 4.5. Weight of 30:

|  |  |
| --- | --- |
| norml (L\_var1) | Produces the number of left shifts needed to normalise the 32 bit variable L\_var1 for positive values on the interval with minimum of 1073741824 and maximum 2147483647, and for negative values on the interval with minimum of ‑2147483648 and maximum of ‑1073741824; in order to normalise the result, the following operation must be done:  L\_norm\_var1 = L\_shl (L\_var1, norm\_l (L\_var1)). |

5. Data moves, logical operations, arithmetic test and other operations

## 5.1. Data moves:

* A data move short (16 bits) is weighted: 1
* A data move long (32 bits) is weighted: 2

A short variable cannot be moved to a long variable directly, and a long variable cannot be moved to a short variable directly. In these cases, functions such as round, extract\_l, extract\_h, L\_deposit\_l, L\_deposit\_h must be used.

There will be no extra weighting for data move for the following functions: extract\_l, extract\_h, L\_deposit\_l and L\_deposit\_h (the weighting of the data move is already included in the weighting of these functions).

Data moves are only counted in the following cases:

1. a data move from a constant to a variable;
2. a data move from a variable to a variable;
3. a data move of the result of a basic operation to an array variable;
4. when an arithmetic test is performed on an array variable.

## 5.2. Logical operations:

A logical operation is one of the following: And, Or, Xor, Not.

* A logical operation short (two 16 bit variables) is weighted: 1
* A logical operation long (two 32 bit variables) is weighted: 2

## 5.3. Arithmetic tests:

* An arithmetic test (short or long) is weighted: 2

All arithmetic test on data must be presented as a compare to zero. To perform comparison between two variables (or a variable and a non-zero constant), a subtract (sub or L\_sub) must be performed first.

## 5.4. Other operations:

Address computation must be excluded from the complexity evaluation. However, in a situation when, or if, extremely complex address computations are found, these address computations should be resolved by accounting for complexity using the basic operations.

There is no complexity counted for any loops, subroutine calls, etc., except for the complexity for arithmetic test on data in program control statement (e.g. do while).