**Source:** Audio Adhoc Group[[1]](#footnote-2)

**Title:** Source code verification plan v1.0

# Agenda item: 8

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Introduction

This document provides a preliminary verification plan for the Rel-6 PSM/MMS/MBMS fixed-point audio codec standardization. The source code of the fixed-point version of both audio codec will be brought to TSG-SA for approval (TSG-SA#27, March 14-17, 2005). Some critical items (as listed in [1]) will be verified by volunteering organizations before the source code is brought to TSG-SA.

The codecs under consideration are the extended-AMRWB (AMRWB+) and the enhanced aacPlus (e-aacPlus). The task under consideration is the item listed as 3 in [1]: “verification of the format of the C-code, correct implementation of complexity counters”.

# Verification of the format of the C-code

## Source-code verification

The verification laboratory checks that the C-code has been correctly implemented with basic operators (according to [2]) and that the C-code correctly implements the instrumentation that shall generate a maximum WMOPS score for each sample file.

Details of the instrumentation that may be used during the memory complexity (as described in Annex A) shall also be taken into consideration when the instrumentation of the source code is verified.

In case of ambiguity, similar cases will be identified from the source code of the AMR speech codec and a decision will be proposed accordingly.

The source code of the audio decoders will be verified first. The core of the AMR-WB contained inside the AMR-WB+ does not need to be verified either.

# Workplan

## Verification laboratories

The verification of the source code will be performed by STMicroelectronics (contact is [stephan.tassart@st.com](mailto:stephan.tassart@st.com)) [and any other volunteering companies].

|  |  |  |
| --- | --- | --- |
| **Task** | **Workload (lines of C code)** | **Company** |
| decoder e-aacPlus | 28000 | ST |
| decoder AMRWB+ | 21000(1) | ST |
| encoder e-aacPlus |  |  |
| encoder AMRWB+ | 14000(2) |  |

(1): some of the libraries needed in the AMRWB+ decoder are shared with the code AMRWB decoder and are not taken into account in this figure.

(2): some of the libraries needed in the AMRWB+ encoder are shared with the AMRWB+ decoder or the AMRWB core encoder and are not taken into account in this figure.

## Schedule

The workplan is organized as follow:

|  |  |
| --- | --- |
| **Date** | **Actions** |
| **16th Feb. 2005** | The source code of the fixed-point version of AMRWB+ is transferred to the verification laboratories. |
| **16th Feb. 2005** | The source code of the fixed-point version of e-aacPlus is transferred to the verification laboratories. |
| **17th - 18th Feb.** | Preliminary checks of the format of the source code. |
| **21th – 25th Feb.** | Meeting SA4#34 – Lisboa |
| **1st - 3rd Mar.** | Verification of the source code instrumentation of the AMRWB+ decoder (25000 lines of C-code). |
| **4th - 8th Mar.** | Verification of the source code instrumentation of the e-aacPlus decoder (25000 lines of C-code). |
| **9th Mar.** | Conf call 2pm CET: Discussion of partial verification results + test results. |
| **11th Mar.** | Partial verification report completed: verification of the format of the source code. |
| **14th - 17th Mar.** | Meeting TSG SA#27 (Tokyo). |

# References

[1] S4-030745 “Audio codec verification phase items”

[2] ETSI SMG-11 AMR#9 “Complexity and delay assessment”

1. RAM and ROM Complexity verification
   1. Definition

The RAM memory used by the software is the sum of all the non-const arrays or variables defined with a global visibility, all the static arrays or variables (known as the static memory or permanent allocation) and the maximum amount of RAM required by the stack (known as the scratch memory).

The ROM memory used by the software is the sum of all the const arrays or variables (defined in a global or in local visibility). The ROM memory does not include the program ROM (cf. [9]).

The following sample source code explains how the RAM and the ROM memory are evaluated.

Word16 buff[16];

const Word32 tab[32];

Word16

func(void \*state, Word16 a, const Word16 v[])

{

Word16 ret;

Word16 local\_buff[8];

static Word16 state=START;

[...]

return ret;

}

Code 1: Example of instrumented C-code

In this small example, the memory complexity would be evaluated as follow:

|  |  |  |
| --- | --- | --- |
| **C instruction** | **Type of memory** | **Accounted for** |
| Word16 buff[16] | static RAM | 16 |
| const Word32 tab[32] | ROM | 64 |
| void \*state | stack | push 1 |
| Word16 a | stack | push 1 |
| const Word16 v[] | stack | push 1 |
| Word16 ret | stack | push 1 |
| Word16 local\_buff[8] | stack | push 8 |
| static Word16 state | static RAM | 1 |
| Return | stack | pop (-12) |

Table 4: Example of memory assessment

* 1. Additional definitions

Static RAM array initialization

Arrays that are allocated and initialised in the static RAM are accounted simultaneously in static RAM and in ROM.

Stack array initialization

Arrays that are allocated and initialised in the stack are accounted only in static RAM. Furthermore, the code shall be instrumented with as many move16() (resp. move32()) basic operations than necessary in order to take into account the actual initialisation process. Here follows a small example:

Word16

func\_proc(Word16 a, Word32 b)

{

[...]

Word16 autoBuff[4]={0x4000, 0x1400, 0xFC00, 0xAFF0};

move16();move16();move16();move16();

[...]

return 0;

}

Code 2: Instrumented C-code initializing an array in the stack

Said differently, the process of initialising an array allocated in the stack is formally equivalent to the following C-code fragment:

Word16

func\_proc(Word16 a, Word32 b)

{

[...]

Word16 autoBuff[4];

autoBuff[0] = 0x4000; move16();

autoBuff[1] = 0x1400; move16();

autoBuff[2] = 0xFC00; move16();

autoBuff[3] = 0xAFF0; move16();

[...]

return 0;

}

Code 3: Unambiguous equivalent C-code for initializing an array in the stack

Constant value usage

Most C compilers for DSP will inline Word16 and Word32 constant values directly in the assembly language code. Therefore, constant values (such as 0x00400000L and 25798L) will not be included in the data ROM; instead they are included in the program source code.

Summary

The following table sums up the different configurations considered for assessing the complexity and the memory usage regarding the usage of constant values in the reference C-code.

|  |  |  |
| --- | --- | --- |
| **C instruction** | **Type of memory** | **Accounted for** |
| Word16 swRand[4]={…}; | ROM + static RAM | 4 each |
| Word16 autoBuff[4]={…}; | stack | push 4 |
| ((Word16)0x(vvvv)) | program | transparent |
| 0x(hhhhllll)L | program | transparent |

Table 4: Memory assessment for initialization of arrays and constant value usage

Example C-code

This following imaginary sample code (which does nothing in particular) illustrates different cases that shall be taken into account for the memory assessment of the audio codecs :

/\* initialization counting for 4 words in the ROM \*/

Word16 swRand[4] = {8, 12, -4, -7};

Word16

func\_proc(Word16 a, Word32 b)

{

Word16 idx, idx2;

/\* constant value counting for 0 words ROM \*/

Word32 enerLog = 0x00400000L;

/\* initialization counting for 0 word ROM \*/

Word16 autoBuff[4] = {0x4000, 0x1400, 0xFC00, 0xAFF0};

/\* enerLog initialization \*/

move32();

/\* autoBuff initialization \*/

move16();move16();move16();move16();

[...]

/\* loop preparation \*/

idx2 = 0; move16();

for (idx=0;idx<4;idx++) {

[...]

autoBuff [idx] = swRand[idx2]; move16();

swRand[idx2] = /\* small constant 25798L counting 0 word ROM \*/

extract\_h(L\_shr(L\_add(25798L,

L\_mult(swRand[idx2], 10037)),2));

move16();

[...]

}

[...]

return 0;

Code 4: Sample instrumented C-code

* 1. ROM verification

The source code is used to evaluate the ROM complexity.

* 1. RAM verification

Permanent RAM verification

The source code is used to evaluate the RAM usage that is not related to the use of the stack. The verification laboratory enumerates all the array and variable definitions corresponding to a permanent allocation.

Stack verification

The source code is used to evaluate the stack usage. The verification laboratory builds the calling tree of the source code and evaluates the worst case for the stack usage.

Conclusion

The verification laboratory sums the amount of static RAM and the maximum amount of RAM required by the stack.

1. **Stéphan Tassart**

   STMicroelectronics,

   Email: stephan.tassart@st.com [↑](#footnote-ref-2)