Coding Rules for AgALag-compliant AL3101 effects

G. Soyez (Axoris)

Version: 1.3
Creation date: 20/11/01
Last modification date: 11/11/2004

Contents

History of changes

Revision number	Editor	Description of changes
1.0	Lall	Initial revision
1.1	Lall	Added remarks from Gregor's review
1.2	Gregor	Compatibility with initial SF AgALag
1.3	Gregor	Synchronisation with VirtuAL3101-1.3

1 Introduction

This document describes some basic coding rules for developping AgALag-compliant modules. The final aim of these rules is to allow easy integration of codes written by different developpers and to allow building one module from different parts using AgALag. The rules are not intended to make life of the developper more complex however using these rules may add some extra complexity at the moment of writting. We have tried to make these rules as flexible as possible so that a non-compliant module can be turned into a compliant one as easily as possible.

Anyhow, the added value of these rules is multiple:

- it allows to build a library of re-usable code, which should make them a logical step into development,
- it is expected to lead to codes with a clear structure,
- it allows for a easy definition of "parameter" relevant to tune an effect.

In this document, the word *module* refers to sound processing modules like an equalizer as well as to basic building functions like biquad, sine generator, etc...

Some practical aspects of use of these rules are covered at the end of the document. We shall also give a tutorial explaining step-by-step how to code a simple distorsion.

2 Layout of a module

Basically, a module is build in three stages. First of all, one should start with a header (not at all related to the AL3101 assembler in itself) providing basic information. Then, the assembler for the module shall be written in two steps. First, we shall declare all addresses and particular constants needed. Then, the code of effect itself, using pre-defined labels.

From a practical point of view, starting your file with a short description enables one external user to re-use the module without necessarily knowing anything about the assembler in itself. The separation of the assembler in declaration and code is the basic fact allowing AgALag to create assembler for large structures from its building blocks.

In this chapter, we shall explain this structure in all details. We shall start by describing the header and then the assembler part.

2.1 Header of a module

The scheme for such a header is as follows:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Effect Name :
; Description :
;
; Nb Input    : number of inputs expected by the module
; Nb Output   : number of outputs generated by the module
;
; Input       :
; Output      :
;
; Cycles      : ? + <init?> cycles (-1=unknownw)
; Memory      : ? words (without INIT)
; Delay Line  : no/yes (remove wrong one)
;
; Author      :
; Date        :
; Version     :
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Each module should begin with some comment lines. The following information should be available from these lines:

1. The name and a short description of the module.
2. A descrition of the number of IOs the module is working with.
3. The memory and cycles requirements. The memory requirements is the number of addressed declared in the Control parameter and Memory variables sections (see

below). This means that your code uses memory addresses from 0 to (Memory-1). The cycles requirement is divided in two parts: first the number of code lines which is the number of assembler lines in the code section (see below) and the number of assembler lines in the initialisation section (see below again). This, of course, does not include the "pre-assembler commands" i.e. addresses, constants and labels definitions.

4. The need of a delay line. Actually, you should probably answer "no" to this question in most cases. If you want to use a delay line (i.e. a "memory segment"), you have the choice between two possibilities:

   (a) the delay line has fixed length (i.e. its length is not a parameter of the module). In that case, you just declare the address of first element of the segment and, when computing the memory requirements, you include the total number of elements required in your delay line. You put "no" to the Delay entry in the header.

   (b) the delay line has a length that may be modified from one usage to another. In that case, you only declare the first element of the section and you answer "yes" to the Delay entry in the header. When computing the memory requirements you just include the first element of the line.

   Note that, for rather complicated internal reasons, we only support one delay line which user-defined lenght per effect.

5. The name of the module developper.

6. The date and version number of the module.

2.2 Sections in a module

Each module should have the same layout and should contain different sections:

1. Section 1: Control parameters
2. Section 2: Initial values of control parameters
3. Section 3: Constants definitions
4. Section 4: Memory variables definition
5. Section 5: Initial values of memory variables
6. Section 6: Initialization
7. Section 7: Module code

Here is an in-depth description of all sections

2.2.1 Section 1: Control parameters.

Control paremeters are variables, coefficients, etc... that are supposed to be modifiables by a user. They are the *parameters* of the module. For example, it can be the value of a scaler, coefficients of a tunable equalizer, etc...

The definition of these parameters should be done by means of ABS statement. They will be stored in data RAM and can be used in the code by their name instread of their absolute value. The fixed address given to the ABS statement have to be consecutive numbers starting from 0.

2.2.2 Section 2: Initial values of control parameters.

The initial values of the control parameters defined in section 1 should be defined here. This is done with EQU statements. The name should be the same as the one used in section 1 but preceeded by an "I". These values will be used by the initialization section and can be considered as the *default values* of the parameter.

2.2.3 Section 3: Constants definitions.

Constants values are variables, coefficients, etc... that are not supposed to be modified after assembling. This can be a fixed amplification factor, the length of a delay line, a definition of π, etc...

These constants will be directly used in the code.

2.2.4 Section 4: Memory variables definition.

Memory variables are the same as control parameters except that they are reserved for internal use i.e. they are not supposed to be modified by a user. They must be declared using ABS statements and their absolute values must be consecutive and must follow the absolute values of the control parameters. These variables are directly used in the code.

IMPORTANT NOTE: These memory variables should be used for information that need to be kept from one cycle to another (*e.g.* previous input for an IIR filter). If you need some temporary memory for internal computation inside one cycle, please use the `DIRx` addresses.

2.2.5 Section 5: Initial values of memory variables.

In some cases, it might be useful to initialize the values of the memory variables to a known state. This works as for the second section: you just use the EQU statements. The name should be the same as the one used in section 4 but preceeded by an "I". It must be clear that only some of the values requires initialization it is thus not mandatory. These values will be used by the initialization section.

2.2.6 Section 6: Initialization.

The initialization of the control parameters and of the memory variables is done in this section. The INIT section should therefore look like this:

```
CM 0x40000 INIT
SKIP !Z EndInit
  ;; put allyour initialisation steps here under the form
  ;;C IParam
  ;;SCA 0x0 Param
C 0x1000000
SCA 0x0 INIT
EndInit:
```

Since the initialisation procedure has only to be done once at the beginning, we have already included 4 assembler lines which, using the INIT variable, ensure that the lines you will add won't be repeated each time. Also note that if you want to reinitialise the module, you just need to set INIT to 0 from outside.

2.2.7 Section 7: Module code.

This section will contain the "real" processing part of the module. Remarks:

- This part should NEVER use any fix reference to the data memory (addresses 0x0 to 0x3FF). You must always refer to these addresses using the memory labels defined in sections 1 and 4.

- The input (`IN1` to `IN8`) and output (`OUT1` to `OUT8`) addresses must be referred by their name and not by their address.
- If you need temporary memory for the purposes of a computation inside one cycle, you should use the `DIRx` addresses.
- If you want to use directly the address pointer for one of the memory labels `Label`, you can use `@Label`. Note that you should use this with great caution and only use it for modules entering in an AgALag structure, this will not work for a standalone application.

3 Tutorial

We shall explain step-by-step what we consider as being the best procedure to keep consistent with these coding rules in real-life examples. This do not cover all the possible cases but we hope that if these two examples are clearly understood, then you probably can go and see directly in the effect database. For these purposes, we indicate which effects can serve for particular requirements.

3.1 Simple example: writing a 2-to-1 mixer

... coming soon ...

3.2 A little bit harder: IIR filter of order 2

3.3 Further information

4 Pratical aspects of use

4.1 Loss of cycles in initialization

The number of cycles that will be spent in initialization may become quite huge if filter coefficients have to be initialized. In this case, it would be better to download the values directly in memory from an off-chip EEPROM or micro-controller. As a consequence, the Axoris mkalproj application[1] will provide an option to generate a second object file containing an image of the data memory at initialization time. If, of course, will take care of removing the initialization section from the program code.

[1] 'mkalproj' is the name of the program that actually puts everything togather to produce the final result.

A Module template

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Effect Name :
;
; Description :
;
; Nb In       :
; Nb Out      :
;
; Input       :
; Output      :
;
; Cycles      : ? + <init?> cycles (-1=unknownw)
; Memory      : ? words (without INIT)
; Delay Line  : no/yes (remove wrong one)
;
; Author      :
; Date        :
; Version     :
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ABS     INIT    0x3FF

        ;; SEC1 Control parameters

        ;; SEC2 Initial values of control parameters (S3.24 format)

        ;; SEC3 Constants definitions

        ;; SEC4 Memory variables definition

        ;; SEC5 Initial values of memory variables

        ;; SEC6 Initialization
CM      0x40000 INIT
SKIP    !Z      EndInit
<<insert control parameters init here >>
<<insert memory variables init here >>
<<C Value>>
<<SCA OxO Address>>
C       0x1000000           ; 1->A
```

```
SCA     0x0     INIT        ; do not reinitialize next time
EndInit:

        ;; SEC7 Module code
```