AASL Language Reference Manual

Group Members:
Vaishnav Janardhan, Rob Katz, Carlos Rene Perez, Albert Tsai

1. Introduction:

The purpose of AASL is to simplify the creation of original sounds by allowing users to easily generate and combine the fundamental elements of sound composition - oscillators, mixers, and envelopes - without needing to create their own data structures and methods to do so. In short, the goal is to present those elements as atomic data types which are then manipulated by intuitive operators.

AASL programs require a basic structure. They are divided into the following sections:

Header - Specifies the configuration of the output sound file

Main function - The first function to be executed
Declaration section
Connection section

User-defined function(s) - Helper functions that the user sees fit to create.
Declaration section
Connection section

The composition of these sections will be described in more detail later on.

1.1 Document Conventions

The word "elements" in this document will hereafter refer to "oscillators, oscillator banks, mixers, and envelopes."
Because AASL programs are structured, there are aspects of the language (statements, for one) which are only applicable to certain sections of a valid AASL program. These will be identified with tags such as [Declaration Section].

2. Lexical Conventions

A character stream first must be transformed into the valid "words" of a language; alternately stated, the atomic units of a valid AASL program. These units are called tokens.

2.1 Tokens
There are seven classes of tokens: identifiers, keywords, string literals, constants, operators, and grouping tokens (parentheses and curly braces). "White space", which includes spaces, tabs, newlines, and comments, is ignored except in as much as they separate tokens.

2.2 Comments
The language allows for both single-line and multi-line comments. Single line comments are initiated with a double-slash (//) and run until the next newline. Multi-line comments are initiated with a slash-star (/*) and terminate with the first occurrence of star-slash(*/).

2.3 Identifiers
Identifiers are sequences of letters, digits and underscores that begin with a letter. The language is case-sensitive, so 'word' and 'Word' represent two different identifiers. The maximum length of an identifier is not defined by the language.

2.4 Keywords

The following are reserved keywords and may not be used as identifiers.

mixer	start	end	header	func	connect	env	osc
def	oscbank	if	else	for	int	float	TONELENGTH
SINE	SQUARE	SAW	REVSAW	OUTPUT	SEGMENTS	SEG	

Keywords can be used for the following reasons:

- To define or create a data type (mixer, env, osc, oscbank, int, float)
- To declare a waveform (SINE, SQUARE, SAW, REVSAW)
- To define a statement or to be used as part of a statement (if, else, for)
- To define a function (func)
- To define a block (start, end, header, connect)
- To define language output parameters (TONELENGTH, OUTPUT, SEGMENTS, SEG)

2.5 String Literals
String literals are surround by double quotes, and are context limited to the header section as the assignment r-value of the OUTPUT keyword.

2.6 Constants
There are two types of constants allowed in AASL, integer-constants and float-constants. Both are numeric constants.

2.6.1 Integers
Integers are decimal whole numbers represented as a sequence of digits.

2.6.2 Floats
Floats in AASL are represented as a sequence of numbers with a single period. There is no support for exponent notation.

2.7 Operators
By and large, all of the following have the same meaning as one would expect. The period operator is similar to that found in most object oriented languages. In AASL, it is used in the context of an oscillator bank, which represents a group of related oscillators, to distinguish one from the other. For example, in a bank of called oscbank1 with two oscillators, they would be refered to as oscbank1[1] and oscbank1[2]
The individual elements of oscbank are referenced as it is done in C-Programming language. The i-th element is accessed by the operation <oscbank-obj>[i-1]

+ - * / % = ! < > <= >= == && || ++ -- [ ]

1.8 Other Tokens
( ) { }

Parenthesis are used in the following contexts.
- In the definition of an envelope, surrounding each time/amplitude pair.
- In connection expressions.
- Surrounding the conditions of an if or for statement.

Braces are used in the following contexts.
- Surrounding the argument list in a function declaration or call to a function, following the function name.
- To contain the bodies of code following conditionals.
- To contain a complete set of time/amplitude pairs in the definition of an envelope.

3. Meaning of Identifiers
Identifiers, or names, refer to functions and objects. Functions are the subroutines of a larger AASL

program; they exist for the user's convenience in organizing and programming AASL code and will be discussed in Section 7, Functions. Objects are named regions of storage containing instances of primitive data types such as integers and floats, or instances of the four fundamental data types of an AASL program: oscillators, oscillator banks, envelopes, and mixers.

3.1 Data types

The standard integers and floats are available in AASL programs. Additionally:

1. Oscillators - Oscillators output a signal in the form of an array of audio samples (44.1k per second). The size of this array is dependent on the declared length of the tone (44,100 * (length in seconds) elements). An oscillator has three defining characteristics, namely, a wave shape, a frequency, and an amplitude. The frequency and amplitude may be constant or controlled by another element.

2. Oscillator Banks - Oscillator banks are simply groups of oscillators. The intention is that the oscillators are somehow related and will often have their characteristics defined in some sort of conditional loop. Individual oscillators within an oscillator bank are referenced in C array notation. For example, given an oscillator bank name ob1 with three oscillators, the individual oscillators may be referenced as ob1[1], ob1[2], and ob1[3].

3. Envelopes - Envelopes consist of time-amplitude pairs that represent a graph over the entire tone. Each member of a pair is a float between 0.0 and 1.0, inclusive. Envelopes are "connected" to an oscillators frequency or amplitude, which then tracks the graph. For example:

We have an envelope with pairs (0.1, 0.0) , (0.5, 1.0) , (0.9, 0.5) connected to the amplitude input of an oscillator. One-tenth of the way through the tone, the oscillators amplitude will be zero. Half way through the tone, it will be at full volume. Nine-tenths of the way through the tone it will be back down to half amplitude. If no amplitude value is provided for time zero, as in our example, we assume a starting volume of zero with a linear increase to the first defined value. In our example, this value is 0.0, so the volume stays at zero for the first tenth of the tone. Between times 0.1 and 0.5, there is a linear increase in volume from 0.0 to 1.0. After time 0.9, the volume holds steady at 0.5.

If an envelope is attached to the frequency input, a central frequency must be supplied. This will be the oscillators output frequency when the envelope's value is 0.5. Similarly, if an oscillator is connected to another oscillator's frequency input, a central frequency must be provided.

4. Mixers - A mixer takes the outputs from multiple oscillators and sums them, generating a new output. The input oscillators may be mixed in unequal amounts, but the output of a mixer is never greater than the maximum output of a single oscillator. A mixer must have at least two inputs.

4. Expressions

Expressions are combinations of values, objects and operators that are evaluated according to the particular rules of **precedence** and **association** to return a value. Common locations of expressions in an AASL program are in conditional blocks, envelope definitions, indexes of oscillator banks, and connection statements.

The precedence of expression operators is organized in the following table as highest to lowest, top to bottom. Within each level of precedence, left or right associativity is specified to define the order of evaluation of operators with the same precedence.

Error handling such as overflow, divide by zero check, and other exceptions in expression evaluation is not defined by the language.

Post-fix Expression	Description	Associativity
( )	Grouping Operator	left-to-right
++ -- !	Increment/Decrement by 1 limited to **for** stmt use expands to ID=ID+1. The operand is an l-value and the result is an r-value, with the side-effect of incrementing/decrementing ID by value of 1 Unary Negation is applicable to an arithmetic type only, the result is boolean TRUE if the value of its operand compares equal to 0, and FALSE otherwise	right-to-left
* / %	Multiplication, division, modulus are only applicable to arithmetic types. The '/' symbol yields the quotient, and the '%' operator the remainder of the division of the first operand by the second. If the second operand to division or modulus is 0 the operation is undefined.	left-to-right
+ -	Addition and subtraction are only applicable to arithmetic types. The '+' operator is the sum of the operands, while the '-' operator is the difference of the operands.	left-to-right
< <= > >=	Relational Operators are only applicable to arithmetic types all yield TRUE if the specified relation is mathematically correct, otherwise FALSE. The symbol '<' represents less, '<=' less than or equal, '>' greater than, '>=' greater than or equal'.	left-to-right
== !=	Equality Operators are only applicable to arithmetic types and produce a boolean type of TRUE if the mathematical relation is correct, otherwise FALSE. The '==' symbol is "equal to" while '!=' is "not equal to".	left-to-right
&&	Logical AND are only applicable to boolean types and also produce a boolean type of TRUE if both of the operands are TRUE, otherwise FALSE.	left-to-right
||	Logical OR are only applicable to boolean types and also produce a boolean type of TRUE if one or both of the operands is TRUE, otherwise FALSE.	left-to-right
=	Assignment requires a modifiable (not constant) l-value as left operand and it cannot be a function or an oscillator block. The type of an assignment expression is that of its left operand and the value is the value stored in the left operand after the assignment has taken place.	right-to-left
,	Comma operator is used to separate expressions, they are evaluated in order from left-to-right.	left-to-right

5. Definitions

Definitions specify the interpretation given to identifiers**.**

5.1 Primitive data types

Integers and floats are declared in the following manner:

int *identifier* = integer ;

float *identifier* = float ;

There are four different types of definition statements, each pertaining to a different element.

5.2 Oscillator Definition

There are two flavors of oscillator definition. One is for stand-alone oscillators, and the other is for oscillators associated with an oscillator bank.

osc *identifier* = *waveshape* ;
oscbank[*oscillator_index*] = *waveshape* ;

waveshape is one of the following: SINE, SQUARE, SAW, REVSAW. These are the four possible waves that an oscillator can output.

5.3 Oscillator Bank definition

oscbank *identifier* = *value*

value is an integer representing the number of oscillators contained in the oscillator bank.

5.4 Envelope Definition

env *identifier* = {($time_1$, $value_1$) , ($time_2$, $value_2$), ... , ($time_n$, $value_n$)} ;

($time_k$, $value_k$) is a time-value pair. Each must be between 0.0 and 1.0. See the description of envelopes in the "Meaning of Identifiers" section. There must be at least one pair in each envelope definition.

5.5 Mixer Definition

mixer *identifier* = *value*

value is an integer representing the number of inputs into the mixer.

6. Statements

Statements are the smallest executable block of code in a programming language. They are logical sequences primarily executed to alter the state of a program (as represented by the values of the objects contained within it) and thus they define how a program behaves. In other words, statements do more than simply return a value (like expressions), but are not full blown functions.

6.1 Conditionals [Definition Section, Connection Section]

For loops and if/else statements may be used in both the definition and connect sections. In the definition section, the body of such loops may contain definitions or nested definition loops.

6.1.1 For loops:

for(*assignment; bool_expression ; delta)*
{
definitions | *nested conditionals*
}

assignment - must be of the form: *identifier = (int* | *float)*
The identifier represents a variable not yet defined in the program. It's scope lasts throughout the body of the following loop.
bool_expression - an expression that evaluates to true or false. See the section on "Expressions".
delta - must be in one of the following three forms: *identifier = expression* *identifier++* *identifier--*
The for loop in AASL acts as one would imagine it to. The assignment initiates a condition. Delta is performed at the end of each loop. Once delta has been performed, the condition_expression is checked and, if it is false, the loop is exited.

6.1.2 If/Else statements:

if(*bool_expression*)
{
definitions | *nested conditionals*
}
[else
{
definitions | *nested conditionals*
}]$_{opt}$

NOTE: The braces denote an optional section and are not ever present in an actual if/else block.

6.2 Connection Statements [Connection Section]

There are two kinds of connection statements. One kind involves oscillators. The other involves mixers.

6.2.1 Oscillator Connections:

oscillator(value1, value2) ;	assigns constant values to frequency and amplitude
oscillator(value1, {oscillator | *envelope* | *function_call}) ;*	assigns a constant value to frequency and assigns an element/function to control amplitude
oscillator({oscillator | *envelope* | *function_call}(central_frequency), value2) ;*	assigns an element/function to control frequency and a constant value to amplitude
oscillator({oscillator | *envelope* | *function_call}(central_frequency), {oscillator* | *envelope* | *function_call}) ;*	assigns elements/functions to control both frequency and amplitude.

NOTE: In the above descriptions, curly braces are used for grouping and would not appear in an actual connection statement. All parentheses shown, however, would appear.

oscillator - is either of the form *identifier* representing a stand-along oscillator or of the form *identifier*[*integer* | *identifier*] representing an oscillator element from a given oscillator bank.
value1 - is an integer or a float equal to the assigned frequency in Hz.

value 2 - is a float between 0.0 and 1.0 equal to the assigned amplitude relative to maximum volume of 1.0.
central_frequency - is an integer or float equal to the central frequency around which the attached element modulates the frequency. The maximum range of variation is a single octave, which corresponds to a minimum of half the central_frequency and a maximum of double the central frequency. As an example, if an oscillator o1 putting out a 1Hz SINE wave at full amplitude is attached to the frequency input of an oscillator o2 and a central frequency of 440 Hz is assigned, then the output of o2 will shift smoothly between 220 Hz and 880 Hz. The instantaneous frequency of the output of o2 will be 440Hz at every zero crossing of o1's sine wave.
function_call - as explained in the "Functions" section, all functions may be thought of as returning complex oscillators. That is, a function call may be used in the "connect" section wherever an oscillator could appear.

6.2.2 Mixer Connections:
mixer = term + term + ... + term ;

mixer - an identifier representing a mixer defined in the "def" section
term - of the form *factor * output* where *factor* is an integer or float and *output* is an oscillator or a function call.
The number of terms in the statement must equal the number of inputs assigned to the mixer in the "def" section. Each term represents an input. The inputs are mixed in ratios proportional to their factors.

Connection conditionals
These are identical to the "def" section conditionals described in section 3.2.1.2 with the exception that the bodies must be composed of connection statements and connection conditionals.

6.3 Output Equation:
OUTPUT = *mixer_connection* ;

mixer_connection - identical to the mixer connections described in 3.2.2.1.

7. Functions
Functions are smaller, self-contained portions of programs that perform specific tasks. When using real synthesis modules, elements are first designed then connected to one another in various ways. AASL functions reflect this practice by having a definition section and a connections section, followed by an OUTPUT assignment statement. Again, within each section, **different kinds of statements are allowed.** See Section 6, Statements, for more information.

start func *identifier*
    definition section
    connection section
    output statement
end func *identifier*

7.1 The Definition section
The definition section is where the synthesizer's elements are declared.

start def
  *(definitions | definition conditionals)**
end def

7.2 The Connection section
The second section is the "connect" section, which contains details about the element's characteristics and how elements are interconnected.

start connect
  *(connection statement | connection conditional)**

end connect

7.3 Return Value

The return value from any user defined function is a wave form, which can replace an oscillator. The output is defined as:

OUTPUT = *expression* ;

The expression must evaluate to a valid oscillator.

8. Scope

8.1 Blocks:

A *block* is a section of code surrounded by braces { }. Block defines the scope, visibility, of any declared variable.

8.2 Scope:

The *scope* of an identifier is the range of the program in which the declared identifier has meaning. Scope is determined by the location of the identifier's declaration.

8.2.1 Block Scope:

An identifier appearing within a block has *block scope* and is visible within the block, unless hidden by an inner block declaration. Block scope begins at the opening brace ({) and ends at the closing brace (}) completing the block. If there is an identifier with the same variable name in a block above the defined block. Then the identifier with in the innermost block takes precedence.

8.2.2 Function Scope:

An identifier declared within a function has function scope and it should be unique throughout the function in which it is declared.

9. Anatomy of an AASL Program

Every AASL program consists of a header section and a main function followed by optional user-defined functions.

9.1 Header Section

An AASL program begins with a short header section. Every header begins with "start header" and ends with "end header." Within the section, one assigns the name of the output file, the length in seconds of the tone to be generated, and optional assignment of the number of segments the tone is to broken up into. This segmentation is used to alter the tone as time progresses through the use of conditionals.

-Output Assignment
   OUTPUT =" *filename*";
filename is the name of the file to which the generated tone will be written.

-Tonelength Assignment
   TONELENTGH = *value*;
value is an integer of float that gives the desired length of the tone in seconds

-Segment Assignment (optional)
   SEGMENTS = *value*;
value is an integer representing the number of equal-length time slices into which the tone will be split. Associated with the number of segments is a keyword SEG which evaluates to an integer representing whichever segment the tone is currently in. The idea of segments is used in writing conditional statements. This is best understood through an example.
Assume our header contains the following lines:

```
TONELENGTH = 10;
SEGMENTS = 5;
```

We may now view our tone as being divided into 5 two-second sections. If we want an oscillator, o1, to vary between SINE and SQUARE wave output, alternating every two seconds. We could write the following in our "def" section:

```
if(SEG % 2 == 0)
    {o1 = SINE;}
else
   {o1 = SQUARE;}
```

9.2 Main Function
All AASL programs must have a main function, which simply follows the form of all functions, except that it has the identifier "main". It must follow the Header section.

9.3 User Defined Functions
User defined functions can have any valid identifier, and they must follow the main function.

10. Sample Program
```
/* Header Section */
start header
OUTPUT = "testsound.wav"
TONELENGTH = 10 //10 second sound
SEGMENTS = 5
end header

/* Main Function */
start func main()

 /* Definition section */
   start def

      mixer S1 = 2; //defines a mixer for 2 oscillators to attach to
      mixer S2 = 2;

      osc o1 = SINE; //defines a sine wave oscillator, o1
      osc o2 = SQUARE;
      oscbank ob1 = 3; //bank of three oscillators
      if(SEG % 2 == 0) //set waveform of oscbank elements depending on which
      {
         //time segment you're in
        ob1[1] = SQUARE;
```

```
          ob1[2] = SINE;
        }
        else
        {
          ob1[1] = SINE;
          ob1[2] = SAW;
        }

        ob1[3] = SINE; //third osc in the bank does not depend on SEGMENT

        env e1 = { (0.0,0.3) (0.2,0.2) (0.4,0.2) (0.8,0.8) (1.0,0) };
        //defines envelope w/ (time, amp)
        env e2 = { (0.0,0.0) (0.3,0.0) (1.0, 1.0) };

      end def //end definitions

    /* Connection Section *
    start connect

      o1(440, e1);
      o2(e2(1000), ob1[2]); //second oscillator in oscbank controls amplitude
      for(x=1; x<ob1.size; x++) //for loop featuring .size feature (returns # of osc in
      {
        //oscbank
        if(x==1 || SEG==2 || SEG==5)
        {
          ob1[x](440, 0.5); //apostrophes differentiate variables
        }
        else
        {
            ob1[x](220*x, ob1[x-1]);
        }
      }

      S1 = 0.4*o1 + 0.6*o2;
      S2 = 2*S1 + ob1[1] + 0.5*ob1[2] + 0.25ob1[3] + 0.25*o2o{1000};
      //note one of the ouput signals is function call

    end connect

    /* Output Section */
    OUTPUT = S1 + o5;

  end func main

  /* User defined funtion...takes one integer as argument */
  start func o2o (int x)

    start def
      osc o1(SAW);
      osc o2(SINE);
    end def

    start connect
      o1(x, o2);
      o2(0.5*SEG, .75);
    end connect

    OUTPUT = o1;
```

```
  end func o2o

   /* End of Sample Program */

/*Notes, questions*/
```