ISO/IEC JTC 1/SC 29/WG 11 **N2503-sec5**

Date: 1999-3-10

ISO/IEC FDIS 14496-3 sec5

ISO/IEC JTC 1/SC 29/WG11

Secretariat: Narumi Hirose

Information technology - Coding of audio-visual objects

Part 3: Audio

Section 5: Structured audio

Contents

Figures

Tables

Section 5: Structured audio

5.1 Scope

5.1.1 Overview of section

5.1.1.1 Purpose

The Structured Audio toolset enables the transmission and decoding of synthetic sound effects and music by standardising several different components. Using Structured Audio, high-quality sound can be created at extremely low bandwidth. Typical synthetic music may be coded in this format at bitrates ranging from 0 kbps (no continuous cost) to 2 or 3 kbps for extremely subtle coding of expressive performance using multiple instruments.

MPEG-4 does not standardise a particular set of synthesis methods, but a method for describing synthesis methods. Any current or future sound-synthesis method may be described in the MPEG-4 Structured Audio format.

5.1.1.2 Introduction to major elements

There are five major elements to the Structured Audio toolset:

1. The Structured Audio Orchestra Language, or SAOL. SAOL is a digital-signal processing language which allows for the description of arbitrary synthesis and control algorithms as part of the content bitstream. The syntax and semantics of SAOL are standardised here in a normative fashion.
2. The Structured Audio Score Language, or SASL. SASL is a simple score and control language which is used in certain object types (see clause 5.6) to describe the manner in which sound-generation algorithms described in SAOL are used to produce sound.
3. The Structured Audio Sample Bank Format, or SASBF. The Sample Bank format allows for the transmission of banks of audio samples to be used in wavetable synthesis and the description of simple processing algorithms to use with them.
4. A normative scheduler description. The scheduler is the supervisory run-time element of the Structured Audio decoding process. It maps structural sound control, specified in SASL or MIDI, to real-time events dispatched using the normative sound-generation algorithms.
5. Normative reference to the MIDI standards, standardised externally by the MIDI Manufacturers Association. MIDI is an alternate means of structural control which can be used in conjunction with or instead of SASL. Although less powerful and flexible than SASL, MIDI support in this standard provides important backward-compatibility with existing content and authoring tools. MIDI support in this standard consists of a list of recognised MIDI messages and normative semantics for each.

5.2 Normative references

[DLS] (c) 1997 MIDI Manufacturers Association, *The MIDI Downloadable Sounds Specification, v. 97.1.*

[DLS2] (c) 1998 MIDI Manufacturers Association, *The MIDI Downloadable Sounds Specification, v. 98.2*.

[MIDI] (c) 1996 MIDI Manufacturers Association, *The Complete MIDI 1.0 Detailed Specification v. 96.2.*

5.3 Definitions

5.3.1 Absolute time: The time at which sound corresponding to a particular event is really created; time in the real-world. Contrast score time.

5.3.2 Actual parameter: The expression which, upon evaluation, is passed to an opcode as a parameter value.

5.3.3 A-cycle: See **audio cycle**.

5.3.4 A-rate: See **audio rate**.

5.3.5 asig: The lexical tag indicating an a-rate variable.

5.3.6 Audio cycle: The sequence of processing which computes new values for all a-rate expressions in a particular code block.

5.3.7 Audio rate: The rate type associated with a variable, expression or statement which may generate new values as often as the sampling rate.

5.3.8 Audio sample: A short snippet or clip of digitally represented sound. Typically used in wavetable synthesis.

5.3.9 Authoring: In Structured Audio, the combined processes of creatively composing music and sound control scripts, creating instruments which generate and alter sound, and encoding the instruments, control scripts, and audio samples in MPEG-4 Structured Audio format.

5.3.10 Backus-Naur Format: (**BNF**) A format for describing the syntax of programming languages, used here to specify the SAOL and SASL syntax. See subclause 5.4.2.2.

5.3.11 Bank: A set of samples used together to define a particular sound or class of sounds with wavetable synthesis.

5.3.12 Beat: The unit in which score time is measured.

5.3.13 BNF: See Backus-Naur Format**.**

5.3.14 Bus: An area in memory which is used to pass the output of one instrument into the input of another.

5.3.15 Context: See state space.

5.3.16 Control: An instruction used to describe how to use a particular synthesis method to produce sound.

EXAMPLES

“Using the piano instrument, play middle C at medium volume for 2 seconds.”
“Glissando the violin instrument up to middle C.”
“Turn off the reverberation for 8 seconds.”

5.3.17 Control cycle: The sequence of processing which computes new values for all control-rate expressions in a particular code block.

5.3.18 Control period: The length of time (typically measured in audio samples) corresponding to the control rate.

5.3.19 Control rate: (1) The rate at which instantiation and termination of instruments, parametric control of running instrument instances, sharing of global variables, and other non-sample-by-sample computation occurs in a particular orchestra. (2) The rate type of variables, expressions, and statements that can generate new values as often as the control rate.

5.3.20 Decoding: The process of turning an MPEG-4 Structured Audio bitstream into sound.

5.3.21 Duration: The amount of time between instantiation and termination of an instrument instance.

5.3.22 Encoding: The process of creating a legal MPEG-4 bitstream, whether automatically, by hand, or using special authoring tools.

5.3.23 Envelope: A loudness-shaping function applied to a sound, or more generally, any function controlling a parametric aspect of a sound

5.3.24 Event: One control instruction.

5.3.25 Expression: A mathematical or functional combination of variable values, symbolic constants, and opcode calls.

5.3.26 Formal parameter: The syntactic element that gives a name to one of the parameters of an opcode.

5.3.27 Future wavetable: A wavetable that is declared but not defined in the SAOL orchestra; its definition must arrive in the bitstream before it is used.

5.3.28 Global block: The section of the orchestra that describes global variables, route and send statements, sequence rules, and global parameters.

5.3.29 Global context: The state space used to hold values of global variables and wavetables.

5.3.30 Global parameters: The sampling rate, control rate, and number of input and output channels of audio associated with a particular orchestra.

5.3.31 Global variable: A variable that can be accessed and/or changed by several different instruments.

5.3.32 Grammar: A set of rules that describes the set of allowable sequences of lexical elements comprising a particular language.

5.3.33 Guard expression: The expression standing at the front of an if, while, or else statement that determines whether or how many times a particular block of code is executed.

5.3.34 I-cycle: See initialisation cycle**.**

5.3.35 Identifier: A sequence of characters in a textual SAOL program that denotes a symbol.

5.3.36 Informative: Aspects of a standards document that are provided to assist implementers, but are not required to be implemented in order for a particular system to be compliant to the standard.

5.3.37 I-pass: See initialisation pass**.**

5.3.38 I-rate: See initialisation rate.

5.3.39 Initialisation cycle: See initialisation pass**.**

5.3.40 Initialisation rate: The rate type of variables, expressions, and statements that are set once at instrument instantiation and then do not change.

5.3.41 Initialisation pass: The sequence of processing that computes new values for each **i-**rate expression in a particular code block.

5.3.42 Instance: See instrument instantiation.

5.3.43 Instantiation: The process of creating a new instrument instantiation based on an event in the score or statement in the orchestra.

5.3.44 Instrument: An algorithm for parametric sound synthesis, described using SAOL. An instrument encapsulates all of the algorithms needed for one sound-generation element to be controlled with a score.

NOTE - An MPEG-4 Structured Audio instrument does not necessarily correspond to a real-world instrument. A single instrument might be used to represent an entire violin section, or an ambient sound such as the wind. On the other hand, a single real-world instrument that produces many different timbres over its performance range might be represented using several SAOL instruments.

5.3.45 Instrument instantiation: The state space created as the result of executing a note-creation event with respect to a SAOL orchestra.

5.3.46 ivar: The lexical tag indicating an i-rate variable.

5.3.47 K-cycle: See control cycle.

5.3.48 K-rate: See control rate.

5.3.49 ksig: The lexical tag indicating a k-rate variable.

5.3.50 Lexical element: See token.

5.3.51 Looping: A typical method of wavetable synthesis. Loop points in an audio sample are located and the sound between those endpoints is played repeatedly while being simultaneously modified by envelopes, modulators**,** etc.

5.3.52 MIDI: The Musical Instrument Digital Interface standards, see **[MIDI]** in subclause 5.2. MIDI is one method for specifying control of synthesis in MPEG-4 Structured Audio.

5.3.53 Natural Sound: A sound created through recording from a real acoustic space. Contrasted with synthetic sound.

5.3.54 Normative: Those aspects of a standard that must be implemented in order for a particular system to be compliant to the standard.

5.3.55 Opcode: A parametric signal-processing function that encapsulates a certain functionality so that it may be used by several instruments.

5.3.56 Orchestra: The set of sound-generation and sound-processing algorithms included in an MPEG-4 bitstream. Includes instruments, opcodes, routing, and global parameters.

5.3.57 Orchestra cycle: A complete pass through the orchestra, during which new instrument instantiations are created, expired ones are terminated, each instance receives one k-cycle and one control period worth of a-cycles, and output is produced.

5.3.58 Parameter fields: The names given to the parameters to an instrument.

5.3.59 P-fields: See parameter fields.

5.3.60 Production rule: In Backus-Naur Form grammars, a rule that describes how one syntactic element may be expressed in terms of other lexical and syntactic elements.

5.3.61 Rate-mismatch error: The condition that results when the **r**ate semantics rules are violated in a particular SAOL construction. A type of syntax error.

5.3.62 Rate semantics: The set of rules describing how rate types are assigned to variables, expressions, statements, and opcodes, and the normative restrictions that apply to a bitstream regarding combining these elements based on their rate types.

5.3.63 Rate type: The “speed of execution” associated with a particular variable, expression, statement, or opcode.

5.3.64 Route statement: A statement in the global block that describes how to place the output of a certain set of instruments onto a bus.

5.3.65 Run-time error: The condition that results from improper calculations or memory accesses during execution of a SAOL orchestra.

5.3.66 SASBF: See Sample Bank Format

5.3.67 SAOL: The Structured Audio Orchestra Language, pronounced like the English word “sail.” SAOL is a digital-signal processing language that allows for the description of arbitrary synthesis and control algorithms as part of the content bitstream.

5.3.68 SAOL orchestra: See orchestra.

5.3.69 SASL: The Structured Audio Score Language. SASL is a simple format that allows for powerful and flexible control of music and sound synthesis.

5.3.70 Sample: See Audio sample.

5.3.71 Sample Bank Format: A component format of MPEG-4 Structured Audio that allows the description of a set of samples for use in wavetable synthesis and processing methods to apply to them.

5.3.72 Scheduler: The component of MPEG-4 Structured Audio that describes the mapping from control instructions to sound synthesis using the specified synthesis techniques. The scheduler description provides normative bounds on event-dispatch times and responses.

5.3.73 Scope : The code within which access to a particular variable name is allowed.

5.3.74 Score: A description in some format of the sequence of control parameters needed to generate a desired music composition or sound scene. In MPEG-4 Structured Audio, scores are described in SASL and/or MIDI.

5.3.75 Score time: The time at which an event happens in the score, measured in beats. Score time is mapped to absolute time by the current tempo.

5.3.76 Send statement: A statement in the global block that describes how to pass a bus on to an effect instrument for post-processing.

5.3.77 Semantics: The rules describing what a particular instruction or bitstream element should do. Most aspects of bitstream and SAOL semantics are normative in MPEG-4.

5.3.78 Sequence rules: The set of rules, both default and explicit, given in the global block that define in what order to execute instrument instantiations during an orchestra cycle.

5.3.79 Signal variable: A unit of memory, labelled with a name, that holds intermediate processing results. Each signal variable in MPEG-4 Structured Audio is instantaneously representable by a 32-bit floating point value.

5.3.80 Spatialisation: The process of creating special sounds that a listener perceives as emanating from a particular direction.

5.3.81 State space: A set of variable-value associations that define the current computational state of an instrument instantiation or opcode call. All the “current values” of the variables in an instrument or opcode call.

5.3.82 Statement: “One line” of a SAOL orchestra.

5.3.83 Structured audio: Sound-description methods that make use of high-level models of sound generation and control. Typically involving synthesis description, structured audio techniques allow for ultra-low bitrate description of complex, high-quality sounds. See **[SAUD]**.

5.3.84 Symbol: A sequence of characters in a SAOL program, or a symbol token in a MPEG-4 Structured Audio bitstream, that represents a variable name, instrument name, opcode name, table name, bus name, etc.

5.3.85 Symbol table: In an MPEG-4 Structured Audio bitstream, a sequence of data that allows the tokenised representation of SAOL and SASL code to be converted back to a readable textual representation. The symbol table is an optional component.

5.3.86 Symbolic constant: A floating-point value explicitly represented as a sequence of characters in a textual SAOL orchestra, or as a token in a bitstream.

5.3.87 Syntax: The rules describing what a particular instruction or bitstream element should look like. All aspects of bitstream and SAOL syntax are normative in MPEG-4.

5.3.88 Syntax error: The condition that results when a bitstream element does not comply with its governing rules of syntax.

5.3.89 Synthesis: The process of creating sound based on algorithmic descriptions.

5.3.90 Synthetic Sound: Sound created through synthesis.

5.3.91 Tempo: The scaling parameter that specifies the relationship between score time and absolute time. A tempo of 60 beats per minute means that the score time measured in beats is equivalent to the absolute time measured in seconds; higher numbers correspond to faster tempi, so that 120 beats per minute is twice as fast.

5.3.92 Terminal: The “client side” of an MPEG transaction; whatever hardware and software are necessary in a particular implementation to allow the capabilities described in this document.

5.3.93 Termination: The process of destroying an instrument instantiation when it is no longer needed.

5.3.94 Timbre: The combined features of a sound that allow a listener to recognise such aspects as the type of instrument, manner of performance, manner of sound generation, etc. Those aspects of sound that distinguish sounds equivalent in pitch and loudness.

5.3.95 Token: A lexical element of a SAOL orchestra: a keyword, punctuation mark, symbol name, or symbolic constant.

5.3.96 Tokenisation: The process of converting a orchestra in textual SAOL format into a bitstream representation consisting of a stream of tokens.

5.3.97 Variable: See signal variable.

5.3.98 Wavetable synthesis: A synthesis method in which sound is created by simple manipulation of audio samples, such as looping, pitch-shifting, enveloping, etc.

5.3.99 Width: The number of channels of data that an expression represents.

5.4 Symbols and abbreviations

5.4.1 Mathematical operations

The mathematical operators used to describe this part of ISO/IEC 14496 are similar to those used in the C programming language.

+	addition
-	subtraction
x or *	multiplication
/	division
exp	exponential function (base *e*)
log	natural logarithm
log10	base-10 logarithm
abs	absolute value
floor(x)	greatest integer less than or equal to x
ceil(x)	least integer greater than or equal to x
>	greater than

< less than
>= greater than or equal to
<= less than or equal to
<> or != not equal to

5.4.2 Description methods

5.4.2.1 Bitstream syntax

The Structured Audio bitstream syntax is described using SDL, the MPEG-4 Syntactic Description Language. See ISO/IEC 14496-1 clause 12.

5.4.2.2 SAOL syntax

The textual SAOL syntax (in clause 5.8) is described using extended Backus-Naur format (BNF) notation [see **DRAG**]. BNF is a description for context-free grammars of programming languages.

BNF grammars are composed of terminals, also called tokens, and production rules. Terminals represent syntactic elements of the language, such as keywords and punctuation; production rules describe the composition of these elements into structural groups.

Terminals will be represented in **boldface**; production rules will be represented in <angle brackets>.

The rewrite rules, which map productions into sequences of other productions and terminals, are represented with the -> symbol.

EXAMPLE

<letter> -> **a**
<letter> -> **b**
<sequence> -> <letter>
<sequence> -> <letter> <sequence>

This grammar (starting from the *sequence* token) describes, using a recursive rewrite rule and a two-symbol alphabet, all strings containing at least one letter that are made up of ‘a’ and ‘b’ characters.

In addition, rewrite rules using optional elements will be described using the [ ] symbols. Using this notation does not increase the power of the syntax description (in terms of the languages it can represent), but makes certain constructs simpler.

EXAMPLE

<head> -> **c**
<seqhead> -> [<head>] <sequence>

This grammar (starting from the seqhead token) describes, in addition to the set above, all strings beginning with a ‘c’ character and followed by a sequence of ‘a’s and ‘b’s.

The **NULL** token may be used to indicate that a sequence of no characters (the empty string) is a permissible rewrite for a particular production.

Other symbols such as the ellipsis (...) will be used occasionally when their meaning is clear from the context.

Normative aspects of the relationship between the BNF grammar, other grammar representation methods, the bitstream syntax, and the textual description format are described in subclause 5.8.1.

5.4.2.3 SASL Syntax

The SASL syntax is specified using extended BNF grammars, as described in subclause 5.4.2.2.

5.5 Bitstream syntax and semantics

5.5.1 Introduction to bitstream syntax

This subclause describes the bitstream format defining an MPEG-4 Structured Audio bitstream.

Each group of classes is notated with normative semantics, which define the meaning of the data represented by those classes.

5.5.2 Bitstream syntax

```
/*********************************
      symbol table definitions
*********************************/

class symbol {
  unsigned int(16) sym;         // no more than 65535 symbols/orch + score
}

class sym_name {                // one name in a symbol table
  unsigned int(4) length;       // names up to 15 chars long
  unsigned int(8) name[length];
}

class symtable {                // a whole symbol table
  unsigned int(16) length;      // no more than 65535 symbols/orch+score
  sym_name name[length];
}
```

A bitstream may contain a symbol table, but this is not required. The symbol table allows textual SAOL and SASL code to be recovered from the tokenised bitstream representation. The inclusion or exclusion of a symbol table does not affect the decoding process.

If a symbol table is included, then all or some of the symbols in the orchestra and score shall be associated with a textual name in the following way: each symbol (a symbol is just an integer) shall be associated with the character string paired with that symbol in a `sym_name` object. There shall be no more than one name associated with a given symbol, otherwise the bitstream is invalid. It is permissible for the symbol table to be incomplete and contain names associated with some, but not all, symbols used in the orchestra and score.

SAOL and SASL implementations that require textual input, rather than tokenised input, are permissible in a compliant decoder, in which case the decoder would detokenise the bitstream before it can be processed. In such a case, any symbols without associated names are suggested to be associated with a default name of the form `_sym_x`, where x is the symbol value. Names of this form are reserved in SAOL for this purpose, and so following this suggestion guarantees that names will not clash with symbol-table-defined symbol names.

```
/*********************************
    orchestra file definitions
*********************************/

class orch_token {              // a token in an orchestra
  int done;

  unsigned int(8) token;         // see standard token table, Annex 5.A
  switch (token) {
  case 0xF0 :                   // a symbol
    symbol sym;                 // the symbol name
    break;
  case 0xF1 :                   // a constant value
    float(32) val;              // the floating-point value
    break;
  case 0xF2 :                    // a constant int value
    unsigned int(32) val;       // the integer value
    break;
  case 0xF3 :                   // a string constant
```

```
    int(8) length;
    unsigned int(8) str[length]; // strings no more than 255 chars
    break;
  case 0xF4 :                    // a one-byte constant
    int(8) val;
    break;
  case 0xFF : // end of orch
    done = 1;
    break;
  }
}

class orc_file {                 // a whole orch file
  unsigned int(16) length;
  orch_token data[length];
}
```

An orchestra file is a string of tokens. These tokens represent syntactic elements such as reserved words, core opcode names, and punctuation marks as given in the table in Annex 5.A; in addition, there are five special tokens. Token 0xF0 is the symbol token; when it is encountered, the next 16 bits in the bitstream shall be a symbol number. Token 0xF1 is the value token; when it is encountered, the next 32 bits in the bitstream shall be a floating-point value. This token shall be used for all symbolic constants within the SAOL program except for those encountered in special integer contexts, as described in clause 5.12. Token 0xF2 is the integer token; when it is encountered, the next 32 bits in the bitstream shall be an unsigned integer value. Token 0xF3 is the string token; when it is encountered, the next several bits in the bitstream shall represent a character string (this token is currently unused). Token 0xF4 is the byte token; when it is encountered, the next 8 bits in the bitstream shall be an unsigned integer value. Token 0xFF is the end-of-orchestra token; this token has no syntactic function in the SAOL orchestra, but signifies the end of the orchestra file section of the bitstream.

Not every sequence of tokens is permitted to occur as an orchestra file. Clause 5.8 contains extensive syntactic rules restricting the possible sequence of tokens, described according to the textual SAOL format. Normative rules for mapping back and forth between the tokenised format and the textual format are given in clause 5.12. The overall sequence of orchestra tokens shall correspond to an <orchestra> production as given in subclause 5.8.4.

```
/*********************************
      score file definitions
*********************************/

class instr_event {              // a note-on event
  bit(1) has_label;
  if (has_label)
    symbol label;
  symbol iname_sym;              // the instrument name
  float(32) dur;                 // note duration
  unsigned int(8) num_pf;
  float(32) pf[num_pf];          // all the pfields (no more than 255)
}

class control_event {            // a control event
  bit(1) has_label;
  if (has_label)
    symbol label;
  symbol varsym;                 // the controller name
  float(32) value;               // the new value
}

class table_event {
  symbol tname;       // the name of the table
  bit(1) destroy;     // a table destructor
  if (!destroy) {
    token tgen;         // a core wavetable generator
    bit(1) refers_to_sample;
    if (refers_to_sample)
      symbol table_sym;          // the name of the sample
    unsigned int(16) num_pf;     // the number of pfields
```

```
    float(32) pf[num_pf];        // all the pfields
  }
}

class end_event {
  // fixed at nothing
}

class tempo_event {  // a tempo event
  float(32) tempo;
}

class score_line {
  bit(1) has_time;
  if (has_time) {
     bit (1) use_if_late;
     float(32) time;              // the event time
  }
  bit (1) high_priority;
  bit(3) type;
  switch (type) {
    case 0b000 : instr_event inst; break;
    case 0b001 : control_event control; break;
    case 0b010 : table_event table; break;
    case 0b100 : end_event end; break;
    case 0b101 : tempo_event tempo; break;
  }
}

class score_file {
  unsigned int(20) num_lines;  // a whole score file
  score_line lines[num_lines];
}
```

A score file is a set of lines of score information provided in the stream information header. Thus, events that are known before the real-time bitstream transmission begins may be included in the header, so that they are available to the decoder immediately, which may aid efficient computation in certain implementations. Each line shall be one of five events. Each type of event has different implications in the decoding and scheduling process, see subclause 5.7.3. An instrument event specifies the start time, instrument name symbol, duration, and any other parameters of a note played on a SAOL instrument. A control event specifies a control parameter that is passed to a instrument or instruments already generating sound. A table event dynamically creates or destroys a global wavetable in the orchestra. An end event signifies the end of orchestra processing. A tempo event dynamically changes the tempo of orchestra playback.

A score file need not be presented in increasing order of event times; the events shall be “sorted” by the scheduler as they are processed. In the score file, every score line shall have a time stamp (**has_time** shall be 1).

The **high_priority** bit indicates that the score event is a high-priority event as described in subclause 5.7.3.3.7. The **use_if_late** bit indicates, if the **has_time** bit is set, that the score event shall be used whether or not it arrives on time (see subclause 5.7.3.3.8).

```
/*******************************
         MIDI definitions
*******************************/

class midi_event {
  unsigned int(24) length
  unsigned int(8) data[length];
}

class midi_file {
  unsigned int(32) length;
  unsigned int(8) data[length];
}
```

The MIDI chunks allow the inclusion of MIDI score information in the bitstream header and bitstream. The MIDI event class contains a single MIDI instruction as specified in **[MIDI]**; the MIDI file class contains an array of bytes corresponding to a Standard Format 0 or Format 1 MIDIFile as specified in **[MIDI]**. Note that not every sequence of data may occur in either case; the legal syntaxes of MIDI events and MIDIFiles as specified in **[MIDI]** place normative bounds on syntactically valid MPEG-4 Structured Audio bitstreams. Only chunks of data up to 2^{24}-1 and 2^{32}-1 bytes long, respectively, may be included; longer messages shall be broken into several bitstream elements. The semantics of MIDI data are given in clause 5.13 (for Object type 1 and 2 implementations) and clause 5.14 (for Object type 3 and 4 implementations).

```
/*********************************
         sample data
*********************************/

class sample {
  /* note that 'sample' can be used for any big chunk of data
     that needs to get into a wavetable */
  symbol sample_name_sym;
  unsigned int(24) length;  // length in samples
  bit(1) has_srate;
  if (has_srate)
    unsigned int(17) srate; // sampling rate (needs to go to 96 KHz)
  bit(1) has_loop;
  if (has_loop) {
    unsigned int(24) loopstart; // loop points in samples
    unsigned int(24) loopend;
  }
  bit(1) has_base;
  if (has_base)
    float(32) basecps;         // base freq in Hz
  bit(1) float_sample;
  if (float_sample) {
    float(32) float_sample_data[length]; // data as floats ...
  }
  else {
    int(16) sample_data[length]; // ... or as ints
  }
}
```

A sample chunk includes a block of data that will be included in a wavetable in a SAOL orchestra. Each sample consists of a name, a length, a block of data, and four optional parameters: the sampling rate, the loop start and loop end points, and the base frequency. Access to the data in the sample is provided through the **sample** core wavetable generator, see subclause 5.10.2.

The sample data may be represented either as 32-bit floating point values, in which case it shall be scaled between –1 and 1, or may be represented as 16-bit integer values, in which case it shall be scaled between -32768 and 32767. In the case that the sample data is represented as integer values, upon inclusion in a wavetable, it shall be rescaled to floating-point as described in subclause 5.10.2.

Each sample is named with a symbol. If two samples in the decoder configuration header or in a single access unit have the same name, the result is unspecified. If a sample in an access unit has the same name as a sample in a previous access unit or one in the decoder configuration header, the new sample shall replace the old sample for accesses to that name through the **sample** core wavetable generator for any table generator executed at the same time or later than the decoding time of the access unit containing the new sample. Tables that have already been generated are not affected.

```
/************************
       sample bank data
************************/
```

The sample bank chunk describes a bank of wavetable data and associated processing parameters for use with the sample bank synthesis procedure in clause 5.13.

```
class sbf {
```

```
      int(32)            length;
      int(8)             data[length];
}
```

The data chunk is opaque with regard to transport by MPEG-4 Systems. It shall conform to the format specification given in [**DLS**] (see clause 5.2) – that is, it shall be a RIFF data chunk beginning “RIFF...”.

```
/*********************************
         bitstream formats
*********************************/

class StructuredAudioSpecificConfig {  // the bitstream header
  bit more_data = 1;

  while (more_data) {  // shall have at least one chunk
    bit(3) chunk_type;
    switch (chunk_type) {
    case 0b000 : orc_file orc; break;
    case 0b001 : score_file score; break;
    case 0b010 : midi_file SMF; break;
    case 0b011 : sample samp; break;
    case 0b100 : sbf sample_bank; break;
    case 0b101 : symtable sym; break;
    }
    bit(1) more_data;
  }
}
```

The bitstream decoder configuration contains all the information required to configure and start up a structured audio decoder. It contains a sequence of one or more chunks, where each chunk is of one of the following types: orchestra file, score file, midi file, sample data, sample bank, or symbol table. Multiple chunks of each of these types may occur in the bitstream (except for **midi_file**), with the following semantics:

1. **orc_file**: The multiple orchestra files shall be merged. It is a syntax error if more than one **global** block appears in the merged orchestra (see subclause 5.8.5).
2. **score_file**: The multiple score files shall be sorted together by event times and merged.
3. **midi_file**: Only one **midi_file** element may occur in a single Structured Audio bitstream.
4. **sample**: The samples may be accessed by the orchestra as described in subclause 5.10.2.
5. **sbf**: The multiple sample banks are all accessible to the synthesis process. The behaviour is undefined if a particular combination of MIDI present and MIDI bank number is used more than once, whether in a single sample bank or in multiple sample banks.
6. **symtable**: The multiple symbol tables each give names to symbols in the orchestra. The N_0 names in the first symbol table in the bitstream apply to symbols 0..N_0-1; the N_1 names in the second symbol table to symbols N_0..N_0+N_1-1; and so on.

```
class SA_access_unit {      // the streaming data
 bit(1) more_data = 1;

 while (more_data) {
   bit(2) event_type;
   switch (event_type) {
     case 0b00 : score_line score_ev; break;
     case 0b01 : midi_event midi_ev; break;
     case 0b10 : sample samp; break;
     }
   bit(1) more_data;
   }
}
```

The Structured Audio access unit contains real-time streaming control information to be provided to a running Structured Audio decoding process. It may contain as many control instructions as desired and as permitted by the

available bandwidth. It shall not contain new instrument definitions; the orchestra configuration is fixed at decoder start-up. It may contain score lines, MIDI events, and new sample data. When provided as part of an access unit, the score line is not required to contain a timestamp. When **has_time** is cleared in the **score_line** class, the event is dispatched immediately according to the rules in subclause 5.7.3.3.6. Score lines without timestamps are not responsive to orchestra tempo changes.

Annex 5.E discusses when the random access point flag, conveyed in the Access Unit packaging in the Systems specification, may be set.

5.6 Object types

There are four object types standardised for Structured Audio. Each of these object types corresponds to a particular set of application requirements. The default object type is Object type 4; when reference is made to MPEG-4 Structured Audio format without reference to a object type, it shall be understood that the reference is to Object type 4.

Terminals implementing MPEG-4 Systems profiles containing the **AudioFX** node (see ISO/IEC 14496-1, subclause 9.4.2.7) shall also provide support for Structured Audio Object type 3 or 4.

1. MIDI only. In this object type, only the **midi_file** chunk shall occur in the stream information header, anzd only the **midi_event** event shall occur in the bitstream data. In this object type, the General MIDI patch mappings are used, and the decoding process is described in subclause 5.14.2. This object type is used to enable backward-compatibility with existing MIDI content and rendering devices. Normative and implementation-independent sound quality cannot be produced in this object type.

2. **Wavetable** synthesis. In this object type, only the **midi_file** and **sbf** chunks shall occur in the stream information header, and only the **midi_event** event shall occur in the bitstream data. This object type is used to describe music and sound-effects content in situations in which the full flexibility and functionality of SAOL, including 3-D audio, is not required. In this case, the decoding process is described in subclause 5.13.3.1.

3. Algorithmic synthesis and AudioFX. In this object type, the **sbf** and **midi_file** chunks shall not occur in the **stream** information header. This object type is used to describe algorithmic synthesis and to provide audio effects processing in the AudioFX node when the use of the SASBF sample bank format (subclause 5.13) is not needed.

4. Main synthetic. All bitstream elements and stream information elements may occur.

The decoding process for Object types 3 and 4 is described in clause 5.7.

5.7 Decoding process

5.7.1 Introduction

This clause describes the algorithmic structured audio decoding process, in which a bitstream conforming to Object type 3 or 4 is converted into sound. The decoding process for Object type 1 bitstreams is described in subclause 5.14.2, and the decoding process for Object type 2 bitstreams in subclause 5.13.3.1.

5.7.2 Decoder configuration header

At the creation of a Structured Audio Elementary Stream, a Structured Audio decoder is instantiated and a bitstream object of class **SA_decoder_config** provided to that decoder as configuration information. At this time, the decoder shall initialise a run-time scheduler, and then parse the decoder configuration header into its component parts and use them as follows:

- Orchestra file: The orchestra file shall be checked for syntactic conformance with the SAOL grammar and rate semantics as specified in clause 5.8. Whatever pre-processing (i.e., compilation, allocation of static storage, etc.) needs to be done to prepare for orchestra run-time execution shall be performed.
- Score file: Each event in the score file shall be registered with the scheduler. To “register” means to inform the scheduler of the presence of a particular parameterised event at a particular future time, and the scheduler’s associated actions.
- MIDI file: Each event in the MIDI file shall be converted into an appropriate event as described in clause 5.13, and those events registered with the scheduler.

- Sample bank: The data in the bank shall be stored, and whatever pre-processing necessary to prepare for using the bank for synthesis shall be performed.
- Sample data: The data in the sample shall be stored, and whatever pre-processing necessary to prepare the data for reference from a SAOL wavetable generator shall be performed. If the sample data is represented as 16-bit integers in the bitstream, it shall be converted to floating-point format at this time.
- Symbol table: No normative decoder behaviour is associated with the symbol table.

If there is more than one orchestra file in the stream information header, the various files are combined together via concatenation and processed as one large orchestra file. That is, each orchestra file within the bitstream refers to the same global namespace, instrument namespace, and opcode namespace.

5.7.3 Bitstream data and sound creation

5.7.3.1 Relationship with systems layer

At each time step within the systems operation, the systems layer may present the Structured Audio decoder with an Access Unit containing data conforming to the **SA_access_unit** class. The run-time responsibility of the Structured Audio decoder is to receive these AU data elements, to parse and understand them as the various Structured Audio bitstream data elements, to execute the on-going SAOL orchestra to produce one Composition Unit of output, and to present the systems layer with that Composition Unit.

5.7.3.2 Bitstream data elements

As Access Units are received from the systems demultiplexer, they are parsed and used by the Structured Audio decoder in various ways, as follows:

- Sample data shall be stored, and whatever pre-processing is necessary for reference by forthcoming score lines containing references to that sample shall be performed. If the sample data is represented as 16-bit integers in the bitstream, it shall be converted to floating-point format at this time. Any samples in an access unit shall be processed before score lines, in case the score lines reference the samples.

- Score line events shall be registered with the scheduler if they have time stamps, or executed in the next k-cycle, if not.
    - MIDI events shall be converted into appropriate SAOL events (see clause 5.14) and then registered with the scheduler, if they have time stamps, or executed in the next k-cycle, if not.

5.7.3.3 Scheduler semantics

5.7.3.3.1 Purpose of scheduler

The scheduler is the central control mechanism of a Structured Audio decoding system. It is responsible for handling events by instantiating and terminating instruments, keeping track of what instrument instantiations are active, instructing the various instrument instantiations to perform synthesis, routing the output of instruments onto busses, and sending busses to effects instruments. Although there are many ways to perform these tasks, the exact nature of what must be done can be clearly specified. This subclause provides normative bounds on the activities of the scheduler.

5.7.3.3.2 Instrument instantiation

To instantiate an instrument is to create data space for its variables and the data space required for any opcodes called by that instrument. When an instrument is instantiated, the following tasks shall be performed. First, space for any parameter fields shall be allocated and their values set according to the p-fields of the instantiating expression or event. Then, space for any locally declared variables shall be allocated and these variable values set to 0. Then, the current values of any imported i-rate variables shall be copied into the local storage space. Then, locally declared wavetables shall be created and filled with data according to their declaration and the appropriate rules in clause 5.10.

5.7.3.3.3 Instrument termination

To terminate an instrument instantiation is to destroy the data space for that instance.

5.7.3.3.4 Instrument execution

To execute an instrument instantiation at a particular rate is to calculate the results of the instructions given in that instrument definition. When an instrument instance is executed at a particular rate, the following steps shall be performed. First, the values of any global variables and wavetables imported by that instrument at that rate shall be copied into the storage space of the instrument. In addition, when executing at the a-rate an instrument instance that is the target of a **send** statement, the current value of the **input** standard name in the instance shall be set to the current value of the bus or busses referenced in the **send** statement. Then, the code block for that instrument shall be executed at the particular rate with regard to the data space of the instrument instantiation, as given by the rules in subclause 5.8.6.6. Then, the values of any global variables and wavetables exported by that instrument at that rate shall be copied into the global storage space. Finally, when executing an instrument instantiation at the a-rate, the value of the instance output shall be added to the bus onto which the instrument is routed according to the rules in subclause 5.8.5.4, unless the instance is the target of a **send** expression referencing the special bus **output_bus**, in which case the output of the instrument instance is the output of the orchestra and may be turned into sound.

5.7.3.3.5 Orchestra start-up and configuration

5.7.3.3.5.1 Introduction

This subclause describes the steps required to begin the decoding process. These tasks (determination of instrument and bus width, global variable allocation, **startup** execution, global wavetable creation, bus and **send** instrument initialisation) shall be performed in the order indicated.

5.7.3.3.5.2 Determination of instrument output width and bus width

The output width of each instrument is determined in the order specified by the global sequencing rules (subclause 5.8.5.6); the width of each bus is determined by the sum of the output widths (subclause 5.8.6.6.8) of the instruments routed to that bus in a single **route** statement (subclause 5.8.5.4). Only for the purposes of calculating bus widths, any instrument that does not receive any bus data according to the sequence rules shall have an **inchannels** width of 1 (this specification is needed since output widths may depend on the value of **inchannels** or the width of **input**).

EXAMPLE 1

Consider the following orchestra.

```
global {
  route(bus1, i1);
  route(bus2, i2, i3);
  send(i2; ; bus1);
  send(i4; ; bus2);
}

instr i1(...) {
  asig a[2];
   ...
  output(a);
}

instr i2(...) {
  asig b;
  ...
  output(input + b);
}

instr i3(...) {
  asig c[inchannels];
  ...
  output(input,c);
}
```

```
instr i4(...) {
  asig d[inchannels];
  ...
  output(d + input);
}
```

In this orchestra, the global sequencing rules (subclause 5.8.5.6) specify that instrument **i1** precedes instrument **i2**, and instruments **i2** and **i3** precede instrument **i4**. Instrument **i1** has two channels, so bus **bus1** has two channels. Instrument **i2** has two channels, since it gets input from **bus1**. Instrument **i3** has two channels, since it gets no input, so **inchannels** is 1 and **c** and **input** have one channel each. The bus **bus2** has four channels, two each from **i1** and **i2**. The instrument **i4** has four channels, since it gets input from **bus2**.

EXAMPLE 2

Consider the following orchestra.

```
global {
  route(bus1, a);
  send(b; ;bus1);
  sequence(b,a);
  route(bus2, a, a);
  route(bus2, b);
  send(c; ; bus2);
}

instr a(...) {
  asig x,y;
  output(x,y,x);
}

instr b(...) {
  output(input,input);
}

instr c(...) { ... }
```

This orchestra contains a syntax error. The global sequence rules prescribe that **b** shall precede **a** (since the **sequence** directive overrides the implicit **send** sequencing), and that **a** and **b** precede **c**. The output width of **b** is two channels, since it does not receive any data according to the sequence rules, and so the width of its **input** is one channel. The output width of **a** is three channels. Thus, the width of **bus1** is three channels (although only the first is used by **b** since the width of its **input** is one channel). Thus, the two route statements onto **bus2** are incompatible, since the first uses six channels, but the second only two.

If the **sequence** directive were removed from the global block, then the syntax error would be resolved. In this case, instrument **a** precedes **b**, so **b** has three channels of input and six of output, and **bus2** can be correctly allocated with six channels.

5.7.3.3.5.3 Variable allocation, startup execution and global wavetable creation

Space for any global signal variables (see subclause 5.8.5.3) shall be allocated and their values set to zero. If there is an instrument called **startup** in the orchestra, that instrument shall be instantiated and executed at the i-rate. After this execution is complete, then all global wavetables are created and filled with data according to their definitions in the global block of the orchestra and the appropriate rules in clause 5.10.

5.7.3.3.5.4 Initialisation of busses and send instruments

After the global wavetable creation, the orchestra busses are created and initialised. Each channel of each bus is set to 0 values. After the busses are created, all instruments that are the targets of **send** statements as described in subclause 5.8.5.5 shall be instantiated and executed at the i-rate in the order specified by the global sequencing rules described in the global block according to subclause 5.8.5.6. Finally, the global absolute orchestra time shall be set to 0.

NOTE - A time is called *absolute* if it is specified in seconds. When a tempo instruction is first decoded and the value of tempo changes from its default value, the score time and the absolute time are not identical anymore; all the times in

the score, subsequent to a tempo line execution, are scaled according to the new tempo and enqueued in absolute dispatch and duration times as specified in subclause 5.7.3.3.6, list item 7.

5.7.3.3.6 Decoder execution while streaming

In each orchestra cycle, one Composition Unit of samples is produced by the real-time synthesis process. This synthesis is performed according to the rules below and the resulting orchestra output, as described in list item 11, is presented to the Systems layer as a Composition Unit. To execute one orchestra cycle, the following tasks shall be performed in the order denoted:

1. If there is an end event whose dispatch time is earlier than the current absolute orchestra time, or an end event has been received without a timestamp since the last execution of this rule, no further output is produced, and all future requests from the systems layer to produce Composition Units are responded to with buffers of all 0s.
2. If there are any instrument events whose dispatch time is earlier than the current absolute orchestra time, or any instrument events have been received without timestamps since the last execution of this rule, an instrument instantiation is created for each such instrument event (see subclause 5.7.3.3.2), and those instantiations are each executed at the i-rate (see subclause 5.7.3.3.4) in the order prescribed by the global sequencing rules. If the instrument event specifies a duration for that instrument instantiation, the instrument instantiation shall be scheduled for termination at the time given by the sum of the current absolute orchestra time and the specified duration (scaled to absolute time units according to the actual tempo, if any).

   NOTE - If the current orchestra time differs from the instrument dispatch time, the former shall be used to schedule instance termination.
3. If there are any active instrument instantiations whose termination time is earlier than the current absolute orchestra time, then the **released** standard name (see subclause 5.8.6.8.16) shall be set to 1 within each such instrument instance, and the instance is marked for termination in step 12, below.
4. If there are any control events whose dispatch time is earlier than the current absolute orchestra time, or any control events have been received without timestamps since the last execution of this rule, the global variables or instrument variables within instrument instantiations labelled by each such control event shall have their values updated accordingly (see subclause 5.11.4). Note that this implies that no more than one control change per variable per control cycle may be received by the orchestra. If multiple control changes that reference the same variable are received in a single control cycle, the resulting value of the instrument or global variable is unspecified.
5. If there are any table events whose dispatch time is earlier than the current absolute orchestra time, or any table events have been received without timestamps since the last execution of this rule, global wavetables shall be created or destroyed as specified by the table event (see subclause 5.11.5).
6. If there are any MIDI events whose timestamp is earlier than the current orchestra time, or that have been received without timestamps since the last execution of this rule, they are dispatched according to their semantics in subclause 5.14.3.
7. If there are any tempo events whose dispatch time is earlier than the current absolute orchestra time, or any tempo events have been received since the last execution of this rule, then the global tempo variable shall be set to the specified value, and the dispatch times of all events pending for execution shall be scaled to new times according to the new tempo value. The already scheduled times for terminations are also scaled in their remaining part, according to the ratio between the old and new tempo. Existing **extend** times are not affected, since they are specified in absolute time and are thus “outside” the score. The value of the **dur** standard name (subclause 5.8.6.8.7) shall be changed in each active instrument instance to reflect the new duration of the instance.

   If multiple tempo events are processed according to the preceding paragraph in the same control cycle, then the global tempo variable shall only be changed once, to the tempo indicated in the tempo event received last or with the latest timestamp, and the other tempo events are discarded. If there are multiple tempo events with the same timestamp, or both an un-timestamped event and a timestamped event shall be dispatched in the same control cycle, then the resulting value of the global tempo variable is unspecified.

NOTE - If the current orchestra time differs from the tempo dispatch time, the former shall be used to calculate the new durations and future dispatch times of events.

8. If the **speed** field of the **AudioSource** scene node responsible for instantiating this decoder (see clause 5.15) has been changed in the last k-cycle, the tempo standard variable shall be set to 60 times the value specified in subclause 9.4.2.9 (the **AudioSource** node) of ISO/IEC 14496-1, and events in the orchestra shall be rescaled as specified in list item 7, above.
9. The value of each channel of each bus shall be set to 0.
10. Each active instrument instance shall be executed once at the k-rate and *n* times at the a-rate, where *n* is the number of samples in the control period (see subclause 5.8.5.2.2). Each execution at the k-rate shall be in the order given by the global sequencing rules, and each corresponding execution at the a-rate (that is, the first a-rate execution in a k-cycle of each, the second a-rate execution in a k-cycle of each, etc.) shall be in the order given by the global sequencing rules.

    NOTE 1 - If instrument **a** is sequenced before instrument **b** according to the rules in subclause 5.8.5.6, then the k-rate execution of **a** shall be strictly before the k-rate execution of **b**, and the k-rate execution of **a** shall be strictly before the first a-rate execution of **a**, and the first a-rate execution of **a** shall be strictly before the first a-rate execution of **b**. However, there is no normative sequencing between the second a-rate execution of **a** and the first a-rate execution of **b**, or between the first a-rate execution of **a** and the k-rate execution of **b**, within a particular orchestra cycle.

    NOTE 2 - In accordance with the conformance rules in subclause 5.7.4, the execution ordering described in this subclause may be rearranged or ignored when it can be determined from examination of the orchestra that to do so will have no effect on the output of the decoding process. “Has no effect” shall be taken to mean that the output of the decoding process in rearranged order is sample-by-sample identical to the output of the decoding process performed strictly according to the rules in this subclause.

    NOTE 3 - The k-cycle execution of each instrument instance shall be executed as an atomic operation; that is, the k-cycle execution of one instance shall be completed before the next begins. It is not permissible to execute k-cycles in parallel. This is not true of a-cycles; if two instruments have no sequencing relationship according to the global sequencing rules, their a-cycles may be executed in any order or in parallel.
11. If the special bus **output_bus** is sent to an instrument, the output of that instrument at each a-cycle is the orchestra output at that a-cycle. Otherwise, the value of the special bus **output_bus** after each instrument has been executed for an a-cycle is the orchestra output at that a-cycle. If the value of the current orchestra output is greater than 1 or less than –1, it shall be set to 1 or -1 respectively (hard clipping).
12. For each instance that was marked for termination in step 3, above: if that instrument instance called **extend** with a parameter greater than the amount of time in a control-cycle, the instrument is not terminated. All other instrument instances marked for termination in step 3 are terminated (see subclause 5.7.3.3.3). As discussed in subclause 5.14.3.2.11, in the case of an “All Notes Off” MIDI message, instances may not extend themselves, and are destroyed at this time.
13. The current global absolute orchestra time is advanced by one control period.

5.7.3.3.7 Event priority

Certain events may be specified as “high priority” events, by setting the corresponding field in the bitstream element or using the ***** token in the textual score. In the case that overloading of the capability of the decoder occurs, this flag allows the content author to have a minimum of control on the performance degradation.

If the **high_priority** flag is set, then the event shall always be executed without degradation unless pathological conditions occur and no instantiations created at low priority levels are active. If the **high_priority** flag is cleared, then the event shall be executed without degradation if no critical conditions occur. Instrument events with the **high_priority** flag cleared may be prematurely terminated if resources are not available to dispatch an event with the **high_priority** flag set.

NOTE - Degradation is not intended as an allowed, normative, technique to lower the computational complexity. Conforming decoders shall be able to decode, in normal conditions, bitstreams of the specified Profile@Level with no degradation. Instead, the priority level is intended as a help to implementers in critical situations due to resource

overload, for instance in the case of a terminal attempting to decode a more complex bitstream than indicated by the level of the terminal, resource sharing with other applications, or high and unexpected degree of user interaction with the host terminal.

5.7.3.3.8 Late-arriving events

In the case of transmission error or encoder error, certain events may arrive with timestamps that have already passed. The **use_if_late** field in the bitstream element indicates the proper behaviour in this case. If this field is cleared, then the event is ignored, and processing shall continue as if the event had never arrived. If this field is set, then the event is immediately dispatched according to the rules in subclause 5.7.3.3.6 as though it had been received with no timestamp.

5.7.4 Conformance

With regard to all normative language in this section of ISO/IEC 14496-3, conformance to the normative language is measured at the time of orchestra output. Any optimisation of SAOL code or rearrangement of processing sequence may be performed as long as to do so has no effect on the output of the orchestra. "Has no effect" in this sense means that the output of the rearranged or optimised orchestra is sample-by-sample identical to the output of the original orchestra according to the decoding rules given in this section.

See also ISO/IEC 14496-4.

5.8 SAOL syntax and semantics

5.8.1 Relationship with bitstream syntax

The bitstream syntax description as given in clause 5.5 specifies the representation of SAOL instruments and algorithms that shall be presented to the decoder in the bitstream. However, the tokenised description as presented there is not adequate to describe the SAOL language syntax and semantics. In addition, for purposes of enabling bitstream creation and exchange in a robust manner, it is useful to have a standard human-readable textual representation of SAOL code in addition to the tokenised binary format.

The Backus-Naur Format (BNF) grammar presented in this subclause denotes a *language*, or an infinite set of *programs*; the legal programs that may be transmitted in the bitstream are restricted to this set. Any program that cannot be parsed by this grammar is not a legal SAOL program – it has a *syntax error* – and a bitstream containing it is an invalid bitstream. Although the bitstream is made up of tokens, the grammar will be described in terms of lexical elements – a *textual representation* – for clarity of presentation. The syntactic rules expressed by the grammar that restrict the set of textual programs also normatively restrict the syntax of the bitstream, through the relationship of the bitstream and the textual format in the normative tokenisation process.

This clause thus describes a textual representation of SAOL that is standardised, but stands outside of the bitstream-decoder relationship. Clause 5.12 describes the mapping between this textual representation and the bitstream representation. The exact normative *semantics* of SAOL will be described in reference to the textual representation, but also apply to the tokenised bitstream representation as created via the normative tokenisation mapping.

Annex 5.C contains a grammar for the SAOL textual language, represented in the 'lex' and 'yacc' formats. Using these versions of the grammar, parsers can be automatically created using the 'lex' and 'yacc' tools. However, these versions are for informative purposes only; there is no requirement to use these tools in building a decoder.

Normative language regarding syntax in this clause provides bounds on syntactically legal SAOL programs, and by extension, the syntactically legal bitstream sequences that can appear in an **orchestra** bitstream class. That is, there are constructions that appear to be permissible upon reading only the BNF grammar, but are disallowed in the normative text accompanying the grammar. The status of such constructions is exactly that of those which are outside of the language defined by the grammar alone. In addition, normative language describing static rate semantics further bounds the set of syntactically legal SAOL programs, and by extension, the set of syntactically legal bitstream sequences.

The decoding process for bitstreams containing syntactically illegal SAOL programs (i.e., SAOL programs that do not conform to the BNF grammar, or contain syntax errors or rate mismatch errors) is unspecified.

Normative language regarding semantics in this clause describes the semantic bounds on the behaviour of the Structured Audio decoder. Certain constructions describe “run-time error” situations; the behaviour of the decoder in such circumstances is not normative, but implementations are encouraged to recover gracefully from such situations and continue decoding if possible.

5.8.2 Lexical elements

5.8.2.1 Concepts

The textual SAOL orchestra contains punctuation marks, which syntactically disambiguate the orchestra; identifiers, which denote symbols of the orchestra; numbers, which denote constant values; string constants, which are not currently used; comments, which allow internal documentation of the orchestra; and whitespace, which lexically separates the various textual elements. These elements do not occur in the bitstream – since each is represented there by a token – but we define them here to ground the subsequent discussion of SAOL. Within the rest of clause 5.8, when we discuss the semantics of “an identifier”, this shall be taken to normatively refer to the semantics of the symbol denoted by that identifier; the language used is for clarity of presentation.

A lexical grammar for parsing SAOL, written in the ‘lex’ language, is provided for informative purposes in Annex 5.C.

5.8.2.2 Identifiers

An identifier is a series of one or more letters, digits and the underscore that begins with a letter or underscore; it denotes a symbol of the orchestra. Every identifier that consists of the same characters in the first 16 characters (is equivalent under string comparison to the first 16 characters) denotes the same symbol. Identifiers are case-sensitive, meaning that identifiers that differ only in the case of one or more characters denote different symbols.

A string of characters equivalent to one of the reserved words listed in subclause 5.8.9, to one of the standard names listed in subclause 5.8.6.8, to the name of one of the core opcodes listed in subclause 5.9.3, or to the name of one of the core wavetable generators listed in clause 5.10 does not denote a symbol, but rather denotes that reserved word, standard name, core opcode, or core wavetable generator.

An identifier is denoted in the BNF grammar below by the terminal symbol **<ident>**.

5.8.2.3 Numbers

There are two kinds of symbolic constants that hold numeric values in SAOL: integer constants and floating-point constants.

The integer constant is required to occur in certain contexts, such as array definitions. An integer token is a series of one or more digits. Since the contexts in which integers are required to occur in SAOL do not allow negative values, there is no provision for negative integers. A string of characters that appears to be a negative integer shall be lexically analysed as a floating-point constant. No integer constant greater than 2^{32} (4294967296) shall occur in the orchestra.

There is no different in SAOL between numbers coded with the bitstream token for integers and those coded with the bitstream token for bytes. The latter is only an aid to compression of the bitstream.

An integer constant is denoted in the BNF grammar below by the terminal symbol **<int>**.

The floating-point constant occurs in SAOL expressions, and denotes a constant numeric value. A floating-point token consists of a base, optionally followed by an exponent. A base is either a series of one or more digits, optionally followed by a decimal point and a series of zero or more digits, or a decimal point followed by a series of one or more digits. An exponent is the letter **e**, optionally followed by either a **+** or **–** character, followed by a series of one or more digits. Since the floating-point constant appears in a SAOL expression, where the unary negation operator is always available, floating-point constants need not be lexically negative. Every floating-point constant in the orchestra shall be representable by a 32-bit floating-point number.

A floating-point constant is denoted in the BNF grammar below by the terminal symbol **<number>**.

5.8.2.4 String constants

String constants are not used in the normative SAOL specification, but a description is provided here so that they may be treated consistently by implementers who choose to add functionality over and above normative requirements to their implementations.

A string constant denotes a constant string value, that is, a character sequence. A string constant is a series of characters enclosed in double quotation marks (**"**). The double quotation character may be included in the string constant by preceding it with a backslash (****) character. Any other character, including the line-break (newline) character, may be explicitly enclosed in the quotation marks.

The interpretation and use of string constants is left open to implementers.

5.8.2.5 Comments

Comments may be used in the textual SAOL representation to internally document an orchestra. However, they are not included in the bitstream, and so are lost on a tokenisation/detokenisation sequence.

A comment is any series of characters beginning with two slashes (***//***), and terminating with a new line. During lexical analysis, whenever the ***//*** element is found on a line, the rest of the line is ignored.

5.8.2.6 Whitespace

Whitespace serves to lexically separate the various elements of a textual SAOL orchestra. It has no syntactic function in SAOL, and is not represented in the bitstream, so the exact whitespacing of a textual orchestra is lost on a tokenisation/detokenisation sequence. Whitespace is not required in SAOL except so far as to disambiguate tokens and reserved words that appear next to each other (to separate "asig" from the variable name declared, for example).

A whitespace is any series of one or more space, tab, and/or newline characters.

5.8.3 Variables and values

Each signal variable within the SAOL orchestra holds a value, or an ordered set of values for array variables, as an intermediate calculation by the orchestra. At any point in time, the value of a variable, sample in a wavetable, or single element of an array variable, shall be represented by a 32-bit floating-point value.

Conformance to this subclause is in accordance with subclause 5.7.4; that is, implementations are free to use any internal representation for variable values, so long as the results calculated are identical to the results of the calculations using 32-bit floating-point values.

NOTE - For certain sensitive digital-filtering operations, the results of using greater precision in a calculation may be equivalently detrimental to orchestra output as the results of using less precision, as the stability of the filter may be critically dependent on the quantisation error that is provided with 32-bit values. It is strongly discouraged for bitstreams to contain code that generates very different results when calculated with 32-bit and 64-bit arithmetic.

At orchestra output, the values calculated by the orchestra should reside between a minimum value of –1 and a maximum value of 1. These values at orchestra output represent the maximum negatively- and positively-valued audio samples that can be produced by the terminal. If the values calculated by the orchestra fall outside that range, they are clipped to [-1,1] as described in subclause 5.7.3.3 list item 11. This sound is presented, control cycle by control cycle, to the MPEG-4 system for use in AudioBIFS composition.

5.8.4 Orchestra

<orchestra> -> <orchestra element> <orchestra>
<orchestra> -> <orchestra element>

The orchestra is the collection of signal processing routines and declarations that make up a Structured Audio processing description. It shall consist of a list of one or more orchestra elements.

<orchestra element> -> <global block>
<orchestra element> -> <instrument declaration>
<orchestra element> -> <opcode declaration>

<orchestra element> -> <template declaration>
<orchestra element> -> **NULL**

There are four kinds of orchestra elements:

1. The global block contains instructions for global orchestra parameters, bus routings, global variable declarations, and instrument sequencing. It is not permissible to have more than one global block in an orchestra.
2. Instrument declarations describe sequences of processing instructions that can be parametrically controlled using SASL or MIDI score files.
3. Opcode declarations describe sequences of processing instruments that provide encapsulated functionality used by zero or more instruments in the orchestra.
4. Template declarations describe multiple instruments that differ only slightly using a concise parametric form.

Orchestra elements may appear in any order within the orchestra; in particular, opcode definitions may occur either syntactically before or after they are used in instruments or other opcodes.

5.8.5 Global block

5.8.5.1 Syntactic form

<global block> -> **global {** <global list> **}**
<global list> -> <global statement> <global list>
<global list> -> NULL

A global block shall contain a global list, which shall consist of a sequence of zero or more global statements.

<global statement> -> <global parameter>
<global statement> -> <global variable declaration>
<global statement> -> <route statement>
<global statement> -> <send statement>
<global statement> -> <sequence definition>
<global statement> -> <interpolation level>

There are several kinds of global statement.

1. Global parameters set orchestra parameters such as sampling rate, control rate, and number of input and output channels of sound
2. Global variable declarations define global variables that can be shared by multiple instruments.
3. Route statements describe the routing of instrument outputs onto busses.
4. Send statements describe the sending of busses to effects instruments.
5. Sequence definitions describe the sequencing of instruments by the run-time scheduler.
6. The interpolation level specifies the quality of interpolation performed in the synthesis process.

5.8.5.2 Global parameter

5.8.5.2.1 srate parameter

<global parameter> -> **srate <int>;**

The **srate** global parameter specifies the audio sampling rate of the orchestra. The decoding process shall create audio internally at this sampling rate. It is not permissible to simplify orchestra complexity or account for terminal capability by generating audio internally at other sampling rates, for to do so may have seriously detrimental effects on certain processing elements of the orchestra.

The **srate** parameter shall be an integer value between 4000 and 96000 inclusive, specifying the audio sampling rate in Hz. If the **srate** parameter is not provided in an orchestra, the default shall be the fastest of the audio signals provided as input (see clause 5.15). If the sampling rate is not provided, and there are no input audio signals, the default sampling rate shall be 32000 Hz. It is a syntax error if more than one **srate** parameter instruction occurs in an orchestra.

In a Object type 3 terminal, or when the SAOL orchestra is used in an **AudioFX** AudioBIFS node (see subclause 5.15.3), the **srate** parameter shall only be one of the following values: (4000, 8000, 11025, 12000, 16000, 22050, 24000, 32000, 44100, 48000, 88200, 96000).

5.8.5.2.2 krate parameter

<global parameter> -> **krate <int>;**

The **krate** global parameter specifies the control rate of the orchestra. The decoding process shall execute k-rate processing internally at this rate. It is not permissible to simplify orchestra complexity or account for terminal capability by executing k-rate processing at other rates, unless it can be determined that to do so will have no effect on orchestra output. In this case, “no effect” means that the resulting output of the orchestra is sample-by-sample identical to the output created if the control rate is not altered.

The **krate** parameter shall be an integer value between 1 and the sampling rate inclusive, specifying the control rate in Hz. If the **krate** parameter is not provided in an orchestra, the default control rate shall be 100 Hz. It is a syntax error if more than one **krate** parameter instruction occurs in an orchestra.

If the control rate as determined by the previous paragraph is not an even divisor of the sampling rate, then the control rate is the next larger integer that does evenly divide the sampling rate. The *control period* of the orchestra is the number of samples, or amount of time represented by these samples, in one control cycle.

5.8.5.2.3 inchannels parameter

<global parameter> -> **inchannels <int>;**

The **inchannels** global parameter specifies the number of input channels to process. If there are fewer than this many audio channels provided as input sources, the additional channels shall be set to continuous zero-valued signals. If there are more than this many audio channels provided as input sources, the extra channels are ignored.

If the **inchannels** parameter is not provided in an orchestra, the default shall be the sum of the numbers of channels provided by the input sources (see clause 5.15). If there are no input sources provided, the value shall be 0. It is a syntax error if more than one **inchannels** parameter instruction occurs in an orchestra.

The only normative way in which audio input is processed by a SAOL orchestra is when the orchestra is embedded in an AudioBIFS **AudioFX** node, see ISO/IEC 14496-1 subclause 9.4.2.7. Nonnormative methods for audio processing include soundfile processing or microphone processing, see Annex 5.F.

5.8.5.2.4 outchannels parameter

<global parameter> -> **outchannels <int>;**

The **outchannels** global parameter specifies the number of output channels of sound to produce. The run-time decoding process shall produce and render this number of channels internally. It is not permissible to simplify orchestra complexity or account for terminal capability by producing fewer channels.

If the **outchannels** parameter is not provided in an orchestra, the default shall be one channel. It is a syntax error if more than one **outchannels** parameter instruction occurs in an orchestra.

5.8.5.2.5 interp parameter

<global parameter> -> **interp <int>;**

The **interp** global parameter specifies the quality of interpolation performed in the synthesis process. Various operations require access to wavetables at non-integer points; to access a wavetable at such a point requires interpolation among the available points in the wavetable.

If the **interp** parameter is 0, then "low level" interpolation is performed. Every interpolation shall be performed using a linear interpolation. That is, let **i** and **j** be two consecutive indices of a wavetable, and let **x** and **y** be the values at points **i** and **j** respectively. Assume that the value at point **k** is required, where **i < k < j**. Then let **q** be the value **k** – floor(**k**). Then the interpolated value is **x + q** * (**y – x**).

If the value required is that "between" the last point and first point of the table as it wraps around, consider **x** to be the value of the last point, **y** the value of the first point, and assert **n < k < n+1** where index **n** is the last point in the wavetable. Then calculate the interpolated value as **x** + **q** * (**y – x**) as above.

If the **interp** parameter is 1, then "high level" interpolation is performed. The method of high-level interpolation is non-normative, but it shall be a higher-quality method than linear interpolation.

It is a syntax error if the parameter is not 0 or 1. It is a syntax error if more than one **interp** parameter instruction occurs in an orchestra.

If the **interp** parameter is not specified in an orchestra, then the interpolation quality is "low" by default. If authors wish to have normative, high-quality interpolation for wavetables, they can rewrite their own versions of **tableread**, **oscil**, and other instructions to perform this.

5.8.5.3 Global variable declaration

5.8.5.3.1 Syntactic form

<global variable declaration> -> **ivar** <namelist> **;**
<global variable declaration> -> **ksig** <namelist> **;**
<global variable declaration> -> <table declaration> **;**

Global variable declarations declare variables that may be shared and accessed by all instruments and by a SASL score. Only **ivar** and **ksig** type variables, as well as wavetables, may be declared globally. A global variable declaration is either a table definition, or an allowed type name followed by a list of name declarations.

A global name declaration specifies that a name token shall be created and space equal to one signal value allocated for variable storage in the global context. A global array declaration specifies that a name token shall be created and space equal to the specified number of signal values allocated in the global context.

5.8.5.3.2 Signal variables

<namelist> -> <name>**,** <namelist>
<namelist> -> <name>

A namelist is a sequence of one or more name declarations.

<name> -> **<ident>**
<name> -> **<ident>[**<array length>**]**

<array length> -> **<int>**
<array length> -> **inchannels**
<array length> -> **outchannels**

A name declaration is an identifier (see subclause 5.8.2.2), or an array declaration. For an array declaration, the parameter shall be either an integer strictly greater than 0, or one of the tokens **inchannels** or **outchannels**. If the latter, the array length shall be the same as the number of input channels or output channels to the instrument, respectively, as described in subclause 5.7.3.3.5.2. It is illegal to use the token **inchannels** if the number of input channels to the instrument is 0.

Not every identifier may be used as a variable name; in particular, the reserved words listed in subclause 5.8.8, the standard names listed in subclause 5.8.6.8, the names of the core opcodes listed in clause 5.9, and the names of the core wavetable generators listed in clause 5.10 shall not be declared as variable names.

5.8.5.3.3 Wavetable declarations

<table declaration> -> **table <ident> (<ident> ,** <expr> [**,** <expr list>] **) ;**
<expr> as defined in subclause 5.8.6.7.
<expr list> as defined in subclause 5.8.6.6.1.

Wavetables are structures of memory allocated for the typical purpose of allowing rapid oscillation, looping, and playback. The wavetable declaration associates a name (the first identifier) with a wavetable created by a core wavetable generator referenced by the second identifier. It is a syntax error if the second identifier is not one of the core wavetable generators named in clause 5.10. The first expression in the comma-delimited parameter sequence is termed the *size expression*; the remaining zero or more expressions comprise the *wavetable parameter list*.

The semantics of the size expression and wavetable parameter list are determined by the particular core wavetable generator, see clause 5.10. Any expression that is i-rate (see subclause 5.8.6.7.2) is legal as part of the table parameter list; in particular, reference to i-rate global variables is allowed (their values may be set by the special instrument **startup**). Each expression shall be single-valued, except in the case of the **concat** generator (subclause 5.10.16), in which case the expressions shall be table references. The order of creation of wavetables is non-deterministic; it is not recommended for calls to the **tableread()** opcode to occur in the table parameter expressions, and to do so gives unspecified results.

A global wavetable may be referenced by a wavetable placeholder in any instrument or opcode. See subclause 5.8.6.5.4. Global wavetables shall be created and initialised with data at orchestra initialisation time, immediately after the execution of the special instrument **startup.** They shall not be destroyed unless they are explicitly destroyed or replaced by a **table** line in a SASL score.

To create a wavetable, first, the expression fields are evaluated in the order they appear in the syntax according to the rules in subclause 5.8.6.7. Then, the particular wavetable generator named in the second identifier is executed; the normative semantics of each wavetable generator detail exactly how large a wavetable shall be created, and which values placed in the wavetable, for each generator.

5.8.5.4 Route statement

<route statement> -> **route (<ident> ,** <identlist> **) ;**

<identlist> -> **<ident> ,** <identlist>
<identlist> -> **<ident>**
<identlist> -> **<NULL>**

A **route** statement consists of a single identifier, which specifies a bus, and a sequence of one or more instrument names, which specify instruments. The route statement specifies that the instruments listed do not produce sound output directly, but instead their results are placed on the given bus. The output channels from the instruments listed each are placed on a separate channel of the bus. Multiple **route** statements onto the same bus indicate that the given instrument outputs shall be summed on the bus. Multiple **route** statements with differing numbers of channels referencing the same bus are illegal, unless each statement has either *n* channels or 1 channel. In this case, each of the one-channel **route** statements places the same signal on each channel of the bus, which is *n* channels wide.

There shall be at least one instrument name in the instrument list (the NULL subclause in the grammar is provided so that constructions appearing later may use the same production).

EXAMPLES

Assume that instruments **a, b,** and **c** produce one, two, and three channels of output, respectively.

1. The sequence

```
route(bus1, a, b);
   route(bus1, c);
```

is legal and specifies a three-channel bus. The first bus channel contains the sum of the output of **a** and the first channel of **c**; the second contains the sum of the first output channel of **b** and the second of **c**; and the third contains the sum of the second channel of **b** and the third channel of **c**.

2. The sequence

```
route(bus1,b);
route(bus1,c);
```

is illegal since the statements refer different numbers of channels to the same bus.

3. The sequence

```
route(bus1,a,c);
route(bus1,a);
route(bus1,b,b);
```

is legal and specifies a four-channel bus. The first and third **route** statements each refer to four channels of audio, and the second refers to one channel, which will be mapped to each of the four channels.

The resulting channel values are as follows, using array notation to indicate the channel outputs from each instrument:

Table 5.1 - Example of calculating bus routing values

Channel	Value
1	`a + a + b[1]`
2	`c[1] + a + b[2]`
3	`c[2] + a + b[1]`
4	`c[3] + a + b[2]`

It is illegal for a **route** statement to reference a bus that is not the special bus **output_bus** and that does not occur in a **send** statement. See subclause 5.8.5.5.

It is illegal for a **route** statement to refer to the special bus **input_bus** (see subclause 5.15.2)**.**

All instruments that are not referred to in **route** statements place their output on the special bus **output_bus,** except for an effect instrument to which **output_bus** was sent (see subclause 5.8.5.5). The same rules for allowable channel combinations to the special bus **output_bus** apply as if the route statements were explicit; these rules are implicit in the rules for the **output** statement, see subclause 5.8.6.6.8.

5.8.5.5 Send statement

<send statement> -> **send (<ident> ;** <expr list> **;** <identlist> **);**
<identlist> as defined in subclause 5.8.5.4
<expr list> as defined in subclause 5.8.6.6.1

The **send** statement creates an instrument instantiation, defines busses, and specifies that the referenced instrument is used as an effects processor for those busses.

All busses in the orchestra are defined by using **send** statements. It is illegal for a statement referencing a bus to refer to a bus that is not defined in a **send** statement. The exception is the special bus **output_bus**, which is always defined.

The identifier in the **send** statement references an instrument that will be used as a bus-processing instrument, also called *effect instrument*. There is no syntactic distinction between effect instruments and other instruments. The identifier list references one or more busses that shall be made available to the effect instrument through its **input** standard name, as follows:

The first n_0 channels of **input,** channels 0 through n_0-1 are the n_0 channels of the first referenced bus;
Channels n_0 through n_0+n_1-1 of **input** are the n_1 channels of the second bus,
and so forth, with a total of $n_0 + n_1 + \ldots + n_k$ channels.

In addition, the grouping of busses in the **input** array shall be made available to the effect instrument through its **inGroup** standard name, as follows:

The first n_0 values of **inGroup** have the value 1;
Channels n_0 through n_0+n_1-1 of **inGroup** have the value 2,
and so forth, through $n_0 + n_1 + \ldots + n_k$, with the last n_k having the value k.

The expression list is a list of zero or more i-rate expressions that are provided to the effect instrument as its parameter fields. Any expression that is i-rate (see subclause 5.8.6.7.2) is legal as part of this list; in particular, reference to i-rate global variables is allowed. The number of expressions provided shall match the number of parameter fields defined in the instrument declaration; otherwise, it is a syntax error.

The effect instrument referred to in a **send** statement shall be instantiated at orchestra start-up; see subclause 5.7.3.3.5.4. These instrument instantiations shall remain in effect until the orchestra synthesis process terminates. One instrument instantiation shall be created for each **send** statement in the orchestra. If such an instrument instantiation utilises the **turnoff** statement, the instantiation is destroyed (and sound is no longer routed to it). No other changes are made in the orchestra.

Any bus may be routed to more than one effect instrument, except for the special bus **output_bus**. The special bus **output_bus** represents the second-to-finalmost processing of a sound stream; it may only be sent to at most one effect instrument, and it is a syntax error if that instrument is itself routed or makes use of the **outbus** statement. If **output_bus** is not sent to an instrument, it is turned into sound at the end of an orchestra cycle (see subclause 5.7.3.3); if **output_bus** is sent to an instrument, the output of that instrument is turned into sound at the end of an orchestra pass. This instrument is not permitted to use the **turnoff** statement.

In the case that the number of input channels received by an instrument instance differs from the width of the bus(es) providing that input (see subclause 5.7.3.3.5.2), the width of **input** and **inGroup** and the value of **inchan** respect the former rather than the latter. In the case that **inchan** is smaller than the number of channels on the buses providing input, only the first **inchan** channels are used; in the case that **inchan** is smaller, the "extra" channels are all 0's in both **input** and **inGroup**.

At least one bus name shall be provided in the **send** instruction.

5.8.5.6 Sequence specification

<sequence specification> -> **sequence (** <identlist> **) ;**
<identlist> as defined in subclause 5.8.5.4.

The **sequence** statement allows the specification of the ordering of execution of instrument instantiations by the run-time scheduler. The identlist references a list of instruments that describes a partial ordering on the set of instruments. If instrument **a** and instrument **b** are referenced in the same **sequence** statement with **a** preceding **b**, then instantiations of instrument **a** shall be executed strictly before instantiations of instrument **b**.

There are several default sequence rules:

1. The special instrument **startup** is instantiated and the instantiation executed at the i-rate at the very beginning of the orchestra.
2. Any instrument instances corresponding to the **startup** instrument are executed first in a particular orchestra cycle.
3. If **output_bus** is sent to an instrument, the instrument instantiation corresponding to that **send** statement is the last instantiation executed in the orchestra cycle.
4. For each instrument routed to a bus that is sent to an effect instrument, instantiations of the routed instrument are executed before instantiations of the effect instrument. If loops are created using **route** and **send** statements, the ordering is resolved syntactically: whichever **send** statement occurs latest, that instrument instantiation is executed latest.

Default rules 2, 3, and 4 may be overridden by use of the **sequence** statement. Rule 1 cannot be overridden.

It is a syntax error if explicit **sequence** statements create loops in ordering. Any **send** statements that are the "backward" part of an implicit **send** loop have no effect.

If the sequence of two instruments is not defined by the default or explicit sequence rules, their instantiations may be executed in any order or in parallel.

It is not possible to specify the ordering of multiple instantiations of the same instrument; these instantiations can be run in any order or in parallel.

EXAMPLES

An orchestra consists of five instruments, **a, b, c, d,** and **e**.

1. The following code fragment

```
route(bus1, a, b);
send(c; ; bus1);
```

is legal and specifies (using the default sequencing rules) that instantiations of instruments **a** and **b** shall be executed strictly before instantiations of instrument **c**. This ordering applies to all instantiations of instrument **c**, not only to the one corresponding to the **send** statement. No ordering is specified between instruments **a** and **b**.

2. The following code fragment

```
route(bus1, a, b);
send(c; ; bus1);
sequence(c,a);
send(d; ; bus1);
```

is legal and specifies that instantiations of instrument **b** shall be executed first, followed by instantiations of instrument **c**, followed by instantiations of instrument **a**, followed by instances of instrument **d**. (See Figure 5.8.1) The ordering of **b** and **c**, and **a** and **b** with **d**, follows from default rule 3; the placement of instrument **c** follows from the explicit **sequence** statement, which overrides default rule 3. Due to this ordering, the output samples of instrument **a** are not provided to instrument **c** (they get put on the bus "too late"), and however many channels of output this represents are set to 0 in instrument **c**. The output samples of instrument **a** are provided to instrument **d**.

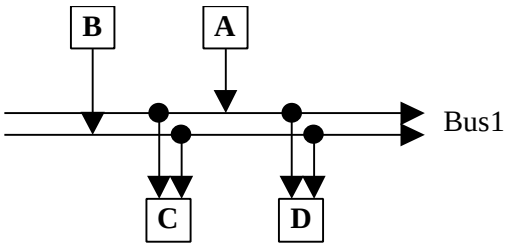


Figure 5.1 - Example of ordering instruments with 'sequence'

3. The following code fragment

```
sequence(a,b);
sequence(b,c,d);
sequence(c,e);
sequence(e,a);
```

is illegal, as it contains an explicit loop in sequencing.

4. The following code fragment

```
route(bus1, a);
send(b; ; bus1);
route(bus2, b);
send(a; ; bus2);
```

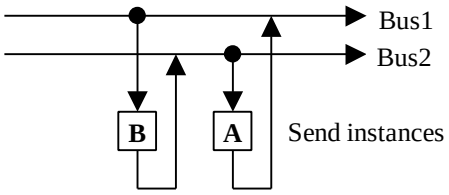


Figure 5.2 - Example of ordering instruments with 'sequence'

is legal, and specifies that instantiations of instrument **b** are executed first, followed by instantiations of instrument **a**. There is an implicit loop here that is resolved syntactically as described in default rule 3. Due to this ordering, the output values of instrument **a** are not provided to instrument **b**. Note that for deciding sequencing, only the order of **send** statements matters, not the order of **route** statements.

5.8.6 Instrument definition

5.8.6.1 Syntactic form

<instrument definition> -> **instr <ident> (** <identlist> **)** [**preset <int>** [**<int> ...**]] **{**
<instr variable declarations>
<block> **}**

An instrument definition has several elements. In order, they are

1. An identifier that defines the name of the instrument,
2. A list of zero or more identifiers that define names for the parameter fields, also called pfields, of the instrument,
3. An optional list of preset values for specifying MIDI preset mappings,
4. A list of zero or more instrument variable declarations, and
5. A block of statements defining the executable functionality of the instrument.

5.8.6.2 Instrument name

Any identifier may serve as the instrument name except that the instrument name shall not be a reserved word (see subclause 5.8.9), the name of a core opcode (see clause 5.9), or the name of a core wavetable generator (see clause 5.10). An instrument name may be the same as a variable in local or global scope; there is no ambiguity so created, since the contexts in which instrument names may occur are very restricted.

No two instruments or opcodes in an orchestra shall have the same name.

5.8.6.3 Parameter fields

<identlist> -> as given in subclause 5.8.5.4

The parameter fields, also called pfields, of the instrument, are the interface through which the instrument is instantiated. In the instrument code, the pfields have the rate semantics of i-rate local variables. Their values shall be set on instrument instantiation, before the creation of local variables, with the appropriate values as given in the score line, score event, MIDI event, **send** statement, or **instr** statement corresponding to the instrument instantiation.

5.8.6.4 Preset tag

The **preset** tag specifies the preset number(s) of the instrument. When MIDI program change events arrive in a MIDI stream or MIDI file controlling the orchestra, the program change numbers refer to the **preset** tags given to the various instruments. No more than one instrument may have the same preset number; if multiple instruments in an orchestra specify the same **preset** tag, the one occurring syntactically last is assigned that preset number. If a **preset** tag is not associated with a particular instrument, then that instrument has no preset number and cannot be referenced with a program change. If more than one tag is given, the instrument responds to all of the listed preset values.

Preset tags in SAOL correspond to both the preset and bank value of a program in MIDI control; the program on preset **x**, bank **y** in MIDI syntax shall be indicated as preset (**y** - 1) *128 + (**x** – 1) in SAOL (since presets and banks are numbered starting with 1 in MIDI).

See clause 5.14 for more normative semantics governing MIDI control of orchestras.

5.8.6.5 Instrument variable declarations

5.8.6.5.1 Syntactic form

<instr variable declarations> -> <instr variable declarations> <instr variable declaration>
<instr variable declarations> -> **<NULL>**

<instr variable declaration> -> [<sharing tag>] **ivar** <namelist> **;**
<instr variable declaration> -> [<sharing tag>] **ksig** <namelist> **;**
<instr variable declaration> -> **asig** <namelist> **;**

<instr variable declaration> -> <table declaration> **;**
<instr variable declaration> -> <sharing tag> **table** <identlist> **;**
<instr variable declaration> -> **oparray <ident> [**<array length> **] ;**
<instr variable declaration> -> <tablemap declaration> **;**

<sharing tag> -> **imports**
<sharing tag> -> **exports**
<sharing tag> -> **imports exports**

<tablemap declaration> -> **tablemap <ident> (** <identlist> **) ;**

<array length> and <namelist> as defined in subclause 5.8.5.3.2
<table declaration> as defined in subclause 5.8.5.3.3
<identlist> as defined in subclause 5.8.5.4

Instrument variable declarations declare variables that may be used within the scope of an instrument. Any rate type variable, as well as wavetables, tablemaps, and wavetable placeholders, may be declared in an instrument. An instrument variable declaration is either a wavetable declaration, or an type name, possibly preceded by a sharing tag, followed by a list of name declarations, or a sharing tag followed by the token **table** followed by a list of identifiers referencing global or future wavetables, or an opcode-array declaration, or a table-map definition.

5.8.6.5.2 Wavetable declaration

The syntax and semantics of subclause 5.8.5.3.3 hold for instrument local wavetables, with the following exceptions and additions:

An instrument local wavetable is available only within the local scope of a single instrument instantiation. As such, it shall be created and initialised with data at the instrument instantiation time, immediately after the pfield values are assigned from the calling parameters. It may be deleted and freed when that instrument instantiation terminates.

Not every expression that is i-rate is legal as part of the table parameter list. Reference to constants, pfields, imported i-rate variables, and i-rate standard names is allowed. However, the instrument wavetable initialisation shall occur before the initialisation pass through the instrument code, and so reference to local i-rate variables is prohibited.

5.8.6.5.3 Signal variables

The syntax and semantics of subclause 5.8.5.3.2 hold for instrument local signal variables, with the following exceptions and additions:

A local name declaration specifies that a name token shall be created and space equal to one signal value allocated for variable storage in each instrument instantiation associated with the instrument definition. A local array declaration specifies that a name token shall be created and space equal to the specified number of signal values allocated in each instrument instantiation associated with the instrument definition.

The sharing tags **imports** and/or **exports** may be used with local i-rate or k-rate signal variable declaration. They shall not be used with a-rate variables. If the **imports** tag is used, then the variable value shall be replaced with the value of the global variable of the same name at instrument initialisation time (for i-rate signal variables) or at the beginning of each control pass (for k-rate signal variables). The **imports** tag may be used for a local k-rate signal variable even if there is no global variable of the same name, in which case it is an indication that the k-rate variable so tagged may be modified with **control** lines in a SASL score. The **imports** tag shall not be used for local i-rate signal variables when there is no global variable of the same name.

If the **exports** tag is used, then the value of the global variable of the same name shall be replaced with the value of the local signal variable after instrument initialisation (for i-rate signal variables) or at the end of each control pass (for k-rate signal variables). The **exports** tag shall not be used if there is no global variable of the same name.

If, for a particular signal variable, the **imports** and/or **exports** tags are used, and there is a global variable with the same name, then the array width of the local and global variables shall be the same.

If, for a particular local variable, the **imports** tag is not used, then its value is set to 0 before instrument initialisation.

If, for a particular local variable declaration, the **imports** and **exports** tags are not used, even if there is a global variable of the same name, there is no semantic relationship between the two variables. The construction is syntactically legal.

5.8.6.5.4 Wavetable placeholder

The sharing tags **imports** and **exports** may be used to reference global and future wavetables. In this case, the local declaration of the table reference is termed a wavetable placeholder. The wavetable placeholder definition does not contain a full wavetable definition, but only a reference to a global or future wavetable name.

If only the **imports** tag is used, and there is a global wavetable with the same name, then at instrument instantiation time, the current contents of the global wavetable are copied into a local wavetable with that name. If the contents of the global wavetable are modified after a particular instrument instantiation referencing that global wavetable is created, the new contents of the global wavetable shall not be copied into the instrument instantiation. Also, if the contents of the local wavetable are modified, these changes shall not be reflected in the global wavetable.

If the **imports** and **exports** tags are both used, and there is a global wavetable with the same name, then at instrument instantiation time and at the beginning of each control pass, the current contents of the global wavetable are made available to a local wavetable with that name. "Made available" in the preceding sentence means that access may be either in the form of copying data from one wavetable to another or by pointer reference to the same memory space, or by any equivalent implementation. Also, at the end of instrument instantiation and at the end of each control pass, the current contents of the local wavetable are similarly made available to the global wavetable with the same name. In this case, if a wavetable is modified at the a-rate in one instrument instance, it is unspecified exactly when these changes are visible to other instrument instances, but it shall be no later than the next orchestra cycle (it is permitted to be in the same orchestra cycle).

It is not permissible to use the **exports** tag alone for a wavetable placeholder.

If the **imports** tag is used, and there is no global wavetable with the same name, then the reference is to a *future wavetable* that will be provided in the bitstream. When the instrument is instantiated, the contents of the most recent wavetable provided in the bitstream with the same name shall be copied into the local wavetable. If no wavetable has been provided in the bitstream with the same name as the wavetable placeholder at the time of instrument instantiation, then the bitstream is invalid. If the wavetable with this name is changed by providing a new wavetable with the same name by using a **table** line in the bitstream (subclause 5.11.6), the reference immediately changes to the new wavetable when the **table** line is dispatched.

It is not permissible to use the **exports** tag if there is no global wavetable with the same name.

5.8.6.5.5 Opcode array declaration

An opcode array, or "oparray" declaration, declares several opcode states for a particular opcode that may be used by the current instrument or opcode. By declaring the states in this manner, access to them is available through the oparray expression, see subclause 5.8.6.7.7. The identifier in the declaration shall be the name of a core opcode or a user-defined opcode declared elsewhere in the orchestra. The array length declares how many states are available for access to this oparray in the local code block; it shall be an integer value or the special tag **inchannels** or **outchannels.**

It is a syntax error if more than one oparray declaration references the same opcode name in a single instrument or opcode.

5.8.6.5.6 Table map definition

<table map definition> -> **tablemap <ident> (** <identlist> **)**

<identlist> as defined in subclause 5.8.5.4.

A table map is a data structure allowing indirect reference of wavetables via array notation. The identifier names the table map; it shall not be the same as the name of any other signal variable or other restricted word in the local scope. The identifier list gives a number of wavetable names for use with the table map. Each of these names shall correspond to a wavetable definition or wavetable placeholder within the current scope. The **tablemap** declaration may come before, after, or in the midst of wavetable declarations and wavetable placeholders in the instrument. All wavetables in the scope of the instrument may be referenced in a **tablemap**, regardless of the syntactic placement of the **tablemap**.

When the tablemap name is used in an array-reference expression (see subclause 5.8.6.7.5), the index of the expression determines to which of the wavetables in the list the expression refers. The first wavetable in the list is number 0, the second number 1, and so on.

EXAMPLE

For the following declarations

```
table t1(harm,2048,1);
imports table t2;
table t3(random,32,1);

tablemap tmap(t1,t2,t3,t2);
ivar i,x,y,z;
```

the following two code blocks are identical in semantics:

BLOCK 1

```
i = 3;
x = tableread(tmap[0],4);
y = tableread(tmap[i],3);
z = tableread(tmap[i > 4 ? 1 : 2],5);
```

BLOCK 2

```
x = tableread(t1,4);
y = tableread(t2,3);
z = tableread(t3,5);
```

Note that, like table references, array expressions using tablemaps may only occur in the context of an opcode or oparray call to an opcode accepting a wavetable reference.

5.8.6.6 Block of code statements

5.8.6.6.1 Syntactic form

<block> -> <statement> [<block>]
<block> -> **<NULL>**

<statement> -> <lvalue> **=** <expr> **;**
<statement> -> <expr> **;**
<statement> -> **if (** <expr> **) {** <block> **}**
<statement> -> **if (** <expr> **) {** <block> **} else {** <block> **}**
<statement> -> **while (** <expr> **) {** <block> **}**
<statement> -> **instr <ident> (** <expr list> **) ;**
<statement> -> **output (** <expr list> **) ;**
<statement> -> **spatialize (** <expr list > **) ;**
<statement> -> **outbus (<ident> ,** <expr list> **) ;**
<statement> -> **extend (** <expr> **) ;**
<statement> -> **turnoff ;**

<expr list> -> <expr> [**,** <expr list>]
<expr list> -> **<NULL>**

<lvalue> as given in subclause 5.8.6.6.2.
<expr> as given in subclause 5.8.6.7.

A block is a sequence of zero or more statements. A statement shall take one of several forms, which are enumerated and described in the subsequent subclauses. Each statement has rate-semantics rules governing the rate of the statement, the rate contexts in which it is allowable, and the times at which various subcomponents shall be executed.

To execute a block of statements at a particular rate, the statements within the block shall be executed, each at that rate, in such order as to produce equivalent results to executing the statements sequentially in linear order, according to the semantics below governing each type of statement.

5.8.6.6.2 Assignment

<statement> -> <lvalue> **=** <expr> **;**

<lvalue> -> **<ident>**
<lvalue> -> **<ident> [** <expr> **]**

<expr> as given in subclause 5.8.6.7.

An assignment statement calculates the value of an expression and changes the value of a signal variable or variables to match that value.

The lvalue, or left-hand-side value, denotes the signal variable or variables whose values are to be changed. An lvalue may be a local variable name, in which case the denotation is to the storage space associated with that name. An lvalue may also be a local array name, in which case the denotation is to the entire array storage space. An lvalue may also be a single element of a local array denoted by indexing a local array name with an expression. An lvalue shall not be a table reference or tablemap expression. An lvalue shall not be a standard name other than **MIDIctrl** (see subclause 5.8.6.8.9).

If the lvalue denotes an entire array, the right-hand-side expression of the assignment shall denote an array-valued expression with the same array length, or a single value, otherwise the construction is syntactically illegal.

If the lvalue denotes a single value, the right-hand-side expression of the assignment shall denote a single value, otherwise the construction is syntactically illegal.

The rate of the lvalue is the rate of the signal variable, if there is no indexing expression, or the faster of the rate of the signal array denoted by the signal variable and the rate of the indexing expression, if there is an indexing expression.

The rate of the right-hand side is the rate of the right-hand-side expression.

The rate of the statement is the rate of the lvalue, however, the statement is illegal if the rate of the right-hand side is faster than the rate of the lvalue.

The assignment shall be performed as follows:

At every pass through the statement occurring at equal rate to the rate of the assignment, the right-hand side expression shall be evaluated. Then, the storage space denoted by the lvalue shall be updated to be equal to the value of the right-hand expression. If the lvalue denotes an entire array, and the right-hand-side expression a single value, then each of the values of each of the elements of the array shall be changed to the single right-hand-side value.

5.8.6.6.3 Null assignment

<statement> -> <expr> **;**

A null assignment contains only an expression; it is provided so that opcodes that do not have useful return values need not be used in the context of an assignment to a dummy variable.

The rate of the statement is the rate of the expression. The expression may be single-valued or array-valued; it shall not be a table reference.

The null assignment shall be performed as follows:

At every pass through the statement occurring at equal rate to the rate of the statement, the expression shall be evaluated.

5.8.6.6.4 If

<statement> -> **if (** <expr> **) {** <block> **}**

An **if** statement allows conditional evaluation of a block of code. The expression that is tested in the **if** statement is termed the guard expression.

The rate of the statement is the rate of the guard expression, or the rate of the fastest statement in the guarded code block, whichever is faster.

It is not permissible for the block of code governed by the **if** statement to contain statements slower than the guard expression. It is further not permissible for any of the statements in the governed block of code to contain calls to opcodes that would be executed slower than the guard expression. The guard expression shall be a single-valued expression.

EXAMPLE

The following code fragment contains a rate-mismatch error:

```
asig a;
ksig k;

a = 0; if (a < 20) {
 k = kline(...);
}
```

The example is illegal because the **kline** assignment statement is slower than the guard **a < 20**. Even if the assignment were to an a-rate variable ("a2 = kline(...)"), thus making the assignment statement an a-rate statement, the example would be illegal, because the **kline** opcode itself is slower than the guard expression.

The **if** statement shall be executed as follows:

At every pass through the statement occurring at equal rate to the rate of the statement, the guard expression shall be evaluated. If the guard statement evaluates to any non-zero value in a particular pass, then the block of code shall be evaluated at the rate corresponding to that pass.

5.8.6.6.5 Else

<statement> -> **if (** <expr> **) {** <block> **} else {** <block> **}**

An **else** statement allows disjunctive evaluation of two blocks of code. The expression that is tested in the **else** statement is termed the guard expression.

The rate of the statement is the rate of the guard expression, or the rate of the fastest statement in the first guarded block of code, or the rate of the fastest statement in the second guarded block of code, whichever is fastest.

It is not permissible for the blocks of code governed by the **else** statement to contain statements slower than the guard expression. It is further not permissible for any of the statements in the governed blocks of code to contain calls to opcodes that would be executed slower than the guard expression. The guard expression shall be a single-valued expression.

The **else** statement shall be executed as follows:

At every pass through the statement occurring at equal rate to the rate of the statement, the guard expression shall be evaluated. If the guard expression evaluates to any non-zero value in a particular pass, then the first guarded block of code shall be at the rate corresponding to that pass. If the guard statement evaluates to zero in a particular pass, then each statement in the second guarded block of code shall be so evaluated.

5.8.6.6.6 While

<statement> -> **while (** <expr> **) {** <block> **}**

The **while** statement allows a block of code to be conditionally evaluated several times in a single rate pass. The expression that is tested in the **while** statement is termed the guard expression.

The rate of the **while** statement is the rate of the guard expression.

It is not permissible for the block of code governed by the **while** statement to contain statements that run at a rate other than the rate of the guard expression. It is further not permissible for any of the statements in the governed block of code to contain calls to opcodes that would be executed at a rate other than the rate of the guard expression. The guard expression shall be a single-valued expression. It is not permissible for the guard expression to contain calls to core opcodes with type **specialop**, see subclause 5.9.2.

The **while** statement shall be executed as follows:

At every pass through the statement occurring at equal rate to the rate of the statement, the guard expression shall be evaluated. If the guard expression evaluates to any non-zero value in a particular pass, then each statement in the guarded block of code shall be evaluated according to the particular rules for that statement, and then the guard expression re-evaluated, iterating until the guard expression evaluates to zero.

5.8.6.6.7 Instr

<statement> -> **instr <ident> (** <expr list> **) ;**

The **instr** statement allows an instrument instantiation to dynamically create other instrument instantiations, for layering or synthetic-performance techniques. It shall consist of an identifier referring to an instrument defined in the current orchestra, a time delay, a duration, and a list of expressions defining parameters to pass to the referenced instrument.

It is a syntax error if the number of expressions in the expression list is not two greater than the number of pfields accepted by the referenced instrument (the first expression is the time delay and the second is the duration). Each expression in the expression list shall be a single-valued expression.

The rate of the **instr** statement is the rate of the fastest expression in the expression list, or the rate of the guarding expression containing the statement, or the rate of the opcode containing the statement, whichever is fastest.

It is not permissible for the rate of the **instr** statement to be a-rate.

The **instr** statement shall be executed as follows:

At every pass through the statement occurring at equal rate to the rate of the statement, each of the expressions in the expression list is evaluated. Then, a new instrument event is registered with the scheduler as described in subclause 5.11.3. The dispatch time of the new instrument event is the sum of the current orchestra time and the value of the first expression in the expression list, the latter scaled by the current global tempo; the duration of the new instrument event is the value of the second expression in the expression list; and the values of the p-fields for the new instrument event are the values of the remaining expressions in the expression list.

An exception to the above occurs when the time-delay (the first expression in the expression list) is less than the length of the orchestra control period. In this case, an instrument event is not created, but a new instrument instantiation is immediately created, where the duration of the new instantiation is the value of the second expression in the expression list, and the values of the instrument p-fields in the new instantiation are set to the values of the remaining expressions.

In this case, the i-rate pass through the new instrument instantiation shall be executed immediately upon its creation, before any more statements from the block of code containing the **instr** statement are executed. However, any changes to global i-rate variables made in the new instance during its i-rate pass are not respected in this instrument (the “caller”) (i-rate variables imported from the global context are set only during the initialisation pass of each instance, and never change afterward). The first k-rate and a-rate passes through the new instrument instantiation shall be executed as appropriate to the sequencing relation between the instantiating and instantiated instruments; that is, if the new instrument is sequenced later than the instantiating instrument, the new instantiation shall be executed at some later time in the same orchestra pass, but if the new instrument is sequenced earlier than the instantiating instrument, then the new instantiation shall not be executed in k-time or a-time until the subsequent orchestra pass.

5.8.6.6.8 Output

<statement> -> **output (** <expr list> **) ;**

The **output** statement creates audio output from the instrument. This output does not get turned directly into sound, but rather gets buffered either on one or more busses based on instructions given in **route** statements (subclause 5.8.5.4) or on the special bus **output_bus** by default. However, if the current instrument instantiation is the one created with a **send** statement referencing the special bus **output_bus**, then the output of the current instantiation, created by summing its calls to **output**, may be turned directly into sound.

The expression list shall contain at least one expression.

The rate of the **output** statement is a-rate.

All statements within a orchestra that reference the same bus, whether through explicit sends, calls to **outbus**, or by default routing to the special bus **output_bus**, shall have compatible numbers of expression parameters representing output channels. “Compatible” means that if any calls to **output** for a particular bus reference more than one expression parameter, then all other calls to **output** referencing this bus shall have either the same number of expression parameters, or else only a single expression parameter. In addition, the number of channels of the special bus **output_bus** shall be the same as the global **outchannels** parameter and uses of **output** by instrument instances that are implicitly or explicitly routed to **output_bus** shall be compatible with this number of channels.

The **output** statement is executed as follows:

At each k-rate pass through the instrument, an output buffer, with number of channels determined by the rules in subclause 5.7.3.3.5.2, shall be cleared to zero values. At every a-rate pass through the statement, the expression parameters shall each be evaluated. Then, the expression parameter values shall be placed in the output buffer: if the **output** statement has more than one parameter expression, then the value of each parameter shall be added to the current value of the output buffer in the corresponding channel. If the **output** statement has only one parameter expression, then the value of that expression shall be added to the current value of the output buffer in each channel.

The expression parameters to the **output** statement may be array-valued, in which the mapping described in the preceding paragraph is not from expressions to buffer channels, but from array value channels to buffer channels.

EXAMPLE

The following code fragment

```
asig a[2], b;

. . .

output(a,b);
output(a[1],b,b);
output(b);
```

is legal and describes an instrument that outputs three channels of sound. The first channel of output contains the value `a[0] + a[1] + b`, the second `a[1] + b + b`, and the third `b + b + b`.

After each a-rate pass through the instrument instantiation during a particular orchestra pass, the values in the output buffer shall be added channel-by-channel to the current values of the bus or busses referenced by the **route** expression or expressions that also reference this instrument. If there are no such **route** statements, the values in the output buffer shall be added channel-by-channel to the current values of the special bus **output_bus**. If this is the instrument instantiation created by referencing the special bus **output_bus** in a **send** statement, then the preceding two sentences do not hold, and instead the values in the output buffer are the output of the orchestra.

5.8.6.6.9 Spatialize

<statement> -> **spatialize (** <expr list > **) ;**

The **spatialize** statement allows instruments to produce spatialised sound, using non-normative methods that are implementation-dependent.

The expression list shall contain four expressions. The second, third, and fourth shall not be a-rate expressions. The first expression represents the audio signal to be spatialised; the second, the azimuth (angle) from which the source sound shall apparently come, measuring in radians clockwise from 0 azimuth directly in front of the listener; the third, the elevation angle from which the sound source shall apparently come, measuring in radians upward from 0 elevation on the listener's horizontal place; and the fourth, the distance from which the sound source shall apparently come, measuring in metres from the listener's position. Each of the four expressions shall be single-valued.

The rate of the **spatialize** statement is a-rate.

The **spatialize** statement shall be executed as follows:

At each a-rate pass through the instrument, the expressions in the expression list shall be evaluated. Then, the sound signal in the first expression shall be presented to the listener as though it has arrived from the azimuth, elevation, and distance given in the second, third, and fourth expressions. No normative requirements are placed on this spatialisation capability, although terminal implementers are encouraged to provide the maximum sophistication possible.

The sound produced via the **spatialize** statement is turned directly into orchestra output; it shall not be affected by bus routings or further manipulation within the orchestra. If multiple calls to **spatialize** occur within an orchestra, the various sounds so produced shall be mixed via simple summation after spatialisation. Similarly, if both spatialised and non-spatialised sound is produced within an orchestra, the final orchestra output of all non-spatialised sound shall be mixed via simple summation with the various spatialised sounds for presentation. The sound produced via each **spatialize** statement shall have as many channels as the global orchestra number of output channels (see subclause 5.8.5.2.4) in order to enable this mixing.

5.8.6.6.10 Outbus

<statement> -> **outbus (<ident> ,** <expr list> **) ;**

The **outbus** statement allows instruments to place dynamically-calculated signals on busses. The identifier parameter shall refer to the name of a bus defined with a **send** statement in the global block. The remaining expressions represent signals to place on the bus.

It is a syntax error if there are no expressions in the expression list, or if the identifier does not refer to a bus defined in the global block with a **send** statement. The number of expressions in the expression list shall be compatible with other statements making reference to the same bus, as defined in subclause 5.8.6.6.8.

The rate of the **outbus** statement is a-rate.

The **outbus** statement shall be executed as follows:

At each a-rate pass through the statement, the expression list shall be evaluated. Then, the expression values shall be added to the current values of the referenced bus. If there is more than one expression in the expression list, then each expression value shall be added to the corresponding channel of the referenced bus. If there is only one expression in the expression list, then the value of that expression shall be added to each channel of the referenced bus.

The expressions in the expression list may be array-valued, in which case the semantics are analogous to those in subclause 5.8.6.6.8.

The **outbus** statement shall not be used in an instrument that is the target of a **send** statement referencing the special bus **output_bus**.

5.8.6.6.11 Extend

<statement> -> **extend (** <expr> **) ;**

The **extend** statement allows an instrument instantiation to dynamically lengthen its duration.

The expression parameter shall not be a-rate. The expression shall be single-valued.

The rate of the **extend** statement is the rate of the expression parameter, or the rate of the guarding expression containing the statement, or the rate of the opcode containing the statement, whichever is fastest.

The **extend** statement shall be executed as follows:

At each pass through the statement at equal rate to the rate of the statement, the expression shall be evaluated. Then, the duration of the instrument instantiation shall be extended by the amount of time, in seconds, given by the value of the expression. That is, if the instrument instance had been previously scheduled to be terminated at time *t*, then after a call to **extend** with an expression evaluating to *s*, the instrument instance shall be scheduled to terminate at time *t+s*. If the instrument instance had no scheduled termination time (its duration was –1 on instantiation), **extend** with an expression evaluating to *s* shall schedule termination of the instrument at time *T* + *s*, where *T* is the current orchestra time.

NOTE - The parameter of **extend** is specified in seconds, not in beats. If it is desirable to have time-extension dependant on tempo in a particular composition, the content author may enable this by rescaling the parameter by the current value of **gettempo()** (subclause 5.9.15.1).

extend may be called with a negative argument to shorten the duration of a note; if *t+s* **<** *T* (that is, if the negatively extended duration has already been exceeded in the instantiation), then the statement acts as the **turnoff** statement, see subclause 5.8.6.6.12.

When the **extend** statement is called, the standard name **dur** shall be updated to reflect the new duration; that is, **dur := dur + x** where **x** is the expression value of the parameter.

5.8.6.6.12 Turnoff

<statement> -> **turnoff ;**

The **turnoff** statement allows an instrument instantiation to dynamically decide to terminate itself.

The rate of the **turnoff** statement is k-rate.

The **turnoff** statement shall be executed as follows:

When the **turnoff** statement is reached at k-rate, the instrument instance shall be scheduled to terminate after the following k-cycle; that is, if the current orchestra time is *T* and the k-pass duration *k*, the instrument instantiation shall be scheduled to terminate at time *T*+*k*.

The **turnoff** statement shall not update the **dur** standard name.

The **turnoff** statement shall not be executed in an instrument instance that is created as the result of a **send** statement referencing the special bus **output_bus**.

NOTE - **turnoff** does not destroy the instantiation immediately; the instantiation is executed for one more orchestra pass, to allow the instrument time to examine the **released** variable. Instruments may call **turnoff** and then "save" themselves on the subsequent k-cycle by calling **extend**.

5.8.6.7 Expressions

5.8.6.7.1 Syntactic form

<expr> -> **<ident>**
<expr> -> **<number>**
<expr> -> **<int>**
<expr> -> **<ident> [** <expr> **]**
<expr> -> **<ident> (** <expr list> **)**
<expr> -> **<ident> [** <expr> **] (** <expr list> **)**
<expr> -> <expr> **?** <expr> **:** <expr>
<expr> -> <expr> <binop> <expr>
<expr> -> **!** <expr>
<expr> -> **-** <expr>
<expr> -> **(** <expr> **)**
<expr> -> **sasbf (** <expr list> **) ;**

<binop> -> **+**
<binop> -> **-**
<binop> -> *****
<binop> -> **/**
<binop> -> **==**
<binop> -> **>=**
<binop> -> **<=**
<binop> -> **!=**
<binop> -> **>**
<binop> -> **<**
<binop> -> **&&**
<binop> -> **||**

An expression can take one of several forms, the semantics of which are enumerated in the subclauses below. Each form has both rate semantics, which describe the rate of the expression in terms of the rates of the subexpressions, and value semantics, which describe the value of the expression in terms of the values of the subexpressions. The syntax above is ambiguous for many expressions; disambiguating precedence rules are given in subclause 5.8.6.7.14.

5.8.6.7.2 Properties of expressions

Each expression is conceptually labelled with two properties: its rate and its width. The rate of an expression determines how fast the value of that expression might change; the width of an expression determines how many channels of sound or other data are represented by the expression. In each expression type, the rate and width of the expression are determined from the type of the expression, and perhaps from the rate and width of the component subexpressions.

NOTE - Any name declared as an array is an array-valued variable regardless of its length. That is, a variable declared as **asig name[1]** is not a single-valued variable.

5.8.6.7.3 Identifier

<expr> -> **<ident>**

An identifier expression denotes a storage location or locations that contain values stored in memory. It is illegal to reference an identifier that is not declared in the local instrument or opcode scope, and that is not a standard name (see subclause 5.8.6.7.14).

The rate of an identifier expression is the rate type at which the identifier was declared, or is implicitly declared in the case of standard names. The rate of a **table** identifier is i-rate.

If the identifier denotes a single-valued name (i.e., one that is not an array type), then the value of the identifier expression is the value stored in memory associated with that identifier in the current scope, and the width of the expression is 1.

If the identifier denotes an array-valued name, then the value of the identifier expression is the ordered sequence of values stored in memory and associated with that identifier in the current scope, and the width of the expression is the width of the array so denoted.

If the identifier denotes a table, then the value of the identifier expression is a reference to the table with the given name. Table references may only appear in calls to opcodes. A table reference has width 1.

5.8.6.7.4 Constant value

<expr> -> **<number>**
<expr> -> **<int>**

A constant value expression denotes a single number.

The rate of a constant value expression is i-rate.

The width of a constant value expression is 1.

The value of a constant expression is the value of the number denoted by the constant. The value of a constant expression is always a floating-point value, whether the token or lexical expression denoting the value was an integer or floating-point token or expression.

5.8.6.7.5 Array reference

<expr> -> **<ident> [** <expr> **]**

An array reference expression allows the selection of one value from an array of several. The identifier in the array-reference syntax is termed the array name, and the expression the index expression. It is illegal to use an identifier in an array reference that is neither declared in the local instrument or opcode scope as an array, nor implicitly defined as an array-valued standard name or table map.

The index expression shall have width 1.

The rate of an array reference expression is the rate of the array name (which is the rate at which the array name was declared explicitly or implicitly), or the rate of the index expression, whichever is faster.

The width of an array reference expression is 1.

If the referenced array is an array-valued signal variable, then the value of the array reference expression is the value of that element of the sequence of values in the array storage corresponding to the value of the indexing expression, where element 0 corresponds to the first value in the sequence. It is a run-time error if the value of the indexing expression is less than 0, or equal to or greater than the declared size of the array. If the indexing expression is not an integer, it is rounded to the nearest integer.

If the referenced array is a table map, then the value of the array reference expression is a reference to that element of the sequence of tables corresponding to the value of the index expression, where element 0 corresponds to the first table in the sequence. It is a run-time error if the value of the indexing expression is less than 0, or equal to or greater than the declared size of the table map. If the indexing expression is not an integer, it is rounded to the nearest integer. Table references may only appear in calls to opcodes. See also the example in subclause 5.8.6.5.6.

NOTE - The syntax `t[i]`, where **t** is a table rather than a table map, is illegal. The **tableread** core opcode is used to directly access elements of a wavetable. See subclause 5.9.6.

5.8.6.7.6 Opcode call

<expr> -> **<ident> (** <expr list> **)**

An opcode call expression allows the use of processing functionality encapsulated within an opcode.

The identifier is termed the opcode name, and the expression list the actual parameters of the opcode call expression. It is illegal to use an identifier that is not the name of a core opcode and is also not the name of a user-defined opcode declared elsewhere in the orchestra. For user-defined opcodes, the number of actual parameters shall be the same as the number of formal parameters in the opcode definition. For core opcodes without variable argument lists, the number of actual parameters required varies from opcode to opcode; see subclause 5.8.9. If a particular formal parameter in an opcode definition is an array, then the corresponding actual parameter shall be an array-typed expression of equal width. If a particular formal parameter in an opcode definition is a table, then the corresponding actual parameter shall be a table reference.

If a particular formal parameter in an opcode definition is at a particular rate, then the corresponding actual parameter expression shall not be at a faster rate.

The rate of the opcode call expression is determined according to the rules in subclause 5.8.7.7.

The width of the opcode call expression is the number of channels provided in the **return** statements in the opcode's code block.

For calls to core opcodes (see clause 5.9), in the absence of normative language specifying otherwise for a particular opcode, it is a syntax error if any of the following statements apply:

- there are fewer actual parameters in the opcode call than required formal parameters
- there are more actual parameters in the opcode call than required and optional formal parameters, and the opcode definition does not include a varargs "..." clause.
- a particular actual parameter expression is of faster rate than the corresponding formal parameter, or than the varargs formal parameter if that is the correspondence
- a particular actual parameter expression is not single-valued, or is not table-valued when the corresponding formal parameter specifies a table.

The context of the opcode call is restricted more than other expressions. When occurring within a block subsidiary to a guarding statement (**if**, **else**, or **while**), opcode calls shall not have a rate slower than the rate of the guarding expression (see subclauses 5.8.6.6.4 and 5.8.6.6.5 and 5.8.6.6.6). A call to an opcode with a particular name shall not occur within the code block of definition of that opcode, nor within the code blocks of any of the opcodes called by that opcode, or any of the opcodes called by them, etc. That is, recursive and mutually-recursive opcodes are prohibited.

To calculate the value of an opcode call expression referencing a user-defined opcode at a particular rate, the values of the actual parameter expression shall be calculated in the order they appear in the expression list. The values of the formal parameters within the opcode scope shall be set to the values of the corresponding actual parameter expressions. If this is the first opcode call expression referencing this opcode scope, opcode storage space shall be created to store local signal variables and wavetables, the local signal variables set to 0, and the local wavetables created as discussed in subclause 5.8.6.5.2. Any global variables imported by the opcode at that rate shall be copied into the opcode storage space. The statement block of the opcode shall be executed at the current rate. The value of the opcode call expression is the value of the first **return** statement encountered when executing the opcode. The value of the opcode call expression may be array-valued (if the expression in the **return** statement is). After the end of opcode execution, any global variables exported by the opcode shall be copied into the global storage space.

NOTE - If an opcode changes and exports the value of a global variable that is imported by the calling instrument or opcode, the change in the global variable is not reflected in the caller until the next orchestra pass.

If a particular actual parameter expression in an opcode call expression is an identifier or an array-reference expression, then that parameter is a reference parameter in that call to that opcode. When the opcode statement block is executed, the final value of the formal parameter associated with that actual parameter shall be copied into the variable value denoted by the identifier or array-reference, unless the actual parameter is a standard name, in which case no copy is performed. This modification shall happen immediately after (but not until) the termination of the statement block, before any other calculation is done. Both single-value and array-value expressions may be reference parameters, but if an array-valued expression is used, the associated formal parameter shall be an array of the same length.

To calculate the value of an opcode call expression referencing a core opcode at a particular rate, the values of the actual parameter expressions shall be calculated in the order they appear in the expression list. Then, the return value of the core opcode shall be calculated according to the rules for the particular opcode given in subclause 5.8.9.

NOTE - The variables declared within the scope of a user-defined opcode are static-valued; that is, they preserve their values from call to call. The values of variables within the scope of a user-defined opcode are set to 0 before the opcode is called the first time. Each syntactically distinct call to an opcode creates one and only one opcode scope (see example in next subclause).

5.8.6.7.7 Oparray call

<expr> -> **<ident> [** <expr> **] (** <expr list> **)**

An oparray call expression allows the dynamic selection of an opcode state from a set of several, and the calculation of encapsulated functionality with respect to that opcode state.

The identifier is termed the opcode name, the expression in brackets is termed the index expression, and the expressions in the parameter list are termed the actual parameters. It is illegal to use an identifier that is not the name of a core opcode and is also not the name of a user-defined opcode declared elsewhere in the orchestra. It is also illegal to use an identifier for which oparray storage is not allocated in the local scope as described in subclause 5.8.6.5.5. For user-defined opcodes, the number of actual parameters shall be the same as the number of formal parameters in the opcode definition. For core, the number of actual parameters required varies from opcode to opcode; see subclause 5.8.9.

The index expression shall be a single-valued expression.

The rate of the oparray call expression is the rate of the opcode referenced, as determined by the rules in subclause 5.8.7. The rate of the index expression shall not be faster than the rate of the opcode referenced.

The width of the oparray call expression is the number of channels returned by **return** statements within the opcode code block.

The context of the oparray call expression is restricted in the same way as described for the opcode call expression in subclause 5.8.6.7.6.

The value of the oparray call expression is determined in the same way as described for the opcode call in subclause 5.8.6.7.6, with the following exceptions and additions:

Before the values of the actual parameter expressions are calculated, the value of the index expression is calculated. It is a run-time error if the value of the index expression is not in the range [0..n-1], where n is the allocation size in the oparray definition for this oparray. If the index expression is not an integer, it is rounded to the nearest integer. The scope storage associated with the opcode name and the value of the index expression is selected from the set of oparray scopes in the local scope. The evaluation of the statement block in the referenced opcode is with regard to the selected scope. Within each oparray scope, local variables retain their values from call to call.

EXAMPLES

Some examples are provided to clarify the distinction between opcode calls and oparray calls.

The following user defined opcode

```
kopcode inc() {
  ksig ct;

  ct = ct + 1;
  return(ct);
  }
```

counts the number of times it is called.

1. After the first execution of the following code fragment

```
a = inc();
b = inc();
```

the value of **a** is 1, and the value of **b** is 1, since each call to **inc()** refers to a different scope.

2. After the first execution of the following code fragment

```
i = 0; while (i < 2) { a = inc(); i = i + 1; }
```

the value of **a** is 2, since there is only one scope for **inc().**

3. After the first execution of the following code fragment

```
oparray inc[2];

a = inc[0]();
b = inc[0]();
```

the value of **a** is 1, and the value of **b** is 2, since each call to **inc()** refers to the same scope (since the value of the indexing expression is the same in both calls).

4. After the first execution of the following code fragment

```
oparray inc[2];

i = 0; while (i < 2) { a = inc[i](); i = i + 1; }
```

the value of **a** is 1, since each iteration refers to a different scope in the call to **inc()** (since the value of the indexing expression is 0 on the first iteration, and 1 on the second).

NOTE - Opcode calls and oparray calls referencing the same opcode may be used in the same scope. In this case, the scopes referenced by each of the opcode calls are different from any of the scopes defined in the oparray definition.

5.8.6.7.8 Combination of vector and scalar elements in mathematical expressions

The subsequent subclauses (subclauses 5.8.6.7.9 through 5.8.6.7.13) describe mathematical expressions in SAOL. For each, the width of the expression is the maximum width of any of its subexpressions. For each expression type, each subexpression within an expression shall have the same width, or else width of 1. If subexpressions with width 1 and width different than 1 are combined in an expression, before the expression is computed, the subexpression(s) with width 1 shall be promoted to have the same width as the expression. That is, a width 1 expression with value **x** that is a subexpression of a width *n* expression shall be promoted to a width *n* expression where the value of each element is **x**.

For each expression type below, the semantics will be given for array-valued expressions. In each case, the semantics for the single-valued expression are the same as for an array-valued expression with width 1, except for the special cases of switch, logical AND, and logical OR, which will be described separately in those subclauses.

5.8.6.7.9 Switch

<expr> -> <expr> **?** <expr> **:** <expr>

The switch expression combines values from two subexpressions based on the value of a third.

The rate of the switch expression is the rate of the fastest of the three subexpressions.

The value of the switch expression is calculated as follows: the three subexpressions are evaluated. Then, for each value of the first subexpression, if this value is non-zero, the corresponding value of the switch expression is the corresponding value of the second subexpression. If this value is zero, the corresponding value of the switch expression is the corresponding value of the third subexpression.

In the special case where all subexpressions have width 1, then the switch expression “short-circuits”: the first subexpression is evaluated, and if its value is non-zero, then the second subexpression is evaluated, and its value is the value of the switch expression. If the value of the first subexpression is zero, then the third subexpression is evaluated, and its value is the value of the switch expression. If the width of the switch expression is 1, then in no case are both the second and third subexpressions evaluated.

5.8.6.7.10 Not

<expr> -> **!** <expr>

The not expression performs logical negation on a subexpression.

The rate of the not expression is the rate of the subexpression.

The value of the not expression is calculated as follows: the subexpression is evaluated. For each nonzero value in the subexpression, the corresponding value of the not expression is zero; for each zero value in the subexpression, the corresponding value of the not expression is 1.

5.8.6.7.11 Negation

<expr> -> **-** <expr>

The negation expression performs arithmetic negation on a subexpression.

The rate of the negation expression is the rate of the subexpression.

The value of the negation expression shall be calculated as follows: the subexpression is evaluated. For each value in the subexpression, the corresponding value of the negation expression is the arithmetic negative of the value.

5.8.6.7.12 Binary operators

<expr> -> <expr> <binop> <expr>

There are 12 binary operators. Each of them calculates a different function on binary subexpressions.

The value of the expression shall be calculated as follows. The two subexpressions shall be evaluated, and for each pair of values of the subexpressions, and the corresponding value of the binary expression shall be calculated according to the following table, where x_1 and x_2 are the values of the first and second subexpressions:

Table 5.2 - Binary operators

Operator	Value of expression
+	$x_1 + x_2$
-	$x_1 - x_2$
*	$x_1 x_2$
/	x_1 / x_2
==	if $x_1 = x_2$, then 1, otherwise 0
>	if $x_1 > x_2$, then 1, otherwise 0
<	if $x_1 < x_2$, then 1, otherwise 0
<=	if $x_1 \leq x_2$, then 1, otherwise 0
>=	if $x_1 \geq x_2$, then 1, otherwise 0
!=	if $x_1 \neq x_2$, then 1, otherwise 0

In each of these cases, if the particular operation would result in a NaN or Inf result (for example, division by 0), a run-time error shall result.

For the “logical and” operator **&&** in the special case where both subexpressions have width 1, the expression is calculated in a “short-circuit” fashion. The first subexpression shall be evaluated. If its value is 0, then the value of the expression is 0; if its value is nonzero, then the second subexpression shall be evaluated, and if its value is 0, then the value of the expression is 0, otherwise the value of the expression is 1.

For the “logical or” operator || in the special case where both subexpressions have width 1, the expression is calculated in a “short-circuit” fashion. The first subexpression shall be evaluated. If its value is nonzero, then the value of the expression is 1; if its value is 0, then the second subexpression shall be evaluated, and if its value is nonzero, then the value of the expression is 1, otherwise the value of the expression is 0.

5.8.6.7.13 Parenthesis

<expr> -> **(** <expr> **)**

The parenthesis operator performs no new calculation, but allows the specification of arithmetic grouping.

The rate of the parenthesis expression is the rate of the subexpression.

The width of the parenthesis expression is the width of the subexpression.

The value of the parenthesis expression is the value of the subexpression.

5.8.6.7.14 Order of operations

Expressions bind in the order prescribed in the following table. That is, operations listed higher in the table are performed before operations lower in the table whenever the ordering is syntactically ambiguous. Operations listed on the same row associate left-to-right. That is, the leftmost expression is performed first.

Table 5.3 - Order of operations

Operator	Function
!	not
-	unary negation
*, /	multiply, divide
+, -	add, subtract
<, >, <=, >=	relational
==, !=	equality
&&	logical and
||	logical or
?:	switch

5.8.6.7.15 SASBF synthesis

<expr> -> **sasbf (** <expr list> **) ;**

The **sasbf** expression allows the use of the DLS-compatible bank synthesis procedure (see subclause 5.13) within a SAOL instrument. It shall not be used in a Object type 3 bitstream and capability for executing it does not have to be provided by a Object type 3 decoder. The parameter list shall have two, three, or four expressions. All shall be single-valued i-rate expressions.

1. The first expression shall correspond to the MIDI pitch desired for synthesis. If this value is not an integer, it shall be rounded to the nearest integer. It is a run-time error if this value is less than 1 or greater than 128.
2. The second expression shall correspond to the MIDI velocity desired for synthesis. If this value is not an integer, it shall be rounded to the nearest integer. It is a run-time error if this value is less than 0 or greater than 128.
3. The third expression, if given, corresponds to the MIDI preset number. If there are less than three expressions, the MIDI preset number is the default value given by **p** mod 128, where **p** is the preset number of the instrument to which the MIDI noteon that created the note instance was directed, or the lowest-numbered instrument preset number for the instrument containing this statement (subclause 5.8.6.4), if the note was not triggered with a MIDI event.
4. The fourth expression, if given, corresponds to the MIDI bank number. If there are less than four expressions in the list, the MIDI bank number is the default value given by floor(**p/128)** + 1, where **p** is the preset number to which the MIDI noteon that created the note instance was directed, or the lowest-numbered instrument preset number for the instrument (subclause 5.8.6.4), if the note was not triggered by a MIDI event. It is a syntax error if there are less than four expressions and no instrument preset number is provided.

The **sasbf** expression is an a-rate expression. The **sasbf** expression has two channels defined by the bank-synthesis procedure as described in subclause 5.13.

The value of the **sasbf** expression is calculated as follows:

On the first execution of the expression, each expression in the expression list shall be evaluated. Using these values, one note of synthesis shall be dispatched to the wavetable bank synthesis procedure described in clause 5.13.

On the first and each subsequent a-rate pass through the **sasbf** expression, the value of the expression shall be the next pair of audio samples from the stereo wavetable bank synthesis process.

During the wavetable bank synthesis process, the values of MIDI controllers and other continuous values on the current channel and note shall be respected. These values are not passed into the **sasbf** expression list, but are made available to the SASBF synthesiser in an implementation-dependent way. If the values of the global MIDIctrl[] standard name are changed by any instrument (see subclause 5.8.6.8.9), the new values shall be respected by all SASBF synthesis processes.

NOTE - Certain MIDI instructions (such as the Registered Parameter Number mechanism, see [MIDI]), cannot be properly understood on an event-by-event basis; rather, their semantics are understood to take effect when the entire RPN change is completed. To ensure this, all control events, whether or not they have SAOL semantics (see subclause 5.14.3.2), shall be passed through the SAOL scheduler to the **sasbf** synthesis processes in their original order.

If the instrument instance containing a particular **sasbf** expression was not instantiated in response to a MIDI event, then that instance is on no channel, and so the SASBF synthesis for that expression cannot be controlled by MIDI-based continuous controllers.

Each syntactically different instance of the **sasbf** expression results in one instance of a SASBF note synthesis procedure. There is no mechanism for interleaving samples from a single **sasbf** expression in multiple lines, or for instantiating multiple bank-synthesis procedures with one syntactic expression. **sasbf** is not an opcode and is not permitted to be used as an oparray construction (see subclause 5.8.6.7.7).

The value of the **released** standard name in the instrument instance (see subclause 5.8.6.8.16) containing the call to **sasbf** shall be made available to each SASBF process in an implementation-dependent way. The SASBF process shall use this flag to determine when to begin synthesis of the release portion of the given note. If a particular SASBF instance needs to extend the duration past the release time, it shall extend the note by one k-cycle in the manner of the **extend** statement (subclause 5.8.6.6.11). If, on the next k-cycle, the SASBF instance is still not finished, it may extend the note by a further k-cycle, and so on.

If multiple SASBF instances in an instrument each require extended duration, together they shall extend the duration by one k-cycle; the duration shall not be extended by one k-cycle per SASBF instance.

The SASBF synthesis process for each note terminates when the instrument instance containing this **sasbf** expression is destroyed. There is no mechanism for ending the SASBF synthesis earlier than this.

EXAMPLE

The following instrument layers two notes using the wavetable synthesiser and filters the result.

```
instr layer(mp, vel) preset 12 {
  asig a1[2], a2[2], i;
  oparray bandpass[2];

  a1 = sasbf(mp,vel);
  a2 = a1 + sasbf(mp,vel,47,2) / 2;   // second note quieter
  i = 0;
  while (i < 2) {
    a2[i] = bandpass[i](a2[i],400,100);
    i = i + 1;
    }
  output(a2);
}
```

The two instances of **sasbf** operate simultaneously and in parallel. The first synthesises sound from preset 13, bank 1 (since this is the preset number to which the instrument responds); the second, from preset 48, bank 2. They return audio signals that are summed together and manipulated with the **bandpass** core opcode.

5.8.6.8 Standard names

5.8.6.8.1 Definition

Not all identifiers to be referenced in an instrument or opcode are required to be declared as variables. Several identifiers, listed in this subclause, are termed standard names, shall not be used as variables, and have fixed semantics that shall be implemented in a compliant SAOL decoder. Standard names may otherwise be used as

variables, embedded in expressions, etc. in any SAOL instrument or opcode. However, the semantics of using a standard name as an lvalue are undefined.

The implicit definition of each standard name, showing the rate semantics and width of that standard name, is listed, and the semantics of the value of the standard name specified in the subsequent subclauses.

5.8.6.8.2 k_rate

```
ivar k_rate
```

The standard name **k_rate** shall contain the control rate of the orchestra, in Hz.

5.8.6.8.3 s_rate

```
ivar s_rate
```

The standard name **s_rate** shall contain the sampling rate of the orchestra, in Hz.

5.8.6.8.4 inchan

```
ivar inchan
```

The standard name **inchan**, in each scope, shall contain the number of channels of input being provided to the instrument instantiation with which that scope is associated. "Associated" shall be taken to mean, for instrument code, the instrument instantiation for which the scope memory was created; for opcode code, the instrument instantiation that called the opcode, or called the opcode's caller, etc.

Different instances of the same instrument may have different numbers of input channels if, for example, they are the targets of different send statements. Instructions for calculating the value of this standard name are provided in subclause 5.7.3.3.5.2

5.8.6.8.5 outchan

```
ivar outchan
```

The standard name **outchan** shall contain the number of channels of output being produced by the orchestra (not by the instrument instance).

5.8.6.8.6 time

```
ivar time
```

The standard name **time**, in each scope, shall contain the time at which the instrument instantiation associated with that scope was created.

NOTE - If the "event time" of an instrument (for example, a score event more precisely timed than one control period) and the actual instantiation time differ, the name time shall contain the latter time, not the former.

5.8.6.8.7 dur

```
ivar dur
```

The standard name **dur**, in each scope, shall contain the duration of the instrument instantiation as originally created, and as potentially revised by use of the **extend** statement (subclause 5.8.6.6.11), or –1 if the duration was not known at instantiation.

Although **dur** is an i-rate variable, it may be changed during the duration of an instrument instance through the **extend** statement or through tempo changes. In this case, the value of expressions at the **ivar** do not change; expressions are only evaluated according to the rules in subclause 5.8.6.6.

5.8.6.8.8 itime

```
ksig itime
```

The standard name **itime**, in each scope, shall contain the elapsed time of the instrument instance. That is, on the first k-cycle through an instrument, **itime** shall be 0, and thereafter shall be incremented by 1/**KR**, where **KR** is the orchestra sampling rate, at the beginning of each k-rate pass.

5.8.6.8.9 MIDIctrl

```
ksig MIDIctrl[128]
```

The **MIDIctrl** standard variable shall contain, for each scope, the current values of the MIDI controllers on the channel corresponding to the channel to which the instrument instantiation associated with that scope is assigned. See clause 5.13 for more details on MIDI control of orchestras.

Instruments may use **MIDIctrl** as an lvalue, that is, to assign new values to it using the **=** statement (subclause 5.8.6.6.2). In this case, when an instrument assigns to **MIDIctrl**, the value for the indicated controller shall be changed on the channel to which the instrument instance associated with that scope is assigned. The value of **MIDIctrl** is changed in all other instrument instances associated with that channel to the new value, and this change shall take effect the next time each of these instrument instances is executed at the k-rate (see subclause 5.7.3.3.6, list item 10).

MIDIctrl[64] is a special controller. It is normatively defined as the sustain controller, and has a special relationship with the MIDI **noteoff** instruction, see subclause 5.14.3.2.4.

5.8.6.8.10 MIDItouch

```
ksig MIDItouch
```

The **MIDItouch** standard variable shall contain, for each scope, the current value of the MIDI aftertouch on the note that caused the associated instrument instantiation to be created. See clause 5.13 for more details on MIDI control of orchestras.

5.8.6.8.11 MIDIbend

```
ksig MIDIbend
```

The **MIDIbend** standard variable shall contain, for each scope, the current value of the MIDI pitchbend on the channel corresponding to the channel to which the instrument instantiation associated with that scope is assigned.

5.8.6.8.12 channel

```
ivar channel
```

The **channel** standard name contains the extended MIDI channel of the note responsible for creating the current instrument instance. See subclause 5.14.3.2.2.

5.8.6.8.13 preset

```
ivar preset
```

The **preset** standard name contains the SAOL preset number (which is one less than the MIDI preset number) of the note responsible for creating the current instrument instance. The **preset** standard name does not contain “all” of the preset numbers for the current instrument, only the one that led to the instantiation of the current instance.

5.8.6.8.14 input

```
asig input[inchannels]
```

The **input** standard variable shall contain, for each scope, the input signal or signals being provided to the instrument instantiation through the **send** instruction. See subclause 5.8.5.5.

5.8.6.8.15 inGroup

```
ivar inGroup[inchannels]
```

The **inGroup** standard variable shall contain, for each scope, the grouping of the input signals being provided to the instrument instantiation. See subclause 5.8.5.5.

5.8.6.8.16 released

```
ksig released
```

The **released** standard name shall contain, for each scope, 1 if and only if the instrument instantiation associated with the scope is scheduled to be destroyed at the end of the current orchestra pass. Otherwise, **released** shall contain 0. See subclause 5.7.3.3.6, list item 3.

5.8.6.8.17 cpuload

```
ksig cpuload
```

The **cpuload** standard name shall contain, for each scope, a measure of the recent CPU load on the CPU most strongly associated with the instrument instantiation associated with the scope. If the instrument instantiation is running entirely on one CPU, then that CPU shall be measured; if the instrument instantiation is running on multiple CPUs, then the exact measurement procedure is nonnormative.

The measure of CPU load shall be as a percentage of real-time capability: if the CPU is entirely loaded and cannot perform any more calculations without slipping out of real-time performance, the value of **cpuload** shall be 1 on that CPU at that k-cycle. If the CPU is entirely unloaded and is not performing any calculations, the value of **cpuload** shall be 0 on that CPU at that k-cycle. If the CPU is half-loaded, and could perform twice as many calculations in real-time as it is currently performing, the value of **cpuload** shall be 0.5 on that CPU at that k-cycle.

The exact calculation method, time window, recency, etc. of the CPU load is left to implementers.

5.8.6.8.18 position

```
imports ksig position[3]
```

The **position** name contains the absolute position of the node responsible for creating the current orchestra in the BIFS scene graph (see ISO/IEC 14496-1 clause 9, particularly subclause 9.4.2.82). The position is given by the current value of the **position** field of the **Sound** node that is the ancestor of this node in the scene graph, as transformed by its ancestors (that is, the final position in world co-ordinates of the **Sound** node). The value is global and shared by all instruments; it may not be changed by the orchestra.

5.8.6.8.19 direction

```
ksig direction[3]
```

The **direction** name contains the orientation of the node responsible for creating the current orchestra in the BIFS scene graph (see ISO/IEC 14496-1 clause 9, particularly subclause 9.4.2.82). The direction is given by the current value of the **direction** field of the **Sound** node that is the ancestor of this node in the scene graph, as transformed by its ancestors (that is, the final direction in world co-ordinates of the **Sound** node). The value is global and shared by all instruments; it may not be changed by the orchestra.

5.8.6.8.20 listenerPosition

```
ksig listenerPosition[3]
```

The **listenerPosition** name contains the absolute position of the listener in the BIFS scene graph (see ISO/IEC 14496-1 clause 9, particularly subclause 9.4.2.82). The position is given by the current value of the **position** field of the active **ListeningPoint** node in the scene graph, as transformed by its ancestors (that is, the final position in world co-ordinates of the **ListeningPoint** node).

5.8.6.8.21 listenerDirection

```
ksig listenerDirection[3]
```

The **listenerDirection** name contains the orientation of the listener in the BIFS scene graph (see ISO/IEC 14496-1 clause 9, particularly subclause 9.4.2.82). The direction is given by the current value of the **direction** field of the active **ListeningPoint** node in the scene graph, as transformed by its ancestors (that is, the final direction in world co-ordinates of the **ListeningPoint** node).

5.8.6.8.22 minFront

```
ksig minFront
```

The **minFront** standard name gives one parameter of the sound radiation pattern of the sound that the current node is a part of. This parameter, and its semantics, are defined by the **minFront** field of the **Sound** node of which this node is an ancestor (see ISO/IEC 14496-1 clause 9, particularly subclause 9.4.2.82).

5.8.6.8.23 maxFront

```
ksig maxFront
```

The **maxFront** standard name gives one parameter of the sound radiation pattern of the sound that the current node is a part of. This parameter, and its semantics, are defined by the **maxFront** field of the **Sound** node of which this node is an ancestor (see ISO/IEC 14496-1 clause 9, particularly subclause 9.4.2.82).

5.8.6.8.24 minBack

```
ksig minBack
```

The **minBack** standard name gives one parameter of the sound radiation pattern of the sound that the current node is a part of. This parameter, and its semantics, are defined by the **minBack** field of the **Sound** node of which this node is an ancestor (see ISO/IEC 14496-1 clause 9, particularly subclause 9.4.2.82).

5.8.6.8.25 maxBack

```
ksig maxBack
```

The **maxBack** standard name gives one parameter of the sound radiation pattern of the sound that the current node is a part of. This parameter, and its semantics, are defined by the **maxBack** field of the **Sound** node of which this node is an ancestor (see ISO/IEC 14496-1 clause 9, particularly subclause 9.4.2.82).

5.8.6.8.26 params

```
imports exports ksig params[128]
```

The **params** standard name is shared globally by all instruments. At each k-cycle of the orchestra, it shall contain the current values of the **params** field of the BIFS **AudioFX** node responsible for instantiating the current orchestra. If the orchestra is created by an **AudioSource** node rather than an **AudioFX** node, the value of **params** shall be 0 on every channel. See subclause 5.15.3 for more details.

5.8.7 Opcode definition

5.8.7.1 Syntactic Form

This subclause describes the definition of new opcodes. Bitstream authors may create their own opcodes according to these rules in order to encapsulate functionality and simplify instruments and the content authoring process.

<opcode definition> -> <opcode rate> **<ident> (** <formal param list> **) {**
<opcode var declarations>
<opcode statement block>
}

<opcode rate> -> **aopcode**
<opcode rate> -> **kopcode**
<opcode rate> -> **iopcode**
<opcode rate> -> **opcode**

An opcode definition has several elements. In order, they are

1. A rate tag that defines the rate at which the opcode executes, or indicates that the opcode is rate-polymorphic,

2. An identifier that defines the name of the opcode,
3. A list of zero or more formal parameters of the opcode,
4. A list of zero or more opcode variable declarations,
5. A block of statements defining the executable functionality of the opcode.

5.8.7.2 Rate tag

The rate tag describes the rate at which the opcode is to run, or else indicates that the opcode is rate-polymorphic. The four rate tags are

1. **iopcode**, indicating that the opcode runs at i-rate,
2. **kopcode**, indicating that the opcode runs at k-rate,
3. **aopcode**, indicating that the opcode runs at i-rate,
4. **opcode**, indicating that the opcode is rate-polymorphic.

See subclause 5.8.7.7 for instructions on determining the rate of a rate-polymorphic opcode.

5.8.7.3 Opcode name

Any identifier may serve as the opcode name except that the opcode name shall not be a reserved word (see subclause 5.8.8)., the name of one of the core opcodes listed in clause 5.9, or the name of one of the core wavetable generators listed in clause 5.10 An opcode name may be the same as the name of a variable in local or global score; there is no ambiguity so created, since the contexts in which opcode names may occur are very restricted.

No two instruments or opcodes in an orchestra shall have the same name.

5.8.7.4 Formal parameter list

5.8.7.4.1 Syntactic form

<formal param list> -> <formal param> [**,** <formal param list>]
<formal param list> -> **<NULL>**

<formal param> -> <opcode variable rate> <name>
<formal param> -> **table <ident>**

<opcode variable rate> -> **asig**
<opcode variable rate> -> **ksig**
<opcode variable rate> -> **ivar**
<opcode variable rate> -> **xsig**

<name> as defined in subclause 5.8.5.3.2.

The formal parameter list defines the calling interface to the opcode. Each formal parameter in the list has a name, a rate type, and may have an array width. If the array width is the special token **inchannels**, then the array width shall be the same as the number of input channels to the associated instrument instantiation (in the sense of subclause 5.15.2); if the array width is the special token **outchannels**, then the array width shall be the same as the number of orchestra output channels as defined in the global block.

There is no way to create user-defined opcodes with variable number of arguments in SAOL, although certain of the core opcodes have this property.

Within the opcode statement block, formal parameters may be used like any other variable. The rate tag of each formal parameter defines the rate of the variable. If an opcode is declared to be at a particular rate, then no formal parameter shall be declared faster than that rate.

There is a special rate tag **xsig** that allows formal parameters to be rate-polymorphic, see subclause 5.8.7.7.2. **xsig** shall not be the rate tag of any formal parameter unless the opcode is of type **opcode**.

5.8.7.5 Opcode variable declarations

5.8.7.5.1 Syntactic form

<opcode var declarations> -> <opcode var declaration> [<opcode var declarations>]
<opcode var declarations> -> **<NULL>**

<opcode var declaration> -> <instr variable declaration>
<opcode var declaration> -> **xsig** <namelist> **;**

<instr variable declaration> as defined in subclause 5.8.6.5.1.
<namelist> as defined in subclause 5.8.5.3.2.

The syntax and semantics of opcode variable declarations are the same as those of instrument variable declarations as given in subclause 5.8.6.5, with the following exceptions and additions:

The opcode variable names are available only within the scope of the opcode containing them. The instrument variable declarations for the instrument instantiation associated with the opcode call are not within the scope of an opcode, and references to these names shall not be made unless the names are also explicitly declared within the opcode, in which case the variable denoted is a different one. However, standard names (subclause 5.8.6.8) are within the scope of every opcode, may be referenced within opcodes, and shall have the semantics given in subclause 5.8.6.8 as applied to the instrument instantiation associated with the opcode call.

The values of opcode variables are static, and are preserved from call to call referencing a particular opcode state. The values of opcode variables shall be set to 0 in an opcode state before the first call referencing that state is executed.

The **tablemap** declaration may reference any tables declared in the local scope, as well as any formal parameters that are tables.

The values of opcode variables in different states of the same opcode (due to different syntactic uses of opcode expressions, or different indexing expressions in oparray expressions) are separate and have no relationship to one another.

There is a special rate tag called **xsig** that may be used to declare opcode variables in rate-polymorphic opcodes, see subclause 5.8.7.7.2. **xsig** shall not be the rate tag of any variable unless the opcode type is **opcode**.

5.8.7.6 Opcode statement block

5.8.7.6.1 Syntactic form

<opcode statement block> -> <opcode statement> [<opcode statement block>]
<opcode statement block> -> **<NULL**>

<opcode statement> -> <statement>
<opcode statement> -> **return (** <expr list> **) ;**

<statement> as defined in subclause 5.8.6.6.1.
<expr list> as defined in subclause 5.8.6.6.

The syntax and semantics of statements in opcodes are the same as the syntax and semantics of statements in instruments, with the following exceptions and additions:

No statement in an opcode shall be faster than the rate of the opcode, as defined in subclause 5.8.7.7.

The assignment statement and the values of all variables refer to the opcode state associated with this particular call to this opcode, or associated with a particular indexing expression in an oparray call.

There is a special statement called **return** that is used in opcodes. This statement allows opcodes to return values back to their callers.

5.8.7.6.2 Return statement

The **return** statement allows opcodes to return values back to their callers.

The expression parameter list may contain both single-valued and array-valued expressions.

The rate of the **return** statement is the rate of the opcode containing it. No expression in the expression parameter list shall be faster than the rate of the opcode.

The **return** statement shall be evaluated as follows. Each expression in the expression parameter list is evaluated, in the order they occur in the list. The return value of the opcode is the array-value formed by sequencing the values of the expression parameters. In the case that there is only one expression parameter that is a single-valued expression, then the return value of the opcode is the single value of that expression. The return value denoted by every **return** statement within an opcode shall have the same width (although it is permissible for them to differ in the number of expressions, so long as the sum of the widths of the expressions is equal).

After a **return** statement is encountered, no further statements in the opcode are evaluated, and control returns immediately to the calling instrument or opcode.

5.8.7.7 Opcode rate

5.8.7.7.1 Introduction

This subclause describes the rules for determining the rate of a call to an opcode, and the semantics of the special tags **opcode** and **xsig.**

The rate of an opcode call depends on the type of the opcode, as follows:

1. If the opcode type is **aopcode**, calls to the opcode are a-rate.
2. If the opcode type is **kopcode**, calls to the opcode are k-rate.
3. If the opcode type is **iopcode**, calls to the opcode are i-rate.
4. If the opcode type is **opcode**, the opcode is rate-polymorphic, and the rate is as described in the next subclause.

5.8.7.7.2 Rate-polymorphic opcodes

Opcodes that are rate-polymorphic take their rates from the context in which they are called. This allows the same opcode statement block to apply to multiple calling rate contexts. Without such a construct, three versions of each opcode of this sort would have to be created and used, depending on the context.

The rate of an **opcode** opcode for a particular call is the rate of the fastest actual parameter expression (not formal parameter expression) in that call, or the rate of the fastest formal parameter in the opcode definition, or the rate of the fastest guarding **if**, **while**, or **else** expression surrounding the opcode call, or the rate of the opcode enclosing the opcode call, whichever is fastest. If an **opcode** opcode has no non-**table** parameters, and is not enclosed in a guarded block or a opcode call, then it is a **kopcode** by default.

Rate-polymorphic opcodes may contain variable declarations and formal parameter declarations using the special rate tag **xsig**. A formal parameter of type **xsig**, for a particular call to that opcode, has the same rate as the actual parameter expression in the calling expression to which it corresponds. A variable of type **xsig**, for a particular call to that opcode, has the same rate as the opcode.

EXAMPLES

Given the following opcode definition:

```
opcode xop(ksig p1, xsig p2) {
  xsig v1;
  . . .
}
```

1. For the following code fragment

```
ksig k;
k = xop(1,2);
```

the rate of the opcode call is k-rate, since the formal parameter **p1** is faster than either of the actual parameters. The rate of **p2** within the call to **xop()** is i-rate, matching the actual parameter. The rate of **v1** within **xop()** is k-rate.

2. For the following code fragment

```
asig a1, a2;

a1 = xop(1,a2);
```

the rate of the opcode call is a-rate, since the actual parameter **a2** is faster than either of the formal parameters. The rates of **p2** and **v1** within the call to **xop()** are k-rate and a-rate respectively.

3. For the following code fragment

```
ksig k;
asig a;

k = xop(1,a)
```

there is a rate mismatch error, since the opcode call is a-rate, and thus shall not be assigned to a k-rate lvalue.

4. For the following code fragment

```
ksig k;
asig a1,a2;
oparray xop[10];

a1 = 0; while (a1 < 10) {
  a2 = a2 + xop[a1](1,k);
  a1 = a1 + 1;
}
```

the rate of the oparray call is a-rate, since the rate of the guarding expression is faster than any of the formal parameters or actual parameters. The rates of **p2** and **v1** within **xop()** are a-rate as well.

5.8.8 Template declaration

5.8.8.1 Syntactic form

<template declaration> -> **template <** <identlist> **>** [**preset** <maplist>] **(** <identlist> **)**
map { <identlist> **} with {** <maplist> **}**
{ <instr variable declarations> <block> **}**

<maplist> -> **<** <expr list> **> ,** <maplist>
<maplist> -> **<** <expr list> **>**

<identlist> as given in subclause 5.8.5.4.
<namelist> as given in subclause 5.8.5.3.2.
<instr variable declarations> as given in subclause 5.8.6.5.1.
<expr list> as given in subclause 5.8.6.6.1.
<block> as given in subclause 5.8.6.6.1.

A template declaration allows the concise declaration of multiple instruments that are similar in processing structure and syntax, but differ in only a few key expressions or wavetable names.

5.8.8.2 Semantics

The first identifier list contains the names for the instruments declared with the template. There shall be at least one identifier in this list. The first optional maplist contains a list of the preset number lists to be associated with each instrument in the template. This maplist may be omitted, in which case there are no preset numbers associated with the template instruments. If the maplist is present, it shall contain as many lists as there are instrument names in the first identified list. The second identifier list contains the pfields for the template declaration. Each instrument declared with the template has the same list of pfields. The third identifier list contains a list of template variables that are to be replaced in the subsequent code block with expressions from the second (first required) map list. There may be no identifiers in this list, in which case each instrument declared by the template is exactly the same.

The map list takes the form of a list of lists. This list shall have as many elements as template variables declared in the third identifier list. Each sublist is a list of expressions, and shall have as many elements as instrument names in the first identifier list. The first (optional) maplist shall not contain identifiers, only numeric values.

5.8.8.3 Template instrument definitions

As many instruments are defined by the template definition as there are names in the first identifier list. To describe each of the instruments, the identifiers described in the third (template variable) list are replaced in turn by each of the expressions from the map list.

That is, to construct the code for the first instrument, the code block given is processed by replacing the first template variable with the first expression from the first map list sublist, the second template variable with the first expression from the second map list sublist, the third template variable with the first expression from the third map list sublist, and so on. To construct the code for the second instrument, the code block given is processed by replacing the first template variable with the *second* expression from the first map list sublist, the second template variable with the *second* expression from the second map list sublist, the third template variable with the second expression from the third map list sublist, and so on.

This code-block processing occurs before any other syntax-checking or rate-checking of the elements of the instruments so defined. That is, the template variables are not true signal variables, and do not need to be declared in the variable declaration block. Once the code-block processing and template expansion is complete, the resulting instruments are treated as any other instruments in the orchestra.

EXAMPLE

The following template declaration:

```
template <oneharm, threeharm> preset <3,4>,<24> (p)
  map {pitch,t,bar} with
  { <440, harm1, mysig>, <p, harm2, mysig * mysig + 2> } {

  table harm1(harm,4096,1);
  table harm3(harm,4096,3,2,1);
  asig mysig;

  mysig = oscil(t,pitch,-1);

  mysig = bar *3;
  output(mysig);
}
```

declares exactly the same two instruments as the following two instrument declarations:

```
instr oneharm(p) preset 3 4 {
  table harm1(harm,4096,1);
  table harm3(harm,4096,3,2,1);
  asig mysig;

  mysig = oscil(harm1,440,-1);

  mysig = mysig * 3;
  output(mysig);
  }

instr threeharm(p) preset 24 {
  table harm1(harm,4096,1);
  table harm3(harm,4096,3,2,1);
  asig mysig;

  mysig = oscil(harm3,p,-1);

  mysig = (mysig * mysig + 2) * 3;  // notice embedding of template expression
  output(mysig);
  }
```

5.8.9 Reserved words

The following words are reserved, and shall not be used as identifiers in a SAOL orchestra or score.

aopcode asig else exports extend global if imports inchannels instr interp iopcode ivar kopcode krate ksig map oparray opcode outbus outchannels output preset return route sasbf send sequence spatialize srate table tablemap template turnoff while with xsig

Also, variable names starting with **_sym_** are reserved for implementation-specific use (for example, bitstream detokenisation) and shall not be the name of any instrument, signal variable, wavetable, or user-defined opcode in the orchestra.

5.9 SAOL core opcode definitions and semantics

5.9.1 Introduction

This clause describes the definitions and normative semantics for each of the core opcodes in SAOL. All core opcodes shall be implemented in every terminal that can decode Object Type 3 or 4.

For each core opcode, the following is described:

- The prototype, showing the rate of the opcode, the parameters which are required in a call to this opcode, and the rates of these parameters.
- The normative semantics of the return value. These semantics describe how to calculate the return value for each call to that opcode.
- The normative semantics of any side effects of the core opcode.

5.9.2 Specialop type

There is a special rate type for certain core opcodes called **specialop**. This rate type tag is not an actual lexical element of the SAOL language, and shall not appear in a SAOL orchestra, but is used in subsequent subclauses as a shorthand for core opcodes with these particular semantics.

Core opcodes with rate type **specialop** describe functions that map from one or more a-rate signals into a k-rate signal. That is, they have one or more parameters that vary at the a-rate, and they have normative semantics described at the a-rate, but they only return values and/or have side effects at the k-rate. When using these opcodes in expressions, they are treated as **kopcode** opcodes for the purposes of determining rate-mismatch errors (except that it is not a rate-mismatch error to pass them an a-rate signal), and as both **kopcode** and **aopcode** opcodes for the purposes of determining when to execute them.

An expression that is prohibited by the rules in subclauses 5.8.6.6 and 5.8.6.7 from being k-rate or a-rate shall not be of type **specialop**. Otherwise, an expression with one or more **specialop** components is also of type **specialop**. A **specialop** expression shall only occur in a guarded context (the code block of an **if**, or **else** statement) if the guard expression is also a **specialop** expression. A **specialop** expression shall not occur in the guarding expression or code block of the **while** statement.

The core opcodes with this type are: **fft**, **rms, sblock, downsamp**, and **decimate**.

EXAMPLE

Given the following code block:

```
asig x;
ksig y;

x = ...;
y = rms(x);          // #1
y = rms(x * 12);     // #2

if (rms(x)) {        // #3
  y = kline(...);    // #4
}

if (x) {              // #5
  y = rms(x);         // #6
}
```

```
if (rms(x) > y) {    // #7
  y = rms(x);        // #8
}
```

the following things are true. Statement #1 is a **ksig** rate statement; however, the right-side expression is a **specialop** expression and is executed at both the k-rate and at the a-rate (the assignment only occurs at the k-rate). Statement #2 is just like statement #1; the **x * 12** expression is a **specialop** expression. Statement #3 is legal, and is a **ksig** rate statement. The guard expression is a **specialop** expression and is evaluated at both the a-rate and the k-rate, and on each k-cycle on which it is nonzero, statement #4 is executed at the k-rate. Statement #5 contains a rate-mismatch error. The guard rate of the **if** statement is an a-rate expression, but statement #6 is only at the k-rate for the purposes of determining rate-mismatch errors. Statement #7 is legal. The guard expression is a **specialop** expression.

5.9.3 List of core opcodes

The several core opcodes are described in the subsequent subclauses. They are divided by category into major subclauses, but there is no normative significance in this division; it is only for clarity of presentation.

Math functionsint, frac, dbamp, ampdb, abs, sgn, exp, log, sqrt, sin, cos, atan, pow, log10, asin, acos, floor, ceil, min, max

Pitch converters gettune, settune, octpch, pchoct, cpspch, pchcps, cpsoct, octcps, midipch, pchmidi, midioct, octmidi, midicps, cpsmidi

Table operations ftlen, ftloop, ftloopend, ftsr, ftbasecps, ftsetloop, ftsetend, ftsetbase, ftsetsr, tableread, tablewrite, oscil, loscil, doscil, koscil

Signal generators kline, aline, kexpon, aexpon, kphasor, aphasor, pluck, buzz, grain

Noise generators irand, krand, arand, ilinrand, klinrand, alinrand, iexprand, kexprand, aexprand, kpoissonrand, apoissonrand, igaussrand, kgaussrand, agaussrand

Filters port, hipass, lopass, bandpass, bandstop, biquad, allpass, comb, fir, iir, firt, iirt

Spectral analysis fft, ifft

Gain control rms, gain, balance, compressor

Sample conversion decimate, upsamp, downsamp, samphold, sblock

Delays delay, delay1, fracdelay

Effects reverb, chorus, flange, speedt, fx_speedc

Tempo changes gettempo, settempo

For each core opcode, an opcode prototype is given. This shows the rate of the opcode, the number of required and optional formal parameters and the rate of each of the formal parameters. Certain parameters to certain core opcodes are presented in brackets, in which case that formal parameter is optional. Certain opcodes use the "…" notation, which means that the opcode can process an arbitrary number of parameters. The "…" is tagged with a rate for such opcodes, which is then the rate type of all of the parameters matching the varargs parameter. If there is not normative language for a particular opcode that specifies otherwise, it is a syntax error if any of the following statements apply:

- there are fewer actual parameters in the opcode call than required formal parameters
- there are more actual parameters in the opcode call than required and optional formal parameters, and the opcode definition does not include a varargs "..." section
- a particular actual parameter expression is of faster rate than the corresponding formal parameter, or than the varargs formal parameter if that is the correspondence

- a particular actual parameter expression is not single-valued, or is not table-valued when the corresponding formal parameter specifies a table.

The names associated with the formal parameters in the core opcode prototypes have no normative significance, but are used for clarity of exposition to refer to the values passed as the corresponding actual parameters when describing how to calculate the return value of the core opcode.

5.9.4 Math functions

5.9.4.1 Introduction

Each of the opcodes in this subclause computes a mathematical function. Whenever the result of calculating the function on the argument or arguments provided results in a NaN or Inf value, a run-time error shall result..

5.9.4.2 int

```
opcode int(xsig x)
```

The **int** core opcode calculates the integer part of its parameter.

The return value shall be the integer part of **x**.

5.9.4.3 frac

```
opcode frac(xsig x)
```

The **frac** core opcode calculates the fractional part of its parameter.

The return value shall be the fractional part of **x**, i.e., **x** – int(**x**). If **x** is negative, then **frac(x)** is also negative.

5.9.4.4 dbamp

```
opcode dbamp(xsig x)
```

The **dbamp** core opcode calculates the decibel equivalent of an amplitude parameter, where the maximum amplitude of 1 corresponds to a decibel level of 90 dB. It is a run-time error if **x** is not strictly positive.

The return value shall be $90 + 20 \log_{10} \mathbf{x}$.

5.9.4.5 ampdb

```
opcode ampdb(xsig x)
```

The **ampdb** core opcode calculates the amplitude equivalent of a decibel-valued parameter, where the maximum amplitude of 1 corresponds to a decibel level of 90 dB.

The return value shall be $10^{(\mathbf{x} - 90) / 20}$.

5.9.4.6 abs

```
opcode abs(xsig x)
```

The **abs** core opcode calculates the absolute value of a parameter.

The return value shall be **–x** if **x** < 0, or **x** otherwise.

5.9.4.7 sgn

```
opcode sgn(xsig x)
```

The **sgn** core opcode calculates the signum (sign function) of a parameter.

The return value shall be –1 if **x** < 0, 0 if **x** = 0, or 1 if **x** > 0.

5.9.4.8 exp

```
opcode exp(xsig x)
```

The **exp** core opcode calculates the exponential function.

The return value shall be $e^{\mathbf{x}}$.

5.9.4.9 log

```
opcode log(xsig x)
```

The **log** core opcode calculates the natural logarithm of a parameter.

It is a run-time error if **x** is not strictly positive.

The return value shall be log **x**.

5.9.4.10 sqrt

```
opcode sqrt(xsig x)
```

The **sqrt** core opcode calculates the square root of a parameter.

It is a run-time error if **x** is negative.

The return value shall be sqrt(**x**).

5.9.4.11 sin

```
opcode sin(xsig x)
```

The **sin** core opcode calculates the sine of a parameter given in radians.

The return value shall be sin **x**.

5.9.4.12 cos

```
opcode cos(xsig x)
```

The **cos** core opcode calculates the cosine of a parameter given in radians.

The return value shall be cos **x**.

5.9.4.13 atan

```
opcode atan(xsig x)
```

The **atan** core opcode calculates the arctangent of a parameter , in radians.

The return value shall be $\tan^{-1}$ **x**, in the range $[-\pi/2, \pi/2)$.

5.9.4.14 pow

```
opcode pow(xsig x, xsig y)
```

The **pow** core opcode calculates the to-the-power-of operation.

It shall be a run-time error if **x** is negative and **y** is not an integer.

The return value shall be $\mathbf{x}^{\mathbf{y}}$**.**

5.9.4.15 log10

```
opcode log10(xsig x)
```

The **log10** core opcode calculates the base-10 logarithm of a parameter.

It is a run-time error if **x** is not strictly positive.

The return value shall be $\log_{10}$ **x**.

5.9.4.16 asin

```
opcode asin(xsig x)
```

The **asin** core opcode calculates the arcsine of a parameter, in radians.

It is a run-time error if **x** is not in the range [-1, 1].

The return value shall be $\sin^{-1}$ **x**, in the range $[-\pi/2, \pi/2)$.

5.9.4.17 acos

```
opcode acos(xsig x)
```

The **acos** core opcode calculates the arccosine of a parameter, in radians.

It is a run-time error if **x** is not in the range [-1, 1].

The return value shall be $\cos^{-1}$ **x**, in the range $[0, \pi)$.

5.9.4.18 ceil

```
opcode ceil(xsig x)
```

The **ceil** core opcode calculates the ceiling of a parameter.

The return value shall be the smallest integer *y* such that **x** $\leq$ *y*.

5.9.4.19 floor

```
opcode floor(xsig x)
```

The **floor** core opcode calculates the floor of a parameter.

The return value shall be the greatest integer *y* such that *y* $\leq$ **x**.

5.9.4.20 min

```
opcode min(xsig x1[, xsig ...])
```

The **min** core opcode finds the minimum of a number of parameters.

The return value shall be the minimum value out of the parameter values.

5.9.4.21 max

```
opcode max(xsig x1[, xsig ...])
```

The **max** core opcode finds the maximum out of the parameter values.

The return value shall be the maximum value out of the parameter values.

5.9.5 Pitch converters

5.9.5.1 Introduction to pitch representations

There are four representations for pitch in a SAOL orchestra; the following twelve functions (after **gettune** and **settune**) convert them from one to another. The four representations are as follows:

- pitch-class, or **pch** representation. A pitch is represented as an integer part, which represents the octave number, where 8 shall be the octave containing middle C (C4); plus a fractional part, which represents the pitch-class, where .00 shall be C, .01 shall be C#, .02 shall be D, and so forth. Fractional parts larger than .11 (B) have no meaning in this representation; fractional parts between the pitch-class steps are rounded

to the nearest pitch-class.

For example, 7.09 is the A below middle C.

- octave-fraction, or **oct** representation. A pitch is represented as an integer part, which represents the octave number, where 8 shall be the octave containing middle C (C4); plus a fractional part, which represents a fraction of an octave, where each step of 1/12 represents a semitone.

  For example, 7.75 is the A below middle C, in equal-tempered tuning.

- MIDI pitch number representation. A pitch is represented as an integer number of semitones above or below middle C, represented as 60.

  For example, 57 is the A below middle C.

- Frequency, or **cps** representation. A pitch is represented as some number of cycles per second.

  For example, 220 Hz is the A below middle C.

Each of the pitch converters represents the conversion that is done by its name, with the new representation first and the original (parameter) representation second. Thus, **cpsmidi** is the converter that returns the frequency corresponding to a particular MIDI pitch.

Changes to the **pitch** field of an AudioBIFS **AudioSource** node (see clause 5.15) controlling the decoding process scale the global tuning around 440 Hz just as if **settune()** was called. The value of the **pitch** field is a multiplier to be applied to 440; that is if the **pitch** field is changed to 1.1, the global tuning becomes 484 Hz. Through this mechanism, changes in the **pitch** field apply to the results of all pitch converters and the **gettune()** opcode, but to no other instructions in the orchestra.

5.9.5.2 gettune

```
opcode gettune([xsig dummy])
```

The **gettune** core opcode returns the value in Hz of the current orchestra global tuning, which is the frequency of A above middle C. The global tuning shall be set by default to 440, but can be changed using the **settune** core opcode, subclause 5.9.5.3.

The **dummy** parameter is used to specify the rate of the opcode call if desired; see subclause 5.8.7.7.2.

5.9.5.3 settune

```
kopcode settune(ksig x)
```

The **settune** core opcode sets and returns the value of the current orchestra global tuning. The global tuning is used by several pitch converters when converting between symbolic pitch representations and cycles-per-second representation.

It is a run-time error if **x** is not strictly positive. (Allowing a wide range for tuning parameters allows unusual “pitch” representations to be used).

This core opcode has side-effects, as follows: The global tuning variable shall be set to the value **x**.

The return value shall be **x**.

5.9.5.4 octpch

```
opcode octpch(xsig x)
```

The **octpch** core opcode converts pitch-class representation to octave representation, with regard to equal scale tempering.

It is a run-time error if **x** is not strictly positive.

Let the integer part of **x** be *y* and the fractional part of **x** be *z*. Then, the return value shall be calculated as follows:

z shall be "rounded" to the nearest value such that 100*z* is an integer. If $z < 0$ or $z > 0.11$, then *z* shall be set to 0 instead.

Then, the return value shall be $y + 100z / 12$.

5.9.5.5 pchoct

```
opcode pchoct(xsig x)
```

The **pchoct** core opcode converts octave representation to pitch-class representation.

It is a run-time error if **x** is not strictly positive.

Let the integer part of **x** be *y* and the fractional part of **x** be *z.* Then, the return value shall be calculated as follows:

z shall be rounded to the nearest value such that 12 *z* is an integer. Then, the return value shall be $y + 12z / 100$.

5.9.5.6 cpspch

```
opcode cpspch(xsig x)
```

The **cpspch** core opcode converts pitch-class representation to cycles-per-second representation, with regard to equal scale tempering and the global tuning.

It is a run-time error if **x** is not strictly positive.

Let the integer part of **x** be *y* and the fractional part of **x** be *z.* Then, the return value shall be calculated as follows:

z shall be "rounded" to the nearest value such that 100*z* is an integer. If $z < 0$ or $z > 11$, then *z* shall be set to 0 instead.

Further let *t* be the global tuning parameter. Then, the return value shall be $t \times 2^{(y + 100z/12 - 8.75)}$.

5.9.5.7 pchcps

```
opcode pchcps(xsig x)
```

The **pchcps** core opcode converts cycles-per-second representation to pitch-class representation, with regard to the global tuning.

It is a run-time error if **x** is not strictly positive.

The return value shall be calculated as follows.

Let *t* be the global tuning parameter. Then, let *k* be $\log_2 (\mathbf{x} / t) + 8.75$. Then, let the integer part of *k* be *y* and the fractional part of *k* be *z*, "rounded" to the nearest value such that 12*z* is an integer. The return value shall be $y + 12z / 100$.

5.9.5.8 cpsoct

```
opcode cpsoct(xsig x)
```

The **cpsoct** core opcode converts octave representation to cycles-per-second representation, with regard to the global tuning.

It is a run-time error if **x** is not strictly positive.

Let *t* be the global tuning value; then, the return value shall be $t \times 2^{(\mathbf{x} - 8.75)}$.

5.9.5.9 octcps

```
opcode octcps(xsig x)
```

The **octcps** core opcode converts cycles-per-second representation to octave representation, with regard to the global tuning.

It is a run-time error if **x** is not strictly positive.

Let *t* be the global tuning value; then, the return value shall be $\log_2$ (**x** / *t*) + 8.75.

5.9.5.10 midipch

```
opcode midipch(xsig x)
```

The **midipch** core opcode converts pitch-class representation to MIDI representation.

It is a run-time error if **x** <= 3.

Let the integer part of **x** be *y* and the fractional part of **x** be *z*. Then, the return value shall be calculated as follows:

z shall be "rounded" to the nearest value such that 100*z* is an integer. If *z* < 0 or *z* > 0.11, then *z* shall be set to 0 instead.

The return value shall be 100*z* + 12 (*y* – 3).

5.9.5.11 pchmidi

```
opcode pchmidi(xsig x)
```

The **midipch** core opcode converts MIDI representation to pitch-class representation.

It is a run-time error if **x** is not strictly positive.

The return value shall be calculated as follows: **x** shall be rounded to the nearest integer, then let *k* be (**x** + 36) / 12, and let *y* be the integer part of *k*, and let *z* be the fractional part of *k*. Then, the return value shall be *y* + 12*z* / 100.

5.9.5.12 midioct

```
opcode midioct(xsig x)
```

The **midioct** core opcode converts octave representation to MIDI representation.

It is a run-time error if **x** <= 3.

The return value shall be calculated as follows. Let *k* be 12 (**x** – 3). Then, the value of *k* rounded to the nearest integer shall be the return value.

5.9.5.13 octmidi

```
opcode octmidi(xsig x)
```

The **octmidi** core opcode converts MIDI representation to octave representation.

It is a run-time error if **x** is not strictly positive.

The return value shall be (**x** +36) / 12.

5.9.5.14 midicps

```
opcode midicps(xsig x)
```

The **midicps** core opcode converts cycles-per-second representation to MIDI representation, with regard to the global tuning.

It is a run-time error if **x** is not strictly positive.

Let *t* be the global tuning parameter, and let *k* be 12 $\log_2$ (**x** / *t*) + 69. Then, the return value shall be *k* rounded to the nearest nonnegative integer.

5.9.5.15 cpsmidi

```
opcode cpsmidi(xsig x)
```

The **cpsmidi** core opcode converts MIDI representation to cycles-per-second representation, with regard to the global tuning and equal scale temperament.

It is a run-time error if **x** is not strictly positive.

Let *t* be the global tuning parameter. Then, the return value shall be $t \times 2^{(\mathbf{x} - \mathbf{69})/12}$.

5.9.6 Table operations

5.9.6.1 ftlen

```
opcode ftlen(table t)
```

The **ftlen** core opcode returns the length of a table. The length of a table is the value calculated based on the **size** parameter in the particular core wavetable generator as described in clause 5.10.

The return value shall be the length of the table referenced by **t**.

5.9.6.2 ftloop

```
opcode ftloop(table t)
```

The **ftloop** core opcode returns the loop start point of a wavetable. The loop point is set either in a sound sample data block in the bitstream, or by the **ftsetloop** core opcode (see subclause 5.9.6.6), or else it is 0.

The return value shall be the loop start point (in samples) of the wavetable referenced by **t**.

5.9.6.3 ftloopend

```
opcode ftloopend(table t)
```

The **ftloopend** core opcode returns the loop end point of a wavetable. The loop point is set either in a sound sample data block in the bitstream, or by the **ftsetend** core opcode (see subclause 5.9.6.7), or else it is 0.

The return value shall be the loop end point (in samples) of the wavetable referenced by **t**.

5.9.6.4 ftsr

```
opcode ftsr(table t)
```

The **ftsr** core opcode returns the sampling rate of a wavetable. The sampling rate is set in a sound sample data block in the bitstream, or else it is 0.

The return value shall be the sampling rate, in Hz, of the wavetable referenced by **t**.

5.9.6.5 ftbasecps

```
opcode ftbasecps(table t)
```

The **ftbasecps** core opcode returns the base frequency of a wavetable, in cycles per second (Hz). The base frequency is set either in a sound sample data block in the bitstream, or in the core wavetable generator **sample** (subclause 5.10.2), or by the core opcode **ftsetbase** (subclause 5.9.6.8), or else it is 0.

The return value shall be the base frequency, in Hz, of the wavetable referenced by **t**.

5.9.6.6 ftsetloop

```
kopcode ftsetloop(table t, ksig x)
```

The **ftbasecps** core opcode sets the loop start point of a wavetable to a new value, and returns the new value.

It is a run-time error if **x** < 0, or if **x** is larger than the size of the wavetable referenced by **t**.

This core opcode has side effects, as follows: the loop start point of the wavetable **t** shall be set to sample number **x**.

The return value shall be **x**.

5.9.6.7 ftsetend

```
kopcode ftsetend(table t, ksig x)
```

The **ftsetend** core opcode sets the loop end point of a wavetable to a new value, and returns the new value. It is a run-time error if **x** < 0, or if **x** is larger than the size of the wavetable referenced by **t**.

This core opcode has side effects, as follows: the loop end point of the wavetable **t** shall be set to sample number **x**.

The return value shall be **x**.

5.9.6.8 ftsetbase

```
kopcode ftsetbase(table t, ksig x)
```

The **ftsetbase** core opcode sets the base frequency of a wavetable to a new value, and returns the new value. It is a run-time error if **x** is not strictly positive.

This core opcode has side effects, as follows: the base frequency of the wavetable **t** shall be set to **x**, where **x** is a value in Hz.

The return value shall be **x**.

5.9.6.9 ftsetsr

```
kopcode ftsetsr(table t, ksig x)
```

The **ftsetsr** core opcode sets the sampling rate parameter of a wavetable to a new value **x**, and returns the new value. It is a run-time error if **x** is not strictly positive.

This core opcode has side effects, as follows: the sampling rate of the wavetable **t** shall be set to **x**, where **x** is a value in Hz.

The return value shall be **x**.

5.9.6.10 tableread

```
opcode tableread(table t, xsig index)
```

The **tableread** core opcode returns a single value from a wavetable. It is a run-time error if **x** < 0, or if **x** is larger than the size of the wavetable referenced by **t**.

The return value shall be the value of the wavetable **t** at sample number **index**, where sample number 0 is the first sample in the wavetable. If **index** is not an integer, then the return value shall be interpolated from nearby points of the wavetable, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

5.9.6.11 tablewrite

```
opcode tablewrite(table t, xsig index, xsig val)
```

The **tablewrite** core opcode sets a single value in a wavetable, and returns that value. It is a run-time error if **index** < 0, or if **index** is larger than the size of the wavetable referenced by **t**.

This core opcode has side effects, as follows: **index** shall be rounded to the nearest integer, and the value of sample number **index** in the wavetable **t** shall be set to the new value **val**, where sample number 0 is the first sample in the wavetable.

The return value shall be **val**.

If global tables are written to at the a-rate by one instrument instance, it is unspecified when the new values become available to other instrument instances. In particular, it is unspecified whether the changes are available in the same orchestra cycle. The changes shall be available to all instrument instances in the next orchestra cycle, and are permitted to be available to instances of instruments sequenced later in the same orchestra cycle.

5.9.6.12 oscil

```
aopcode oscil(table t, asig freq[, ivar loops])
```

The **oscil** core opcode loops several times around the wavetable **t** at a rate of **freq** loops per second, returning values at the audio-rate. **loops** shall be rounded to the nearest integer when the opcode is evaluated. If **loops** is not provided, its value shall be –1.

It is a run-time error if **loops** is not strictly positive and is also not –1.

The return value is calculated according to the following procedure.

On the first a-rate call to **oscil** relative to a particular state, the *internal phase* shall be set to 0, and the *internal number of loops* set to **loops**. On subsequent calls, the internal phase shall be incremented by **freq/SR,** where **SR** is the orchestra sampling rate. If, after the incrementation, the internal phase is not in the interval [0,1] and the internal loop count is strictly positive, the phase shall be set to the fractional portion of its value ($p := p - \text{floor}(p)$) and the loop count decremented.

If the internal loop count is zero, the return value shall be 0. Otherwise the return value shall be the value of sample number x in the wavetable, where $x = p * l$, where p is the current internal phase, and l is the length of table **t**. If x is not an integer, then the value shall be interpolated from the nearby table values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

NOTE - The **oscil** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **oscil** is referenced twice in the same a-cycle, then the effective loop frequency is twice as high as given by **freq**.

5.9.6.13 loscil

```
aopcode loscil(table t, asig freq[, ivar basefreq, ivar loopstart, ivar loopend])
```

This opcode loops around the wavetable **t**, returning values at the audio-rate. The looping continues as long as the opcode is active, and is performed at a special rate that depends on the base frequency **basefreq** and the sampling rate of the table. In this way, samples that were recorded at a particular known pitch may be interpolated to any other pitch.

If **basefreq** is not provided, it shall be set to the base frequency of the table **t** by default. If the table **t** has base frequency 0 and **basefreq** is not provided, it is a run-time error. If **basefreq** is not strictly positive, it is a runtime error. The **basefreq** parameter shall be specified in Hz.

If **loopstart** and **loopend** are not provided, they shall be set to the loop start point and loop end point of the table **t**, respectively. If **loopend** is not provided and the loop end point of **t** is 0, then it shall be set to the end of the table (l – 1), where l is the length of the table in sample points). If **loopstart** is not strictly less than **loopend**, or either is negative, it is a runtime error.

The return value is calculated according to the following procedure.

Let l be the length of the table, m be the value **loopstart** / l, and n be the value **loopend** / l. On the first a-rate call to **loscil** relative to a particular state, the *internal phase* shall be set to *0*. On subsequent calls, the internal phase shall be incremented by **freq * TSR / (basefreq *SR),** where **TSR** is the sampling rate of the table and **SR** is the orchestra sampling rate. If the incrementation has caused the internal phase to leave the interval [m,n] or to become less than 0, the phase shall be set to $m + \boldsymbol{p - kn}$, where p is the internal phase and k is the value floor(p/n).

The return value shall be the value of sample number x in the wavetable, where $x = \boldsymbol{p * l}$, where p is the current internal phase, and l is the length of table **t**. If x is not an integer, then the value shall be interpolated from the nearby table values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

NOTE - The **loscil** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **loscil** is referenced twice in the same a-cycle, then the effective loop frequency is twice as high as given by **freq**.

5.9.6.14 doscil

```
aopcode doscil(table t)
```

The **doscil** core opcode plays back a sample once, with no frequency control or looping. It is useful for sample-rate matching sampled drum sounds to an orchestra rate.

The return value is calculated according to the following procedure.

On the first a-rate call to **doscil** relative to a particular state, the *internal phase* shall be set to 0. On subsequent calls, the internal phase shall be incremented by **TSR/SR,** where **TSR** is the sampling rate of the table **t** and **SR** is the orchestra sampling rate. If, after the incrementation, the internal phase is greater than 1, then the opcode is done.

If the opcode is done, the return value shall be 0. Otherwise the return value shall be the value of sample number *x* in the wavetable, where $x = p * l$, where *p* is the current internal phase, and *l* is the length of table **t**. If *x* is not an integer, then the value shall be interpolated from the nearby table values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

NOTE - The **doscil** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **doscil** is referenced twice in the same a-cycle, then the sample is played back at twice its original frequency.

5.9.6.15 koscil

```
kopcode koscil(table t, ksig freq[, ivar loops])
```

This opcode loops several times around the wavetable **t** at a rate of **freq** loops per second, returning values at the control-rate. **loops** shall be rounded to the nearest integer when the opcode is evaluated. If **loops** is not provided, its value shall be set to –1.

It is a run-time error if **loops** is not strictly positive and is also not –1.

The return value is calculated according to the following procedure.

On the first k-rate call to **koscil** relative to a particular state, the *internal phase* shall be set to 0, and the *internal number of loops* set to **loops**. On subsequent calls, the internal phase shall be incremented by **freq/KR**, where **KR** is the orchestra control rate. If, after the incrementation, the phase is not in the interval [0,1]and the internal loop count is strictly positive, the phase shall be set to the fractional portion of its value ($p := p - \text{floor}(p)$) and the loop count decremented.

If the internal loop count is zero, the return value shall be 0. Otherwise the return value shall be the value of sample number *x* in the wavetable, where $x = p * l$, where *p* is the current internal phase, and *l* is the length of table **t**. If *x* is not an integer, then the value shall be interpolated from the nearby table values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

NOTE - The **koscil** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **koscil** is referenced twice in the same k-cycle, then the effective loop frequency is twice as high as given by **freq**.

5.9.7 Signal generators

5.9.7.1 kline

```
kopcode kline(ivar x1, ivar dur1, ivar x2[, ivar dur2, ivar x3, …]);
```

The **kline** core opcode produces a line-segmented or "ramp" function, with values changing at the k-rate. This function takes **dur1** seconds to go from **x1** to **x2**, **dur2** seconds to go from **x2** to **x3**, and so on.

It is a run-time error if any of the following conditions apply:

- there are an even number of parameters
- any of the **dur** values are negative

The return value shall be calculated as follows:

On the first call to **kline** with regard to a particular state, the *internal time* shall be set to 0, the *current left point* to **x1**, the *current right point* to **x2**, and the *current duration* to **dur1**. On subsequent calls, the internal time shall be incremented by 1/**KR**, where **KR** is the orchestra control rate. So long as the internal time is thereby greater than the current duration and there is another duration parameter, the internal time shall be decremented by the current duration, the current duration shall be set to the next duration parameter, the current left point to the current right point, and the current right point to the next control point (x-value) (these steps repeat if necessary so long as the internal time is greater than the current duration). If there is no additional duration parameter, the generator is done.

If the generator is done, then the return value is 0. Otherwise, the return value is *l* + (*r* – *l*)*t*/*d*, where *l* is the current left point, *r* is the current right point, *t* is the internal time, and *d* is the current duration.

NOTE - The **kline** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **kline** is referenced twice in the same k-cycle, then the effective segment duration is half as long as given by the corresponding duration value.

5.9.7.2 aline

```
kopcode aline(ivar x1, ivar dur1, ivar x2[, ivar dur2, ivar x3, …]);
```

The **aline** core opcode produces a line-segmented or "ramp" function, with values changing at the a-rate. This function takes **dur1** seconds to go from **x1** to **x2**, **dur2** seconds to go from **x2** to **x3**, and so on.

It is a run-time error if any of the following conditions apply:

- there are an even number of parameters
- any of the **dur** values are negative

The return value shall be calculated as follows:

On the first call to **aline** with regard to a particular state, the *internal time* shall be set to 0, the *current left point* to **x1**, the *current right point* to **x2**, and the *current duration* to **dur1**. On subsequent calls, the internal time shall be incremented by 1/**SR**, where **SR** is the orchestra sampling rate. So long as the internal time is thereby greater than the current duration and there is another duration parameter, the internal time shall be decremented by the current duration, the current duration shall be set to the next duration parameter, the current left point to the current right point, and the current right point to the next control point (x-value) (these steps repeat if necessary so long as the internal time is greater than the current duration). If there is no additional duration parameter, the generator is done.

If the generator is done, then the return value is 0. Otherwise, the return value is *l* + (*r* – *l*) *x t*/*d*, where *l* is the current left point, *r* is the current right point, *t* is the internal time, and *d* is the current duration.

NOTE - The **aline** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **aline** is referenced twice in the same a-cycle, then the effective segment duration is half as long as given by the corresponding duration value.

5.9.7.3 kexpon

```
kopcode kexpon(ivar x1, ivar dur1, ivar x2[, ivar dur2, ivar x3, …]);
```

The **kexpon** core opcode produces a segmented function made out of exponential curves, with values changing at the k-rate. This function takes **dur1** seconds to go from **x1** to **x2**, **dur2** seconds to go from **x2** to **x3**, and so on.

It is a run-time error if any of the following conditions apply:

- there are an even number of parameters
- any of the **dur** values are negative
- the **x** values are not all the same sign
- any **x** value is 0

The return value shall be calculated as follows:

On the first call to **kexpon** with regard to a particular state, the *internal time* shall be set to 0, the *current left point* to **x1**, the *current right point* to **x2**, and the *current duration* to **dur1**. On subsequent calls, the internal time shall be incremented by 1/**KR**, where **KR** is the orchestra control rate. So long as the internal time is thereby greater than the current duration and there is another duration parameter, the internal time shall be decremented by the current duration, the current duration shall be set to the next duration parameter, the current left point to the current right point, and the current right point to the next control point (x-value) (these steps repeat if necessary so long as the internal time is greater than the current duration). If there is no additional duration parameter, the generator is done.

If the generator is done, then the return value is 0. Otherwise, the return value is $l\ (r / l)^{t/d}$, where *l* is the current left point, *r* is the current right point, *t* is the internal time, and *d* is the current duration.

NOTE - The **kexpon** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **kexpon** is referenced twice in the same k-cycle, then the effective segment duration is half as long as given by the corresponding duration value.

5.9.7.4 aexpon

```
aopcode aexpon(ivar x1, ivar dur1, ivar x2[, ivar dur2, ivar x3, …]);
```

The **aexpon** core opcode produces a segmented function made out of exponential curves, with values changing at the a-rate. This function takes **dur1** seconds to go from **x1** to **x2**, **dur2** seconds to go from **x2** to **x3**, and so on.

It is a run-time error if any of the following conditions apply:

- there are an even number of parameters
- any of the **dur** values are negative
- the **x** values are not all the same sign
- any **x** value is 0

The return value shall be calculated as follows:

On the first call to **aexpon** with regard to a particular state, the *internal time* shall be set to 0, the *current left point* to **x1**, the *current right point* to **x2**, and the *current duration* to **dur1**. On subsequent calls, the internal time shall be incremented by 1/**SR**, where **SR** is the orchestra sampling rate. . So long as the internal time is thereby greater than the current duration and there is another duration parameter, the internal time shall be decremented by the current duration, the current duration shall be set to the next duration parameter, the current left point to the current right point, and the current right point to the next control point (x-value) (these steps repeat if necessary so long as the internal time is greater than the current duration). If there is no additional duration parameter, the generator is done.

If the generator is done, then the return value is 0. Otherwise, the return value is $l\ (r / l)^{t/d}$, where l is the current left point, r is the current right point, t is the internal time, and d is the current duration.

NOTE - The **aexpon** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **aexpon** is referenced twice in the same a-cycle, then the effective segment duration is half as long as given by the corresponding duration value.

5.9.7.5 kphasor

```
kopcode kphasor(ksig cps)
```

The **kphasor** core opcode produces a moving phase value, looping from 0 to 1 repeatedly, **cps** times per second.

The return value shall be calculated as follows:

On the first call to **kphasor** with regard to a particular state, the *internal phase* shall be set to 0. On subsequent calls, the internal phase shall be incremented by **cps/KR**, where **R** is the orchestra control rate. If the internal phase is thereby not in the interval [0,1], the internal phase shall be set to the fractional part of its value (p = frac(p)). The return value is the internal phase.

NOTE - The **kphasor** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **kphasor** is referenced twice in the same k-cycle, then the effective frequency is twice as fast as given by **cps**.

5.9.7.6 aphasor

```
aopcode aphasor(asig cps)
```

The **aphasor** opcode produces a moving phase value, looping from 0 to 1 repeatedly, **cps** times per second.

The return value shall be calculated as follows:

On the first call to **aphasor** with regard to a particular state, the *internal phase* shall be set to 0. On subsequent calls, the internal phase shall be incremented by **cps/SR**, where **SR** is the orchestra sampling rate. If the internal phase is

thereby not in the interval [0,1], the internal phase shall be set to the fractional part of its value (p = frac(p)). The return value is the internal phase.

NOTE - The **aphasor** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **aphasor** is referenced twice in the same a-cycle, then the effective frequency is twice as fast as given by **cps**.

5.9.7.7 pluck

```
aopcode pluck(asig cps, ivar buflen, table init, ksig atten, ksig smoothrate)
```

This opcode uses a simple form of the Karplus-Strong algorithm to generate plucked-string sounds by repeated sampling and smoothing of a buffer.

It is a run-time error if **buflen** is not strictly positive.

The return value is calculated as follows:

On the first call to **pluck** with regard to a particular opcode state, a *buffer* of length **buflen** shall be created and filled with the values from the table **init**, as follows. Let x be the length of the table **init**. If x is less than **buflen**, then the values of the buffer shall be set to the first **buflen** sample values of the table **init**. If x is greater than or equal to **buflen**, then the first **buflen** values of the buffer shall be set to the sample values in the table **init**, and the remainder of the buffer filled as described in this paragraph for the whole table. That is, as many full and partial cycles of the table are used as necessary to fill the buffer.

Also on the first call to **pluck** with regard to a particular state, the *internal phase* shall be set to 0, and the *smooth count* shall be set to 0.

On subsequent calls to **pluck** with regard to a state, the smooth count is incremented. If the smooth count is equal to **smoothrate**, then **smoothrate** is set to 0, and the buffer shall be smoothed, as follows. A new buffer of length **buflen** shall be created, and its values set by averaging over the current buffer. Each sample value in the new buffer shall be set to the value of the attenuated mean of the five surrounding samples of the current buffer. That is, for each sample x of the new buffer, its value shall be set to **atten** * (b[x-2] + b[x-1] + b[x] + b[x+1] + b[x+2])/5, where the b[.] notation refers to values of the current buffer, and the indices are calculated modulo **buflen** (that is, they "wrap around"). Then, the values of the current buffer shall be set to the values of the new buffer.

Whether or not the buffer has just been smoothed, the internal phase shall be incremented by **cps/SR**, where **SR** is the orchestra sampling rate, and if the resulting value is not in the interval [0,1], then the internal phase shall be set to the fractional part of the internal phase ($p = p$ – floor(p)).

The return value shall be the value of the buffer at the point p * **buflen**, where p is the internal phase. If this index is not an integer, the value shall be interpolated from nearby buffer values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5).

NOTE - The **pluck** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **pluck** is referenced twice in the same a-cycle, then the effective frequency is twice as fast as given by **cps**.

5.9.7.8 buzz

```
aopcode buzz(asig cps, ksig nharm, ksig lowharm, ksig rolloff)
```

The **buzz** opcode produces a band-limited pulse train formed by adding together cosine overtones of a fundamental frequency **cps** given in Hz. These noisy sounds are useful as complex sound sources for subtractive synthesis.

lowharm gives the lowest harmonic used, where 0 is the fundamental, at frequency **cps**. It is a runtime error if **lowharm** is negative.

nharm gives the number of harmonics used starting from **lowharm**. If **nharm** is not strictly positive, then every overtone up to the orchestra Nyquist frequency is used (**nharm** shall be set to **SR** / 2 / **cps** – **lowharm**).

rolloff gives the multiplicative rolloff that defines the spectral shape. If **rolloff** is negative, then the partials alternate in phase; if |**rolloff**| > 1, then the partials increase in amplitude rather than attenuating.

The return value is calculated as follows. On the first call to **buzz** with regard to a particular scope, the *internal phase* shall be set to 0. On subsequent calls, the internal phase shall be incremented by **cps / SR**, where **SR** is the orchestra sampling rate. If, after this incrementation, the internal phase is greater than 1, the internal phase shall be set to the fractional part of its value (*p := frac(p)*).

The return value shall be

$$scale * \sum_{f=\mathbf{lowharm}}^{\mathbf{lowharm}+\mathbf{nharm}} \mathbf{rolloff}^{(f-\mathbf{lowharm})} \cos 2\pi fp$$

where *p* is the internal phase and ***scale*** is the value (1-abs(**rolloff**)) / (1-abs(**rolloff**$^{\mathbf{nharm}}$)).

5.9.7.9 grain

```
aopcode grain(table wave, table env, ksig density, ksig freq, ksig amp,
              ksig dur, ksig time, ksig phase)
```

The **grain** opcode uses granular synthesis [**GRAN**] to synthesize periodic, quasi-periodic, noisy, and textured sounds. A sound in granular synthesis is represented as the sum of a number of short sound samples or “grains” distributed throughout the time-frequency space.

wave is the waveshape for the grain. **env** is the envelope to apply to the grain. **density** is the time-spacing of grain trigger points in Hz. **freq** is the frequency in Hz at which to place each new grain. **amp** is the amplitude of each new grain, as a scaling factor. **dur** is the duration of each grain in seconds. **time** is the offset in seconds from the trigger at which grains start (jitter). **phase** is the starting phase of each grain, in the range [0,1].

It is a run-time error if any of the following conditions apply: **density** is not positive, **dur** is negative, **time** is negative, or **phase** is not in the range [0,1].

On the first call to **grain** with regard to a particular opcode state, the *number of grains* is set to 0 and the *density clock* and *trigger clock* are both set to 0.

On the first and each subsequent a-rate call to **grain**, the following steps are executed:

The density clock is incremented by 1/SR, where SR is the orchestra sampling rate. If the density clock is thereby greater than or equal to 1/**density**, then the density clock is set to zero, and then if **time** < 1/**density**, the trigger clock is set to **time**. If **time >=** 1/**density**, the trigger clock remains at zero and no grain is created.

If the trigger clock is positive, then it is decremented by 1/SR. If the trigger clock is thereby less than or equal to zero, a new grain is dispatched. To dispatch a new grain, the number of active grains is incremented and henceforth denoted as **i**. Space shall be allocated to hold the current phase **phase[i]**, frequency **freq[i]**, amplitude **amp[i]**, duration **dur[i]**, and time **gtime[i]** of the new grain. These values shall be set to the current value of **phase**, **freq**, **amp**, **dur**, and 0 respectively.

For each active grain **i**, the current value of the grain **x[i]** is calculated as follows. There are three conditions, depending on the format of the wavetable **wave**:

1. If **wave** has both its sampling rate and base frequency parameters set, it’s assumed to be a pitched sample, and is pitch-matched to **freq** in the manner of **loscil()**. Let **loopstart** be set to the loop start parameter of the table **wave**; let **loopend** be set to the loop end parameter of the table, or **l-**1 where **l** is the length of the table **wave,** if the loop end parameter is 0. Let ***m*** be the value **loopstart / l**, and let *n* be the value **loopend** / **l**. Each time after the first that the grain value is calculated, the phase **phase[i]** shall be incremented by **freq[i] * TSR / (basefreq * SR)**, where **TSR** is the table sampling rate and **basefreq** the base frequency of table **wave**. If, after this incrementation, the phase is not in the range [0,1], the phase shall be set to **phase[i] - floor(phase[i]).** The current value of the grain **x[i]** is the value of sample number **q** of the wavetable **wave**, where **q = phase[i]** * (**m-n**) + **m**. If the value of **q** is not an integer, then the value of **x[i]** shall be interpolated from the nearby table values, as described in subclause 5.8.5.2.5.

2. If **wave** has its sampling rate parameter set, but not its base frequency parameter, then it’s assumed to be an unpitched sample, and is sample-rate-matched to the orchestra in the manner of **doscil()** and **freq[i]** is ignored. Each time after the first that the grain value is calculated, the phase **phase[i]** shall be incremented by **TSR/SR**, where **TSR** is the table sampling rate and **SR** is the orchestra sampling rate. If, after this incrementation, the phase is greater than 1, then the current and all future values of this grain are 0. Otherwise, the current value of the grain **x[i]** is the value of sample number **q** of the wavetable **wave**, where **q = phase[i] * l** and **l** is the length of the wavetable **wave**. If the value

of **q** is not an integer, then the value of **x[i]** shall be interpolated from the nearby table values, as described in subclause 5.8.5.2.5.

3. If **wave** has neither its sampling rate nor base frequency parameters set, then it's assumed to be a waveshaped, and is oscillated in the manner of **oscil()**. Each time after the first that the grain value is **calculated**, the phase **phase[i]** shall be incremented by **freq[i]/SR**. If, after this incrementation, the phase value is outside the range [0,1], then the phase shall be set to the fractional part of its value **phase[i]-floor(phase[i])**. The current value of the grain **x[i]** shall be the value of sample number **q** of the wavetable **wave**, where **q** = **phase[i]** * **l** and **l** is the length of the wavetable **wave**. If the value of **q** is not an integer, then the value of **x[i]** shall be interpolated from the nearby table values, as described in subclause 5.8.5.2.5.

After the output value of the grain **x[i]** is calculated in one of these three manners, it is modulated by the envelope wavetable according to the grain duration. The time **gtime[i]** is incremented by **1/SR**, where **SR** is the orchestra sampling rate. If **gtime[i] > dur**, then the grain is over; it shall be noted as non-active, and its space may be deallocated. Otherwise, the modulator value **m[i]** is the value of sample number **q** of the wavetable **env**, where **q = floor(gtime[i] / dur[i] * l)** and **l** is the length of the wavetable **env**. The final output value of the grain **x[i]** is rescaled by this modulator value as in **x[i] := x[i] * m[i]**.

The output value from the opcode is the sum of the output values for all active grains.

NOTE (1) - If the input values **freq, dur,** and so forth change during the use of the instrument (as they would, for example, to stochastically vary grain distribution), the values of **freq[i], dur[i]** and so forth do not change for currently-active grains. Only the parameters of new grains are affected. The parameter values of active grains do not change during the lifetime of the grain.

NOTE (2) - The **grain** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **grain** is referenced twice in the same a-cycle, than the effective frequency, duration, density, jitter, and so forth are scaled appropriately.

5.9.8 Noise generators

5.9.8.1 Note on noise generators and pseudo-random sequences

The following core opcodes generate noise, that is, pseudo-random sequences of various statistical properties. In order to provide maximum decorrelation among multiple noise generators, it is important that all references to pseudo-random generation share a single feedback state. That is, all random values required by the various states of various noise generators shall make use of sequential values from a single "master" pseudo-random sequence.

It is strictly prohibited for an implementation to maintain multiple pseudo-random sequences to draw from (using the same algorithm) for various states of noise generation opcodes, because to do so may result in strong correlations between multiple noise generators.

This point does not apply to implementations that do not use "linear congruential", "modulo feedback", or similar mathematical structures to generate pseudo-random numbers.

The standard mathematical description of probability density functions is used in this subclause. This means that if the pdf of a random variable *x* is *f*(*x*), then the probability of it taking a value in the range [*y,z*] is $\int_y^z f(x)dx$.

5.9.8.2 irand

```
iopcode irand(ivar p)
```

The **irand** core opcode generates a random number from a linear distribution.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \begin{cases} 1/2\mathbf{p} : x \in [-\mathbf{p}, \mathbf{p}] \\ 0 : otherwise \end{cases}$$

5.9.8.3 krand

```
kopcode krand(ksig p)
```

The **krand** core opcode generates random numbers from a linear distribution.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \begin{cases} 1/2\mathbf{p} : x \in [-\mathbf{p}, \mathbf{p}] \\ 0 : otherwise \end{cases}$$

5.9.8.4 arand

```
aopcode arand(asig p)
```

The **arand** core opcode generates random noise according to a linear distribution.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \begin{cases} 1/2\mathbf{p} : x \in [-\mathbf{p}, \mathbf{p}] \\ 0 : otherwise \end{cases}$$

5.9.8.5 ilinrand

```
iopcode ilinrand(ivar p1, ivar p2)
```

The **ilinrand** core opcode generates a random number from a linearly-ramped distribution.

The return value shall be a random number *x* chosen according to the pdf

p(*x*) = abs(2 / (**p2** – **p1**) * [(*x* **– p1**) / (**p2** – **p1**)]) if *x* ∈ [**p1**,**p2**]
0 otherwise

5.9.8.6 klinrand

```
kopcode klinrand(ksig p1, ksig p2)
```

The **klinrand** core opcode generates random numbers from a linearly-ramped distribution.

The return value shall be a random number *x* chosen according to the pdf

p(*x*) = abs(2 / (**p2** – **p1**) * [(*x* **– p1**) / (**p2** – **p1**)]) if *x* ∈ [**p1**,**p2**]
0 otherwise

5.9.8.7 alinrand

```
aopcode alinrand(asig p1, asig p2)
```

The **alinrand** core opcode generates random noise from a linearly-ramped distribution.

The return value shall be a random number *x* chosen according to the pdf

p(*x*) = abs(2 / (**p2** – **p1**) * [(*x* **– p1**) / (**p2** – **p1**)]) if *x* ∈ [**p1**,**p2**]
0 otherwise

5.9.8.8 iexprand

```
iopcode iexprand(ivar p1)
```

The **iexprand** core opcode generates a random number from a exponential distribution with mean **p1**. It is a run-time error if **p1** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

$p(x) = \begin{cases} 0 & \text{if } x \le 0, \text{ or} \\ k \exp(-kx), \text{ where } k = 1 / \mathbf{p1}, & \text{otherwise.} \end{cases}$

5.9.8.9 kexprand

```
kopcode kexprand(ksig p1)
```

The **kexprand** core opcode generates random numbers from an exponential distribution with mean **p1**. It is a run-time error if **p1** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

$p(x) = \begin{cases} 0 & \text{if } x \le 0, \text{ or} \\ k \exp(-kx), \text{ where } k = 1 / \mathbf{p1}, & \text{otherwise.} \end{cases}$

5.9.8.10 aexprand

```
aopcode aexprand(asig p1)
```

The **aexprand** core opcode generates random noise according to an exponential distribution with mean **p1**. It is a run-time error if **p1** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

$p(x) = \begin{cases} 0 & \text{if } x \le 0, \text{ or} \\ k \exp(-kx), \text{ where } k = 1 / \mathbf{p1}, & \text{otherwise.} \end{cases}$

5.9.8.11 kpoissonrand

```
kopcode kpoissonrand(ksig p1)
```

The **kpoissonrand** core opcode generates a random binary (0/1) sequence of numbers such that the mean time between 1's is **p1** seconds. It is a run-time error if **p1** is not strictly positive.

On the first call to **kpoissonrand** with regard to a particular opcode state, a random number *x* shall be chosen according to the pdf

$p(x) = \begin{cases} 0 & \text{If } x \le 0, \text{ or} \\ k \exp(-kx), \text{ where } k = 1/ (\mathbf{p1} * \mathbf{KR}), & \text{otherwise.} \end{cases}$

where **KR** is the orchestra control rate.

The return value shall be 0 and the floor of this random value shall be stored.

On subsequent calls, the stored value shall be decremented by 1. If the decremented value is -1, the return value shall be 1 and a new random value shall be generated and stored as described above. Otherwise, the return value shall be 0.

NOTE - The **kpoissonrand** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **kpoissonrand** is referenced twice in the same k-cycle, then the effective mean time between 1 values is half as long as given by **t**.

5.9.8.12 apoissonrand

```
aopcode apoissonrand(asig p1)
```

The **apoissonrand** core opcode generates random binary (0/1) noise such that the mean time between 1's is **p1** seconds. It is a run-time error if **p1** is not strictly positive.

On the first call to **apoissonrand** with regard to a particular opcode state, a random number *x* shall be chosen according to the pdf

$p(x) = \begin{cases} 0 & \text{If } x \le 0, \text{ or} \\ k \exp(-kx), \text{ where } k = 1 / (\mathbf{p1} * \mathbf{SR}), & \text{otherwise.} \end{cases}$

where **SR** is the orchestra sampling rate.

The return value shall be 0 and the floor of this random value shall be stored.

On subsequent calls, the stored value shall be decremented by 1. If the decremented value is -1, the return value shall be 1 and a new random value shall be generated and stored as described above. Otherwise, the return value shall be 0.

NOTE - The **apoissonrand** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **apoissonrand** is referenced twice in the same a-cycle, then the effective mean time between 1 values is half as long as given by **t**.

5.9.8.13 igaussrand

```
iopcode igaussrand(ivar mean, ivar var)
```

The **igaussrand** core opcode generates a random number drawn from a Gaussian (normal) distribution with mean **mean** and variance **var**.

It is a run-time error if **var** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2p \times \mathbf{var}}}$$

that is, *p(x)* **~** N(**mean**, **var**) where **mean** is the mean and **var** the variance of a normal distribution.

5.9.8.14 kgaussrand

```
kopcode kgaussrand(ksig mean, ksig var)
```

The **kgaussrand** core opcode generates random numbers drawn from a Gaussian (normal) distribution with mean **mean** and variance **var**.

It is a run-time error if **var** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2p \times \mathbf{var}}},$$

that is, *p(x)* **~** N(**mean**, **var**) where **mean** is the mean and **var** the variance of a normal distribution.

5.9.8.15 agaussrand

```
aopcode agaussrand(asig mean, asig var)
```

The **agaussrand** core opcode generates random noise drawn from a Gaussian (normal) distribution with mean **mean** and variance **var**.

It is a run-time error if **var** is not strictly positive.

The return value shall be a random number *x* chosen according to the pdf

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2p \times \mathbf{var}}},$$

that is, *p(x)* **~** N(**mean**, **var**) where **mean** is the mean and **var** the variance of a normal distribution.

5.9.9 Filters

5.9.9.1 port

```
kopcode port(ksig ctrl, ksig htime)
```

The **port** core opcode converts a step-valued control signal into a portamento signal. **ctrl** is an incoming control signal, and **htime** is the half-transition time in seconds to slide from one value to the next.

The return value is calculated as follows. On the first call to **port** with regard to a particular state, the *current value* and *old value* are both set to **ctrl**. On subsequent calls, if **ctrl** is not equal to the new value, then the old value is set to the current value, the *new value* is set to **ctrl** and the *current time* is set to 0.

If **htime** is 0, the current value is set to the new value.

The return value is calculated as follows. If the current value and new value are equal, then the return value is the new value. Otherwise, the current time is incremented by 1/**KR**, where **KR** is the orchestra control rate. Then, the current value shall be set to $o + (n - o)(1 - 2^{-t / \mathbf{htime}})$, where t is the current time, n is the new value, and o is the old value.

NOTE - The **port** opcode does not have a "proper" representation of time, but infers it from the number of calls. If the same state of **port** is referenced twice in the same k-cycle, then the effective half-transition time is half as long as given by **htime**.

5.9.9.2 hipass

```
aopcode hipass(asig input, ksig cut)
```

The **hipass** core opcode high-pass filters its input signal. **cut** is the –6 dB cut-off point of the filter, and is specified in Hz. It is a run-time error if **cut** is not strictly positive.

The particular method of high-pass filtering is not normative. Any filter with the specified characteristic may be used.

The return value shall be the result of filtering **input** with a high-pass filter at **cut**.

NOTE - The **hipass** opcode is not required to have a "proper" representation of time, but is permitted to infer it from the number of calls. If the same state of **hipass** is referenced twice in the same a-cycle, the result is unspecified.

5.9.9.3 lopass

```
aopcode lopass(asig input, ksig cut)
```

The **lopass** core opcode low-pass filters its input signal. **cut** is the –6 dB cut-off point of the filter, and is specified in Hz. It is a runtime error if **cut** is not strictly positive.

The particular method of low-pass filtering is not normative. Any filter with the specified characteristic may be used.

The return value shall be the result of filtering **input** with a low-pass filter at **cut**.

NOTE - The **lopass** opcode is not required to have a "proper" representation of time, but is permitted to infer it from the number of calls. If the same state of **lopass** is referenced twice in the same a-cycle, the result is unspecified.

5.9.9.4 bandpass

```
aopcode bandpass(asig input, ksig cf, ksig bw)
```

The **bandpass** core opcode band-pass filters its input signal. **cf** is the centre frequency of the passband, and is specified in Hz. **bw** is the bandwidth of the filter, measuring from the –6 dB cut-off point below the centre frequency to the –6 dB point above, and is specified in Hz. It is a runtime error if **cf** and **bw** are not both strictly positive.

The particular method of bandpass filtering is not normative. Any filter with the specified characteristic may be used.

The return value shall be the result of filtering **input** with a bandpass filter with centre frequency **cf** and bandwidth **bw**.

NOTE - The **bandpass** opcode is not required to have a "proper" representation of time, but is permitted to infer it from the number of calls. If the same state of **bandpass** is referenced twice in the same a-cycle, the result is unspecified.

5.9.9.5 bandstop

```
aopcode bandstop(asig input, ksig cf, ksig bw)
```

The **bandstop** core opcode band-stop (notch) filters its input signal. **cf** is the centre frequency of the stopband, and is specified in Hz. **bw** is the bandwidth of the filter, measuring from the –6 dB cut-off point below the centre frequency to the –6 dB point above, and is specified in Hz. It is a runtime error if **cf** and **bw** are not both strictly positive.

The particular method of notch filtering is not normative. Any filter with the specified characteristic may be used.

The return value shall be the result of filtering **input** with a bandstop filter with centre frequency **cf** and bandwidth **bw**.

NOTE - The **bandstop** opcode is not required to have a "proper" representation of time, but is permitted to infer it from the number of calls. If the same state of **bandstop** is referenced twice in the same a-cycle, the result is unspecified.

5.9.9.6 biquad

```
aopcode biquad(asig input, ivar b0, ivar b1, ivar b2, ivar a1, ivar a2)
```

The **biquad** core opcode performs exactly normative filtering using the canonical second-order filter in a "Transposed Direct Form II" structure. Using cascades of **biquad** opcodes allows the construction of arbitrary filters with exactly normative results.

The return value is calculated as follows. On the first call to **biquad** with regard to a particular state, the intermediate variables **ti, to, w0, w1,** and **w2** are set to 0. Then, on the first call and each subsequent call, the following pseudo-code defines the functionality:

```
ti := input + a1 * w1 + a2 * w2
to := b0 * ti + b1 * w1 + b2 * w2
w2 := w1
w1 := w0
w0 := ti
```

and the return value is **to**.

A runtime error is produced if this process produces out-of-bounds values (i.e., the filter is unstable).

The **biquad** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **biquad** is referenced twice in the same a-cycle, the effective sampling rate of the filter is twice as high as the orchestra sampling rate.

5.9.9.7 allpass

```
aopcode allpass(asig input, ivar time, ivar gain)
```

The **allpass** core opcode performs allpass filtering on an input signal. The length of the feedback delay is **time** and is specified in seconds. It is a run-time error if **time** is not strictly positive.

Let **t** be the value **time * SR**, where **SR** is the orchestra sampling rate. On the first call to **comb** with regard to a particular state, a delay line of length **t** is initialised and set to all zeros.

On the first and each subsequent call, let **x** be the value that was inserted into the delay line **t** calls ago, or 0 if there have not been **t** calls to this state. Let the value **output** be **x – input * gain**. Insert the value **output** * **gain + input** into the beginning of the delay line. The return value shall be **output**.

NOTE - The **allpass** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **allpass** is referenced twice in the same a-cycle, then the effective allpass length is half as long as **len**.

5.9.9.8 comb

```
aopcode comb(asig input, ivar time, ivar gain)
```

The **comb** core opcode performs comb filtering on an input signal. The length of the feedback delay is **time** and is specified in seconds. It is a run-time error if **time** is not strictly positive.

Let **t** be the value **time * SR**, where **SR** is the orchestra sampling rate. On the first call to **comb** with regard to a particular state, a delay line of length **t** is initialised and set to all zeros.

On the first and each subsequent call, let **x** be the value that was inserted into the delay line **t** calls ago, or 0 if there have not been **t** calls to this state. Insert the value **x** * **gain** + **input** into the beginning of the delay line. The return value shall be **x**.

NOTE - The **comb** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. That is, if the same state of **comb** is referenced twice in the same a-cycle, the effective length is half of **t**.

5.9.9.9 fir

```
aopcode fir(asig input, ksig b0[, ksig b1, ksig b2, ksig …])
```

The **fir** core opcode applies a specified FIR filter of arbitrary order to an input signal. The particular method of implementing FIR filters is not specified and left open to implementers.

The parameters **b0, b1, b2, …** specify a FIR filter

$$H(z) = \mathbf{b0} + \mathbf{b1}\, z^{-1} + \mathbf{b2}\, z^{-2} + \ldots$$

The return value shall be the successive values given by the application of this filter to the signal given by the value of **input** in successive calls to **fir**.

NOTE - The **fir** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **fir** is referenced twice in the same a-cycle, the effective filter sampling rate is double that of the orchestra sampling rate.

5.9.9.10 iir

```
aopcode iir(asig input, ksig b0[, ksig a1, ksig b1, ksig a2, ksig b2, ksig …])
```

The **iir** core opcode applies a specified IIR filter of arbitrary order to an input signal. The particular method of implementing IIR filters is not specified and left open to implementers.

The parameters **b0, b1, b2, …** and **a1, a2, …** specify an IIR filter

$$H(z) = (\mathbf{b0} + \mathbf{b1}z^{-1} + \mathbf{b2}z^{-2} + \ldots) / (1 + \mathbf{a1}z^{-1} + \mathbf{a2}z^{-2} + \ldots).$$

The return value shall be the successive values of the signal given by the application of this filter to the signal given by **input** in successive calls to **iir**. It is a run-time error if this application produces out-of-range values (that is, if the filter is unstable).

NOTE - The **iir** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **iir** is referenced twice in the same a-cycle, the effective filter sampling rate is double that of the orchestra sampling rate.

5.9.9.11 firt

```
aopcode firt(asig input, table t[, ksig order])
```

The **firt** core opcode applies a specified FIR filter of arbitrary order, given in a table, to an input signal. The particular method of implementing FIR filters is not specified and left open to implementers.

The values stored in samples 0, 1, 2, … **order** of table **t** specify a FIR filter

$$H(z) = \mathbf{t}[0] + \mathbf{t}[1]\, z^{-1} + \mathbf{t}[2]\, z^{-2} + \ldots\ \mathbf{t}[\mathbf{order}\text{-}1]\, z^{-\mathbf{order}+1}$$

where array notation is used to indicate wavetable samples. If **order** is not given or is greater than the size of the wavetable **t**, then **order** shall be set to the size of the wavetable. It is a run-time error if **order** is zero or negative.

The return values shall be the successive values of the signal given by the application of this filter to the signal given by the value of **input** in successive calls to **firt**.

NOTE - The **firt** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **firt** is referenced twice in the same a-cycle, the effective filter sampling rate is double that of the orchestra sampling rate.

5.9.9.12 iirt

```
aopcode iirt(asig input, table a, table b[, ksig order])
```

The **iirt** core opcode applies a specified IIR filter of arbitrary order, given in two tables, to an input signal. The particular method of implementing IIR filters is not specified and left open to implementers.

The values stored in samples 1, 2, … **order** of table **a** and samples 0, 1, 2, …, **order** of wavetable **b** specify a IIR filter

$$H(z) = (\mathbf{b}[0] + \mathbf{b}[1]z^{-1} + \mathbf{b}[2]z^{-2} + ...) / (1 + \mathbf{a}[1]z^{-1} + \mathbf{a}[2]z^{-2} + ...)$$

where array notation is used to indicate wavetable samples. (Note that sample 0 of wavetable **a** is not used). If **order** is not given or is greater than the size of the larger of the two wavetables, then **order** shall be set to the size of the greater of the two wavetables. If one wavetable is smaller than given by **order**, then the "extra" values shall be taken as zero coefficients It is a run-time error if **order** is zero or negative.

The return values shall be the successive values of the signal given by the application of this filter to the signal given by the value of **input** in successive calls to **iirt**. It is a run-time error if this application produces out-of-range values (that is, if the filter is unstable).

NOTE - The **iirt** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **iirt** is referenced twice in the same a-cycle, the effective filter sampling rate is double that of the orchestra sampling rate.

5.9.10 Spectral analysis

5.9.10.1 fft

```
specialop fft(asig input, table re, table im[, ivar len, ivar shift, ivar size, table win])
```

The **fft** core opcode calculates windowed and overlapped DFT frames and places the complex-valued result in two tables. It is a "special opcode"; that is, it accepts values at the audio rate, but only returns then at the control rate.

There are several optional parameters. **len** specifies the length of the sample frame (the number of input samples to use). If **len** is zero or not provided, it is set to the next power of two greater than **SR/KR**, where **SR** is the orchestra sampling rate and **KR** is the orchestra control rate. **shift** specifies the number of samples by which to shift the analysis window. If **shift** is zero or not provided, it is set to **len**. **size** is the length of the DFT calculated by the opcode. If **size** is zero or not provided, it is set to **len**. **win** is the analysis window to apply to the analysis. If **win** is not provided, a boxcar window of length **len** is used.

It is a runtime error if any of the following apply: **len** is negative, **shift** is negative, **size** is negative, **size** is not a power of two, **size** is greater than 8192, **win** has fewer than **len** samples, **re** has fewer than **size** samples, or **im** has fewer than **size** samples.

The calculation of this opcode is as follows: On the first call to the **fft** opcode with respect to a particular state, a holding buffer of length **len** is created. On each a-rate call to the opcode, the **input** sample is inserted into the buffer. When there are **len** samples in the buffer, the following steps are performed:

1. A new buffer is created of length **size**, for which each value **new[i]** is set to the value **buf[i]** * **win[i]**, where **new[i]** is the **i**th value of the new buffer, **buf[i]** is the value of the holding buffer, and **win[i]** is the value of the **i**th sample in the analysis-window wavetable. (The new buffer contains the pointwise product of the holding buffer and the analysis window). If **size > len**, then the values of **new[i]** for **i > len** are set to zero. If **size < len**, then only the first **size** values of the holding buffer are used.

2. The first **shift** samples are removed from the holding buffer and the remaining **len-shift** samples shifted to the front of the holding buffer. The **shift** samples at the end of the buffer after this shift are set to zero. If **shift** > **len**, the holding buffer is cleared.

3. The real DFT of the new buffer is calculated, resulting in a length-**size** complex vector of frequency-domain values. The real components of the DFT are placed in the first **size** samples of table **re**; the imaginary components of the DFT are placed in the first **size** samples of table **im**. The DFT is arranged such that the lowest frequencies, starting with DC, are at the zero point of the output tables, going up to the Nyquist frequency at **size/2**; the reflection of the spectrum from the Nyquist to the sampling frequency is placed in the second half of the tables.

The DFT is defined as

$$d[i] = \frac{\sum_{k=0}^{len-1} e^{-ijk\,2p/len} x[k]}{\sqrt{len}}$$

where **d[i]** are the resulting complex components of the DFT, 0 < **i** < **size**;
x[k] are the input samples, 0 < **k** < **len**;
and ***j*** is the square root of –1.

The return value on a particular k-cycle is 1 if a DFT was calculated since the last k-cycle, or 0 if one was not.

The **fft** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **fft** is referenced twice in the same a-cycle, then the effective FFT period is half as long as given by **len** and **shift**.

5.9.10.2 ifft

```
aopcode ifft(table re, table im[, ivar len, ivar shift, ivar size, table win])
```

The **ifft** core opcode calculates windowed and overlapped IDFTs and streams the result out as audio. **re** and **im** are wavetables that contain the real and imaginary parts of a DFT, respectively. There are several optional arguments that control the synthesis procedure. **len** is the number of output samples to use as audio; if **len** is zero or not given, it is taken as the next power of two greater than **SR/KR**, where **SR** is the orchestra sampling rate and **KR** is the orchestra control rate. **shift** is the number of samples by which the analysis window is shifted between frames; if **shift** is not given or is zero, it is taken as **len**. **size** is the size of the IDFT; if **size** is not given or is zero, it is taken as **len**. **win** is the synthesis window; if it is not given, a boxcar window of length **len** is assumed.

It is a run-time error if any of the following apply: **re** or **im** are shorter than length **size**, **win,** if given, is shorter than length **len**, **size** is not a power of two, **size** is greater than 8192, or **len**, **shift**, or **size** are negative.

The calculation for this opcode is as follows. On the first call to **ifft** with respect to a particular state, the size **size** IDFT of the tables **re** and **im** is calculated. If **re** and/or **im** are longer than **size** samples, only the first **size** samples of these tables shall be used. The result of this IDFT is a sequence of **size** values, potentially complex-valued. The real components of the first **len** elements of this sequence are multiplied point-by-point by the corresponding samples of the window **win** and placed in an output buffer of length **len**. (**out[i] = seq[i] * win[i]** for 0 < **i** < **len**).

The IDFT is calculated with the assumption that the lowest-numbered elements of the tables **re** and **im** are the lowest frequencies of the audio signal, beginning with DC in sample 0, proceeding up to the Nyquist frequency in sample **size/**2, and then the reflected spectrum in samples **size/**2 up to **size**-1.

The IDFT is defined as

$$x[i] = \frac{\sum_{k=0}^{len-1} e^{ijk\,2p/len} d[k]}{\sqrt{len}}$$

where **d[i]** are the complex frequency components of the DFT, 0 < **i** < **size** (**d[i] = re[i] + *j* im[i]**)
x[k] are the input samples, 0 < **k** < **len**;
and ***j*** is the square root of –1.

Also on the first call to **ifft** with respect to a particular state, the output point of the synthesis is set to 0.

At each call to **ifft**, the following calculation is performed. The output value of the opcode is the value of the output buffer at the output point. Then, the output point is incremented. If the output point is thereby equal to **shift**, then the following steps shall be performed:

1. The first **shift** samples of the output buffer are discarded, the remaining **len-shift** samples of the output buffer are shifted into the beginning of the buffer, and the last **shift** samples are set to 0.

2. The IDFT of the current values of the **re** and **im** wavetables is calculated as described above. The first **len** values of the real part of the resulting audio sequence are multiplied point-by-point by the synthesis window **win**, and the result is added point-by-point to the output buffer (**out[i] = out[i] + seq[i] * win[i]** for 0 < **i** < **len**).

3. The output point is set to 0.

NOTE - The **ifft** opcode shall not have a “proper” representation of table, but shall infer it from the number of calls. If the same state of **ifft** is referenced twice in the same a-cycle, the result is undefined.

EXAMPLE

The **ifft** and **fft** opcodes can be used together to write instruments that use FFT-based spectral modification techniques, with logical syntax at the instrument level, in which the FFT frame rate is asynchronous with the control rate. The structure of such an instrument is:

```
instr spec_mod() {
  asig out;
  ksig new_fft;
  ivar length;
  table t(empty,1025);

  length = 256;
  new_fft = fft(input,re,im,1024,length); // no windowing; place FFT in “t”
  if (new_fft) { // modify table data if there’s a new spectrum
     .
     .
     .
  }

  // and output IFFT
  out = ifft(re,im,1024,length);
  output(out);
  }
```

Thus, every 256 samples (assuming 256 is greater than the number of samples in the control period), we compute the 1024-point IFFT. On those k-cycles during which we compute the FFT, we modify the table values in some interesting way. The IFFT operator produces continuous output, where every 256 samples the new table data is inspected and the IFFT calculated

There is nothing preventing us from manipulating the table data every control period, but only those values present in the table at the IFFT times will actually be turned into sound.

FFTs and IFFTs do not need to be implemented in pairs; other methods (such as table calculations) can be used to generate spectra to be turned into sound with IFFT, or rudimentary audio-pattern-recognition tools can be constructed that compute functions of the FFT and return or export the results.

5.9.11 Gain control

5.9.11.1 rms

```
specialop rms(asig x[, ivar length])
```

The **rms** core opcode calculates the power in a signal. It is a “special opcode”; that is, it accepts values at the audio rate, but only returns them at the k-rate.

If **length** is not provided, it shall be set to the length of the control period. It is a run-time error if **length** is provided and is negative. The **length** parameter is specified in seconds.

The return value is calculated as follows. Let *l* be the value floor(**length * SR)**, where **SR** is the orchestra sampling rate. A buffer *b***[],** of length *l,* is maintained of the most recent values provided as the **x** parameter. Each control period, the RMS of these values is calculated as

$$p = \sqrt{\frac{\sum_{i=0}^{l-1} b[i]^2}{l}}$$

and the return value is *p*.

NOTE - The **rms** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **rms** is referenced twice in the same a-cycle, then the effective length is half as long as given by **length**.

5.9.11.2 gain

```
aopcode gain(asig x, ksig gain[, ivar length])
```

The **gain** core opcode attenuates or increases the amplitude of a signal to make its power equal to a specified power level.

If **length** is not provided, it shall be set to the length of the control period. It is a run-time error if **length** is provided and is not strictly positive. The **length** parameter is specified in seconds.

The return value is calculated as follows. Let *l* be the value floor(**length * SR)**, where **SR** is the orchestra sampling rate.

At the first call to the opcode, the attenuation level is set to 1. At each subsequent call, the input value **x** shall be stored in a buffer **b[]** of length *l.* When the buffer is full, the attenuation level is recalculated as

$$\frac{\text{gain}}{\sqrt{\frac{\sum_{i=0}^{l-1} b[i]^2}{l}}}$$

and the buffer is cleared.

The return value at each call is **x** * **A**, where **A** is the current attenuation level.

NOTE - The **gain** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **gain** is referenced twice in the same a-cycle, then the effective buffer length is half as long as given by **length**.

5.9.11.3 balance

```
aopcode balance(asig x, asig ref[, ivar length])
```

The **balance** core opcode attenuates or increases the amplitude of a signal to make its power equal to the power in a reference signal.

If **length** is not provided, it shall be set to the length of the control period. It is a run-time error if **length** is provided and is not strictly positive. The **length** parameter is specified in seconds.

The return value is calculated as follows. Let *l* be the value floor(**length * SR)**, where **SR**, is the orchestra sampling rate.

At the first call to the opcode, the attenuation level is set to 1. At each subsequent call, the input value **x** shall be stored in a buffer **b[]** of length *l***,** and the input value **ref** stored in a buffer **r**[] of length *l*. When the buffers are full, the attenuation level is recalculated as

$$\frac{\sqrt{\frac{\sum_{i=0}^{l-1} r[i]^2}{l}}}{\sqrt{\frac{\sum_{i=0}^{l-1} b[i]^2}{l}}}$$

and the buffers are cleared.

The return value at each call is **x** * **A**, where **A** is the current attenuation level.

NOTE - The **balance** opcode shall not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **balance** is referenced twice in the same a-cycle, then the effective buffer length is half as long as given by **length**.

5.9.11.4 compressor

```
aopcode compressor (asig x, asig comp, ksig thresh, ksig loknee, ksig hiknee,
                    ksig ratio, ksig att, ksig rel, isig look)
```

The **compressor** core opcode functions as an audio compressor, limiter, expander, or noise gate, using either soft-knee or hard-knee mapping, and with dynamically variable performance characteristics. It takes two audio-rate input signals, **x** and **comp**, the first of which is modified by a running analysis of the second. Both signals may be the same, or the first can be modified by a different controlling signal.

It is a run-time error if any of the following conditions apply: **thresh** is negative, **loknee < thresh**, **hiknee < loknee**, **ratio** is 0, **look** is negative, **att** is negative, or **rel** is negative.

compressor first examines the controlling signal **comp** by performing envelope detection. This is directed by two control values **att**, **rel**, and an initialisation value **look**, defining the attack, release, and look-ahead times (in seconds) of the envelope detector.

look is the look-ahead time (in seconds) of the envelope detector. This determines how far ahead the detector looks for a new peak in a decaying signal. If a new peak is found, the release envelope is adjusted to interpolate between the current and future peaks.

att and **rel** are the attack and release times of the envelope detector (in seconds). They define the time it takes the envelope to reach a detected peak value (for **att**) and to reach zero (for **rel**).

The envelope estimate is converted to decibels, then passed through a mapping function (see Figure 5.3) to determine what compressor action (if any) shall be taken. The mapping function is defined by four regions: the zero region, the 1:1 (no change) region, the knee, and the compression/expansion region.

The locations of these regions are defined by the decibel control values, **thresh**, **loknee**, **hiknee**, and **ratio**.

thresh is the minimum decibel level that will be allowed through. For a noise gate, this value will be greater than zero. If the value is less than zero, it has the same effect as zero.

loknee and **hiknee** are the decibel values that define where compression or expansion will begin. These set the boundaries of a soft-knee curve joining the low-amplitude 1:1 line and the higher-amplitude compression/expansion line. If the two breakpoints are equal, a hard-knee mapping will result.

ratio is the ratio of compression above the knee. Higher numbers result in greater compression. Numbers between zero and one result in expansion. Negative numbers perform an inversion in addition to compression and expansion. A value of zero will result in a run-time error.

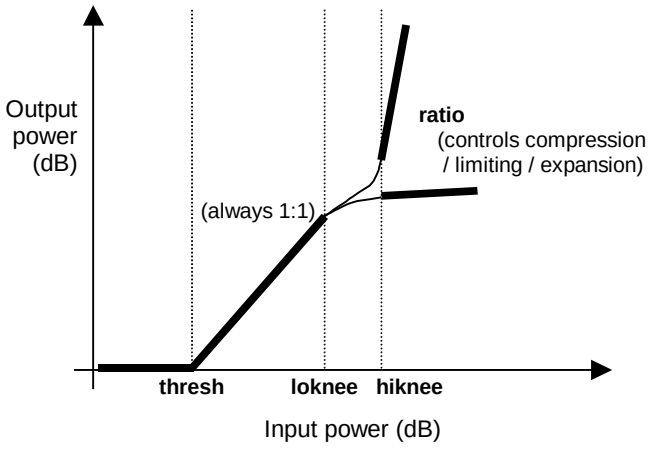


Figure 5.3 - Compressor characteristic function

The actions of **compressor()** will depend on the parameter settings given. A hard-knee compressor-limiter, for instance, is obtained from a near-zero attack time, equal-value **loknee** and **hiknee** break-points, and a very high compression index (**ratio**). A noise-gate plus expander is obtained from some positive **thresh**, and a fractional **ratio** above the knee. A voice-activated music compressor (ducker) will result from feeding the music into **x** and the speech into **comp**. A voice de-esser will result from feeding the voice into both, with the **comp** version being preceded by a band-pass filter that emphasises the sibilants.

At initialisation, space shall be allocated for two buffers **xdly** and **compdly**. The length in samples of these buffers will be SR * **look**, where SR is the orchestra sampling rate. The initial values of both buffers will be set to zero.

Space is allocated for the following variables:

gain, the amplitude multiplier to be applied, initialised to 1.

change, the estimated change in envelope from sample to sample, initialised to 0.

comp1, the value of the current value of **comp** in dB, initialised to 0.

comp2, the delayed value of **comp** (from **compdelay**) in dB, initialised to zero.

env, the current envelope estimate, initialised to 0.

projEnv, the projected envelope value at the look-ahead point, initialised to 0.

At each a-rate call to **compressor()**, the following happens:

1. The sample **x** is placed into the beginning of the buffer **xdly**, pushing all values down and the oldest value off the end. The value pushed off the end is saved as **oldval**.
2. The next envelope value is calculated as follows:

   abs(**comp**) is converted to decibels. This value is called **comp1** (**comp1 :=** 90 + 20*log10(abs(**comp**)). It is put into the end of the buffer **compdly**. The value **comp2** is taken from the beginning of the buffer **compdly**.

   if (**comp2** > **env**) **change** = (**comp2** – **env**) / (SR * **att**)
   else **projEnv** = **change** * (SR * **look**), if **projEnv** < 0, **projEnv** = 0

   if (**comp1** > **projEnv**) & (**comp1** < **comp2**), **change** = (**comp1** – **comp2**) / (SR * **rel**)

   env = **env** + **change**

   if **env** < 0, then **env** = 0
3. The amplitude multiplier **gain** is calculated as follows:

   if (**env** < **thresh**) **gain** = 0

   else if (**env** > **thresh**)

   if (**env** < **loknee**)

   gain = 1

   else if (**env** = **hiknee**)

   gain = 10 ^ ([**comp** / **ratio** + (**hiknee** – **loknee**) / 2 * (1 – 1/**ratio**)] / 20)

   else if (**loknee** = **env** < **hiknee**)

   gain shall be interpolated smoothly between the **loknee** and **hiknee** points to create a soft-knee curve. This curve shall be monotonically increasing, and have continuous derivative equal to 1 at **loknee** and **ratio** at **hiknee**. The exact curve is not specified; it is left to the implementation.
4. The output value of the opcode shall be **oldval** multiplied by **gain**.

5.9.12 Sample conversion

5.9.12.1 decimate

```
specialop decimate(asig input)
```

The **decimate** core opcode decimates a signal from the audio rate to the control rate. It is a “special opcode”; that is, it accepts values at the audio rate, but only returns them at the k-rate.

The return value is calculated as follows. Each k-cycle, one of the values given as **input** in the preceding k-period of a-samples shall be returned.

NOTE - The **decimate** opcode is not required to have a “proper” representation of time, but is allowed to infer it from the number of calls. If the same state of **decimate** is referenced twice in the same a-cycle, then the return value for each call at the subsequent k-cycle may be taken from any of the values provided to the state during the preceding k-period.

EXAMPLE

```
oparray decimate[2];
ksig a,b,c;

a = decimate[0](1);
b = decimate[0](0);
c = decimate[1](2);
```

The value of **a** and **b** at each k-cycle shall be either 0 or 1, in an implementation-dependant manner. The value of **c** shall be 2.

5.9.12.2 upsamp

```
asig upsamp(ksig input[, table win])
```

The **upsamp** core opcode upsamples a control signal to an audio signal. **win** is an optional interpolation window. If **win** is not provided, it is taken to be a boxcar window (all values equal 1) of length **SR / KR**, where **SR** is the orchestra sampling rate and **KR** is the orchestra control rate. If **win** is provided and is shorter than **SR / KR** samples, it is zero-padded at the end to length **SR/KR** for use in this opcode (the table itself is not changed).

On the first call to **upsamp** with regard to a particular state, an output buffer of length **win** is created and set to zero. Also, the *output point* is set to 0.

On the first call to **upsamp** in a particular k-cycle with regard to a particular cycle, the output buffer is shifted by **SR / KR** samples: the first **SR / KR** samples are discarded, the remaining samples are shifted to the front of the output buffer, and the last **SR / KR** samples are set to 0. Then, the window function is scaled by **input** and added into the output buffer (**buf[i] = buf[i] + input * win[i],** 0 < **i** < length(**win**)). Then, the output point is set to 0.

On the first call and each subsequent call to **upsamp**, the return value is the value of the output buffer at the current output point. Then, the output point shall be incremented.

It is a run-time error if the same state of **upsamp** is referenced more times than the length of **win** in a single k-cycle.

5.9.12.3 downsamp

```
specialop downsamp(asig input[, table win])
```

The **downsamp** core opcode downsamples an audio signal to a control signal. It is a “special opcode”; that is, it accepts samples at the audio rate but only returns values at the control rate. **win** is an optional analysis window.

It is a run-time error if **win** is shorter than **SR / KR** samples, where **SR** is the orchestra sampling rate and **KR** is the orchestra control rate.

The return value is calculated as follows: at each k-cycle, the values of each sample of **input** provided in the previous a-cycle are placed in a buffer. If **win** is not provided, then the return value is the mean of the samples in the buffer. If **win** is provided, then the return value is calculated by multiplying the analysis window point-by-point with the input signal (**rtn** = Σ **input[i] * win[i]** for 0 < **i** < **SR * KR**, where **SR** is the orchestra sampling rate and **KR** is the orchestra control rate).

NOTE - The **decimate** opcode does not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **decimate** is referenced twice in the same a-cycle, then the return value is calculated from the input values in the second half of the k-cycle.

5.9.12.4 samphold

```
opcode samphold(xsig input, ksig gate)
```

The **samphold** core opcode gates a signal with a control signal.

The return value is calculated as follows. On the first call to **samphold** with regard to a particular state, the last passed value is set to 0. If the value of **gate** is non-zero, then the last passed value is set to **input**. The last passed value is returned.

5.9.12.5 sblock

```
specialop sblock(asig x, table t)
```

The **sblock** core opcode creates control-rate blocks of samples and places them in a wavetable. It is a "special opcode"; that is, it accepts values at the audio rate, but only returns them at the k-rate.

It is a run-time error if the table **t** is not allocated with as much space as there are samples in the control period of the orchestra.

The return value of this opcode is always 0.

This opcode has side effects, as follows. Let *k* be the number of samples in a control period. At each k-cycle, the most recent *k* values of **x** are placed in table **t** such that the oldest value is placed in sample 0.

NOTE - The **sblock** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **sblock** is referenced twice in the same a-cycle, then the samples placed in the table shall be the interleaved values given in the two calls during the second half of the k-period.

5.9.13 Delays

5.9.13.1 delay

```
aopcode delay(asig x, ivar t)
```

The **delay** opcode implements a fixed-length end-to-end (i.e., untapped) delay line. **t** gives the length of the delay line in seconds. It is a run-time error if **t** < 0, unless the terminal is running in a negative-time universe.

Let *y* be floor(**t** * **SR**) samples, where **SR** is the orchestra sampling rate. At each call to **delay** with respect to a particular opcode state, the value of **x** is inserted into a FIFO buffer of length *y*. The return value is the value that was inserted into the delay line *y* calls ago to **delay** with regard to the same state.

NOTE - The **delay** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **delay** is referenced twice in the same a-cycle, then the effective delay line is half as long as given by **t**.

5.9.13.2 delay1

```
aopcode delay1(asig x)
```

The **delay1** opcode implements a single-sample delay.

At each call to **delay1** with regard to a particular state, the value of **x** is stored, and the return value is the value stored on the previous call.

NOTE - The **delay1** opcode shall not have a "proper" representation of time, but shall infer it from the number of calls. If the same state of **delay1** is referenced twice in the same a-cycle, then the return value for the second call is the parameter value of the first.

5.9.13.3 fracdelay

```
aopcode fracdelay(ksig method[, xsig p1, xsig p2])
```

The **fracdelay** core opcode implements fractional, variable-length, and/or multitap delay lines. Several methods for manipulating the delay line are provided; in this way, **fracdelay** is like an object-oriented delay-line "class".

The semantics of **p1**and **p2** and the calculation of the return value differ depending on the value of **method**. It is a run-time error if **method** is less than 1 or greater than 5.

If **method** is 1, the "initialise" method is specified. In this case, **p1** is the length of the delay line in seconds. It is a run-time error if **p1** is not provided, or is less than 0. Any currently existing delay line in this opcode state shall be destroyed, a new delay line with the specified length (floor(**p1*** **SR**), where **SR** is the orchestra sampling rate) shall be created, and all values on this delay line shall be initialised to 0. The return value is 0. **p2** is not used, and is ignored if provided.

If **method** is 2, the "tap" method is specified. In this case, **p1** is the position of the tap in seconds. It is a run-time error if method 1 has not yet been called for this opcode state, or if **p1** is not provided, or if **p1** is less than 0, or if **p1** is greater than the most recent initialisation length. The return value is the current value of the delay line at position **p1** * **SR**, where **SR** is the orchestra sampling rate. If **p1 * SR** is not an integer, the return value shall be interpolated from the nearby values, as described by the global interpolation-quality parameter (subclause 5.8.5.2.5). **p2** is not used, and is ignored if provided.

If **method** is 3, the "set" method is specified. In this case, **p1** is the position of the insertion in seconds, and **p2** is the value to insert. It is a run-time error if method 1 has not yet been called for this opcode state, or if **p1** is not provided, or if **p1** is less than 0, or if **p1**is greater than the most recent initialisation length, or if **p2** is not provided. The value of the delay line at position floor(**p1 * SR)**, where **SR** is the orchestra sampling rate, is updated to **p2**. The return value is 0.

If **method** is 4, the "add into" method is specified. In this case, **p1** is the position of the insertion in seconds, and **p2** is the value to add in. It is a run-time error if method 1 has not yet been called for this opcode state, or if **p2** is not provided. Let *x* be the current value of the delay line at position floor(**p1 * SR**), where **SR** is the orchestra sampling rate; then, the value of the delay line at this position is updated to *x* + **p2**. The return value is *x* + **p2**.

If **method** is 5, the "shift" method is specified. It is a run-time error if method 1 has not yet been called for this opcode state. All values of the delay line are shifted forward by one sample; that is, for each sample *x* where 0 < *x* <= L, where L is the length of the delay line, the new value of sample *x* of the delay line is the current value of sample *x* – 1. Sample 0 is set to value 0. The return value is the value shifted "off the end" of the delay line, that is the current value of sample L. **p1**and **p2** are not used, and are ignored if provided.

EXAMPLE

The following user-defined opcode implements the block diagram in Figure 5.4. We assume that the orchestra sampling rate is 10 Hz for clarity.

```
aopcode example(asig a) {
  asig t1, t2, t3, x, first;
  oparray fracdelay[1];

  if (!itime) {
    fracdelay[0](1,1);       // initialise to 1 sec long
  }

  // flow network
  fracdelay[0](3,0,a);       // insert a at beginning
  t1 = fracdelay[0](2,0);    // tap at 0
  t2 = fracdelay[0](2,0.3);  // tap at 0.3
  t3 = fracdelay[0](2,0.5);  // tap at 0.5
  fracdelay[0](4,0.1,t3);    // feedback
  fracdelay[0](4,0.8,t1+t2); // feedforward
  x = fracdelay[0](5);       // shift and get output
  return(x);
}
```

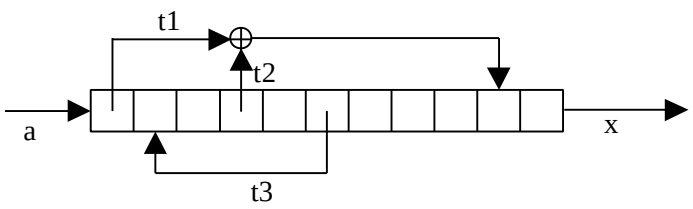


Figure 5.4 - Block diagram for 'fracdelay' example

Notice the use of the oparray construction (subclause 5.8.6.7.7) to implement this network. If an oparray is not used, then each call to **fracdelay** refers to a different delay line, and the algorithm makes no sense. Also note that **fracdelay**, unlike **delay**, does not shift automatically. For "typical" operations, method 5 should be called once per a-cycle.

5.9.14 Effects

5.9.14.1 reverb

```
aopcode reverb(asig x, ivar f0[, ivar r0, ivar f1, ivar r1, ivar …])
```

The **reverb** core opcode produces a reverberation effect according to the given parameters.

It is a run-time error if any **f** or **r** value is negative, or if there are an even number of parameters greater than 2.

If only one value **f0** is given as an argument, it is taken as a full-range reverberation time, that is, the amount of time delay until the sound amplitude is attenuated 60 dB compared to the source sound (RT60).

If more values are given, the **f** – **r** pairs represent responses at different frequencies. At each frequency **f** given as a parameter, the reverberation time (RT60) at that frequency is given by the corresponding **r** value.

The exact method of calculating the reverberation according to the specified parameters is not normative. If content authors wish to have exactly normative reverberations, they can easily be authored using the **comb**, **allpass**, **biquad**, **delay**, **fracdelay**, and other strictly normative core opcodes (q.v.).

The output shall be the reverberated sound signal.

5.9.14.2 chorus

```
asig chorus(asig x, ksig rate, ksig depth)
```

The **chorus** core opcode creates a sound with a chorusing effect, with rate **rate** and depth **depth**, from the input sound **x**. **rate** is specified in cycles per second; **depth** is specified as percent excursion.

The exact method of chorusing is non-normative and left open to implementers.

5.9.14.3 flange

```
asig flange(asig x, ksig rate, ksig depth)
```

The **flange** core opcode creates a sound with a flanged effect, with rate **rate** and depth **depth**, from the input sound **x**. **rate** is specified in cycles per second; **depth** is specified as percent excursion.

The exact method of flanging is non-normative and left open to implementers.

5.9.14.4 fx_speedc

```
kopcode fx_speedc(ivar speed_control_factor)
```

The **fx_speedc** core opcode creates a sound with a speed change effect. This core opcode is only available in effects-processing orchestras (see subclause 5.15.3.5); it may not be used in the SAOL block of a Structured Audio bitstream.

The **fx_speedc** core opcode directly accesses the special bus **input_bus** (at ksmps per 1/k-rate) and performs a speed change at k-rate. Then the processed samples are once stored in a special buffer defined only for the speed control and are output as the signal at a rate of *k*/**speed_control_factor**, where *k* = SR/KR, SR is the orchestra sampling rate and KR is the orchestra control rate.

This indicates that the number of the input samples and that of the output samples are different when the sampling frequency is unchanged and the **fx_speedc** core opcode shall be operated virtually at a rate of **speed_control_factor*** KR in order to keep the proper output sample-rate of *k* per control cycle. Therefore the input

samples shall be also provided to the decoder at a rate of *k* per 1/ **speed_control_factor** * KR according to the **speed_control_factor** desired.

NOTE - This functionality is not supported for streaming transfer in ISO/IEC 14496-1 (MPEG-4 Systems). It is not therefore possible to apply the speed change to streaming media and the use is limited to applications such as storage media in which streaming data transfer is not required.

The exact method of speed change is non-normative and left open to implementers. Implementers are encouraged to provide the highest-quality speed change possible. As an example algorithm, the PICOLA speed change tool is described in Annex 5.D.

5.9.14.5 speedt

```
iopcode speedt(table in, table out, ivar factor)
```

The **speedt** core opcode modifies a sound sample via time-scaling.

The **speedt** core opcode fills the sound sample in the wavetable **out** with a sound derived from the wavetable **in** by time-stretching without modifying the pitch. If **factor** < 1, the sound is compressed (accelerated) by the indicated factor; if **factor** > 1, the sound is expanded (slowed down) by the indicated factor. It is a run-time error if **factor** is not strictly positive, or if the wavetable **out** is not at least as long as **factor** multiplied by the length of the wavetable **in**.

The exact method of speed change is non-normative and left open to implementers. Implementers are encouraged to provide the highest-quality speed change possible. As an example algorithm, the PICOLA speed change tool is described in Annex 5.D.

5.9.15 Tempo functions

5.9.15.1 gettempo

```
opcode gettempo([xsig dummy])
```

The **gettempo** core opcode returns the value in beats-per-minute of the current orchestra global tempo. The tempo by default is 60 beats per minute, but can be changed through the use of the **tempo** score line (subclause 5.11.5) or the **settempo** core opcode (subclause 5.9.15.2).

The **dummy** parameter is used to specify the rate of the opcode call if desired; see subclause 5.8.7.7.2.

5.9.15.2 settempo

```
kopcode settempo(ksig x)
```

The **settempo** core opcode changes the value of the global orchestra tempo. The parameter **x** specifies the new tempo in beats-per-minute. It is a run-time error if **x** is not strictly positive. The return value is **x**.

This opcode has side-effects, as follows. All pending events are rescheduled as described in subclause 5.7.3.3.6, list item 7. The global orchestra tempo is set to **x**.

5.10 SAOL core wavetable generators

5.10.1 Introduction

This clause describes each of the core wavetable generators in SAOL. All core wavetable generators shall be implemented in every terminal that can decode Object type 3 or 4.

For each core wavetable generator, the following is described:

- A usage description, showing the parameters that are required to be provided in a table definition utilising this core wavetable generator.
- The normative semantics of the generator. These semantics describe how to calculate values and place them in the wavetable for each table definition using this generator.

For each core wavetable generator, the first field in the table definition is the name of the generator, and the value of the expression in the second field is the size of the wavetable. Many wavetable generators also allow the value –1 in this field to signify dynamic calculation of the wavetable size. If the size is not –1, and is also not strictly greater than zero, then the syntax of the generator call is illegal. In each case, the **size** parameter shall be rounded to the nearest integer before evaluating the semantics as described below.

The subsequent expressions are the required and optional parameters to the generator. For ease of exposition, each of these parameter fields will be given a name in the description of the generators, but there is no normative significance to these names. Parameter fields enclosed in brackets are optional and may or may not occur in a table definition using that generator.

Each wavetable, as well as a block of data, has four parameters associated with it: the sampling rate loop start, loop end, and base frequency. For all wavetable generators except **sample**, these parameters shall be set to zero initially.

5.10.2 Sample

```
t1 table(sample, size, which[, skip])
```

The **sample** core wavetable generator allows the inclusion of audio samples (or other blocks of data) in the bitstream and subsequent access in the orchestra.

If **size** is –1, then the size of the table shall be the length of the audio sample. If **size** is given, and larger than the length of the audio sample, then the audio sample shall be zero-padded at the end to length **size**. If **size** is given, and smaller than the length of the audio sample, only the first **size** samples shall be used.

The **which** field identifies a sample. It is either a symbol, in which case the generator refers to a sample in the bitstream, by symbol number; or a number, in which case the generator refers to a sample stored as an **AudioBuffer** in the BIFS scene graph (ISO/IEC 14496-1 subclause 9.4.2.4).

In the case where the generator refers to a sample in the bitstream, for compliant bitstream implementations, the sample data is simply a stream of raw floating-point values. The most recent sample block of data with the given name (see subclause 5.5.2) shall be placed in the wavetable. If the bitstream sample data block contains sampling rate, loop start, loop end, and/or base frequency values, these parameters of the wavetable shall be set accordingly. If the sampling rate is not provided, it shall be set to the orchestra sampling rate by default. Any other parameters not so provided shall be set to 0.

In the case where the generator refers to a sample stored as an **AudioBuffer**, any audio coder described in ISO/IEC 14496-3 may be used to compress samples. The **children** fields of the **AudioSource** node responsible for instantiation of this orchestra refer to **AudioBuffer** nodes in this case. Each **AudioBuffer** contains, after buffering as described in ISO/IEC 14496-1 subclause 9.2.2.13, several channels of audio data. If the first child has n_0 channels, the second n_1 channels, and so forth up to child *k*-1, then this **AudioSource** node has $K = n_0 + n_1 + ... + n_{k-1}$ channels in all, and **which** shall be a value between 0 and *K-1*. Channel **which** (where **which** is rounded to the nearest integer if necessary), numbering in order across children and their channels, shall be placed in the wavetable. The sampling rate of the wavetable shall be set to the sampling rate of the **AudioBuffer** node from which channel **which** is taken. The loop start, loop end, and base frequency values shall be set to 0.

If the selected **AudioBuffer** node is not finished capturing data when the generator is executed (that is, the generator is executed less than **length** seconds after the **length** field of the **AudioBuffer** is set or changed), then the bitstream is in error. That is, this form of this generator shall only be used in cases where there is time allotted in the bitstream for the other decoders to produce samples (in real-time) before the generator executes. This is likely done by including the table generator in a score line scheduled to execute after the Composition Time (see ISO/IEC 14496-1 subclause 7.3.5) of the last audio Access Unit needed in the **AudioBuffer** node.

For standalone systems such as authoring tools, implementers are encouraged to provide access to other audio file formats and disk file access using this field (for example, to allow a filename as a string constant here). However, the only normative behaviours are those described in this subclause.

If **skip** is provided and is a positive value, it is rounded to the nearest integer, and the data placed in the wavetable begins with sample **skip**+1 of the bitstream or **AudioBuffer** sample data.

5.10.3 Data

```
t1 table(data, size, p1, p2, p3, ...)
```

The **data** core wavetable generator allows the orchestra to place data values directly into a wavetable.

If **size** is –1, then the size of the table shall be the number of data values specified. If **size** is given, and larger than the number of data values, then the wavetable shall be zero-padded at the end to length **size**. If **size** is given, and smaller than the number of data values, then only the first **size** values shall be used.

The **p1, p2, p3 ...** fields are floating-point values that shall be placed in the wavetable

5.10.4 Random

```
t1 table(random, size, dist, p1[, p2])
```

The **random** core wavetable generator fills a wavetable with pseudo-random numbers according to a given distribution. For all pseudo-random number generation algorithms, they shall be reseeded upon orchestra start-up such that each execution of an orchestra containing these instructions generates different numbers.

If **size** is –1, the generator is illegal and a run-time error generated. If the **size** field is a positive value, then this shall be the length of the table, and this many independent random numbers shall be computed to place in the table.

The **dist** field specifies which random distribution to use, and the meanings of the **p1** and **p2** fields vary accordingly.

If **dist** is 1, then a uniform distribution is used. Pseudo-random numbers are computed such that all floating-point values between **p1** and **p2** inclusive have equal probability of being chosen for any sample.

If **dist** is 2, then a linearly ramped distribution is used. Pseudo-random numbers are computed such that the probability distribution function of choosing *x* for any sample is given by

p(x) = 0 if *x* £ **p1** or *x* > **p2**, or
abs(2 / (**p2** – **p1)** x [(*x* **–** **p1**) / (**p2** – **p1**)]) otherwise.

A run-time error is generated if **dist** is 2 and **p1 = p2**.

If **dist** is 3, then an exponential distribution is used. Pseudo-random numbers are computed such that the probability distribution function of choosing *x* for any sample is

p(x) = 0 if $x \le 0$, or
k exp(-*kx*), where *k* = 1 / **p1**, otherwise.

If **dist** is 3, then **p2** is not used and is ignored if it is provided.

If **dist** is 4, then a Gaussian distribution is used. Pseudo-random numbers are computed such that the probability distribution function of choosing x for any sample is

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2p \times \mathbf{var}}},$$

that is, *p(x)* **~** N(**p1**, **p2**) where **p1** is the mean and **p2** the variance of a normal distribution.

If **dist** is 4, then **p2** shall be strictly greater than 0, otherwise a run-time error is generated.

If **dist** is 5, then a Poisson process is modelled, where the mean number of samples between 1's is given by an exponential distribution with mean **p1**. A pseudo-random value is computed according to *p*(*x*) as given for **dist =** 3 (the exponential distribution), above. This value is rounded to the nearest integer *y*. The first *y* values of the table (elements 0 through *y*-1) are set to 0, and the next value (element *y*) to 1. Another pseudo-random value is computed as if **dist** =3, and rounded to the nearest integer *z*. The next *z* values (elements *y* + 1 through *y* + *z* in the table) are set to 0, and the next value (element *y* + *z* + 1) to 1. This process is repeated until the table is full through element **size**. The resulting table has length **size** regardless of the values generated in the pseudo-random process; the last element may be a zero or 1.

If **dist** is 5, then **p2** is not used and is ignored if provided.

If **dist** is less than 0 or greater than 5, a run-time error is generated.

5.10.5 Step

```
t1 table(step, size, x1, y1, x2, y2, ...)
```

The **step** core wavetable generator allows arbitrary step functions to be placed in a wavetable. The step function is computed from pairs of (**x, y**) values.

If **size** is –1, then the size of the wavetable shall be the size of the largest x-value parameter. If **size** is larger than the largest x-value parameter, then the wavetable shall be padded with 0 values at the end to size **size**. If **size** is smaller than the largest x-value provided, then only the first **size** values shall be computed and used.

It is a run-time error if:

- **x1** is not 0,
- the x-values are not a non-decreasing sequence, or
- there are an even number of parameters, not counting the **size** parameter.

For the **step** generator, sample values 0 through **x2-**1 shall be set to **y1**, **x2** through **x3**-1 shall be set to **y2**, **x3** through **x4-1** shall be set to **y3**, and so forth.

5.10.6 Lineseg

```
t1 table(lineseg, size, x1, y1, x2, y2, ...)
```

The **lineseg** core wavetable generator allows arbitrary line-segment functions to be placed in a wavetable. The line segment function is computed from pairs of (**x**, **y**) values.

If **size** is –1, then the size of the wavetable shall be the size of the largest x-value parameter. If **size** is larger than the largest x-value parameter, then the wavetable shall be padded with 0 values at the end to size **size**. If **size** is smaller than the largest x-value provided, then only the first **size** values shall be computed and used.

It is a run-time error if:

- **x1** is not 0,
- the x-values are not a non-decreasing sequence, or
- there are an odd number of parameters, not counting the **size** parameter.

For the **step** generator, sample values for samples
x in the range **x1** through **x2** shall be set to **y1 +** (**y2-y1)**(*x* – **x1**) / (**x2** – **x1**)**,**
x in the range **x2** through **x3** shall be set to **y2 + (y3-y2)**(*x* **– x2**) / (**x3** – **x2**),

and so forth.

If any two successive x-values are equal, a discontinuous function is generated, and no values shall be calculated for the “range” corresponding to those values.

5.10.7 Expseg

```
t1 table(expseg, size, x1, y1, x2, y2, ...)
```

The **expseg** core wavetable generator allows arbitrary exponential-segment functions to be placed in a wavetable. The function is computed from pairs of (**x**, **y**) values.

If **size** is –1, then the size of the wavetable shall be the size of the largest x-value parameter. If **size** is larger than the largest x-value parameter, then the wavetable shall be padded with 0 values at the end to size **size**. If **size** is smaller than the largest x-value provided, then only the first **size** values shall be computed and used.

It is a run-time error if:

- **x1** is not 0,
- the x-values are not a non-decreasing sequence,
- the y-values are not all of the same sign,
- any y-value is equal to 0, or
- there are an odd number of parameters, not counting the **size** parameter.

For the **expseg** generator, sample values for samples
x in the range **x1** through **x2** shall be set to $\mathbf{y1}(\mathbf{y2/y1})^{(x-\mathbf{x1})/(\mathbf{x2}-\mathbf{x1})}$,
x in the range **x2** through **x3** shall be set to $\mathbf{y2}(\mathbf{y3/y2})^{(x-\mathbf{x1})/(\mathbf{x2}-\mathbf{x1})}$,

and so forth.

If any two successive x-values are equal, a discontinuous function is generated, and no values shall be calculated for the "range" corresponding to those values.

5.10.8 Cubicseg

```
t1 table(cubicseg, size, infl1, y1, x1, y2, infl2, y3, x2, y4, infl3, y5, ...)
```

The **cubicseg** core wavetable generator creates a function made up of segments of cubic polynomials. Each segment is specified in terms of endpoints and an inflection point. If, for successive segments, the y-values at the inflection points are between the y-values at the endpoints, then the function is smooth; otherwise, the function is pointy or "comb-like".

If **size** is –1, then the size of the wavetable shall be the size of the largest x-value parameter. If **size** is larger than the largest x-value parameter, then the wavetable shall be padded with 0 values at the end to size **size**. If **size** is smaller than the largest x-value provided, then only the first **size** values shall be computed and used.

It is a run-time error if:

- **infl1** is not 0,
- the x-values are not a non-decreasing sequence,
- any infl-value is not strictly between the two surrounding x-values,
- there are less than two x-values, or
- the sequence of control values does not end with an y-value

For the **cubicseg** generator, sample values for samples numbered:

x in the range **infl1** to **infl2** shall be set to $\mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d}$, where **a**, **b**, **c**, and **d** are the coefficients of a cubic polynomial that passes through (**infl1,y1**), (**x1,y2**), and (**infl2,y3**) and that has 0 derivative at **x1**;

x in the range **infl2** to **infl3** shall be set to $\mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d}$, where **a**, **b**, **c**, and **d** are the coefficients of a cubic polynomial that passes through (**infl2,y3**), (**x2,y4**), and (**infl3,y5**) and that has 0 derivative at **x2**;

and so on.

If, for any segment, such a cubic polynomial does not exist or does not have real values through the segment range, it is a run-time error.

5.10.9 Spline

```
t1 table(spline, size, x1, y1, k2, x2, y2, k3, ...)
```

The **spline** core wavetable generator creates a smoothly varying "spline" function for a set of control points.

If **size** is –1, then the size of the wavetable shall be the size of the largest x-value parameter. If **size** is larger than the largest x-value parameter, then the wavetable shall be padded with 0 values at the end to size **size**. If **size** is smaller than the largest x-value provided, then only the first **size** values shall be computed and used.

It is a run-time error if:

- **x1** is not 0,
- the x-values are not a non-decreasing sequence,
– there are less than two x-values, or
- there are an odd number of parameters, not counting the **size** parameter.

For the **spline** generator, sample values for samples numbered:

x in the range **x1** to **x2** shall be set to $\mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d}$, where **a**, **b**, **c**, and **d** are the coefficients of a cubic polynomial that passes through (**x1,y1**), and (**x2,y2**) and that has derivative 0 at **x1** and derivative **k2** at **x2**;

x in the range **x2** to **x3** shall be set to $\mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d}$, where **a**, **b**, **c**, and **d** are the coefficients of a cubic polynomial that passes through (**x2,y2**), and (**x3,y3**) and that has derivative **k2** at **x2** and derivative **k3** at **x3**;

x in the range **x3** to **x4** shall be set to $\mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d}$, where **a**, **b**, **c**, and **d** are the coefficients of a cubic

polynomial that passes through (**x3,y3**), and (**x4,y4**) and that has derivative **k3** at **x3** and derivative **k4** at **x4**; and so on.

The derivative of the last cubic section shall be zero at **xn**, the last x-point of the sequence.

If, for any segment, such a cubic polynomial does not exist or is not real-valued over the segment range, it is a run-time error.

5.10.10 Polynomial

```
t1 table(polynomial, size, xmin, xmax, a0, a1, a2, ...)
```

The **polynomial** core wavetable generator allows an arbitrary section of an arbitrary polynomial function to be placed in a wavetable. The polynomial function used is $p(x)$ = **a0** + **a1**x + **a2**x^2 + ...; it is evaluated over the range [**xmin**, **xmax**].

It is a run-time error if size is not strictly positive, or if there are not at least 3 parameters, not counting the **size** parameter, or if **xmin = xmax**.

For the **polynomial** generator, the sample value for sample x in the range [0,**size-1**] inclusive shall be set to

a0 + **a1**y **+ a2**y^2 + ..., where y = **xmin** + (**size -** x) / **size** $\times$ (**xmax** – **xmin**).

5.10.11 Window

```
t1 table(window, size, type[, p])
```

The **window** core wavetable generator allows a windowing function to be placed in a table.

It is a run-time error if the **size** parameter is not strictly positive, or if **type** = 5 and the **p** parameter is not included.

The window type is specified by the **type** parameter. This parameter shall be rounded to the nearest integer, and then interpreted as follows:

If **type**=1, a Hamming window shall be used. For sample number x in the range [0, **size** – 1], the value placed in the table shall be

0.54 – 0.46 cos (2π x / (**size** – 1)).

If **type**=2, a Hanning (raised cosine) window shall be used. For sample number x in the range [0, **size** – 1], the value placed in the table shall be

0.50 (1 - cos (2π x / (**size** – 1))).

If **type**=3, a Bartlett (triangular) window shall be used. For sample number x in the range [0, **size** – 1], the value placed in the table shall be

1 – 2 | x - (**size – 1)** / 2 | / (**size** – 1).

If **type**=4, a Gaussian window shall be used. For sample number x in the range [0, **size** – 1], the value placed in the table shall be

$$\frac{e^{-(m-x)^2/(2v)}}{\sqrt{2p \times v}}$$, where m **= size**/2 and v = (**size**/6)$^{1/2}$.

If **type**=5, a Kaiser window shall be used, with parameter **p**. For sample number x in the range [0, **size** – 1], the value placed in the table shall be

$$\frac{I_0\left[p\sqrt{\left(\frac{\text{size}-1}{2}\right)^2-\left(x-\frac{\text{size}-1}{2}\right)^2}\right]}{I_0\left[p\left(\frac{\text{size}-1}{2}\right)\right]},$$

where $I_0(x)$ is the zero-order modified Bessel function of the first kind.

If **type**=6, a boxcar window shall be used. Each sample in the range [0, **size** – 1] shall be given the value 1.

5.10.12 Harm

```
t1 table(harm, size, f1, f2, f3...)
```

The **harm** generator creates one cycle of a composite waveform made up of a weighted sum of zero-phase sinusoids.

It is a run-time error if **size** is not strictly positive.

For each sample *x* in the range [0, **size** –1], the sample shall be assigned the value

f1 sin (2 π *x*/**size**) + **f2** sin (4 π *x*/**size**) + **f3** sin (6 π *x*/**size**) + ...

5.10.13 Harm_phase

```
t1 table(harm_phase, size, f1, ph1, f2, ph2, ...)
```

The **harm_phase** core wavetable generator creates one cycle of a composite waveform made up of a weighted sum of zero-DC sinusoids, each with specified initial phase in radians.

It is a run-time error if **size** is not strictly positive, or if there are an odd number of parameters, not counting the **size** parameter.

For each sample *x* in the range [0, **size** –1], the sample shall be assigned the value

f1 sin (2 π *x*/**size** + **ph1**) + **f2** sin (4 π *x*/**size** + **ph2**) + **f3** sin (6 π *x*/**size** + **ph3**) + ...

5.10.14 Periodic

```
t1 table(periodic, size, p1, f1, ph1, p2, f2, ph2, ...)
```

The **periodic** core wavetable generator creates one cycle of an arbitrary periodic waveform, parameterised as the sum of several sinusoids with arbitrary frequency, magnitude and phase. The phase values (**ph1, ph2, ...**) are specified in radians.

It is a run-time error if **size** is not strictly positive, or if the number of parameters, not counting the **size** parameter, is not evenly divisible by three.

For each sample *x* in the range [0, **size** –1], the sample shall be assigned the value

f1 sin (2 **p1**π *x*/**size** + **ph1**) + **f2** sin (2 **p2** π *x*/**size** + **ph2**) + **f3** sin (2 **p3** π *x*/**size** + **ph3**) + ...

Any of the **p1**, **p2**, **p3**, etc. values may be zero, in which case the corresponding term of the calculation is a DC term; or non-integral, in which case there is a discontinuity at the table wrap point, or negative, which means the corresponding term evolves as a negative phase term. In all cases, the above value expression holds as specified.

5.10.15 Buzz

```
t1 table(buzz, size, nharm, lowharm, rolloff)
```

The **buzz** core wavetable generator creates one cycle of the sum of a series of spectrally-sloped cosine partials (band-limited pulse train). This waveform is a good source for subtractive synthesis.

It is a run-time error if **size** is not strictly positive, and **nharm** is also not strictly positive.

lowharm and **nharm** shall be rounded to the nearest integer before further processing.

If **size** is not strictly positive, then the size of the table is given by the highest harmonic included, such that **size** = 2 (**lowharm** + **nharm**).

If **nharm** is not strictly positive, then the number of harmonics shall be given by the size of the table, such that **nharm** is the greatest integer smaller than **size/2** – **nharm**.

For each sample *x* in the range [0, **size** –1], the sample shall be assigned the value

$$scale * \sum_{f=\mathbf{lowharm}}^{\mathbf{lowharm}+\mathbf{nharm}} \mathbf{rolloff}^{(f-\mathbf{lowharm})} \cos 2\pi f p$$

where *p* is the value *x* / **size** and ***scale*** is the value (1-abs(**rolloff**)) / (1-abs(**rolloff**$^{\mathbf{nharm}}$)).

If **rolloff** is negative, then alternating partials alternate phase direction; if **|r**| < 1, then partials attenuate as they get higher in frequency; otherwise, they stay the same or grow in magnitude; in all cases, the above value expression holds as specified.

5.10.16 Concat

`table t1(concat, size, ft1, ft2, ...)`

The **concat** generator allows several tables to be concatenated together into a new table.

It is a runtime error if no tables are provided as arguments.

If **size** is not strictly positive, the size of the wavetable shall be the sum of the sizes of the parameter wavetables. If **size** is strictly positive, but smaller than the sum of the sizes of the parameter wavetables, then only the first **size** points of the parameter wavetables shall be used. If **size** is larger than the sum of the sizes of the parameter wavetables, then the generated wavetables shall be zero-padded at the end to size **size**.

The values of the wavetable shall be calculated as follows: for each sample *x* in the range [0, s_1-1], where s_1 is the size of the wavetable referenced by **p1**, the sample shall be assigned the same value as sample *x* of **p1**; for each sample *x* in the range [$\mathbf{s_1}$, s_1+s_2-1], where s_2 is the size of the wavetable referenced by **p2**, the sample shall be assigned the same value as sample *x* – s_1 of **p2**; and so on, up to sample **size**.

5.10.17 Empty

`t1 table(empty,size)`

The **empty** generator allocates space and fills it with zeros.

It is a run-time error if **size** is not strictly positive.

For each sample in the range [0,**size**-1], the sample is assigned value 0.

This generator is useful in conjunction with user-defined opcodes that fill up a table with data.

5.11 SASL syntax and semantics

5.11.1 Introduction

This clause describes the syntax and semantics of the score language SASL. SASL allows the simple parametric description of events that use an orchestra to generate sound, including notes, controllers, and dynamic wavetable generation. SASL is simpler than many previously existing score languages; this is intentional, as it enables easier cross-coding of score data from other formats into SASL. Since in many cases, SASL code is automatically generated by authoring tools, it is not a great disadvantage to have relatively simple syntax and few “defaults”.

As with the SAOL description in clause 5.8, this clause describes a textual representation of SASL that is standardised, but stands outside of the bitstream-decoder relationship. It also describes the mapping between the textual representation and the bitstream representation. The exact normative semantics of SASL will be described in

reference to the textual representation, but also apply to the tokenised bitstream representation as created via the normative tokenisation mapping.

All times in the score file (start times and durations) are specified in *score time*, which is measured in *beats*. By default, the score time is equivalent to the absolute time, and thus events with duration of one beat last one second, and an event dispatched two beats of score time after another is dispatched two seconds later by the scheduler. However, this mapping can be changed with the **tempo** command, see below.

Each score line may be prefaced by an optional ***** tag. This tag indicates that the event is a high-priority event as described in subclause 5.7.3.3.7.

NOTE - In streaming performance of Structured Audio bitstreams, some events have no timestamps. This is possible because the streaming mechanism contains intrinsic time, i.e. “right now”. For the textual score format, there is no such intrinsic time, and thus every score line in textual format is required to have a time field. A Structured Audio encoder (see Appendix B) has the option of retaining or removing time fields when tokenizing the orchestra and packing tokenized score events into Access Units, depending on the requirements of the application. For the same reason, the “use if late” bitstream flag is not used in the textual score format.

5.11.2 Syntactic form

<score file> -> <score line> [<score file>]
<score file> -> <score line>

<score line> -> (*****) <instr line> **<newline>**
<score line> -> (*****) <control line> **<newline>**
<score line> -> (*****) <tempo line> **<newline>**
<score line> -> (*****) <table line> **<newline>**
<score line> -> <end line> **<newline>**

<instr line> -> [**<ident> :**] **<number> <ident> <number>** <pflist>

<control line> -> **<number>** [**<ident>**] **control <ident> <number>**

<tempo line> -> **<number> tempo <number>**

<table line> -> **<number> table <ident> <ident>** <pflist>

<end line> -> **<number> end**

<pflist> -> **<number>** [<pflist>]
<pflist> -> **<NULL>**

<number> as given in subclause 5.8.2.3.

<ident> as given in subclause 5.8.2.2.

5.11.3 Instr line

The **instr** line specifies the construction of an instrument instantiation at a given time.

The first identifier, if given, is a label that is used to identify the instantiation for use with further control events.

The first number is the score time of the event. As much precision as desired may be used to specify times; however, instruments are only dispatched as fast as the orchestra control rate, as described in subclause 5.7.3.3. Event times do not have to be received, or present in the score file, in temporal order.

The second identifier (the first required identifier) is the name of the instrument, used to select one instrument from the orchestra described in the SAOL bitstream element. It is a syntax error if there is not an instrument with this name in the orchestra when the orchestra is started.

The second number is the score duration of the instrument instance. When the instrument instantiation is created, a termination event shall be scheduled (see subclause 5.7.3.3) at the time given by the sum of the instantiation time and the note duration. If this field is –1, then the instrument shall have no scheduled duration.

The pflist is the list of parameter fields to be passed to the instrument instance when it is created. If there are more pfields specified in the instrument declaration than elements of this list, the remaining pfields shall be set to 0 upon instantiation. If there are fewer pfields than elements, the extra elements shall be ignored.

5.11.4 Control line

The **control** line specifies a control instruction to be passed to the orchestra, or to a set of running instruments.

The first number is the score time of the control event. When this time arrives in the orchestra, the control event is dispatched according to its particular semantics.

The first identifier, if provided, is a label specifying which instrument instances are to receive the event. If this label is provided, when the control event is dispatched, any active instrument instances that were created by **instr** events with the same label receive the control event. If the label is provided, and there are no such active instrument instances, the control event shall be ignored. If the label is not provided, then the control event references a global variable of the orchestra.

The second identifier (the first required identifier) is the name of a variable that will receive the event. For labelled control lines, the name references a variable in instruments that were created based upon **instr** events with the same label. If there is no such name in a particular instrument instance, then the control event shall be ignored for that instance. For unlabelled lines, the name references a global variable of the orchestra with the same name. If there is no such global variable, then the control event shall be ignored.

The second number is the new value for the control variable. When the control event is dispatched, variables in the orchestra as identified in the preceding paragraph shall have their values set to this value.

5.11.5 Tempo line

The **tempo** line in the score specifies the new tempo for the decoding process. The tempo is specified in beats-per-minute; the default tempo shall be sixty beats per minute, and thus by default the score time is measured in seconds.

The first number in the tempo line is the score time at which the tempo changes. When this time arrives, the tempo event shall be dispatched as described in subclause 5.7.3.3, list item 7.

The second number is the new tempo, specified in beats per minute. Consequently, one beat lasts 60/**tempo** seconds, so that a tempo of 120 beats per minute is twice as fast as the default. When a tempo line is decoded, the time numbers in the score continue progressively, with the increments now in accordance with the new time unit.

5.11.6 Table line

The **table** line in the score specifies the creation or destruction of a wavetable.

The first number in the score line is the score time at which the wavetable is created or destroyed. For creation events, the wavetable shall be created at this time. For destruction events, the wavetable shall not be destroyed before this time.

The first identifier is the name of the wavetable. This name references a wavetable in the global orchestra scope.

The second identifier is either the name of the table generator, or the special name **destroy**. It is a syntax error if this identifier is not the name of one of the core wavetable generators listed in clause 5.10, or the special name **destroy**.

The pfield list is the list of parameters to the particular core wavetable generator. Not every sequence of parameters is legal for every table generator; see the definitions in clause 5.10.

The **sample** core wavetable generator refers to a sound sample (see subclause 5.10.2). Implementations providing textual interfaces are suggested to provide access to commonly-used “soundfile” formats in the first pfield as a string constant. However, this is non-normative; the only normative aspect is as follows. In a bitstream **table** score line object, the **refers_to_sample** bit may be set. If this is the case, then the **sample** token of that score line object shall refer to another bitstream object containing the sample data, and it is this sample data that shall be placed in the wavetable.

When the dispatch time of the table event is received, if the table line references the **destroy** name, then any global wavetable with that name may be destroyed and its memory freed. If the table line specifies creation of a wavetable,

and there is already a global wavetable with the same name, the new wavetable replaces the existing wavetable. That is, the global wavetable with that name may be destroyed and its memory freed.

When a new table is to be created, memory space is allocated for the table and filled with data according to the particular wavetable generator. Any reference to a wavetable with this name (including indirect references through import into a instrument instance) in existing or new instrument instances shall be taken as direction to the new wavetable.

NOTE - According to this paragraph, the wavetables referenced by running instrument instances shall be replaced upon dispatch of a **table** score line using the same name. That is, in the midst of the sound generation process, when the **table** score line is dispatched, any table-reference opcodes in an instrument referencing that name shift reference to the new wavetable.

5.11.7 End line

The **end** line in the score specifies the end of the sound-generation process. The number given is the end time, in score time, for the orchestra. When this time is reached, the orchestra ceases, and all future Composition Buffers based on this Structured Audio decoding process contain only 0 values.

5.12 SAOL/SASL tokenisation

5.12.1 Introduction

This clause describes the normative process of mapping between the SAOL textual format used to describe syntax and semantics in clause 5.8, and the tokenised bitstream representation used in the bitstream definition in clause 5.5. The textual representation stands outside of the bitstream-decoder relationship, and as such is not required to be implemented or used. The only aspect of SAOL decoding that is strictly normative is the process of turning a tokenised bitstream representation into sound as described in clause 5.7. However, it is highly recommended that implementations that allow access to bitstream contents use the textual representation described in clause 5.8 rather than the tokenised representation. It is nearly impossible for a human reader to understand a SAOL program presented in tokenised format.

5.12.2 SAOL tokenisation

To tokenise a textual SAOL orchestra, the following steps shall be performed. First, the orchestra shall be divided into lexical elements, where a lexical element is one of the following:

1. A punctuation mark,
2. A reserved word (see subclause 5.8.9),
3. A standard name (see subclause 5.8.6.8),
4. A core opcode name (see subclause 5.9.3),
5. A core wavetable generator name (see clause 5.10),
6. A symbolic constant (a string, integer, or floating-point constant; see subclause 5.8.2.3), or
7. An identifier (see subclause 5.8.2.2).

Whitespace (see subclause 5.8.2.6) may be used to separate lexical elements as desired; in some cases, it is required in order to lexically disambiguate the orchestra. In neither case shall whitespace be treated as a lexical element of the orchestra. Comments (see subclause 5.8.2.5) may be used in the textual SAOL orchestra but are removed upon lexical analysis; comments are not preserved through a tokenisation/detokenisation sequence.

After lexical analysis, all identifiers in the orchestra shall be numbered with symbol values, so that a single symbol is associated with a particular textual identifier. All identifiers that are textually equivalent (equal under string comparison) shall be associated with the same symbol regardless of their syntactic scope. This association of symbols to identifiers is called the *symbol table.*

Using the lexical analysis and the symbol table, a tokenised representation of the orchestra may be produced. The lexical analysis is scanned in the order it was presented in the textual representation, and for each lexical element:

- If the element is of type (1) – (5) from above, the token value associated in the table in Annex 5.A with that element shall be produced.
- If the element is of type (6) from above, one of the special tokens 0xF1, 0xF2, 0xF3, 0xF4 shall be produced, depending on the type of the symbolic constant, and the succeeding bitstream element shall be the bitstream representation of the value. For integer constants in the range [0,255], either token 0xF1 or token 0xF4 may be produced.
- If the element is of type 7, the special token 0xF0 shall be produced, and the succeeding bitstream element shall be the symbol associated with the identifier in the symbol table.

After the sequence of lexical elements presented in the textual orchestra is tokenised, the special token 0xFF, representing end-of-orchestra, shall be produced.

5.12.3 SASL tokenisation

A SASL score shall be tokenised with respect to a particular SAOL orchestra, since the symbol values must correspond in order for the semantics to be according to the author's intent.

To tokenise a SASL file, the following steps are taken. First, the SASL file is divided into lexical elements, where each element is either an identifier, a reserved word, the name of a core wavetable generator, or a number. After lexical analysis, each identifier shall be associated with the appropriate symbol number from the SAOL orchestra reference. That is, for the associated SAOL orchestra, if there is an identifier in the orchestra equivalent to the identifier in the score, the identifier in the score shall receive the same symbol number that it received in the orchestra. If there is no such identifier in the orchestra, any unused symbol number may be assigned to the identifier in the score.

Using the lexical analysis and the symbol table, a tokenised representation of the orchestra may be produced. Each score line is taken in turn, in the order presented in the textual representation, and used to produce a **score_line** bitstream element, according to the semantics in clause 5.11 and the bitstream syntax for the various score elements, as given in subclause 5.5.2.

5.13 Sample Bank syntax and semantics

5.13.1 Introduction

This section describes the operation of the Sample Bank synthesis method for Object types 2 and 4. In Object type 2, only Sample Bank and MIDI class types shall appear in the bitstream, and this section describes the normative process of generating sound from a Sample Bank bitstream data element and a sequence of MIDI instructions. In Object type 4, Sample Banks are used in the context of a SAOL instrument as described in subclause 5.8.6.7.15, and this section describes the normative process of generating sound and returning it to the SAOL decoding process, depending on the Sample Bank bitstream data element and the particular call to **sasbf**.

The Structured Audio Sample Bank Format is derived from the MIDI Manufacturers Association (MMA) Downloadable Sounds format, which has been adopted as a standard for sample-data exchange by electronic musical instrument and PC audio manufacturers. The MMA DLS-2 standard [**DLS2**] contains provisions which, through reference, constitute provisions of this part of this standard. Subsequent amendments or revision to this publication do not apply, but parties are encouraged to investigate the possibility of applying the most recent editions of the referenced document. In particular, though it is assumed that typically an MPEG encoder will strip all unnecessary chunks from valid DLS files, DLS "chunks" that are not defined in the DLS Level 2 text shall be syntactically valid within an MPEG Sample Bank stream, though it is acceptable to ignore them and not use them for synthesis if not specifically defined in the DLS-2 standard.

5.13.2 Elements of bitstream

The **SASBF** bitstream element is a block of data defined by the MIDI DLS file structure [**DLS**]. This block of data is opaque to the MPEG-4 bitstream parser; among other reasons for this opaqueness, it contains values that are byte-swapped (big-endian) compared to the rest of the MPEG-4 bitstream.

5.13.3 Decoding process

5.13.3.1 Object type 2

5.13.3.1.1 Overview

In Object type 2, all synthesis is performed through wavetable-bank synthesis as defined in the MIDI DLS Level 2 specification [**DLS2**]. Control is through the use of the Standard MIDIFile bitstream element and the MIDI command bitstream element.

5.13.3.1.2 Channels, sample format, and sampling rate

For the purposes of attaching a Object type 2 Structured Audio (i.e., SASBF) decoder to an AudioBIFS **AudioSource** node, the resulting audio stream shall have two channels, 32-bit floating point samples, and a 22050 Hz sampling rate. Calculation is not required to occur in stereo 32-bit samples; if the internal representation is otherwise, the result shall be converted to this format after decoding is complete.

5.13.3.1.3 Decoder configuration

In the stream header (decoder configuration element), one or more **sbf** chunks may appear; a standard MIDIFile **midi_file** chunks may also appear. The **sbf** chunks are passed to the SASBF synthesiser, which uses the data there to prepare for synthesis as described in [**DLS2**]. The **midi_file** chunk, if any, is unrolled and organised in time according to its semantics as given in [**MIDI**].

5.13.3.1.4 Runtime decoding

Two types of events may control runtime synthesis in Object type 2: cached MIDI events that were transmitted as a MIDI file in the stream header, and real-time MIDI events that are transmitted over the streaming connection.

A *decoding clock* is maintained to control dispatch of events, but the exact properties of this clock are nonnormative. At each step, the MIDI scheduler shall dispatch any MIDI events that have arrived in the bitstream with Decoding Time Stamp less than the current value of the decoding clock, as well as any MIDI events sequenced in **midi_file** chunks in the stream header whose unrolled time-stamps are less than the current value of the decoding clock.

Interactive manipulations to the **speed** field of the **AudioSource** scene graph node pointing to this decoding process affect the playback speed of cached Standard MIDIFile events, but have no effect on the dispatch of streaming MIDI events. Thus, if might be the case that events that are synchronised between a MIDIFile and a streaming control end up no longer synchronised if the **speed** field is manipulated.

The sound that results shall be the sound described by the synthesis process in [**DLS2**] according to the sample banks in the stream header and the sequence of MIDI bytes dispatched by the scheduler. These sound samples are provided to the **AudioSource** node in the scene graph that references this bitstream as the output of the Object type 2 Structured Audio decoder.

NOTE - For compliant operation, the sound output from the SASBF synthesis process is not immediately turned into audio and played to the listener. Instead, the sound is made available for further processing by the AudioBIFS scene graph. Implementations that make use of DLS-2 hardware synthesisers that produce analogue output shall “recapture” this output and convert back to a digital signal for use in the scene graph. This is necessary because interactive scene-graph manipulations may alter, attenuate, or eliminate the sound produced in this synthesis process before it is finally played to the listener.

5.13.3.2 Object type 4

5.13.3.2.1 Overview

In Object type 4, the SASBF synthesiser is not controlled directly by MIDI data, but dispatched note-by-note in response to commands in SAOL. The **sasbf** statement (subclause 5.8.6.7.15) performs this dispatching function.

5.13.3.2.2 Decoder configuration

In Object type 4 operation, **sbf** data chunks in the bitstream configuration header are passed to the SASBF synthesiser, where they are used to prepare it for real-time synthesis.

5.13.3.2.3 Runtime decoding

In Object type 4 operation, each note of synthesis is performed separately. The note-on command is executed when the i-rate pass of an instrument containing the **sasbf** expression is executed. This instruction contains note, velocity, preset, and bank select values; the synthesis of one note indicated by this preset number and bank is performed for this note number and velocity according to the SASBF instrument. The resulting stereo sound is returned by the **sasbf** expression (see subclause 5.8.6.7.15).

The SASBF decoder shall make use of the MIDI controller and other continuously changing MIDI information for the channel specified. This data is not passed directly into the SASBF synthesiser in the **sasbf** command, but is made available in an implementation-specific manner. See subclause 5.8.6.7.15.

5.14 MIDI semantics

5.14.1 Introduction

This clause describes the normative decoding process for Object type 1 implementations, and the normative mapping from MIDI events in the stream information header and bitstream data into SAOL semantics for Object type 3 and 4 implementations.

The MIDI standards referenced are standardised externally by the MIDI Manufacturers Association. In particular, we reference the Standard MIDI File format, the MIDI protocol, and the General MIDI patch mapping, all standardised in **[MIDI]**. The MIDI terminology used in this clause is defined in that document.

5.14.2 Object type 1 decoding process

Little normative needs be said about the Object type 1 decoding process. The rules given in **[MIDI]** apply as standardised in those documents. As described in clause 5.6, only **midi** and **midi_file** bitstream elements shall occur in a Object type 1 bitstream.

There are no normative aspects to producing sound in Object type 1.

5.14.3 Mapping MIDI events into orchestra control

5.14.3.1 Introduction

For Object types 3 and 4, events coded as MIDI data shall be converted, when they are received in the terminal as part of a Standard MIDI File or MIDI event, into the appropriate scheduler semantics. This subclause lists the various MIDI events and their corresponding semantics in MPEG-4. These semantics apply only to Object types 3 and 4, not to Object types 1 and 2. For the latter, the semantics of MIDI events are exactly as given in [MIDI].

5.14.3.2 MIDI events

5.14.3.2.1 Introduction

This subclause describes the semantics of the various types of events that may arrive in a continuous bitstream as a **MIDI_event** object. The syntax of these objects is standardised externally in **[MIDI]**.

5.14.3.2.2 Extended channel values

An actual MIDI Channel event in **[MIDI]** has a channel number in the range 0…15. There is no direct way in **[MIDI]** to specify a channel number outside this range. Each MIDI input port, output port or track chunk is associated with a distinct stream or collection of MIDI events and a corresponding distinct set of 16 channels (some of which may be unused). MIDI applications commonly use port names, track names or other labels to identify different channel sets.

Extended channel numbers are used in MPEG-4 to avoid the need for such channel set labels. In MPEG-4, the **channel** value of a **MIDI_event** is not limited to the range 0…15. Instead, an extended channel value is generated based on both the original MIDI channel number and a number associated with the port or stream that is the source of the event. Subclause 5.14.3.3.4 describes the mapping used with Standard MIDI Files. Annex 5.F.2 describes a recommended mapping that may be used with events from a live MIDI device.

5.14.3.2.3 NoteOn

```
noteon channel note velocity
```

When a **noteon** event is received with nonzero velocity, the instrument in the orchestra (if any) currently assigned to channel **channel** shall be instantiated with duration –1 and the first two p-fields set to **note** and **velocity**. Each value of **MIDIctrl[]** within the instrument instance is set to the most recent value of a controller change for that controller on channel **channel** or to the default value (see subclause 5.14.3.3.2) if there have been no controller changes for that controller on that channel. The value of **MIDIbend** is set to the most recent value of the MIDI pitch bend. The value of **MIDItouch** is set to the most recent aftertouch value on the channel.

If there is no instrument currently assigned to channel **channel**, there is no action associated with this event.

An instrument instance created in response to a **noteon** message on a particular channel is referred to as being “on” that channel.

noteon messages with velocity 0 shall be treated as **noteoff** messages, see subclause 5.14.3.2.4.

5.14.3.2.4 NoteOff

```
noteoff channel note velocity
```

When a **noteoff** event is received, each instrument instance on channel **channel** that was instantiated with note number **note** is scheduled for termination at the end of the k-cycle; that is, its **released** flag is set, and if the instrument does not call **extend**, it shall be de-instantiated after the current k-cycle of computation.

If **MIDIctrl[64]** on the indicated channel is non-zero, then the execution of the **noteoff** event shall be delayed until **MIDIctrl[64]** on the indicated channel becomes zero. This behaviour maintains whether the value of **MIDIctrl[64]** is set in the bitstream or by assignment to the **MIDIctrl** standard name (see subclause 5.8.6.8.9).

5.14.3.2.5 Control change

```
cc channel controller value
```

When a **cc** or control change event is received, the new value of the specified controller is set to **value**. This value shall be cached so that future instrument instances on the given channel have access to it; also, all currently active instrument instances on the channel **channel** shall have the standard name **MIDIctrl[controller]** updated to **value**.

5.14.3.2.6 Aftertouch

```
touch channel note velocity
```

When a **touch** event is received, the value of the **MIDItouch** variable of each instrument instance on channel **channel** that was instantiated with note number **note** is set to **velocity**.

5.14.3.2.7 Channel aftertouch

```
ctouch channel velocity
```

When a **ctouch** event is received, the value of the **MIDItouch** variable of each instrument instance on channel **channel** is set to **velocity**.

5.14.3.2.8 Program change

```
pchange channel program
```

When a **pchange** event is received, the current instrument receiving events on channel **channel** shall be changed to the instrument with preset number **program** (see subclause 5.8.6.4). Only the instrument with preset number **program** is assigned to the channel. If there is no instrument with this preset number, then future note-on events on the channel, until another program change is received, shall be ignored.

5.14.3.2.9 Bank select

```
bankselect channel bank
```

When a **bankselect** event is received, the next time a **pchange** event is received, the current instrument receiving events on channel **channel** shall be changed to the instrument with preset number **bank** * 128 + **program**. The **bankselect** event has no direct effect by itself; it only changes the meaning of future **pchange** events on the channel.

5.14.3.2.10 Pitch wheel change

```
pwheel channel value
```

When a **pwheel** event is received, the **MIDIbend** value for each instrument instance on channel **channel** shall be set to **value**.

5.14.3.2.11 All notes off

```
notesoff
```

When a **notesoff** event is received, all instrument instances in the orchestra are terminated at the end of the current k-cycle. Instruments may not save themselves from termination by using the **extend** statement in this case.

5.14.3.2.12 Tempo change

```
tempochange value
```

When a **tempochange** event is received, the global orchestra tempo is changed as described in subclause 5.7.3.3.6, list item 7. **value** here indicates a beats/minute value as in the SASL **tempo** score event (subclause 5.11.5); it shall be converted from the native MIDI tempo format (see [MIDI]) to this format.

5.14.3.2.13 MIDI messages not respected

The following MIDI messages have no meaning in MPEG-4 Object types 3 and 4:

Local Control
Omni Mode On/Off
Mono Mode On/Off
Poly Mode On/Off
System Exclusive
Tune Request
Timing Clock
Song Select/Continue/Stop
Song Position
Active Sensing
Reset

5.14.3.3 Standard MIDI Files

5.14.3.3.1 Introduction

MIDI files have data with the same semantics as the MIDI messages described above; however, the timing semantics are more complicated due to the use of multiple tracks and delta-time timestamps.

5.14.3.3.2 Overview of MIDI file processing

To process a **midi_file** stream information element, the following steps shall be taken. First, the entire stream element is parsed and cached. Then, using the sequence instructions and the sequencer model described in **[MIDI]**, the delta-times of the various **midi_file** events are converted into score event times (in beats). During this step, the actual channel numbers of these events are also mapped to extended **channel** values, as described in subclause 5.14.3.3.4. Converting delta-times to score event times requires converting each **midi_file** track chunk into a timelist containing **midi_event** objects and then interleaving the various track timelists.

5.14.3.3.3 Converting MIDI file track chunks

To convert the track chunk into a timelist, first parse the track chunk to generate a series of MIDI events. This requires converting MIDI file delta-times to MIDI event score times relative to the beginning of each track chunk. It is also necessary to map **midi_file** event channel numbers to extended **midi_event channel** values, as described in subclause 5.14.3.3.4. When a Set Tempo **midi_file** meta-event is processed to generate a corresponding tempo change **midi_event**, the tempo value shall be converted from the microseconds-per-quarter-note units used in [MIDI] to the beats-per-minute units used in MPEG-4.

MIDI time-stamps may only appear in the “MIDI Time Code” syntax, not the “SMPTE” syntax, as described in [MIDI]. The SMPTE Offset meta-event and the SMPTE format delta-time object are not supported in Structured Audio Object 3 and Object 4 bitstreams.

As events are converted from MIDI to SAOL semantics, each event shall be registered with the scheduler according to its event time and semantics.

NOTE - MIDI tempo events are not used to calculate the event time of MIDI events in a MIDI file. Rather, they are scheduled as regular events and dispatched according to the tempo semantics in 5.7.3.3.6 list item 7.

5.14.3.3.4 Converting MIDI channels to scheduler channels

Mapping the channels of events in MIDI files to **midi_event channel** numbers is accomplished as follows. Successive track chunks within a **midi_file** are assigned track numbers in ascending monotonic sequence, with an initial track number of 0. The **channel** value for a particular **midi_file** event is given by:

channel = **midi_file** event channel number + (track number * 16).

If there are multiple **midi_file** chunks within the bitstream header, then subsequent **midi_file** elements have tracks numbered sequentially from the end of the first chunk. That is, if the i^{th} **midi_file** element has k_i tracks, $0 < i < n$, then the tracks from the first **midi_file** are numbered $0..k_0$-1, those from the second are numbered $k_0..k_1$-1, and so forth.

EXAMPLE

A **midi_file** containing ten track chunks would be mapped onto a set of 160 **midi_event channel** values. Events in track chunk 0 would map to **channel** values in the range 0…15. Events in track chunk 1 would map to **channel** values in the range 16…31. Events in track chunk 9 would map to **channel** values in the range 144…159.

If some channel numbers within a given track chunk are unused, the corresponding **channel** values are also unused.

5.14.3.4 Default controller values

The following table gives the default values for certain continuous controllers. If a particular controller is not listed here, then its default value shall be zero.

There is no normative significance to these “function names” excepting controller 64; however, content authors who wish to use General MIDI score files with SAOL orchestras are advised to consult **[MIDI]** for the normative meaning of the controllers and controller values within General MIDI bitstreams and MIDIfiles.

Table 5.4 - Default MIDI Controller Values

Controller	Function	Default
1	Mod Wheel	0
5	Portamento Speed	0
7	Volume	100
10	Pan	64

11	Expression	127
64	Sustain Pedal	0
65	Portamento On/Off	0
66	Sostenuto	0
67	Soft Pedal	0
84	Portamento Control	0
Pitch Bend	Pitch Bend	8192

5.15 Input sounds and relationship with AudioBIFS

5.15.1 Introduction

This clause describes the use of SAOL orchestras as the effects-processing functionality in the AudioBIFS (Binary Format for Scene Description) system, described in ISO/IEC 14496-1 subclause 9.2.2.13.3. In ISO/IEC 14496, SAOL is used not only as a sound-synthesis description method, but also as a description method for sound-effects and other post-production algorithms. The BIFS **AudioFX** node (ISO/IEC 14496-1, subclause 9.4.2.7) allows the inclusion of signal-processing algorithms described in SAOL that are applied to the outputs of the sound nodes subsidiary to that node in the scene graph. This functionality fits well into the bus-send methodology in Structured Audio, but requires some additional normative text to exactly describe the process.

5.15.2 Input sources and phaseGroup

Each node in a BIFS scene graph that contains SAOL code is either an **AudioSource** node or an **AudioFX** node. If the former, there are no input sources to the SAOL orchestra, and so the default orchestra global **inchannels** value is 0 (see subclause 5.8.5.2.3). In this case, the special bus **input_bus** may not be sent to an instrument or otherwise used in the orchestra.

If the latter, the child nodes of the **AudioFX** node provide several channels of input sound to the orchestra. These channels of input sound, calculated as described in ISO/IEC 14496-1 subclause 9.4.2.7, are placed on the special bus **input_bus**. From this bus, they may be sent to any instrument(s) desired and the audio data thereby provided shall be treated normally. The number of orchestra input channels---the default value of orchestra global **inchannels**---is the sum of the numbers of channels of sound provided by each of the children.

In any instrument that receives a **send** from the special bus **input_bus**, the value of the **inGroup** standard name (see subclause 5.8.6.8.15) shall be constructed using the **phaseGroup** values of the child nodes in the scene graph, as follows. The **inGroup**[] values, when non-zero, shall have the property that **inGroup**[**i**] **= inGroup**[**j**] when **i** ≠ **j** exactly when **input** channel **i** is output channel **n** of child **c1**, **input** channel **j** is output channel **m** of child **c2**, **c1 = c2**, and **phaseGroup**[**n**] = **phaseGroup**[**m**] within **c1**. (That is, when the two channels come from the same child and are phase-grouped in that child's output).

This rule applies in addition to the usual **inGroup** rules as given in subclause 5.8.6.8.15.

EXAMPLE

Assume that the two child nodes of an **AudioFX** node produce two and three channels of output respectively, and their **phaseGroup** fields are [1,1] and [1,0,1] respectively. That is, in the first child, the two channels form a stereo pair; and in the second, the first and third channels form a stereo pair that has no phase relationships with the second channel.

For the following global orchestra definitions:

```
send(input_bus ; ; a);
route(a, bus2);
send(bus2,input_bus,bus2 ; ; b);
```

Assume that instrument **a** produces two channels of output. Then, a legal value for the **inGroup** name within **a** is [1,1,2,0,2], and a legal value for the **inGroup** name within **b** is [1,1,2,2,3,0,3,4,4]. The value for the **inGroup** name within **a** shall not be [1,1,1,0,1], and the value for the **inGroup** name within **b** shall not be [1,1,2,2,2,2,2,3,3] (among other illegal possibilities).

5.15.3 The AudioFX node

5.15.3.1 Introduction

The **AudioFX** node in AudioBIFS is described in the MPEG-4 Systems document, ISO/IEC 14496-1, subclause 9.4.2.7. It is used therein to download audio effects-processing algorithms within the AudioBIFS toolset. SAOL is the language for description of audio effects-processing algorithms in AudioBIFS. This clause describes the execution of a Structured Audio orchestra for the purpose of processing sounds in the AudioBIFS scene.

The aspects described in this clause are normative, but the behaviour is not "decoding" behaviour. Rather, this section expands the normative processing semantics of the **AudioFX** node description in ISO/IEC 14496-1.

5.15.3.2 AudioFX orchestra parameters

When a SAOL orchestra is instantiated due to an **AudioFX** BIFS node, only an orchestra file (the **orch** field in the node) and, optionally, a SASL score file (the **score** field) are provided. These files correspond to tokenised sequences of orchestra and score data forming legal **orchestra** and **score_file** bitstream elements as described in subclause 5.5.2. Further, a score may not contain new instrument events, but only control parameters for the **send** instruments defined in the orchestra. The set of allowable sampling rates is restricted, see subclause 5.8.5.2.1.

5.15.3.3 AudioFX orchestra instantiation

To instantiate the orchestra for the AudioFX node requires the following steps:

1. Decoding of the **orch** and **score** (if any) elements in the node
2. Parsing and syntax-checking of these elements
3. Instantiation of **send** instruments in the orchestra (as described in subclause 5.7.2).

Each of these **send** instances shall be maintained until it is turned off by the **turnoff** statement, or the node containing the orchestra is deleted from the scene graph. If the **turnoff** statement is used in one of these instruments, it shall be taken as producing zero values for all future time.

5.15.3.4 AudioFX orchestra execution

The run-time synthesis process proceeds according to the rules cited in subclause 5.7.3.3 for a standard SA decoding process, with the following exceptions and additions:

As no access units will be received by an **AudioFX** process, no communication with the systems layer need be maintained for this purpose. The only events used are those that are in the **score** field of the node itself. At each time step, the **AudioFX** orchestra shall request from the systems layer the input audio buffers that correspond to the child nodes. These audio buffers shall be placed on the special bus **input_bus** and then sent to whatever instruments are specified in the global orchestra header.

Also, at each control-rate step, the **params[]** fields of the **AudioFX** node shall be copied into the global **params[]** array of the orchestra. These fields are exposed in the scene graph so that interactive aspects of other parts of the scene graph may be used to control the orchestra. At the end of each control cycle, the **params[]** array values shall be copied back into the corresponding fields of the **AudioFX** node and then routed to other nodes as specified within the scene graph. (It is not possible to give a more semantically meaningful field name than **params** since the purpose of the field may vary greatly from application to application, depending on the needs of the content).

At every point in time, the output of the orchestra becomes the output of the **AudioFX** node.

5.15.3.5 Speed change functionality in the AudioFX node

Speed change functionality for sounds provided from the input sources is supported in the AudioFX node. The SAOL core opcode **fx_speedc** is provided for this purpose; see subclause 5.9.14.4.

5.15.4 Interactive 3-D spatial audio scenes

When an **AudioSource** or **AudioFX** node is the child of a **Sound** node, the spatial location, direction, and propagation pattern of the sound subtree represented at the position of the **Sound** node, and the spatial location and direction of

the listener, are provided to the SAOL code in the node. In this way, subtle spatial effects such as source directivity modelling may be written in SAOL.

The standard names **position**, **direction**, **listenerPosition, listenerDirection, minFront, maxFront, minBack,** and **maxBack** (see subclauses 5.8.6.8.18-5.8.6.8.25) are used for this purpose.

It is not recommended that content providing 3-D spatial audio in the context of audio-visual virtual reality applications in BIFS use the **spatialize** statement within SAOL to provide this functionality. In most terminals, the scene-composition 3-D audio functionality will be able to use more information about the interaction process to provide the best-quality audio rendering. In particular, spatial positioning and source directivity are implemented at the end terminal with a sophistication suitable for the terminal itself (see ISO/IEC 14496-1, **Sound** node specification, subclause 9.4.2.82). Content authors can use SAOL and the **AudioFX** node to create enhanced spatial effects that include reverberation, environmental attributes and complex attenuation functions, and then let the terminal-level spatial audio presentation be used to interface with the available rendering method for the terminal. The **spatialize** statement in SAOL is provided for the creation of non-interactive spatial audio effects in musical compositions, so that composers may tightly integrate the spatial presentation with other aspects of the musical material.

Annex 5.A
(normative)

Coding tables

5.A.1 Introduction

This Annex contains the bitstream token table as referenced in clause 5.5 and clause 5.12. Certain tokens are indicated as **(reserved),** which means they are not currently used in the bitstream, but may be used in future versions of the standard. Tokens 0xF5 through 0xFF may be used by implementers for implementation-dependent purposes.

5.A.2 Bitstream token table

Token	Text
0x00	**(reserved)**
0x01	aopcode
0x02	asig
0x03	else
0x04	exports
0x05	extend
0x06	global
0x07	if
0x08	imports
0x09	inchannels
0x0A	instr
0x0B	iopcode
0x0C	ivar
0x0D	kopcode
0x0E	krate
0x0F	ksig
0x10	map
0x11	oparray
0x12	opcode
0x13	outbus
0x14	outchannels
0x15	output
0x16	return
0x17	route
0x18	send
0x19	sequence
0x1A	sasbf
0x1B	spatialize
0x1C	srate
0x1D	table
0x1E	tablemap
0x1F	template
0x20	turnoff
0x21	while
0x22	with
0x23	xsig
0x24	interp
0x25	preset
0x26-0x2F	**(reserved)**
0x30	k_rate
0x31	s_rate
0x32	inchan
0x33	outchan
0x34	time
0x35	dur
0x36	MIDIctrl
0x37	MIDItouch
0x38	MIDIbend
0x39	input
0x3A	inGroup
0x3B	released
0x3C	cpuload
0x3D	position
0x3E	direction
0x3F	listenerPosition
0x40	listenerDirection
0x41	minFront
0x42	minBack
0x43	maxFront
0x44	maxBack
0x45	params
0x46	itime
0x47	**(reserved)**
0x48	channel
0x49	input_bus
0x4A	output_bus
0x4B-0x4F	**(reserved)**
0x50	&&
0x51	||
0x52	>=
0x53	<=
0x54	!=
0x55	==
0x56	-
0x57	*
0x58	/
0x59	+
0x5A	>

0x5B	<
0x5C	?
0x5D	:
0x5E	(
0x5F	)
0x60	{
0x61	}
0x62	[
0x63	]
0x64	;
0x65	,
0x66	=
0x67	!
0x68-0x6E	**(reserved)**
0x6F	sample
0x70	data
0x71	random
0x72	step
0x73	lineseg
0x74	expseg
0x75	cubicseg
0x76	polynomial
0x77	spline
0x78	window
0x79	harm
0x7A	harm_phase
0x7B	periodic
0x7C	buzz
0x7D	concat
0x7E	empty
0x7F	destroy
0x80	int
0x81	frac
0x82	dbamp
0x83	ampdb
0x84	abs
0x85	exp
0x86	log
0x87	sqrt
0x88	sin
0x89	cos
0x8A	atan
0x8B	pow
0x8C	log10
0x8D	asin
0x8E	acos
0x8F	floor
0x90	ceil
0x91	min
0x92	max
0x93	pchoct
0x94	octpch
0x95	cpspch
0x96	pchcps
0x97	cpsoct
0x98	octcps
0x99	pchmidi
0x9A	midipch
0x9B	octmidi
0x9C	midioct
0x9D	cpsmidi
0x9E	midicps
0x9F	sgn
0xA0	ftlen
0xA1	ftloop
0xA2	ftloopend
0xA3	ftsetloop
0xA4	ftsetend
0xA5	ftbasecps
0xA6	ftsetbase
0xA7	tableread
0xA8	tablewrite
0xA9	oscil
0xAA	loscil
0xAB	doscil
0xAC	koscil
0xAD	kline
0xAE	aline
0xAF	sblock
0xB0	kexpon
0xB1	aexpon
0xB2	kphasor
0xB3	aphasor
0xB4	pluck
0xB5	buzz
0xB6	grain
0xB7	irand
0xB8	krand
0xB9	arand
0xBA	ilinrand
0xBB	klinrand
0xBC	alinrand
0xBD	iexprand
0xBE	kexprand
0xBF	aexprand
0xC0	kpoissonrand
0xC1	apoissonrand
0xC2	igaussrand
0xC3	kgaussrand
0xC4	agaussrand
0xC5	port
0xC6	hipass
0xC7	lopass
0xC8	bandpass
0xC9	bandstop
0xCA	fir
0xCB	iir
0xCC	firt
0xCD	iirt
0xCE	biquad
0xCF	fft
0xD0	ifft

0xD1	rms
0xD2	gain
0xD3	balance
0xD4	decimate
0xD5	upsamp
0xD6	downsamp
0xD7	samphold
0xD8	delay
0xD9	delay1
0xDA	fracdelay
0xDB	comb
0xDC	allpass
0xDD	chorus
0xDE	flange
0xDF	reverb
0xE0	compressor
0xE1	gettune
0xE2	settune
0xE3	ftsr
0xE4	ftsetsr
0xE5	gettempo
0xE6	settempo
0xE7	fx_speedc
0xE8	speedt
0xE9-0xEF	**(reserved)**
0xF0	<symbol>
0xF1	<number>
0xF2	<integer>
0xF3	<string>
0xF4	<byte>
0xF5-0xFF	**(free)**
0xFF	<EOO>

Annex 5.B
(informative)

Encoding

5.B.1. Introduction

This Annex, provided for informative purposes only, provides guidelines for building a typical Structured Audio encoder. Unlike for the natural audio coders described in Sections 2, 3 and 4, at the time of the completion of ISO/IEC 14496, there is no known technique for generally and automatically encoding Structured Audio bitstreams from acoustic data. The computational methods that would be required to accomplish this—known variously as “computational auditory scene analysis”, “automatic polyphonic transcription”, and “musical acoustic source separation”—are still in a research stage and not likely to be generally available for several years.

Thus, the creation of bitstreams complying to this clause is a process that requires human intervention and assistance. This Annex describes the functions of possible tools for creating such bitstreams; the techniques described here are for informative purposes only, and are not required for compliance to ISO/IEC 14496-3.

5.B.2. Basic encoding

5.B.2.1. Introduction

This clause describes the operation of a *basic encoder* of the sort provided with IS0/IEC 14496-5. A basic Structured Audio encoder takes as input the component units of a bitstream conforming to the description in clause 5.5 (such as orchestra files and sound samples), and converts them into in a legal bitstream representation. It is outside the scope of this Annex to discuss the origin of the component units themselves; perhaps they have been created by hand or with the use of other general-purpose computer tools.

For the purposes of this discussion, it is assumed that the component units are in the following formats: SAOL and SASL programs are in their respective textual formats as described in clause 5.8 and clause 5.11 respectively; sound samples are individually stored in computer sound-file format such as AIFF or WAVE; MIDI data is stored as a Standard MIDI File; and SASBF banks are stored as binary data files.

The steps required in bitstream creation are as follows: tokenisation of the SAOL and SASL programs, disassembly of the sound samples, assembly of the decoder configuration information, and (optionally) reorganisation of the score and MIDI events into streaming data. Each of these steps is described in the following sections.

5.B.2.2. Tokenisation of SAOL data

The tokenisation of SAOL data is conducted as described in subclause 5.12.2. This process converts the SAOL program given in the textual format into a binary block of data. During this process, a *symbol table* may be constructed if desired by enumerating the names of the instruments, user-defined opcodes, wavetables, and signal variables in the orchestra, and associating each with a numeric value. This table may be incorporated in the decoder configuration header of the bitstream as described in subclause 5.5.2; it has no normative significance in decoding, but allows human-readable SAOL and SASL programs in the textual format to be recovered from the bitstream.

5.B.2.3. Tokenisation of SASL data

The tokenisation of SASL data is conducted as described in subclause 5.12.3. This process converts the SASL program given in the textual format into a binary block of data. Any symbols used in the SASL score may be incorporated into the symbol table if one was constructed in the process described in subclause 5.B.2.2; however, at most one symbol table may be used in the orchestra.

5.B.2.4. Disassembly of sound samples

The sound samples stored as computer sound files are disassembled into blocks of sample values. It is not permissible to include sound samples with formatted data (such as an AIFF or WAVE file) directly in the Structured Audio bitstream. The length (in samples), sampling rate (in Hz) if available, base frequency (in Hz) if needed, and loop

start and end points (in sample number) if needed are accounted from the formatted information in the computer sound file. The sound samples are converted from whatever format they were stored in the computer sound file into either 16-bit signed integer values (that is, the values are scaled into the range [-32768, 32767]) or into 32-bit signed floating point values. Either format may be used for a sample in the Structured Audio bitstream; the first is more efficient and the second is more precise.

NOTE - If very long sound samples are to be used, and one or more natural sound decoders (that is, the functionality described in clauses 2, 3, and 4) are present in the terminal, the natural sound decoders may be used to compress sound samples as described in subclause 5.10.2 of this document and subclause 9.4.2.4 of ISO/IEC 14496-1. In this case, the sound sample(s) is (are) not contained in the Structured Audio bitstream, but in one or more natural audio bitstreams that are associated with the Structured Audio bitstream through use of the **AudioBuffer** AudioBIFS node as described in subclause 9.4.2.4 of ISO/IEC 14496-1. This process can result in more efficient transmission for Structured Audio bitstreams containing long samples.

5.B.2.5 Assembly of decoder configuration information

The decoder configuration header is constructed according to the format given in clause 5.5 from a tokenised SAOL orchestra, and possibly one or more tokenised SASL scores, sound samples, MIDI files, and SASBF banks. The blocks may be in any desired order, and are indexed by the **more_data** and **chunk_type** bit fields. The **high_priority** bit of each score event in the header may be set for each score line in the score that contained the ***** tag, or for any set of important events desired.

5.B.2.6 Assembly of streaming bitstream

In the Structured Audio bitstream format, streaming data in the form of access units is not strictly required; all of the information required for decoding may be present in the decoder configuration header. Including streaming data in the form of access units may make it easier for general-purpose MPEG-4 tools to reorganise the bitstream data for the purposes of editing or resynchronisation, or to execute random-access control of the bitstream (see Annex 5.C).

Sound samples, score events, and MIDI commands may all be included in the streaming-data part of the Structured Audio bitstream. A sound sample is included simply by packaging the sound data, after it has been disassembled from the computer sound file format, into an access unit as specified in subclause 5.5.2. A score event may be included with or without a timestamp as discussed in subclause 5.5.2. If it is included with a timestamp, it is subject to internal orchestra tempo control as discussed in subclause 5.7.3.3.6, but is difficult to reschedule at the Access Unit level; if it is included without timestamps, it is easier to reschedule at the Access Unit level, and is not subject to score-based control of the tempo. If there is no explicit timestamp, the event timing is controlled by the synchronisation information in the Access Unit, see subclause 7.2.3 of ISO/IEC 14496-1. In this case, the **use_if_late** tag may be used to indicate whether the event shall be used if it arrives late; see subclause 5.7.3.3.8.

For each score event, the **high_priority** bit may be set if the corresponding line in the score contained the ***** tag, or for any set of important events desired.

MIDI commands are first converted from the Standard MIDI File format to MIDI data representations. To accomplish this, the absolute time of each event in the Standard MIDIFile is computed according to the syntax and semantics in [MIDI]. Then, the events are not included with delta-times, but are placed directly in the Access Unit, so that the synchronisation information in the Access Unit controls the event timing for the MIDI events. The MIDI data in each Access Unit in the bitstream is the same as that that is conveyed in the MIDI protocol in real-time MIDI-based performance; that is, it consists of un-timestamped note on, note off, and controller information.

MIDI events are wrapped in the length indicator as indicated in clause 5.5; the format of streaming MIDI data in MPEG-4 is not bit-for-bit identical (and thus, not as compact) as MIDI data in the strict MIDI protocol as specified in **[MIDI]**, clause 5.2.

NOTE - The streaming data constructed by the process is noted to be “bursty” and highly variable-rate. That is, when a large access unit is conveyed, for example, a sound sample, the effective bitrate is suddenly much higher than when only small access units such as note events are conveyed. The Access-Unit repackaging techniques described in clauses 10 and 11 of ISO/IEC 14496-1 may be used to smooth out the bitrate of the Structured Audio bitstream.

Annex 5.C
(informative)

lex/yacc grammars for SAOL

5.C.1 Introduction

This Annex provides grammars using the widely-available tools ‘lex’ and ‘yacc’ that conform to the SAOL specification in this document. They are provided for informative purposes only; implementers are free to use whichever tools they desire, or no tools, in building an implementation.

The reference software for Structured Audio in ISO/IEC 14496-5 builds the lexer and parser for SAOL out of these grammars by augmenting them with more processing and data-structures.

5.C.2 Lexical grammar for SAOL in lex

```
STRCONST    \"(\\.|[^\\"])*\"
IDENT       [a-zA-Z_][a-zA-Z0-9_]*
INTGR       [0-9]+
NUMBER      [0-9]+(\.[0-9]*)?(e[-+]?[0-9]+)?|-?\.[0-9]*(e-+?[0-9]+)?

%{
void comment(void);
%}

%%

"//"          { comment(); }
"aopcode"     { return(AOPCODE) ; }
"asig"        { return(ASIG) ; }
"else"        { return(ELSE) ; }
"exports"     { return(EXPORTS) ; }
"extend"      { return(EXTEND) ; }
"global"      { return(GLOBAL) ; }
"if"          { return(IF) ; }
"imports"     { return(IMPORTS); }
"inchannels"  { return(INCHANNELS) ; }
"instr"       { return(INSTR); }
"interp"      { return(INTERP); }
"iopcode"     { return(IOPCODE); }
"ivar"        { return(IVAR) ; }
"kopcode"     { return(KOPCODE); }
"krate"       { return(KRATE) ; }
"ksig"        { return(KSIG) ; }
"map"         { return(MAP) ; }
"oparray"     { return(OPARRAY) ; }
"opcode"      { return(OPCODE) ; }
"outbus"      { return(OUTBUS) ; }
"outchannels" { return(OUTCHANNELS) ; }
"output"      { return(OUTPUT) ; }
"preset"      { return(PRESET) ; }
"return"      { return(RETURN) ; }
"route"       { return(ROUTE) ; }
"send"        { return(SEND) ; }
"sequence"    { return(SEQUENCE) ; }
"sasbf"       { return(SASBF) ; }
"spatialize"  { return(SPATIALIZE) ; }
"srate"       { return(SRATE); }
```

```
"table"       { return(TABLE); }
"tablemap"    { return(TABLEMAP); }
"template"    { return(TEMPLATE); }
"turnoff"     { return(TURNOFF); }
"while"       { return(WHILE); }
"with"        { return(WITH); }
"xsig"        { return(XSIG) ; }
"&&"          { return(AND); }
"||"          { return(OR); }
">="          { return(GEQ); }
"<="          { return(LEQ); }
"!="          { return(NEQ); }
"=="          { return(EQEQ); }
"-"           { return(MINUS); }
"*"           { return(STAR); }
"/"           { return(SLASH); }
"+"           { return(PLUS); }
">"           { return(GT); }
"<"           { return(LT); }
"?"           { return(Q); }
":"           { return(COL); }
"("           { return(LP); }
")"           { return(RP); }
"{"           { return(LC); }
"}"           { return(RC); }
"["           { return(LB); }
"]"           { return(RB); }
";"           { return(SEM); }
","           { return(COM); }
"="           { return(EQ); }
"!"           { return(NOT); }

{STRCONST}    { yytext[strlen(yytext)-1] = 0; /* strip quotes */
                yylval = strdup(&yytext[1]);
                return(STRCONST); }
{IDENT}       { yylval = strdup(yytext);
                return(IDENT) ; }
{INTGR}       { yylval = strdup(yytext);
                return(INTGR) ; }
{NUMBER}      { yylval = strdup(yytext);
                return(NUMBER) ; }
[ \t\n\r]     { /* whitespace */ }

.             { printf("Line %d: Unknown character: '%s'\n",
                        yyline,yytext); } /* parse error */

%%

void comment() {
  char c;

  while ((c = input()) != '\n'); /* skip */
  yyline++;
  thisline[0] = 0;
  yycol = 0;
  }
```

5.C.3 Syntactic grammar for SAOL in yacc

```
/*

   This is a grammar for SAOL, written in 'yacc'.

   It has one shift/reduce conflict that arises when
   looking at table definitions.  Within the grammar,
   'table t1;' is allowed as a definition even though
   it is  a prohibited construction (it is flagged in the syntax-
   check).  So there's an ambiguity between

     table t1(...)
     table t1;

  and you don't know with one lookahead where you are when you
  get 'table' if 't1' is the next token.

  This grammar is somewhat less strict than it could be about
  creating parse errors for all syntax errors.  As it is used
  in ISO/IEC 14496-5, many illegal constructions are allowed here,
  but then prohibited in the syntax check.

  The code in ISO/IEC 14496-5 for Structured Audio uses this grammar
  as a basis, and augments it with error productions and construction
  of a parse tree.

*/

%token IDENT INTGR NUMBER STRCONST AOPCODE ELSE EXPORTS EXTEND GLOBAL
%token IF IMPORTS INCHANNELS INTERP
%token INSTR IOPCODE IVAR TABLE KOPCODE KRATE KSIG ASIG MAP
%token OPARRAY OPCODE OUTBUS OUTCHANNELS OUTPUT ROUTE SEND SEQUENCE
%token SRATE TEMPLATE TURNOFF WHILE WITH XSIG AND OR GEQ LEQ
%token NEQ EQEQ MINUS STAR SPATIALIZE SASBF TABLEMAP
%token SLASH PLUS GT LT Q COL LP RP LC RC LB RB SEM COM EQ RETURN NOT
%token ARRAYREF OPCALL IMPEXP VARDECL NOTAG SPECIALOP PRESET

%

%start orcfile
%left Q
%left AND OR
%nonassoc LT GT LEQ GEQ EQEQ NEQ
%left PLUS MINUS
%left STAR SLASH
%right UNOT
%right UMINUS
%token HIGHEST

%%

orcfile       : proclist
              ;

proclist      : proclist instrdecl
              | proclist opcodedecl
              | proclist globaldecl
```

```
              | proclist templatedecl
              | /* null */
              | error
              ;

instrdecl     : INSTR IDENT LP identlist RP miditag LC vardecls block RC
              ;

miditag       : PRESET int_list
              | /* null */
              ;

int_list      : int_list INTGR
              | INTGR
              ;

opcodedecl    : optype IDENT LP paramlist RP LC opvardecls block RC
              ;

globaldecl    : GLOBAL LC globalblock RC
              ;

templatedecl  : TEMPLATE LT identlist GT  /* with preset */
                 PRESET mapblock
                 LP identlist RP
                 MAP LC identlist RC
                 WITH LC mapblock RC LC
                 vardecls block RC
              | TEMPLATE LT identlist GT  /* no preset */
                 LP identlist RP
                 MAP LC identlist RC
                 WITH LC mapblock RC LC
                 vardecls block RC
              ;

mapblock      : mapblock COM LT terminal_list GT
              | LT terminal_list GT
              |
              ;

terminal_list : terminal_list COM terminal
              | terminal
              ;

terminal      : IDENT
              | const
              | STRCONST
              ;

globalblock   : globalblock globaldef
              | /* null */
              ;

globaldef     : rtparam
              | vardecl
              | routedef
              | senddef
              | seqdef
              ;
```

```
rtparam       : SRATE INTGR SEM
              | KRATE INTGR SEM
              | INCHANNELS INTGR SEM
              | OUTCHANNELS INTGR SEM
              | INTERP INTGR SEM
              ;

routedef      : ROUTE LP IDENT COM identlist RP SEM
              ;

senddef       : SEND LP IDENT SEM exprlist SEM identlist RP SEM
              ;

seqdef        : SEQUENCE LP identlist RP SEM
              ;

block         : block statement
              | /* null */
              ;

statement     : lvalue EQ expr SEM
              | expr SEM
              | IF LP expr RP LC block RC
              | IF LP expr RP LC block RC ELSE LC block RC
              | WHILE LP expr RP LC block RC
              | INSTR IDENT LP exprlist RP SEM
              | OUTPUT LP exprlist RP SEM
              | SPATIALIZE LP exprlist RP SEM
              | OUTBUS LP IDENT COM exprlist RP SEM
              | EXTEND LP expr RP SEM
              | TURNOFF SEM
              | RETURN LP exprlist RP SEM
              ;

lvalue        : IDENT
              | IDENT LB expr RB
              ;

identlist     : identlist COM IDENT
              | IDENT
              | /* null */
              ;

paramlist     : paramlist COM paramdecl
              | paramdecl
              |   /* null */
              ;

vardecls      : vardecls vardecl
              | /* null */
              ;

vardecl       : taglist stype namelist SEM
              | stype namelist SEM
              | tabledecl SEM
              | TABLEMAP IDENT LP identlist RP SEM
              ;
```

```
opvardecls    : opvardecls opvardecl
              | /* null */
              ;

opvardecl     : taglist otype namelist SEM
              | otype namelist SEM
              | tabledecl SEM
              ;

paramdecl     : otype name
              ;

namelist      : namelist COM name
              | name
              ;

name          : IDENT
              | IDENT LB INTGR RB
              | IDENT LB INCHANNELS RB
              | IDENT LB OUTCHANNELS RB
              ;

stype         : IVAR
              | KSIG
              | ASIG
              | TABLE
              | OPARRAY
              ;

otype         : XSIG
              | stype
              ;

tabledecl     : TABLE IDENT LP IDENT COM exprstrlist RP
              ;

taglist       : IMPORTS
              | EXPORTS
              | IMPORTS EXPORTS
              | EXPORTS IMPORTS
              ;

optype        : AOPCODE
              | KOPCODE
              | IOPCODE
              | OPCODE
              ;

expr          : IDENT
              | const
              | IDENT LB expr RB
              | SASBF LP exprlist RP
              | IDENT LP exprlist RP
              | IDENT LB expr RB LP exprlist RP
              | expr Q expr COL expr %prec Q
              | expr LEQ expr
              | expr GEQ expr
              | expr NEQ expr
              | expr EQEQ expr
```

```
              | expr GT expr
              | expr LT expr
              | expr AND expr
              | expr OR expr
              | expr PLUS expr
              | expr MINUS expr
              | expr STAR expr
              | expr SLASH expr
              | NOT expr     %prec UNOT
              | MINUS expr
              | LP expr RP
              ;

exprlist      : exprlist COM expr
              | expr
              | /* null */
              ;

/* this is used for table declarations; string constants provide
   above-the-standard file handling for “sample” */

exprstrlist   : exprstrlist COM expr
              | exprstrlist COM STRCONST
              | STRCONST
              | expr
              ;

const         : INTGR
              | NUMBER
              ;
```

Annex 5.D
(informative)

PICOLA Speed change algorithm

5.D.1 Tool description

PICOLA (Pointer Interval Controlled OverLap Add) speed control tool supports speed change functionality for mono-channel signals. The speed control is achieved by replacing a part of the input signal with an overlap-added waveform or by inserting the overlap-added waveform into the input signal.

5.D.2 Speed control process

The block diagram of the speed controller is shown in Figure 5.D.1. The input signal InputSignal, which is an output from the audio source with a given frame length NumInputSample, is stored in the buffer memory. Adjacent waveforms with the same length are extracted in pairs from the memory buffer and the pair with the minimum distortion between the two waveforms is selected. The selected waveforms are overlap-added. The speed control is achieved by replacing a part of the input signal with the overlap-added waveform or by inserting it into the input signal. The speed controller outputs the speed changed signal with a certain fixed length frame calculated as NumInputSample /SpeedControlFactor. Details of the processing are described below.

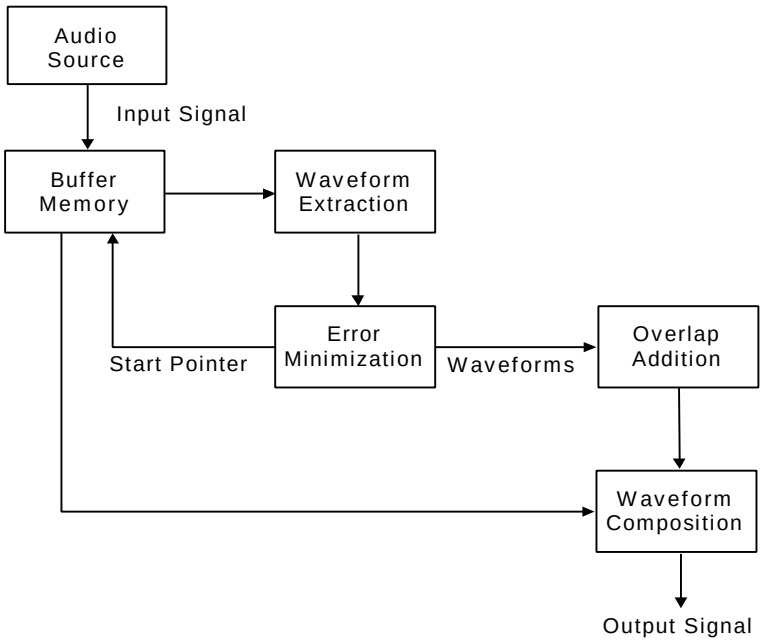


Figure 5.D.1 - Block Diagram of the Speed Controller

5.D.3 Time scale compression (High speed replay)

The compression principle is shown in Figure 5.D.2. P0 is the pointer that indicates the starting sample of the current processing frame in the memory buffer. The processing frame has a length of LF samples and comprises adjacent waveforms each length of LW samples. The average distortion per sample between the first half of the processing frame (waveform A) and the second half (waveform B) is calculated as shown below.

$$D(LW) = \frac{1}{LW}\sum_{n=0}^{LW-1}\{x(n) - y(n)\}^2 \qquad (PMIN \le LW \le PMAX)$$

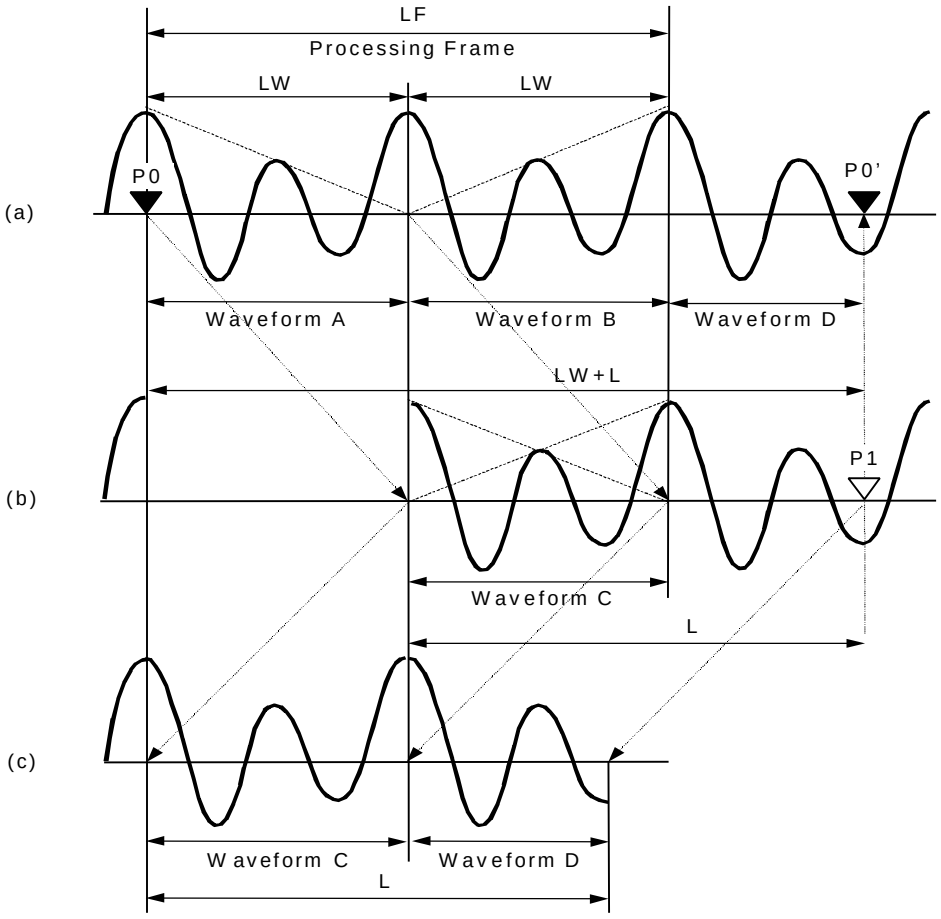
where D(LW) is the average distortion between the two waveforms when the waveform length is LW, $x(n)$ is the waveform A, $y(n)$ is the waveform B, PMIN is the minimum length of the candidate waveform and PMAX is the

maximum length of the candidate waveform. Typically PMIN=32 and PMAX=160 for 8kHz sampling rate, PMIN=80 and PMAX=320 for 16kHz sampling.

The length LW that minimizes the distortion D(LW) is selected, and corresponding waveforms A and B are determined. If the cross-correlation between the selected waveforms A and B is negative, the length LW is set to PMIN-1. After the waveform length LW is determined, the waveform A is windowed by a half triangular window with a descending slope, and the waveform B is windowed by a half triangular window with an ascending slope. The overlap-added waveform C is obtained by linearly adding the windowed waveform A and waveform B. Then, the pointer P0 moves to the point P1. The distance L from the beginning of waveform C to the pointer P1 is given by;

$$L = LW \bullet \frac{1}{\text{SpeedControlFactor} - 1} \qquad (\text{SpeedControlFactor} > 1)$$

L samples from the beginning of waveform C are output as the compressed signal. If L is greater than LW, the original waveform D that follows the waveform B is also output. Therefore the length of the signal is shortened from LW+L samples to L samples. The updated pointer P1 indicates the starting sample P0' of the next processing frame.



(a) Orginal signal; (b) Overlap-added waveform; (c) Compressed signal

Figure 5.D.2 - Principle of Time Scale Compression

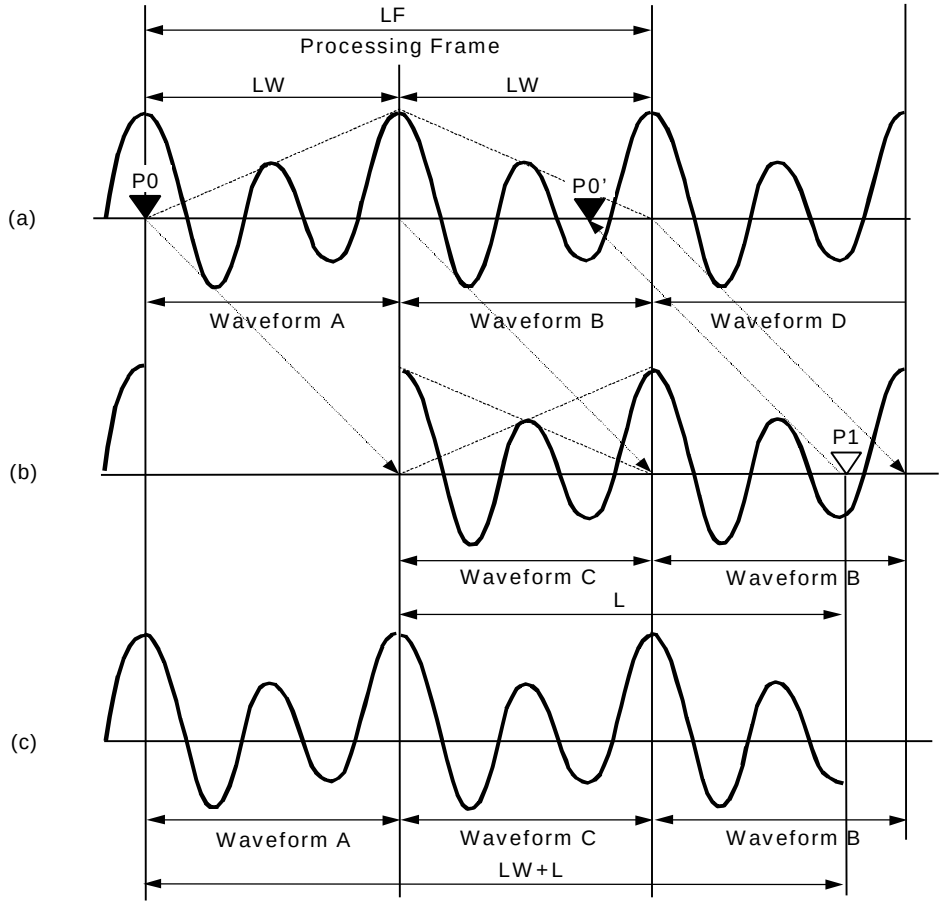
5.D.4 Time scale expansion (Low speed replay)

The expansion principle is shown in Figure 5.D.3. P0 is the pointer that indicates the starting sample of the current processing frame in the memory buffer. The processing frame has a length of LF samples and includes adjacent waveforms each length of LW samples. After the waveform length LW is determined using the same method as described in the time scale compression, the first half of the processing frame (waveform A) is outputted without any

modification. Next, the first half (waveform A) is windowed by a half triangular window with an ascending slope, and the second half (waveform B) is windowed by a half triangular window with a descending slope. The overlap-added waveform C is obtained by linearly adding the windowed waveform A and waveform B. Then, the pointer P0 moves to the point P1. The distance L from the beginning of waveform C to the pointer P1 is given by;

$$L = LW \bullet \frac{SpeedControlFactor}{1 - SpeedControlFactor} \qquad (0.5 \le SpeedControlFactor < 1)$$

L samples from the beginning of waveform C are output as the expanded signal. If L is greater than LW, the original waveform B is repeated as the output. The length of the signal is therefore expanded from L samples to LW+L samples. The updated pointer P1 indicates the starting sample P0' of the next processing frame.



(a) Original signal; (b) Overlap-added waveform; (c) Expanded signal

Figure 5.D.3 - Principle of Time Scale Expansion

Annex 5.E
(informative)

Random access to Structured audio bitstreams

5.E.1 Introduction

This Annex describes practices for constructing Structured Audio bitstreams that allow easy random access to sound content. As discussed in annex 5.E.2, not every sound described with a bitstream conforming to the Structured Audio standard can be easily accessed at random points. Thus, a presentation of guidelines for content authors regarding random access can encourage the development of randomly-accessible bitstreams when this functionality is needed for a particular application. The techniques here are presented for informative purposes only, and are not required for compliance to this part of ISO/IEC 14496.

5.E.2 Difficulties in general-purpose random access

The major problem with enabling random access to every bitstream is that of hidden state. Consider the following orchestra:

```
global {
  srate 24000;
  krate 1000;
  ivar count;
}

instr alternate() {
  imports exports ivar count;
  count = count + 1;
  if ((floor(count/2) * 2 == count && ltime == 0)  {  // divisible by 2?
    instr tone(...);
  }
  // notice no regular output
}

instr tone(...) { ... }  // make some sound
```

This orchestra counts up the number of times the alternate() instrument is instantiated. Every even-numbered instance, it passes on control to the tone() instrument (which presumably makes some noise); on the odd-numbered instances, nothing happens. Thus, with a score containing the following section:

```
.
.
.
25.0 alternate 1.0
26.0 alternate 1.0
27.0 alternate 1.0
.
.
.
```

it is impossible to determine from this segment alone the parity of the first instrument line referring to alternate(); we cannot randomly access this point in the sound except by scanning backward through the score to discover how many previous times the instrument has been instantiated.

The general problem here is with the *hidden state variable* count. The instrument alternate() depends on **count** for its behavior, and this variable has two properties: (1) it has a long “memory” of its past behavior (it maintains state), and (2) it is not controlled by the score (it is hidden). Any orchestra that depends on such hidden state variables is not easily randomly-accessible.

The example here generalizes, not only to instrument dispatch, but to controllers, tempo changes, and even to the development of very long notes containing ambient sound. Suppose, for instance, that there was a long note begun in

the above score at time 2.0 that is supposed to dominate the sound presentation at time 25.0 – in order to discover this, we have to similarly scan backward through the score.

5.E.3 Making Structured Audio bitstreams randomly-accessible

5.E.3.1 Introduction

This Annex describes recommended practices for providing random access points to Structured Audio bitstreams. It covers two topics. First, a set of constructs to avoid is presented; then, a discussion is conducted regarding methods for converting non-randomly-accessible to randomly-accessible bitstreams.

5.E.3.2 Constructs to avoid

In order to make bitstreams randomly-accessible, the following constructs should only be used carefully. Note that there is nothing "wrong" with this constructs in general, and they are useful in applications where random access to sound data is not required. Note also that examples can be constructed for each of these in which the described technique is used and random-accessibility is still maintained.

1. Very long notes. A note that lasts for a long time tends to prohibit random access while it is sustaining, particularly if the sound it makes changes a great deal over the duration.
2. Global variables. These are often used to preserve state between instrument instances or to share data between instruments. When used in this way, they tend to add hidden state to the orchestra.
3. Score-based creation of wavetables. If random access skips over the important point at which a wavetable is created, then it will not be available if some instrument instance needs it.
4. Modification of global wavetables. This is a special case of point (2).
5. Complex score-based control. If score-based controllers are used to modify greatly the behaviour of the orchestra, then random access will produce the wrong result if the control instructions are skipped. This applies to MIDI controllers as well as SASL controllers.

If these constructs are avoided, then in general a bitstream allows random access at the point of each instrument line in the score.

5.E.3.3 Altering bitstreams to make them randomly accessible

5.E.3.3.1 Introduction

For each of the points listed above, it is possible to modify a bitstream that uses the technique to create a bitstream that produces the same sound, but is higher-bandwidth and randomly accessible. This annex provides general guidelines for accomplishing this. Note that it is not possible (due to computational incompleteness) to develop an automated tool that determines whether a particular bitstream is randomly accessible, but it is possible to develop a tool that applies transforms of the following sort automatically.

5.E.3.3.2 Break up very long notes

Very long notes can be broken into several shorter notes. To take a simple example:

```
global {
  krate 100;
}

instr tone(freq) {
  asig a;
  table sine(harm,2048,1);
  a = oscil(sine,freq);
  output(a);
}

tone 10.0 440
```

This orchestra and score play a 440-Hz sine tone for 10 seconds. The same sound is produced by the score

```
0.0 tone 0.99 440
1.0 tone 0.99 440
2.0 tone 0.99 440
3.0 tone 0.99 440
4.0 tone 0.99 440
5.0 tone 0.99 440
6.0 tone 0.99 440
7.0 tone 0.99 440
8.0 tone 0.99 440
9.0 tone 0.99 440
```

which is randomly-accessible once per second rather than only at the beginning. Note that the sound is the same here because the tone is only broken into a new note at the zero-phase point so that the phase remains continuous. (The duration is 0.99 because note instances execute for one extra control period, as specified in subclause 5.7.3.3.6).

For a note that changes an internal state during playback, the internal state must be exposed as a p-field (or control parameter). For example:

```
instr tone(freq) {
  ksig env;
  asig a;
  table sine(harm,2048,1);
  env = kline(0,dur/2,1,dur/2,0);
  a = oscil(sine,freq) * env;
  output(a);
}

0.0 tone 10.0 440
```

In this case, the **kline()** core opcode encapsulates hidden state (the “current value” of the line segment) that must be exposed to break the note into sections. Thus:

```
instr tone(freq, startenv, endenv) {
  ksig env;
  asig a;
  table sine(harm,2048,1);
  env = kline(startenv,dur,endenv);
  a = oscil(sine,freq) * env;
  output(a);
}

0.0 tone 0.99 440 0.0 0.2
1.0 tone 0.99 440 0.2 0.4
2.0 tone 0.99 440 0.4 0.6
3.0 tone 0.99 440 0.6 0.8
4.0 tone 0.99 440 0.8 1.0
5.0 tone 0.99 440 1.0 0.8
6.0 tone 0.99 440 0.8 0.6
7.0 tone 0.99 440 0.6 0.4
8.0 tone 0.99 440 0.4 0.2
9.0 tone 0.99 440 0.2 0.0
```

This orchestra/score specifies the same sound, but is randomly accessible once per second. Application of this technique clearly becomes difficult for very complex instruments.

5.E.3.3.3 Replace global variables with controllers

Consider the initial example from subclause 5.E.2:

```
global {
  srate 24000;
  krate 1000;
  ivar count;
}
```

```
instr alternate() {
  imports exports ivar count;
  count = count + 1;
  if ((floor(count/2) * 2 == count && ltime == 0)  {  // divisible by 2?
    instr tone(...);
  }
  // notice no regular output
}

instr tone(...) { ... }  // make some sound

.
.
.
25.0 alternate 1.0
26.0 alternate 1.0
27.0 alternate 1.0
.
.
.
```

We can make this bitstream randomly accessible by moving the manipulation of the **count** global variable out of the **alternate** instrument and into the score.

```
global {
  srate 24000;
  krate 1000;
  imports ivar count;
}

instr alternate() {
  imports ivar count;

  if ((floor(count/2) * 2 == count && ltime == 0)  {  // divisible by 2?
    instr tone(...);
  }
  // notice no regular output
}

instr tone(...) { ... }  // make some sound

.
.
25.0 control count 14.0
25.0 alternate 1.0
26.0 control count 15.0
26.0 alternate 1.0
27.0 control count 16.0
27.0 alternate 1.0
.
.
.
```

With this change, the bitstream is now randomly accessible (and requires twice the bandwidth).

5.E.3.3.4 Replicate score-based wavetable generators

Consider the following example:

```
global {
  srate 24000;
  krate 1000;
}
```

```
instr tone(freq) {
  imports table shape;

  output(oscil(shape,freq));
}

0.0 table shape harm 2048 1
.
.
.
25.0 tone 1.0 440 // *
26.0 tone 1.0 485
```

At the random-access point (marked *), it is not possible to determine the desired waveshape from local information. To fix this, we simply replicate the table generator throughout the score:

```
0.0 table shape harm 2048 1
.
.
.
25.0 table shape harm 2048 1
25.0 tone 1.0 440 // *
26.0 table shape harm 2048 1
26.0 tone 1.0 485
```

This bitstream is again more expensive, both in bandwidth and in computation, but it allows random access.

Annex 5.F
(informative)

Directly-connected MIDI and microphone control of the orchestra

5.F.1 Introduction

Although it is outside the normative scope of this part of ISO/IEC 14496, some discussion of connecting live MIDI devices and other controllers, and live microphones, directly to a sound generator making use of the Structured Audio tools (especially a SAOL orchestra) is presented and recommended practices developed. This discussion is presented for informative purposes only; the techniques presented here are not required for compliance to this part of ISO/IEC 14496.

The connection of live MIDI control of an orchestra allows a Structured Audio decoder device to be used as a real-time musical instrument in performance or recording situations. The sound quality and flexibility of the Structured Audio tools is an improvement over fixed hardware synthesizers for use in these situations. The connection of live microphones allows a Structured Audio decoder to act as an effects processor, a live or interactive karaoke device, or to enable other kinds of interactive electroacoustic performances.

This Annex presents recommendations for the connection of live MIDI devices and other controllers, and live microphones, to a Structured Audio decoder.

5.F.2 MIDI controller recommended practices

As the MIDI-event bitstream format is very similar to the MIDI control data generated by a live MIDI device (it exactly encapsulates the data bytes generated by such a device in the MPEG-4 Access Unit), no direct modification needs to be made to a Structured Audio decoder to enable control of an orchestra by such a devices. All that need be accomplished is a connection between the MIDI input of the terminal and the scheduler input, so that non-timestamped events from the MIDI input are passed directly to the scheduler. The following practices are recommended in this method:

The MIDI input should not be converted into a legal Structured Audio bitstream, but should generate events directly in the scheduler.

Any note-on events generated with a live MIDI device should not be executed in the order prescribed by the global sequencing rules (see subclause 5.8.5.6). Rather, the note instantiation and first k-cycle of the instrument instance should be executed in the current orchestra pass as soon as possible after they are received by the orchestra. Upon the second k-cycle pass through the instrument instance, the instance begins to be processed according to the global sequencing rules. This practice may cause unpredictable results if it is used in conjunction with instruments containing certain global-variable or bus-routing constructions, and thus such constructions must be used advisedly. If possible, the latency between the time the event is triggered by the live performer and the time at which the first k-cycle of the instrument sound is audible should be no greater than 5 ms.

The live MIDI events should not be subject to orchestra tempo control.

The live MIDI events should otherwise be treated as any other orchestra MIDI event. Streaming performance and live performance should be possible at the same time.

If multiple MIDI devices are connected to the same terminal, the terminal should allow that the channel numbering be managed logically, so that “MIDI channel 1” from Device A is a different channel than “MIDI channel 1” from Device B. For this purpose, it is recommended that each such MIDI device be assigned device numbers in ascending monotonic sequence. The **channel** value for a particular live MIDI event is then given by:

channel = live MIDI event channel + (device number * 16).

The initial device number should be chosen so that **channel** values assigned to live MIDI events do not conflict with **channel** values assigned to **midi_file** events as described in subclause 5.14.3.3.

Although at the time of writing ISO/IEC 14496, the vast majority of live musical control devices generate MIDI data, it is also possible to construct musical control devices that generate SASL events directly. In this case, the recommended

practices are the same as above, except the incoming data is represented in terms of non-timestamped SASL events. The method of constructing devices that generate legal SASL events in real-time is outside the scope of this Annex.

5.F.3 Live microphone recommended practices

As the Structured Audio orchestra already handles acoustic data very well, it is easy to allow the connection of a live microphone. The special bus **input_bus** is defined as in subclause 5.15.2, and the audio data captured by the microphone is placed on this bus, from which it can be sent to other instruments for processing. The following practices are recommended in this method:

A special bus called **input_bus** is defined. This bus has the number of channels specified by the **inchannels** global parameter (see subclause 5.8.5.2.3). If the **inchannels** global parameter is not specified, this bus (and thus the microphone) cannot be used.

At the beginning of each control cycle (i.e., between step 9 and step 10 of subclause 5.7.3.3.6), the microphone input is sampled at the orchestra sampling rate and placed on the special bus **input_bus**. If there are more channels of microphone input than on **input_bus**, only the first channels are used and the rest are discarded; if there are fewer, then the “extra” channels of **input_bus** are set to all 0 values. If there is no microphone connected, then all channels of **input_bus** are set to all 0 values.

The **input_bus** is treated as any other bus in the orchestra.

There is no recommended practice for using a microphone in conjunction with a SAOL orchestra that is used to process effects for an AudioFX node.

Annex 5.G
(informative)

Bibliography

[BIFS] Scheirer, Eric D., and Riitta Väänänen and Jyri Huopaniemi, "AudioBIFS: The MPEG-4 standard for effects processing." *Proc. 1st Cost-G6 Workshop on Digital Audio Effects Processing (DAFX-98)*, Barcelona, 1998.

[BOOK] Scheirer, Eric D., and Youngjik Lee and Jae-Woo Yang, "Synthetic audio and SNHC audio in MPEG-4." In Puri, Atul and Tsuhan Chen (eds.), *Advances in Multimedia: Systems, Standards, and Networks*. New York: Marcel Dekker, in press.

[DRAG] Aho, Alfred V., and Ravi Sethi and Jeffrey Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, Mass: Addison-Wesley, 1984.

[GRAN] Cavaliere, Sergio, and Aldo Piccialli, "Granular synthesis of musical signals", in Curtis Roads, Stephen Travis Pope, Aldo Piccialli, and Giovanni de Poli (eds.), *Musical Signal Processing*. London: Swets & Zeitlinger, 1998, pp. 155-186.

[ICASSP] Scheirer, Eric D., "The MPEG-4 Structured Audio standard", *Proc 1998 IEEE ICASSP*, Seattle, 1998, pp. 3801-3804.

[NETSOUND] Casey, Michael A., and Paris G. Smaragdis, "Netsound", *Proc. 1996 ICMC*, Hong Kong, 1996, p. 143

[SAFX] Scheirer, Eric D., "Structured audio and effects processing in the MPEG-4 multimedia standard", *Multimedia Systems* **7**:1 (Jan. 1999), pp. 11-22.

[SAOL] Scheirer, Eric D., and Barry L. Vercoe, "SAOL: The MPEG-4 Structured Audio Orchestra Language", *Computer Music Journal,* in press.

[SAUD] Vercoe, Barry, and William G. Gardner and Eric D. Scheirer , "Structured Audio: Creation, Transmission, and Rendering of Parametric Sound Descriptions". *Proc. IEEE* **85**:5 (May 1998), pp. 922-940.

[WAVE] Scheirer, Eric D., and Lee Ray, "Algorithmic and wavetable synthesis in the MPEG-4 multimedia standard". Presented at the 105th Convention of the Audio Engineering Society, San Francisco, 1998 (AES reprint #4811).

Alphabetical Index to Section 5 of ISO/IEC 14496-3

Page numbers in **boldface** refer to definitions in clause 5.3.