ISO/JTC 1/SC 29 WG 11 N2203SA

Date: 1998-05-15

ISO/IEC FCD 14496-3 Subpart 5

ISO/JTC 1/SC 29/WG11

Secretariat: Narumi Hirose

Information Technology - Coding of Audiovisual Objects – Low Bitrate Coding of Multimedia Objects

Part 3: Audio

Subpart 5: Structured Audio

Document type: Final Committee Draft
Document:sub-type if applicable
Document:stage (20) Préparation
Document:language E

C:\WINDOWS\Profiles\eds\Desktop\SA FCD 980507.doc ISOSTD Basic Version 1.8 1996-10-30

FCD 14496-3 Subpart 5

MPEG-4 Structured Audio

Editor:
Eric D. Scheirer, MIT Media Laboratory
eds@media.mit.edu
+1 617 253 0112
<http://sound.media.mit.edu/mpeg4>

5.0 Introduction

5.0.1 Overview of subpart

5.0.1.1 Purpose

The Structured Audio decoder allows for the transmission and decoding of synthetic sound effects and music using several techniques. Using Structured Audio, high-quality sound can be created at extremely low bandwidth. Typical synthetic music may be coded in this format at bit rates 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.

5.0.1.2 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 profiles (see Subclause 5.2) 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.

5.0.2 Normative References

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

5.0.3 Glossary of Terms

Absolute time	The time at which sound corresponding to a particular **event** is really created; time in the real-world. Contrast **score time**.
Actual parameter	The **expression** which, upon evaluation, is passed to an **opcode** as a parameter value.
A-cycle	See **audio cycle**.
A-rate	See **audio rate**.
asig	The lexical tag indicating an **a-rate variable**.
Audio cycle	The sequence of processing which computes new values for all **a-rate expressions** in a particular code block.
Audio rate	The **rate type** associated with a **variable**, **expression** or **statement** which may generate new values as often as the **sampling rate**.
Audio sample	A short snippet or clip of digitally represented sound. Typically used in **wavetable synthesis**.
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.
Backus-Naur Format	(**BNF**) A format for describing the syntax of programming languages, used here to specify the **SAOL** and **SASL** syntax. See Subclause 5.0.4.2.
Bank	A set of **samples** used together to define a particular sound or class of sounds with wavetable **synthesis**.
Beat	The unit in which **score time** is measured.
BNF	See **Backus-Naur Format.**
Bus	An area in memory which is used to pass the output of one **instrument** into the input of another.
Context	See **state space**.
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.”
Control cycle	The sequence of processing which computes new values for all **control-rate expressions** in a particular code block.

Control period	The length of time (typically measured in audio samples) corresponding to the **control rate**.
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** which can generate new values as often as the control rate.
Decoding	The process of turning an MPEG-4 Structured Audio bitstream into sound.
Duration	The amount of time between **instantiation** and **termination** of an instrument **instance**.
Encoding	The process of creating a legal MPEG-4 bitstream, whether automatically, by hand, or using special **authoring** tools.
Envelope	A loudness-shaping function applied to a sound, or more generally, any function controlling a parametric aspect of a sound
Event	One **control** instruction.
Expression	A mathematical or functional combination of **variable** values, **symbolic constants**, and **opcode** calls.
Formal parameter	The syntactic element which gives a name to one of the parameters of an **opcode**.
Future wavetable	A **wavetable** which is declared but not defined in the **SAOL orchestra**; its definition must arrive in the bitstream before it is used.
Global block	The section of the **orchestra** which describes **global variables**, **route** and **send** statements, **sequence rules**, and **global parameters**.
Global context	The **state space** used to hold values of **global variables** and **wavetables**.
Global parameters	The **sampling rate**, **control rate**, and number of input and output channels of audio associated with a particular **orchestra**.
Global variable	A **variable** which can be accessed and/or changed by several different **instruments**.
Grammar	A set of rules which describes the set of allowable sequences of **lexical elements** comprising a particular language.
Guard expression	The expression standing at the front of an **if**, **while**, or **else statement** which determines whether or how many times a particular block of code is executed.
I-cycle	See **initialisation cycle.**
Identifier	A sequence of characters in a textual **SAOL** program which denotes a **symbol**.

Informative	Aspects of a standards document which are provided to assist implementors, but are not required to be implemented in order for a particular system to be compliant to the standard.
I-pass	See **initialisation pass.**
I-rate	See **initialisation rate**.
Initialisation cycle	See **initialisation pass.**
Initialisation rate	The **rate type** of **variables**, **expressions**, and **statements** which are set once at **instrument instantation** and then do not change.
Initialisation pass	The sequence of processing which computes new values for each **i-rate expression** in a particular code block.
Instance	See **instrument instantiation**.
Instantiation	The process of creating a new **instrument instantiation** based on an **event** in the **score** or **statement** in the **orchestra**.
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 which produces many different **timbres** over its performance range might be represented using several SAOL instruments.
Instrument instantiation	The **state space** created as the result of executing a note-creation **event** with respect to a **SAOL orchestra**.
ivar	The lexical tag indicating an **i-rate variable**.
K-cycle	See **control cycle**.
K-rate	See **control rate**.
ksig	The lexical tag indicating a **k-rate variable**.
Lexical element	See **token**.
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.
MIDI	The Musical Instrument Digital Interface standards, see **[MIDI]** in Subclause 5.0.2. MIDI is one method for specifying **control** of **synthesis** in MPEG-4 **Structured Audio**.

Natural Sound	A sound created through recording from a real acoustic space. Contrasted with **synthetic sound**.
Normative	Those aspects of a standard which must be implemented in order for a particular system to be compliant to the standard.
Opcode	A parametric signal-processing function which encapsulates a certain functionality so that it may be used by several **instruments**.
Orchestra	The set of sound-generation and sound-processing algorithms included in an MPEG-4 bitstream. Includes **instruments**, **opcodes**, **routing**, and **global parameters**.
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.
Parameter fields	The names given to the parameters to an **instrument**.
P-fields	See **parameter fields**.
Production rule	In **Backus-Naur Form** grammars, a rule which describes how one syntactic element may be expressed in terms of other lexical and syntactic elements.
Rate-mismatch error	The condition that results when the **rate semantics** rules are violated in a particular **SAOL** construction. A type of **syntax error**.
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**.
Rate type	The “speed of execution” associated with a particular **variable**, **expression**, **statement**, or **opcode**.
Route statement	A statement in the **global block** which describes how to place the output of a certain set of **instruments** onto a **bus**.
Run-time error	The condition that results from improper calculations or memory accesses during execution of a **SAOL** orchestra.
SASBF	See **Sample Bank Format**
SAOL	The Structured Audio Orchestra Language, pronounced like the English word “sail.” SAOL is a digital-signal processing language which allows for the description of arbitrary **synthesis** and **control** algorithms as part of the content bitstream.
SAOL orchestra	See **orchestra**.
SASL	The Structured Audio Score Language. SASL is a simple format which allows for powerful and flexible **control** of music and sound **synthesis**.
Sample	See **Audio sample**.

Sample Bank Format	A component format of MPEG-4 **Structured Audio** which allows the description of a set of samples for use in **wavetable synthesis** and processing methods to apply to them.
Scheduler	The component of MPEG-4 **Structured Audio** which 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.
Scope	The code within which access to a particular **variable** name is allowed.
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**.
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**.
Send statement	A statement in the **global block** which describes how to pass a **bus** on to an **effect instrument** for post-processing.
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.
Sequence rules	The set of rules, both default and explicit, given in the **global block** which define in what order to execute **instrument instantiations** during an **orchestra cycle**.
Signal variable	A unit of memory, labelled with a name, which holds intermediate processing results. Each signal variable in MPEG-4 Structured Audio is instantaneously representable by a 32-bit floating point value.
Spatialisation	The process of creating special sounds which a listener perceives as emanating from a particular direction.
State space	A set of variable-value associations which 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.
Statement	"One line" of a **SAOL orchestra**.
Structured audio	Sound-description methods which 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]** in Subclause 5.0.5.
Symbol	A sequence of characters in a **SAOL** program, or a symbol **token** in a MPEG-4 Structured Audio bitstream, which represents a variable name, instrument name, opcode name, table name, bus name, etc.
Symbol table	In an MPEG-4 Structured Audio bitstream, a sequence of data which 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.

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.
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.
Syntax error	The condition that results when a bitstream element does not comply with its governing rules of **syntax**.
Synthesis	The process of creating sound based on algorithmic descriptions.
Synthetic Sound	Sound created through **synthesis**.
Tempo	The scaling parameter which 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.
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.
Termination	The process of destroying an **instrument instantiation** when it is no longer needed.
Timbre	The combined features of a sound which allow a listener to recognise such aspects as the type of instrument, manner of performance, manner of sound generation, etc. Those aspects of sound which distinguish sounds equivalent in pitch and loudness.
Token	A lexical element of a **SAOL orchestra**: a keyword, punctuation mark, **symbol** name, or symbolic constant.
Tokenisation	The process of converting a **orchestra** in textual **SAOL** format into a bitstream representation consisting of a stream of **tokens**.
Variable	See **signal variable**.
Wavetable synthesis	A **synthesis** method in which sound is created by simple manipulation of **audio samples**, such as **looping**, pitch-shifting, **enveloping**, etc.
Width	The number of channels of data which an expression represents.

5.0.4 Description methods

5.0.4.1 Bitstream syntax

The Structured Audio bitstream syntax is described using MSDL, the MPEG-4 Syntactic Description Language. See 14496-1 Subclause XXX.

5.0.4.2 SAOL syntax

The textual SAOL syntax (in Subclause 5.4) is described using extended Backus-Naur format (BNF) notation [see **DRAG** in Subclause 5.0.5]. BNF is a description for context-free grammars of programming languages. Normative BNF rules will be described in the ARIEL font.

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 which 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.

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 5.4.1.

5.0.4.3 SASL Syntax

The SASL syntax is specified using extended BNF grammars, as described in Subclause 5.0.4.2.

5.0.5 Bibliography

[DRAG] Aho, Alfred V., and Ravi Sethi and Jeffrey Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, Mass: Addison-Wesley, 1984.

[ICASSP] Scheirer, Eric, “The MPEG-4 Structured Audio standard”, *Proc 1998 IEEE ICASSP*, Seattle, 1998.

[**NETSOUND]** Casey, Michael, and Paris Smaragdis, “Netsound”, *Proc. 1996 ICMC*, Hong Kong, 1996.

[SAFX] Scheirer, Eric, “Structured audio and effects processing in the MPEG-4 multimedia standard”, *ACM Multimedia Sys. J.*, in press.

[SAOL] Scheirer, Eric, “SAOL: The MPEG-4 Structured Audio Orchestra Language”, *Proc 1998 ICMC*, Ann Arbor, MI, 1998.

[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.

[WAVE] Scheirer, Eric, and Lee Ray, “Algorithmic and wavetable synthesis in the MPEG-4 multimedia standard”. *Proc 105th Conv AES*, San Francisco, 1998.

5.1 Bitstream syntax and semantics

5.1.1 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.

5.1.2 Bitstream syntax

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

class symbol {
  unsigned int(16) sym;         // no more than 65536 symbols/orch + score
}

class sym_name {                // one name in a symbol table
  unsigned int(4) length;       // names up to 16 chars long
  unsigned int(8) name[length];
}

class symtable {                // a whole symbol table
  unsigned int(16) length;      // no more than 65536 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 which require textual input, rather than tokenised input, are permissible in a compliant decoder, in which case the decoder must 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 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 256 chars
    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 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 Subclause 5.8. 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 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. Subclause 5.4 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 Subclause 5.8. The overall sequence of orchestra tokens shall correspond to an <orchestra> production as given in Subclause 5.4.4.

```
/*********************************
      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 256)
}

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 {
  float(32) time;              // the event time
  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 which 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 5.3.3. 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 which 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.

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

/* NB that a midi_file (SMF format) is not just an array
   of MIDI events */

class midi_event {
  // not done yet
}

class midi_file {
  /* Right now it’s just an array of bytes; I’d rather do
```

```
     this that lay out the whole SMF format here */
  unsigned int(20) 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 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. The semantics of MIDI data are given in Subclause 5.9 (for Profile 1 and 2 implementations) and Subclause 5.10 (for Profile 4 implementations).

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

class sample {
  /* note that 'sample' can be used for any big chunk of data
     which 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];
  }
  else {
    int(16) sample_data[length]; // all the data
  }
}
```

A sample chunk includes a block data which 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 5.6.2.

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 -32767 and 32768. 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 5.6.2.

```
/**********************************
         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 Subclause 5.9.

```
const int sbf_chunk_ID = 0x7366626b;                // 'sfbk'
const int INFO_list_ID      = 0x494e464f;           // 'INFO'
const int ifil_chunk_ID     = 0x6966696c;           // 'ifil'
const int isng_chunk_ID      = 0x69736e67;           // ‘isng’
const int INAM_chunk_ID      = 0x494e414d;           // ‘INAM’
const int irom_chunk_ID      = 0x69726f5d;           // ‘irom’
const int iver_chunk_ID      = 0x69766572;           // ‘iver’
const int ICRD_chunk_ID      = 0x49435244;           // ‘ICRD’
const int IENG_chunk_ID      = 0x49454e47;           // ‘IENG’
const int IPRD_chunk_ID      = 0x49505244;           // ‘IPRD’
const int ICOP_chunk_ID      = 0x49434f50;           // ‘ICOP’
const int ICMT_chunk_ID      = 0x49434d54;           // ‘ICMT’
const int ISFT_chunk_ID      = 0x49534654;           // ‘ISFT’
const int sdta_chunk_ID      = 0x73647461;           // ‘sdta’
const int smpl_chunk_ID      = 0x736d706c;           // ‘smpl’
const int pdta_chunk_ID      = 0x70647461;           // ‘pdta’
const int phdr_chunk_ID      = 0x70686472;           // ‘phdr’
const int pbag_chunk_ID      = 0x70626167;           // ‘pbag’
const int pmod_chunk_ID      = 0x706d6f64;           // ‘pmod’
const int pgen_chunk_ID      = 0x7067656e;           // ‘pgen’
const int inst_chunk_ID      = 0x696e7374;           // ‘inst’
const int ibag_chunk_ID      = 0x69626167;           // ‘ibag’
const int imod_chunk_ID      = 0x696d6f64;           // ‘imod’
const int igen_chunk_ID      = 0x6967656e;           // ‘igen’
const int shdr_chunk_ID      = 0x73686472;           // ‘shdr’


aligned(16) class chunk: bit(32) ckID = 0x00000000 {
  unsigned int(32) ckSize; // size of chunk data in bytes
}

class ifil_chunk extends chunk: bit(32) ckID = ifil_chunk_ID {
  unsigned int(16) wMajor;       // file format version number
  unsigned int(16) wMinor;
}

class isng_chunk extends chunk: bit(32) ckID = isng_chunk_ID {
  char(8) isng[ck_hdr.ckSize];          // sound engine identifier
}

class INAM_chunk extends chunk: bit(32) ckID = INAM_chunk_ID {
  char(8) INAM[ck_hdr.ckSize];          // bank name
}

class irom_chunk extends chunk: bit(32) ckID = irom_chunk_ID {
  char(8) irom[ck_hdr.ckSize];          // rom name
}

class iver_chunk extends chunk: bit(32) ckID = iver_chunk_ID {
  unsigned int(16) wMajor;       // rom version
  unsigned int(16) wMinor;
}

class ICRD_chunk extends chunk: bit(32) ckID = ICRD_chunk_ID {
  char(8) ICRD[ck_hdr.ckSize];          // creation date
}

class IENG_chunk extends chunk: bit(32) ckID = IENG_chunk_ID {
  char(8) IENG[ck_hdr.ckSize];          // sound designer name
}

class IPRD_chunk extends chunk: bit(32) ckID = IPRD_chunk_ID {
  char(8) IPRD[ck_hdr.ckSize];          // product name
}

class ICOP_chunk extends chunk: bit(32) ckID = ICOP_chunk_ID {
  char(8) ICOP[ck_hdr.ckSize];          // copyright string
}

class ICMT_chunk extends chunk: bit(32) ckID = ICMT_chunk_ID {
  char(8) ICMT[ck_hdr.ckSize];          // comment string
}

class ISFT_chunk extends chunk: bit(32) ckID = ISFT_chunk_ID {
  char(8) ISFT[ck_hdr.ckSize];          // tool name
}

class INFO_list extends chunk: bit(32) ckID = INFO_list_ID {
  ifil_chunk ifil_ck;
  isng_chunk isng_ck;
  INAM_chunk INAM_ck;
  aligned(16) bit(32)* test0;
  if (test0 == irom_chunk_ID) {
    irom_chunk irom_ck;
  }
  aligned(16) bit(32)* test1;
  if (test1 == iver_chunk_ID) {
    iver_chunk iver_ck;
  }
  aligned(16) bit(32)* test2;
  if (test2 == ICRD_chunk_ID) {
    ICRD_chunk ICRD_ck;
  }
  aligned(16) bit(32)* test3;
  if (test3 == IENG_chunk_ID) {
    IENG_chunk IENG_ck;
  }
  aligned(16) bit(32)* test4;
  if (test4 == IPRD_chunk_ID) {
    IPRD_chunk IPRD_ck;
  }
  aligned(16) bit(32)* test5;
  if (test5 == ICOP_chunk_ID) {
    ICOP_chunk ICOP_ck;
  }
  aligned(16) bit(32)* test6;
  if (test6 == ICMT_chunk_ID) {
    ICMT_chunk ICMT_ck;
  }
  aligned(16) bit(32)* test7;
  if (test7 == ISFT_chunk_ID) {
    ISFT_chunk ISFT_ck;
  }
}

class smpl_chunk extends chunk: bit(32) ckID = smpl_chunk_ID {
  int(16) smpl[ck_hdr.ckSize / 2];      // sample data
}

class sdta_list extends chunk: bit(32) ckID = sdta_chunk_ID {
  smpl_chunk smpl_ck;
}

class phdr_chunk extends chunk: bit(32) ckID = phdr_chunk_ID {
  unsigned int i;
  for (i = 0; i < ck_hdr.ckSize / 38; i++) {
    char(8) achPresetName[20];
    unsigned int(16) wPreset;
    unsigned int(16) wBank;
    unsigned int(16) wPresetBagNdx;
    unsigned int(32) dwLibrary;
    unsigned int(32) dwGenre;
    unsigned int(32) dwMorphology;
  }
}

class bag_chunk extends chunk {
  unsigned int i;
  for (i = 0; i < ck_hdr.ckSize / 4; i++) {
    unsigned int(16) wGenNdx;
    unsigned int(16) wModNdx
  }
}

class pbag_chunk extends bag_chunk: bit(32) ckID = pbag_chunk_ID {
}

class ibag_chunk extends bag_chunk: bit(32) ckID = ibag_chunk_ID {
}

class mod_chunk extends chunk {
  unsigned int i;
  for (i = 0; i < ck_hdr.ckSize / 10; i++) {
    unsigned int(16) sfModSrcOper;
    unsigned int(16) sfModDestOper;
    int(16) modAmount;
    unsigned int(16) sfModAmtSrcOper;
    unsigned int(16) sfModTransOper;
  }
}

class pmod_chunk extends mod_chunk: bit(32) ckID = pmod_chunk_ID {
}

class imod_chunk extends mod_chunk: bit(32) ckID = imod_chunk_ID {
}

class gen_chunk extends chunk {
  unsigned int i;
  for (i = 0; i < ck_hdr.ckSize / 4; i++) {
    unsigned int(16) sfGenOper;
    bit(16) genAmount;
  }
}

class pgen_chunk extends gen_chunk: bit(32) ckID = pgen_chunk_ID {
}

class igen_chunk extends gen_chunk: bit(32) ckID = igen_chunk_ID {
}

class inst_chunk extends chunk {
  unsigned int i;
  for (i = 0; i < ck_hdr.ckSize / 22; i++) {
    char(8) achInstName[20];
    unsigend int(16) wInstBagNdx;
  }
}

class shdr_chunk extends chunk {
  unsigned int i;
  for (i = 0; i < ck_hdr.ckSize / 46; i++) {
    char(8) achSampleName[20];
    unsigned int(32) dwStart;
    unsigned int(32) dwEnd;
    unsigned int(32) dwStartloop;
    unsigned int(32) dwEndloop;
    unsigned int(32) dwSampleRate;
    unsigned int(8) byOriginalPitch;
    int(8) chCorrection;
    unsigned int(16) wSampleLink;
    unsigned int(16) sfSampleType;
  }
}
```

```
class pdta_list extends chunk: bit(32) ckID = pdta_chunk_ID {
  phdr_chunk phdr_ck;                  // preset headers
  pbag_chunk pbag_ck;                  // preset index list
  pmod_chunk pmod_ck;                  // preset modulator list
  pgen_chunk pgen_ck;                  // preset generator list
  inst_chunk inst_ck;                  // instrument names and indices
  ibag_chunk ibag_ck;                  // instrument index list
  imod_chunk imod_ck;                  // instrument modulator list
  igen_chunk igen_ck;                  // instrument generator list
  shdr_chunk shdr_ck;                  // sample headers
}

class sbf extends chunk: bit(32) ckID = sbf_chunk_ID {
  INFO_list INFO_lt;
  sdta_list sdta_lt;
  pdta_list pdta_lt;
}

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

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

  while (more_data) {  // must 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.

```
class SA_access_unit {      // the streaming 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;
   }
}
```

The Structured Audio access unit contains real-time streaming control information to be provided to a running Structured Audio decoding process. It shall not contain new instrument definitions; the orchestra configuration is fixed at decoder startup. It may contain score lines, MIDI events, and new sample data.

5.2 Profiles

There are three profiles standardised for Structured Audio, called Profile1, Profile2, and Profile4. Each of these profiles corresponds to a particular set of application requirements. The default profile is Profile 4; when reference is made to MPEG-4 Structured Audio format without reference to a profile, it shall be understood that the reference is to Profile 4.

Terminals implementing MPEG-4 Systems Audio Composition Profile XXX (see ISO/IEC 14496-1, Subclause XXX) shall also implement Structured Audio Profile 4.

1. MIDI only. In this profile, only the **midi_file** chunk shall occur in the stream information header, and only the **midi_event** event shall occur in the bitstream data. In this profile, the General MIDI patch mappings are used, and the decoding process is described in Subclause 5.9. This profile is used to enable backward-compatibility with existing MIDI content and rendering devices. Implementation-independent sound quality cannot be produced in this profile.

1. Wavetable synthesis. In this profile, 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 profile is used to describe music and sound-effects content in situations 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 5.9.6.

4. Standard profile. All bitstream elements and stream information elements may occur.

The decoding process for Profile 4 is described in Subclause 5.3.

5.3 Decoding process

5.3.1 Introduction

This Subclause describes the decoding process, in which a bitstream conforming to Profile 4 is converted into sound. The decoding process for Profile 1 bitstreams is described in Subclause 5.9, and the decoding process for Profile 2 bitstreams in Subclause 5.9.6.

5.3.2 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 stream information object 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 Subclause 5.4. Whatever preprocessing (i.e., compilation, allocation of static storage, etc.) need 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 parametrised 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 Subclause 5.9, and those events registered with the scheduler.

- Sample bank: The data in the bank shall be stored, and whatever preprocessing necessary to prepare for using the bank for synthesis shall be performed.

- Sample data: The data in the sample shall be stored, and whatever preprocessing 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.

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.

5.3.3 Bitstream data and sound creation

5.3.3.1 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, parse and understand them as the various Structured Audio bitstream data elements, execute the on-going SAOL orchestra, via the scheduler, to produce one Composition Unit of output, and present the systems layer with that Composition Unit.

5.3.3.2 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:

- Score line events shall be registered with the scheduler.
- MIDI events shall be converted into appropriate SAOL events (see Subclause 5.10) and then registered with the scheduler, if they have time stamps, or executed in the next k-cycle, if not.
- Sample data shall be stored, and whatever preprocessing 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.

5.3.3.3 Scheduler semantics

5.3.3.3.1 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.

5.3.3.3.2 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 Subclause 5.6.

5.3.3.3.3 Instrument termination

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

5.3.3.3.4 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 which 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 5.4.6.6. 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 5.4.5.4, 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

5.3.3.3.5 Orchestra startup and configuration

At orchestra startup time, before the first Composition Unit of audio samples is created in the scheduler, the following tasks shall be performed. First, space for any global signal variables (see Subclause 5.4.5.3) 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 Subclause 5.6.

After the global wavetable creation, the orchestra busses are initialised. Each bus’s width is determined, in the order specified by the global sequencing rules (Subclause 5.4.5.6), as the width of the output expression by instruments on that bus. For the purposes of calculating bus widths, any instrument which does not receive any bus data according to the sequence rules shall have an **inchannels** width of 0 (this specification is needed since output widths may depend on the value of **inchannels**).

After busses are created, all instruments which are the targets of **send** statements as described in Subclause 5.4.5.5 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 5.4.5.6. 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 5.3.3.3.6, list item 7.

5.3.3.3.6 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, no further output is produced, and all future requests from the systems layer 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, an instrument instantiation is created for each such instrument event (see Subclause 5.3.3.3.2), and those instantiations are each executed at the i-rate (see Subclause 5.3.3.3.4) 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 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, the global variables or instrument variables within instrument instantiations labelled by that control event shall have their values updated accordingly (see Subclause 5.7.4). 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 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, global wavetables shall be created or destroyed as specified by the table event (see Subclause 5.7.5).

6. If there are any MIDI events whose timestamp is earlier than the current orchestra time, or which have been received without timestamps since the last execution of this rule, they are dispatched according to their semantics in Subclause 5.10.3.

7. If there are any tempo events whose dispatch time is earlier than the current absolute orchestra time, then the global **tempo** standard variable shall be set to the specified value, and all the score times after the current absolute time shall be scaled 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.

   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 Subclause 5.11) has been changed in the last k-cycle, the **tempo** standard variable shall be set to 60 times the value specified in Subclause XXX of FCD ISO 14496-1, and events in the orchestra shall be rescaled as specified in (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 5.4.5.2.2). 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 5.4.5.6, 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 to the conformance rules in Subclause 5.3.4, 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.

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 which 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 5.3.3.3.3). As discussed in Subclause 5.10.3.2.9, 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.

5.3.4 Conformance

With regard to all normative language in this Sub-Part of ISO 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 Subpart.

5.4 SAOL syntax and semantics

5.4.1 Relationship with bitstream syntax

The bitstream syntax description as given in Subclause 5.1 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 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 which may be transmitted in the bitstream are restricted to this set. Any program which 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 which 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 Subclause thus describes a textual representation of SAOL which is standardised, but stands outside of the bitstream-decoder relationship. Subclause 5.8 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 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 Subclause provides bounds on syntactically legal SAOL programs, and by extension, the syntactically legal bitstream sequences which can appear in an **orchestra** bitstream class. That is, there are constructions which 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 which do not conform to the BNF grammar, or contain syntax errors or rate mismatch errors) is unspecified.

Normative language regarding semantics in this Subclause 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.

5.4.2 Lexical elements

5.4.2.1 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 Subclause 5.4, 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.

5.4.2.2 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 which 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 which 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 5.4.9, to one of the standard names listed in Subclause 5.4.6.8, to the name of one of the core opcodes listed in Subclause 5.5.3, or to the name of one of the core wavetable generators listed in Subclause 5.6 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>**.

5.4.2.3 Numbers

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

The integer constant must 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 must occur in SAOL do not allow negative values, there is no provision for negative integers. A string of characters which 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.

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>**.

5.4.2.4 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 implementors 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 implementors.

5.4.2.5 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.

5.4.2.6 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.

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

5.4.3 Variables and values

Each 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 5.3.4; 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 quantization error which is provided with 32-bit values. It is strongly deprecated for bitstreams to contain code which generates widely 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 which 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 5.3.3.3 list item 11. When the terminal presents the sound to a listener, it is likely that further rescaling of the signal will be necessary, as required by the particular digital-analog converter present in the terminal. This scaling is not done by the orchestra, but is outside the scope of the standard and happens after all processing described in Subclause 5.3.3.3 is completed.

5.4.4 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 which can be parametrically controlled using SASL or MIDI score files.

3. Opcode declarations describe sequences of processing instruments which provide encapsulated functionality used by zero or more instruments in the orchestra.

4. Template declarations describe multiple instruments which 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.

5.4.5 Global block

5.4.5.1 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>

There are five 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 which 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.

5.4.5.2 Global parameter

5.4.5.2.1 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 Subclause 5.11). If the sampling rate is not provided, and there are no input audio signals, the default sampling rate shall be 32000 Hz.

5.4.5.2.2 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.

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 which 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.

5.4.5.2.3 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 Subclause 5.11). If there are no input sources provided, the value shall be 0.

5.4.5.2.4 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.

5.4.5.3 Global variable declaration

5.4.5.3.1 Syntactic form

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

Global variable declarations declare variables which 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.

5.4.5.3.2 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 5.4.2.2), 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 5.4.5.2. 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 5.4.8, the standard names listed in Subclause 5.4.6.8, the names of the core opcodes listed in Subclause 5.5, and the names of the core wavetable generators listed in Subclause 5.6 shall not be declared as variable names.

5.4.5.3.3 Wavetable declarations

<table declaration> -> **table <ident> (<ident> ,** <expr> [**,** <expr list>] **) ;**
<expr> as defined in Subclause 5.4.6.7.
<expr list> as defined in Subclause 5.4.6.6.1.

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 Subclause 5.6. 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 Subclause 5.6. Any expression which is i-rate (see Subclause 5.4.6.7.2) 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 must be single-valued, except in the case of the **concat** generator (Subclause 5.6.16), in which case the expressions must 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 5.4.6.5.4. 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 5.4.6.7. 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.

5.4.5.4 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 should 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:

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 which is not the special bus **output_bus** and which does not occur in a **send** statement. See Subclause 5.4.5.5.

It is illegal for a **route** statement to refer to the special bus **input_bus** (see Subclause 5.11.2)**.**

All instruments which 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 5.4.5.5). 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 5.4.6.6.8.

5.4.5.5 Send statement

<send statement> -> **send (<ident> ;** <expr list> **;** <identlist> **);**
<identlist> as defined in Subclause 5.4.5.4
<expr list> as defined in Subclause 5.4.6.6.1

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 which 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 which 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 which 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 $n_0 + n_1 + \dots + n_k$ 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 $n_0 + n_1 + \dots + n_k$, with the last n_k having the value k.

The expression list is a list of zero or more i-rate expressions which are provided to the effect instrument as its parameter fields. Any expression which is i-rate (see Subclause 5.4.6.7.2) 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 no later than immediately after the first instantiation of an instrument which either is routed to a bus which is sent to the effect instrument or refers to the bus in an **outbus** or **sbsynth** statement. 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 5.3.3.3); 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.

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

5.4.5.6 Sequence specification

<sequence specification> -> **sequence (** <identlist> **) ;**
<identlist> as defined in Subclause 5.4.5.4.

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 which 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 which 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 which 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**.

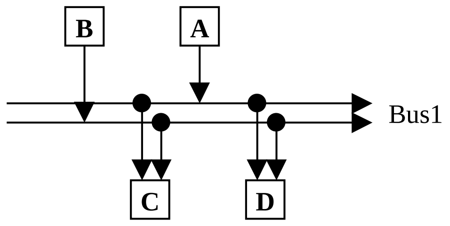
1. The following code fragment

```
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**. 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**.

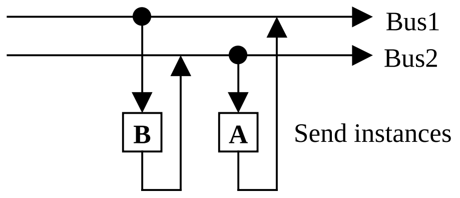
3. The following code fragment

```
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);
```



is legal, and specifies that instantiations of instrument **b** are executed first, followed by instantiations of instrument **a**. There is an implicit loop here which 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.

5.4.6 Instrument definition

5.4.6.1 Syntactic form

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

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

1. An identifier which defines the name of the instrument,

2. A list of zero or more identifiers which define names for the parameter fields, also called pfields, of the instrument,

3. An optional preset value for specifying a MIDI preset mapping,

4. An optional channel value for specifying a MIDI channel mapping,

5. A list of zero or more instrument variable declarations, and

6. A block of statements defining the executable functionality of the instrument.

5.4.6.2 Instrument name

Any identifier may serve as the instrument name except that the instrument name shall not be a reserved word (see Subclause 5.4.9), the name of a core opcode (see Subclause 5.5), or the name of a core wavetable generator (see Subclause 5.6). 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.

5.4.6.3 Parameter fields

<identlist> -> as given in Subclause 5.4.5.4

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.

5.4.6.4 Preset and channel tags

5.4.6.4.1 Preset tag

The **preset** tag specifies the preset number 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.

Preset values are fixed and do not change throughout an orchestra synthesis process.

See Subclause 5.10 for more normative semantics on MIDI control of orchestras

5.4.6.4.2 Channel tag

The **channel** tag specifies the channel assignment of the instrument. When MIDI instructions arrive in a MIDI stream or MIDI file controlling the orchestra, the channel numbers refer to all instruments with the given channel assignment. When continuous control instructions arrive in a MIDI stream or MIDI file, the control instructions refer to all instrument instantiations created from instruments with the given channel assignment. Zero or more instruments may have the same channel value. If a **channel** tag is not associated with a particular instrument, then that instrument is on channel 1 by default.

Channel values may be changed with the MIDI program change instruction.

See Subclause 5.10 for more normative semantics on MIDI control of orchestras.

5.4.6.5 Instrument variable declarations

5.4.6.5.1 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 5.4.5.3.2
<table declaration> as defined in Subclause 5.4.5.3.3
<identlist> as defined in Subclause 5.4.5.4

Instrument variable declarations declare variables which 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.

5.4.6.5.2 Wavetable declaration

The syntax and semantics of Subclause 5.4.5.3.3 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 which is i-rate is legal as part of the table parameter list. Reference to constants, pfields, 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.

5.4.6.5.3 Signal variables

The syntax and semantics of Subclause 5.4.5.3.2 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 must 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.

5.4.6.5.4 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.

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* which 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.

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

5.4.6.5.5 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 5.4.6.7.7. The identifier in the declaration shall be the name of a core opcode or an 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.

5.4.6.5.6 Table map definition

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

<identlist> as defined in Subclause 5.4.5.4.

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 5.4.6.7.5), 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.

5.4.6.6 Block of code statements

5.4.6.6.1 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>	-> **sbsynth (** <expr list> **;** <identlist> **;** <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 5.4.6.6.2.
<expr> as given in Subclause 5.4.6.7.

A block is a sequence of zero or more statements. A statement shall take one of 12 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

5.4.6.6.2 Assignment

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

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

<expr> as given in Subclause 5.4.6.7.

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.

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 indexing expression 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 lesser or equal rate to the rate of the assignment, the right-hand side expression shall be evaluated. At each pass equal in rate to the lvalue, 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.

5.4.6.6.3 Null assignment

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

A null assignment contains only an expression; it is provided so that opcodes which 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 lesser or equal rate to the rate of the statement, the expression shall be evaluated.

5.4.6.6.4 If

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

An **if** statement allows conditional evaluation of a block of code. The expression which 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 which would be executed slower than the guard expression. The guard expression shall be a single-valued expression.

EXAMPLE

The following code fragment is illegal:

```
asig a;
ksig k;

a = 0; while (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.

5.4.6.6.5 Else

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

An **else** statement allows disjunctive evaluation of two blocks of code. The expression which 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 which 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.

5.4.6.6.6 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 which 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 which 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 which would be executed at a rate other than the rate of the guard expression. The guard expression shall be a single-valued expression.

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.

5.4.6.6.7 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 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 one greater than the number of pfields accepted by the referenced instrument (the first expression 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.

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 lesser or equal rate to the rate of the statement, each of the expressions in the expression list is evaluated. Then, at every pass through the statement occurring at equal rate to the rate of the statement, a new instrument instantiation is created, where the duration of the new instantiation is the value of the first 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.

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 instantation 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.

5.4.6.6.8 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 5.4.5.4) 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 which reference the same bus, whether through explicit sends, calls to **outbus** or **sbsynth**, 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 which 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 compatibility rules above, shall be cleared to zero values. At every pass through the statement at any rate, the expression parameters shall each be evaluated. Then, if the pass is at a-rate, 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 which 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 which 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.

5.4.6.6.9 Wavetable bank synthesis (sbsynth)

<statement> -> **sbsynth (** <expr list> **;** <identlist> **;** <expr list> **) ;**

The **sbsynth** statement allows the use of the standardised bank synthesis procedure (see Subclause 5.9) within a SAOL instrument. There are three parameter lists to the **sbsynth** statement:

1. The first list shall be a list of three expressions, with the first expression corresponding to the sample bank number, the second to the MIDI pitch, and the third to the MIDI velocity. Each expression shall be single-valued.

2. The second list shall be a list of one, two, or three identifiers referencing busses which are defined in **send** statements, Subclause 5.4.5.5. The first bus is the stereo output bus, and the **sbsynth** statement contains two parameter expressions to this bus for the purposes of the compatibility rules in Subclause 5.4.6.6.8. The second bus, if given, is the mono reverb bus, and the **sbsynth** statement contains one parameter expression to this bus for the purposes of the compatibility rules in Subclause 5.4.6.6.8. The third bus, if given, is the mono chorus bus, and the **sbsynth** statement contains one parameter expression to this bus for the purposes of the compatibility rules in Subclause 5.4.6.6.8. It is a syntax error if busses are given in this list but are not defined in **send** statements in the global orchestra block.

3. The third list shall be a list of zero, one, two, or three expressions. The first expression, if given, corresponds to the MIDI bank number. If there are no expressions in the list, the MIDI bank number is the default value 1. The second expression, if given, corresponds to the MIDI channel number. If there are fewer than two expressions in the list, the MIDI channel number is the default value given by the instrument channel number (Subclause 5.4.6.4.2). The third expression, if given, corresponds to the MIDI preset number. If there are fewer than three expressions in the list, the MIDI preset number is the default value given by the instrument preset value (Subclause 5.4.6.4.1), if there is one, or else the default value 1. Each expression given shall be single-valued.

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

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

At each pass through the instrument, the expressions in the first list, and any expressions in the third list, shall be evaluated. Then, at each pass through the instrument at a-rate, the wavetable bank synthesis procedure described in Subclause 5.9 shall be executed using the six parameters as defined in Subclauses (1) and (3) of the numbered list in this Subclause. The output of the stereo output calculation in this procedure shall be added to the first bus referenced in the second list; then, if there is a second bus referenced in the second list, the output of the mono reverb calculation in this procedure shall be added to it; then, if there is a third bus referenced in the second list, the output of the mono chorus calculation shall be added to it.

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

5.4.6.6.10 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 should 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 should 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 should 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 pass through the instrument, the expressions in the expression list shall be evaluated. Then, at each pass through the instrument at a-rate, 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 implementors 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 5.4.5.2.4) in order to enable this mixing.

5.4.6.6.11 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 5.4.6.6.8.

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

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

At each pass through the statement, the expression list shall be evaluated. Then, at each a-rate pass through the statement, 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 5.4.6.6.8.

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

5.4.6.6.12 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.

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

At each pass through the statement at lesser or equal rate to the rate of the statement, the expression shall be evaluated. Then, at each pass through the statement at the rate of the statement, 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 must enable this by rescaling the parameter by the current value of **tempo** (Subclause 5.4.6.8.8).

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 5.4.6.6.13.

When the **extend** statement is called, the standard name **dur** shall not be updated to reflect the new duration, but keeps the value of the original duration.

5.4.6.6.13 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 which 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**.

5.4.6.7 Expressions

5.4.6.7.1 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> **)**

<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 5.4.6.7.14.

5.4.6.7.2 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.

5.4.6.7.3 Identifier

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

An identifier expression denotes a storage location or locations which contain values stored in memory. It is illegal to reference an identifier which is not declared in the local instrument or opcode scope, and which is not a standard name (see Subclause 5.4.6.7.14).

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 which 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.

5.4.6.7.4 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.

5.4.6.7.5 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 which 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 5.4.6.5.6.

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 5.5.6.

5.4.6.7.6 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 which 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 5.4.9. 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 **Error! Reference source not found.**.

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 Subclause 5.5), 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 "..." Subclause
- 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 5.4.6.6.4, 5.4.6.6.5, 5.4.6.6.6). 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 5.4.6.5.2. 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 which 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. 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 5.4.9.

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).

5.4.6.7.7 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 which 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 5.4.6.5.5. 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 5.4.9.

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 5.4.7. 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 5.4.6.7.6.

The value of the oparray call expression is determined in the same way as described for the opcode call in Subclause 5.4.6.7.6, 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.

5.4.6.7.8 Combination of vector and scalar elements in mathematical expressions

The subsequent Subclauses (Subclauses 5.4.6.7.9 through 5.4.6.7.13) 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** which 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.

5.4.6.7.9 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.

5.4.6.7.10 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.

5.4.6.7.11 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.

5.4.6.7.12 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:

Operator	Value of expression
+	$x_1 + x_2$
-	$x_1 – x_2$_
*	x_1x_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.

5.4.6.7.13 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.

5.4.6.7.14 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.

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

5.4.6.8 Standard names

5.4.6.8.1 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 which 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.

5.4.6.8.2 k_rate

```
ivar k_rate
```

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

5.4.6.8.3 s_rate

```
ivar s_rate
```

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

5.4.6.8.4 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 which 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.

5.4.6.8.5 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).

5.4.6.8.6 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.

5.4.6.8.7 dur

```
ivar dur
```

The standard name **dur**, in each scope, shall contain the duration of the instrument instantiation as originally created, or –1 if the duration was not known at instantiation.

5.4.6.8.8 tempo

```
ksig tempo
```

The standard name **tempo** shall contain the value of the current global tempo, in beats per minute. The default value is 60 beats per minute.

5.4.6.8.9 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 Subclause 5.9 for more details on MIDI control of orchestras.

5.4.6.8.10 MIDItouch

```
ksig MIDItouch
```

The **MIDItouch** standard variable shall contain, for each scope, the current value of the MIDI aftertouch on the note which caused the associated instrument instantiation to be created. See Subclause 5.9 for more details on MIDI control of orchestras.

5.4.6.8.11 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.

5.4.6.8.12 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.

5.4.6.8.13 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 5.4.5.5.

5.4.6.8.14 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.

5.4.6.8.15 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 implementors.

5.4.6.8.16 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 14496-1 Subclause XXX). The position is given by the current value of the **position** field of the **Sound** node which 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.

5.4.6.8.17 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 14496-1 Subclause XXX). The direction is given by the current value of the **direction** field of the **Sound** node which 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.

5.4.6.8.18 listenerPosition

```
ksig listenerPosition[3]
```

The **listenerPosition** name contains the absolute position of the listener in the BIFS scene graph (see ISO 14496-1 Subclause XXX). 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).

5.4.6.8.19 listenerDirection

```
ksig listenerDirection[3]
```

The **listenerDirection** name contains the orientation of the listener in the BIFS scene graph (see ISO 14496-1 Subclause XXX). 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).

5.4.6.8.20 minFront

```
ksig minFront
```

The **minFront** standard name gives one parameter of the sound radiation pattern of the sound which 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 14496-1 Subclause XXX).

5.4.6.8.21 maxFront

```
ksig maxFront
```

The **maxFront** standard name gives one parameter of the sound radiation pattern of the sound which 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 14496-1 Subclause XXX).

5.4.6.8.22 minBack

```
ksig minBack
```

The **minBack** standard name gives one parameter of the sound radiation pattern of the sound which 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 14496-1 Subclause XXX).

5.4.6.8.23 maxBack

```
ksig maxBack
```

The **maxBack** standard name gives one parameter of the sound radiation pattern of the sound which 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 14496-1 Subclause XXX).

5.4.6.8.24 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 5.11.3 for more details.

5.4.7 Opcode definition

5.4.7.1 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 which defines the rate at which the opcode executes, or indicates that the opcode is rate-polymorphic,
2. An identifier which 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.

5.4.7.2 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 **Error! Reference source not found.** for instructions on determining the rate of a rate-polymorphic opcode.

5.4.7.3 Opcode name

Any identifier may serve as the opcode name except that the opcode name shall not be a reserved word (see Subclause 5.4.8)., the name of one of the core opcodes listed in Subclause 5.5, or the name of one of the core wavetable generators listed in Subclause 5.6 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.

5.4.7.4 Formal parameter list

5.4.7.4.1 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 5.4.5.3.2.

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 5.11.2); 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** which allows formal parameters to be rate-polymorphic, see Subclause **Error! Reference source not found.**. **xsig** shall not be the rate tag of any formal parameter unless the opcode is of type **opcode**.

5.4.7.5 Opcode variable declarations

5.4.7.5.1 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 5.4.6.5.1.
<namelist> as defined in Subclause 5.4.5.3.2.

The syntax and semantics of opcode variable declarations are the same as those of instrument variable declarations as given in Subclause **Error! Reference source not found.**, 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 5.4.6.8) are within the scope of every opcode, may be referenced within opcodes, and shall have the semantics given in Subclause 5.4.6.8 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 which 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** which may be used to declare opcode variables in rate-polymorphic opcodes, see Subclause **Error! Reference source not found.**. **xsig** shall not be the rate tag of any variable unless the opcode type is **opcode**.

5.4.7.6 Opcode statement block

5.4.7.6.1 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 5.4.6.6.1.
<expr list> as defined in Subclause 5.4.6.6.

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 **Error! Reference source not found.**.

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** which is used in opcodes. This statement allows opcodes to return values back to their callers.

5.4.7.6.2 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 which 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.

5.4.7.7 Opcode rate

5.4.7.7.1 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.

5.4.7.7.2 Rate-polymorphic opcodes

Opcodes which 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.

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.

5.4.8 Template declaration

5.4.8.1 Syntactic form

<template declaration> -> **template <** <identlist> **> (** <identlist> **)**
map { <identlist> **} with {** <maplist> **}**
{ <instr variable declarations> <block> **}**<maplist> -> **<** <expr list> **> ,** <maplist>
<maplist> -> **<** <expr list> **>**

<identlist> as given in Subclause 5.4.5.4.
<namelist> as given in Subclause 5.4.5.3.2.
<instr variable declarations> as given in Subclause 5.4.6.5.1.
<expr list> as given in Subclause 5.4.6.6.1.
<block> as given in Subclause 5.4.6.6.1.

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

5.4.8.2 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 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 which are to be replaced in the subsequent code block with expressions from the 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.

5.4.8.3 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>(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) {
  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) {
  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);
  }
```

5.4.9 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 inputmod instr iopcode ivar kopcode krate ksig map oparray opcode outbus outchannels output return route send sequence sbsynth 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 – see Annex B).

5.5 SAOL core opcode definitions and semantics

5.5.1 Introduction

This Subclause describes the definitions and normative semantics for each of the core opcodes in SAOL. All core opcodes shall be implemented in every terminal complying to Profile 4.

For each core opcode, the following is described:

- The prototype, showing the rate of the opcode, the parameters which must be provided 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.

5.5.2 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 which map from one or more a-rate signals into a k-rate signal. That is, they have one or more parameters which 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 the rate of the expression (although it is not a rate-mismatch error to pass them an a-rate signal), and as **aopcode** opcodes for the purposes of determining when to execute them.

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

5.5.3 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 functions	int, 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, tableread, tablewrite, oscil, loscil, doscil, koscil
Signal generators	kline, aline, kexpon, aexpon, kphasor, aphasor, pluck, buzz, fof
Noise generators	irand, krand, arand, ilinrand, klinrand, alinrand, iexprand, kexprand, iexprand, 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, compress, pcompress
Sample conversion	decimate, upsamp, downsamp, samphold, sblock
Delays	delay, delay1, fracdelay
Effects	reverb, chorus, flange

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 which 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 "..." Subclause
- 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.

5.5.4 Math functions

5.5.4.1 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..

5.5.4.2 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**.

5.5.4.3 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.

5.5.4.4 dbamp

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

The **dbamp** 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 $10^{(\mathbf{x} - 90) / 10}$.

5.5.4.5 ampdb

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

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

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

5.5.4.6 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.

5.5.4.7 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.

5.5.4.8 exp

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

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

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

5.5.4.9 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**.

5.5.4.10 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 .

5.5.4.11 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**.

5.5.4.12 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**.

5.5.4.13 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 $[0, \pi)$.

5.5.4.14 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 $\mathbf{x}^{\mathbf{y}}$**.**

5.5.4.15 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 $\log_{10} \mathbf{x}$.

5.5.4.16 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 $\sin^{-1} \mathbf{x}$, in the range $[0, \pi)$.

5.5.4.17 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 $\cos^{-1} \mathbf{x}$, in the range $[0, \pi)$.

5.5.4.18 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 $\mathbf{x} \leq y$.

5.5.4.19 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 $y \leq \mathbf{x}$.

5.5.4.20 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.

5.5.4.21 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.

5.5.5 Pitch converters

5.5.5.1 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 0.16667 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 from the bottom of the piano keyboard, where 60 shall be middle C (C4).

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

5.5.5.2 gettune

```
opcode gettune()
```

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 5.5.5.3.

5.5.5.3 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**.

5.5.5.4 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 $100z$ is an integer. If $z < 0$ or $z > 0.11$, then z shall be set to 0 instead.

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

5.5.5.5 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 $y + 12z / 100$.

5.5.5.6 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 $t \times 2^{(y\,+\,100z/12\,-\,8.75)}$.

5.5.5.7 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 $y + 12z / 100$.

5.5.5.8 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 $t \times 2^{(\mathbf{x}\,-\,8.75)}$.

5.5.5.9 octcps

```
opcode octcps(asig 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 $\log_2(\mathbf{x} / t) + 8.75$.

5.5.5.10 midipch

```
opcode midipch(asig x)
```

The **midipch** core opcode converts pitch-class representation to MIDI 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 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 $60 + 100z + 12\,(y - 8)$.

5.5.5.11 pchmidi

```
opcode pchmidi(asig 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 $(\mathbf{x} - 60) / 12$, and let *y* be the integer part of *k*, and let *z* be the fractional part of *k*. Then, the return value shall be $y + 8 + 12z / 100$.

5.5.5.12 midioct

```
opcode midioct(asig x)
```

The **midioct** core opcode converts octave representation to MIDI representation.

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

The return value shall be calculated as follows. Let *k* be $12\,(\mathbf{x} - 8) + 60$. Then, the value of *k* rounded to the nearest integer shall be the return value.

5.5.5.13 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 $(\mathbf{x} - 60) / 12 + 8$.

5.5.5.14 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 $\log_2 (\mathbf{x} / t) + 69$. Then, the return value shall be *k* rounded to the nearest integer.

5.5.5.15 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 $t \times 2^{(\mathbf{x} - \mathbf{69}) / 12}$.

5.5.6 Table operations

5.5.6.1 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 Subclause 5.6.

The return value shall be the length of the table referenced by **t**.

5.5.6.2 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 5.5.6.6), or else it is 0.

The return value shall be the loop start point (in samples) of the wavetable referenced by **t**.

5.5.6.3 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 5.5.6.7), or else it is 0.

The return value shall be the loop end point (in samples) of the wavetable referenced by **t**.

5.5.6.4 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**.

5.5.6.5 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 5.6.2), or by the core opcode **ftsetbase** (Subclause 5.5.6.8), or else it is 0.

The return value shall be the base frequency, in Hz, of the wavetable referenced by **t**.

5.5.6.6 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**.

5.5.6.7 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**.

5.5.6.8 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**.

5.5.6.9 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 in Subclause 1.X, with a maximum passband ripple of 1 dB, a minimum stopband attenuation of 80 dB, and a maximum transition width of 10% of Nyquist.

5.5.6.10 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**.

5.5.6.11 oscil

```
aopcode oscil(table t, sig 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 must be interpolated from the nearby table values, as described in Subclause 1.X, with a maximum passband ripple of 2.5 dB, a minimum stopband attenuation of 60 dB, and a maximum transition width of 10% of Nyquist.

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**.

5.5.6.12 loscil

```
aopcode loscil(table t, sig 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 which depends on the base frequency **basefreq** and the sampling rate of the table. In this way, samples which 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 **oscil** relative to a particular state, the *internal phase* shall be set to m. 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, after the incrementation, the internal phase is not in the interval $[m,n]$, the phase shall be set to $m + \boldsymbol{p - kn}$, where p is the internal phase and k is the value floor(ph/n).

Otherwise 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 must be interpolated from the nearby table values, as described in Subclause 1.X, with a maximum passband ripple of 2.5 dB, a minimum stopband attenuation of 60 dB, and a maximum transition width of 10% of Nyquist.

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**.

5.5.6.13 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 **oscil** 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 $x = \boldsymbol{p * l}$, where p is the current internal phase, and l is the length of table **t**. If x is not an integer, then the value must be interpolated from the nearby table values, as described in

Subclause 1.X, with a maximum passband ripple of 2.5 dB, a minimum stopband attenuation of 60 dB, and a maximum transition width of 10% of Nyquist.

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.

5.5.6.14 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 **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/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 must be interpolated from the nearby table values, as described in Subclause 1.X, with a maximum passband ripple of 2.5 dB, a minimum stopband attenuation of 60 dB, and a maximum transition width of 10% of Nyquist.

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**.

5.5.7 Signal generators

5.5.7.1 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.

5.5.7.2 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 $l + (r - l) \times t/d$, 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.

5.5.7.3 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.

5.5.7.4 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.

5.5.7.5 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**.

5.5.7.6 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**.

5.5.7.7 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 **bufval**, 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**, 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 ($p = p - \text{floor}(p)$).

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 must be interpolated from nearby buffer values, as described in Subclause 1.X, with a maximum passband ripple of 2.5 dB, a minimum stopband attenuation of 60 dB, and a maximum transition width of 10% of Nyquist.

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**.

5.5.7.8 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 which 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

$$\sum_{f=\mathbf{lowharm}}^{\mathbf{lowharm}+\mathbf{nharm}} \mathbf{rolloff}^{(f-\mathbf{lowharm})} \cos 2\pi f p$$

where *p* is the internal phase.

5.5.7.9 fof

```
aopcode fof(asig f0, asig formant, table wave, table amp, ksig dur)
```

The **fof** opcode uses Rodet’s FOF method [ref XXX] to synthesise a voiced vowel formant.

5.5.8 Noise generators

5.5.8.1 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 should 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 which 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 $\int_y^z f(x)dx$.

5.5.8.2 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

$$p(x) = \begin{cases} 1/2\mathbf{p} : x \in [-\mathbf{p},\mathbf{p}] \\ 0 : otherwise \end{cases}$$

5.5.8.3 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

$$p(x) = \begin{cases} 1/2\mathbf{p} : x \in [-\mathbf{p},\mathbf{p}] \\ 0 : otherwise \end{cases}$$

5.5.8.4 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

$$p(x) = \begin{cases} 1/2\mathbf{p} : x \in [-\mathbf{p},\mathbf{p}] \\ 0 : otherwise \end{cases}$$

5.5.8.5 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

$$p(x) = \begin{cases} \text{abs}(2 / (\mathbf{p2} - \mathbf{p1}) * [(x - \mathbf{p1}) / (\mathbf{p2} - \mathbf{p1})]) & \text{if } x \in [\mathbf{p1},\mathbf{p2}] \\ 0 & \text{otherwise} \end{cases}$$

5.5.8.6 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

$$p(x) = \begin{cases} \mathrm{abs}(2 / (\mathbf{p2} - \mathbf{p1}) * [(x - \mathbf{p1}) / (\mathbf{p2} - \mathbf{p1})]) & \text{if } x \in [\mathbf{p1},\mathbf{p2}] \\ 0 & \text{otherwise} \end{cases}$$

5.5.8.7 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

$$p(x) = \begin{cases} \mathrm{abs}(2 / (\mathbf{p2} - \mathbf{p1}) * [(x - \mathbf{p1}) / (\mathbf{p2} - \mathbf{p1})]) & \text{if } x \in [\mathbf{p1},\mathbf{p2}] \\ 0 & \text{otherwise} \end{cases}$$

5.5.8.8 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

$$p(x) = \begin{cases} 0 & \text{if } x \le 0\text{, or} \\ k \exp(-kx)\text{, where } k = 1 / \mathbf{p1}, & \text{otherwise.} \end{cases}$$

5.5.8.9 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

$$p(x) = \begin{cases} 0 & \text{if } x \le 0\text{, or} \\ k \exp(-kx)\text{, where } k = 1 / \mathbf{p1}, & \text{otherwise.} \end{cases}$$

5.5.8.10 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) =	0	if $x \le 0$, or
	$k \exp(-kx)$, where $k = 1 / $ **p1**,	otherwise.

5.5.8.11 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) =	0	If $x \le 0$, or
	$k \exp(-kx)$, where $k = 1/$ (**p1 * KR)**,	otherwise.

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**.

5.5.8.12 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) =	0	If $x \le 0$, or
	$k \exp(-kx)$, where $k = 1 /$ (**p1 * SR)**,	otherwise.

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**.

5.5.8.13 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

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2\pi\times\mathbf{var}}}$$

that is, *p(x)* ~ N(**mean**, **var**) where **mean** is the mean and **var** the variance of a normal distribution.

5.5.8.14 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

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2\pi\times\mathbf{var}}},$$

that is, *p(x)* ~ N(**mean**, **var**) where **mean** is the mean and **var** the variance of a normal distribution.

5.5.8.15 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

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2\pi\times\mathbf{var}}},$$

that is, *p(x)* ~ N(**mean**, **var**) where **mean** is the mean and **var** the variance of a normal distribution.

5.5.9 Filters

5.5.9.1 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. It is a run-time error if **ctrl** is 0 or has opposite sign from the current value.

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 $o + (1 - [n - o]\, 2^{-t\,/\,\mathbf{htime}})$, 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**.

5.5.9.2 hipass

```
aopcode hipass(asig input, ksig cut)
```

The **hipass** core opcode high-pass filters its input signal. **cut** is the –6 dB cutoff 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.

5.5.9.3 lopass

```
aopcode lopass(asig input, ksig cut)
```

The **lopass** core opcode low-pass filters its input signal. **cut** is the –6 dB cutoff 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.

5.5.9.4 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 cutoff 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.

5.5.9.5 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 cutoff 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.

5.5.9.6 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.

5.5.9.7 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 which 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 output shall be **x – input * gain**.

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**.

5.5.9.8 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 which 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**.

5.5.9.9 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 implementors.

The parameters **b0, b1, b2, …** specify a FIR filter

$$H(z) = \mathbf{b0} + \mathbf{b1}\, \mathbf{z}^{-1} + \mathbf{b2}\, \mathbf{z}^{-2} + \ldots$$

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.

5.5.9.10 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 implementors.

The parameters **b0, b1, b2, …** and **a1, a2, …** specify an IIR filter

$$H(z) = \frac{\mathbf{b0} + \mathbf{b1} z^{-1} + \mathbf{b2} z^{-2} + \Lambda}{\mathbf{a1} z^{-1} + \mathbf{a2} z^{-2} + \Lambda}.$$

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.

5.5.9.11 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 implementors.

The values stored in samples 0, 1, 2, … **order** of table **t** specify a FIR filter

$$H(z) = \mathbf{t}[0] + \mathbf{t}[1]\,\mathbf{z}^{-1} + \mathbf{t}[2]\,\mathbf{z}^{-2} + \ldots\ \mathbf{t}[\mathbf{order}-1]\,\mathbf{z}^{-\mathrm{order}+1}$$

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.

5.5.9.12 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 implementors.

The values stored in samples 1, 2, … **order** of table **a** and samples 0, 1, 2, …, **order** of wavetable **b** specify a IIR filter

$$H(z) = \frac{\mathbf{b}[0] + \mathbf{b}[1]z^{-1} + \mathbf{b}[2]z^{-2} + \Lambda}{\mathbf{a}[1]z^{-1} + \mathbf{a}[2]z^{-2} + \Lambda}$$

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.

5.5.10 Spectral analysis

5.5.10.1 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 **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, **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

$$d[i] = \frac{\sum_{k=0}^{len} e^{-ijk/2\pi} x[k]}{\sqrt{2\pi}}$$

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**.

5.5.10.2 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 which contain the real and imaginary parts of a DFT, respectively. There are several optional arguments which 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 **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**, 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

$$x[i] = \frac{\sum_{k=0}^{len} e^{ijk/2\pi} d[k]}{\sqrt{2\pi}}$$

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,t,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(out,t,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 which compute functions of the FFT and return or export the results.

5.5.11 Gain control

5.5.11.1 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**.

5.5.11.2 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

$$\text{gain} \Big/ \sqrt{\frac{\sum_{i=0}^{l-1} b[i]^2}{l}}$$

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**.

5.5.11.3 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

$$\sqrt{\frac{\sum_{i=0}^{l-1} r[i]^2}{l}} \Bigg/ \sqrt{\frac{\sum_{i=0}^{l-1} b[i]^2}{l}}$$

and the buffer is cleared.

The return value at each call is **x** * **A**, where **A** is the current attenuation level.

NOTE

The **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**.

5.5.11.4 compress

```
aopcode compress(asig x, table t)
```

The **compress** core opcode performs waveform compression, expansion, waveshaping, or other waveform manipulation of an input signal, by passing it through a lookup table.

The return value is calculated as follows: let *l* be the length of table **t**. Then, if the value **x** is greater than 1 or less than –1, it shall be clipped to 1 or –1 respectively. Then, let *p* be the value *l* * (**x** / 2 + 0.5). The return value is the value of table **t** at sample *p*. If *p* is not an integer, the return value is interpolated from nearby points as described in Subclause 1.X, with a maximum passband ripple of 2.5 dB, a minimum stopband attenuation of 60 dB, and a maximum transition width of 10% of Nyquist..

5.5.11.5 pcompress

```
aopcode pcompress(asig x, table t[, ivar length])
```

The **pcompress** core opcode performs power-level compression, expansion, or other manipulation on an input 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.* When the buffer is full, the value *p* is calculated as

$$p = \sqrt{\frac{\sum_{i=0}^{l-1} b[i]^2}{l}}$$

and the buffer is cleared. Let *k* be the length of table **t**. Then, if *p* is greater than 1 or less than –1, it is clipped to 1 or –1 respectively, and the new attenuation level shall be set as the value of the table **t** at the sample *k* * *p*. If this value is not an integer, the attenuation level shall be interpolated from nearby points of **t**, as described in Subclause 1.X, with a maximum passband ripple of 2.5 dB, a minimum stopband attenuation of 60 dB, and a maximum transition width of 10% of Nyquist..

The return value at each call is **x** * **A**, where **A** is the current attenuation level.

NOTE

The **pcompress** opcode shall not have a “proper” representation of time, but shall infer it from the number of calls. If the same state of **pcompress** is referenced twice in the same a-cycle, then the effective buffer length shall be half as long as given by **length**.

5.5.11.6 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.

5.5.12 Sample conversion

5.5.12.1 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.

5.5.12.2 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.

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.

5.5.12.3 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.

5.5.12.4 samphold

```
opcode samphold(sig 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.

5.5.13 Delays

5.5.13.1 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 which 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**.

5.5.13.2 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.

5.5.13.3 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 4.

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 must be interpolated from the nearby values, as described in Subclause 1.X, with a maximum passband ripple of 1 dB, a minimum stopband attenuation of 80 dB, and a maximum transition width of 10% of Nyquist. **p2** is not used, and is ignored if provided.

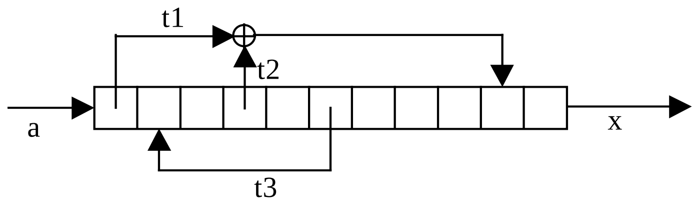
If **method** is 3, the "set" method is specified. In this case, **p1** is the position of the insertion in seconds, and **p2** is the value to insert. It is a run-time error if method 1 has not yet been called for this opcode state, or if **p1** is not provided, or if **p1** is less than 0, or if **p1**is 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 block diagram is implemented by the user-defined opcode which follows it. 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 (!first) {
    fracdelay[0](1,1);       // initialise to 1 sec long
    first = 1;
    }

  // 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);
}
```

Notice the use of the oparray construction 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.

5.5.14 Effects

5.5.14.1 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.

5.5.14.2 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 implementors.

5.5.14.3 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 implementors.

5.6 SAOL core wavetable generators

5.6.1 Introduction

This Subclause describes each of the core wavetable generators in SAOL. All core wavetable generators shall be implemented in every terminal complying to Profile 4.

For each core wavetable generator, the following is described:

- A usage description, showing the parameters which must 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.

5.6.2 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 **AudioClip** in the BIFS scene graph (ISO 14496-1 Subclause XXX).

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. This sample block of data 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 **AudioClip**, other audio coders described in this Part of ISO 14496 may be used to compress samples. The **children** fields of the **AudioSource** node responsible for instantiation of this orchestra refer to **AudioClip** nodes. Each **AudioClip** contains, after buffering as described in ISO 14496-1 Subclause XXX, several channels of audio data. If the first child has n_1 channels, the second n_2 channels, and so forth up to child *k*, then this **AudioSource** node has $K = n_0 + n_1 + ... + n_k$ 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 bitstream. The sampling rate of the wavetable shall be set to the sampling rate of the **AudioClip** node from which channel **which** is taken. The loop start, loop end, and base frequency values shall be set to 0.

If the **isReady** flag of the selected **AudioClip** node is not set when the generator is executed, then the bitstream is in error. That is, this form of this generator must 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 14496-1 Subclause XXX) of the last audio Access Unit needed in the **AudioClip** node.

For standalone systems such as authoring tools, implementors 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 aspect is that in which the tokenised bitstream element for the generator refers to a **sample** element in the bitstream.

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 **AudioClip** sample data.

5.6.3 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 smaller than the number of data values, then the wavetable shall be zero-padded at the end to length **size**. If **size** is given, and larger than the number of data values, then only the first **size** values shall be used.

The **p1, p2, p3 ...** fields are floating-point values which shall be placed in the wavetable

5.6.4 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 startup 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

$$p(x) = \begin{cases} 0 & \text{if } x \leq \mathbf{p1} \text{ or } x > \mathbf{p2}, \text{ or} \\ \text{abs}(2 / (\mathbf{p2} - \mathbf{p1}) \ \text{x} \ [(x - \mathbf{p1}) / (\mathbf{p2} - \mathbf{p1})] &) \text{ otherwise.} \end{cases}$$

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

$$p(x) = \begin{cases} 0 & \text{if } x \leq 0, \text{ or} \\ k \exp(-kx), \text{ where } k = 1 / \mathbf{p1}, & \text{otherwise.} \end{cases}$$

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

$$p(x) = \frac{e^{-(\mathbf{mean}-x)^2/(2\mathbf{var})}}{\sqrt{2\pi \times \mathbf{var}}},$$

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.

5.6.5 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.

5.6.6 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.

5.6.7 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 $\mathbf{y1(y2/y1)}^{(x\mathbf{-x1})/(\mathbf{x2-x1})}$,
x in the range **x2** through **x3** shall be set to $\mathbf{y2(y3/y2)}^{(x\mathbf{-x1})/(\mathbf{x2-x1})}$,

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.

5.6.8 Cubicseg

```
t1 table(cubicseg, size, infl1, y1, x1, y2, infl2, y3, x2, y4, infl3,...)
```

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 infl-value

For the **cubicseg** generator, sample values for samples
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 which passes through (**infl1,y1**), (**x1,y2**), and (**infl2,y3**) and which 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 which passes through (**infl2,y3**), (**x2,y4**), and (**infl3,y5**) and which 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.

5.6.9 Spline

```
t1 table(spline, size, x1, y1, x2, y2, ...)
```

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
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 which passes through (**x1,y1**), and (**x2,y2**) and which 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 which passes through (**x2,y2**), and (**x3,y3**) and which 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 which passes through (**x3,y3**), and (**x4,y4**) and which has derivative **k3** at **x3** and derivative **k4** at **x4**; and so on.

The derivative of the last cubic Subclause 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.

5.6.10 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 $p(x) = \mathbf{a0} + \mathbf{a1}x + \mathbf{a2}x^2 + \ldots$; 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

$$\mathbf{a0} + \mathbf{a1}y + \mathbf{a2}y^2 + \ldots, \text{ where } y = \mathbf{xmin} + (\mathbf{size} - x) / \mathbf{size} \times (\mathbf{xmax} - \mathbf{xmin}).$$

5.6.11 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

[XXX to be completed in editing]

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

$$\cos\left((x - \mathbf{size}) / \mathbf{size} \times \pi/2 \right).$$

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

$$1 - | (\mathbf{size}/2 - x) / (\mathbf{size}/2) |.$$

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

$$\frac{e^{-(m-x)^2/(2v)}}{\sqrt{2\pi \times v}}, \text{ where } m = \mathbf{size}/2 \text{ and } v = (\mathbf{size}/6)^{1/2}.$$

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

[XXX to be completed in editing]

If **type**=6, a boxcar window shall be used. Each sample in the range [0, **size** – 1] shall be given the value 1.

5.6.12 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

$$\mathbf{f1} \sin(2 \pi\, x/\mathbf{size}) + \mathbf{f2} \sin(4 \pi\, x/\mathbf{size}) + \mathbf{f3} \sin(6 \pi\, x/\mathbf{size}) + \ldots$$

5.6.13 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

$$\mathbf{f1} \sin(2 \pi\, x/\mathbf{size} + \mathbf{ph1}) + \mathbf{f2} \sin(4 \pi\, x/\mathbf{size} + \mathbf{ph2}) + \mathbf{f3} \sin(6 \pi\, x/\mathbf{size} + \mathbf{ph3}) + \ldots$$

5.6.14 Periodic

```
t1 table(periodic, size, p1, f1, ph1, p2, f2, ph2, ...)
```

The **periodic** core wavetable generator creates one cycle of an arbitrary periodic waveform, parametrised 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

$$\mathbf{f1}\sin(2\,\mathbf{p1}\pi\, x/\mathbf{size} + \mathbf{ph1}) + \mathbf{f2}\sin(2\,\mathbf{p2}\,\pi\, x/\mathbf{size} + \mathbf{ph2}) + \mathbf{f3}\sin(2\,\mathbf{p3}\,\pi\, x/\mathbf{size} + \mathbf{ph3}) + \ldots$$

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.

5.6.15 Buzz

```
t1 table(buzz, size, numh, lowh, fl, r)
```

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 **numh** is also not strictly positive.

lowh and **numh** 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 (**lowh** + **numh**).

If **numh** is not strictly positive, then the number of harmonics shall be given by the size of the table, such that **numh** is the greatest integer smaller than **size/2 – lowh**.

For each sample *x* in the range [0, **size** –1], the sample shall be assigned the value

If **fl** 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.

5.6.16 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 $[0, s_1\text{-}1]$, 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 $[\boldsymbol{s_1}, s_1+s_2\text{-}1]$, 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**.

5.6.17 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.

5.7 SASL syntax and semantics

5.7.1 Introduction

This Subclause describes the syntax and semantics of the score language SASL. SASL allows the simple parametric description of events which 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 Subclause 5.4, this Subclause describes a textual representation of SASL which 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.

[What happens if an event gets received in streaming score data and the time has already gone by?]

5.7.2 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 5.4.2.3.

<ident> as given in Subclause 5.4.2.2.

5.7.3 Instr line

The **instr** line specifies the construction of an instrument instantiation at a given time.

The first identifier, if given, is a label which 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 5.3.3.3. 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 file. 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 5.3.3.3) at sum of the instantiation time and the 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.

5.7.4 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 which 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 which will receive the event. For labelled control lines, the name references a variable in instruments which 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.

5.7.5 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 5.3.3.3, 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.

5.7.6 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 Subclause 5.6, 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 Subclause 5.6.

The **sample** core wavetable generator refers to a sound sample (see Subclause 5.6.2). 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 which 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.

5.7.7 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.

5.8 SAOL/SASL tokenisation

5.8.1 Introduction

This Subclause describes the normative process of mapping between the SAOL textual format used to describe syntax and semantics in Subclause 5.4, and the tokenised bitstream representation used in the bitstream definition in Subclause 5.1. 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 which is strictly normative is the process of turning a tokenised bitstream representation into sound as described in Subclause 5.3. However, it is highly recommended that implementations which allow access to bitstream contents use the textual representation described in Subclause 5.4 rather than the tokenised representation. It is nearly impossible for a human reader to understand a SAOL program presented in tokenised format.

Annex D describes the analogous detokenisation process, for informative purposes only.

5.8.2 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 5.4.9),
3. A standard name (see Subclause 5.4.6.8),
4. A core opcode name (see Subclause 5.5.3),
5. A core wavetable generator name (see Subclause 5.6),
6. A symbolic constant (a string, integer, or floating-point constant; see Subclause 5.4.2.3), or
7. An identifier (see Subclause 5.4.2.2).

Whitespace (see Subclause 5.4.2.6) 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 5.4.2.5) 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 which 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 A with that element shall be produced.

- If the element is of type (6) from above, one of the special tokens 0xF1, 0xF2, or 0xF3 shall be produced, depending on the type of the symbolic constant, and the succeeding bitstream element shall be the bitstream representation of the value.

- 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.

5.8.3 SASL Tokenisation

A SASL score must 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 Subclause 5.7 and the bitstream syntax for the various score elements, as given in Subclause 5.1.2.

5.9 Sample Bank syntax and semantics

5.9.1 Introduction

This Subclause describes the operation of the Sample Bank synthesis method for both Profile 2 and Profile 4. In Profile 2, only Sample Bank and MIDI class types shall appear in the bitstream, and this Subclause describes the normative process of generating sound from a Sample Bank bitstream data element and a sequence of MIDI instructions. In Profile 4, Sample Banks are used in the context of a SAOL instrument as described in Subclause 5.4.6.6.8, and this Subclause describes the normative process of generating sound and placing it on three busses, depending on the Sample Bank bitstream data element and the particular call to **sbsynth**.

5.9.2 Elements of bitstream

5.9.2.1 RIFF Structure

5.9.2.1.1 General RIFF File Structure

The RIFF (Resource Interchange File Format) is a tagged file structure developed for multimedia resource files, and is described in some detail in the Microsoft Windows 3.1 SDK Multimedia Programmer's Reference. The Tagged-file structure is useful because it helps prevent compatibility problems which can occur as the file definition changes over time. Because each piece of data in the file is identified by a standard header, an application that does not recognise a given data element can skip over the unknown information. The relevant information is provided in the bitstream description, Subclause 0.

A RIFF file is constructed from a basic building block called a "chunk." In 'C' syntax, a chunk is defined:

```
typedef DWORD FOURCC;      //  Four-character code
typedef struct {
  FOURCC ckID;             //  Chunk ID identifies type of data in the chunk.
  DWORD ckSize;            //  Size of chunk data in bytes, excluding pad byte.
  BYTE ckDATA[ckSize];     //  Actual data + a pad byte if req'd to word align.
};
```

Two types of chunks, the "RIFF" and "LIST" chunks, may contain nested chunks called subchunks as their data.

Within a given level of the hierarchy, the ordering of the chunks is arbitrary. Any chunk with an unknown chunk ID should be ignored.

5.9.2.1.2 The Sample Bank Chunks and Subchunks

A Structured Audio Sample Bank bitstream element comprises three chunks: an INFO-list chunk containing a number of required and optional subchunks describing the bitstream element, its history, and its intended use, an sdta-list chunk comprising a single subchunk containing any referenced digital audio samples, and a pdta-list chunk containing nine subchunks which define the articulation of the digital audio data.

5.9.2.1.3 Redundancy and Error Handling in the RIFF structure

The RIFF bitstream element structure contains redundant information regarding the length of the bitstream element and the length of the chunks and subchunks. This fact enables any reader of a SASBF bitstream element to determine if the bitstream element has been damaged by loss of data.

If any such loss is detected, the SASBF bitstream element is termed “structurally unsound” and in general should be rejected.

5.9.2.2 The INFO-list Chunk

The INFO-list chunk in a SASBF bitstream element contains three mandatory and a variety of optional subchunks as defined below. The INFO-list chunk gives basic information about the SASBF bank contained in the bitstream element.

5.9.2.2.1 The ifil Subchunk

The ifil subchunk is a mandatory subchunk identifying the SASBF specification version level to which the bitstream element complies. It is always four bytes in length, and contains data according to the structure:

```
struct sfVersionTag
{
       WORD wMajor;
       WORD wMinor;
};
```

The WORD wMajor contains the value to the left of the decimal point in the SASBF specification version. The WORD wMinor contains the value to the right of the decimal point. For example, version 2.11 would be implied if wMajor=2 and wMinor=11.

These values can be used by applications which read SASBF bitstream elements to determine if the format of the bitstream element is usable by the program. Within a fixed wMajor, the only changes to the format will be the addition of Generator, Source and Transform enumerators, and additional info subchunks. These are all defined as being ignored if unknown to the program.

If the ifil subchunk is missing, or its size is not four bytes, the bitstream element should be rejected as structurally unsound.

5.9.2.2.2 The isng Subchunk

The isng subchunk is a mandatory subchunk identifying the wavetable sound engine for which the bitstream element was optimised. It contains an ASCII string of 256 or fewer bytes including one or two terminators of value zero, so as to make the total byte count even.

The ASCII should be treated as case-sensitive. In other words “engine” is not the same as “ENGINE.”

The isng string may be optionally used by chip drivers to vary their synthesis algorithms to emulate the target sound engine.

If the isng subchunk is missing not terminated in a zero valued byte, or its contents are an unknown sound engine, the field should be ignored.

5.9.2.2.3 The INAM Subchunk

The INAM subchunk is a mandatory subchunk providing the name of the SASBF bank. It contains an ASCII string of 256 or fewer bytes including one or two terminators of value zero, so as to make the total byte count even. A typical inam subchunk would be the fourteen bytes representing “General MIDI” as twelve ASCII characters followed by two zero bytes.

The ASCII should be treated as case-sensitive. In other words “General MIDI” is not the same as “GENERAL MIDI.”

If the inam subchunk is missing, or not terminated in a zero valued byte, the field should be ignored and the user supplied with an appropriate error message if the name is queried. If the bitstream element is re-written, a valid name should be placed in the INAM field.

5.9.2.2.4 The irom Subchunk

The irom subchunk is an optional subchunk identifying a particular wavetable sound data ROM to which any ROM samples refer. It contains an ASCII string of 256 or fewer bytes including one or two terminators of value zero, so as to make the total byte count even. A typical irom field would be the six bytes representing “1MGM” as four ASCII characters followed by two zero bytes.

The ASCII should be treated as case-sensitive. In other words “1mgm” is not the same as “1MGM.”

The irom string is used by drivers to verify that the ROM data referenced by the bitstream element is available to the sound engine.

If the irom subchunk is missing, not terminated in a zero valued byte, or its contents are an unknown ROM, the field should be ignored and the bitstream element assumed to reference no ROM samples. If ROM samples are accessed, any accesses to such instruments should be terminated and not sound. A bitstream element should not be written which attempts to access ROM samples without both irom and iver present and valid.

5.9.2.2.5 The iver Subchunk

The iver subchunk is an optional subchunk identifying the particular wavetable sound data ROM revision to which any ROM samples refer. It is always four bytes in length, and contains data according to the structure:

```
struct sfVersionTag
{
       WORD wMajor;
       WORD wMinor;
};
```

The WORD wMajor contains the value to the left of the decimal point in the ROM version. The WORD wMinor contains the value to the right of the decimal point. For example, version 1.36 would be implied if wMajor=1 and wMinor=36.

The iver subchunk is used by drivers to verify that the ROM data referenced by the bitstream element is located in the exact locations specified by the sound headers.

If the iver subchunk is missing, not four bytes in length, or its contents indicate an unknown or incorrect ROM, the field should be ignored and the bitstream element assumed to reference no ROM samples. If ROM samples are accessed, any accesses to such instruments should be terminated and not sound. Note that for ROM samples to function correctly, both iver and irom must be present and valid. A bitstream element should not be written which attempts to access ROM samples without both irom and iver present and valid.

5.9.2.2.6 The ICRD Subchunk

The ICRD subchunk is an optional subchunk identifying the creation date of the SASBF bank. It contains an ASCII string of 256 or fewer bytes including one or two terminators of value zero, so as to make the total byte count even. A typical ICRD field would be the twelve bytes representing “May 1, 1995” as eleven ASCII characters followed by a zero byte.

Conventionally, the format of the string is “Month Day, Year” where Month is initially capitalised and is the conventional full English spelling of the month, Day is the date in decimal followed by a comma, and Year is the full decimal year. Thus the field should conventionally never be longer than 32 bytes.

The ICRD string is provided for library management purposes.

If the ICRD subchunk is missing, not terminated in a zero valued byte, or for some reason incapable of being faithfully copied as an ASCII string, the field should be ignored and if re-written, should not be copied. If the field’s contents are not seemingly meaningful but can faithfully reproduced, this should be done.

5.9.2.2.7 The IENG Subchunk

The IENG subchunk is an optional subchunk identifying the names of any sound designers or engineers responsible for the SASBF bank. It contains an ASCII string of 256 or fewer bytes including one or two terminators of value zero, so as to make the total byte count even. A typical IENG field would be the twelve bytes representing “Tim Swartz” as ten ASCII characters followed by two zero bytes.

The IENG string is provided for library management purposes.

If the IENG subchunk is missing, not terminated in a zero valued byte, or for some reason incapable of being faithfully copied as an ASCII string, the field should be ignored and if re-written, should not be copied. If the field’s contents are not seemingly meaningful but can faithfully reproduced, this should be done.

5.9.2.2.8 The IPRD Subchunk

The IPRD subchunk is an optional subchunk identifying any specific product for which the SASBF bank is intended. It contains an ASCII string of 256 or fewer bytes including one or two terminators of value zero, so as to make the total byte count even. A typical IPRD field would be the twelve bytes representing “MPEG SASBF” as ten ASCII characters followed by two zero bytes.

The ASCII should be treated as case-sensitive. In other words “mpeg sasbf” is not the same as “MPEG SASBF.”

The IPRD string is provided for library management purposes.

If the IPRD subchunk is missing, not terminated in a zero valued byte, or for some reason incapable of being faithfully copied as an ASCII string, the field should be ignored and if re-written, should not be copied. If the field’s contents are not seemingly meaningful but can faithfully reproduced, this should be done.

5.9.2.2.9 The ICOP Subchunk

The ICOP subchunk is an optional subchunk containing any copyright assertion string associated with the SASBF bank. It contains an ASCII string of 256 or fewer bytes including one or two terminators of value zero, so as to make the total byte count even. A typical ICOP field would be the 38 bytes representing "Copyright (c) 1998 Content Developer" as 36 ASCII characters followed by two zero bytes.

The ICOP string is provided for intellectual property protection and management purposes.

If the ICOP subchunk is missing, not terminated in a zero valued byte, or for some reason incapable of being faithfully copied as an ASCII string, the field should be ignored and if re-written, should not be copied. If the field's contents are not seemingly meaningful but can faithfully reproduced, this should be done.

5.9.2.2.10 The ICMT Subchunk

The ICMT subchunk is an optional subchunk containing any comments associated with the SASBF bank. It contains an ASCII string of 65,536 or fewer bytes including one or two terminators of value zero, so as to make the total byte count even. A typical ICMT field would be the 40 bytes representing "This space unintentionally left blank." as 38 ASCII characters followed by two zero bytes.

The ICMT string is provided for including comments.

If the ICMT subchunk is missing, not terminated in a zero valued byte, or for some reason incapable of being faithfully copied as an ASCII string, the field should be ignored and if re-written, should not be copied. If the field's contents are not seemingly meaningful but can faithfully reproduced, this should be done.

5.9.2.2.11 The ISFT Subchunk

The ISFT subchunk is an optional subchunk identifying the SASBF tools used to create and most recently modify the SASBF bank. It contains an ASCII string of 256 or fewer bytes including one or two terminators of value zero, so as to make the total byte count even. A typical ISFT field would be the twenty-six bytes representing "Editor 2.00a:Editor 2.00a" as twenty-five ASCII characters followed by a zero byte.

The ASCII should be treated as case-sensitive. In other words "Editor" is not the same as "EDITOR."

Conventionally, the tool name and revision control number are included first for the creating tool and then for the most recent modifying tool. The two strings are separated by a colon. The string should be produced by the creating program with a null modifying tool field (e.g. "Editor 2.00a:), and each time a tool modifies the bank, it should replace the modifying tool field with its own name and revision control number.

The ISFT string is provided primarily for error tracing purposes.

If the ISFT subchunk is missing, not terminated in a zero valued byte, or for some reason incapable of being faithfully copied as an ASCII string, the field should be ignored and if re-written, should not be copied. If the field's contents are not seemingly meaningful but can faithfully reproduced, this should be done.

5.9.2.3 The sdta-list Chunk

The sdta-list chunk in a SASBF bitstream element contains a single optional smpl subchunk which contains all the sound data associated with the SASBF bank. The smpl subchunk is of arbitrary length, and contains an even number of bytes.

5.9.2.3.1 Sample Data Format in the smpl Subchunk

The smpl subchunk, contains one or more samples of digital audio information in the form of linearly coded sixteen bit, signed, little endian (least significant byte first) words. Each sample is followed by a minimum of forty-six zero valued sample data points. These zero valued data points are necessary to guarantee that any reasonable upward pitch shift using any reasonable interpolator can loop on zero data at the end of the sound.

5.9.2.3.2 Sample Data Looping Rules

Within each sample, one or more loop point pairs may exist. The locations of these points are defined within the pdta-list chunk, but the sample data points themselves must comply with certain practices in order for the loop to be compatible across multiple platforms.

The loops are defined by “equivalent points” in the sample. This means that there are two sample data points which are logically equivalent, and a loop occurs when these points are spliced atop one another. In concept, the loop end point is never actually played during looping; instead the loop start point follows the point just prior to the loop end point. Because of the bandlimited nature of digital audio sampling, an artefact free loop will exhibit virtually identical data surrounding the equivalent points.

In actuality, because of the various interpolation algorithms used by wavetable synthesisers, the data surrounding both the loop start and end points may affect the sound of the loop. Hence both the loop start and end points must be surrounded by continuous audio data. For example, even if the sound is programmed to continue to loop throughout the decay, sample data points must be provided beyond the loop end point. This data will typically be identical to the data at the start of the loop. A minimum of eight valid data points are required to be present before the loop start and after the loop end.

The eight data points (four on each side) surrounding the two equivalent loop points should also be forced to be identical. By forcing the data to be identical, all interpolation algorithms are guaranteed to properly reproduce an artefact-free loop.

5.9.2.4 The pdta-list Chunk

5.9.2.4.1 The PHDR Subchunk

The PHDR subchunk is a required subchunk listing all presets within the SASBF bitstream element. It shall be a multiple of thirty-eight bytes in length, and shall contain a minimum of two records, one record for each preset and one for a terminal record according to the structure:

```
struct sfPresetHeader
{
       CHAR achPresetName[20];
       WORD wPreset;
       WORD wBank;
       WORD wPresetBagNdx;
       DWORD dwLibrary;
       DWORD dwGenre;
       DWORD dwMorphology;
};
```

The ASCII character field achPresetName shall contain the name of the preset expressed in ASCII, with unused terminal characters filled with zero valued bytes. Preset names are case-sensitive. A unique name shall always be assigned to each preset in the SASBF bank.

The WORD wPreset shall contain the MIDI Preset Number and the WORD wBank the MIDI Bank Number which apply to this preset. Note that the presets are not ordered within the SASBF bank. Presets shall have a unique set of wPreset and wBank numbers. The special case of a General MIDI percussion bank is handled conventionally by a wBank value of 128. If the value in either field is not a valid MIDI value of 0 through 127, or 128 for wBank, the preset cannot be played but shall be maintained in memory for future renumbering.

The WORD wPresetBagNdx is an index to the preset's zone list in the PBAG subchunk. Because the preset zone list is in the same order as the preset header list, the preset bag indices shall be monotonically increasing with increasing preset headers. The size of the PBAG subchunk in bytes shall be equal to four times the terminal preset's wPresetBagNdx plus four. If the preset bag indices are non-monotonic or if the terminal preset's wPresetBagNdx does not match the PBAG subchunk size, the SASBF chunk is structurally defective and shall be rejected at load time. All presets except the terminal preset shall have at least one zone.

The DWORDs dwLibrary, dwGenre and dwMorphology are reserved for future implementation in a preset library management function and should be preserved as read, and created as zero.

The terminal sfPresetHeader record should never be accessed, and exists only to provide a terminal wPresetBagNdx with which to determine the number of zones in the last preset. All other values shall be conventionally zero, with the exception of achPresetName, which may be optionally be "EOP" indicating end of presets.

If the PHDR subchunk is missing, contains fewer than two records, or its size is not a multiple of 38 bytes, the SASBF bitstream element shall be rejected as structurally unsound.

5.9.2.4.2 The PBAG Subchunk

The PBAG subchunk is a required subchunk listing all preset zones within the SASBF bitstream element. It shall be a multiple of four bytes in length, and shall contain one record for each preset zone plus one record for a terminal zone according to the structure:

```
struct sfPresetBag
{
       WORD wGenNdx;
       WORD wModNdx;
};
```

The first zone in a given preset shall be located at that preset's wPresetBagNdx. The number of zones in the preset shall be determined by the difference between the next preset's wPresetBagNdx and the current wPresetBagNdx.

The WORD wGenNdx shall be an index to the preset's zone list of generators in the PGEN subchunk, and the wModNdx shall be an index to its list of modulators in the PMOD subchunk. Because both the generator and modulator lists are in the same order as the preset header and zone lists, these indices will be monotonically increasing with increasing preset zones. The size of the PMOD subchunk in bytes shall be equal to ten times the terminal preset's wModNdx plus ten and the size of the PGEN subchunk in bytes shall be equal to four times the terminal preset's wGenNdx plus four. If the generator or modulator indices are non-monotonic or do not match the size of the respective PGEN or PMOD subchunks, the bitstream element is structurally defective and shall be rejected at load time.

If a preset has more than one zone, the first zone may be a global zone. A global zone is determined by the fact that the last generator in the list is not an Instrument generator. All generator lists must contain at least one generator with one exception - if a global zone exists for which there are no generators but only modulators. The modulator lists can contain zero or more modulators.

If a zone other than the first zone lacks an Instrument generator as its last generator, that zone should be ignored. A global zone with no modulators and no generators should also be ignored.

If the PBAG subchunk is missing, or its size is not a multiple of four bytes, the bitstream element should be rejected as structurally unsound.

5.9.2.4.3 The PMOD Subchunk

The PMOD subchunk is a required subchunk listing all preset zone modulators within the SASBF bitstream element. It is always a multiple of ten bytes in length, and contains zero or more modulators plus a terminal record according to the structure:

```
struct sfModList
{
       SFModulator sfModSrcOper;
       SFGenerator sfModDestOper;
       SHORT modAmount;
       SFModulator sfModAmtSrcOper;
       SFTransform sfModTransOper;
};
```

The preset zone's wModNdx points to the first modulator for that preset zone, and the number of modulators present for a preset zone is determined by the difference between the next higher preset zone's wModNdx and the current preset's wModNdx. A difference of zero indicates there are no modulators in this preset zone.

The sfModSrcOper is one of the SFModulator enumeration type values. Unknown or undefined values are ignored. Modulators with sfModAmtSrcOper set to 'link' which have no other modulator linked to it are ignored. This value indicates the source of data for the modulator. Note that this enumeration is two bytes in length.

The sfModDestOper indicates the destination of the modulator. The destination is a value of one of the SFGenerator enumeration type. Unknown or undefined values are ignored. Modulators with links which point to modulators which would exceed the total number of modulators for a given zone are ignored. Linked modulators that are part of circular links are ignored. Note that this enumeration is two bytes in length.

The SHORT modAmount is a signed value indicating the degree to which the source modulates the destination. A zero value indicates there is no fixed amount.

The sfModAmtSrcOper is one of the SFModulator enumeration type values. Unknown or undefined values are ignored. Modulators with sfModAmtSrcOper set to 'link' are ignored. This enumerator indicates that the specified modulation source controls the degree to which the source modulates the destination. Note that this enumeration is two bytes in length.

The sfModTransOper is one of the SFTransform enumeration type values. Unknown or undefined values are ignored. This value indicates that a transform of the specified type will be applied to the modulation source before application to the modulator. Note that this enumeration is two bytes in length.

The terminal record conventionally contains zero in all fields, and is always ignored.

A modulator is defined by its sfModSrcOper, its sfModDestOper, and its sfModSrcAmtOper. All modulators within a zone must have a unique set of these three enumerators. If a second modulator is encountered with the same three enumerators as a previous modulator with the same zone, the first modulator will be ignored.

Modulators in the PMOD subchunk act as additively relative modulators with respect to those in the IMOD subchunk. In other words, a PMOD modulator can increase or decrease the amount of an IMOD modulator.

In SASBF, no modulators have yet been defined, and the PMOD subchunk will always consist of ten zero valued bytes.

If the PMOD subchunk is missing, or its size is not a multiple of ten bytes, the bitstream element should be rejected as structurally unsound.

5.9.2.4.4 The PGEN Subchunk

The PGEN chunk is a required chunk containing a list of preset zone generators for each preset zone within the SASBF bitstream element. It is always a multiple of four bytes in length, and contains one or more generators for each preset zone (except a global zone containing only modulators) plus a terminal record according to the structure:

```
struct sfGenList
{
      SFGenerator sfGenOper;
      genAmountType genAmount;
};
```

where the types are defined:

```
typedef struct
{
      BYTE byLo;
      BYTE byHi;
} rangesType;

typedef union
{
      rangesType ranges;
      SHORT shAmount;
      WORD wAmount;
} genAmountType;
```

The sfGenOper is a value of one of the SFGenerator enumeration type values. Unknown or undefined values are ignored. This value indicates the type of generator being indicated. Note that this enumeration is two bytes in length.

The genAmount is the value to be assigned to the specified generator. Note that this can be of three formats. Certain generators specify a range of MIDI key numbers or MIDI velocities, with a minimum and maximum value. Other generators specify an unsigned WORD value. Most generators, however, specify a signed 16 bit SHORT value.

The preset zone's wGenNdx points to the first generator for that preset zone. Unless the zone is a global zone, the last generator in the list is an "Instrument" generator, whose value is a pointer to the instrument associated with that zone. If a "key range" generator exists for the preset zone, it is always the first generator in the list for that preset zone. If a "velocity range" generator exists for the preset zone, it will only be preceded by a key range generator. If any generators follow an Instrument generator, they will be ignored.

A generator is defined by its sfGenOper. All generators within a zone must have a unique sfGenOper enumerator. If a second generator is encountered with the same sfGenOper enumerator as a previous generator with the same zone, the first generator will be ignored.

Generators in the PGEN subchunk are applied relative to generators in the IGEN subchunk in an additive manner. In other words, PGEN generators increase or decrease the value of an IGEN generator.

If the PGEN subchunk is missing, or its size is not a multiple of four bytes, the bitstream element should be rejected as structurally unsound. If a key range generator is present and not the first generator, it should be ignored. If a velocity range generator is present, and is preceded by a generator other than a key range generator, it should be ignored. If a non-global list does not end in an instrument generator, the zone should be ignored. If the instrument generator value is equal to or greater than the terminal instrument, the bitstream element should be rejected as structurally unsound.

5.9.2.4.5 The INST Subchunk

The inst subchunk is a required subchunk listing all instruments within the SASBF bitstream element. It is always a multiple of twenty-two bytes in length, and contains a minimum of two records, one record for each instrument and one for a terminal record according to the structure:

```
struct sfInst
{
       CHAR achInstName[20];
       WORD wInstBagNdx;
};
```

The ASCII character field achInstName contains the name of the instrument expressed in ASCII, with unused terminal characters filled with zero valued bytes. Instrument names are case-sensitive. A unique name should always be assigned to each instrument in the SASBF bank to enable identification. However, if a bank is read containing the erroneous state of instruments with identical names, the instruments should not be discarded. They should either be preserved as read or, preferably, uniquely renamed.

The WORD wInstBagNdx is an index to the instrument’s zone list in the IBAG subchunk. Because the instrument zone list is in the same order as the instrument list, the instrument bag indices will be monotonically increasing with increasing instruments. The size of the IBAG subchunk in bytes will be four greater than four times the terminal (EOI) instrument’s wInstBagNdx. If the instrument bag indices are non-monotonic or if the terminal instrument’s wInstBagNdx does not match the IBAG subchunk size, the bitstream element is structurally defective and should be rejected at load time. All instruments except the terminal instrument must have at least one zone; any preset with no zones should be ignored.

The terminal sfInst record should never be accessed, and exists only to provide a terminal wInstBagNdx with which to determine the number of zones in the last instrument. All other values are conventionally zero, with the exception of achInstName, which should be “EOI” indicating end of instruments.

If the INST subchunk is missing, contains fewer than two records, or its size is not a multiple of 22 bytes, the bitstream element should be rejected as structurally unsound. All instruments present in the inst subchunk are typically referenced by a preset zone. However, a bitstream element containing any “orphaned” instruments need not be rejected. SASBF applications can optionally ignore or filter out these orphaned instruments based on user preference.

5.9.2.4.6 The IBAG Subchunk

The IBAG subchunk is a required subchunk listing all instrument zones within the SASBF bitstream element. It is always a multiple of four bytes in length, and contains one record for each instrument zone plus one record for a terminal zone according to the structure:

```
struct sfInstBag
{
       WORD wInstGenNdx;
       WORD wInstModNdx;
};
```

The first zone in a given instrument is located at that instrument’s wInstBagNdx. The number of zones in the instrument is determined by the difference between the next instrument’s wInstBagNdx and the current wInstBagNdx.

The WORD wInstGenNdx is an index to the instrument zone’s list of generators in the IGEN subchunk, and the wInstModNdx is an index to its list of modulators in the IMOD subchunk. Because both the generator and modulator lists are in the same order as the instrument and zone lists, these indices will be monotonically increasing with increasing zones. The size of the IMOD subchunk in bytes will be equal to ten times the terminal instrument’s wModNdx plus ten and the size of the IGEN subchunk in bytes will be equal to four times the terminal instrument’s wGenNdx plus four. If the generator or modulator indices are non-monotonic or do not match the size of the respective IGEN or IMOD subchunks, the bitstream element is structurally defective and should be rejected at load time.

If an instrument has more than one zone, the first zone may be a global zone. A global zone is determined by the fact that the last generator in the list is not a sampleID generator. All generator lists must contain at least one generator with one exception - if a global zone exists for which there are no generators but only modulators. The modulator lists can contain zero or more modulators.

If a zone other than the first zone lacks a sampleID generator as its last generator, that zone should be ignored. A global zone with no modulators and no generators should also be ignored.

If the IBAG subchunk is missing, or its size is not a multiple of four bytes, the bitstream element should be rejected as structurally unsound.

5.9.2.4.7 The IMOD Subchunk

The IMOD subchunk is a required subchunk listing all instrument zone modulators within the SASBF bitstream element. It is always a multiple of ten bytes in length, and contains zero or more modulators plus a terminal record according to the structure:

```
struct sfModList
{
       SFModulator sfModSrcOper;
       SFGenerator sfModDestOper;
       SHORT modAmount;
       SFModulator sfModAmtSrcOper;
       SFTransform sfModTransOper;
};
```

The zone’s wInstModNdx points to the first modulator for that zone, and the number of modulators present for a zone is determined by the difference between the next higher zone’s wInstModNdx and the current zone’s wModNdx. A difference of zero indicates there are no modulators in this zone.

The sfModSrcOper is one of the SFModulator enumeration type values. Unknown or undefined values are ignored. Modulators with sfModAmtSrcOper set to ‘link’ which have no other modulator linked to it are ignored. This value indicates the source of data for the modulator. Note that this enumeration is two bytes in length.

The sfModDestOper indicates the destination of the modulator. The destination is a value of one of the SFGenerator enumeration type values. Unknown or undefined values are ignored. Modulators with links which point to modulators which would exceed the total number of modulators for a given zone are ignored. Linked modulators that are part of circular links are ignored. Note that this enumeration is two bytes in length.

The SHORT modAmount is a signed value indicating the degree to which the source modulates the destination. A zero value indicates there is no fixed amount.

The sfModAmtSrcOper is one of the SFModulator enumeration type values. Unknown or undefined values are ignored. This enumerator indicates that the specified modulation source controls the degree to which the source modulates the destination. Note that this enumeration is two bytes in length.

The sfModTransOper is one of the SFTransform enumeration type values. Unknown or undefined values are ignored. This value indicates that a transform of the specified type will be applied to the modulation source before application to the modulator. Note that this enumeration is two bytes in length.

The terminal record conventionally contains zero in all fields, and is always ignored.

A modulator is defined by its sfModSrcOper, its sfModDestOper, and its sfModSrcAmtOper. All modulators within a zone must have a unique set of these three enumerators. If a second modulator is encountered with the same three enumerators as a previous modulator within the same zone, the first modulator will be ignored.

Modulators in the IMOD subchunk are absolute. This means that an IMOD modulator replaces, rather than adds to, a default modulator.

In SASBF, no modulators have yet been defined, and the IMOD subchunk will always consist of ten zero valued bytes.

If the IMOD subchunk is missing, or its size is not a multiple of ten bytes, the bitstream element should be rejected as structurally unsound.

5.9.2.4.8 The IGEN Subchunk

The IGEN chunk is a required chunk containing a list of zone generators for each instrument zone within the SASBF bitstream element. It is always a multiple of four bytes in length, and contains one or more generators for each zone (except a global zone containing only modulators) plus a terminal record according to the structure:

```
struct sfInstGenList
{
      SFGenerator sfGenOper;
      genAmountType genAmount;
};
```

where the types are defined as in the PGEN zone above.

The genAmount is the value to be assigned to the specified generator. Note that this can be of three formats. Certain generators specify a range of MIDI key numbers of MIDI velocities, with a minimum and maximum value. Other generators specify an unsigned WORD value. Most generators, however, specify a signed 16 bit SHORT value.

The zone’s wInstGenNdx points to the first generator for that zone. Unless the zone is a global zone, the last generator in the list is a “sampleID” generator, whose value is a pointer to the sample associated with that zone. If a “key range” generator exists for the zone, it is always the first generator in the list for that zone. If a “velocity range” generator exists for the zone, it will only be preceded by a key range generator. If any generators follow a sampleID generator, they will be ignored.

A generator is defined by its sfGenOper. All generators within a zone must have a unique sfGenOper enumerator. If a second generator is encountered with the same sfGenOper enumerator as a previous generator within the same zone, the first generator will be ignored.

Generators in the IGEN subchunk are absolute in nature. This means that an IGEN generator replaces, rather than adds to, the default value for the generator.

If the IGEN subchunk is missing, or its size is not a multiple of four bytes, the bitstream element should be rejected as structurally unsound. If a key range generator is present and not the first generator, it should be ignored. If a velocity range generator is present, and is preceded by a generator other than a key range generator, it should be ignored. If a non-global list does not end in a sampleID generator, the zone should be ignored. If the sampleID generator value is equal to or greater than the terminal sampleID, the bitstream element should be rejected as structurally unsound.

5.9.2.4.9 The SHDR Subchunk

The SHDR chunk is a required subchunk listing all samples within the smpl subchunk and any referenced ROM samples. It is always a multiple of forty-six bytes in length, and contains one record for each sample plus a terminal record according to the structure:

```
struct sfSample
{
       CHAR achSampleName[20];
       DWORD dwStart;
       DWORD dwEnd;
       DWORD dwStartloop;
       DWORD dwEndloop;
       DWORD dwSampleRate;
       BYTE byOriginalPitch;
       CHAR chPitchCorrection;
       WORD wSampleLink;
       SFSampleLink sfSampleType;
};
```

The ASCII character field achSampleName contains the name of the sample expressed in ASCII, with unused terminal characters filled with zero valued bytes. Sample names are case-sensitive. A unique name should always be assigned to each sample in the SASBF bank to enable identification. However, if a bank is read containing the erroneous state of samples with identical names, the samples should not be discarded. They should either be preserved as read or, preferably, uniquely renamed.

The DWORD dwStart contains the index, in sample data points, from the beginning of the sample data field to the first data point of this sample.

The DWORD dwEnd contains the index, in sample data points, from the beginning of the sample data field to the first of the set of 46 zero valued data points following this sample.

The DWORD dwStartloop contains the index, in sample data points, from the beginning of the sample data field to the first data point in the loop of this sample.

The DWORD dwEndloop contains the index, in sample data points, from the beginning of the sample data field to the first data point following the loop of this sample. Note that this is the data point “equivalent to” the first loop data point, and that to produce portable artefact free loops, the eight proximal data points surrounding both the Startloop and Endloop points should be identical.

The values of dwStart, dwEnd, dwStartloop, and dwEndloop must all be within the range of the sample data field included in the SASBF bank or referenced in the sound ROM. Also, to allow a variety of hardware platforms to be able to reproduce the data, the samples have a minimum length of 48 data points, a minimum loop size of 32 data points and a minimum of 8 valid points prior to dwStartloop and after dwEndloop. Thus dwStart must be less than dwStartloop-7, dwStartloop must be less than dwEndloop-31, and dwEndloop must be less than dwEnd-7. If these constraints are not met, the sound may optionally not be played if the hardware cannot support artefact-free playback for the parameters given.

The DWORD dwSampleRate contains the sample rate, in hertz, at which this sample was acquired or to which it was most recently converted. Values of greater than 50000 or less than 400 may not be reproducible by some hardware platforms and should be avoided. A value of zero is illegal. If an illegal or impractical value is encountered, the nearest practical value should be used.

The BYTE byOriginalPitch contains the MIDI key number of the recorded pitch of the sample. For example, a recording of an instrument playing middle C (261.62 Hz) should receive a value of 60. This value is used as the default “root key” for the sample, so that in the example, a MIDI key-on command for note number 60 would reproduce the sound at its original pitch. For unpitched sounds, a conventional value of 255 should be used. Values between 128 and 254 are illegal. Whenever an illegal value or a value of 255 is encountered, the value 60 should be used.

The CHAR chPitchCorrection contains a pitch correction in cents which should be applied to the sample on playback. The purpose of this field is to compensate for any pitch errors during the sample recording process. The correction value is that of the correction to be applied. For example, if the sound is 4 cents sharp, a correction bringing it 4 cents flat is required; thus the value should be -4.

The value in sfSampleType is an enumeration with eight defined values: monoSample = 1, rightSample = 2, leftSample = 4, linkedSample = 8, RomMonoSample = 32769, RomRightSample = 32770, RomLeftSample = 32772, and RomLinkedSample = 32776. It can be seen that this is encoded such that bit 15 of the 16 bit value is set if the sample is in ROM, and reset if it is included in the SASBF bank. The four LS bits of the word are then exclusively set indicating mono, left, right, or linked.

If the sound is flagged as a ROM sample and no valid “irom” subchunk is included; the bitstream element is structurally defective and should be rejected at load time.

If sfSampleType indicates a mono sample, then wSampleLink is undefined and its value should be conventionally zero, but will be ignored regardless of value. If sfSampleType indicates a left or right sample, then wSampleLink is the sample header index of the associated right or left stereo sample respectively. Both samples should be played entirely synchronously, with their pitch controlled by the right sample’s generators. All non-pitch generators should apply as normal; in particular the panning of the individual samples to left and right should be accomplished via the pan generator. Left-right pairs should always be found within the same instrument. Note also that no instrument should be designed in which it is possible to activate more than one instance of a particular stereo pair. The linked sample type is not currently fully defined in the SASBF specification, but will ultimately support a circularly linked list of samples using wSampleLink. Note that this enumeration is two bytes in length.

The terminal sample record is never referenced, and is conventionally entirely zero with the exception of achSampleName, which should be “EOS” indicating end of samples. All samples present in the smpl subchunk are typically referenced by an instrument, however a bitstream element containing any “orphaned” samples need not be rejected. SASBF applications can optionally ignore or filter out these orphaned samples according to user preference.

If the SHDR subchunk is missing, or its is size is not a multiple of 46 bytes the bitstream element should be rejected as structurally unsound.

5.9.3 Enumerators

5.9.3.1 Generator Enumerators

Subclause 7.1 defines the generator and generator kinds. Subclause 8.4 defines the generator operation model.

5.9.3.1.1 Kinds of Generator Enumerators

Five kinds of Generator Enumerators exist: Index Generators, Range Generators, Substitution Generators, Sample Generators, and Value Generators.

An Index Generator's amount is an index into another data structure. The only two Index Generators are Instrument and sampleID.

A Range Generator defines a range of note-on parameters outside of which the zone is undefined. Two Range Generators are currently defined, keyRange and velRange.

Substitution Generators are generators which substitute a value for a note-on parameter. Two Substitution Generators are currently defined, overridingKeyNumber and overridingVelocity.

Sample Generators are generators which directly affect a sample's properties. These generators are undefined at the preset level. The currently defined Sample Generators are the eight address offset generators, the sampleModes generator, the Overriding Root Key generator and the Exclusive Class generator.

Value Generators are generators whose value directly affects a signal processing parameter. Most generators are value generators.

5.9.3.1.2 Generator Enumerators Defined

The following is an exhaustive list of SASBF generators and their strict definitions:

0	startAddrsOffset	The offset, in sample data points, beyond the Start sample header parameter to the first sample data point to be played for this instrument. For example, if Start were 7 and startAddrsOffset were 2, the first sample data point played would be sample data point 9.
1	endAddrsOffset	The offset, in sample data points, beyond the End sample header parameter to the last sample data point to be played for this instrument. For example, if End were 17 and endAddrsOffset were -2, the last sample data point played would be sample data point 15.
2	startloopAddrsOffset	The offset, in sample data points, beyond the Startloop sample header parameter to the first sample data point to be repeated in the loop for this instrument. For example, if Startloop were 10 and startloopAddrsOffset were -1, the first repeated loop sample data point would be sample data point 9.
3	endloopAddrsOffset	The offset, in sample data points, beyond the Endloop sample header parameter to the sample data point considered equivalent to the Startloop sample data point for the loop for this instrument. For example, if Endloop were 15 and endloopAddrsOffset were 2, sample data point 17 would be considered equivalent to the

		Startloop sample data point, and hence sample data point 16 would effectively precede Startloop during looping.
4	startAddrsCoarseOffset	The offset, in 32768 sample data point increments beyond the Start sample header parameter and the first sample data point to be played in this instrument. This parameter is added to the startAddrsOffset parameter. For example, if Start were 5, startAddrsOffset were 3 and startAddrsCoarseOffset were 2, the first sample data point played would be sample data point 65544.
5	modLfoToPitch	This is the degree, in cents, to which a full scale excursion of the Modulation LFO will influence pitch. A positive value indicates a positive LFO excursion increases pitch; a negative value indicates a positive excursion decreases pitch. Pitch is always modified logarithmically, that is the deviation is in cents, semitones, and octaves rather than in Hz. For example, a value of 100 indicates that the pitch will first rise 1 semitone, then fall one semitone.
6	vibLfoToPitch	This is the degree, in cents, to which a full scale excursion of the Vibrato LFO will influence pitch. A positive value indicates a positive LFO excursion increases pitch; a negative value indicates a positive excursion decreases pitch. Pitch is always modified logarithmically, that is the deviation is in cents, semitones, and octaves rather than in Hz. For example, a value of 100 indicates that the pitch will first rise 1 semitone, then fall one semitone.
7	modEnvToPitch	This is the degree, in cents, to which a full scale excursion of the Modulation Envelope will influence pitch. A positive value indicates an increase in pitch; a negative value indicates a decrease in pitch. Pitch is always modified logarithmically, that is the deviation is in cents, semitones, and octaves rather than in Hz. For example, a value of 100 indicates that the pitch will rise 1 semitone at the envelope peak.
8	initialFilterFc	This is the cutoff and resonant frequency of the lowpass filter in absolute cent units. The lowpass filter is defined as a second order resonant pole pair whose pole frequency in Hz is defined by the Initial Filter Cutoff parameter. When the cutoff frequency exceeds 20kHz and the Q (resonance) of the filter is zero, the filter does not affect the signal.
9	initialFilterQ	This is the height above DC gain in centibels which the filter resonance exhibits at the cutoff frequency. A value of zero or less indicates the filter is not resonant; the gain at the cutoff frequency (pole angle) may be less than zero when zero is specified. The filter gain at DC is also affected by this parameter such that the gain at DC is reduced by half the specified gain. For example, for a value of 100, the filter gain at DC would be 5 dB below unity gain, and the height of the resonant peak would be 10 dB above the DC gain, or 5 dB above unity gain. Note also that if initialFilterQ is set to zero or less and the cutoff frequency exceeds 20 kHz, then the filter response is flat and unity gain.

10	modLfoToFilterFc	This is the degree, in cents, to which a full scale excursion of the Modulation LFO will influence filter cutoff frequency. A positive number indicates a positive LFO excursion increases cutoff frequency; a negative number indicates a positive excursion decreases cutoff frequency. Filter cutoff frequency is always modified logarithmically, that is the deviation is in cents, semitones, and octaves rather than in Hz. For example, a value of 1200 indicates that the cutoff frequency will first rise 1 octave, then fall one octave.
11	modEnvToFilterFc	This is the degree, in cents, to which a full scale excursion of the Modulation Envelope will influence filter cutoff frequency. A positive number indicates an increase in cutoff frequency; a negative number indicates a decrease in filter cutoff frequency. Filter cutoff frequency is always modified logarithmically, that is the deviation is in cents, semitones, and octaves rather than in Hz. For example, a value of 1000 indicates that the cutoff frequency will rise one octave at the envelope attack peak.
12	endAddrsCoarseOffset	The offset, in 32768 sample data point increments beyond the End sample header parameter and the last sample data point to be played in this instrument. This parameter is added to the endAddrsOffset parameter. For example, if End were 65536, startAddrsOffset were -3 and startAddrsCoarseOffset were -1, the last sample data point played would be sample data point 32765.
13	modLfoToVolume	This is the degree, in centibels, to which a full scale excursion of the Modulation LFO will influence volume. A positive number indicates a positive LFO excursion increases volume; a negative number indicates a positive excursion decreases volume. Volume is always modified logarithmically, that is the deviation is in decibels rather than in linear amplitude. For example, a value of 100 indicates that the volume will first rise ten dB, then fall ten dB.
14	unused1	Unused, reserved. Should be ignored if encountered.
15	chorusEffectsSend	This is the degree, in 0.1% units, to which the audio output of the note is sent to the chorus effects processor. A value of 0% or less indicates no signal is sent from this note; a value of 100% or more indicates the note is sent at full level. Note that this parameter has no effect on the amount of this signal sent to the “dry” or unprocessed portion of the output. For example, a value of 250 indicates that the signal is sent at 25% of full level (attenuation of 12 dB from full level) to the chorus effects processor.
16	reverbEffectsSend	This is the degree, in 0.1% units, to which the audio output of the note is sent to the reverb effects processor. A value of 0% or less indicates no signal is sent from this note; a value of 100% or more indicates the note is sent at full level. Note that this parameter has no effect on the amount of this signal sent to the “dry” or unprocessed portion of the output. For example, a value of 250 indicates that the signal is sent at 25% of full level (attenuation of 12 dB from full level) to the reverb effects processor.

17	pan	This is the degree, in 0.1% units, to which the “dry” audio output of the note is positioned to the left or right output. A value of -50% or less indicates the signal is sent entirely to the left output and not sent to the right output; a value of +50% or more indicates the signal is sent entirely to the right and not sent to the left. A value of zero sends the signal equally to left and right. For example, a value of -250 indicates that the signal is sent at 75% of full level to the left output and 25% of full level to the right output.
18	unused2	Unused, reserved. Should be ignored if encountered.
19	unused3	Unused, reserved. Should be ignored if encountered.
20	unused4	Unused, reserved. Should be ignored if encountered.
21	delayModLFO	This is the delay time, in absolute timecents, from key on until the Modulation LFO begins its upward ramp from zero value. A value of 0 indicates a 1 second delay. A negative value indicates a delay less than one second and a positive value a delay longer than one second. The most negative number (-32768) conventionally indicates no delay. For example, a delay of 10 msec would be 1200log2(.01) = -7973.
22	freqModLFO	This is the frequency, in absolute cents, of the Modulation LFO’s triangular period. A value of zero indicates a frequency of 8.176 Hz. A negative value indicates a frequency less than 8.176 Hz; a positive value a frequency greater than 8.176 Hz. For example, a frequency of 10 MHz would be 1200log2(.01/8.176) = -11610.
23	delayVibLFO	This is the delay time, in absolute timecents, from key on until the Vibrato LFO begins its upward ramp from zero value. A value of 0 indicates a 1 second delay. A negative value indicates a delay less than one second; a positive value a delay longer than one second. The most negative number (-32768) conventionally indicates no delay. For example, a delay of 10 msec would be 1200log2(.01) = -7973.
24	freqVibLFO	This is the frequency, in absolute cents, of the Vibrato LFO’s triangular period. A value of zero indicates a frequency of 8.176 Hz. A negative value indicates a frequency less than 8.176 Hz; a positive value a frequency greater than 8.176 Hz. For example, a frequency of 10 mHz would be 1200log2(.01/8.176) = -11610.
25	delayModEnv	This is the delay time, in absolute timecents, between key on and the start of the attack phase of the Modulation envelope. A value of 0 indicates a 1 second delay. A negative value indicates a delay less than one second; a positive value a delay longer than one second. The most negative number (-32768) conventionally indicates no delay. For example, a delay of 10 msec would be 1200log2(.01) = -7973.
26	attackModEnv	This is the time, in absolute timecents, from the end of the Modulation Envelope Delay Time until the point at which the Modulation Envelope value reaches its peak. Note that the attack is

		"convex"; the curve is nominally such that when applied to a decibel or semitone parameter, the result is linear in amplitude or Hz respectively. A value of 0 indicates a 1 second attack time. A negative value indicates a time less than one second; a positive value a time longer than one second. The most negative number (-32768) conventionally indicates instantaneous attack. For example, an attack time of 10 msec would be 1200log2(.01) = -7973.
27	holdModEnv	This is the time, in absolute timecents, from the end of the attack phase to the entry into decay phase, during which the envelope value is held at its peak. A value of 0 indicates a 1 second hold time. A negative value indicates a time less than one second; a positive value a time longer than one second. The most negative number (-32768) conventionally indicates no hold phase. For example, a hold time of 10 msec would be 1200log2(.01) = -7973.
28	decayModEnv	This is the time, in absolute timecents, for a 100% change in the Modulation Envelope value during decay phase. For the Modulation Envelope, the decay phase linearly ramps toward the sustain level. If the sustain level were zero, the Modulation Envelope Decay Time would be the time spent in decay phase. A value of 0 indicates a 1 second decay time for a zero sustain level. A negative value indicates a time less than one second; a positive value a time longer than one second. For example, a decay time of 10 msec would be 1200log2(.01) = -7973.
29	sustainModEnv	This is the decrease in level, expressed in 0.1% units, to which the Modulation Envelope value ramps during the decay phase. For the Modulation Envelope, the sustain level is properly expressed in percent of full scale. Because the volume envelope sustain level is expressed as an attenuation from full scale, the sustain level is analogously expressed as a decrease from full scale. A value of 0 indicates the sustain level is full level; this implies a zero duration of decay phase regardless of decay time. A positive value indicates a decay to the corresponding level. Values less than zero are to be interpreted as zero; values above 1000 are to be interpreted as 1000. For example, a sustain level which corresponds to an absolute value 40% of peak would be 600.
30	releaseModEnv	This is the time, in absolute timecents, for a 100% change in the Modulation Envelope value during release phase. For the Modulation Envelope, the release phase linearly ramps toward zero from the current level. If the current level were full scale, the Modulation Envelope Release Time would be the time spent in release phase until zero value were reached. A value of 0 indicates a 1 second decay time for a release from full level. A negative value indicates a time less than one second; a positive value a time longer than one second. For example, a release time of 10 msec would be 1200log2(.01) = -7973.
31	keynumToModEnvHold	This is the degree, in timecents per KeyNumber units, to which the hold time of the Modulation Envelope is decreased by increasing MIDI key number. The hold time at key number 60 is always unchanged. The unit scaling is such that a value of 100 provides a hold time which tracks the keyboard; that is, an upward octave

		causes the hold time to halve. For example, if the Modulation Envelope Hold Time were -7973 = 10 msec and the Key Number to Mod Env Hold were 50 when key number 36 was played, the hold time would be 20 msec.
32	keynumToModEnvDecay	This is the degree, in timecents per KeyNumber units, to which the decay time of the Modulation Envelope is decreased by increasing MIDI key number. The hold time at key number 60 is always unchanged. The unit scaling is such that a value of 100 provides a hold time which tracks the keyboard; that is, an upward octave causes the hold time to halve. For example, if the Modulation Envelope Hold Time were -7973 = 10 msec and the Key Number to Mod Env Hold were 50 when key number 36 was played, the hold time would be 20 msec.
33	delayVolEnv	This is the delay time, in absolute timecents, between key on and the start of the attack phase of the Volume envelope. A value of 0 indicates a 1 second delay. A negative value indicates a delay less than one second; a positive value a delay longer than one second. The most negative number (-32768) conventionally indicates no delay. For example, a delay of 10 msec would be 1200log2(.01) = -7973.
34	attackVolEnv	This is the time, in absolute timecents, from the end of the Volume Envelope Delay Time until the point at which the Volume Envelope value reaches its peak. Note that the attack is “convex”; the curve is nominally such that when applied to the decibel volume parameter, the result is linear in amplitude. A value of 0 indicates a 1 second attack time. A negative value indicates a time less than one second; a positive value a time longer than one second. The most negative number (-32768) conventionally indicates instantaneous attack. For example, an attack time of 10 msec would be 1200log2(.01) = -7973.
35	holdVolEnv	This is the time, in absolute timecents, from the end of the attack phase to the entry into decay phase, during which the Volume envelope value is held at its peak. A value of 0 indicates a 1 second hold time. A negative value indicates a time less than one second; a positive value a time longer than one second. The most negative number (-32768) conventionally indicates no hold phase. For example, a hold time of 10 msec would be 1200log2(.01) = -7973.
36	decayVolEnv	This is the time, in absolute timecents, for a 100% change in the Volume Envelope value during decay phase. For the Volume Envelope, the decay phase linearly ramps toward the sustain level, causing a constant dB change for each time unit. If the sustain level were -96dB, the Volume Envelope Decay Time would be the time spent in decay phase. A value of 0 indicates a 1 second decay time for a zero sustain level. A negative value indicates a time less than one second; a positive value a time longer than one second. For example, a decay time of 10 msec would be 1200log2(.01) = -7973.
37	sustainVolEnv	This is the decrease in level, expressed in centibels, to which the Volume Envelope value ramps during the decay phase. For the

	Volume Envelope, the sustain level is best expressed in centibels of attenuation from full scale. A value of 0 indicates the sustain level is full level; this implies a zero duration of decay phase regardless of decay time. A positive value indicates a decay to the corresponding level. Values less than zero are to be interpreted as zero; conventionally 1000 indicates full attenuation. For example, a sustain level which corresponds to an absolute value 12dB below of peak would be 120.
38 releaseVolEnv	This is the time, in absolute timecents, for a 100% change in the Volume Envelope value during release phase. For the Volume Envelope, the release phase linearly ramps toward zero from the current level, causing a constant dB change for each time unit. If the current level were full scale, the Volume Envelope Release Time would be the time spent in release phase until 96dB attenuation were reached. A value of 0 indicates a 1 second decay time for a release from full level. A negative value indicates a time less than one second; a positive value a time longer than one second. For example, a release time of 10 msec would be 1200log2(.01) = -7973.
39 keynumToVolEnvHold	This is the degree, in timecents per KeyNumber units, to which the hold time of the Volume Envelope is decreased by increasing MIDI key number. The hold time at key number 60 is always unchanged. The unit scaling is such that a value of 100 provides a hold time which tracks the keyboard; that is, an upward octave causes the hold time to halve. For example, if the Volume Envelope Hold Time were -7973 = 10 msec and the Key Number to Vol Env Hold were 50 when key number 36 was played, the hold time would be 20 msec.
40 keynumToVolEnvDecay	This is the degree, in timecents per KeyNumber units, to which the decay time of the Volume Envelope is decreased by increasing MIDI key number. The hold time at key number 60 is always unchanged. The unit scaling is such that a value of 100 provides a hold time which tracks the keyboard; that is, an upward octave causes the hold time to halve. For example, if the Volume Envelope Hold Time were -7973 = 10 msec and the Key Number to Vol Env Hold were 50 when key number 36 was played, the hold time would be 20 msec.
41 instrument	This is the index into the INST subchunk providing the instrument to be used for the current preset zone. A value of zero indicates the first instrument in the list. The value should never exceed two less than the size of the instrument list. The instrument enumerator is the terminal generator for PGEN zones. As such, it should only appear in the PGEN subchunk, and it must appear as the last generator enumerator in all but the global preset zone.
42 reserved1	Unused, reserved. Should be ignored if encountered.
43 keyRange	This is the minimum and maximum MIDI key number values for which this preset zone or instrument zone is active. The LS byte indicates the highest and the MS byte the lowest valid key. The

		keyRange enumerator is optional, but when it does appear, it must be the first generator in the zone generator list.
44	velRange	This is the minimum and maximum MIDI velocity values for which this preset zone or instrument zone is active. The LS byte indicates the highest and the MS byte the lowest valid velocity. The velRange enumerator is optional, but when it does appear, it must be preceded only by keyRange in the zone generator list.
45	startloopAddrsCoarseOffset	The offset, in 32768 sample data point increments beyond the Startloop sample header parameter and the first sample data point to be repeated in this instrument's loop. This parameter is added to the startloopAddrsOffset parameter. For example, if Startloop were 5, startloopAddrsOffset were 3 and startAddrsCoarseOffset were 2, the first sample data point in the loop would be sample data point 65544.
46	keynum	This enumerator forces the MIDI key number to effectively be interpreted as the value given. This generator can only appear at the instrument level. Valid values are from 0 to 127.
47	velocity	This enumerator forces the MIDI velocity to effectively be interpreted as the value given. This generator can only appear at the instrument level. Valid values are from 0 to 127.
48	initialAttenuation	This is the attenuation, in .4 centibel units, by which a note is attenuated below full scale. A value of zero indicates no attenuation; the note will be played at full scale. For example, a value of 60 indicates the note will be played at 2.4 dB below full scale for the note.
49	reserved2	Unused, reserved. Should be ignored if encountered.
50	endloopAddrsCoarseOffset	The offset, in 32768 sample data point increments beyond the Endloop sample header parameter to the sample data point considered equivalent to the Startloop sample data point for the loop for this instrument. This parameter is added to the endloopAddrsOffset parameter. For example, if Endloop were 5, endloopAddrsOffset were 3 and endAddrsCoarseOffset were 2, sample data point 65544 would be considered equivalent to the Startloop sample data point, and hence sample data point 65543 would effectively precede Startloop during looping.
51	coarseTune	This is a pitch offset, in semitones, which should be applied to the note. A positive value indicates the sound is reproduced at a higher pitch; a negative value indicates a lower pitch. For example, a Coarse Tune value of -4 would cause the sound to be reproduced four semitones flat.
52	fineTune	This is a pitch offset, in cents, which should be applied to the note. It is additive with coarseTune. A positive value indicates the sound is reproduced at a higher pitch; a negative value indicates a lower pitch. For example, a Fine Tuning value of -5 would cause the sound to be reproduced five cents flat.

53	sampleID	This is the index into the SHDR subchunk providing the sample to be used for the current instrument zone. A value of zero indicates the first sample in the list. The value should never exceed two less than the size of the sample list. The sampleID enumerator is the terminal generator for IGEN zones. As such, it should only appear in the IGEN subchunk, and it must appear as the last generator enumerator in all but the global zone.
54	sampleModes	This enumerator indicates a value which gives a variety of Boolean flags describing the sample for the current instrument zone. The sampleModes should only appear in the IGEN subchunk, and should not appear in the global zone. The two LS bits of the value indicate the type of loop in the sample: 0 indicates a sound reproduced with no loop, 1 indicates a sound which loops continuously, 2 is unused but should be interpreted as indicating no loop, and 3 indicates a sound which loops for the duration of key depression then proceeds to play the remainder of the sample.
55	reserved3	Unused, reserved. Should be ignored if encountered.
56	scaleTuning	This parameter represents the degree to which MIDI key number influences pitch. A value of zero indicates that MIDI key number has no effect on pitch; a value of 100 represents the usual tempered semitone scale.
57	exclusiveClass	This parameter provides the capability for a key depression in a given instrument to terminate the playback of other instruments. This is particularly useful for percussive instruments such as a hi-hat cymbal. An exclusive class value of zero indicates no exclusive class; no special action is taken. Any other value indicates that when this note is initiated, any other sounding note with the same exclusive class value should be rapidly terminated. The exclusive class generator can only appear at the instrument level. The scope of the exclusive class is the entire preset. In other words, any other instrument zone within the same preset holding a corresponding exclusive class will be terminated.
58	overridingRootKey	This parameter represents the MIDI key number at which the sample is to be played back at its original sample rate. If not present, or if present with a value of -1, then the sample header parameter Original Key is used in its place. If it is present in the range 0-127, then the indicated key number will cause the sample to be played back at its sample header Sample Rate. For example, if the sample were a recording of a piano middle C (Original Key = 60) at a sample rate of 22.050 kHz, and Root Key were set to 69, then playing MIDI key number 69 (A above middle C) would cause a piano note of pitch middle C to be heard.
59	unused5	Unused, reserved. Should be ignored if encountered.
60	endOper	Unused, reserved. Should be ignored if encountered. Unique name provides value to end of defined list.

5.9.3.1.3 Generator Summary

The following tables give the ranges and default values for all SASBF defined generators.

#	Name	Unit	Abs Zero	Min	Min Useful	Max	Max Useful	De-fault	Def Value
0	startAddrsOffset +	smpls	0	0	None	*	*	0	None
1	endAddrsOffset +	smpls	0	*	*	0	None	0	None
2	startloopAddrsOffset +	smpls	0	*	*	*	*	0	None
3	endloopAddrsOffset +	smpls	0	*	*	*	*	0	None
4	startAddrsCoarseOffset +	32k	0	0	None	*	*	0	None
5	modLfoToPitch	cent fs	0	-12000	-10 oct	1200	10 oct	0	None
6	vibLfoToPitch	cent fs	0	-12000	-10 oct	1200	10 oct	0	None
7	modEnvToPitch	cent fs	0	-12000	-10 oct	1200	10 oct	0	None
8	initialFilterFc	cent	8.176 Hz	1500	20 Hz	1350 0	20 kHz	13500	Open
9	initialFilterQ	cB	0	0	None	960	96 dB	0	None
10	modLfoToFilterFc	cent fs	0	-12000	-10 oct	1200	10 oct	0	None
11	modEnvToFilterFc	cent fs	0	-12000	-10 oct	1200	10 oct	0	None
12	endAddrsCoarseOffset +	32k	0	*	*	0	None	0	None
13	modLfoToVolume	cB fs	0	-960	-96 dB	960	96 dB	0	None
15	chorusEffectsSend	0.1%	0	0	None	1000	100%	0	None
16	reverbEffectsSend	0.1%	0	0	None	1000	100%	0	None
17	pan	0.1%	Cntr	-500	Left	+500	Right	0	Centre
21	delayModLFO	timecent	1 sec	-12000	1 msec	5000	20 sec	-12000	<1 msec
22	freqModLFO	cent	8.176 Hz	-16000	1 mHz	4500	100 Hz	0	8.176 Hz
23	delayVibLFO	timecent	1 sec	-12000	1 msec	5000	20 sec	-12000	<1 msec
24	freqVibLFO	cent	8.176 Hz	-16000	1 mHz	4500	100 Hz	0	8.176 Hz
25	delayModEnv	timecent	1 sec	-12000	1 msec	5000	20 sec	-12000	<1 msec
26	attackModEnv	timecent	1 sec	-12000	1 msec	8000	100sec	-12000	<1 msec
27	holdModEnv	timecent	1 sec	-12000	1 msec	5000	20 sec	-12000	<1 msec
28	decayModEnv	timecent	1 sec	-12000	1 msec	8000	100sec	-12000	<1 msec
29	sustainModEnv	-0.1%	attk peak	0	100%	1000	0%	0	attk pk
30	releaseModEnv	timecent	1 sec	-12000	1 msec	8000	100sec	-12000	<1 msec
31	keynumToModEnvHold	tcent/key	0	-1200	-oct/ky	1200	oct/ky	0	None
32	keynumToModEnvDecay	tcent/key	0	-1200	-oct/ky	1200	oct/ky	0	None
33	delayVolEnv	timecent	1 sec	-12000	1 msec	5000	20 sec	-12000	<1 msec
34	attackVolEnv	timecent	1 sec	-12000	1 msec	8000	100sec	-12000	<1 msec
35	holdVolEnv	timecent	1 sec	-12000	1 msec	5000	20 sec	-12000	<1 msec
36	decayVolEnv	timecent	1 sec	-12000	1 msec	8000	100sec	-12000	<1 msec
37	sustainVolEnv	cB attn	attk peak	0	0 dB	1440	144dB	0	attk pk
38	releaseVolEnv	timecent	1 sec	-12000	1 msec	8000	100sec	-12000	<1 msec
39	keynumToVolEnvHold	tcent/key	0	-1200	-oct/ky	1200	oct/ky	0	None
40	keynumToVolEnvDecay	tcent/key	0	-1200	-oct/ky	1200	oct/ky	0	None
43	keyRange	MIDI ky#	key# 0	0	lo key	127	hi key	0-127	full kbd
44	velRange	MIDI vel	0	0	min vel	127	max vel	0-127	all vels
45	startloopAddrsCoarseOffset +	smpl	0	*	*	*	*	0	None

46	keynum +	MIDI ky#	key# 0	0	lo key	127	hi key	-1	None
47	velocity +	MIDI vel	0	1	min vel	127	max vel	-1	None
48	initialAttenuation	.4 cB	0	0	0 dB	1440	144dB	0	None
50	endloopAddrsCoarseOffset	smpls	0	*	*	*	*	0	None
51	CoarseTune	semitone	0	-120	-10 oct	120	10 oct	0	None
52	fineTune	cent	0	-99	-99cnt	99	99cent	0	None
54	sampleModes +	Bit Flags	Flags	**	**	**	**	0	No Loop
56	scaleTuning	cent/key	0	0	none	1200	oct/ky	100	semi-tone
57	exclusiveClass +	arbitrary #	0	1	--	127	--	0	None
58	overridingRootKey +	MIDI ky#	key# 0	0	lo key	127	hi key	-1	None

* Range depends on values of start, loop, and end points in sample header.

** Range has discrete values based on bit flags

+ This generator is only valid at the instrument level.

5.9.3.2 Default Modulators

The “default” modulators are described below.

5.9.3.2.1 MIDI Key Velocity to Initial Attenuation

The MIDI key number is used as a Negative Unipolar source, thus the input value of 0 is mapped to a value of 127/128, an input value of 127 is mapped to 0 and all other values are mapped between 127/128 and 0 in a concave fashion. There is no secondary source for this modulator; thus its effect is the same as the effect of multiplying the amount by 1. The amount of this modulator is 960 cB (or 96 dB) of attenuation.

The product of these values is added to the initial attenuation generator.

5.9.3.2.2 MIDI Key Velocity to Filter Cutoff

The MIDI key number is used as a Negative Unipolar source, thus the input value of 0 is mapped to a value of 127/128, an input value of 127 is mapped to 0 and all other values are mapped between 127/128 and 0 in a linear fashion. There is no secondary source for this modulator; thus its effect is the same as the effect of multiplying the amount by 1. The amount of this modulator is -2400 Cents.

The product of these values is added to the Initial Filter Cutoff generator summing node.

5.9.3.2.3 MIDI Channel Pressure to Vibrato LFO Pitch Depth

The MIDI Channel Pressure data value is used as a Positive Unipolar source, thus the input value of 0 is mapped to a value of 0, an input value of 127 is mapped to 127/128 and all other values are mapped between 0 and 127/128 in a linear fashion. There is no secondary source for this modulator; thus its effect is the same as the effect of multiplying the amount by 1. The amount of this modulator is 50 cents per max excursion of vibrato modulation.

The product of these values is added to the Vibrato LFO to Pitch generator summing node.

5.9.3.2.4 MIDI Continuous Controller 1 to Vibrato LFO Pitch Depth

The MIDI Continuous Controller 1 data value is used as a Positive Unipolar source, thus the input value of 0 is mapped to a value of 0, an input value of 127 is mapped to 127/128 and all other values are mapped between 0 and 127/128 in a linear fashion. The MIDI Continuous Controller 33 data value may be optionally used for increased resolution of the controller input.

There is no secondary source for this modulator; thus its effect is the same as the effect of multiplying the amount by 1.

The amount of this modulator is 50 cents/max excursion of vibrato modulation.

The product of these values is added to the Vibrato LFO to Pitch generator summing node.

5.9.3.2.5 MIDI Continuous Controller 7 to Initial Attenuation

The MIDI Continuous Controller 7 data value is used as a Negative Unipolar source, thus the input value of 0 is mapped to a value of 127/128, an input value of 127 is mapped to 0 and all other values are mapped between 127/128 and 0 in a concave fashion. There is no secondary source for this modulator; thus its effect is the same as the effect of multiplying the amount by 1. The amount of this modulator is 960 cB (or 96 dB) of attenuation.

The product of these values is added to the initial attenuation generator.

5.9.3.2.6 MIDI Continuous Controller 10 to Pan Position

The MIDI Continuous Controller 10 data value is used as a Positive Bipolar source, thus the input value of 0 is mapped to a value of -1, an input value of 127 is mapped to 63/64 and all other values are mapped between -1 and 127/128 in a linear fashion. There is no secondary source for this modulator; thus its effect is the same as the effect of multiplying the amount by 1. The amount of this modulator is 1000 tenths of a percent panned-right.

The product of these values is added to the Pan generator summing node.

5.9.3.2.7 MIDI Continuous Controller 11 to Initial Attenuation

The MIDI Continuous Controller 11 data value is used as a Negative Unipolar source, thus the input value of 0 is mapped to a value of 127/128, an input value of 127 is mapped to 0 and all other values are mapped between 127/128 and 0 in a concave fashion. There is no secondary source for this modulator; thus its effect is the same as the effect of multiplying the amount by 1. The amount of this modulator is 960 cB (or 96 dB) of attenuation.

The product of these values is added to the initial attenuation generator.

5.9.3.2.8 MIDI Continuous Controller 91 to Reverb Effects Send

The MIDI key number is used as a Positive Unipolar source, thus the input value of 0 is mapped to a value of 0, an input value of 127 is mapped to 127/128 and all other values are mapped between 0 and 127/128 in a linear fashion. There is no secondary source for this modulator; thus its effect is the same as the effect of multiplying the amount by 1.

The amount of this modulator is 200 tenths of a percent added reverb send.

The product of these values is added to the Reverb Send generator summing node.

5.9.3.2.9 MIDI Continuous Controller 93 to Chorus Effects Send

The MIDI key number is used as a Positive Unipolar source, thus the input value of 0 is mapped to a value of 0, an input value of 127 is mapped to 127/128 and all other values are mapped between 0 and 128 in a linear fashion. There is no secondary source for this modulator; thus its effect is the same as the effect of multiplying the amount by 1.

The amount of this modulator is 200 tenths of a percent added chorus send.

The product of these values is added to the Chorus Send generator summing node.

5.9.3.2.10 MIDI Pitch Wheel to Initial Pitch Controlled by MIDI Pitch Wheel Sensitivity

The MIDI Pitch Wheel data values are used as a Positive Bipolar source, thus the input value of 0 is mapped to a value of -1, an input value of 16383 is mapped to 8191/8192 and all other values are mapped between -1 and 8191/8192 in a linear fashion.

The MIDI Pitch Wheel Sensitivity data values are used as a secondary source. This source is Positive Unipolar, thus an input value of 0 is mapped to a value of 0, an input value of 127 is mapped to 127/128 and all other values are mapped between 0 and 127/128 in a linear fashion.

The amount of this modulator is 12700 Cents.

The product of these values is added to the Initial Pitch generator summing node.

5.9.3.3 Precedence and Absolute and Relative values.

Most SASBF generators are available at both the Instrument and Preset Levels, as well as having a default value. Generators at the Instrument Level are considered “absolute” and determine an actual physical value for the associated synthesis parameter, which is used instead of the default. For example, a value of 1200 for the attackVolEnv generator would produce an absolute time of 1200 timecents or 2 seconds of attack time for the volume envelope, instead of the default value of -12000 timecents or 1 msec.

Generators at the Preset Level are instead considered “relative” and additive to all the default or instrument level generators within the Preset Zone. For example, a value of 2400 timecents for the attackVolEnv generator in a preset zone containing an instrument with two zones, one with the default attackVolEnv and one with an absolute attackVolEnv generator value of 1200 timecents would cause the default zone to actually have a value of -9600 timecents or 4 msec, and the other to have a value of 3600 timecents or 8 seconds attack time.

There are some generators which are not available at the Preset Level. These are:

#	Name
0	startAddrsOffset
1	endAddrsOffset
2	startloopAddrsOffset
3	endloopAddrsOffset
4	startAddrsCoarseOffset
12	endAddrsCoarseOffset
45	startloopAddrsCoarseOffset
46	keynum
47	velocity
50	endloopAddrsCoarseOffset
54	sampleModes

57	exclusiveClass
58	overridingRootKey

If these generators are encountered in the Preset Level, they should be ignored.

The effect of modulators on a given destination is always relative to the generator value at the Instrument level. However modulators may supersede or add to other modulators depending on their position within the hierarchy. Please see Subclause 8.4 for details on the Modulator implementation and the hierarchical details.

5.9.4 Parameters and Synthesis Model

The SASBF standard has been established with the intent of providing support for an expanding base of wavetable based synthesis models. The model supported by the SASBF specification originates with the EMU8000 wavetable synthesiser chip. The description below of the underlying synthesis model and the associated parameters are provided to allow mapping of this synthesis model onto other hardware platforms.

5.9.4.1 Synthesis Model

The SASBF specification Synthesis Model comprises a wavetable oscillator, a dynamic lowpass filter, an enveloping amplifier, and programmable sends to pan, reverb, and chorus effects units. An underlying modulation engine comprises two low frequency oscillators (LFOs) and two envelope generators with appropriate routing amplifiers.

5.9.4.1.1 Wavetable Oscillator/Interpolator

The SASBF specification wavetable oscillator model is capable of playing back a sample at an arbitrary sampling rate with an arbitrary pitch shift. In practice, the upward pitch shift (downward sample rate conversion) will be limited to a maximum value, typically at least two octaves. The pitch is described in terms of an initial pitch shift which is based on the sample's sampling rate, the root key at which the sample should be unshifted on the keyboard, the coarse, fine, and correction tunings, the effective MIDI key number, and the keyboard scale factor. All modulations in pitch are in octaves, semitones, and cents.

In general, interpolators have a symmetric impulse response, and thus a linear phase characteristic. The interpolation filter's magnitude response can be characterised by three regions - the pass band, the transition band, and the stop band.

The interpolation filter's pass band is the portion of its response which corresponds to the frequencies of the signal stored in waveform memory from DC to that signal's Nyquist frequency

The interpolation filter's transition band is the portion of its response which corresponds to the first image of the signal stored in waveform memory, that is from that signal's Nyquist frequency back to DC.

The interpolation filter's stop band is the portion of its response which corresponds to the remaining images above the transition band to the Nyquist frequency of the upsampled signal.

The guardband will be assumed to begin at 5/6 of the Nyquist frequency, as is the case for a 20 kHz bandwidth in a 48 kHz sample rate system.

Due to poor aliasing rejection in the stopband, linear interpolation does not satisfy this specification.

The specification constrains the Fourier transform of the impulse response of the interpolator. The impulse response is taken with a downward pitch shift of eight octaves. This gives a response which has been upsampled by a factor of 2**8 or 256. In other words, a frequency of the impulse response's Nyquist frequency divided by 256 gives the filter frequency corresponding to the waveform memory's Nyquist frequency. In the specification below, Fn represents this waveform memory Nyquist frequency.

(All responses measured with downward pitch shift of 8 octaves.)	
Passband Response:	
Ripple:	No more than +/- 0.5 dB
Roll-off:	Minimal, but not to exceed 6 dB at 83.3% Fn.
Transition Band Response:	
Roll-off:	Monotonic and as rapid as possible to at least -80 dB attenuation
Return Lobes:	None above -80 dB
Attenuation:	Greater than 80 dB for all frequencies DC to at least 2% Fn in the first lobe.
Stop Band Response:	
Attenuation:	Greater than 90 dB for all frequencies DC to at least 1% Fn
	Greater than 80 dB for all frequencies DC to at least 20%Fn
	Greater than 60 dB for all frequencies

5.9.4.1.2 Sample Looping

The wavetable oscillator is playing a digital sample which is described in terms of a start point, end point, and two points describing a loop. The sound can be flagged as unlooped, in which case the loop points are ignored. If the sound is looped, it can be played in two ways. If it is flagged as "loop during release", the sound is played from the start point through the loop, and loops until the note becomes inaudible. If not, the sound is played from the start point through the loop, and loops until the key is released. At this point, the next time the loop end point is reached, the sound continues through the loop end point and plays until the end point is reached, at which time audio is terminated.

5.9.4.1.3 Lowpass Filter

The synthesis model contains a resonant lowpass filter, which is characterised by a dynamic cutoff frequency and a fixed resonance (Q).

The filter is idealised at zero resonance as having a flat passband to the cutoff frequency, then a rolloff at 12 dB per octave above that frequency. The resonance, when non-zero, comprises a peak at the cutoff frequency, superimposed on the above response. The resonance is measured as a dB ratio of the resonant peak to the DC gain. The DC gain at any resonance is half of the resonance value below the DC gain at zero resonance; hence the peak height is half the resonance value above DC gain at zero resonance.

All modulations in cutoff frequency are in cents.

Topology: 2^{nd} order AR lowpass filter or equivalent. Filter coefficients are updated in a smooth manner at the sample rate.

Noise and Distortion: Less than 0.003% of full scale on any sinusoidal input for any parameter setting.

Control Parameters: Cutoff Frequency and Resonance.

Resonance Parameter: Specifies the ratio in decibels of the gain of a resonant peak to the gain at DC, when the filter cutoff frequency is set to 1.5 kHz and the modulation is zero. The DC gain of a filter is reduced from the DC gain of a filter with zero resonance by half the resonance value. The resonance parameter will remain fixed throughout the sounding of a musical note.

For a specified resonance, the actual gain ratio should be accurate within +/- 1 dB, and the DC gain accurate within +/-0.5 dB. The ratio of the gain at the resonant peak to the DC gain for all cutoff frequency values shall not deviate by more than 2dB per octave deviation from 1.5 kHz. Nominal gain at DC for a minimum resonance parameter is unity gain.

Cutoff Frequency Parameter: Specifies the frequency at which the filter achieves exactly 3dB of attenuation. The Cutoff Frequency is computed by the combination of the Initial Cutoff Frequency, specified in Hz, and the modulation, specified in semitones and fractions thereof. The Initial Cutoff Frequency is fixed throughout the duration of the a musical note; the modulation can vary at the sample rate. Within the region from 200 Hz to 6.4 kHz, the cutoff frequency parameter accuracy will be +/- 2 semitones.

Frequency Magnitude Response: When the cutoff frequency is at its maximum value and the resonance is at zero, the filter must pass audio without alteration. The filter must support cutoff frequencies down to 200 Hz. There must be a minimum of 2048 distinct cutoff frequencies per octave within the region from 200 Hz to 6.4 kHz, with a worst case spacing between distinct frequencies of 0.01 semitones.

5.9.4.1.4 Final Gain Amplifier

The final gain amplifier is a multiplier on the filter output, which is controlled by an initial gain in dB. This is added to the volume envelope. Additional modulation can also be added. The gain is always specified in dB.

5.9.4.1.5 Effects Sends

The output of the final gain amplifier can be routed into the effects unit. This unit causes the sound to be located (panned) in the stereo field, and a degree of reverberation and chorus to be added. The pan is specified in terms of percentage left and right, which also could be considered as an azimuth angle. The reverb and chorus sends are specified as a percentage of the signal amplitude to be sent to these units, from 0% to 100%.

5.9.4.1.6 Low Frequency Oscillators

The synthesis model provides for two low frequency oscillators (LFOs) for modulating pitch, filter cutoff, and amplitude. The “vibrato” LFO is only capable of modulating pitch. The “modulation” LFO can modulate any of the three parameters.

An LFO is defined as having a delay period during which its value remains zero, followed by a triangular waveshape ramping linearly to positive one, then downward to negative 1, then upward again to positive one, etc.

Each parameter can be modulated to a varying degree, either positively or negatively, by the associated LFO. Modulations of pitch and cutoff are in octaves, semitones, and cents, while modulations of amplitude are in dB. The degree of modulation is specified in cents or dB for the full scale positive LFO excursion.

5.9.4.1.7 Envelope Generators

The synthesis model provides for two envelope generators. The volume envelope generator controls the final gain amplifier and hence determines the volume contour of the sound. The modulation envelope can control pitch and/or filter cutoff.

An envelope generates a control signal in six phases. When key-on occurs, a delay period begins during which the envelope value is zero. The envelope then rises in a convex curve to a value of one during the attack phase. When a value of one is reached, the envelope enters a hold phase during which it remains at one. When the hold phase ends, the envelope enters a decay phase during which its value decreases linearly to a sustain level. When the sustain level is reached, the envelope enters sustain phase, during which the envelope stays at the sustain level. Whenever a key-off occurs, the envelope immediately enters a release phase during which the value linearly ramps from the current value to zero. When zero is reached, the envelope value remains at zero.

Modulation of pitch and filter cutoff are in octaves, semitones, and cents. These parameters can be modulated to varying degree, either positively or negatively, by the modulation envelope. The degree of modulation is specified in cents for the full scale attack peak.

The volume envelope operates in dB, with the attack peak providing a full scale output, appropriately scaled by the initial volume. The zero value, however, is actually zero gain. When 96 dB of attenuation is reached in the final gain amplifier, an abrupt jump to zero gain (infinite dB of attenuation) occurs. In a 16-bit system, this jump is inaudible.

5.9.4.1.8 Modulation Interconnection Summary

The following diagram shows the interconnections expressed in the SASBF specification synthesis model:

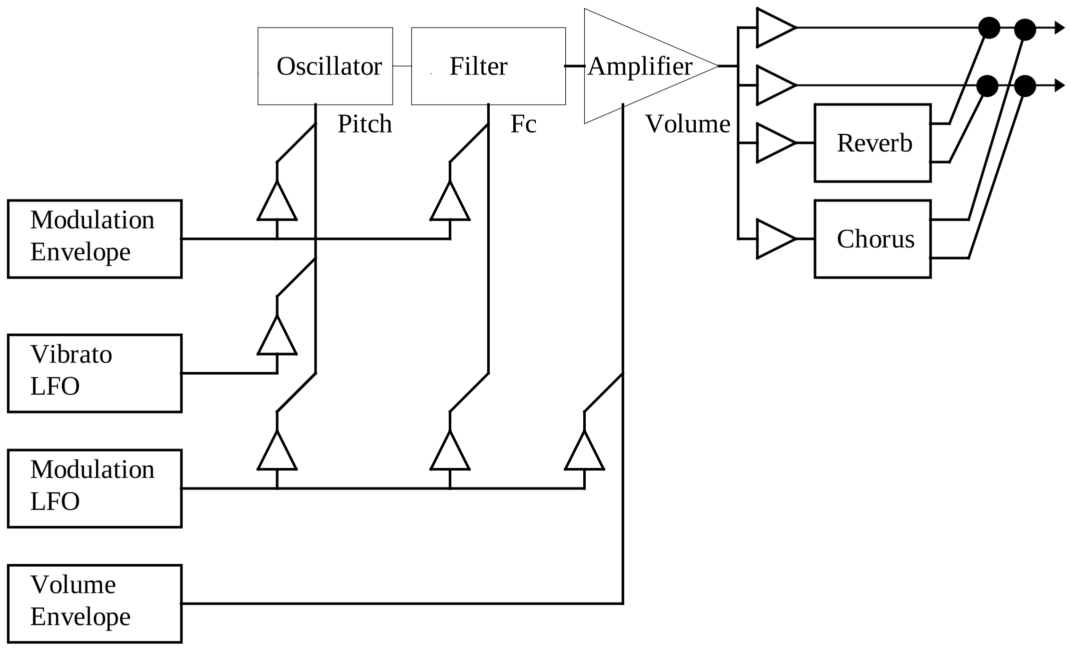


Figure 1: Generator Based Modulation Structure

5.9.4.2 MIDI Functions

The response to certain MIDI commands is defined within the MIDI specification, and therefore not considered to be part of the SASBF specification. These MIDI commands may not be used as sources for the Modulator implementation.

For completeness, the expected responses are given here.

MIDI CC0 Bank Select - When received, the following program change should select the MIDI program in this bank value instead of the default bank of 0.

MIDI CC6 - Data Entry MSB - When received, its value should be sent to either the RPN or NRPN implementation mechanism depending on the Data Entry mode.

MIDI CC32 Bank Select LSB - When received, may behave in conjunction with CC0 Bank Select to provide a total of 16384 possible MIDI banks of programs.

MIDI CC38 Data Entry LSB - When received, its value should be sent to either the RPN or NRPN implementation mechanism, depending on the Data Entry mode.

MIDI CC64 Sustain - ACTIVE when greater than or equal to 64. When the sustain function is active, all notes in the key-on state remain in the key-on state regardless of whether a key-off command for the note arrives. The key-off commands are stored, and when sustain becomes inactive, all stored key-off commands are executed.

MIDI CC66 Soft - ACTIVE when greater than or equal to 64. When active, all new key-ons are modulated in such a way to make the note sound “soft.” This typically affects initial attenuation and filter cutoff in a pre-defined manner.

MIDI CC67 Sostenuto - ACTIVE when greater than or equal to 64. When sostenuto becomes active, all notes currently in the key-on state remain in the key-on state until the sostenuto becomes inactive. All other notes behave normally. Notes maintained by sostenuto in key-on state remain in key-on state even if sustain is switched on and off.

MIDI CC98 NRPN LSB - When received, should be processed by the NRPN implementation mechanism.

MIDI CC99 NRPN MSB - When received, should put the synthesiser in NRPN Data Entry mode and then should be processed by the NRPN implementation mechanism.

MIDI CC100 RPN LSB - When received, should be processed by the RPN implementation mechanism.

MIDI CC101 RPN MSB - When received, should put the synthesiser in RPN Data Entry mode and then should be processed by the RPN implementation mechanism.

MIDI CC120 All Sound Off - When received with any data value, all notes playing in the key-on state bypass the release phase and are shut off, regardless of the sustain or sostenuto positions.

MIDI CC121 Reset All Controllers - Defined as Reset All Controllers as defined by the MIDI specification.

MIDI CC123 All Notes Off - When received with any data value, all notes playing in the key-on state immediately enter release phase, pending their status in SUSTAIN or SOSTENUTO state.

5.9.4.3 Parameter Units

The units with which SASBF generators are described are all well defined. The strict definitions appear below:

ABSOLUTE SAMPLE DATA POINTS - A numeric index of 16 bit sample data point words as stored in ROM or supplied in the smpl-ck, indexing the first sample data point word of memory or the chunk as zero.

RELATIVE SAMPLE DATA POINTS - A count of 16 bit sample data point words based on an absolute sample data point reference. A negative value implies a relative count toward the beginning of the data.

ABSOLUTE SEMITONES - An absolute logarithmic measure of frequency based on a reference of MIDI key numbers. A semitone is 1/12 of an octave, and value 69 is 440 Hz (A-440). Negative values and values above 127 are allowed.

RELATIVE SEMITONES - A relative logarithmic measure of frequency ratio based on units of 1/12 of an octave, which is the twelfth root of two, approximately 1.059463094.

ABSOLUTE CENTS - An absolute logarithmic measure of frequency based on a reference of MIDI key number scaled by 100. A cent is 1/1200 of an octave, and value 6900 is 440 Hz (A-440). Negative values and values above 12700 are allowed.

RELATIVE CENTS - A relative logarithmic measure of frequency ratio based on units of 1/1200 of an octave, which is the twelve hundredth root of two, approximately 1.000577790.

ABSOLUTE CENTIBELS - An absolute measure of the attenuation of a signal, based on a reference of zero being no attenuation. A centibel is a tenth of a decibel, or a ratio in signal amplitude of the two hundredth root of 10, approximately 1.011579454.

RELATIVE CENTIBELS - A relative measure of the attenuation of a signal. A centibel is a tenth of a decibel, or a ratio in signal amplitude of the two hundredth root of 10, approximately 1.011579454.

ABSOLUTE TIMECENTS - An absolute measure of time, based on a reference of zero being one second. A timecent represents a ratio in time of the twelve hundredth root of two, approximately 1.011579454.

RELATIVE TIMECENTS - A relative measure of time ratio, based on a unit size of the twelve hundredth root of two, approximately 1.011579454.

ABSOLUTE PERCENT - An absolute measure of gain, based on a reference of unity. In SASBF, absolute percent is measured in 0.1% units, so a value of zero is 0% and a value of 1000 is 100%.

RELATIVE PERCENT - A relative measure of gain difference. In SASBF, relative percent is measured in 0.1% units. When the gain goes below zero, zero is assumed; when the gain exceeds 100%, 100% is used.

5.9.4.4 The SASBF Generator Model

Five kinds of Generator Enumerators exist: Index Generators, Range Generators, Substitution Generators, Sample Generators, and Value Generators.

In case it is not clear in the general description of the SASBF hierarchy, the following is the precedence of SASBF generator in the hierarchy.

- A ‘generator’ sets or offsets the value of a destination or a synthesis parameter. In exception cases, it sets ranges (Range Generators), or sets values and never offsets values (Index Generators, Sample Generators, and Substitution Generators).
- A generator is defined as identical to another generator if its generator operator is the same in both generators.
- A generator in a global instrument zone which is identical to a default generator supersedes or replaces the default generator.
- A generator in a local instrument zone which is identical to a default generator or to a generator in a global instrument zone supersedes or replaces that generator.
- Points below (until noted) apply to Value Generators ONLY.
- A generator at the preset level adds to a generator at the instrument level if both generators are identical.
- A generator in a global preset zone which is identical to a default generator or to a generator in an instrument adds to that generator.
- A generator in a global preset zone which is not identical to a default generator and is not identical to a generator in an instrument has its effect added to the given synthesis parameter.
- A generator in a local preset zone which is identical to a generator in a global preset zone supersedes or replaces that generator in the global preset zone. That generator then has its effects added to the destination summing node of all zones in the given instrument.
- A generator in a local preset zone which is not identical to a default generator or a generator in a global preset zone has its effects added to the destination summing node of all zones in the given instrument.
  If the generator operator is a Range Generator, the generator values are NOT ADDED to those

in the instrument level, rather they serve as an intersection filter to those key number or velocity ranges in the instrument which is used in the preset zone.

- If the generator operator is a Substitution Generator or a Sample Generator, they are illegal at the preset level. The only Index Generator legal at the Preset Level is 'instrumentID', whereas the only Index Generator legal at the Instrument Level is 'sampleID'

5.9.4.5 The SASBF Modulator Controller Model

SASBF Modulators are used to allow real-time control over the sound in sound designer programmable manner. Each instance of a SASBF modulator structure defines a real-time perceptually additive effect to be applied to a given destination or synthesiser parameter.

Modulators provide future extensibility to the SASBF standard. They are not used in this version.

5.9.5 Error Handling

5.9.5.1 Structural Errors

Structural Errors are errors which are determined from the implicit redundancy of the SASBF RIFF bitstream element structure, and indicate that the structure is not intact. Examples are incorrect lengths for the chunks or subchunks, pointers out of valid range, or missing required chunks or subchunks for which no error correction procedure exists.

In all cases, bitstream elements should be checked for structural errors at load time, and if any are found, the bitstream elements should be rejected. Separate tools or options can be used to "repair" structurally defective bitstream elements, but these tools should validate that the reconstructed bitstream element is not only a valid SASBF bank but also complies with the intended timbral results in all cases.

5.9.5.2 Unknown Chunks

In parsing the RIFF structure, unknown but well formed chunks or subchunks may be encountered. Unknown chunks within the INFO-list chunk should simply be ignored. Other unknown chunks or subchunks are illegal and should be treated as structural errors.

5.9.5.3 Unknown Enumerators

Unknown enumerators may be encountered in Generators, Modulator Sources, or Transforms. This is to be expected if the ifil field exceeds the specification to which the application was written. Even if unexpected, unknown enumerators should simply cause the associated Generator or Modulator to be ignored.

5.9.5.4 Illegal Parameter Values

Some SASBF parameters are defined for only a limited range of the possible values which can be expressed in their field. If the value of the field is not in the defined range, the parameter has an illegal value.

Illegal values for may be detected either at load or at run time. If detected at load time, the bitstream element may optionally be rejected as structurally unsound. If detected at run time, the default value for the parameter should be used if the parameter is required, or the entire Generator or Modulator ignored if it is optional. Certain parameters may have more specific procedures for illegal values as expressed elsewhere in this specification.

5.9.5.5 Out-of-range Values

SASBF parameters have a specified minimum and useful range the span the perceptually relevant values for the associated sonic property. When the parameter value is exceeds this useful range, the parameter is said to have an out of range value.

Out of range values can result from two distinct causes. An out of range value can be actually present as a SASBF generator value, or the out of range value can be the result of the summation of instrument and preset values.

Out of range values should be handled by substituting the nearest perceptually relevant or realisable value. SASBF banks should not be created with out of range values in the instrument generators. While it is acceptable practice to create SASBF banks which produce out of range values as a result of summation, it is undesirable and should be avoided where practical.

5.9.5.6 Missing Required Parameter or Terminator

Certain parameters and terminators are required by the SASBF specification. If these are missing, the bitstream element is technically not within specification. If such a problem is detected at load time, the bitstream element may optionally be rejected as structurally unsound. If detected at run time, the instrument or zone for which the required parameter is missing should simply be ignored. If this causes no sound, the corresponding key-on event is ignored.

5.9.5.7 Illegal enumerator

Certain enumerators are illegal in certain contexts. For example, key and velocity ranges must be the first generators in a zone, instruments are not allowed in instrument zones, and sampleIDs are not allowed in preset zones. If such a problem is detected at load time, the bitstream element may optionally be rejected as structurally unsound. If detected at run time, the enumerator should simply be ignored.

5.9.6 Profile 2 (Sample Bank and MIDI decoding)

This Subclause describes the decoding process in which a bitstream conforming to Profile 2 is converted into sound. Profile 2 supports the Structured Audio Sample Bank Format and the General MIDI Format (Profile 1) for sound description. Profile 2 supports the MIDI Protocol and the Standard MIDI File Format for score specification.

5.9.6.1 Stream information header

The SASBF bitstream element is part of the bitstream header. Score elements in the form of MIDI files may be included in the bitstream header.

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

- MIDI file: The events in the MIDI file shall be time ordered, and those events registered with the scheduler.

- Sample bank: The data in the bank shall be stored, and whatever preprocessing necessary to prepare for using the bank for synthesis shall be performed. The sample bank requires no processing for this preparation. Processing may be used to improve the computational efficiency of a decoder. Banks shall be assigned consecutive increasing bank ID numbers in the order of the banks' position in the bitstream. The first bank in the bitstream shall be numbered 0 unless a bank resident in the terminal is being used. In that case, the resident bank shall be numbered 0, an other banks shall be numbered proceeding from there.

5.9.6.2 Bitstream data and sound creation

The MIDI Note On message is defined in the MIDI specification. When the SASBF decoder receives a Note On message, a voice shall be instantiated. Synthesis parameters for the voice shall be determined by the data in the sample bank containing the preset corresponding to the MIDI channel of the Note On message. The number of wavetable oscillators that are used by the voice is defined by the sample bank. Each required oscillator shall have the state variables required for synthesis allocated to it. The state variables shall be initialised according to the preset data from the sample bank.

The MIDI Program Change message is defined in the MIDI specification. When the SASBF decoder receives a Program Change message, an assignment shall be made in the scheduler between the specified MIDI channel and the specified preset. Until this assignment is changed by a subsequent Program Change message, subsequent voice instantiations using the specified MIDI channel shall use sample bank data corresponding to the assigned preset. In a MIDI program change message, if no preset exists in the specified bank with the specified preset number, a replacement preset is used. The replacement preset is the preset with the specified preset number in the bank with the highest bank ID number less than the specified bank ID number which contains a preset with the specified preset number.

The run time scheduler is used to issue MIDI messages to the SASBF decoder. MIDI messages from a MIDI standard file shall be issued by the scheduler when the scheduler clock equals or exceeds the time stamp associated with the message.

5.9.6.3 Conformance

Floating point computation is not required in Profile 2. Audio samples are stored in the bitstream as 16-bit integers. The resolution of the interpolator filter is constrained by the interpolator specification given in Subclause 0.

5.9.7 Profile 4 (Sample Bank decoding in SAOL instruments)

5.9.8 Sample Bank Format Glossary

absolute - Describes a parameter which gives a definitive real-world value. Contrast to relative.

additive - Describes a parameter which is to be numerically added to another parameter.

articulation - The process of modulation of amplitude, pitch, and timbre to produce an expressive musical note.

attack - That phase of an envelope or sound during which the amplitude increases from zero to a peak value.

attenuation - A decrease in volume or amplitude of a signal.

bag - A Sample Bank Format data structure element containing a list of zones.

balance - A form of stereo volume control in which both left and right channels are at maximum when the control is centred, and which attenuates only the opposite channel when taken to either extreme.

bank - A collection of presets. See also MIDI bank.

bipolar - In the SASBF standard, said of a modulator source whose minimum is -1 and whose maximum is 1. Contrast “unipolar”

big endian - Refers to the organisation in memory of bytes within a word such that the most significant byte occurs at the lowest address. Contrast “little endian.”

byte - A data structure element of eight bits without definition of meaning to those bits.

BYTE - A data structure element of eight bits which contains an unsigned value from 0 to 255.

case-insensitive - Indicates that an ASCII character or string treats alphabetic characters of upper or lower case as identical. Contrast “case-sensitive.”

case-sensitive - Indicates that an ASCII character or string treats alphabetic characters of upper or lower case as distinct. Contrast “case-insensitive.”

cent - A unit of pitch ratio corresponding to the twelve hundredth root of two, or one hundredth of a semitone, approximately 1.000577790.

centibel - A unit of amplitude ratio corresponding to the two hundredth root of ten, or one tenth of a decibel, approximately 1.011579454.

CHAR - A data structure of eight bits which contains a signed value from -128 to +127.

chorus - An effects processing algorithm which involves cyclically shifting the pitch of a signal and remixing it with itself to produce a time varying comb filter, giving a perception of motion and fullness to the resulting sound.

chunk - The top-level division of a RIFF file.

convex - A curve which is bowed in such a way that it is steeper on its lower portion.

concave - (1) A curve which is bowed in such a way that it is steeper on its upper portion. (2) In the SASBF standard, said of a modulator source whose shape is that of the amplitude squared characteristic. Contrast with “convex” and “linear.”

cutoff frequency - The frequency of a filter function at which the attenuation reaches a specified value.

data points - The individual values comprising a sample. Sometimes also called sample points. Contrast “sample.”

decay - The portion of an envelope or sound during which the amplitude declines from a peak to steady state value.

decibel - A unit of amplitude ratio corresponding to the twentieth root of ten, approximately 1.122018454.

delay - The portion of an envelope or LFO function which elapses from a key-on event until the amplitude becomes non-zero.

destination - The generator to which a modulator is applied.

DC gain - The degree of amplification or attenuation a system presents to a static or zero frequency signal.

digital audio - Audio represented as a sequence of quantised values spaced evenly over time. The values are called “sample data points.”

doubleword - A data structure element of 32 bits without definition of meaning to those bits.

downloadable - Said of samples which are loaded from a file into RAM, in contrast to samples which are maintained in ROM.

dry - Refers to audio which has not received any effects processing such as reverb or chorus.

DWORD - A data structure of 32 bits which contains an unsigned value from zero to 4,294,967,295.

envelope - A time varying signal which typically controls the pitch, volume, and/or filter cutoff frequency of a note, and comprises multiple phases including attack, decay, sustain, and release.

enumerated - Said of a data element whose symbols correspond to particular assigned functions.

flat - A. Said of a tone that is lower in pitch than another reference tone. B. Said of a frequency response that does not deviate significantly from a single fixed gain over the audio range.

generator - In the SASBF standard, a parameter which directly affects sound reproduction. Contrast with “modulator.”

global - Refers to parameters which affect all associated structures. See “global zone.”

global zone - A zone whose generators and modulators affect all other zones within the object.

header - A data structure element which describes several aspects of a SASBF element.

instrument - In the SASBF standard, a collection of zones which represents the sound of a single musical instrument or sound effect set.

instrument zone - A sample and associated articulation data defined to play over certain key numbers and velocities.

interpolator - A circuit or algorithm which computes intermediate points between existing sample data points. This is of particular use in the pitch shifting operation of a wavetable synthesiser, in which these intermediate points represent the output samples of the waveform at the desired pitch transposition.

key number - See MIDI key number.

LFO - Acronym for Low Frequency Oscillator. A slow periodic modulation source.

linear - In the SASBF standard, said of a modulator source whose shape is that of a straight line. Contrast with “concave.”

linear coding - The most common method of encoding amplitudes in digital audio in which each step is of equal size.

little endian - A method of ordering bytes within larger words in memory in which the least significant byte is at the lowest address. Contrast “big endian.”

loop - In wavetable synthesis, a portion of a sample which is repeated many times to increase the duration of the resulting sound.

loop points - The sample data points at which a loop begins and ends.

lowpass - Said of a filter which attenuates high frequencies but does not attenuate low frequencies.

modulator - In the SASBF standard, a parameter which routes an external controller to dynamically alter the setting of a “generator.” Contrast with “generator.”

monotonic - Continuously increasing or decreasing. Said of a sequence which never reverses direction.

MIDI - Acronym for Musical Instrument Digital Interface. The standard protocol for sending performance information to a musical synthesiser.

MIDI bank - A group of up to 128 presets selected by a MIDI “change bank” command.

MIDI continuous controller - A construct in the MIDI protocol.

MIDI key number - A construct in the MIDI protocol which accompanies a MIDI key-on or key-off command and specifies the key of the musical instrument keyboard to which the command refers.

MIDI pitch bend - A special MIDI construct akin to the MIDI continuous controllers which controls the real-time value of the pitch of all notes played in a MIDI channel.

MIDI preset - A “preset” selected to be active in a particular MIDI channel by a MIDI “change preset” command.

MIDI velocity - A construct in the MIDI protocol which accompanies a MIDI key-on or key-off command and specifies the speed with which the key was pressed or released.

modulator - In the SASBF standard, a set of parameters which affect a particular generator. Contrast with “generator.”

mono - Short for “monophonic.” Indicates a sound comprising only one channel or waveform. Contrast with “stereo.”

negative - In the SASBF standard, said of a modulator which has a negative sloping characteristic. Contrast with “positive.”

octave - A factor of two in ratio, typically applied to pitch or frequency.

orphan - Said of a data structure which under normal circumstances is referenced by a higher level, but in this particular instance is no longer linked. Specifically, it is an instrument which is not referenced by any preset zone, or a sample which is not referenced by any instrument zone.

oscillator - In wavetable synthesis, the wavetable interpolator is considered an oscillator.

pan - Short for “panorama.” This is the control of the apparent azimuth of a sound source over 180 degrees from left to right. It is generally implemented by varying the volume at the left and right speakers.

pitch - The perceived value of frequency. Generally can be used interchangeably with frequency.

pitch shift - A change in pitch. Wavetable synthesis relies on interpolators to cause pitch shift in a sample to produce the notes of the scale.

pole - A mathematical term used in filter transform analysis. Traditionally in synthesis, a pole is equated with a rolloff of 6dB per octave, and the rolloff of a filter is specified in “poles.”

positive - In the SASBF standard, said of a modulator source which has a positive sloping characteristic. Contrast “negative.”

preset - A keyboard full of sound. Typically the collection of samples and articulation data associated with a particular MIDI preset number.

preset zone - A subset of a preset containing generators, modulators, and an instrument.

proximal - Closest to. Proximal sample data points are the data points closest in either direction to the named point.

Q - A mathematical term used in filter transform analysis. Indicates the degree of resonance of the filter. In synthesis terminology, it is synonymous with resonance.

RAM - Random Access Memory. Conventionally, this term implies read-write memory. Contrast “ROM.”

record - A single instance of a data structure.

relative - Describes a parameter which merely indicates an offset from an otherwise established value. Contrast to absolute.

release - The portion of an envelope or sound during which the amplitude declines from a steady state to zero value or inaudibility.

resonance - Describes the aspect of a filter in which particular frequencies are given significantly more gain than others. The resonance can be measured in dB above the DC gain.

resonant frequency - The frequency at which resonance reaches its maximum.

reverb - Short for reverberation. In synthesis, a synthetic signal processor which adds artificial spaciousness and ambience to a sound.

RIFF - Acronym for Resource Interchange File Format.

ROM - Acronym for Read Only Memory. A memory whose contents are fixed at manufacture, and hence cannot be written by the user. Contrast with RAM.

sample - This term is often used both to indicate a “sample data point” and to indicate a collection of such points comprising a digital audio waveform. The latter meaning is exclusively used in this Subclause (the SASBF specification.)

sample rate - The frequency, in Hertz, at which sample data points are taken when recording a sample.

semitone - A unit of pitch ratio corresponding to the twelfth root of two, or one twelfth of an octave, approximately 1.059463094.

sharp - Said of a tone that is higher in pitch than another reference tone.

SHORT - A data structure element of sixteen bits which contains a signed value from -32,768 to +32,767.

soft - The pedal on a piano, so named because it causes the damper to be lowered in such a way as to soften the timbre and loudness of the notes. In MIDI, continuous controller #66, which behaves in a similar manner.

sostenuto - The pedal on a piano which causes the dampers on all keys depressed to be held until the pedal is released. In MIDI, continuous controller #67, which behaves in a similar manner.

sustain - The pedal on a piano which prevents all dampers on keys as they are depressed from being released. In MIDI, continuous controller #64, which behaves in a similar manner.

source - In a SASBF modulator, the enumerator indicating the particular real-time value which the modulator will transform, scale, and add to the destination generator.

stereo - Literally indicating three dimensions. In this specification, the term is used to mean two channel stereophonic, indicating that the sound is composed of two independent audio channels, dubbed left and right. Contrast monophonic.

subchunk - A division of a RIFF file below that of the chunk.

synthesis engine - The hardware and software associated with the signal processing and modulation path for a particular synthesiser.

synthesiser - A device ideally capable of producing arbitrary musical sound.

terminator - A data structure element indicating the final element in a sequence.

timecent - A unit of duration ratio corresponding to the twelve hundredth root of two, or one twelve hundredth of an octave, approximately 1.000577790.

transform - In a SASBF modulator, the enumerator indicating the particular transfer function through which the source will be passed prior to scaling and addition to the destination generator.

tremolo - A periodic change in amplitude of a sound, typically produced by applying a low frequency oscillator to the final volume amplifier.

triangular - A waveform which ramps upward to a positive limit, then downward at the opposite slope to the symmetrically negative limit periodically.

unipolar - In the SASBF standard, said of a modulator source whose minimum is 0 and whose maximum is 1. Contrast “bipolar.”

velocity - In synthesis, the speed with which a keyboard key is depressed, typically proportionally to the impact delivered by the musician. See also MIDI velocity.

vibrato - A periodic change in the pitch of a sound, typically produced by applying a low frequency oscillator to the oscillator pitch.

volume - The loudness or amplitude of a sound, or the control of this parameter.

wavetable - A music synthesis technique wherein musical sounds are recorded or computed mathematically and stored in a memory, then played back at a variable rate to produce the desired pitch. Additional timbral adjustments are often made to the sound thus produced using amplifiers, filters, and effects processing such as reverb and chorus.

WORD - A data structure of 16 bits which contains an unsigned value from zero to 65,535.

word - A data structure element of 16 bits without definition of meaning to those bits.

zone - An object and associated articulation data defined to play over certain key numbers and velocities.

5.10 MIDI semantics

5.10.1 Introduction

This Subclause describes the normative decoding process for Profile 1 implementations, and the normative mapping from MIDI events in the stream information header and bitstream data into SAOL semantics for Profile 4 implementations.

The MIDI standards referenced are standardised externally by the MIDI Manufacturers Association. In particular, we reference the Standard MIDIFile format, the MIDI protocol, and the General MIDI patch mapping, all standardised in **[MIDI]**. The MIDI terminology used in this Subclause is defined in that document.

5.10.2 Profile 1 decoding process

Little normative needs be said about the Profile 1 decoding process. The rules given in **[MIDI]** apply as standardised in those documents. As described in Subclause 5.2, only **midi** and **midi_file** bitstream elements shall occur in a Profile 1 bitstream.

There are no normative aspects to producing sound in Profile 1.

5.10.3 Mapping MIDI events into orchestra control

5.10.3.1 Introduction

For Profile 4, events coded as MIDI data must be converted, when they are received in the terminal as part of a Standard MIDIFile or MIDI event, into the appropriate scheduler semantics. This Subclause lists the various MIDI events and their corresponding semantics in MPEG-4.

5.10.3.2 MIDI events

5.10.3.2.1 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]**.

5.10.3.2.2 NoteOn

```
noteon channel note velocity
```

When a **noteon** event is received, each instrument in the orchestra 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 5.10.3.4) 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.

An instrument instance created in response to a **noteon** message on a particular channel is referred to as being "on" that channel.

5.10.3.2.3 NoteOff

```
noteoff channel note velocity
```

When a **noteoff** event is received, each instrument instance on channel **channel** which 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.

5.10.3.2.4 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**.

5.10.3.2.5 Aftertouch

```
touch channel note velocity
```

When a **touch** event is received, the value of the **MIDItouch** variable of each instrument instance on channel **channel** which was instantiated with note number **note** is set to **velocity**.

5.10.3.2.6 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**.

5.10.3.2.7 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 5.4.6.4.1). If there is no instrument with this preset number, then future events on the channel, until another program change is received, shall be ignored.

5.10.3.2.8 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**.

5.10.3.2.9 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.

5.10.3.2.10 MIDI messages not respected

The following MIDI messages have no meaning in MPEG-4 Profiles 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

5.10.3.3 Standard MIDI Files

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 chunks, sequences, and varying tempo.

To process a **smf** stream information element, the following steps must be taken. First, the entire stream element is parsed and cached. Then, using the sequence instructions, the tempo commands, and the sequencer model described in **[MIDI]**, the delta-times of the various events are converted into absolute timestamps. To perform this step requires converting each MIDI track chunk into a timelist, and then using the song select and song position commands to interleave the various tracks as required.

Then, for each event labelled with its proper absolute time, the event is registered with the scheduler and dispatched according to the semantics in the preceding Subclause when the appropriate time arrives.

5.10.3.4 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”; 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.

Controller	Function	Default
1	Mod Wheel	0
5	Portamento Speed	0
7	Volume	100
10	Pan	64
11	Expression	127
65	Portamento	0
66	Sus Pedal	0
67	Soft Pedal	0
84	Portamento Control	0
Pitch Bend	Pitch Bend	8192

5.11 Input sounds and relationship with AudioBIFS

5.11.1 Introduction

This Subclause describes the use of SAOL orchestras as the effects-processing functionality in the AudioBIFS (Binary Format for Scene Description) system, described in ISO 14496-1 Subclause XXX. 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 allows the inclusion of signal-processing algorithms described in SAOL which 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.

5.11.2 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 5.4.5.2.3). 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 14496-1 Subclause XXX, 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 5.4.6.8.13) 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 5.4.6.8.13.

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 which 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).

5.11.3 The AudioFX node

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 5.1.2. Further, a score may not contain new instrument events, but only control parameters for the **send** instruments defined in the orchestra.

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 5.3.2).

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.

The run-time synthesis process proceeds according to the rules cited in Subclause 5.3.3.3 for a standard SA decoding process, with the following exceptions and additions:

As no run-time events 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 which 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 which 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 is the output of the **AudioFX** node.

5.11.4 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 5.4.6.8.16-5.4.6.8.23) 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

Systems CD, Sound node specification, Subclause XXX). Content authors can use SAOL and the **AudioFX** node to creat*e* enhanced spatial effects that include reverberation, environmental attributes and complex attenuation functions, and then let the terminal-level spatial audio presentation be used to any available rendering method. 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 A Tables (normative)

This Annex contains the bitstream token table as referenced in Subclause 5.1 and Subclause 5.8. 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 0xE2 through 0xEF may be used by implementors for implementation-dependent purposes.

Bitstream token table

Token	Text
0x00	aopcode
0x01	asig
0x02	else
0x03	exports
0x04	extend
0x05	global
0x06	if
0x07	imports
0x08	inchannels
0x09	inputmod
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	sbsynth
0x1B	spatialize
0x1C	srate
0x1D	table
0x1E	tablemap
0x1F	template
0x20	turnoff
0x21	while
0x22	with
0x23	xsig
0x24	channel
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	minFront
0x41	minBack
0x42	maxFront
0x43	maxBack
0x44	params
0x45-0x4F	**(reserved)**
0x50	&&
0x51	||
0x52	>=
0x53	<=
0x54	!=
0x55	==
0x56	-
0x57	*
0x58	/
0x59	+
0x5A	>
0x5B	<
0x5C	?
0x5D	:
0x5E	(
0x5F	)
0x60	{
0x61	}
0x62	[
0x63	]
0x64	;

0x65	,	0xA2	ftloopend
0x66	=	0xA3	ftsetloop
0x67	!	0xA4	ftsetend
0x68-0x6F	**(reserved)**	0xA5	ftbasecps
0x70	sample	0xA6	ftsetbase
0x71	data	0xA7	tableread
0x72	random	0xA8	tablewrite
0x73	step	0xA9	oscil
0x74	lineseg	0xAA	loscil
0x75	expseg	0xAB	doscil
0x76	cubicseg	0xAC	koscil
0x77	polynomial	0xAD	kline
0x78	window	0xAE	aline
0x79	harm	0xAF	sblock
0x7A	harm_phase	0xB0	kexpon
0x7B	periodic	0xB1	aexpon
0x7C	buzz	0xB2	kphasor
0x7D	concat	0xB3	aphasor
0x7E	empty	0xB4	pluck
0x7F	**(reserved)**	0xB5	buzz
0x80	int	0xB6	fof
0x81	frac	0xB7	irand
0x82	dbamp	0xB8	krand
0x83	ampdb	0xB9	arand
0x84	abs	0xBA	ilinrand
0x85	exp	0xBB	klinrand
0x86	log	0xBC	alinrand
0x87	sqrt	0xBD	iexprand
0x88	sin	0xBE	kexprand
0x89	cos	0xBF	aexprand
0x8A	atan	0xC0	kpoissonrand
0x8B	pow	0xC1	apoissonrand
0x8C	log10	0xC2	igaussrand
0x8D	asin	0xC3	kgaussrand
0x8E	acos	0xC4	agaussrand
0x8F	floor	0xC5	port
0x90	ceil	0xC6	hipass
0x91	min	0xC7	lopass
0x92	max	0xC8	bandpass
0x93	pchoct	0xC9	bandstop
0x94	octpch	0xCA	fir
0x95	cpspch	0xCB	iir
0x96	pchcps	0xCC	firt
0x97	cpsoct	0xCD	iirt
0x98	octcps	0xCE	biquad
0x99	pchmidi	0xCF	fft
0x9A	midipch	0xD0	ifft
0x9B	octmidi	0xD1	rms
0x9C	midioct	0xD2	gain
0x9D	cpsmidi	0xD3	balance
0x9E	midicps	0xD4	decimate
0x9F	**(reserved)**	0xD5	upsamp
0xA0	ftlen	0xD6	downsamp
0xA1	ftloop	0xD7	samphold

0xD8	`delay`	0xE2	`gettune`
0xD9	`delay1`	0xE3	`settune`
0xDA	`fracdelay`	0xE4	`ftsr`
0xDB	`comb`	0xE5-0xEF	**(free)**
0xDC	`allpass`	0xF0	`<symbol>`
0xDD	`chorus`	0xF1	`<integer>`
0xDE	`flange`	0xF2	`<number>`
0xDF	`reverb`	0xF3	`<string>`
0xE0	`compress`	0xF4-0xFF	**(reserved)**
0xE1	`pcompress`	0xFF	`<EOO>`

Annex B Encoding (informative)

This Annex, provided for informative purposes only, provides guidelines as to the functioning of a typical Structured Audio encoder. As of this writing, the capabilities of audio-processing algorithms for performing automatic source separation, identification of instruments, identification of spatial qualities, etc., are not sufficiently advanced to perform fully automatic structured-audio encoding. Instead, for the near future, encoding Structured Audio bitstreams will likely occur in conjunction with authoring tools, where sound designers compose and orchestrate soundtracks and design instruments, and the results are then packaged into a legal Structured Audio bitstream.

[to be expanded]

Annex C lex/yacc grammars for SAOL (informative)

C.1 Introduction

This Annex provides grammars using the widely-available tools 'lex' and 'yacc' which conform to the SAOL specification in this document. They are provided for informative purposes only; implementors are free to use whichever tools they desire, or no tools, in building an implementation.

The reference software for Structured Audio in ISO 14496-5 builds the lexer and parser for SAOL out of these grammars by augmenting them with more processing and data-structure construction.

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]+)?

%%

"//"          { }
"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) ; }
"inputmod"    { return(INPUTMOD) ; }
"instr"       { return(INSTR); }
"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) ; }
"return"      { return(RETURN) ; }
"route"       { return(ROUTE) ; }
"send"        { return(SEND) ; }
"sequence"    { return(SEQUENCE) ; }
"sbsynth"     { return(SFSYNTH) ; }
"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}    { return(STRCONST); }
{IDENT}      { return(IDENT) ; }
{INTGR}       { return(INTGR) ; }
{NUMBER}      { return(NUMBER) ; }
[ \t\n]       { }
.                 { printf("Line %d: Unknown character: ’%s’\n",yyline,yytext);
}

%%
```

C.3 Syntactic grammar for SAOL in yacc

```
%start orcfile
%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
                | proclist
                | /* null */
                | error
                ;
instrdecl       : INSTR IDENT LP identlist RP LC vardecls block
                | error
                ;
opcodedecl      : optype IDENT LP paramlist RP LC opvardecls block RC
                | error
                ;
globaldecl      : GLOBAL LC globalblock
                | error
                ;
templatedecl    : TEMPLATE LT identlist GT LP identlist RP
                   MAP LC identlist RC
                   WITH LC mapblock RC LC
                   vardecls block
                | error
                ;
mapblock        : mapblock COM LT terminal_list
                | LT terminal_list GT
                | /* null */
                | error
                ;
terminal_list   : terminal_list COM
                | terminal
                | error
                ;
terminal        : IDENT
                | const
                | STRCONST
                ;
globalblock     : globalblock globaldef
                | /* null */
                | error
                ;
globaldef       : rtparam
                | vardecl
                | routedef
                | senddef
                | inputmoddef
                | seqdef
                | error
                ;
rtparam         : SRATE INTGR SEM
                | KRATE INTGR SEM
                | INCHANNELS INTGR SEM
                | OUTCHANNELS INTGR SEM
```

```
               | error
               ;
routedef       : ROUTE LP IDENT COM identlist RP SEM
senddef        : SEND LP IDENT SEM exprlist SEM identlist RP SEM
               | error
               ;
inputmoddef    : INPUTMOD LP IDENT SEM exprlist RP SEM
               | error
               ;
seqdef         : SEQUENCE LP identlist RP SEM
               | error
               ;
block          : block statement
               | /* null */
               | error
               ;
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
               | SFSYNTH LP exprlist SEM identlist SEM 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

               | error
               ;
lvalue         : IDENT
               | IDENT LB expr RB
               | error
               ;
identlist      : identlist COM IDENT
               | IDENT
               | /* null */
               | error
               ;
paramlist      : paramlist COM paramdecl
               | paramdecl
               |   /* null */
               | error
               ;
vardecls       : vardecls vardecl
               | /* null */
               | error
               ;
vardecl                : taglist stype namelist SEM
               | tabledecl SEM
               | TABLEMAP IDENT LP identlist RP SEM
               | error
               ;
opvardecls     : opvardecls opvardecl
               | /* null */
               | error
               ;
opvardecl      : taglist otype namelist SEM
               | tabledecl SEM
               | error
               ;
paramdecl      : otype name
               | error
               ;
namelist       : namelist COM name
               | name
               | error
               ;
name           : IDENT
               | IDENT LB INTGR RB
               | error
               ;
stype          : IVAR
               | KSIG
               | ASIG
               | TABLE
               | OPARRAY
               | error
               ;
otype          : XSIG
               | stype
               | error
               ;
tabledecl      : TABLE IDENT LP IDENT COM exprstrlist RP
               | error
               ;
taglist              : IMPORTS
               | EXPORTS
               | IMPORTS EXPORTS
               | EXPORTS IMPORTS
               |
               | error
               ;
optype         : AOPCODE
               | KOPCODE
               | IOPCODE
               | OPCODE
               | error
               ;
expr           : IDENT
               | const
               | IDENT LB expr RB
               | IDENT LP exprlist RP
               | IDENT LB expr RB LP exprlist RP
               | expr Q expr COL expr
               | 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
               | error
               ;
exprlist       : exprlist COM expr
               | expr
               | /* null */
               | error
               ;
exprstrlist    : exprstrlist COM expr
               | exprstrlist COM STRCONST
               | STRCONST
```

```
                | expr
                | error
                ;
const           : INTGR
                | NUMBER
                | error
                ;
%%
```

Annex D Detokenisation (informative)

This Annex describes the process of converting a tokenised bitstream representation into a human-readable program in the textual SAOL format. It is provided for informative purposes only. The result of this process is much more satisfying if the bitstream contains the optional symbol table element, see Subclause 5.1.

[Needs to be completed or removed].

Index to Subpart 5

Page numbers in **boldface** refer to definitions in the Glossary, Subclause 5.0.3.