External Documentation and Release Notes for *saolc*

Eric D. Scheirer
MIT Media Laboratory
eds@media.mit.edu

15 June 1999
saolc version 1.0

Abstract

The **saolc** package is the reference software for the Structured Audio part of the MPEG-4 Audio standard (ISO 14496-3 Section 5). **saolc** provides non-real-time decoding of Structured Audio bitstreams, and demonstrates the proper functioning of a normative SA decoder. The structure of **saolc** is documented for implementors who wish to make use of the reference software, beginning at a high-level overview and proceeding to a list of important data structures and a module-by-module description. Bugs, extensions, and areas of non-conformance to the paper specification are documented. This documentation augments the internal documentation provided by comments in the code.

Contents

1. Introduction

1.1. Scope and Purpose

This document provides external documentation for **saolc**, the MPEG-4 reference software for the Structured Audio tools in ISO/IEC 14496-3, the MPEG-4 audio standard. It is not a tutorial on creating bitstreams or making music with Structured Audio, but technical documentation to assist developers in making use of the reference software.

Other resources for developers of tools using MPEG-4 Structured Audio tools include:

- The MPEG-4 Structured Audio homepage at http://sound.media.mit.edu/mpeg4
- The SAOL developers mailing list; send email to saol-dev-request@media.mit.edu to subscribe
- Publications in the technical literature (see Section 9.2 of this document)
- The MPEG-4 Audio Standard, ISO/IEC 14496-3, available from the International Standardization Organization at http://www.iso.ch

The functioning of **saolc**, and the concepts in the Structured Audio part of the MPEG-4 standard, are very different from those in other audio decoding standards. **saolc** performs many of the functions of a high-level language interpreter, and of a software synthesizer, as well as those of an audio decoder. Readers whose primary expertise is in traditional audio coding may find it difficult initially to grasp all of the concepts in the SA standard and software; some suggestions for background reading are provided in Section 9.3.

This document and the **saolc** code are not meant to serve as a substitute for the MPEG-4 standard text. In particular, this document assumes that the reader has a general familiarity with the scope, requirements, and functionalities of the MPEG-4 Structured Audio standard. Readers with no previous exposure to the standard should begin with the technical papers in Section 9.2 and then skim through the standard before trying to understand the code.

1.2. About saolc

saolc is the reference software for the MPEG-4 Structured Audio standard (Section 4 of ISO 14496-3), henceforth called SA. It is not intended to be a user-friendly tool for creation or playback of MPEG-4 bitstreams. It serves the role of providing a demonstration of how to correctly implement all aspects of the SA toolset in an MPEG-4 decoder. Many optimizations, even obvious ones, have not been applied to this software, in order to preserve maximum clarity in the coding style. It is expected that future publications in the technical literature and source-code contributions by the SA community will examine the role of optimization in SA implementation more fully.

It is important to understand that **saolc** is not SAOL. SAOL is a computer language for the description of sound, standardized in MPEG-4. **saolc** is one implementation of that language, created to serve as a reference for the standard. Nearly every functionality of SAOL can be properly implemented in a number of different ways that all produce the same result. **saolc** does not act to restrict the manner of operation of a normative terminal, but serves as one example of a properly-implemented SAOL decoder.

saolc was implemented in conjunction with the development of the SA standard, roughly Mar 1997 – May 1999. The primary developer was Eric D. Scheirer from the Machine Listening Group of the MIT Media Laboratory, but there were a number of other contributing developers, see Section 9.4.

saolc totals approximately 40,000 lines, about one MByte, of C and C++ code. It is widely portable and has been successfully tested (as of June 1999) on win32, DEC Alpha, SGI, PowerMac, Linux, OS/2, and DECStation platforms. The only major porting issue is the availability of the C++ standard template library – the **vector** template is needed to compile the bitstream-processing code. It has been extensively checked for word-size and word-order portability. See Section 7 for a description of modules known to be problematic to compile on various architectures.

saolc includes extensive internal documentation (comments) as well as this external documentation. The 'hard parts' of the standard are internally documented more carefully than the 'easy parts'.

Every effort has been made to have a robust and stable platform for the reference software; however, it is expected that in a software effort of this magnitude, bugs and areas of non-conformance will remain. Bug reports and other offers to help clarify issues in the software are always appreciated, and can be sent to **saol-bugs@media.mit.edu**. The current list of known bugs and other issues is in Section 7.

1.3. saolc licensing

The official reference-software release of the **saolc** code is under the licensing terms demanded by MPEG and the ISO. Thus, each source code file in the reference-software release begins with a license statement asserting that the MIT Media Laboratory *et al.* are the owners of the software and release it to ISO only for the purpose of developing tools complying to the MPEG-4 standard. If your copy of the code has these license statements, *you must not remove the license statements or distribute the code.* Only the ISO may distribute code under this license.

In addition to the code released to ISO, the vast majority (more than 95%) of the code in **saolc** has been released into the public domain by its developers. The most recent public-domain version of the code, which contains no code licenses, can be downloaded from the Structured Audio homepage noted in Section 1.1. This code release is performed as a service to the computer-music and SA communities. However, it is important to note the following:

> ***Only the reference software purchased from ISO as part of ISO/IEC 14496-5 has official MPEG status. The public-domain source release may be useful in other ways, but it is not part of the reference software for ISO 14496-3. For the construction of fully-compliant MPEG-4 tools, it is imperative to understand the differences, if any, between the reference software in ISO/IEC 14496-5 and various open-source versions of similar software. The public-domain software must not be used for testing conformance of an MPEG-4 application that is desired to be fully compliant.***

For the code in the public-domain release, there are no restrictions placed on its use. It may be used by any individual or organization for any personal or commercial purpose. The code in the public-domain release has no licensing statement at the beginning of the source modules.

1.4. About this document

This document is structured as follows. Section 2 provides brief man-page style usage instructions for **saolc**. Section 3 gives a general overview of the structure and functioning of **saolc**. Section 4 provides definitions of implementation-specific terms for concepts in **saolc**. Section 5 provides a module-by-module overview of **saolc**, going into relationships between modules and their primary functions. Section 6 discusses the relationship of **saolc** to other reference software in ISO/IEC 14496-5. Section 7 presents bugs and other known issues with **saolc**. Section 8 describes the **saenc** tool for creating MPEG-4 Structured Audio bitstreams that is included in the **saolc** package. Section 9 provides references, credits, and a bibliography for introductory and further reading.

In many places in this document, the MPEG-4 Standard is used as a reference to explain or document some point. If not otherwise stated, references of the form "Clause 5.X" or "Subclause 5.X.X.X" are references to the MPEG-4 Structured Audio standard, ISO/IEC 14496-3:1999 Section 5.

2. saolc usage

2.1. Introduction

This section provides usage instructions for **saolc** – that is, how to use the **saolc** executable to decode MP4 bitstreams and otherwise perform synthesis.

2.2. Command-line syntax

saolc is run from the command line with the following syntax:

`saolc –orc saolfile [options]` or

`saolc –bit bitstream [options]`

where `[options]` are any of:

```
-sco saslfile
–midi midifile
–verbose
–text
–out soundfile
–in infile
–temp directory
–iq quality
–sbank0 sasbf
–sbank1 sasbf
–segfault
–version
```

At least one of `–midi`, `-sco`, `-bit` or `–in` must be provided.

The options are interpreted as follows:

`-bit bitstream` : Decode the MP4 bitstream given in the file (normative operation).

`-orc saolfile` : Use the given file as a SAOL orchestra in the textual format.

`-sco saslfile` : Use the given file as a SASL score in the textual format.

`-verbose` : Generate extra warnings and debugging information.

`-text` : Dump the output samples to **stdout** as floating-point numbers.

`-out soundfile :` Make an AIFF soundfile with the output sound.

`-in infile` : Use the given AIFF soundfile as input. This option may be used multiple times, in which case the multiple input files are used as multiple channels of input.

`-temp directory` : Place temporary files created by bitstream processing in the given directory.

`-iq quality :` Use the specified quality for "high-quality" interpolations (default is "3", higher values give better sound quality but compute more slowly).

`-sbank0/1` : Use the given DLS-2 bank files as SASBF banks in the orchestra.

`-segfault` : Dump core or cause GPF on run-time error (useful for debugging).

`-version`: Print out the version (revision number) of **saolc**.

Only the `–bit` option describes a normative mode of operation for the decoder. The other modes are provided for convenience, but other implementations are not required to support textual decoding, or processing input sounds from the command line (see also Section 7.5).

The C preprocessor or other text-munging utility can be used to preprocess SAOL files in conjunction with the `–orc` command. If the environment variable `SAOL_CPP` is set to the name of the preprocessor (for example, "`/usr/sbin/cpp`") when **saolc** is executed, the command "`$SAOL_CPP < saolfile > temp.sao`" is executed before SAOL decoding, and then `temp.sao` is used as the orchestra input.

3. Structure of saolc

3.1. Introduction

This section describes the overall structure and function of **saolc**. All readers should begin with this section in order to gain an understanding of the general operation of the decoder/interpreter. Occasional reference will be made to specific modules and functions, and an inclusive list is provided in Section 5. Section 3.2 provides a "top-down" perspective on the construction and flow-of-control of **saolc**. Section 5.30 describes the key data structures that are used to represent the decoder status and keep track of the scheduler.

There are many ways in which other compliant implementations might differ from **saolc**. An implementation that compiles code onto a DSP for direct execution would necessarily have much more preprocessing (since it would have to perform code generation and optimization as well as parsing and syntax checking), but much less run-time code, since the run-time code would be an external library linked with the bitstream SAOL algorithms. **saolc** doesn't really live in a streaming environment; it pretends to expose an interface to a network with the `receive_au()` function in `sa_bitstream.cpp`, but this function only reads data out of a file. In a real streaming decoder, the interface to Systems would be more complex, and the error checking and real-time demands much higher.

The main core of **saolc** is implemented in ANSI C. Some contributed components—the bitstream-processing functions developed by Columbia University, the SASBF synthesizer contributed by Creative, and the DLS-format parser contributed by Microsoft—were originally written in C++. Therefore, the overall system is in integrated C and C++. In practice, there is relatively little exposure and interconnection among these pieces, so the language integration is not difficult.

3.2. Basic flow-of-control

Figure 1 shows a schematic overview of the major modules and data structures in **saolc.** This flowchart shows only the normative operation of **saolc**, in which it reads a disk file containing bitstream data. **saolc** also implements standalone modes of operation, in which it reads orchestra and score files directly from disk. These modes will be described in Section 7.5.

3.2.1. Decoder configuration

The main program (`saol_main.cpp`) executes initial session startup, and then calls the bitstream parser to deal with the decoder configuration header (`process_bitstream()` in `sa_decode.cpp`). `process_bitstream()` uses the fundamental bitstream-parsing routines in `sa_bitstream.cpp` to read the bitstream (from a file) and dump textual orchestra files, score files, Standard MIDI files, samples, and SASBF blocks to disk space for temporary storage.

After the decoder configuration header is parsed, the SAOL parser reads in the orchestra file, tokenizes it, and builds a syntax tree. *Tokenization* is the term for converting the sequence of characters in the orchestra file into a structured sequence of keywords, variable names, punctionation marks, etc. The tokenization is performed with a *lexer* (lexical analyzer) written in the **lex** language. **lex** is a special tool that reads a lexical language description and generates C code. The **lex** code for SAOL – that is, the formal lexical description of **saol** – lives in `saol.lex`, and the **lex** tool generates a C code file called `saol.yy.c`. This source file is very difficult to understand (it consists mostly of automatically-generated jump tables) and should not be modified by hand. If it is necessary to modify the lexer, `saol.lex` should be modified and recompiled with **lex**. **lex** is available as a standard utility on all Unix platforms, but is not needed to build **saolc** unless `saol.lex` is modified.

The syntax tree is built from a parser written in the **yacc** language. **yacc** is a special tool, often used in conjunction with **lex**, called a *parser-generator*. **yacc** accepts a formal syntactic description of a programming language in a format similar to the Backus-Naur format used in the Structured Audio standard to describe SAOL, and produces as output C code that parses that programming language. The **yacc** code for SAOL lives in `saol.yacc`, and is used to generate a C code file called `saol.tab.c`. Like `saol.yy.c`, this file is difficult to understand, and should not be modified by hand. Note that when **saolc**

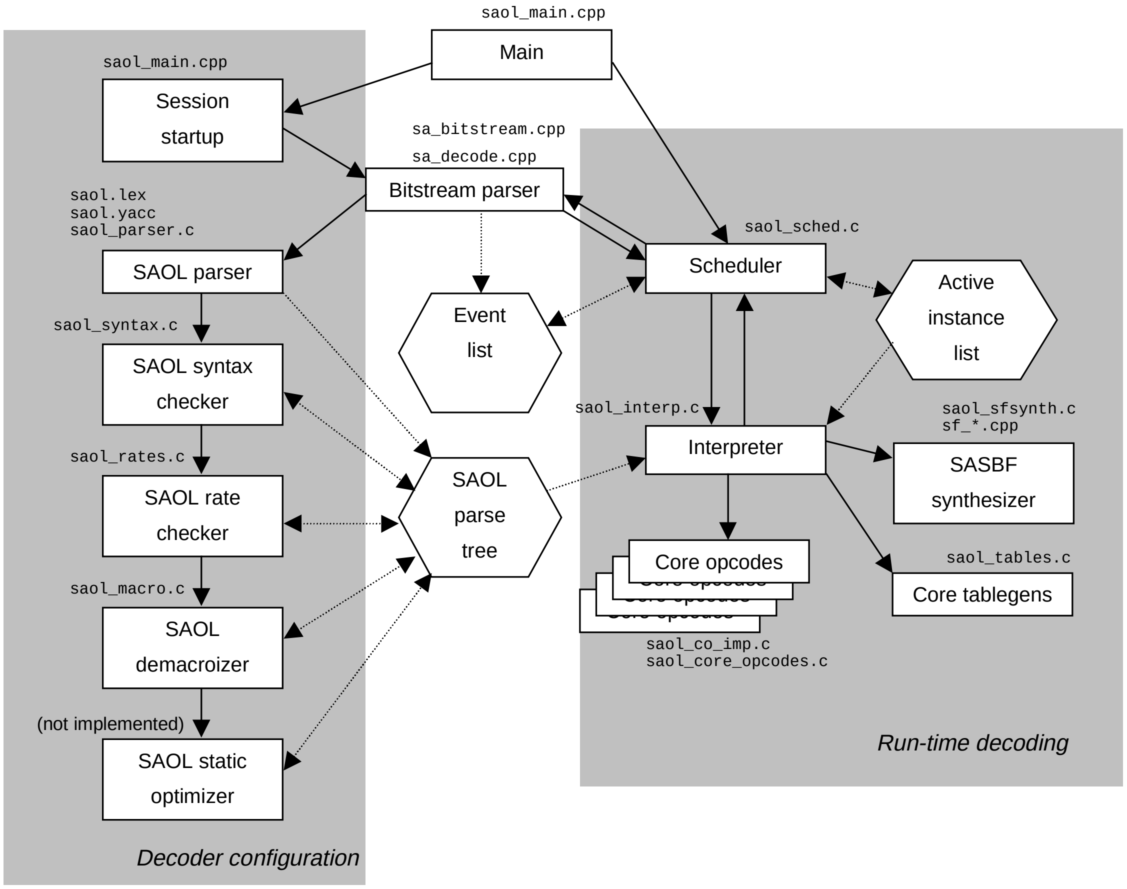

Figure 1: Basic flow-of-control in **saolc**. Rectangular boxes represent modules, hexagonal boxes represent data structures. Solid arrows represent program flow and module-to-module connections; dashed arrows represent reading and writing of data. On the left, the decoder configuration process executes at session startup to create the SAOL parse tree, which is the data structure that represents the orchestra code. On the right, the run-time decoding process manages a continuing interaction between the scheduler and the interpeter. The interpreter reads the parse tree, and makes use of the various core opcode implementations, the table generators, and the SASBF synthesizer as needed.

is built, `saol.yy.c` is `#included` rather than linked with `saol.tab.c`. (Also, the code generated by many versions of **lex** and **yacc** will generate compiler warnings under an ANSI compiler; these should be ignored, as they are harmless and very difficult to fix).

The **yacc** parser uses auxiliary functions that live in `saol_parser.c` to build the syntax tree. The auxiliary functions do simple things like allocate data structures and link them together properly.

After the syntax tree is constructed, it is *syntax-checked* to ensure that opcode calls have the right number of parameters, user-defined opcodes actually exist in the orchestra, and so forth. This module (`saol_syntax.c`) also performs the important function of building the *symbol table* for each instrument and user-defined opcode. The symbol table is a list of all the variable names that are used in the instrument, their rate type, and their array size. The next module (`saol_rates.c`) checks all of the statements and expressions in the orchestra to make sure that the rates of the variables match up – that is, that `asig` variables are not used as parameters to `kopcode` opcodes, and other sorts of rules described in Clause 5.8 of the standard.

The final step of the orchestra pre-processing is to *demacroize* the user-defined opcodes. Since user-defined opcodes may not be recursive (Clause 5.8.6.7.6), it is always possible to implement them as macro-expansion with appropriate renaming (although of course it is not required to implement them this way).

The code in `saol_macro.c` performs this function, thereby removing all of the user-defined opcodes from the orchestra (the `-verbose` flag may be used to view the demacroized orchestra before synthesis). This step makes direct interpretation of the SAOL code easier, since procedural frames don't have to be managed. The demacroization step requires careful renaming and management, especially in the handling of opcode arrays, so the code in `saol_macro.c` is somewhat complex.

At this point, static optimization of the SAOL code could be performed with standard techniques as found in the literature. The only optimization performed in **saolc** is the removal of variables from the symbol tables of instruments (`remove_unused_symbols()` in `saol_syntax.c`) in which they are not used; this speeds up synthesis a lot for some instruments.

If there are score files or MIDIfiles in the bitstream header, they are parsed by functions in `saol_score.c`. Each MIDI event or line of the score file becomes an *event*; all of the events are stored in an event list that is managed at run-time by the scheduler.

3.2.2. Run-time decoding

Run-time decoding switches control back and forth between `saol_sched.c` and `saol_interp.c`. The general breakdown is that `saol_sched.c` contains functions dealing with the scheduler and other issues pertaining to the whole orchestra, while `saol_interp.c` contains functions dealing with synthesis of individual notes. Most of the normative scheduler description in Clause 5.7 is contained in `saol_sched.c`, while most of the semantics of the expressions and statements of SAOL in Clause 5.8 is contained in `saol_interp.c`.

`saol_sched.c` calls `receive_au()` in `sa_decode.cpp` to "receive" access units (AUs) from the systems layer, which is implemented by simply reading them from the MP4 file. The AUs contain *events* which are registered into a time-sorted event list. At each time step, the scheduler deals with any events that are ready (according to their timestamps) to be dispatched. The events do things like start up new instrument instances (`new_instr_instance()` in `saol_interp.c`), destroy existing ones, or control exposed variables. At each time step, for each active instrument instance, the scheduler calls `run_kacycle()` in `saol_interp.c` once at the k-rate, and several times at the a-rate, to do the synthesis itself.

`run_kacycle()`, in `saol_interp.c`, calls `eval_block()` as the main point of entry to SAOL interpretation. `eval_block()` and `eval_expr()` walk down the SAOL parse tree that was created at decoder configuration time to calculate the output of an instrument instance. `eval_block()` and `eval_expr()` are mutually-recursive; `eval_expr()` in particular is highly recursive, as is the nature of a recursive-descent interpreter. `saol_interp.c` makes calls to functions in `saol_co_imp.c` ("SAOL core-opcode implementations") and `saol_tables.c` to execute core opcodes and generate wavetables. If the **sasbf** expression is used by the orchestra, `saol_interp.c` calls `sbsynth()`, in `saol_sbsynth.c`, to perform SASBF sample-bank synthesis. `sbsynth()` calls functions in the SASBF synthesizer, which lives in files named `sf_*.cpp`.

The output of the orchestra is managed by `instr_output()` in `saol_sched.c`. Each instrument instance calls this function to add output to the global output buffer, unless the special bus **output_bus** is used (Clause 5.7.3.3.6 list item 11). At the end of each k-cycle, the global orchestra output is calculated and dumped to a file or to **stdout**. Hooks are in place to provide real-time output using the function `soundOutQueue()`, but this function is not currently provided. To make real-time output to the DAC on a particular platform, implement the `soundOutOpen()`, `soundOutClose()`, and `soundOutQueue`() functions as noted in the code. And either optimize **saolc** a lot, or have a really fast computer.

4. Definitions

This section contains some definitions useful for understanding the discussion elsewhere in this document. Also see Clause 5.3 of the standard.

Context: The context of an instrument instance is the current values of all the variables in that instance, and the current state of all of the opcodes called by that instance.

Event: An event is one of the instructions delivered in a score or MIDI file (or streaming score or MIDI event) that controls the decoder by asking for a new note or changing a control parameter.

Handle: A handle is the data structure that keeps track of all the information about an active instrument instance. This includes the context, the label (which allows the instance to receive control events), the MIDI channel the note is on if any, the amount of time the note has been executing, and some other things. Handles are kept in a list of active instrument instances that is sorted by the execution sequence order of the instruments.

Instrument instance: An instrument instance is the data keeping track of one “note” of synthesis. It is created in response to a score **instr** event or a MIDI **noteon** event.

Note: Another name for an instrument instance.

Parse tree: The data structure that organizes the SAOL code in a way easily interpreted. The first step in decoding is to convert the SAOL orchestra into a parse tree through lexical analysis (Section 5.11) and parsing (5.10). During run-time, the interpreter looks at the parse tree to figure out what the code says.

Terminal: (1) The hardware/software system that is doing the decoding. (2) A leaf of the parse tree; that is, a variable name or numeric constant.

5. Module-by-module description of saolc

5.1. Introduction

This section contains a short description of each of the code modules making up **saolc**. For each, the basic purposes of the functions contained in that module are described. Where appropriate, specific functions are highlighted and described briefly. For functional descriptions at a finer level of detail, please refer to the comments in the code itself. The author of each module is credited here (see Section 9.4 for a complete list) – uncredited modules are by Eric Scheirer of the MIT Media Laboratory.

The modules are listed here in ASCII alphabetical order. See Section 3.2 for a more functional description of the connections between many of the modules. `saol_main.cpp` contains the top-level `main()` function.

Not every module listed here is available for public release. Thus, some of the modules here are missing in the public-domain source release and are available only through the ISO (see Section 1.3).

Following the list of code modules, the final part of this section contains a description of the important data structures used in **saolc**.

5.2. aifif.c

`aifif.c` contains code that reads and writes AIFF audio files. It is used to read sample data and input data from disk files, and to write out the final output of the orchestra if the `-out` option is used.

This module was written by Dan Ellis and was stolen from the public Csound code for use in **saolc**.

The functions here are used mostly by the main scheduler loop in `saol_sched.c` as it pulls in data from input files and writes out the output.

5.3. bitstream.cpp

`bitstream.cpp` contains the methods for the `Bitstream` class. It was contributed by Alexandros Eleftheriadis and his colleagues at Columbia University. `Bitstream` is used to hold the information about the binary bitstream file; its methods are used for reading and writing the bitstream format. Important public methods are the constructor (`Bitstream::Bitstream()`), which opens the bitstream file for reading or writing, `getbits()` and `putbits()`, which get or put a fixed number of bits from or to the bitstream file, and `getfloat()` and `putfloat()`, which get or put floating-point values from the bitstream.

The `Bitstream` class is used by the classes in `sa_bitstream.h` to read and write bitstream elements of the SA bitstream. The distinction between these modules is that the `Bitstream` class is low-level and doesn't know that this is an SA bitstream; the `sa_bitstream` classes are semantically tied to the elements of the SA bitstream as described in Clause 5.2.

There is sometimes not enough error-checking in these functions; this can cause mysterious bugs in the following way. Since you can't really put 1 bit at a time into a file, the inner workings of `putbits()` buffer up data until full bytes are stored, and then put out these full bytes. The buffering is done with bitwise operations. So sometimes invalid bitstream method calls can have "pre-causal" effects; that is, to affect bits *earlier* than their target. So for example:

```
bit.putbits(0x00, 1);  /* data, number of bits */
bit.putbits(0x00, 1);
bit.putbits(<garbage>, <garbage>);  /* bad call */
```

When we inspect the binary data generated by this sequence, we find that the first bit is 1 rather than 0 as we expect (we expect it to start 00... in binary). In trying to debug this problem, we naturally focus on the first call, since this is the call that "generates" the first bit. We become mystified (speaking from experience), since this call appears to be fine. In fact, it *is* fine; the problem is that since the buffer wasn't flushed between the second and third calls, the bad third call corrupted the whole buffer, leading to the error. (The error is not that the buffer wasn't flushed—if we flush the buffer after the second call, we get a whole bunch of extra 0 bits to round out the incomplete byte). The moral is to, when debugging bitstream weirdness, look at all of the calls to `putbits()` in the region surrounding the error before becoming too paranoid about single-stepping the code. The debugging `printf()`s commented out in `putbits()` and `getbits()` are useful for this purpose.

5.4. byteswap.c

`byteswap.c` was contributed by Dan Ellis as part of the Csound code. It is used to byteswap data as it is read out of AIFF files into the proper format for the local architecture. This should be integrated somehow with byteswapping in other parts of the code, really. It is called by functions in `aifif.c` and isn't of general interest.

5.5. fft.c

`fft.c` was contributed by Dan Ellis; it was borrowed from the Csound public source. The code in `fft.c` calculates the DFT and IDFT using the Fast Fourier Transform. It is called only by the implementations of the core opcodes **fft()** and **ifft().** This code may be buggy for non-power-of-2 FFT size, but that's the only case required in SAOL anyway.

5.6. fx_picola.c

`fx_picola.c` was contributed by Naoya Tanaka from Matsushita. It implements the PICOLA speed-change function described in Annex 5.D. PICOLA is one possible method for implementing the **fx_speedc()** and **speedt()** core opcodes. It is not the only permissible method; there is a fair technical literature on different methods for implementing speed-change.

The description in Annex 5.D of the standard provides more detail on the functioning of the PICOLA algorithm.

This code is currently only partly-functional, and is not included in the public-domain software release.

5.7. IEEE80.c

`IEEE80.c` was written by Bill Gardner for the Csound public code. It provides extra functions to `aifif.c` – for some reason, the AIFF format has a couple of fields (sampling rate, in particular) that must be coded in the weird IEEE-80 floating-point format, and it takes a little bit of work to get and put these data properly.

5.8. sa_bitstream.h

`sa_bitstream.h` contains all the class definitions for the SA bitstream format, which is defined in Clause 5.2. The basic structure of this code looks funny, because the first version was automatically generated by the **flavor** tools donated to MPEG by Columbia University. This is why the variable names are weird in some of the functions. Most of the recent modifications were done by hand since the SA format is fairly simple. The Flavor code used to define the format in Clause 5.2 should generate similar code to what's there now; the important difference is in the way multiple data blocks within a chunk (for example, multiple score lines in a chunk) are managed. As of **flavor** v.2, this generated fixed arrays which are hard to manage (these are still left for some of the classes) – the fixed arrays were changed by hand to **vector**s for better dynamic management.

Each class declared in `sa_bitstream.h` represents one syntactic element of the SA bitstream format. The order of elements reflected Clause 5.2 – the simple (low-level) elements like tokens are at the beginning, and the important high-level elements like the decoder configuration header are towards the end. Each class has a `get()` method and a `put()` method that call the bitstream methods in `bitstream.h` to get and put that element from and to the bitstream file.

The classes are nested like complex structures, not inherited like objects. That is, an `orc_file` object contains a whole bunch of `orc_token` objects that represent the sequence of tokens in the orchestra. In turn, one or several `orc_file` objects are contained in a `SA_decoder_config` object.

The `put()` methods are used by **saenc**, the simple SA encoder included in the **saolc** distribution; see Section 8.

5.9. sa_decode.cpp

`sa_decode.cpp` is the top-level interface to the SAOL bitstream. This module contains the functions that the main decoder calls to get information from the bitstream. It instantiates `SA_decoder_config` and `SA_access_unit` objects as defined in `sa_bitstream.h`, and uses their `get()` methods to fill them up with data.

There are two main exposed interfaces. `process_header()` reads in the decoder configuration header and then dumps out all the data to temporary files so that it can be read and parsed (it would naturally be possible to parse directly from the bitstream data, but **saolc** doesn't choose to do that). `receive_au()` reads in all the Access Units that have "arrived" at a certain time (as marked by their Decoding Time Stamps).

We always keep the next AU cached if there are any left. Thus, at decoder configuration, we grab the first AU and cache it. When we process AUs in `receive_au()`, we first check to see if it's time to deal with the cached AU. If so, we deal with that AU (which means creating a scheduler event to do whatever it says), then get the next one and see if its time to deal with that one too. If so, we loop until we get one that isn't ready to be decoded. Then we cache that one and return. This works since the AUs have to be in order (since conceptually, they're coming in on the wire).

The auxiliary functions figure out how to deal with the events in each AU.

It is important to note that the AU format read by **saolc** contains simple packaging (just an extra time-stamp at the beginning) to hold the DTS, which is properly part of the systems syntax. Sooner or later, the MP4 file format will be supported and then the AU parsing will be fixed. The file format read by **saolc** is not the official MP4 file format – it's much simpler than MP4, which contains lots of packaging intended to make editing and manipulating the file useful.

5.10. saol.tab.c / saol.yacc

`saol.tab.c` is automatically created by the **yacc** tool, see Section 3.2.1. It's really hairy and you shouldn't deal with it by hand unless you're really sure of yourself.

The input file to **yacc** is `saol.yacc.` This file contains the SAOL grammar in the **yacc** format. It is an expansion of the one included in Annex 5.B. Most importantly, it contains a lot of added `error` productions that help to locate syntax errors in the SAOL code.

The individual "paragraphs" in `saol.yacc` correspond approximately to the BNF grammar for SAOL as broken out in Clause 5.8. Each paragraph describes the syntax of one element of SAOL, and how to create the parse tree for that element. To take one example:

```
statement    : lvalue EQ expr SEM {
    statement *st;
    long i;
    st = new_statement(EQ,ptr_index($3),NULL);
    set_statement_lvalue(st,ptr_index($1));
    i = add_ptr_index(st);
    $$ = i;
    }
```

This is the code that corresponds to the production

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

which is in Subclause XXX. Statements that match this production look like

```
a[p*4] = 3;
q = z;
mysig[midicps(pch) ] = myop[3](midicps(pch),foo);
```

That is, this sort of statement has some expression (the *lvalue*), followed by an =, followed by some other expression, followed by a ; (note the relationship between this description and the first line of the quoted **yacc** code).

The code block following the production (in the { } brackets) describes how to create a parse tree element for this little bit of code. Each of the matching subexpressions is numbered with `$x` in **yacc**. So within the code block, the lvalue expression is denoted `$1`, the = with $2, the right-hand expression with `$3`, and the semicolon with `$4`. When **yacc** converts the **yacc** code to C code, it turns these *metavariables* into regular variables depending on the context.

The rest of the code that follows calls regular C functions (most of which live in `saol_parser.c`) to create the tree from the subelements. This is a inductive process; once we're executing this **yacc** code, we've already executed the code to parse the subproductions (the lvalue and right-hand expressions in this case). So all the code does is make a new statement (with `new_statement()`, in `saol_parser.c`) of type `EQ` using the right-hand expression as a parameter, and then sets the lvalue of the statement to the lvalue parameter. The `$$ = i` line means that the "return value" of this piece of syntax is the value `i`. This is the value that becomes the `$x` value at the next higher induction level.

The `ptr_index()` and `add_ptr_index`() functions are used to keep track of all the subexpressions by number, since **yacc** is most happy passing longs rather than void pointers around. See the comments on these functions in `saol_parser.c` for more details.

The whole parser created by **yacc** from `saol.yacc` generates a single public function, `yyparse()`. `yyparse()` is called by the main function to parse the textual SAOL code. Note that unlike the rest of the decoder, `yyparse()` is not re-entrant (it uses global variables), and so only one parsing process may run at once. The run-time parts of **saolc** are re-entrant, and so once the parsing is complete, multiple decoding processes may be run at the same time.

If you wanted to add new non-normative syntax to **saolc** (for example, a **for** loop construction like the one in C), you'd start by adding any new keywords in `saol.lex`, then add the syntax in `saol.yacc`, then add the parsing functions in `saol_parser.c`, then add rate-checking and syntax-checking in `saol_rates.c` and `saol_syntax.c`, then add interpretation for that statement or expression in `saol_interp.c`. (This is only for modifying the syntax itself – you don't have to go to all this work to add new core opcodes. See Section 5.13).

5.11. saol.yy.c / saol.lex

`saol.yy.c` is automatically generated from `saol.lex` by the **lex** tool. Like `saol.tab.c`, you shouldn't have to edit this file by hand or try to read it.

The **lex** code in `saol.lex` is pretty much the same as the example **lex** code given in Annex 5.C of the standard. Each line in the **lex** code shows how to recognize one of the basic lexical elements of SAOL. So a line like

```
"ivar"        { count(); return(IVAR) ; }
```

says that when the scanner sees the text string "`ivar`" it should call the function `count()` (which just keeps track of lines and characters we've scanned) and then return the `IVAR` value. The enumeration of these values is in `saol.tab.h`, which is automatically generated when `saol.yacc` is converted with **yacc**.

Some of the lines use regular expressions, for example

```
{IDENT}       { count(); yylval = add_ptr_index(strdup(yytext));
                                 return(IDENT) ; }
```

where the `IDENT` macro is defined at the beginning of the file as

```
IDENT         [a-zA-Z_][a-zA-Z0-9_]*
```

Thus, an `IDENT` is a letter or underscore, followed by a sequence of 0 or more letters, digits, and underscores. When an `IDENT` is found, we save a copy of the matched sequence (called `yytext`), return the `IDENT` token, and set the "scanner data" `yylval` to the pointer-index number of the new identifier. All of these funny variables have special meanings in the **yacc** code; to better understand the relationship between **lex** and **yacc**, see a compilers textbook such as the one cited in Section 9.3.

The whole lexer specified by `saol.lex` creates a single function, `yylex()`. `yylex()` is called by `yyparse()` to get lexels from the orchestra.

When `saol.yy.c` is compiled, it is `#included` in `saol.tab.c` rather than linked in with the rest of the code. To make a new Makefile or project for **saolc**, don't try to compile `saol.yy.c` directly into an object file; rather, mark it as a dependency of `saol.tab.c`.

5.12. saol_co_imp.c

This module contains all of the implementations of core opcodes in SAOL. The SAOL core opcodes are listed in Clause 5.9. For each core opcode **xxx()**, there is a corresponding C-code module `co_xxx()`. The core opcodes are called by `saol_interp.c` through the table of pointers `co_ptr()` that's defined in `saol_core_opcodes.c`.

Each core opcode implementation has the same prototype, as follows:

```
co_xxx(sa_decoder *sa, opval *op, actparam *pf, int pf_ct, long rate)
```

`sa` points to the current decoding process; `op` contains all the current values of state variables in the opcode; `pf` contains all of the actual-parameter values; `pf_ct` specifies how many actual parameters there are; and `rate` specifies what rate we're currently working on (i-rate, k-rate, or a-rate). Most of the core opcodes do different things on the i-rate, k-rate, and a-rate passes as specified in Clause 5.9.

Each opcode has *state* that keeps track of the values of all of its variables (for example, the current values of the filter feedback for the **iir()** opcode). Since each opcode is potentially used many times in parallel, we can't use static storage within the opcode implementation to hold this state. Instead, the instruments keep track of the states of each of the opcodes they use. The `op->local` structure for each instance of each opcode is a `(void *)` that gets casted to the storage structure of the opcode. All of the storage structures are defined in `saol_co_imp.h.`

Some of the opcodes need to dynamically allocate more memory. For example, for **delay()** we have to create delay-line memory, and we don't know how much until runtime. The `op->dyn` field is used to hold a pointer to this memory. For a very few (for example, **pluck()**), we need more than one dynamic segment. There's only one handle, so `op->dyn` points to a big block corresponding to everything we need, then we break it up locally and manage it ourselves.

5.13. saol_core_opcodes.c

`saol_core_opcodes.c` contains the main core-opcode table. This table is used by the syntax-checker to make sure that the parameters to core opcodes are at the right rate, by the scheduler to figure out how much memory needs to be allocated for each opcode instance in each instrument, and which function from `saol_co_imp.c` to call to execute each core opcode.

In order to add another non-normative core opcode to the implementation, all you have to do is give it an entry in the opcode table, a storage structure in `saol_co_imp.h`, and an implementation in `saol_co_imp.c`. Everything else is handled automatically.

5.14. saol_interp.c

This module, with `saol_sched.c`, forms the run-time core of **saolc**. `saol_interp.c` contains all of the functions for interpreting SAOL code, handling variables, doing math, and so forth. It communicates with the scheduler, which keeps track of what instrument instances are running, and with the core opcode implementations in `saol_co_imp.c`, which execute the algorithms corresponding to core opcodes.

The top-level functions are `run_itime()` and `run_katime()`, at the bottom. These are the functions that are called by the scheduler to run an instrument instance for its i-rate, k-rate, or a-rate execution. `run_itime()` updates ("pushes") the context of the instrument instance by looking up the values of all the global variables and controllers the instrument needs, creates any local wavetables in the instrument, calls `eval_block()` to execute the instrument, and then copies out the values of exported variables ("pops the context").

`run_katime()` is used to run an instance at either the k-rate or the a-rate. For the k-rate execution, it pushes the context, updates the local time of the instrument (the **itime** standard name), calls `eval_block()`, and pops the context. For the a-rate execution, it calls `eval_block()` multiple times, once for each a-cycle in the k-cycle, and updates the output sample pointer for each. That is, the value of the expression in an **output** statement on the first iteration is the first sample of output in the k-cycle, the value on the second iteration is the second sample of output, and so on.

`eval_block()` is the central function of the interpreter. It evaluates a block of code (a sequence of statements) at a particular rate according to the rules in Subclause 5.8.6.6. This is just a big **switch** statement depending on the type of the statement. For **if, else,** and **while** statements—which themselves contain blocks of code—`eval_block()` is called recursively. `eval_block()` makes heavy use of `eval_expr()`, which recursively descends the expression tree for an expression and evaluate it. At the bottom of the expression tree, we sometimes reach a variable name (tagged **IDENT** in the code), at which point the function `get_var_value()` is used to get the value of variable given the instrument context.

When we find a core opcode in the code (more properly, an opcode expression or oparray expression referencing a core opcode), we call `eval_opcode()` or `eval_oparray()` as needed. Each of these functions evaluates the parameter expressions, looks up the proper core opcode storage structure in the context, and then calls the proper function in `saol_co_imp.c`. There are no user-defined functions left in the code during interpretation—they were removed during macro expansion (Section 5.15).

There are a number of auxiliary functions in this module that are relatively straightforward.

5.15. saol_macro.c

`saol_macro.c` is the central module of the preprocessor. It is also poorly written, kludgy, and probably impossible to understand. It should really be rewritten.

Since SAOL opcodes may not be recursive or mutually-recursive (Subclause 5.8.6.7.6) it is always possible to convert instruments that call one or more user-defined opcodes into instruments that do not, by macro-expanding the user-define opcode code into the instrument. The advantage of this is runtime efficiency, since a-rate context switches may add 30% or more overhead to the execution (in fact, **saolc** runs twice as fast as the version that used context-switching did). The disadvantage is ease of debugging and straightforwardness of the implementation. `saol_macro.c` is the only major part of **saolc** that is optimized a little for speed instead of for maximum clarity.

The process of expanding user-defined opcodes (UDOs) as macros is much more complex than the C preprocessor macro functionality. This is because UDOs:

(a) may contain statically-scoped variables of their own, which have to be renamed during macro expansion,

(b) may reference core-opcode oparrays, global variables and standard names, which act as exceptions to (a)

(c) may return arrays rather than scalar values,

(d) may be used in the SAOL opcode-array construction (Subclause 5.8.6.7.7),

(e) contain a block of statements rather than an expression, and

(f) may contain the **return** statement, which short-circuits the evaluation of the code block.

We have to deal with all of these cases carefully to avoid changing the semantics of the code. This example shows what has to happen. We are given an instrument that makes a call to the UDO `foo()`:

```
instr inst1(p1) {
  asig x;

   x = foo(p1 * 3);
  output(x);
}
```

The UDO has some code:

```
aopcode foo(asig x) {
   asig y;

    y = x * 4;
   return(y * y * 12);
}
```

We want to paste the UDO code into the instrument, so that when it's done, it looks like

```
instr inst1(p1) {
   asig x;
   asig __foo_x, __foo_y;

   __foo_x = p1 * 3;
   __foo_y = __foo_x * 4;
   __foo_rtn = __foo_y * __foo_y * 12;
   x = __foo_rtn;

   output(x);
}
```

This simple example doesn't show any of the interesting problems except (a) and (e).

The top-level function of `saol_macro.c` is `macro_expand()`. It is called by `main()` after the syntax- and rate-checking of the orchestra code is complete. `macro_expand()` converts the orchestra, which contains user-defined opcodes as well as instruments, into a new orchestra that contains only instruments and has no UDOs. `macro_expand()` traverses the list of instruments and calls `macro_expand_block()` for each.

`macro_expand_block()` pastes all the code for the necessary UDOs into a block of orchestra code. For each occurrence of a UDO in a block of code, we do the following:

- (a) make a new variable that corresponds to the **return** statement in the UDO, to hold the dummy return value
- (b) figure out the new names of all the local variables in the UDO
- (c) turn the UDO parameters into assignment statements if they're call-by-value parameters
- (d) add all of the variables from the UDO to the symbol table of the caller
- (e) make a deep copy of the code block that contains the UDO code (we need to save the UDO code in the pristine form since we might have to use it again somewhere else)
- (f) in the new code block:
  1. replace all the local variables and formal parameters with their new names
  2. expand all of the UDOs that are called by this UDO (via a recursive call to `macro_expand_block()`)
  3. add an assignment statement assigning the expression from each **return** statement to the dummy return variable
- (g) add a **jump** flag after each **return** statement to show that it has to jump out of the code block
- (h) paste the munged code block in the place of the call to the UDO
- (i) in the special case where a UDO is embedded in the **while** loop guard, make another deep copy of the munged code block, and add it in at the end of the **while** code block

It is a doubly-recursive procedure. That is, it contains two separate recursive calls, with different purposes, back to itself. The first is the more complex case: at the time we expand a UDO into the instrument code, we have to macro-expand the block of code from the UDO (mentioned as step e.1. above). The second is the simple case: we traverse the list of statements within the code block and expand the UDOs in each statement. Sometimes, while doing this, we encounter an **if**, **else**, or **while** statement. These statements contain ("guard") subsidiary code blocks that must be recursively traversed.

In order to manage the recursion, there are several data structures that keep track of things. The **name_map** array holds all of the renamings as we recurse downward – it shows the old name and the new name. Sometimes the new name is an array-reference (if the UDO was called in an **oparray** expression) even if the old name wasn't. This means that conceptually sometimes we have to deal with multidimensional arrays; they don't exist in SAOL, so we fake them by munging the index expression with the `fake_md_array()` function.

As we recurse downward, we build up new names for variables by adding function names and counters to the front. The `add_name_map()` function figures out the new names, adds them to the name map, and also adds the new names to the symbol table of the instrument.

5.16. saol_main.cpp

This is the top-level module of **saolc**. It contains the `main()` function and a few other high-level functions. The overall function of `main()` is very simple: read in the bitstream and/or standalone files, configure the decoder (by calling `saol_startup()`), and then run the decoder until it finishes.

`saol_startup()` does most of the decoder startup. It does the following things:

- open all the input files
- initialize speed control buffers
- initialize all the MIDI controllers
- if we're doing bitstream processing
  - make temporary names for the bitstream orchestra and score files
  - parse the bitstream header and dump the files to disk (call `process_header()` in `saol_decode.cpp`).
- load a SASBF bank if there is one
- open and parse the orchestra code
- build the symbol tables and syntax and rate-check the orchestra
- macro-expand the user-defined opcodes away (see `saol_macro.c`)
- make the global context (in `saol_sched.c`)
- start the scheduler (in `saol_sched.c`)
- parse the score and MIDI files

The `help()` function dumps out command-line help.

5.17. saol_midi.c

Functions in this module parse MIDI events from MIDI files. This module does not contain the functions that deal with incoming streaming MIDI events (that's in `saol_decode.cpp`) or that dispatch MIDI events during run-time (that's part of the main scheduler loop in `saol_sched.c`). This module interprets a Standard MIDIFile as a sequence of MIDI events, either for decoder configuration in **saolc** or for encoding into streaming format in **saenc** (see Section 8).

For decoding, the top-level function is `parse_midi()`. This function reads through all the events in a MIDI File and creates scheduler events for them. For encoding, the top-level function is `get_next_event()`. This function returns one event at a time to the caller, which is `doStreamMIDI()` in `sa_encode.cpp`. (In the encoder, the macro **_SAENC** should be **#define**d; this macro removes some of the other functions from compilation. These other functions depend on decoder header files, and so it simplifies the encoder executable to not use them.)

The overall function of this module is straightforward; it reads values out of the MIDIfile, byteswaps them if necessary according to the architecture, and then converts the events into MIDI values. Reference to the MIDI specification (see Section 9.3) is useful for understanding some of the trickier bits in the MIDI format, such as the variable-length-quantity.

5.18. saol_parser.c

This module contains a whole bunch of short functions that are called by the parser to construct the parse tree. These functions are all of the nature of "build a new expression node" or "add a parameter to a linked list," so they won't be documented in any great depth. Once you understand the data structures for the parse tree (see Section 5.30), you should pretty much understand what these functions do. The **yacc**-generated code (Section 5.10) makes heavy use of these functions, calling them to recursively build up the parse tree.

Two notes on coding style: if I were to rewrite **saolc** from scratch (it probably needs it!) I would use C++ for the parse tree and make heavy use of the Standard Template Library. The STL wasn't generally available yet when **saolc** was begun in Mar 1997. Further, if you think of the different elements of the parse tree (parameter lists, expressions, etc) as classes, there's not a full encapsulation that provides functions for reading and writing the elements of each class. In most cases, the caller simply accesses the elements itself; to some ways of thinking, this is poor coding style since direct element access should be a private operation.

5.19. saol_rates.c

This module contains functions that rate-check the orchestra. Rate-checking is done after syntax-checking, to insure that the rate-semantics rules given in the specification are followed. An example of the sort of problem we're looking for is:

```
  ivar i;
  ksig k;
i = k * 2; // illegal
```

This is illegal per Subclause 5.8.6.7.12 ("the rate of the binary expression is the faster of the rate of the two subexpressions", making the right-hand side k-rate and 5.8.6.6.2 ("the statement is illegal if the rate of the right-hand sie is faster than the rate of the lvalue.").

We want to search through the code and calculate the rate of each statement and expression, and make sure the rules are being followed. We store the rates in fields of the parse tree since they'll be needed during run-time interpretation.

There are three main mutually-recursive functions that do this: `block_rate()`, `statement_rate()`, and `expr_rate()`. The only hard case is when **opcode** user-defined opcodes are used, in which case the rate of the opcode depends on the context in which it is called. This part of the checking could be simplified if it was better-integrated with the macro-expansion routines.

5.20. saol_sched.c

This module, along with `saol_interp.c`, makes up the central run-time component of **saolc**. It is the main control mechanism of the running Structured Audio orchestra; the actions that it implements generally match those in Subclause 5.7.3.3 of the specification.

The top-level function in `saol_sched.c` is `sched_run_kycle()`. This function is called repeatedly (once per k-cycle) by `main()` until the synthesis process is complete. Each time `sched_run_kcycle()` is executed, it runs the orchestra for one k-cycle as specified in Subclause 5.7.3.3.6. This consists mainly of calling `receive_au()` in `sa_decode.cpp` to add any new events to the main list of "pending" events, dispatching any events scheduled to occur at the current time, and then calling the functions in `saol_interp.c` to execute each active instrument instance for one k-cycle and one set of a-cycles. At the end of the k-cycle, any instrument instances marked for termination are destroyed.

There is a few other important external functions in `saol_sched.c.` `start_scheduler()` performs the tasks required at decoder startup as described in Subclauses 5.7.3.3.5.3 and 5.7.3.3.5.4. This includes opening output files, creating and initializing an instance of the **startup()** instrument, initializing global variables, making global wavetables, and creating and initializing one instance of each instrument referenced in a **send** statement in the orchestra. `schedule_event()` adds a new event to the list of pending events. This is done repeatedly at startup for the scores and MIDI files in the decoder configuration header, and once for each new event as it comes in during the streaming execution of the decoder.

The rest of the functions are utility functions that encapsulate scheduler behavior. `output_sound()` figures out what the final orchestra output is and outputs it to the DAC or to a sound file as requested at the command line. `register_inst()` is everything the scheduler does in response to the creation of a new instrument instance. It creates a handle (Section 5.30.14) for the instance, makes space for the controllers (host variables), puts the handle in the active instance list, and makes a "turnoff" event if the duration of the instance is known. `instr_turnoff()` and `instr_extend()` handle the execution of the SAOL **extend** and **turnoff** statements.

`start_send_instrs()`, called at decoder startup, goes through all of the **send** statements, makes a new instrument instance for each, and executes the new instance at i-rate. `make_global_context()` allocates space for global variable and figures out the sampling rate and control rate of the orchestra if they weren't explicit in the SAOL code. `add_global_table()` makes a table in the global context by allocating space, evaluating the parameters and calling the appropriate table generator. `destroy_inst()` destroys an instrument instance by deallocating the memory and removing the instrument handle from the list of active instruments.

`scale_sched_events()` is called on tempo changes received from the score, from a MIDIfile, or by changes to the **speed** field in an AudioSource BIFS node binding the decoder.

There are several functions used for processing command-line audio input files when the decoder is running in standalone mode. The top-level function for these is `do_input()`. These functions are not needed for normative operation. For normative operation, only the first few lines of `do_input()` are used; they move the input data out of the decoder input buffer (provided by the **AudioFX** node code, see Section 6) into the special bus **input_bus**.

Stubs are provided for implementing real-time audio output (although actually hooking the stubs up hasn't been tested in more than a year.) To provide live output, they should be written to communicate with the system DAC. `soundOutOpen()` should open the DAC; `soundOutClose()` should close it, and `soundOutQueue()` should write a block of samples to it.

5.21. saol_score.c

This module contains functions that parse a SASL score file. This code is used only at decoder startup, when a textual SASL file (conceptually, from the bitstream header) is being read. For each line of the SASL file, one scheduler event is created and scheduled for execution at the proper time.

`parse_score()` is the top-level function and is called by `main()`. This module is very simple; it's essentially all text-processing.

5.22. saol_sequence.c

`saol_sequence.c` is the module that figures out a legal order of execution of the instruments in the orchestra, according to Subclause 5.8.5.6 of the specification. The inputs to this module are the **sequence**, **route** and **send** statements from the global block. At output, the `sa->seq[]` array has been filled such that the `sa->seq[i]` entry contains the ordinal position of instrument number *i*. (Instruments are numbered according to their order in the orchestra).

`make_sequence()` is the topmost call; it is called from `build_sym_table()` in `saol_syntax.c`. It uses an *assertion* model to determine an ordering given the constraints provided by **sequence**, **route**, and **send** statements. This means an alternation of asserting precedence rules and calculating of the transitive closure of these precedences. First, we assert each precedence given in a **sequence** statement. Then we calculate the transitive closure of those precedences. If there is a loop at this point, it is a syntax error per 5.8.5.6.

After this, we assert each precedence given by a **route**/**send** pair if it is not in conflict with the current transition matrix, and recalculate the transitive closure after each. Once this is finished, we have a complete precedence matrix, and `put_sequence()` calculates a legal sequence. This is very simple: at the start, there is at least one instrument with no predecessor. Take one such instrument as the first, and remove it from the set of candidates. Now, among the candidates, there is at least one instrument that has no predecessor. It is the second instrument in sequence, and is removed from the set. This process continues until all of the instruments have been placed in sequence.

If the **–verbose** command-line flag is set, the sequence of instruments is dumped out after it is calculated.

5.23. saol_sbsynth.c

`saol_sfsynth.c` sets things up to pass off to the SASBF synthesizer (see below). It is the point of contact between the SAOL interpreter and synthesizer, and the SASBF synthesizer. The `sfsynth()` function (it has this name for historical reasons) is called every time an **sbsynth** statement is executed in `eval_block()` in `saol_interp.c`.

5.24. saol_spatial.c

`saol_spatial.c` is not yet implemented. It should contain 3-D audio spatialization code. Right now, it has only a stub.

5.25. saol_syntax.c

This is an important module in the decoder configuration. It syntax-checks all of the orchestra code, to make sure that variables are declared and user-defined opcodes actually exist. It also figures out the width of each expression, instrument, opcode, and bus in the orchestra, and builds the global, instrument, and user-defined opcode (UDO) symbol tables.

`build_sym_table()` is the top-level function; it is called by `main()`. It performs the following functions:

- Figure out the number of input and output channels to the orchestra.
- Build the global symbol table and add the global variables, UDOs, and instrument names to it.
- Look for undeclared variables in the global block.
- Figure out the overall sequence of instruments (`saol_sequence.c`).
- Make a symbol table for each instrument and add the pfields and local variables to it.
- Make a symbol table for each UDO and add the formal parameters and local variables to it.
- Check each instrument’s code for undeclared variables and opcodes.
- Figure out the width of each instrument and bus.
- Check each opcode’s code for undeclared variables and opcodes.
- Remove unused variables from the global symbol table and from each instrument.

As we’re checking the code of each instrument, we mark each variable name as we see it used. Then, as the last step, we remove unused variables from the symbol table for efficiency.

Most of the functions in `saol_syntax.c` are pretty obvious in their operation. Notable is `add_sym_table_decls()`, which goes through a list of variable declarations (created by the parser) and adds all the variable names in each declaration to the symbol table. This function (which uses `add_sym_table_namelist()` to process each line) also checks for mismatches between local variables and global variables (an **imports** variable has to be the same type as its global counterpart if there is one).

5.26. saol_tables.c

This module contains the implementations of all of the core wavetable generators from Clause 5.10 of the specification. The mapping from name to function is contained in the **core_tablegens** array; if you want to add nonnormative wavetable generators, you just add the name and function pointer there and update the value of **NUM_CORE_TABLEGEN**. Each table generator is passed a pointer to a **table_storage** structure (Section 5.30.26) and a list of the parameters that the tablegen was called with. It fills up the table structure with data.

The table generators are called during global orchestra startup in `saol_sched.c` and during i-rate execution of an new instrument instance in `saol_interp.c`. The `gen_table()` function looks up the right one by its name and then calls it.

There are a few other utility functions dealing with tables here as well, to check the size of a table, check that a name is really the name of a core table generator, and to allocate space for a new table.

5.27. saol_templates.c

This module exposes the function `deal_with_template()`, which is called from the parser code (Section 5.10) when a template is found. When this happens, the template is immediately replaced by the set of instruments that it defines. The process is quite simple; we make a bunch of instrument data structures and fill them up with deep copies of the data in the template, where we replace the template expressions with their corresponding values from the name map. All this requires is recursive descent through the parse tree with deep copying at the appropriate places.

The deep-copy functions for various SAOL elements are in `saol_parser.c`.

5.28. saol_tok_table.cpp

`saol_tok_table.cpp` contains two important tables: one that matches up SAOL parsing tokens (those generated by the lexer (Section 5.11)) with the corresponding bitstream tokens, and one that matches up bitstream tokens with the corresponding character strings. The first table is used by the encoder **saenc** (Section 8) to generate bitstream tokens from lexical analysis of a SAOL program. The function `lexel_map()` is used for this—it returns the bitstream token corresponding to a particular lexical element. The second table is used by the encoder to turn strings in the input (particularly the names of core opcodes) into bitstream tokens, performed in `is_builtin()`, and by the decoder to turn bitstream tokens back into textual SAOL during detokenization, performed in `tok_str()`.

Neither of these tables is strictly necessary in a SA decoder—it would be more efficient to parse the SAOL code directly from the bitstream sequence rather than go through a detokenization step. They are used here since **saolc** already supports text-mode decoding, and so it is simpler to have only one parser rather than two.

5.29. sf_*.cpp

These files implement aspects of the SASBF synthesizer. They are not currently included in the public-domain release (although the possibility is being investigated). They will be documented better in the future.

5.30. Important data structures

5.30.1. Introduction

saolc, like any high-level language interpreter or compiler, depends on the interaction of a lot of complicated data structures. Most of the data structures in **saolc** are either linked lists or trees. This makes the code simple and straightforward, but less efficient than using structures like hash tables or threaded trees. This section provides an overview of the data structures in **saolc**. Only the most cursory view is presented; the comments in the header files provide a more detailed commentary.

Unless otherwise specified, a structure named **element_list** is a simple linked list of elements of type **element**. The list structures are not highlighted specifically below. There are a few linked list types that directly encapsulate the element data (that is, instead of pointing to an **element** structure, the fields of the **element** structure are incorporated in **element_list**). It is ugly that the two methods are used together; this inconsistancy should be repaired.

The list of data structures is in ASCII alphabetical order by the **typedef** name. Each is also tagged with the header file that defines it.

5.30.2. actparam (saol_interp.h)

An **actparam** structure is an actual parameter, used to pass data from the running code of a note to a core opcode. It contains one of the possible things that can be passed to a core opcode—either a floating point value or a pointer to a wavetable (Section 5.30.26).

5.30.3. block (saol.h)

A **block** structure holds a block of SAOL source code. It is just a linked list of statements (Section 5.30.23).

5.30.4. bustable (saol.h)

The **bustable** structure holds the list of all the busses that are declared in the orchestra. It is a linked-list with the data directly encapsulated in the linking structure. Each node contains information about one bus, associating the name of the bus, its width, the current contents (audio signal) on the bus, and the **send** statement that defined the bus.

5.30.5. cmdinfo (saol.h)

The **cmdinfo** structure holds all of the flags given on the command line (Section 2.2), such as the orchestra filename, bitstream filename, and output filename; the interpolation quality; and whether or not the decoder is running in “verbose” mode.

5.30.6. context (saol_interp.h)

The **context** structure contains all of the state for a note that is running in the orchestra. This includes the following fields:

- **framevals**, which is an array of **frameval** (Section 5.30.10) that contains the current values of all the local variables.
- **sfstorage**, which is the current state of all the SASBF synthesizers under the control of this note.
- **localvars**, which is the symbol table of the instrument that corresponds to this note (Section 5.30.24)
- **outval**, which are the output values of the note at the current time step
- **instr**, a pointer back to the handle for this note (Section 5.30.14)

- **cop_storage**, the states of all the core opcodes under the control of this note (Section 5.30.18)
- **asample_ptr**, the number of the a-cycle within the current k-cycle that is being worked on
- **globalout**, a flag indicating whether the output of this note is the global orchestra output.

5.30.7. event (saol_sched.h)

An **event** holds one event from a score or a MIDI file that has not yet been dispatched (executed by the orchestra). It contains the name of the instrument, controller, or wavetable, the score label of the of event, the name of a sample referenced by a wavetable score line, the time at which the event will be dispatched, the duration of the event for SASL instrument events, the parameters of the event, the handle of an executing note (Section 5.30.14) for a SASL note-off event (that is, a note-off event created from a SASL instrument event with a duration), and the type of the event.

5.30.8. expr (saol.h)

The **expr** structure holds an expression of the SAOL. code. It contains the type of the expression (a terminal, an arithmetic expression, an opcode call, and so forth), the subexpressions of the expression (thus making it one node of a tree), the rate of the expression, the actual parameters (a list of expressions) for anopcode call, a terminal structure (Section 5.30.28), a pointer to the core opcode called by the expression, a pointer, the width of the epxression, and a number of extra fields used for keeping track of the macro expansion process (Section 5.15).

Not all of the fields of this structure are used for every type of expression. For example, for an arithmetic expression, the list of actual parameters is not used and is left as NULL.

5.30.9. formalparam (saol.h)

A **formalparam** is one formal parameter to a user-defined opcode. It contains the **name** (Section 5.30.16) of the formal parameter and the rate of the variable associated with it.

5.30.10. frameval (saol_interp.h)

A **frameval** is the current value of one variable in a note that's running. It is either a floating-point value or a pointer to a wavetable (Section 5.30.26).

5.30.11. global_block and global_decl (saol.h)

A **global_decl** corresponds to one line of global declaration in the orchestra. It is a parsing data structure, which means that it is directly filled up by the **yacc**-generated parser code (Section 5.10). After the orchestra is parsed, we figure out what the data in each **global_decl** means and move it into the overall **orc_global** structure (Section 5.30.20).

A **global_block** is a linked list of **global_decl** structures. It holds the whole global block as it is parsed.

5.30.12. hostvar_list (saol_sched.h)

A **hostvar_list** contains all of the information about the controllers and global variables attached to one instrument handle (Section 5.30.14). It is a linked list type with the data directly encapsulated in the linking structure. Each node of the list contains the data for one controller or global variable (the name "host variable" in **saolc** is due to historical reasons), and associates together the name of the controller, the symbol in the instrument symbol table for the controller (Section 5.30.24), the current value(s) of the controller, the width of the controller, and a flag indicating whether the controller value has changed since the last k-cycle of the instrument.

5.30.13. instr_decl (saol.h)

The **instr_decl** structure holds everything associated with the code of one instrument. This includes the name of the instrument, the list of instrument p-fields, the list of local variable declarations (Section 5.30.29), the code itself (Section 5.30.3), the local symbol table (Section 5.30.24), the list of busses the instrument routes to (Section 5.30.15), the list of preset numbers for the instrument, and the number of input and output channels of the instrument.

5.30.14. instr_handle (saol_sched.h)

The **instr_handle** structure contains all of the information describing one note in the pool of active notes. It associates together the following fields:

- **label**, which contains the score-line label of the note, if any
- **cx,** which contains the context of the note (Section 5.30.6)
- **id**, a link back to the instrument declaration and code that corresponds to the note (Section 5.30.13)
- **input** and **output**, the input and output sounds of the note
- **itime**, the amount of time the note has been executing
- **hvl**, the list of all the controllers used by the instrument (Section 5.30.12)
- flags indicating whether the note is being turned off, whether it is waiting for the sustain pedal to be released, and whether it is making sound
- parameters specifying the number of input channels, why the note was created (in response to to a SASL event, MIDI event, **send** statement, **instr** statement, or because it is the **startup** instrument), and the MIDI channel and MIDI pitch of the note

5.30.15. instr_route_list (saol.h)

The **instr_route_list** structure keps track of all the places the output of one instrument is routed. It is a linked list type, with the data directly incorporated into the linking structure. For each instrument, a separate list is maintained. Each node of the list corresponds to one **route** statement that that instrument is involved in. Each node associates together the list of all the instruments in that **route** statement, the bus that that **route** statement points to, and the starting channel of this instrument within the **route** (that is, the first channel of the bus that receives data from this instrument). A flag called **allchan** is set iff the instrument output goes onto all of the channels of the bus.

5.30.16. name (saol.h)

A **name** associates the name of a variable (as a character string) with an array width. It is used in places where a name is declared, such as a variable declaration (Section 5.30.29) or formal parameter list (Section 5.30.9).

5.30.17. opcode_decl (saol.h)

The **opcode_decl** structure holds everything associated with the code of one user-defined opcode. This includes the name of the instrument, the list of formal parameters (Section 5.30.9), the list of local variables (5.30.29), the code (Section 5.30.3), the local symbol table (Section 5.30.24) , and the rate and width (number of return channels) of the opcode.

User-defined opcodes only exist as separate structures until macro pre-processing of the orchestra (see Sections 3.2.1 and 5.15). After that, they are fully embedded in the instrument code. This data structure is not used during run-time decoding.

5.30.18. opval (saol_interp.h)

The **opval** structure contains the current state of one core opcode. This includes any local storage allocated by the opcode, and any dynamic storage allocated by the opcode (see Section 5.12).

5.30.19. orc (saol.h)

The **orc** structure associates the three main data structures describing the orchestra source code: **g**, the "global orchestra" structure (Section 5.30.20), **il**, the list of all the instruments (Section 5.30.13), and **op**, the list of all the opcodes (Section 5.30.17). This is the top of the SAOL parse tree.

5.30.20. orc_global (saol.h)

The **orc_global** structure keeps track of everything that pertains to the global block in the code. This includes the sample and control rates, the list of global variable declarations (Section 5.30.29), the list of route statements, the list of send statements, the symbol table of global variables (Section 5.30.24), and the list of all the busses in the orchestra (Section 5.30.4).

5.30.21. sa_decoder (saol.h)

The **sa_decoder** structure is the main structure containing the current state of the decoder. Everything that is needed to run the decoder is included or linked into this structure. Since it encapsulates all decoder state, multiple SA decoders may be run in the same address space without multi-threading; this feature is used in the Audio/Systems software (Section 6).

Unfortunately, **sa_decoder** is a "flatter" structure than it really ought to be—too many fields are thrown into it at the top level, which makes it very difficult to keep track of everything it stores. Some of the important fields of **sa_decoder** are:

- **global_cx**, which holds the current global context (see Section 5.30.6)
- **all**, which holds the parse tree and auxiliary data pertaining to the orchestra code (see Section 5.30.19).
- **sched**, which holds the current state of the scheduler (see Section 5.30.22)
- **outbuf** and **aifout**, which are used to buffer the output and keep track of the writing-to-the-output-file process
- **ended**, which is 0 as long as the decoder is still running
- **midicc**, which contains the values of all the MIDI controllers on all of the channels.
- **channels**, which contains the instrument currently active on each MIDI channel
- **seq**, which contains the sequence in which instruments are executed
- **cmd**, which contains the command-line controls specified for **saolc** (see Sections 2.2 and 5.30.5)
- **cached_au**, which contains the next AU from the bitstream to be decoded
- **in_bitstream**, which contains the status of the bitstream reading (see Section 5.9)

5.30.22. scheduler (saol_sched.h)

The **scheduler** structure associates all of the run-time data together. This includes the current orchestra time, the list of all the active note event handles (Section 5.30.14), the list of all the pending events (Section 5.30.7) that is kept sorted in order of dispatch time, a tail pointer to the list of pending events, the current tempo of the synthesizer, and a flag indicating whether the orchestra is running or not.

5.30.23. statement (saol.h)

A **statement** is one statement of SAOL source code. It associates together all of the possible parts of a line of code. This includes the kind of statement (assignment, **while**, and so forth), the rate type of the statement, subblocks for **if**, **while**, and **else** clauses, the lvalue expression for assignments, a main expression (Section 5.30.8), the name of an instrument called by the **instr** statement, and the name of a bus referenced by the **outbus** statement. There is a special field called **jump** that is used to indicate the next statement to execute in cases where this is not the next statement in the linked list. This field is only set by the macro-expansion code, where it is used to indicate the next statement to execute after a **return** statement.

Not all of the fields in this structure are filled for each kind of statement. For example, for a **while** statement, the **lvalue**, else-block, instrument-name, and bus-name fields are left empty. Some of the fields have slightly different meanings depending on the kind of statement. For example, for a **while** statement, the **expr** field holds the guard expression, while for an assignment statement, the **expr** field holds the right-hand-side expression.

5.30.24. symbol and symtable (saol.h)

The **symbol** structure contains one entry in a symbol table. A symbol table holds the list of all the variables in a context (5.30.6); it is just a list of symbols. Each symbol contains the name, rate type, and width of the variable; a link to the table definition (Section 5.30.25) if the variable is a wavetable; flags indicating if the variable is imported, exported, or both; a link to the global variable if the variable is imported or exported and there is a global variable of the same name; a pointer back to the line of code that declared the variable (Section 5.30.29); a list of names if the variable is a tablemap; and the offset of the variable within the frame. This last value gives the offset of the particular **framevalue** for this symbol within the **framevals** array of a **context** using this symbol table (Sections 5.30.10 and 5.30.6). There is also a special field **mark** that is used to mark variables that are actually used in the code; if a variable is not used, it is removed before run-time for efficiency.

5.30.25. tabledef (saol.h)

The **tabledef** structure contains the data from one table definition. This includes the name of the table, the table generator to be used, and a list of expressions (Section 5.30.8) that serve as the parameters to the table.

5.30.26. table_storage (saol_interp.h)

The **table_storage** structure contains a wavetable. This includes the name, size, sampling rate, looping points, and base frequency of the wavetable, and all of the current values of the wavetable data.

5.30.27. template_decl (saol.h)

The **template_decl** structure is used only during orchestra parsing, to hold all of the information associated with a template in the SAOL orchestra code. As soon as it is parsed, it is converted (see Section 5.27) into a list of instruments stored as **instr_decl** structures (Section 5.30.13).

5.30.28. terminal (saol.h)

The **terminal** structure holds a terminal in the code. A terminal is a code element with no further structure. This includes a constant value, a string constant, or a variable name. In the case of a variable name, the **iname** field holds the character-string name, and the **sym** field is used to point to the entry in the appropriate symbol table for the variable (Section 5.30.24).

5.30.29. vardecl (saol.h)

The **vardecl** structure contains all of the data that might be associated with a declaration of one local or global variable. This includes the rate type of the variable, the list of **name**s (Section 5.30.16) in the declaration, a pointer to a table declaration (Section 5.30.25), the name of a tablemap, and flags indicating whether or not the variable is **import**ed or **export**ed or both.

Not all of the fields of this structure are filled for every variable declaration – for example, for a regular **ksig** declaration, the table pointer is left **NULL**.

6. saolc and the rest of ISO 14496-5

This section will discuss the relation with the AudioBIFS reference software.

7. Bugs and other known issues

7.1. Introduction

This section describes known bugs and other issues in **saolc.** The issues fall into five main categories: things that are supposed to work, but don't ("bugs"); things that are required to be in a compliant implementation, but aren't implemented yet; places where **saolc** fails to conform to an instruction in the specification ("violations"); things that **saolc** does that it doesn't need to ("extensions"); and bugs in the specification that are fixed in **saolc** and will be fixed with corrigenda to the standard at a future time.

A short section on portability is also included.

To report additional bugs, or to provide patches that fix bugs or implement non-implemented features, please send e-mail to **saol-bugs@media.mit.edu.** The Media Laboratory is no longer actively developing this codebase; therefore, we can't necessarily provide bugfixes immediately on request. However, we are happy to act as a clearinghouse for bugfixes contributed by the community.

7.2. Bugs

There are a few known bugs in **saolc** v. 1.0. This list will likely grow as the software is exercised more completely.

1. The template parser does not handle expressions in the mapblock properly – this is also an error in Annex C.3 of the specification. A template like
   template <i1, i2> (p1, p2) map { a, b } with { < i + 4, p1>, <i + 2, p2> } { ... }
   can't be parsed—the expressions will crash. (It was an unfortunate choice to use the angle bracket as the delimeter for the mapblock because it means the grammar is not LR(1)).
2. The PICOLA speed-change tools are broken.

7.3. Features not yet implemented

Some features of the MPEG-4 standard have not been implemented in **saolc** v. 1.0. The following aspects of the standard have not yet been implemented:

- The **spatialize** statement
- The **reverb, chorus,** and **speed_t** core opcodes
- The MP4 file format (see Section 8).

7.4. Features implemented in a non-compliant manner

Features implemented in a non-compliant manner are indicated in comments in the code containing the word VIOLATION. The following violations are currently known:

- Global wavetables are linked into instrument instances through pointer copies. This means that changes to **imported** tables are globally seen even if the table is not **exported**. For proper operation, a deep copy of the table should be made at each k-rate instance, or else some list of modifications should be maintained and rolled into the table for **exported** tables (`saol_interp.c`).
- There is simple packaging for Access Units that contains their Decoding Time Stamps. The full MPEG-4 File Format (MP4) is not yet supported (`sa_bitstream.h`).
- There are fixed maximum lengths for numbers of pfields to an instrument and number of parameters to a core opcode or core table generator. There should be no such limits. (`sa_interp.c`).
- Because of the way macro-expansion is implemented, when a global variable is imported by both an instrument and a user-defined opcode called by that instrument, the opcode and

instrument share a single instance of the variable. This violates the NOTE in Subclause 5.8.6.7.6; in particular, changes made in the caller are seen in the callee immediately, rather than taking a k-cycle to propagate, and changes made in the callee will show up in the caller even if the callee does not export the variable.

- When two events have different time stamps, but are dispatched in the same k-cycle due to control-rate quantization of event dispatch, they should be ordered as though they occurred at the same time. That is, instrument events should precede control events, and so forth per Subclause 5.7.3.3.6. In the current **saolc**, though, the event with the earlier timestamp will be dispatched first. This only makes a difference in rare cases, which can be worked around by making small (< 1 k-period) adjustments to the event times (`saol_sched.c`).

7.5. Extensions

Some features of **saolc** provide functions that are not required of normative operations. Content authors using **saolc** should not depend upon these functions if they wish their bitstreams to be properly decoded by a normative decoder.

- Command-line processing. There are many options that can be used on the **saolc** command line to use **saolc** in a variety of standalone ways as well as to do bitstream decoding.
- Preprocessing. The C preprocessor can be used to expand macros and #defines in SAOL code before it is parsed. Content authors using **saolc** should be wary of depending too heavily on this feature, as it is not likely to be widely supported in other implementations.
- **idump()**, **kdump()**, and **adump()** debugging core opcodes. These opcodes each dump one or more arguments to **stdout** and can be used to debug SAOL code. **kdump()** and **adump()** may be transported in the MP4 bitstream created with **saencode—**two of the "free" tokens are used for this purpose.
- Sound-file input for the **sample** core wavetable generator. When SAOL or SASL code in the textual format uses the **sample** generator, the first argument is a string constant, giving the name of an AIFF sound file containing a sample. There is no normative requirement for a Structured Audio decoder to handle AIFF files or samples on disk..
- Sound-file input to **input_bus**. Normally, sound on the global **input_bus** is only provided when the SAOL process is running as an **AudioFX** node in AudioBIFS. Using the `-input` command-line switch allows the decoder to run in standalone mode as an effects-processor.

7.6. Bugs in the standard that are fixed in saolc

There are a few bugs left in the text of the standard that will be submitted as official corrigenda. When possible, **saolc** conforms to the correct version, not the broken version. Here is the list of such items:

- Subclause 5.10.5 (**buzz** core table generator): should include the statement "If **nharm** < 1, then **nharm** = **size** / 2 – **lowharm**."
- Subclauses 5.9.3.6 (buzz core opcode) and 5.10.5 (**buzz** core table generator): Should include the statement "If **rolloff** is equal to 1, then **scale** is 1."
- Annex A (token table). There is no token for **startup**. This is needed for orchestras that convey a **startup** instrument with no token table. The token **0x4B** is used for the lexeme **startup**.
- Annex C.3. The **yacc** code for the template expression is wrong (see also Section 7.2 #1).

7.7. saolc portability

saolc is widely portable. All of the core bitstream-processing and algorithmic synthesis (Object 3) code works without difficulty on a variety of platforms. This section contains a short list of the known obstacles to porting **saolc** to new platforms.

- The ANSI C++ standard template library is required. The bitstream-parsing code in `sa_bitstream.h` uses the **<vector>** template. The simple encoder **saenc** (see Section 8) additionally requires the <**priority queue**> template. Most modern C++ compilers provide support for the standard template library.
- The **sasbf** implementation is not 64-bit clean, as the file parser for DLS-2 files makes assumptions about the relationship between pointer size and **float** size. Thus, the **sasbf()** expression in **saolc** cannot be used on architectures with 64-bit pointers such as the DEC Alpha.

7.8. Possible optimizations

saolc runs very slowly right now compared to well-tuned software synthesizers such as Csound or Supercollider. This is mostly because, as a reference implementation, it is important to keep the functioning as clear as possible, even at the expense of efficiency. Only a few of the aspects of the current codebased originated with an intent to increase speed. Primary among these is the use of macro expansion to implement user-defined opcodes; this improved the speed by nearly 40% over a previous version that used dynamic stack frames. (The macro expansion is hard to understand, but so were the stack frames!)

The removal of unused variables was originally intended to improve speed—it did in an earlier version of the code—but since identifiers are now linked directly to the symbol table and thereby to the offset within the **framevals** array, there's no real improvement anymore.

Optimizations that might give good speedups include:

1. Converting the parse tree into a p-code structure, for example, a 3-argument machine with much simpler opcodes. This would eliminate most of the recursion overhead due to traversing the parse tree (currently 30% of the cycles in **saolc**), as well as simplifying variable-lookup, as the local variable addresses would be implemented in terms of a stack pointer.
2. Detection of algorithms which can be block-processed (see the discussion of block processing in Scheirer & Vercoe, 1999). Not every algorithm can be legally block-processed in SAOL, but the ones that can, can be speeded up greatly. This would involve writing a static analyzer to determine when it is not possible for an instrument to be using single-sample feedback (which is the case that prohibits block-processing), and then updating all of the interpreter clauses and core opcode implementation functions to be responsive to that information.
3. A stronger focus on static optimization using standard techniques from the compiler literature—for example, common subexpression elimination and removal of dead code. This would likely be most effective in conjunction with (1).
4. On some architectures, integer arithmetic runs faster than floating-point arithmetic. It is often possible to statically detect when a variable is an integer, and when it is a float, even though there are no explicit integer types in SAOL. (For example, if all you ever do with **i** is say "**i = 0"** and "**i = i + 1**" you can tell statically that **i** takes on only integral values). Integer detection and corresponding manipulations of loop structures and variable types might give improved performance.
5. Most of the core opcodes could be improved a bit simply by tuning loops, using different data structures, etc. This would give good results for instruments making heavy use of those opcodes.

It is to be emphasized that most of the time in **saolc** is currently overhead—only about 15% of the processing cycles are actually doing synthesis! Thus, tuning the synthesis code for speed will have little effect unless the overall framework is improved first. A system like Csound spends about 85% of its cycles doing synthesis, just the inverse of **saolc**. (But Csound has the advantage of block-based semantics and a much simpler expression grammar).

The Media Lab is happy to host any patches or new versions of **saolc** that run faster on the main SA web site. Some other groups working on SAOL implementations are more focused on efficiency—see the pointers on the SA web site.

8. A simple Structured Audio encoder: saenc

8.1. Introduction

The **saolc** package includes a very simple Structured Audio encoder program, called **saenc**. **saenc** takes a number of component files stored on disk—SAOL programs, SASL scores, sample data, and MIDI files—and allows them to be flexibly assembled into the official bitstream format. It then wraps the bitstream format with simple framing information and calls the result an MP4 file.

The MP4 file produced by this tool is not stored in the official MP4 file format, which had not been completed as of this document. In the long run, these two tools may be easily converged. The MP4 file produced by **saenc** may be viewed as a "raw stream data" file—it contains the decoder configuration information, and a sequence of SA Access Units, but without the complex framing and timestamping available within the final MP4 format.

saolc can read and parse raw stream MP4 files produced by **saenc,** and decode the resulting instructions into sound. In the long run, it will also read and parse the official MP4 file format, but this is not supported yet (see section 7.3).

This section describes the operation and functioning of **saenc**. Subclause 5.5.2, which describes the bitstream format, is a useful reference.

8.2. Using saenc

The **saenc** command line is very simple:

```
saenc filename
```

`filename` is typically given with the `.mp4` extension, eg `saenc test.mp4`.

During execution, **saenc** leads the user through a series of command-line prompts that enable different **chunks** of data to be added to the encoded file. First, chunks are added to the decoder configuration header. Then, when all the chunks desired are included in the configuration header, chunks are added to the streaming data. When all the data desired are included in the streaming data segment, the file is written to disk.

8.3. Adding chunks to the decoder configuration header

During the decoder-configuration-header part of the encoding process, the following prompt is given:

```
Chunk types: 1=SAOL 2=SASL 3=sample 4=MIDI 5=SASBF (0=done)
Type, name:
```

The user enters a value indicating the type of the chunk, and the filename that contains the data. For example:

```
Type, name: 1 piano.saol
```

For type 1 (SASL) chunks, **saenc** reads in the requested file, tokenizes the code, and adds the corresponding SAOL chunk to the header.

For type 2 (SASL) chunks, an additional prompt is presented:

```
start end shift:
```

This prompt indicates that **saenc** allows the score to be segmented and time-shifted as it is added to the header. Only score events between the given **start** and **end** times are incorporated, and each score event is time-shifted by **shift** beats. For example

```
start end shift: 2.5 0 -2
```

indicates that only score events more than 2.5 beats into the score file should be used (**end**=0 indicates that all events to the end-of-file should be used), and each event should be shifted 2 beats earlier in time. The given SASL score is segmented, shifted, tokenized, and added to the header.

For type 3 (sample) chunks, an AIFF file is read. If the file is stereo, the user is prompted to select a channel to include (since only mono samples are in the **sample** block of the bitstream[1]). If 0 is entered, all channels of the sound file are averaged.

For type 4 (MIDI) files, a Standard MIDIFile is read. No parsing of the MIDIFile occurs during encoding; the entire file is placed in the header regardless of its contents.

For type 5 (SASBF) files, a SASBF (DLS-2) file is read. No parsing of the DLS-2 file occurs during encoding; the entire file is placed in the header regardless of its contents.

When **0** is entered, the header is complete. **saenc** then prompts for the inclusion of the *symbol table*. This table gives a list of all the user-defined symbols in the orchestra. When SAOL code is tokenized, each operator, keyword, core opcode, and so on is turned into a 8-bit value. An escape value (0xF0) is used to indicate *symbols*, which are the different variable names, user-defined opcode names, and so forth in the instrument. Each symbol is conveyed with a 16-bit value; each time the same symbol appears, the same value is conveyed. The bitstream symbol table gives the value-name associations for the orchestra, which allows the tokenized orchestra to be converted back into readable textual SAOL.

The symbol table is not necessary; it is possible to decode the orchestra correctly simply by giving the first symbol the name `_sym_0`, the second `_sym_1`, and so forth. To do this makes the orchestra less human-readable, but syntactically identical. Using the symbol table allows human-readable text to be recovered at the expense of some added bits in the compressed file (the exact number of added bits depends on the number of symbols in the orchestra).

8.4. Adding chunks to the streaming bitstream data

After the header is complete, the streaming part of the bitstream may be formed. This is accomplished by giving several files containing time-stamped events to the encoder; the encoder sorts them together, and produces time-stamped Access Units according the proper bitstream format. In a streaming encoder, each Access Unit would be sent off over the network; in **saenc** they are all just written into the MP4 file directly after the decoder configuration header.

The prompt for streaming data looks like this:

```
Chunk types: 2=SASL 3=sample 4=MIDI 6=data blocks (0=done)
Type, name:
```

Options 2 and 4 are very similar to their counterparts in the decoder configuration header. They both allow the events to be segmented and time-shifted as in the score-processing for the header. For option 4, the Standard MIDIFile is parsed and the delta-times are interpreted, in order to give each event an actual timestamp. These timestamps are used to time-stamp the access units.

Option 3 is not done yet as of this writing.

Option 6 allows "streaming wavetable" data to be included in the bitstream. A set of datafiles should be prepared, with filenames sharing a common stem, like `my_data0.dat`, `my_data1.dat`, `my_data2.dat`.... Each datafile should contain frames of signed 16-bit big-endian integers. Each frame begins with a 16-bit length tag and then consists of that many 16-bit values. All of these files are read in order to make one long sequence of frames:

```
my_data0.dat
3 -45 6 140
5 200 -12348 72 0 -1
...

my_data1.dat
2 -1 1
12 0 0 0 0 0 45 0 0 0 0 -85 5
...
```

---

[1] This is because SAOL wavetables are mono-only, and thus there is no way to store a stereo sample in a wavetable. A better encoder could turn a stereo input file into multiple samples, with names related in some useful way.

This datablock sequence consists of four frames, with three values in the first frame, five in the second, two in the third, and 12 in the fourth.

The filename given for option 6 should not be the whole filename, but only the stem ("my_data" for this example). **saenc** requests an offset, which is the time at which to send the first frame of data, and the blockrate, which is the number of blocks to send per second.

Each frame of data is packed into a pair of bitstream elements: a **table sample** score line in SASL and a **sample** bitstream data chunk. These elements are associated together so that when the Access Unit is parsed, the data in the frame is poured into the wavetable. The name of the wavetable is the same as the name of the datafile stem ("my_data"). In a SAOL instrument that imports the wavetable with this name (that is, with the declaration `imports table my_data`), each time one of the data frames occurs in the bitstream, the values in the wavetable are silently replaced with new values.

This function allows the construction of analysis-synthesis tools and natural audio coders in the SA bitstream. See the papers by Scheirer & Kim and by Wright & Scheirer, referenced in Section 9.2, for concrete examples.

9. References and Credits

9.1. Introduction

This section provides references to the technical literature, an annotated bibliography for further reading, and credits to all the people who worked on **saolc**.

9.2. Technical references

This is a list of papers in the technical literature as of June 1999 that deal directly or indirectly with the MPEG-4 Structured Audio standard. Copies of many of these can be found at the SA home page.

1. Casey, M. A. & Smaragdis, P. J. (1996). Netsound: Real-time audio from semantic descriptions. In *Proceedings of the 1996 Int. Computer Music Conf.* (pp. 143). Hong Kong: International Computer Music Association.
2. Casey, M. A. (1998) *Auditory Group Theory with Applications to Statistical Basis Methods for Structured Audio*. Unpublished Ph.D. dissertation, MIT Media Laboratory, Cambridge MA.
3. Koenen, R. (1999). MPEG-4: Multimedia for our time. *IEEE Spectrum* **36**(2), 26-33.
4. Scheirer, E. D. (1998). The MPEG-4 Structured Audio standard. In *Proceedings of the 1998 IEEE Int. Conf. Acoust. Speech Sig. Proc.* (pp. 3801-3804). Seattle.
5. Scheirer, E. D. (1998). The MPEG-4 Structured Audio Orchestra Language. In *Proceedings of the 1998 Int. Computer Music Conf.* (pp. 432-438). Ann Arbor, MI.
6. Scheirer, E. D. (1999). Structured audio and effects processing in the MPEG-4 multimedia standard. *Multimedia Systems* **7**(1), 11-22.
7. Scheirer, E. D. & Kim, Y. E. (1999). Generalized audio coding with MPEG-4 Structured Audio. In *Proceedings of the Audio Eng. Soc. 17th International Convention on High-Quality Audio Coding.* Florence, IT.
8. Scheirer, E. D., Lee, Y. & Yang, J.-W. (in press, a). Synthetic audio and SNHC audio in MPEG-4. In A. Puri & T. Chen (eds.), *Advances in Multimedia: Systems, Standards, and Networks.* New York: Marcel Dekker.
9. Scheirer, E. D., Lee, Y. & Yang, J.-W. (in press, b). Synthetic audio and SNHC audio in MPEG-4. *Image Communications.*
10. Scheirer, E. D. & Ray, L. (1998). Algorithmic and wavetable synthesis in the MPEG-4 multimedia standard. Presented at the 105th Convention of the Audio Eng. Soc. (AES reprint #4811). San Francisco.
11. Scheirer, E. D., Väänänen, R. & Huopaniemi, J. (1998). AudioBIFS: The MPEG-4 standard for effects processing. In *Proceedings of the 1998 Digital Audio Effects Workshop (DAFX-98)* (pp. 159-167). Barcelona: Audiovisual Institute, Pompeu Fabra University.
12. Scheirer, E. D. & Vercoe, B. L. (1999). SAOL: The MPEG-4 Structured Audio Orchestra Language. *Computer Music Journal* **23**(2), 31-51.
13. Smith, J. O (1991). Viewpoints on the history of digital synthesis. In *Proceedings of the 1991 Int. Computer Music Conf.* (pp. 1-10). Montreal.
14. Vercoe, B. L., Gardner, W. G. & Scheirer, E. D. (1998). Structured audio: The creation, transmission, and rendering of parametric sound representations. *Proceedings of the IEEE* **85**(5), 922-940.
15. Wessel, D. (1991). Let's develop a common language for synth programming. *Electronic Musician* magazine, pp. 114 (August issue).
16. Wright, M. & Scheirer, E. D. (1999). Cross-coding SDIF into MPEG-4 Structured Audio. *In Proceedings of the 1999 Int. Computer Music Conf.* Beijing.

9.3. Bibliography

This is a list of books and other references that provide background reading regarding the concepts in the implementation of **saolc**.

1. Aho, A. V., Sethi, R. & Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley.

This is a classic text on compiler-design. Particularly for readers unfamiliar with parsing, syntax-checking, and interpreting computer code, this is valuable reading for anyone who wants to understand **saolc** and other implementations of SA tools.

2. Mathews, M. V. (1961). An acoustic compiler for music and psychological stimuli. *Bell Systems Technical Journal* **40**, 677-694.
3. Mathews, M. V. (1963). The digital computer as a musical instrument. *Science* **142**, 553-557.
4. Mathews, M. V. (1969). *The technology of computer music*. Cambridge, MA: MIT Press.

Max Mathews was the inventor of the basic concept underlying the SAOL synthesizer – that of the “unit generator” or “Music-N” computer language. A Music-N language uses a textual language to patch together unit generators, oscillators, and wavetables to describe a desired synthesis algorithm. These three references were among the earliest in the computer-music field and set off a chain of implementation and refinement that ultimately led to the development of the Structured Audio standard.

5. MIDI Manufacturers Association. 1996. *The Complete MIDI 1.0 Detailed Specification*. Protocol specification, Los Angeles, MIDI Manufacturers Association.

This is the official specification of the MIDI protocol. It makes interesting reading, both as a historical document and to help understand the MIDI capabilities of the Structured Audio tools.

6. Roads, C. (1996). *The Computer Music Tutorial*. Cambridge, MA: MIT Press.

This is an encyclopedic presentation of nearly all topics in the computer-music field as of its writing. Readers who want to learn more about the functioning and history of the core unit generators or the kinds of synthesis algorithms that might be implemented in Structured Audio should start here and then proceed through its voluminous bibliography.

9.4. Credits

This is a list of people who have contributed code to **saolc**. If you have contributed code or bugfixes and you’re not listed here, it’s due to an oversight. Please send email to `eds@media.mit.edu` and it will be corrected in the next release. All responsibility for bugs in **saolc** belongs to Eric Scheirer as the primary integrator.

Name	**Organization**	**Contributions**
Steven Curtin	Lucent Technology	Assorted bugfixes
Luke Dahl	E-Mu/Creative	Interface between main SAOL synth and DLS-2 synth.
Alexandros Eleftheriadis Yihan Fang	Columbia Univ.	Bitstream-parsing code: `bitstream.cpp, flerror.cpp, bitstream.h`, and associated header files. Also the Flavor bitstream-parser generator, which was used to create the basis (with later modifications) of `sa_bitstream.cpp`
Daniel P. W. Ellis	MIT Media Lab & ICSI Berkeley	Code borrowed from Csound, for reading/writing audio files and performing FFTs: `aifif.c`, `byteswap.c`, `fft.c`, and associated header files
Todor Fay	Microsoft	DLS parser: `loadicol.`cpp, `dls_wrapper.`cpp,

		`instr.cpp`, `clist.cpp`, and associated header files.
William Gardner	MIT Media Lab	`IEEE80.c`, which is needed by `aifif.c`, originally written for Csound.
John Lazzaro	Berkeley CS Dept	Assorted bugfixes
Eric D. Scheirer	MIT Media Lab	Main codebase, code integration, and everything else not listed here.
David Sparks	E-Mu/Creative	DLS-2 software synth: `sf_*.{c,h}`
Naoya Tanaka	Matsushita	PICOLA speed-change tool and connection to decoder: `fx_picola.{c,h}` and `co_fx_speedc()` in `saol_co_imp.c`
Giorgio Zoia	EPFL Switzerland	tempo-change code in `saol_sched.c`, assorted bugfixes