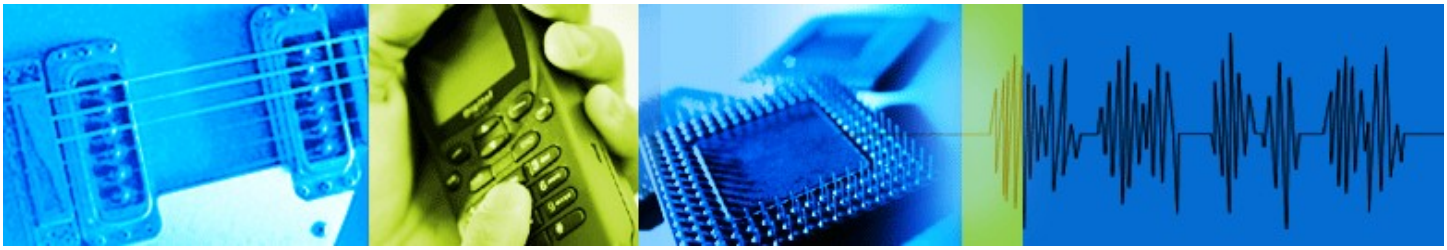
SONIC NETWORK

EMBEDDED AUDIO SYNTHESIS



Sonic Network, Inc.
Embedded Audio Synthesis (EAS)

EAS API Reference

(Version 3.6)

Sonic Network, Inc.
561 Windsor Street
Suite A402
Somerville, MA 02143
USA

Table of Contents

1 Introduction

1.1 Abstract

This document outlines the implementation of the Sonic Network Inc Embedded Audio Synthesizer Library. This document provides a high level overview of the EAS synthesizer interface, and does not discuss the underlying EAS synthesizer. The Sonic EAS Library is implemented using a combination of fixed point C and assembly code for a variety of processors.

1.2 Intended Audience

This document is intended for the engineer integrating the Embedded Audio Synthesizer into the target system.

1.3 Abbreviations

EAS	Embedded Audio Synthesis
Synth	Synthesizer
HW	Host Wrapper
CM	Configuration Module

1.4 Revision History

Rev	Date	Author	Comments
1.00	Feb 4, 2003	jt	Initial Draft
1.01	Sep 3, 2004	jt	Revised for new public interface
1.02	Sep 22, 2004	jt	Added new functions to public interface, SynthStop, SynthSetStopTime, SynthPause, and SynthResume
2.00	Feb 7, 2005	jt	Revised for new (Gen3.2) public interface
2.01	May 24, 2005	dls	Added new API calls
2.02	Jun 16, 2005	dls	Documented streaming MIDI interface, minor updates
2.03	Jul 9, 2005	dls	Documented EAS_SetParameter, EAS_GetParameter, SRS WOW XT interface
2.04	Jul 16, 2005	dls	Documented additional ringtone parsers
2.05	Jul 16, 2005	jah	Added Mobile DLS/XMF parser define
2.06	Jul 27, 2005	dls	Documented metadata retrieval functions and wave file parser
3.00	Feb 9, 2006	dls	Updates for Version 3.4
4.00	May 18, 2006	dls	Updates for Version 3.5
4.01	Jun 2, 2006	dls	Updated metadata functions and minor cleanup
4.02	Jul 21, 2006	dls	Added EAS_GetFileType function
5.00	Mar 8, 2007	dls	Update for V3.6
5.01	Jul 2, 2007	dls	Added EAS_SetPlayMode and EAS_SetHeaderSearchFlag

1.5 References and related documents

1	Complete MIDI 1.0 Detailed Specification – MIDI Manufacturers Association
2	Scalable Polyphony MIDI Specification & Profiles – MIDI Manufacturers Association
3	WOW® XT For ARM Usage – SRS Labs, Inc.

2 EAS Library

2.1 Overview

The EAS Library is compiled to have the attributes listed in Table 1. Most of these attributes are fixed at compile time and cannot be changed after the library has been generated. However, some of these features can be set at run time, such as polyphony.

Table 1 - EAS Library Attributes

Attribute	Value
Max Polyphony (number of voices)	1-256 voices
Sample Rate	8000, 16000, 22050, 24000, 32000, 44100 Hz
Wavetable	GM Set: 128 Melodic Instruments, 47 Rhythmic (Drum) instruments 8-bit or 16-bit samples
File Formats	MIDI Type 0, SP-MIDI (based on Type 0), MIDI Type 1
Optional File Parsers	SMAF, iMelody, OTA, RTTTL/RTX, WAVE, CMX, MFi
Optional Audio Decoders	SMAF, IMA ADPCM
Optional EAS Modules	Enhancer, Booster, Reverb, Chorus, EQ, Mixer
Audio Outputs	Mono or Stereo

2.2

Table 2 shows a list of all the files and their purpose.

Table 2- List of Files

Filename	Purpose
eas.lib	EAS synthesizer module library, compiled for the particular processor.
eas_main.c	A sample client application. This file contains main and should be replaced by the target system's application framework.
eas_host.c, .h	Host Wrapper (HW) interface for fileio, memory allocation, etc.
eas_config.c, .h	Configuration Module (CM)
eas_wave.c, .h	Wave file output used for sample application
eas_report.c, .h	Debug message reporting
eas_hostmm.c	Memory mapped version of eas_host.c
eas_types.h	Public typedefs and defines
eas.h	Public interface

The EAS synthesizer module library has been designed for flexible integration with virtually any system. All memory is allocated using whatever method, dynamic or static, is best for your system. In addition, when memory is dynamically allocated and freed, the synth calls abstracted wrapper functions that allow you to have control over the details, or simply make use of the default implementation provided by us.

The sample client application eas_main.c has been designed specifically for evaluation / demonstration purposes. It does not require the client application to use any specific operating system or application framework, and instead, has been configured to use abstracted file input/output functions. These functions are wrappers that can be implemented with whatever calls your system requires. The sample code is dependent on some standard ANSI C library functions, which may required modification if your system does not support them.

All functions that might need to be modified are part of eas_host.c, eas_report.c and eas_wav.c. Other public functions provided by the library allow you to interface with the EAS synthesizer in simple yet powerful ways.

2.3 Upgrading From Previous Versions

In Version 3.5, we introduced several new features including support for multiple streams. This necessitated changes to a few API functions to facilitate the use of multiple streams. We also added stronger type checking to eliminate certain programming errors. New underlying data types have been introduced to distinguish between the primary EAS data handle, stream handles, and host wrapper handles.

While these changes will require minor modifications to existing host application code, the changes are relatively simple and should only require a few minutes to complete. See Appendix B for more information on updating your application if you are upgrading from V3.4 or earlier.

3 Theory of Operation

EAS incorporates a modular design approach to facilitate easy customization and configuration. While some features must be configured by Sonic Network when the library is built, others can be configured during the application build process. More detail on this process can be found in the section on the Configuration Module.

3.1 General API usage

Most EAS API calls require an EAS_DATA_HANDLE parameter and an EAS_HANDLE parameter. The EAS_DATA_HANDLE pointer is obtained by calling EAS_Init during initialization of the library and should be retained by the application until EAS_Shutdown is called. The EAS_HANDLE pointer is obtained when a new stream is opened via EAS_OpenFile or EAS_OpenMIDIStream and should be retained by the application until the EAS_CloseFile or EAS_CloseMIDIStream is called.

Most EAS API calls return an EAS_RESULT type. The return values are defined in the eas_types.h header file. When success, the return value will be EAS_SUCCESS. It is advisable to check return values on all EAS API calls.

3.2 Host Wrapper

All platform specific features have been isolated into a single module referred to as the “Host Wrapper” module – the name reflects the fact that all platform specific functionality has been wrapped by calls to this module. This makes it easy to adapt the library to work on various platforms, and indeed, the library has been ported to a variety of platforms and processors.

The audio file parsers use an I/O protocol similar to POSIX file I/O, a standard I/O interface for most C run-time libraries. The protocol is abstracted through the host wrapper functions making it easy to adapt to different operating environments. The sample wrapper module maps directly to POSIX calls – if your platform supports them, you can use the module without modification.

The file I/O wrappers can be modified to work with other storage paradigms, provided that they support some form of random access. For example, if your audio files are stored at predetermined locations in flash, you can pass a pointer to the data as your file locator, and the host wrapper can fetch the data directly. For devices with slow random access (e.g. serial flash), the host wrapper has a buffering system that can be scaled to the appropriate size to maintain performance.

3.3 Memory Model

EAS supports both dynamic and static memory models. The dynamic memory functions are modelled on the standard POSIX malloc/free memory functions and abstracted in the host wrapper module. It is usually straightforward to adapt these functions to work with a typical RTOS. The number of memory allocations has been minimized to reduce the likelihood of memory fragmentation, and de-allocations usually occur in the reverse order of allocations which also tends to minimize fragmentation.

The static memory model has some limitations. It does not support multiple streams and it does not support the XMF/DLS parser which requires allocated memory for the parser DLS instrument collection.

3.4 Streams

EAS now supports rendering multiple audio streams simultaneously (nominally up to four, but that number can be increased upon request), provided that you are using the dynamic memory model. This means that a ring-tone can play while a game is using EAS for sound effects, for example. If you plan to take advantage of multiple streams, you may need to think through your application strategy for using EAS.

With a single stream, the typical application of EAS is to open the audio file, begin calls to the EAS render function monitoring the state of the audio stream, close the file when playback completes, and then stop calling the render function.

With multiple streams, you may want to adopt the strategy of always calling the render function. The overhead of the call is very low if no streams are playing. Alternatively, you can monitor the state of the streams and stop the call to render if no streams are playing.

Each stream has individual controls for pause/resume, locate, volume, polyphony, and priority. These controls are in addition to the master volume and polyphony controls. You can pause or resume individual streams without affecting the output of other streams.

3.5 Polyphony

Polyphony is the number of simultaneous notes that can be played with the MIDI synthesizer. The maximum polyphony is determined when the EAS library is built and is nominally 64 voices, but can be changed upon request. Your build will include a readme file that shows the build configuration including the maximum polyphony.

Polyphony is normally the single largest factor in determining the processor load. Reducing the EAS processor load is usually as simple as calling the EAS_SetSynthPolyphony function to reduce the maximum number of voices in use. See the section on Performance Tuning for more information.

When a new stream is opened with the EAS_OpenFile function, the polyphony setting for the stream defaults to zero, which indicates that the stream is allowed to use as many voices as needed up to the maximum. If multiple streams are playing and collectively they require more voices than are available, they will compete for voices, possibly resulting in a jumbled sound. We suggest when multiple streams are active, that each stream should be allocated a portion of the total polyphony to prevent competition.

3.6 Priority

By default, all streams operate at the same priority, with the default priority being roughly in the middle. If you want to be sure that one stream receives the voice allocation it requires, you can increase the priority of that stream. For example, assume that you want to ensure that the ring-tone always takes precedence over anything else that might be taking place on the device. One method of accomplishing this is to set the priority of the ring-tone stream to 1 (the highest priority) and limiting the polyphony to half the available voices. This assures that the ring-tone will always play, while still allocating half the voice pool to other applications.

4 Public Interface

The following functions can be called by your client application and are provided by the library. Some of them must be called to use the synthesizer, others are optional to use as desired.

4.1 Main Library Functions

These functions include initialization, configuration, rendering, polyphony control, and shutdown functions.

4.1.1 EAS_Init

Call this to init the data for the EAS synthesizer. Pass in the address of an EAS_DATA_HANDLE which will be set to point to the data used by the synth. This data handle (hereafter referred to as pEASData) will be used by other public interface routines.

```
EAS_RESULT EAS_Init(EAS_DATA_HANDLE *ppEASData);
```

4.1.2 EAS_Render

This function performs the actual audio rendering via the synthesizer. The synth calls the appropriate file parser as needed. Call this repeatedly to render audio from the song file. Pass in pEASData obtained from EAS_Init, a pointer into the the host buffer at a particular offest, the value of pConfig->mixBufferSize, and the address of a counter which will return the actual number of output samples rendered.

```
EAS_RESULT EAS_Render(EAS_DATA_HANDLE pEASData, EAS_PCM *pOut, EAS_I32
   numRequested, EAS_I32 *pNumGenerated);
```

4.1.3 EAS_Shutdown

Shuts down the library. Deallocates any memory associated with the synthesizer (dynamic memory model only). Pass in pEASData the data handle that was obtained from EAS_Init.

```
EAS_RESULT EAS_Shutdown(EAS_DATA_HANDLE pEASData);
```

4.1.4 EAS_Config

Returns a pointer to a structure containing the configuration options in this library build. Below is the structure definition. A description of the returned data follows:

```
typedef struct
{
```

```
    EAS_U32 libVersion;
    EAS_BOOL checkedVersion;
    EAS_I32 maxVoices;
    EAS_I32 numChannels;
    EAS_I32 sampleRate;
    EAS_I32 mixBufferSize;
    EAS_U32 buildTimeStamp;
    EAS_CHAR *buildGUID;
} S_EAS_LIB_CONFIG;

const S_EAS_LIB_CONFIG *EAS_Config(void);
```

libVersion: A 32-bit library version number formatted as 4 octets. The function EASLibraryCheck in the sample source module eas_host.c demonstrates how to check that the header file eas.h matches the library binary. We suggest you include this in your debug or checked build to verify that the library and header file are in sync.

checkedVersion: A boolean flag indicating that the library is a checked build that includes additional debug information. The checked build code size is larger and performance is not as good as the production build due to the overhead of validating parameters.

maxVoices: Returns the maximum number of voices the synthesizer can support. This parameter is set at compile time. You can reduce the number of voices using the EAS_SetPolyphony API, but you cannot increase it beyond maxVoices.

numChannels: The number of output channels (1 for mono, 2 for stereo). This value is also set at compile-time and cannot be changed.

sampleRate: The output sample rate, also a compile-time parameter that cannot be changed.

mixBufferSize: The size of the output buffer in samples. To calculate the output buffer size in bytes, multiply by numChannels and sizeof(EAS_PCM), which can be used to determine the size of buffer to allocate in the dynamic memory model. If you are using the static memory model, we suggest you check this value against the size of your statically allocated buffer in your checked build.

buildTimeStamp: The timestamp for the library build compatible with the ANSI ctime function.

buildGUID: A 128-bit globally unique identifier (GUID) in ASCII format that uniquely identifies the build. You may be asked for this number when identifying a problem so that we can track it to the specific build that you received.

4.1.5 EAS_SetMaxLoad

Sets the maximum work load of the parser for a given audio frame which helps to balance the workload across audio frames. By default, maxLoad is set to zero which puts no restrictions on the parser. The units are arbitrary and will require some experimentation to find the optimum tradeoff between sound quality and workload balancing. A good value to start with is 300 – and lower values willl reduce the peak load. See the Performance Tuning section for more detail.

```
EAS_RESULT EAS_SetMaxLoad(EAS_DATA_HANDLE pEASData, EAS_I32 maxLoad);
```

4.1.6 EAS_SetParameter

This function is a generic interface for setting parameters in the optional modules. The module parameter is an enumerated type E_FX_MODULES defined in eas.h. param is an enumerated type defined in the module header file for the specific module. For example, the enhancer parameters can be found as E_ENHANCER_PARAMS in eas_enhancer.h. The value is specific to the module and parameter under control.

```
EAS_RESULT EAS_SetParameter(EAS_DATA_HANDLE pEASData, EAS_I32 module,
   EAS_I32 param, EAS_I32 value);
```

All effects have a bypass control as their first parameter. A zero value will enable processing, a non-zero value will disable processing. For example, to bypass the enhancer effect and disable processing, make the following call:

```
if (EAS_SetParameter(pEASData, EAS_MODULE_ENHANCER,
      EAS_PARAM_ENHANCER_BYPASS, EAS_TRUE) != EAS_SUCCESS)
      printf("Error occurred\n");
```

4.1.7 EAS_GetParameter

This function is a generic interface for retrieving parameters in the optional modules. The module parameter is an enumerated type E_FX_MODULES defined in eas.h. param is an enumerated type defined in the module header file for the specific module. For example, the enhancer parameters can be found as E_ENHANCER_PARAMS in eas_enhancer.h. The value is specific to the module and parameter under control.

```
EAS_RESULT EAS_GetParameter(EAS_DATA_HANDLE pEASData, EAS_I32 module,
   EAS_I32 param, EAS_I32 value);
```

All effects have a bypass control as their first parameter. A zero value will enable processing, a non-zero value will disable processing. For example, to retrieve the current value of the bypass setting of the enhancer effect, make the following call:

```
EAS_I32 value;
if (EAS_GetParameter(pEASData, EAS_MODULE_ENHANCER,
      EAS_PARAM_ENHANCER_BYPASS, &value) != EAS_SUCCESS)
      printf("Error occurred\n");
```

4.2 Stream Functions

These functions control the operation of individual audio streams during rendering. This includes opening a new file for rendering, pause, resume, locate, polyphony and priority control.

4.2.1 EAS_OpenFile

Opens a particular file that is to be played by the EAS synth. Pass in pEASData the data handle that was obtained from EAS_Init, a file locator of some sort (typically the file name, but it can be anything, since the value is merely passed to the host wrapper function EAS_HWOpenFile, which you can implement in any fashion you desire) and the address of a handle of type EAS_HANDLE. The returned handle (hereafter referred to as a stream handle) uniquely identifies the instance data for the file and is used by other public interface routines.

```
EAS_RESULT EAS_OpenFile(EAS_DATA_HANDLE pEASData, EAS_FILE_LOCATOR
   locator, EAS_HANDLE *pStreamHandle);
```

4.2.2 EAS_Prepare

This function does initial parsing for the first frame of audio that is about to be rendered, and any last minute initialization for the synth. Pass in pEASData obtained from EAS_Init, and the stream handle obtained from EAS_OpenFile. NOTE: The polyphony parameter has been eliminated in Version 3.5 and later. Use the EAS_SetPolyphony function to set the stream polyphony.

```
EAS_RESULT EAS_Prepare(EAS_DATA_HANDLE pEASData, EAS_HANDLE
   streamHandle);
```

4.2.3 EAS_CloseFIle

Closes a particular file that was previously opened via EAS_OpenFile. Pass in pEASData obtained from EAS_Init and the handle obtained from EAS_OpenFile. This call releases the internal resources associated with the file and calls the host wrapper function EAS_HWCloseFile, which you can implement in any fashion you desire.

```
EAS_RESULT EAS_CloseFile(EAS_DATA_HANDLE pEASData, EAS_HANDLE
   streamHandle);
```

4.2.4 EAS_State

This function retrieves the current parser state. Pass in pEASData obtained from EAS_Init, the handle obtained from EAS_OpenFile, and the address of a state variable (of type EAS_STATE).

```
EAS_RESULT EAS_State(EAS_DATA_HANDLE pEASData, EAS_HANDLE streamHandle,
   EAS_STATE *pState);
```

The returned state value will tell you when the state has reached a value of EAS_STATE_STOPPED or EAS_STATE_ERROR. When the state is EAS_STATE_STOPPED, the file has been completely played. If the state is EAS_STATE_ERROR or the result is not equal to EAS_SUCCESS, an error has occurred, and file playback has been stopped prematurely.

4.2.5 EAS_Locate

This function is used to set desired offset into sequence, in ms. Pass in pEASData (obtained from EAS_Init), the stream handle obtained from EAS_OpenFile, the desired time in milliseconds, and a boolean flag. The boolean flag, if the value is equal to EAS_TRUE, causes the time to be treated as an offset from the current location, rather than an absolute time.

```
EAS_RESULT EAS_Locate(EAS_DATA_HANDLE pEASData, EAS_HANDLE streamHandle,
   EAS_I32 milliseconds, EAS_BOOL offset);
```

4.2.6 EAS_GetLocation

This function is used to get the current playback offset into sequence, in ms. Pass in pEASData (obtained from EAS_Init), the stream handle obtained from EAS_OpenFile, and the address of the destination for the returned time in milliseconds.

```
EAS_I32 EAS_GetLocation(EAS_DATA_HANDLE pEASData, EAS_HANDLE
   streamHandle, EAS_I32 *pTime);
```

4.2.7 EAS_Pause

Tells the synth to pause until EAS_Resume is called. Pass in pEASData obtained from EAS_Init, and the stream handle obtained from EAS_OpenFile.After calling EAS_Pause you should continue to call EAS_Render until the parser state changes to EAS_STATE_PAUSED (call EAS_Stateto retrieve the parser state). The audio output is ramped down gradually to prevent clicking or popping noises during this time. After the parser is fully paused, EAS_Render will return zero filled buffers until EAS_Resume is called, or you can stop calling EAS_Render until you call EAS_Resume.

```
EAS_RESULT EAS_Pause(EAS_DATA_HANDLE pEASData, EAS_HANDLE streamHandle);
```

4.2.8 EAS_Resume

Resume playback of the specified stream. See EAS_Pause for more detail.

```
EAS_RESULT EAS_Resume(EAS_DATA_HANDLE pEASData, EAS_HANDLE streamHandle)
```

4.2.9 EAS_SetPriority

Sets the stream priority. The range for priority is 1-15 and default setting is 5. See the Priority section for more information on how stream priority works.

```
EAS_RESULT EAS_SetPriority(EAS_DATA_HANDLE pEASData, EAS_HANDLE
    streamHandle, EAS_I32 priority);
```

4.2.10 EAS_GetPriority

Retrieves the stream priority. Set EAS_SetPriority for more information.

```
EAS_RESULT EAS_GetPriority(EAS_DATA_HANDLE pEASData, EAS_HANDLE
    streamHandle, EAS_I32 *pPriority);
```

4.2.11 EAS_SetRepeat

Controls the repeat function of the file parser. By default, the parser will play through the file once and stop. Call EAS_SetRepeat after EAS_OpenFile passing pEASData obtained from EAS_Init) the stream handle (obtained from EAS_OpenFile), and repeatCount to change the repeat function. If repeatCount is set to -1, the file will repeat forever, or until you change it by calling EAS_SetRepeat again. If repeatCount is set to 0, the file will not repeat. If repeatCount is set to a positive non-zero value *n*, the file will repeat *n* times, i.e. it will play *n+1* times.

```
EAS_RESULT EAS_SetRepeat(EAS_DATA_HANDLE pEASData, EAS_HANDLE
    streamHandle, EAS_I32 repeatCount);
```

4.2.12 EAS_SetPlaybackRate

Controls the relative tempo of the playback. This function currently only works for SMF files and will return an error if called for a different file type. Rate is a 32-bit fixed point value with the lower 28-bits as a fraction. To playback at normal speed, set the rate parameter to 0x10000000. To playback at 150%, set the rate parameter to 0x18000000. To playback at half speed, set the rate parameter to 0x08000000. It is not recommended to set the rate value higher than 150% speed as increasing the playback speed increases the CPU usage.

```
EAS_RESULT EAS_SetPlaybackRate(EAS_DATA_HANDLE pEASData, EAS_HANDLE
    streamHandle, EAS_U32 rate);
```

4.2.13 EAS_SetTransposition

Modifies the relative key of the file being played. The transposition parameter is in semitones – a positive number will increase the pitch and a negative number will decrease the pitch, and zero returns to the original key. All notes currently playing are muted to prevent stuck notes or pitch conflicts.

```
EAS_RESULT EAS_SetTransposition(EAS_DATA_HANDLE pEASData, EAS_HANDLE
   streamHandle, EAS_I32 transposition);
```

4.3 MIDI Stream Functions

EAS supports a real-time MIDI streaming function. These functions can be used to stream live MIDI data into the synthesizer engine for game play or other real-time activities. The MIDI stream API works similarly to the MIDI file playback API, except that MIDI events are “pushed” to the stream engine, whereas the file playback “pulls” events from the file through the host wrapper.

To render a live MIDI stream, call the EAS_OpenMIDIStream API call, which returns a stream handle. After opening the stream, begin calling EAS_Render to render audio. To send MIDI data to the rendering engine, call the EAS_WriteMIDIStream API using the stream handle, passing the buffer pointer to the MIDI data along with the size of the data in bytes. MIDI data must conform to the MIDI protocol as specified in the MIDI specification[1]. To minimize call overhead, make a single call to EAS_WriteMIDIStream just before calling EAS_Render passing all the MIDI data accumulated since the last call to EAS_Render.

When the MIDI stream is no longer needed, call EAS_CloseMIDIStream with the stream handle to close it. Even though all notes will be automatically released at the time EAS_CloseMIDIStream is called, it may take some time for them to decay to zero. To avoid audio artifacts caused by prematurely cutting off the audio, it is generally prudent to call EAS_Render a few more times after closing the stream.

In Version 3.5 and later, EAS_OpenMIDIStream now accepts a stream handle as a parameter. If the stream handle is NULL, a new synthesizer instance is created under the sole control of the real-time MIDI stream. If the stream handle is a handle returned by EAS_OpenFile, the synthesizer is shared with the file and can be used to manipulate the synthesizer used by the file in real-time. This capability is available to support the MIDIControl interface in the Java MMAPI. Note that when a synthesizer is shared between a file and real-time MIDI interface, both stream handles must be closed via separate calls to EAS_CloseFile and EAS_CloseMIDIStream to release the instance of the virtual synthesizer.

4.3.1 EAS_OpenMIDIStream

Opens a MIDI stream for real-time MIDI event processing. Pass in pEASData, the handle that was obtained from EAS_Init call and an optional stream handle. Returns a new MIDI stream handle in the variable pointed to by pHandle. If streamHandle is NULL, a new instance of the synthesizer is created. If streamHandle is a handle returned by EAS_OpenFile, the real-time MIDI stream will use the same synthesizer as the file opened by EAS_OpenFile.

```
EAS_RESULT EAS_OpenMIDIStream(EAS_DATA_HANDLE pEASData, EAS_HANDLE
   *pMIDIHandle, EAS_HANDLE streamHandle);
```

4.3.2 EAS_WriteMIDIStream

Streams MIDI data into the rendering engine. Pass in pEASData, the handle obtained from EAS_Init, the stream handle obtained from OpenMIDIStream, pBuffer – a pointer to the MIDI stream data, and count – the number of bytes in the buffer.

```
EAS_RESULT EAS_WriteMIDIStream(EAS_DATA_HANDLE pEASData, EAS_HANDLE
   midiHandle, EAS_U8 *pBuffer, EAS_I32 count);
```

4.3.3 EAS_CloseMIDIStream

Closes the MIDI stream and releases all notes that may still be playing. Pass in pEASData, the handle obtained from EAS_Init, and the stream handle returned by EAS_OpenMIDIStream. If notes are still playing at the time EAS_CloseMIDIStream is called, it is advised that you continue calling EAS_Render a few more times after closing the stream to ensure that notes are properly released in order to prevent clicks and other audio artifacts.

```
EAS_RESULT EAS_CloseMIDIStream(EAS_DATA_HANDLE pEASData, EAS_HANDLE
   midiHandle);
```

4.4 Volume Control

These functions are for setting and retrieving the master volume as well as the volume of individual streams.

4.4.1 EAS_SetVolume

This function controls volume in 1dB increments. Valid values are 0 – 100 where 100 represents maximum volume, and 99 represents 1dB below maximum volume, and 0 is no output.

```
EAS_RESULT EAS_SetVolume(EAS_DATA_HANDLE pEASData, EAS_HANDLE
   streamHandle, EAS_I32 volume);
```

If streamHandle is NULL, this function controls the master volume and will affect all streams playing. The default setting for the master volume is 90, or -10dB below maximum output.

If streamHandle is a handle returned by EAS_OpenFile, this function controls the output volume of the individual stream in 1dB increments. The default setting for the stream volume is 100.

Be aware that the output levels of song files can vary significantly, and clipping can occur at volumes even below maximum volume, depending on specific file contents.

4.4.2 EAS_GetVolume

Returns the current volume setting. Pass in pEASData obtained from EAS_Init. If streamHandle is NULL, the returned value is the master volume setting. If streamHandle is a handle returned by EAS_OpenFile, the value returned is the stream volume setting.

```
EAS_I32 EAS_GetVolume(EAS_DATA_HANDLE pEASData, EAS_HANDLE streamHandle);
```

4.5 Polyphony Control

These functions control the polyphony of the synthesizer as well as the polyphony of individual streams.

4.5.1 EAS_SetSynthPolyphony

This function is used to control the maximum synth polyphony. It allows the polyphony to be set to any value between 1 and the maximum that the library was built for. Use pConfig-> maxVoices to determine the maximum for this particular library. pEASData (obtained from EAS_Init), handle (obtained from EAS_OpenFile), and the desired polyphony count.

```
EAS_RESULT EAS_SetSynthPolyphony(EAS_DATA_HANDLE pEASData, EAS_I32
   synthNum, EAS_I32 polyphonyCount);
```

The parameter synthNum selects the synthesizer to be updated. For standard configurations, always use the enumerated value EAS_MCU_SYNTH. For split architecture configurations, the enumerated value EAS_DSP_SYNTH can be used to control the secondary synth in the DSP. See the section on Streams, Polyphony, and Priority for more background on the use of this function.

4.5.2 EAS_GetSynthPolyphony

Returns the current polyphony setting. Pass in pEASData obtained from EAS_Init. The parameter synthNum selects the synthesizer to be queried. For standard configurations, always use the enumerated value EAS_MCU_SYNTH. For split architecture configurations, the enumerated value EAS_DSP_SYNTH can be used to query the secondary synth in the DSP.

```
EAS_I32 EAS_GetSynthPolyphony(EAS_DATA_HANDLE pEASData, EAS_I32
    synthNum);
```

4.5.3 EAS_SetPolyphony

This function is used to control the maximum polyphony of the stream. Pass in pEASData obtained from EAS_Init, and the stream handle obtained from EAS_OpenFile. Passing a polyphonyCount of 0 allows the stream to use all the available voices in the synthesizer as determined by EAS_SetSynthPolyphony. Smaller values will restrict the stream to fewer voices. See the section on Streams, Polyphony, and Priority for more background on the use of this function.

```
EAS_RESULT EAS_SetPolyphony(EAS_DATA_HANDLE pEASData, EAS_HANDLE
    streamHandle, EAS_I32 polyphonyCount);
```

4.5.4 EAS_I32 EAS_GetPolyphony (EAS_DATA_HANDLE pEASData, EAS_HANDLE streamHandle)

Returns the current polyphony setting for the stream. Pass in pEASData obtained from EAS_Init, and the stream handle obtained from EAS_OpenFile.

4.6 Metadata

EAS allows the host application to retrieve metadata from the song file for display purposes. Because every song file may contain different metadata, the metadata functions have been standardized on a few specific metadata types that are most useful for mobile applications. Not all song files will contain metadata, and even when they do, they may not contain all the supported metadata types.

Metadata can be retrieved before rendering the song file, or during the rendering process. The host application first calls EAS_OpenFile to open the target song file. Next, the host must register a callback function, buffer for storing the metadata, and optional user data by calling EAS_RegisterMetaDataCallback. The host can then call the metadata retrieval function EAS_ParseMetaData to retrieve all the metadata from the file, or it can render the file, and the metadata will be extracted as it is encountered in real-time. For sample code, see xxxx.

4.6.1 RegisterMetaDataCallback

Registers a metadata callback function and buffer for the metadata. The pointer metaDataBuffer points to a char buffer of size metaDataBufSize. The prototype for the metadata callback function is as follows:

```
void MetaDataCallback (EAS_I32 metaDataType, char *buffer);
EAS_RESULT EAS_RegisterMetaDataCallback(EAS_DATA_HANDLE pEASData,
    EAS_HANDLE streamHandle, EAS_METADATA_CBFUNC cbFunc, char
    *metaDataBuffer, EAS_I32 metaDataBufSize, EAS_VOID_PTR pUserData);
```

The callback function may be called from the API functions EAS_Prepare, EAS_Render, or EAS_ParseMetaData. The metaDataType is an enumerated type defined in eas_types.h called E_EAS_METADATA_TYPE. It is set according to the type of data found in the file. Some song files, such as MIDI, have ambiguous metadata specifications. The parser will attempt a best fit for the metadata it encounters in the file.

The buffer will contain a null-terminated ASCII string. The null is included in the character count, so to retrieve 20 characters, the buffer must specified as 21 bytes. Only ASCII is supported at this time (no Unicode or other characters).

pUserData is a host supplied void pointer that is passed transparently back in the metadata callback. The value is unique to each stream, providing a mechanism for differentiating the metadata from each stream if multiple streams are active at the same time.

4.6.2 EAS_ParseMetaData

Instructs the parser to parse the file for metadata. This call may result in multiple calls to the registered metadata callback function as the parser encounters metadata in the file. It returns the length of the song file in milliseconds in the variable pointed to by pPlayLength.

```
EAS_RESULT EAS_ParseMetaData(EAS_DATA_HANDLE pEASData, EAS_HANDLE
   streamHandle, EAS_I32 *pPlayLength);
```

4.6.3 EAS_GetFileType

Retrieves the file type from the stream parser. The file type enumerations can be found as E_EAS_FILE_TYPE in the eas_types.h file.

```
EAS_RESULT EAS_GetFileType (EAS_DATA_HANDLE pEASData, EAS_HANDLE
   streamHandle, EAS_I32 *pFileType);
```

4.7 Miscellaneous

4.7.1 EAS_SetHeaderSearchFlag

Sets the header search mode flag – default is EAS_FALSE. By default, EAS checks the first few bytes of the file to determine the file type. In some applications, SMF and SMAF files may be embedded inside another file. In this case, the normal EAS header search routines will fail to recognize the file.

If *searchFlag* is set to EAS_TRUE, after the normal search fails, EAS will search the entire file in an attempt to find SMF and SMAF headers embedded in the file. Once negative side effect of setting *searchFlag* to true is that it may take EAS longer to process the EAS_OpenFile() request. It may be necessary to set *searchFlag* to true in order to meet MFi requirements that the terminal recognize an SMF file embedded in a file.

```
EAS_PUBLIC EAS_RESULT EAS_SetHeaderSearchFlag (EAS_DATA_HANDLE pEASData,
   EAS_BOOL searchFlag)
```

4.7.2 EAS_SetPlayMode

Sets the playback mode (currently restricted to MFi files). Call this function after EAS_Prepare() to set the playback mode to partial play mode. The modes are enumerated in “eas_types.h” as IMODE_PLAY_ALL and IMODE_PLAY_PARTIAL. This corresponds to the cue/jump points in the MFi file format.

5 Configuration Module (CM)

The file eas_config.c contains the Configuration Module (CM), which provides a mechanism whereby you can utilize the desired features in an EAS library module, without requiring a custom library for your particular needs. It allows the library to find optional components, and links to static memory allocations if appropriate (static memory model only).

The CM should be used without modification – in fact, unapproved modifications may cause unexpected problems. Details of the inner workings of the CM are beyond the scope of this document and are not necessary to use the EAS library. The only information required to use the CM is the following pre-processor defines that allow you to tailor the EAS library to your application.

5.1 CM Preprocessor Defines

5.1.1 _STATIC_MEMORY

If this symbol is defined, the EAS library will use only static memory, and no dynamic memory allocation will be utilized (except in eas_main.c or eas_wave.c). If this symbol is **not** defined, dynamic memory allocation will be utilized via the Host Wrapper (HW) interface.

Note: The host source code examples (eas_main.c and eas_wave.c) call malloc and free directly due to issues with the availability of the host wrapper instance data handle (EAS_HW_DATA_HANDLE). If you use this code as the starting point for your implementation, you may need to redirect these to appropriate system functions, or use a statically allocated memory object.

5.2 Parser Options

5.2.1 _CMX_PARSER

If this symbol is defined, the CM will attempt to link in the optional CMX parser module. If the library does not include the CMX module and this symbol is defined, a linker error will occur.

5.2.2 _IMELODY_PARSER

If this symbol is defined, the CM will attempt to link in the optional iMelody parser module from the EAS library. If the library does not include the iMelody module and this symbol is defined, a linker error will occur

If the symbol is not defined, the CM will **not** attempt to link in the optional iMelody parser. You can reduce memory usage by not defining this symbol if the iMelody parser is not required for a particular application.

5.2.3 _MFI_PARSER

If this symbol is defined, the CM will attempt to link in the optional MFi parser module. If the library does not include the MFi module and this symbol is defined, a linker error will occur.

5.2.4 _OTA_PARSER

If this symbol is defined, the CM will attempt to link in the optional OTA parser module from the EAS library. If the library does not include the OTA module and this symbol is defined, a linker error will occur

If the symbol is not defined, the CM will **not** attempt to link in the optional OTA parser. You can reduce memory usage by not defining this symbol if the OTA parser is not required for a particular application.

5.2.5 _RTTTL_PARSER

If this symbol is defined, the CM will attempt to link in the optional RTTTL/RTX parser module from the EAS library. If the library does not include the RTTTL module and this symbol is defined, a linker error will occur

If the symbol is not defined, the CM will **not** attempt to link in the optional RTTTL parser. You can reduce memory usage by not defining this symbol if the RTTTL parser is not required for a particular application.

5.2.6 _SMAF_PARSER

If this symbol is defined, the CM will attempt to link in the optional SMAF parser from the EAS library. If the library does not include the SMAF module and this symbol is defined, a linker error will occur

If the symbol is not defined, the CM will **not** attempt to link in the optional SMAF parser. You can reduce memory usage by not defining this symbol if the SMAF parser is not required for a particular application.

5.2.7 _WAVE_PARSER

If this symbol is defined, the CM will attempt to link in the optional WAVE file parser module from the EAS library. If the library does not include the WAVE module and this symbol is defined, a linker error will occur.

If the symbol is not defined, the CM will **not** attempt to link in the optional WAVE file parser. You can reduce memory usage by not defining this symbol if the WAVE file parser is not required for a particular application.

5.2.8 _XMF_PARSER

If this symbol is defined, the CM will attempt to link in the optional Mobile XMF/DLS parser module. If the library does not include the Mobile XMF/DLS module and this symbol is defined, a linker error will occur.

5.3 Effects Options

5.3.1 _ENHANCER_ENABLED

If this symbol is defined, the CM will attempt to link in the optional Enhancer effect module from the EAS library. If the library does not include the Enhancer module and this symbol is defined, a linker error will occur

If the symbol is not defined, the CM will **not** attempt to link in the optional Enhancer effect. You can reduce memory usage by not defining this symbol if the Enhancer effect is not required for a particular application.

5.3.2 _COMPRESSOR_ENABLED

If this symbol is defined, the CM will attempt to link in the optional Compressor effect module from the EAS library. If the library does not include the Compressor module and this symbol is defined, a linker error will occur.

If the symbol is not defined, the CM will **not** attempt to link in the optional Compressor effect. You can reduce memory usage by not defining this symbol if the Compressor effect is not required for a particular application.

5.3.3 _WOW_ENABLED

If this symbol is defined, the CM will attempt to link in the optional SRS WOW® XT interface module. This option requires that you add the appropriate WOW XT library to your build. For more details, refer to 12.2 for more detail.

6 Host Wrapper (HW) Interface

We provide public wrappers for all functions that might need to be modified for a particular host implementation. The wrapper functions include file IO, dynamic memory, as well as a few more specialized ones. By providing the source code for these functions, we allow you to modify them to work as you need them to. We will provide support as needed to help customize these routines for your system.

The file eas_host.c contains the various wrapper functions, with a working default implementation, which you may need to modify. The file I/O wrapper functions are mapped to ANSI C standard file I/O functions. If your system does not support these functions, you will need to map them to the appropriate functions.

EAS_HW_DATA_HANDLE is a data type that points to persistent data used by the host wrapper module. This data type is opaque to EAS, i.e. EAS accepts the handle it receives from the EAS_HWInit call and it passes it back each time it calls a host wrapper function. The handle is stored in the persistent data pointed to be EAS_DATA_HANDLE, so it can be unique to each instance of EAS if necessary.

The file eas_hostmm.c provides an alternate default implementation, with memory mapped versions of the fileIO routines. As each file is opened, a buffer is allocated and the entire file is copied into memory, and all subsequent file operations are performed on the memory buffer. If you have sufficient system memory to support this and are using dynamic memory allocation, it is considerably more efficient than using file I/O, particularly for SMF1 and SMAF files. Note that eas_hostmm.c will not work with the _STATIC_MEMORY option enabled.

If your song files are stored in memory, you can remove the file I/O references in the host wrapper and use the file locator to point directly to the memory location where the song file can be found. The file locator is an opaque data object to the EAS library itself – it can be used in any way you see fit, provided the handle value itself remains constant from the time EAS_OpenFile is called until EAS_CloseFile is called.

6.1 Initialization and Shutdown

6.1.1 EAS_HWInit

Initializes the host wrapper interface. Allocates any memory that might be needed by the host wrapper (dynamic memory model only). Returns a EAS_HW_DATA_HANDLE which is then passed in to any of the other host wrapper functions. This can be useful if multiple instances of the EAS library need to operate independently.

```
EAS_RESULT EAS_HWInit (EAS_HW_DATA_HANDLE *pHWInstData);
```

6.1.2 EAS_HWShutdown

Shuts down the host wrapper interface. Frees any memory that might have been allocated (dynamic memory model only). Parameter is hwInstData obtained from EAS_HWInit.

```
EAS_RESULT EAS_HWShutdown(EAS_HW_DATA_HANDLE hwInstData);
```

6.2 Memory Functions

6.2.1 EAS_HWMalloc

Wrapper function for malloc.Parameters are hwInstData obtained from EAS_HWInit and the number of bytes required. Like malloc, will return a NULL pointer if no memory is available (or if dynamic memory allocation is disabled). This function need not be supported if the _STATIC_MEMORY option is enabled.

```
void *EAS_HWMalloc(EAS_HW_DATA_HANDLE hwInstData, EAS_I32 size);
```

6.2.2 EAS_HWFree

Wrapper function for free.Parameters are hwInstData obtained from EAS_HWInit and a pointer to memory allocated via EAS_HWMalloc. This function need not be supported if the _STATIC_MEMORY option is enabled.

```
void EAS_HWFree(EAS_HW_DATA_HANDLE hwInstData, void *p);
```

6.2.3 EAS_HWMemCpy

Wrapper function for memcpy. Parameters are a pointer to the destination buffer, a pointer to the source buffer, and the number of bytes to copy.

```
void *EAS_HWMemCpy(void *dest, const void *src, EAS_I32 amount);
```

6.2.4 EAS_HWMemSet

Wrapper function for memset. Parameters are a pointer to the destination buffer, the value to use for setting, and the number of bytes to set.

```
void *EAS_HWMemSet(void *dest, int val, EAS_I32 amount);
```

6.3 File I/O Functions

6.3.1 EAS_HWOpenFile

Wrapper function for fopen. Opens a file. Parameters are hwInstData obtained from EAS_HWInit, locator (typically the file name, but it can be anything, since the value is really a void * which you can use in any fashion you desire to identify the file you want to open), the address of a variable of type EAS_HANDLE (which allows this function to return a file handle), and mode.

```
EAS_RESULT EAS_HWOpenFile(EAS_HW_DATA_HANDLE hwInstData, EAS_FILE_LOCATOR
   locator, EAS_HANDLE *pHandle, EAS_FILE_MODE mode);
```

Possible values for mode are below:

```
#define EAS_FILE_READ  1
#define EAS_FILE_WRITE 2
```

The file handle returned by this function (herein referred to as handle) is passed to most of the other host wrapper functions. The EAS_FILE_WRITE mode is currently not used by any EAS library modules, and is included only for future compatibility.

6.3.2 EAS_HWCloseFile

Wrapper function for fclose.Closes a file opened with EAS_HWOpen. Parameters are hwInstData (obtained from EAS_HWInit), and handle (obtained from EAS_HWOpenFile).

```
EAS_RESULT EAS_HWCloseFile (EAS_HW_DATA_HANDLE hwInstData, EAS_HANDLE
   handle);
```

6.3.3 EAS_HWReadFile

Wrapper function for fread. Reads data from a file opened with EAS_HWOpen. Parameters are hwInstData obtained from EAS_HWInit, handle (obtained from EAS_HWOpenFile), a pointer to the destination data buffer, the number of bytes requested, and the address of a count values to return the actual number read.

```
EAS_RESULT EAS_HWReadFile(EAS_HW_DATA_HANDLE hwInstData, EAS_HANDLE
   handle, void *pBuffer, EAS_I32 n, EAS_I32 *pBytesRead);
```

6.3.4 EAS_HWGetByte

Wrapper function for fgetc.Reads one byte of data from a file opened with EAS_HWOpen. Parameters are hwInstData obtained from EAS_HWInit, handle obtained from EAS_HWOpenFile, and a pointer to the destination data buffer.

```
EAS_RESULT EAS_HWGetByte(EAS_HW_DATA_HANDLE hwInstData, EAS_HANDLE
   handle, void *p);
```

6.3.5 EAS_HWGetWord

Reads two byte quantity from a file opened with EAS_HWOpen. Parameters are hwInstData obtained from EAS_HWInit, handle obtained from EAS_HWOpenFile, a pointer to the destination data buffer, and a boolean that controls byte order.

```
EAS_RESULT EAS_HWGetWord(EAS_HW_DATA_HANDLE hwInstData, EAS_HANDLE
   handle, void *p, EAS_BOOL msbFirst);
```

6.3.6 EAS_HWGetDWord

Reads four byte quantity from a file opened with EAS_HWOpen. Parameters are hwInstData obtained from EAS_HWInit, handle obtained from EAS_HWOpenFile, a pointer to the destination data buffer, and a boolean that controls byte order.

```
EAS_RESULT EAS_HWGetDWord(EAS_HW_DATA_HANDLE hwInstData, EAS_HANDLE
   handle, void *p, EAS_BOOL msbFirst);
```

6.3.7 EAS_HWFilePos

Wrapper function for ftell. Reports the current byte-based read/write offset in a file opened with EAS_HWOpen. Parameters are hwInstData obtained from EAS_HWInit, handle obtained from EAS_HWOpenFile, and a pointer to variable that will get the returned offset value.

```
EAS_RESULT EAS_HWFilePos(EAS_HW_DATA_HANDLE hwInstData, EAS_HANDLE
   handle, EAS_I32 *pPosition);
```

6.3.8 EAS_HWFileSeek

Wrapper function for fseek. Sets the desired byte-based read/write offset in a file opened with EAS_HWOpen. Parameters are hwInstData obtained from EAS_HWInit, handle obtained from EAS_HWOpenFile, and the desired offset value.

```
EAS_RESULT EAS_HWFileSeek(EAS_HW_DATA_HANDLE hwInstData, EAS_HANDLE
   handle, EAS_I32 position);
```

6.3.9 EAS_HWFileLength

Provides the length in bytes of a file opened with EAS_HWOpen. Parameters are hwInstData (obtained from EAS_HWInit), handle (obtained from EAS_HWOpenFile), and a pointer to variable that will get the returned file length.

```
EAS_RESULT EAS_HWFileLength(EAS_HW_DATA_HANDLE hwInstData, EAS_HANDLE
   handle, EAS_I32 *pLength);
```

6.3.10 EAS_HWDupHandle

Duplicates a file handle which refers to a file opened with EAS_HWOpen. Parameters are hwInstData (obtained from EAS_HWInit), handle (obtained from EAS_HWOpenFile), and a pointer to a variable that will get the duplicated handle.

```
EAS_RESULT EAS_HWDupHandle(EAS_HW_DATA_HANDLE hwInstData, EAS_HANDLE
   handle, void **pHandle);
```

6.4 Hardware Functions

6.4.1 EAS_HWVibrate

Provides a hook for vibrator functionality. Parameters are hwInstData (obtained from EAS_HWInit), handle (obtained from EAS_HWOpenFile), and a boolean value. Possible values are EAS_TRUE and EAS_FALSE. The EAS library will call the function during render time to enable or disable the vibrator as directed by the audio file.

```
EAS_RESULT EAS_HWVibrate(EAS_HW_DATA_HANDLE hwInstData, EAS_BOOL state);
```

6.4.2 EAS_HWLED

Provides a hook for LED functionality. Parameters are hwInstData (obtained from EAS_HWInit), handle (obtained from EAS_HWOpenFile), and a boolean value. Possible values are EAS_TRUE and EAS_FALSE. The EAS library will call the function during render time to enable or disable the LED as directed by the audio file.

```
EAS_RESULT EAS_HWLED(EAS_HW_DATA_HANDLE hwInstData, EAS_BOOL state);
```

6.5 Performance Functions

6.5.1 EAS_HWYield

Provides a hook to limit file parsing based on system load. Parameter is hwInstData (obtained from EAS_HWInit).

```
EAS_BOOL EAS_HWYield(EAS_HW_DATA_HANDLE hwInstData);
```

This function is called periodically by the EAS library to give the host an opportunity to allow other tasks to run. There are two ways to use this call:

If you have a multi-tasking OS, you can call the yield function in the OS to allow other tasks to run. In this case, return EAS_FALSE to tell the EAS library to continue processing when control returns from this function.

If tasks run in a single thread by sequential function calls (sometimes call a "commutator loop"), return EAS_TRUE to cause the EAS Library to return to the caller. Be sure to check the number of bytes rendered before passing the audio buffer to the codec - it may not be filled. The next call to EAS_Render will continue processing until the buffer has been filled.

7 Memory Models

The EAS library can work with either static or dynamic memory allocation models. The default model is a dynamic memory model, where all instance data is allocated through the host wrapper function EAS_HWMalloc and released through the wrapper function EAS_HWFree. The default implementation in the host wrapper modules eas_host.c and eas_hostmm.c maps these to the ANSI C malloc and free functions. If your system does not support these functions, you will need to modify the implementation to call the appropriate functions.

To use the static memory model, define the pre-processor symbol _STATIC_MEMORY when you build the eas_config.c module. In this case, the eas_config module will provide static links to pre-allocated data structures included in the EAS library. There are some limitations to the static memory model due to the fact that data objects must be pre-allocated. If you are using the static memory model, the default implementation of eas_hostmm.c will not work due to dependencies on dynamic memory. Use the eas_host.c module instead.

Note that some optional modules may **require** the use of dynamic memory. In this case, an error message will be generated while compiling the eas_config.c module if the _STATIC_MEMORY symbol is defined. See the documentation on the optional modules for more information.

8 Debug Message Reporting

The file eas_report.c provides simple routines that provide a mechanism for debug messages from the EAS synthesizer to be displayed. To keep the production image as small as possible, we use a pre-processing step that strips debug reporting from the EAS library, leaving only the debug messages in the sample client source code. If you replace the debug calls in the application code, you do not need to include the debug module.

In rare instances, where we are unable to duplicate a problem you report, we may ask you to use the debug module to capture output from a debug library build that we supply you. The debug routines are similar to the host wrapper routines in that they are called by the library instead of direct calls to printf. This allows you to redirect output to a JTAG interface or serial port based on the functionality of your hardware or simulation environment.

The default implementation of the module calls the ANSI C library function vsprintf to format the data for output. In some ANSI C library implementations, this may also pull in floating point code and other modules that significantly increase the size of the executable.

8.1 Reporting Functions

8.1.1 EAS_ReportEx

This version is used if the preprocessor define _NO_DEBUG_PROCESSOR is **not** defined, which means that the source files have been pre-processed with our tool to consolidate all debug messages, to make code release ready.

```
void EAS_ReportEx(int severity, unsigned long hashCode, int serialNum,
    ...);
```

8.1.2 EAS_Report

This version is used is the preprocessor define _NO_DEBUG_PROCESSOR is defined, which means that the code is still in development. It will automatically display the line number and module name when it is later converted to EAS_ReportEx by our debug preprocessor.

```
void EAS_Report(int severity, const char *fmt, ...);
```

8.1.3 EAS_ReportX

This version is used if the preprocessor define _NO_DEBUG_PROCESSOR is defined, which means that the code is still in development. It is similar to EAS_Report, but this version will not display line number and module name when it is later converted to EAS_ReportEx by our debug preprocessor.

```
void EAS_ReportX(int severity, const char *fmt, ...);
```

8.1.4 EAS_SetDebugLevel

Allows you to control what severity debug messages will actually be displayed.

```
void EAS_SetDebugLevel(int severity);
```

The following possible severity values are defined:

```
#define _EAS_SEVERITY_NOFILTER       0
#define _EAS_SEVERITY_FATAL   1
#define _EAS_SEVERITY_ERROR   2
#define _EAS_SEVERITY_WARNING        3
#define _EAS_SEVERITY_INFO    4
#define _EAS_SEVERITY_DETAIL  5
```

Setting the severity to ERROR will allow only ERROR and WARNING messages to be displayed.

8.1.5 EAS_SetDebugFile

Optionally allows you to select an output debug file. The default is standard error.

```
void EAS_SetDebugFile(void *file, int flushAfterWrite);
```

9 Performance Tuning

The EAS library provides three different methods to dynamically control CPU usage. These methods give you some control over average CPU usage and peak CPU usage.

9.1 Controlling Average CPU Usage

The most important factor in average CPU usage is the number of active voices. To dynamically reduce CPU usage, you can call EAS_SetSynthPolyphony to reduce the number of active voices. CPU usage scales increases linearly with the number of voices. To control audio artifacts, the number of voices is reduced gradually over a number of audio frames as voices naturally stop during playback.

One method of controlling the number of voices is to implement a control loop that monitors the amount of audio data buffered at the output of the synthesizer. If the buffered data is decreasing, reduce the polyphony to allow the synthesizer to catch up. If it is increasing, increase the polyphony to provide better audio quality.

Another method is to measure the CPU usage of the synthesizer by sampling a real-time clock before and after the call to EAS_Render. If it is taking too long to render, reduce the polyphony, and vice versa.

Note that there are some processor loading costs associated with reducing polyphony, i.e. if the call to EAS_SetSynthPolyphony reduces the number of available voices, there is some load associated with determining which voices will be shutdown to accommodate the reduced polyphony. By contrast, there is almost no overhead associated with increasing the polyphony. Therefore, we suggest that when reducing the polyphony, it is better to reduce aggressively – perhaps more than necessary – and gradually increase the polyphony as needed, rather than repeatedly reduce the polyphony. The control loop should also include some hysteresis to prevent oscillation.

9.2 Controlling Peak CPU Usage

The most important factor in peak CPU usage is the file parser overhead. Song files tend to have clustered events where many notes are started and stopped in a short period of time. This can cause significant variations in peak CPU usage. EAS has a proprietary algorithm for smoothing out these peaks with very little compromise in the quality of the audio. The control for this is the EAS_SetMaxLoad function (see the API call for a more detailed description). By using a smaller maxLoad parameter, you can smooth out the CPU loading so that the peak loads are closer to the average load. We suggest a maxLoad parameter of 300 as a good compromise between audio quality and load averaging. If you reduce it too much, the audio quality will suffer.

9.3 Multi-tasking Operation

Another method of controlling CPU peak usage is through the use of the EAS_HWYield wrapper function. The main parsing loop in the EAS library periodically calls the yield function to allow the client to switch away to other tasks. If you have a full-featured RTOS, this is an opportune place to add a call to the OS yield function.

The audio rendering operation is monolithic at this time and no calls occur to EAS_HWYield while audio rendering is in process. This typically represents the bulk of time spent in the EAS_Render call, so it is important to control average CPU usage through the EAS_SetPolyphony call. See the API reference for specifics on the EAS_HWYield call itself.

10 Wave File Output

The file eas_wave.c provides simple routines that are used by eas_main.c to create .wav files. Typically these routines are only used as part of our test harness (example code in eas_main.c) and would not be needed in a real target system. You may wish to modify these routines to suit your system needs.

10.1 Functions

10.1.1 WaveFileCreate

Opens a .wav file for use as part of the eas test harness.

```
WAVE_FILE *WaveFileCreate(const char *filename, EAS_I32 nChannels,
   EAS_I32 nSamplesPerSec, EAS_I32 wBitsPerSample);
```

10.1.2 WaveFileWrite

Writes audio data to a .wav file as part of the eas test harness.

```
EAS_I32 WaveFileWrite(WAVE_FILE *wFile, void *buffer, EAS_I32 n);
```

10.1.3 WaveFileClose

Closes a .wav file as part of the eas test harness.

```
EAS_BOOL WaveFileClose(WAVE_FILE *wFile);
```

11 Using the EAS Library

The EAS library has a fairly simple interface that consists of a small number of functions that are described in this section.

11.1 Detailed Walkthrough of Interface to EAS Library

Take a look at the code in eas_main.c which will provide you with a simple example of how the various routines work together. Realize that most of the routines mentioned in this section return a value of type EAS_RESULT, which we suggest you check that result after each call.

We recommend that you **call EAS_Config** to get the current configuration from the CM. The configuration information (of type S_EAS_LIB_CONFIG *) that is returned (hereafter referred to as pConfig) allows you to check various library build options. See the API reference for EAS_Config for more detail on the returned configuration data. In particular, you will need to know pConfig->mixBufferSize. Only one call to EAS_Config is needed at the start of time, regardless of any errors that might occur later.

Next optionally **call EASLibraryCheck** passing in the configuration info (pConfig) obtained in the previous step. This provides version checking and displays info about ths current cofiguration. The code for this function is in eas_main.c so you can modify it as needed. For production builds, you may choose to omit the call to EAS_Config and EASLibraryCheck.

Next **calculate** the **output buffer size**, typically:

```
bufferSize = pConfig->mixBufferSize * pConfig->numChannels *
   sizeof(EAS_PCM) * NUM_BUFFERS
```

Realize that the defined value NUM_BUFFERS (in eas_main.c) is the only real control you have over the size of the output buffer, since the other values are fixed by the configuration. You can set NUM_BUFFERS to 1 if desired.

Next **allocate or check** the **output buffer**. You can either use dynamic allocation and allocate it now, or if you use static allocation, or an automatic variable, you can check the size of it using the size calculated in the previous step. (If this buffer is later freed, you will need to allocate it again before playing another file.)

Then **call EAS_Init** to initialize the EAS synth, passing in the address of an EAS_DATA_HANDLE. EAS_Init will assign the data handle to point to the synth's data. The returned handle (hereafter referred to as pEASData) will be used later. This function makes sure that all data is allocated (dynamic memory model only) and initialized. (This will need to be called again if another file is to be played after a call to EAS_Shutdown.)

Loop **for each file** that is to be played:

Call EAS_OpenFile, passing in pEASData (the data handle that was obtained from EAS_Init), a file locator of some sort (typically the file name, but it can be anything, since the value is merely passed to the host wrapper function EAS_HWOpenFile, which you can implement in any fashion you desire) and the address of a file handle. The returned handle (hereafter referred to as handle) will be used later. This function makes sure the file is ready to play.

Call EAS_Prepare, passing in pEASData (obtained from EAS_Init), handle (obtained from EAS_OpenFile), and the desired initial polyphony (should be <= pConfig->maxVoices). This function does initial parsing for the first frame of audio that is about to be rendered, and any last minute initialization for the synth.

In the example code, we call WaveFileCreate, but in a real system you will likely skip this step. Take a look at eas_wave.c and eas_main.c if you need to make use of this function.

Initialize the result to EAS_SUCCESS.

Loop **while the result is EAS_SUCCESS**. At any place inside this loop you can call the various optional public interface functions, such as EAS_Locate, EAS_SetPolyphony, etc.

Call EAS_Render multiple times to fill one complete host buffer. The number of times you will need to call it equals NUM_BUFFERS. Pass in pEASData (the data handle obtained from EAS_Init), a pointer to the current offset in the host buffer, the value of pConfig->mixBufferSize, and the address of a counter (which will return the actual number of output samples rendered). This function performs the actual audio rendering via the synthesizer. The synth calls the appropriate file parser as needed. The file parser calls one or more of the host wrapper functions for any fileIO operations.

If the result is still EAS_SUCCESS, now that we have rendered a complete host buffer, **hand off the rendered audio data**.

In the example code, we call WaveFileWrite, but in a real system you will likely skip this step. Take a look at eas_wave.c and eas_main.c if you need to make use of this function.

Call EAS_State to check the parser state. Pass in pEASData, handle, and the address of a state variable (of type EAS_STATE). The returned state value will tell you when the state has reached a value of EAS_STATE_STOPPED or EAS_STATE_ERROR. When the state is EAS_STATE_STOPPED, the file has been completely played. If the state is EAS_STATE_ERROR or the result is not equal to EAS_SUCCESS, an error has occurred, so file playback has been stopped prematurely. The complete list of possible state values is as follows:

```
#define EAS_STATE_IDLE          0
#define EAS_STATE_READY         1
#define EAS_STATE_PLAY          2
#define EAS_STATE_PAUSING       3
#define EAS_STATE_PAUSED        4
#define EAS_STATE_RESUMING      5
#define EAS_STATE_STOPPING      6
#define EAS_STATE_STOPPED       7
#define EAS_STATE_ERROR         8
```

If no errors have occurred and the file is still being played (meaning state is not equal to EAS_STATE_STOPPED), **loop back to EAS_Render**.

Call EAS_CloseFile, passing pEASData and handle. This function does whatever is needed to close the file.

The example code calls WaveFileClose, but in a real system you will likely skip this step. Take a look at eas_wave.c and eas_main.c if you need to make use of this function.

If no errors have occurred, and another file needs to be played, **loop back to EAS_OpenFile**.

If all files are completed or an error occurs, **call EAS_Shutdown,** passing in pEASData (the data handle obtained from EAS_Init). This function frees any memory that may have been allocated, and does any other needed cleanup.

Finally, **free the output buffer** if it was dynamically allocated.

12 Optional Modules

12.1 Standalone Audio Mixer

The standalone audio mixer module (eas_audiomix.c) contains a signal processing algorithm for mixing two audio streams with separate gain controls. Here is the function prototype:

```
void EAS_AudioMixer(EAS_PCM *inputBuffer1, EAS_U16 inputGain1, EAS_PCM
   *inputBuffer2, EAS_U16 inputGain2, EAS_PCM *outputBuffer, EAS_I32
   count);
```

The function takes two input buffer pointers and an output buffer pointer, plus two gain control parameters. The input buffers are scaled by the gain parameter and mixed using saturating arithmetic to prevent harsh audio artifacts from overflows.

The gain parameters are 16-bit fixed point with a 15-bit fraction, thus a value of 0x8000 will result in unity gain. To reduce the effects of clipping due to saturation, we suggest using -3dB gain (a value of 0x5a9e) if both input signals are at full scale.

12.2 SRS WOW XT Interface

An optional module that allows EAS audio to be post-processed by the SRS WOW® XT audio library. The WOW XT library is available through a separate license directly from SRS. Note that due to the nature of the WOW XT library, it is a requirement to use the dynamic memory model and stereo audio output, even if the final audio output is monophonic. The WOW library includes a monophonic optimization function that requires a stereo input.

To enable WOW audio processing, define the pre-processor symbol _WOW_ENABLED when eas_config.c is compiled and link the WOW XT library to your project. When EAS_Init is called, the library will automatically initialize the WOW audio processor at the correct sample rate with the default values, and all audio will be processed by the WOW audio processor.

WOW allows for extensive tuning of parameters to meet the specific application. A pointer to the WOW channel control structure is required to set or retrieve parameters. To retrieve the pointer, use the EAS_GetParameter function, as shown:

```
WowXTChannel *wowxtChannel;
if (EAS_SetParameter(pEASData,EAS_MODULE_WOW,EAS_PARAM_WOW_CONTROL,
(EAS_I32*) &wowxtChannel)!= EAS_SUCCESS)
   printf("Error occurred\n");
```

Note the cast to an EAS_I32 pointer to bypass the compiler warning. The wowxtChannel pointer can now be used to access the WOW control functions in the usual fashion.

There are two ways to disable WOW audio processing. The first is to call the EAS_SetParameter function, as detailed in . Here is an example:

```
if (EAS_SetParameter(pEASData,EAS_MODULE_WOW,EAS_PARAM_WOW_BYPASS,
EAS_TRUE) != EAS_SUCCESS)
   printf("Error occurred\n");
```

This call will completely disable all WOW processing, and no audio will be passed to the WOW library for processing. This effectively disables even the post-gain processing done by the WOW library. To re-enable WOW processing, call EAS_SetParameter again with a value of EAS_FALSE.

Processing can also be disabled using the WOW SetWowXTProcessEnable function. First, retrieve the WOW channel control pointer as described above. Then call SetWowXTProcessEnable to disable or enable processing. Other control parameters can be accessed in this same fashion.

12.3 Wave File Parser

The Wave File Parser is an optional module that will parse and render WAVE files containing 8- or 16-bit linear PCM audio or IMA ADPCM audio. It should be treated just like any other song file, by calling the EAS_OpenFile interface to open the file and then EAS_Prepare and EAS_Render to render it.

At this time, the Wave File Parser does not support the EAS_Locate function, so attempts to call EAS_Locate with a Wave File handle will result in an error. A future enhancement will allow for locating with a wave file.

Appendix A – Metadata Sample Code

Rendering Function

```
EAS_RESULT result;
EAS_DATA_HANDLE easData;
EAS_I32 playLength;

/* initialize an instance of the EAS library */
if ((result = EAS_Init(&easData)) != EAS_SUCCESS)
    return result;

/* register metadata callback */
if ((result = EAS_RegisterMetaDataCallback(easData, MetaDataCallback,
    metaDataBuffer, sizeof(metaDataBuffer, NULL)) != EAS_SUCCESS)
    return result;

/* call EAS library to open file */
if ((result = EAS_OpenFile(easData, filename, &handle)) != EAS_SUCCESS)
    return result;

/* parse the metadata */
if ((result = EAS_ParseMetaData(easData, handle, &playLength)) != EAS_SUCCESS)
    return result;

/* normal rendering code here... */

/* close the file */
if ((result = EAS_CloseFile(easData, handle)) != EAS_SUCCESS)
    return result;

/* close this instance of the EAS library */
if ((result = EAS_Shutdown(easData)) != EAS_SUCCESS)
    return result;
```

Metadata Callback Function

```
void MetaDataCallback (EAS_I32 metaDataType, char *buffer, EAS_VOID_PTR pUserData)
{

    switch (metaDatType)
    {
        case EAS_METADATA_TITLE:
            printf("Title: %s\n", buffer);
            break;

        case EAS_METADATA_AUTHOR:
            printf("Author: %s\n", buffer);
            break;

        case EAS_METADATA_COPYRIGHT:
            printf("Copyright: %s\n", buffer);
            break;

        case EAS_METADATA_LYRIC:
            printf("Lyric: %s\n", buffer);
            break;
    }
}
```

Appendix B – Upgrading From Earlier Versions

Beginning with EAS Version 3.5, we added new features and stronger type checking to help prevent programming errors. If you are currently using EAS Version 3.4 or earlier, you will need to make some minor modifications to existing host application code. This appendix explains the modifications necessary to upgrade your host application to the current version.

New Handle Types

Previous versions of EAS used a void pointer as the base for all EAS data types intended to be opaque to the application or the library. EAS now has 3 new underlying data types to prevent programming errors due to passing the wrong data type to a function.

If you compile with strict type checking enabled and previously used void pointers for your handles, you will see data type errors in calls to EAS API functions. To eliminate these errors, it is necessary to change the data types as defined in eas_types.h when you pass them as arguments to EAS. It is recommended that you follow this procedure to avoid errors caused by passing incorrect pointer types.

EAS_DATA_HANDLE: The handle returned by the EAS_Init function. This is a pointer to persistent data used by the EAS library and is used in nearly all EAS API function calls.

EAS_FILE_HANDLE: The handle used the file I/O host wrapper functions. This handle is opaque to the EAS library and is simply passed from the EAS_OpenFile function to the various host wrapper functions in eas_host.c or eas_hostmm.c.

EAS_HW_DATA_HANDLE: This handle is a pointer to the persistent data used by the host wrapper functions. It is opaque to the EAS library and simply passed to the various host wrapper functions in eas_host.c or eas_hostmm.c.

EAS_HANDLE: This handle is a pointer to the persistent data used in streams. As of EAS Version 3.5, this handle is only used to refer to streams that have been opened by EAS_OpenFile or EAS_OpenMIDIStream.

New Functions

Two new functions have been introduced to manage synthesizer polyphony to accommodate multiple streams and “split” architectures where synthesis performed on multiple processors. EAS_SetSynthPolyphony sets the maximum polyphony of the synthesizer for all streams, and EAS_GetSynthPolyphony retrieves the current polyphony setting. For split architectures, these functions also accept a synthesizer number to allow addressing of secondary synthesizer resources that might reside on a separate processor.

Two new functions have been added to manage stream priority when multiple streams are being processed. In Version 3.5 and later, the priority affects the way voices are allocated when multiple streams are being played simultaneously. EAS_SetPriority sets the priority of a stream and EAS_GetPriority retrieves the priority.

Modified Functions

Some functions have been modified to accommodate the control of multiple streams. EAS_SetVolume and EAS_GetVolume now require a stream handle for controlling the volume of individual streams, or NULL to control the master volume.

The polyphony parameter in EAS_Prepare has been dropped and the stream polyphony management function has been moved to EAS_SetPolyphony and EAS_GetPolyphony, which accept a stream handle as a parameter.

EAS_RegisterMetaDataCallback now requires a stream handle so that metadata can be extracted from a single stream. It also takes a pUserData void pointer that can be used by the application for unique instance data for each stream (use NULL if unused). The metadata callback function now also takes a pUserData parameter.

EAS_OpenMIDIStream now requres a stream handle to allow for real-time control of the synthesizer while a file is playing. This functionality was added to support the Java MMAPI MIDIControl interface. Alternatively, passing a NULL to EAS_OpenMIDIStream will create a new instance of the synthesizer.

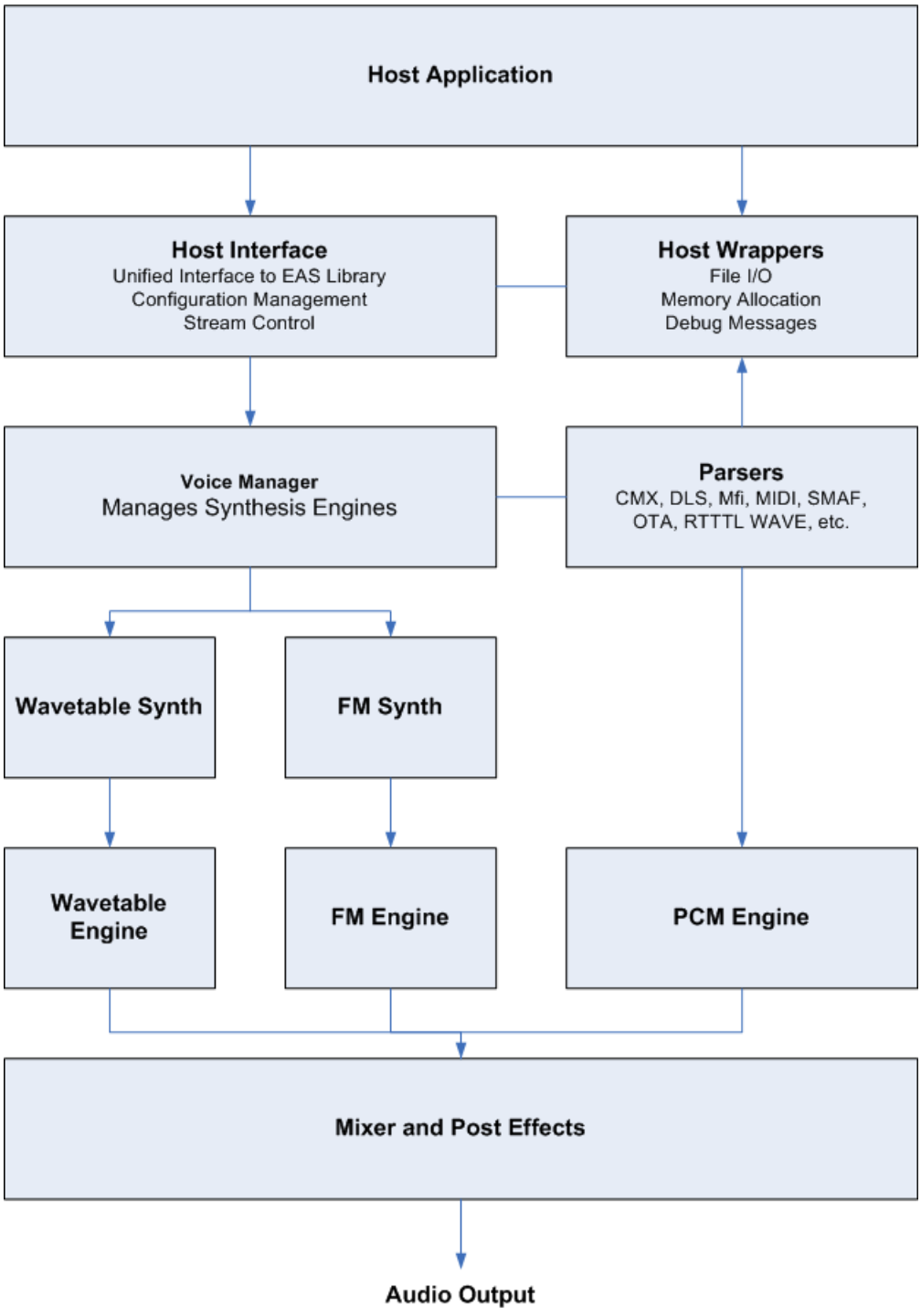


Figure 1: EAS Block Diagram