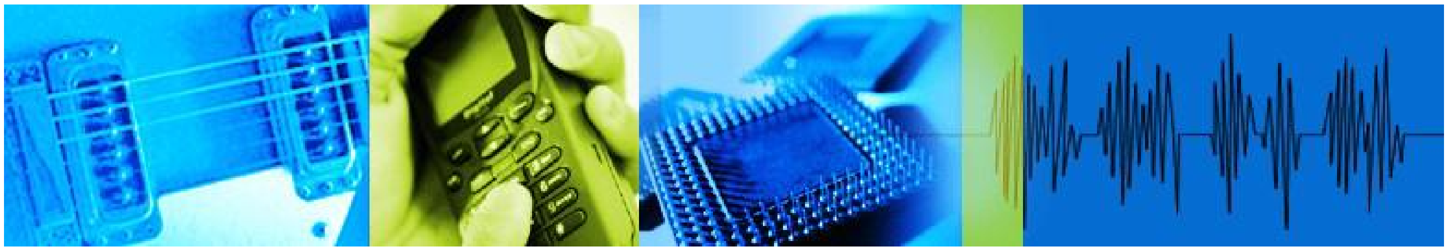
EMBEDDED AUDIO SYNTHESIS



Sonic Network, Inc.
EAS Library

Integration Guide

Copyright 2008 Sonic Network, Inc.

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

Table of Contents

1 Introduction

1.1 Revision History

Rev	Date	Author	Comments
0.1	08/30/2005	dls	Initial Draft

1.2 Abstract
The EAS Library is designed for easy integration into a variety of hardware and software platforms. This guide will assist you through the integration process. For the purposes of this document, when we refer to “application code” or “host code”, we are referring to the code on your platform that calls the EAS library.

1.3 Overview
The EAS library interface consists of two major area: The EAS public applications interface (EAS API); and the EAS Host Wrapper interface. The EAS API is a set of function calls that implement the functionality of the EAS library. These function calls provide the audio rendering functions and means for the host to control the rendering of audio.

The EAS Host Wrapper Interface is a source code module that abstracts host OS platform features to allow the EAS library to be customized to a particular hardware/software platform. We supply sample source code for the host wrapper that you can modify to suit your platform. In some cases, you may be able to use this code as-is, in other cases, it may be necessary to modify the code to adapt it to your platform.

The diagram below illustrates the flow of data between the application code, EAS Library, and Host Wrapper interface. Further details on the types of data that is passed between the modules can be found in the EAS API and Host Wrapper Interface sections.

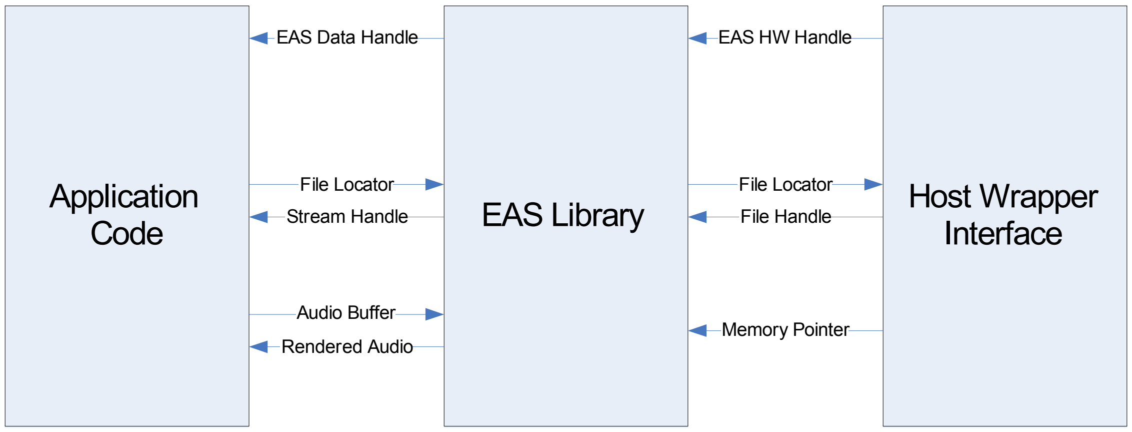


Figure 1: EAS Architecture

1.4 Integration Process
We suggest you follow these steps in the integration process:

1. Read through this guide and study the supplied source code.
2. Familiarize yourself with the EAS Library documentation.
3. Build the sample host code and run it in simulation to make certain there are no compatibility issues between the EAS code and your development environment.
4. Modify the host wrapper functions as necessary for your platform and test in simulation.
5. Integrate the initialization and audio rendering code into your application and begin integration testing.

2 Configuration and Compilation

2.1 Configuration Module

The Configuration Module is a source code module that provides configuration information to the EAS Library. The Configuration Module options are determined by the setting of compile-time preprocessor symbols. The primary configuration options are the memory model and the inclusions of optional modules such as file parsers and effects processing algorithms. Information on configuring optional modules can be found in the EAS Library documentation.

2.2 Memory Model

The EAS Library supports both static and dynamic memory allocation models. By default, the library uses dynamic memory allocation through heap allocation functions that are abstracted in the host wrapper interface. Certain modules require the use of dynamic memory to function, see the EAS Library documentation for more detailed information.

If your platform requires static memory, define the preprocessor symbol "_STATIC_MEMORY" when you compile the Configuration Module (eas_config.c) and the Host Wrapper Interface module (eas_host.c or eas_hostmm.c). The EAS Library contains static memory modules for all code modules that support the static memory model. The Configuration Model contains an interface for the EAS Library to obtain links to the static memory modules.

2.3 Compilation

After you have selected the appropriate memory model and host wrapper interface module, create a project, makefile, or build script to compile the sample host modules and link them with the EAS library. Make sure to add the appropriate preprocessor symbols to the build for the options that you plan to use. You can verify the options by examining the config.txt included with the library.

3 EAS API

The EAS API provides the means for the host to render audio. This guide describes the basic operation of the EAS API. Please refer to the EAS Library Documentation for detailed information on specific API calls. Appendix A contains sample pseudo-code that shows how the application code might access the basic functions of the EAS API.

3.1 Handles

The EAS Library uses opaque pointers called "handles" to facilitate communication between the host and the EAS library. In some cases, the handle is opaque to the host; in other cases the handle is opaque to the library. The following are brief descriptions the most commonly used handles:

EAS_DATA_HANDLE: This is a pointer to the persistent storage for the instance of the EAS Library. The host receives a copy of this pointer as part of data returned by EAS_Init() during initialization. This handle is required for nearly all the EAS API function calls. In dynamic memory models, it is possible to create multiple instances of the EAS Library by calling EAS_Init() for each instance.

EAS_HW_DATA_HANDLE: This handle is provided for use by the host in the host wrapper interface. It is opaque to the EAS Library and is returned to the EAS Library through the

EAS_HWInit() function. The EAS Library calls the host wrapper functions with this handle. The sample implementation of the host wrapper interface uses this handle to point to instance data it uses to track file handles and buffer pointers used by the file I/O functions.

The EAS_HW_DATA_HANDLE is provided primarily as a means to support multiple instances of the EAS Library so that each can have unique host wrapper data. Because the data type is opaque to the EAS Library, the host wrapper interface can use it to store any persistent data that it requires for the life of an instance of the library. If there are multiple instances of the EAS Library, the host wrapper interface may provide a unique handle to each instance. The host wrapper might use this information to clean up process data in the case of a killed or crashed process.

EAS_HANDLE: This is a generic opaque pointer returned or used by a number of EAS API function calls. The most common use is a pointer to the persistent data associated with an audio file, such as a MIDI file. When the host calls EAS_OpenFile(), it receives an EAS_HANDLE stream pointer in the return data. This pointer can be used to pause, resume, and close the stream, as well as other stream-related functions.

EAS_HANDLE is also used as a generic file pointer by the EAS Library. When the host requests that the EAS Library open an audio file for rendering, the library calls the host wrapper interface to open the file. The EAS_HWOpenFile() function returns an EAS_HANDLE that the EAS Library can use to read data, seek to a location, or close the file. In this case, the EAS_HANDLE is opaque to the EAS Library.

3.2 Initialization

The EAS Library must be initialized prior to use. The EAS_Init() function performs the initialization function and must be called prior to calling any other API functions. In turn, EAS_Init() calls EAS_HWInit to initialize the host wrapper interface (see Host Wrapper Functions for more details). In dynamic memory models, some blocks of dynamic memory will be allocated during the call to EAS_Init() for persistent data.

The EAS Library assumes that read/write date is persistent from the time that EAS_Init() called until EAS_Shutdown() is called. If read/write data is lost due to power conservation or power loss, the host must call EAS_Init() to re-initialize the persistent data.

3.3 Shutdown

EAS_Shutdown() is an optional API function call that de-allocates any dynamic memory that has been allocated. It is not necessary to call EAS_Shutdown() after each file is played. It is not necessary to call EAS_Shutdown() at all if you are using the static memory model.

3.4 Audio Rendering

After initialization, the EAS library is ready to begin rendering. The host does not have to open an input audio file before starting the rendering process. If no file is currently playing, the rendering function will return a buffer of silence.

To render audio, the host calls EAS_Render() with the EAS_DATA_HANDLE returned by EAS_Init() and a pointer to the buffer where the audio is to be rendered. The size of the buffer depends on the audio sample rate. The host may call EAS_Config() to determine the required size of the buffer.

The process of rendering an audio file begins with a call to EAS_OpenFile(), which returns an EAS_HANDLE stream pointer. If successful, the host can then call EAS_Prepare() to prepare the audio file for playback. After EAS_Prepare(), calls to EAS_Render() will result in audio from the file being rendered into the supplied audio buffer. The host then passes the rendered audio to the DAC or other audio output device.

The host can check on the state of an audio stream by calling EAS_State(). The most common states are EAS_STATE_PLAY, EAS_STATE_STOPPING, and EAS_STATE_STOPPED. Normally, when the stream state is EAS_STATE_STOPPED, the host will call the EAS_CloseFile() function to close the file and free the resources associated with it.

4 Host Wrapper Interface

The Host Wrapper Interface provides abstractions of file I/O and memory allocation, move, and compare operations. These functions insulate the EAS Library from any hardware or OS specific functions and allow you to optimize these functions for your own platform. We provide two versions of sample source code for the host wrapper interface.

Note that both versions of the source code include sample code for duplicating file handles. Duplicate file handles provide double-buffering for file formats such as Standard MIDI File Type 1 that have multiple streams that must be processed in parallel. Without double-buffer, there is a tendency for thrashing, as the file parser must read multiple streams within the file that are often dispersed over the entire file.

If your OS supports duplicate file handles natively (such as the dup() function in Unix environments), you can eliminate the double-buffering and use the native handle duplication function. This will likely result in increased efficiency in the file I/O operations.

NOTE: If you are using the static memory model, you should modify the EAS_HWMalloc() function, to remove the call to malloc() and simply return NULL. You should also modify the EAS_HWFree() function and remove the call to free().

4.1 File I/O Version (eas_host.c)

The file eas_host.c contains sample source code for a host wrapper interface that uses stdio functions to read files. In this version, all file I/O is simply passed through to the corresponding stdio function. Memory allocation functions are passed to malloc() and free(), and memory set, move, and compare functions are passed to memset, memcpy, and memcmp, respectively.

4.2 Memory Mapped Version (eas_hostmm.c)

The file eas_hostmm.c contains sample source code for a host wrapper interface that uses stdio functions to read files. However, in this version, a memory buffer is allocated when a file is first opened, and the entire file is read into memory. If a duplicate handle is needed, the host wrapper simply creates a new read pointer into the same block of memory, keeping an instance count. When the final instance is closed, the memory containing the file image is freed.

Memory allocation functions are passed to malloc() and free(), and memory set, move, and compare functions are passed to memset, memcpy, and memcmp, respectively. Note that this version is not compatible with the static memory model, because it requires dynamic memory for the memory buffers.

4.3 Read-Only Memory

If your audio file images are stored at absolute addresses in read-only memory that can be directly accessed by the processor (e.g. parallel flash memory), you can modify the eas_hostmm.c source code to access the file images directly. This requires some coordination between the code that calls the EAS_OpenFile() function and the EAS_OpenFile() host wrapper function.

The locator pointer that is passed to EAS_OpenFile() is an opaque pointer that the EAS Library passes directly to the EAS_OpenFile() function. In both sample versions of the host wrapper functions, the pointer is assumed to point to a zero-terminated string that contains the name of the file. Alternatively, you can pass a pointer to any kind of data structure that is useful to the host wrapper to locate the file. The only requirement is that the host wrapper be able to locate the file image and size.

For example, you could pass a pointer to the file image, and store the size of the image at the pointer location – 4. The duplicate handle processing is similar, except that you can eliminate the memory allocation, deallocation and instance counting code for the file buffer.

4.4 Using DMA for Audio Input Files

It is not practical to use DMA for moving the data in an audio input file to the EAS Library rendering engine. While linear audio streams like PCM or MP3 have predictable data rates, standard MIDI files and similar file formats are unpredictable and often require random access.

However, if the overhead for your DMA engine is not too high, you may want to use it to move data from the audio files into system memory buffers that are, in turn, consumed by the EAS Library. Make sure that the EAS_FILE_BUFFER_SIZE define is large enough to warrant the additional overhead. You should also be aware that the total memory used for file buffering is EAS_MAX_FILE_HANDLES * EAS_FILE_BUFFER_SIZE.

Appendix A

Sample Host Code

This sample code shows how you might integrate the EAS Library into an application. To make the logic easy to follow, we have omitted the typical error recovery and code hardening that you would normally find in an embedded system.

This code assumes a hardware DMA engine to move audio from system memory to the DAC. The same mechanism works well for programmed I/O if the DAC has a small FIFO buffer. If the DAC FIFO is deep enough to allow the EAS Library to render a buffer in less time than it takes the FIFO to empty, you can use a single buffer. In this case, render a buffer of audio and fill the FIFO from there. When the buffer is empty, render the next frame of audio into the buffer. When the FIFO reaches the low-water mark, fill it again from the audio buffer.

During initialization, you will call InitEASLibrary() to initialize the EAS library and establish the buffers and state information for the rendering code. When you want to play a file, call StartPlayback() with the filename to be played. The main task loop should call RenderAudio() often to ensure that the next buffer of audio will be rendered before the DMA is complete on the previous buffer.

To stop playback while the file is still playing, simply call StopPlayback() which will cause the audio stream to pause and continue to call RenderAudio(). The rendering code gracefully shuts down the audio stream to prevent any clicks or pops in the audio output. When the stream reaches the paused state, the input file will be closed and stream will be set to NULL. You can stop calling AudioRender() at this time.

If you have a multi-tasking OS, you can eliminate the polling loop and create an audio task that waits for an event to be triggered by the DMA interrupt before rendering a new buffer. Bear in mind, that the EAS Library is not thread-safe, so you will need to serialize access to the it. We recommend using messages to control audio playback in the audio task, as this localizes the state data, reduces the likelihood of deadlocks, and eliminates unnecessary context switches, but you can also use a semaphore to serialize access from multiple tasks.

```
/* include the EAS public API header file */
#include “eas.h”

/* Error codes for detected error conditions. The EAS Library uses
 * negative values, so we will use positive values. Success is
 * indicated by zero (EAS_SUCCESS).
 */
#define ERROR_BUFFER_ALLOCATION     1
#define ERROR_BUFFER_UNDERRUN       2

/* Persistent variables that are needed for EAS API calls. You can
 * either create them as static data, as we have done here, or create
 * them as stack variables at the top level of the task stack.
 */
static EAS_DATA_HANDLE pEASData;
static EAS_HANDLE stream;
static EAS_PCM *buffer[2];
static int renderBuffer;
static int dmaBuffer;
static int sampleCount;
static int polyphony;

/*********************************************************************
 * InitEASLibrary()
 *
 * This function initializes the EAS library and allocates a pair of
 * “ping-ping” buffers for audio rendering.
 ***************************************************************/
EAS_RESULT InitEASLibrary ()
{
      EAS_RESULT result;
      const S_EAS_LIB_CONFIG *pLibConfig;
      int bufferSize;

      /* get the library configuration */
      pLibConfig = EAS_Config();
      polyphony = pLibConfig-> maxVoices;
      sampleCount = pLibConfig->mixBufferSize;

      /* initialize the library */
      if ((result = EAS_Init(&pEASData)) != EAS_SUCCESS)
            return result;

      /* calculate the size of the audio buffers */
      bufferSize = pLibConfig->mixBufferSize *
            pLibConfig->numChannels *
            (EAS_I32)sizeof(EAS_PCM);

      /* allocate the audio buffers */
      buffer[0] = malloc(bufferSize);
      buffer[1] = malloc(bufferSize);
      if (!buffer[0] || !buffer[1])
            return ERROR_BUFFER_ALLOCATION;

      stream = NULL;
      return EAS_SUCCESS;
}

/****************************************************************
 * StartPlayback()
 *
 * This function starts playback of an audio file
 ***************************************************************/
EAS_RESULT StartPlayback (const char *filename)
{
      EAS_RESULT result;
      EAS_I32 count;

      /* open the file */
      if ((result = EAS_OpenFile(pEASData, filename, &stream)) !=
            EAS_SUCCESS)
            return result;

      /* prepare for playback */
      if ((result = EAS_Prepare(pEASData, streamstream, polyphony)) !=
            EAS_SUCCESS)
            return result;

      /* render the first buffer of audio */
      if ((result = EAS_Render(easData, buffer[0], sampleCount,
            &count)) != EAS_SUCCESS)
            return result;

      /* this sample implementation assumes that a hardware DMA is used
       * to move the audio data to the DAC. If you use a FIFO or other
       * mechanism, alter the code accordingly.
       */
      renderBuffer = 1;
      dmaBuffer = 0;
      StartDMA(buffer[0], sampleCount);

      return EAS_SUCCESS;
}

/*********************************************************************
 * RenderAudio ()
 *
 * This function is called from the main task loop after the
 * initialization code has executed. It will render audio only when an
 * audio input file has been opened. It uses polling to determine if
 * another buffer needs to be rendered.
 *********************************************************************/
EAS_RESULT RenderAudio ()
{
      EAS_RESULT result;
      EAS_I32 count;
      EAS_STATE state;

      /* only render audio if an audio stream is open */
      if (stream == NULL)
            return EAS_SUCCESS;

      /* if the buffers indices are equal, both buffers are full, or
       * we have an underrun condition
       */
      if (renderBuffer == dmaBuffer)
      {
            if ((result = EAS_CloseFile(easData, stream)) !=
                  EAS_SUCCESS)
                  return result;
            return ERROR_BUFFER_UNDERRUN;
      }

      /* we have an empty buffer, render into it */
      if ((result = EAS_Render(easData, buffer[renderBuffer],
            sampleCount, &count)) != EAS_SUCCESS)
            return result;

      /* advance the render buffer pointer */
      renderBuffer = 1 - renderBuffer;

      /* check the stream state */
      if ((result = EAS_State(easData, stream, &state)) != EAS_SUCCESS)
            return result;

      /* if the stream has stopped, close it */
      if ((state == EAS_STATE_STOPPED) || (state == EAS_STATE_PAUSED))
      {
            if ((result = EAS_CloseFile(easData, stream)) !=
                  EAS_SUCCESS)
                  return result;
      }
}

/**************************************************************
 * DMAComplete()
 *
 * This function is called from the DMA interrupt when the DMA of an
 * audio buffer is complete
 **************************************************************/
EAS_void DMAComplete ()
{

      /* advance the DMA pointer */
      dmaBuffer = 1 - dmaBuffer;

      /* if the buffers are equal, we have run out of audio */
      if (dmaBuffer == renderBuffer)
            return;

      StartDMA(buffer[0], sampleCount);
}

/**************************************************************
 * StopPlayback()
 *
 * This function stops audio playback of an audio file
 **************************************************************/
EAS_RESULT StopPlayback (const char *filename)
{
      EAS_RESULT result;

      return EAS_Pause(pEASData, stream);
}
```