# Getting Started

Adding music and sound effects to an Alpha Engine project is actually a fairly simple process, once you know how to do it. The purpose of this document is to help new students incorporate the FMOD Studio API into their projects.

# Download API

The FMOD sound library may be downloaded from <http://www.fmod.org/>. A recent change to the website now requires users to log in before downloading files.

* Go to <http://www.fmod.org/>.
* Click on the “Download” link.
* Click on the “FMOD Studio API” link or scroll down to the “FMOD Studio Programmer’s API and Low Level Programmer API” section.
* Click on the “Log In / Register” button next to “Windows”.
  + First time users will need to register an account first.
* Again, click on the “FMOD Studio API” link or scroll down to the “FMOD Studio Programmer’s API and Low Level Programmer API” section.
* Click on the “Download” button next to “Windows”.
* Download and install the FMOD Studio API.
  + Typically, the installed files may be found in the following location:
    - “C:\Program Files (x86)\FMOD SoundSystem\FMOD Studio API Windows”

# Copy FMOD Files

It isn’t necessary for every member of your team to install the FMOD Studio API. Instead, you should add a select set of FMOD files to your project and commit these files to your Version Control System (VCS). You will need to include the following sets of files:

* Include files
  + Relative path: “.\api\lowlevel\inc”
  + Required files: \*.h
* Library files
  + Relative path: “.\api\lowlevel\lib”
  + Required files: fmod\_vc.lib, fmodL\_vc.lib
* .DLL files
  + Relative path: “.\api\lowlevel\lib”
  + Required files: fmod.dll, fmodL.dll
* Help files (optional but highly recommended!)
  + Relative path: “.\doc”
  + Required files: FMOD Studio Programmers API for Windows.chm, LICENSE.TXT

Create a new folder (recommend name is “FMOD”) either next to or within your project folder. To this folder copy the Include, Library and .DLL files. Optionally, these files may be copied to separate “Include” and “Library” folders, at your discretion.

Eventually, the .dll files will need to exist in the same folder as the .exe compiled for your project, specifically the “Debug” and/or “Release” folders. A number of methods exist for doing so, including:

* Add a post-build step that “xcopy”’s the required .dll from the FMOD folder
* Commit copies of the .dll’s within the “Debug” and “Release” folders to your VCS.

# Visual Studio Integration

Visual Studio needs to be told how to find the FMOD files and, in the case of the library (.lib) file, which library file to include. When adding the library files, both .lib and .dll, it is important to note that there are two possible options:

* fmod\_vc.lib, fmod.dll
* fmodL\_vc.lib, fmodL.dll

The “L” option enables output to a log file, for the purposes of debugging problems. Using this option can cause noticeable delays during file loading and is best avoided except when initially implementing FMOD support or when trying to track down audio issues.

The first step is to instruct Visual Studio on where to find the FMOD files.

1. In the Solution Explorer right-click on your project (the second entry in the view tree).
2. Select "Properties".
3. On this page you will see a "Configuration" drop-down.  You will need to repeat steps 4 - 10 for both the "Debug" and "Release" configurations. Alternatively, you may select “All Configurations” to set both configurations simultaneously.
4. Select "Configuration Properties/VC++ Directories".
5. To the right of "Include Directories" click within the text field and open the drop-down which appears.
6. Select "<Edit...>".
7. Add the path to the included files (e.g. “.\FMOD” or “.\FMOD\include”).
8. To the right of "Library Directories" click within the text field and open the drop-down which appears.
9. Select "<Edit...>".
10. Add the path to the included files (e.g. “.\FMOD” or “.\FMOD\lib”).

The next step is to instruct Visual Studio to include the FMOD library during the link step.

1. In the Solution Explorer right-click on your project (the second entry in the view tree).
2. Select "Properties".
3. On this page you will see a "Configuration" drop-down.  You will need to repeat steps 4 - 7 for both the "Debug" and "Release" configurations. Alternatively, you may select “All Configurations” to set both configurations simultaneously.
4. Select "Configuration Properties/Linker/Input".
5. To the right of "Additional Dependencies" click within the text field and open the drop-down which appears.
6. Select "<Edit...>"
7. Add the name of the .lib to the list.

# Basic Implementation

Example projects that demonstrate how to use the FMOD Studio API may be found in the FMOD install folder in the following location:

* “api\lowlevel\examples”

For most purposes, it is recommended that you begin with one of the following two examples:

* play\_sound.cpp
* play\_stream.cpp

Please note that, while these are C++ files, the sample code can very easily be converted into C. Following are some examples of the conversions that are required:

## Data Structures

### (C++)

FMOD::System \*system;

FMOD::Sound \*sound1, \*sound2, \*sound3;

FMOD::Channel \*channel = 0;

### (C)

static FMOD\_SYSTEM \*FmodSystem;

static FMOD\_SOUND \*sound[cSoundMax];

static FMOD\_CHANNEL \*channel = 0;

The static keyword has been added to demonstrate how these variables may be kept private, as per Chapter 11 (“Hide Information”) of the *C++ Coding Standards* book.

## Initialization

The FMOD initialization code needs to be called only once, while the game engine first boots up. Please make sure to follow the initialization steps as outlined in the sample code.

### (C++)

result = FMOD::System\_Create(&system);

ERRCHECK(result);

result = system->getVersion(&version);

ERRCHECK(result);

result = system->init(32, FMOD\_INIT\_NORMAL, 0);

ERRCHECK(result);

result = system->createSound(Common\_MediaPath("drumloop.wav"), FMOD\_DEFAULT, 0, &sound1);

ERRCHECK(result);

### (C)

result = FMOD\_System\_Create(&FmodSystem);

ERRCHECK(result);

result = FMOD\_System\_GetVersion(FmodSystem, &version);

ERRCHECK(result);

result = FMOD\_System\_Init(FmodSystem, 32, FMOD\_INIT\_NORMAL, NULL);

ERRCHECK(result);

result = FMOD\_System\_CreateSound(FmodSystem, "./assets/audio/drumloop.wav", FMOD\_HARDWARE, 0, &sound[0]);

ERRCHECK(result);

Please note the pattern where the C function names begin with “FMOD\_” and incorporate the type of object that must be passed as the first parameter. Some examples include:

* “FMOD\_System”
* “FMOD\_Sound”
* “FMOD\_Channel”

## Update

The FMOD Studio API automatically handles multithreading so you typically do not need to worry about that. However, it is important to note that FMOD requires periodic updates on the main thread. If you do not call FMOD’s update function every game loop, then sound channels may never be freed up and all audio may eventually stop.

### (C++)

result = system->update();

ERRCHECK(result);

### (C)

FMOD\_System\_Update(FmodSystem);

ERRCHECK(result);

## Shutdown

As your game exits, it is a good idea to release all allocated resources.

### (C++)

result = sound1->release();

ERRCHECK(result);

result = system->close();

ERRCHECK(result);

result = system->release();

ERRCHECK(result);

### (C)

result = FMOD\_Sound\_Release(sound[soundId]);

ERRCHECK(result);

result = FMOD\_System\_Close(FmodSystem);

ERRCHECK(result);

result = FMOD\_System\_Release(FmodSystem);

ERRCHECK(result);

# Additional Concepts

## Music vs. Sound Effects

Using the low-level API it is possible to implement music and sounds effects using the same functions. However, there are several factors to consider when differentiating between these two categories of audio assets. The examples below are “typical” for GAM150/152 projects but should not be considered true in all possible cases, most notably professional AAA titles.

* Typically, music tracks are longer and require more memory than sound effects.
* Typically, only one or two music tracks will be playing at any given time, while many sound effects may be playing simultaneous.
* Typically, music tracks loop while most sound effects are “one shot”.

This leads to the decision of whether to use CreateSound() or CreateStream() before playing an audio resource.

* CreateSound() loads the specified audio resource into memory. This is ideal for sound effects, given that they are typically smaller and more numerous than music tracks.
* CreateStream() opens the specified audio resource and prepares it for streaming off of a mass storage device (i.e. hard drive). This is ideal for music tracks which, due to their larger size should not be loaded into memory. Additionally, only one or two are playing at any given time, so contention for access to the mass storage device should pose no problems.

Within an Alpha Engine project, one approach would be to create a single, reusable Sound object for playing streaming music tracks and an array of Sound objects for preloading sound effects. One would call CreateStream() using the reusable Sound object each time a new music track needed to be played. The array of Sound objects could be initialized just once, at the beginning of the game, using CreateSound(). The later assumes that your game has just one fixed set of sound effects that remain unchanged during gameplay. A more complex solution will be required if sound effects need to be loaded dynamically during gameplay.

## Looping

When looping music and sound effects it is usually necessary to maintain a pointer to the Channel object being used to play the audio asset. Having access to this Channel object allows you to stop, pause or otherwise change the properties of the looping asset.

Typically, there is a 1-to-1 relationship between Sound and Channel objects when working with simple music tracks. However, this is often not the case when using sound effects. As a result, looping sound effects may require a bit more work than “one shot” sound effects. Note that the Channel objects used for “one shot” sound effects can be considered as “fire and forget”. The Channel object allocated for the sound effect will be reclaimed by FMOD’s update function once the sound effect is no longer playing.

## Creating A Sound Module

The sound module should have some sort of data structure (for example a linked list or an array) to hold all the active sounds that are in your game. In addition to that, the FMOD structure, “FMOD\_SYSTEM” should also be saved as a static variable in the module because this would only be initialized once at the beginning of the game.

While creating a sound module it’s usually helpful to have a struct for each ‘sound’ that would hold both FMOD structs and your own data about the sound. The following shows an example of what structure can be used:

typedef struct Sound

{

const char \*soundPath; // file path

const char \*name; // sound name to be used a future reference

FMOD\_SOUND \*fmodSound; // fmod struct managing fmod stuff

FMOD\_CHANNEL \*channel; // another fmod struct managing fmod stuff

SoundType soundType; // whether a song is effect or a background sound

}Sound;

This would essentially be the struct that you would maintain in a data structure to access and play particular sounds in your engine.

## Other Tips:

* Make sure you release each sound only once (if your game crashes on exit because of FMOD this is the most likely problem).
* Do not call FMOD\_System\_CreateSound or FMOD\_System\_CreateStream twice on a sound (you’ll end up playing the sound the number of times you call that function)
* For background sounds, create the sounds in the gamestate load functions and release them in the gamestate unload function.
* For gameobjects that have sound effects, create the sounds in the gameobject’s initialize and release them in the gameobject’s shutdown.
* On exiting your game, make sure ALL the sounds you created are released using the “FMOD\_Sound\_Release” function and after that use the functions “FMOD\_System\_Close” and “FMOD\_System\_Release” to shutdown the FMOD system.
* Use FMOD\_RESULT as a way to check for possible errors in the FMOD system. Every FMOD function returns FMOD\_OK if it was successful in doing its task. If an error has occurred, use FMOD\_ErrorString to convert FMOD\_RESULT to a char \* to print it out. It can be used like this:

printf("(FMOD): %s", FMOD\_ErrorString(ErrorResult)); // ErrorResult is an FMOD\_RESULT