FMOD Studio Plug-in SDK

Plug-in Development Guide

Introduction

Game studios and third-party developers can augment FMOD Studio's built-in suite of effect and sound modules by creating their own plug-ins. By placing plug-ins in FMOD Studio's plug-ins folder, these can be added to tracks or buses, modulated and automated by game parameters just like built-in effect and sound modules.

This document describes how to create plug-ins and make them available to FMOD Studio and the game. An example Visual Studio/Xcode plug-in project, *fmod_gain*, can be found in the *Lowlevel* examples installed with the FMOD Studio API.

Accessing Plug-ins in FMOD Studio

A plug-in must be built as a 32-bit dynamic linked library and placed in the plug-ins folder specified in FMOD Studio's *Preferences* dialog under the *Plug-ins* tab. FMOD Studio scans the folder and all sub-folders both on start-up and when the folder is changed by the user. Studio tries to load any libraries it finds (*.dll on Windows or *.dylib on Mac) and ignores libraries which don't support the API.

Detected plug-in sounds will be available via the track context menu in the *Event Editor*, whereas detected plug-in effects will show up in the effect deck's *Add Effect* and *Insert Effect* context menus. When a plug-in module is added to a track or bus, its panel will be displayed in the effect deck. The panel will be automatically populated with dials, buttons and data drop-zones for each parameter.

Contents

Basics

Two versions of the plug-in will usually be required - one for FMOD Studio and one for the game.

Studio will require a 32-bit dll or dylib file if running in Windows or Mac respectively. These will be loaded dynamically in Studio as described in the previous section.

Another version of the plug-in must be compiled for the game's target platform. This may also be a dynamic library but, in most cases, can (or must) be a static library or simply compiled along with the game code. In each case, game code is required to load the plug-in prior to loading the project or object referencing the plug-in.

Building a Plug-in

The `fmod_dsp.h` header file includes all the necessary type definitions and constants for creating plug-ins including the struct `FMOD_DSP_DESCRIPTION` which defines the plug-in's capabilities and callbacks.

If creating a dynamic library, the library must export `FMODGetDSPDescription`, e.g.:

```
extern "C" {
    F_DECLSPEC F_DLLEXPORT FMOD_DSP_DESCRIPTION* F_STDCALL FMODGetDSPDescription();
}
```

Dynamic libraries must be compiled for the same architecture as the host (whether FMOD Studio or the game), so if the game is 64-bit, the game version of the plug-in must be 64-bit otherwise the plug-in should be 32-bit.

A free tool such as Dependency Walker can be used to verify that the library is able to be loaded and the proper symbol is exported. In Windows, the symbol will look like `_FMODGetDSPDescription@0`.

Loading the Plug-in in the Game

The plug-in must be registered using the FMOD Studio or low-level API before the object referencing the plug-in is loaded in the game.

The following functions can be used to register a plug-in if it is statically linked or compiled with the game code:

```
FMOD_RESULT FMOD::Studio::System::registerPlugin(const FMOD_DSP_DESCRIPTION* description);
FMOD_RESULT FMOD::System::registerDSP(const FMOD_DSP_DESCRIPTION *description, unsigned int
*handle);
```

If the plug-in library is to be dynamically loaded, a plug-in path can be specified prior to initialising the system using the function:

```
FMOD_RESULT FMOD::System::setPluginPath(const char *path)
```

Any plug-ins in this folder will be automatically registered during initialization. Alternatively, a particular plug-in library can be registered using:

```
FMOD_RESULT FMOD::System::loadPlugin(const char *filename, unsigned int *handle, unsigned
int priority = 0)
```

Plug-ins do not normally need to be unregistered, but it is possible with either of the following functions:

```
FMOD_RESULT FMOD::Studio::System::unregisterPlugin(const char* name)
FMOD_RESULT FMOD::System::unloadPlugin(unsigned int handle)
```

In these functions, `name` refers to the name of the plug-in defined in the plug-ins descriptor and `handle` refers to handle returned by `FMOD::System::loadPlugin()`.

Plug-in Types

There are two main plug-in types:

- *Effect Modules*
- *Sound Modules*

Both module types are created in the same way - the difference lies in whether the plug-in processes an audio input.

Effect Modules apply effects to an audio signal, they have an input and an output. *Effect Modules* can be inserted anywhere in FMOD Studio's signal routing, whether it be on an *Event*'s track or a mixer bus. Examples of different types of plug-in effects include:

- Effects which have the same input and output channel counts such as EQ, compression, distortion etc...
- Effects which perform up- or down-mixing as part of the processing algorithm such as panning or reverb
- Spatialization and any distance/direction effects which respond to a sound's 3D location in the game such as 3D panning, distance filtering, early reflections or binaural audio
- Side-chaining effects such as compression or audio modulation (e.g. ring modulators)

Sound Modules produce their own sound - they do not have an audio input. *Sound modules* can be placed on tracks inside *Events* and can be made to trigger from the timeline, game parameter or within another sound module.

The Plug-in Descriptor

The plug-in descriptor is a struct, `FMOD_DSP_DESCRIPTION` defined in `fmod_dsp.h`, which describes the capabilities of the plug-in and contains function pointers for all callbacks needed to communicate with FMOD. Data in the descriptor cannot change once the plug-in is loaded. The original struct and its data must stay around until the plug-in is unloaded as data inside this struct is referenced directly within FMOD throughout the lifetime of the plug-in.

The first member, `pluginsdkversion`, must always hold the version number of the plug-in SDK it was complied with. This version is defined as `FMOD_PLUGIN_SDK_VERSION`. The SDK version is incremented whenever changes to the API occur.

The following two members, `name` and `version`, identify the plug-in. Each plug-in must have a unique name, usually the company name followed by the product name. Version numbers should not be included in the name in order to allow for future migration of saved data across different versions. Names should not change across versions for the same reason. The version number should be incremented whenever any changes to the plug-in have been made.

Here is a code snippet from the *FMOD Gain* example which shows how to initialize the first five members of `FMOD_DSP_DESCRIPTION`:

```
FMOD_DSP_DESCRIPTION FMOD_Gain_Desc =
{
    FMOD_PLUGIN_SDK_VERSION,
    "FMOD Gain",    // name
    0x00010000,     // plug-in version
    1,              // number of input buffers to process
    1,              // number of output buffers to process
    ...
};
```

The other descriptor members will be discussed in the following sections.

Thread Safety

Audio callbacks `read, process and shouldiprocess` are executed in FMOD's mixer thread whereas all other callbacks are executed in the host's thread (game or Studio UI). It is therefore important to ensure thread safety across parameters and states which are shared between those two types of callbacks.

In the *FMOD Gain* example, two gains are stored: *target gain* and *current gain*. *target gain* stores the parameter value which is set and queried from the host thread. This value is then assigned to *current gain* at the start of the audio processing callback and it is *current gain* that is then applied to the signal. *FMOD Gain* shows how this method can be used to perform parameter ramping by not directly assigning *current gain* but interpolating between *current gain* and *target gain* over a fixed number of samples so as to minimize audio artefacts during parameter changes.

Plug-in Parameters

Plug-in effect and sound modules can have any number of parameters. Once defined, the number of parameters and each of their properties cannot change. Parameters can be one of four types:

- floating-point
- integer
- boolean (two-state)
- data

Parameters are defined in `FMOD_DSP_DESCRIPTION` as a list of pointers to parameter descriptors, `paramdesc`. The `numparameters` specifies the number of parameters. Each parameter descriptor is of type `FMOD_DSP_PARAMETER_DESC`. As with the plug-in descriptor, parameter descriptors must stay around until the plug-in is unloaded as the data within these descriptors are directly accessed throughout the lifetime of the plug-in.

Common to each parameter type are the members `name` and `units`, as well as `description` which should describe the parameter in a sentence or two. The `type` member will need to be set to one of the four types and either of the floatdesc, intdesc, booldesc or datadesc members will need to specified. The different parameter types and their properties are described in more detail the sections below.

Floating-point Parameters

Floating-point parameters have `type` set to `FMOD_DSP_PARAMETER_TYPE_FLOAT`. They are continuous, singled-valued parameters and their minimum, maximum and default values are defined by the `floatdesc` members `min`, `max` and `defaultval`.

The following units should be used where appropriate:

- `"Hz"` for frequency or cut-off
- `"ms"` for duration, time offset or delay
- `"st"` (semitones) for pitch
- `"dB"` for gain, threshold or feedback
- `"%"` for mix, depth, feedback, quality, probability, multiplier or generic 'amount'.
- `"Deg"` for angle or angular spread

These are preferred over other denominations (such as *kHz* for cut-off) as they are recognised by Studio therefore allowing values to be displayed in a more readable and consistent manner. Unitless 0-to-1 parameters should be avoided in favour of *dB* if the parameter describes a gain, *%* if it describes a multiplier, or a unitless 0-to-10 range is preferred if describing a generic amount.

The FMOD_DSP_DESCRIPTION members setparameterfloat and getparameterfloat will need to point to static functions of type FMOD_DSP_SETPARAM_FLOAT_CALLBACK and FMOD_DSP_GETPARAM_FLOAT_CALLBACK, respectively, if any floating-point parameters are declared.

These will be displayed as dials in FMOD Studio's effect deck.

Integer Parameters

Integer parameters have type set to FMOD_DSP_PARAMETER_TYPE_INT. They are discrete, singled-valued parameters and their minimum, maximum and default values are defined by the intdesc members min, max and defaultval. The member goestoinf describes whether the maximum value represents infinity as maybe used for parameters representing polyphony, count or ratio.

The FMOD_DSP_DESCRIPTION members setparameterint and getparameterint will need to point to static functions of type FMOD_DSP_SETPARAM_INT_CALLBACK and FMOD_DSP_GETPARAM_INT_CALLBACK, respectively, if any integer parameters are declared.

These will be displayed as dials in FMOD Studio's effect deck.

Boolean Parameters

Boolean parameters have type set to FMOD_DSP_PARAMETER_TYPE_BOOL. They are discrete, singled-valued parameters and their default value is defined by the booldesc member defaultval.

The FMOD_DSP_DESCRIPTION members setparameterbool and getparameterbool will need to point to static functions of type FMOD_DSP_SETPARAM_BOOL_CALLBACK and FMOD_DSP_GETPARAM_BOOL_CALLBACK, respectively, if any boolean parameters are declared.

These will be displayed as buttons in FMOD Studio's effect deck.

Data Parameters

Data parameters have type set to FMOD_DSP_PARAMETER_TYPE_DATA. These parameters can represent any type of data including built-in types which serve a special purpose in FMOD. The datadesc member datatype specifies the type of data stored in the parameter. Values 0 and above may be used to describe user types whereas negative values are reserved for special types.

The FMOD_DSP_DESCRIPTION members setparameterdata and getparameterdata will need to point to static functions of type FMOD_DSP_SETPARAM_DATA_CALLBACK and FMOD_DSP_GETPARAM_DATA_CALLBACK, respectively, if any data parameters with datatype 0 and above are declared.

Data parameters with datatype 0 and above will be displayed as drop-zones in FMOD Studio's effect deck. You can drag any file containing the data onto the drop-zone to set the parameter's value. Data is stored will be stored with the project just like other parameter types.