FMOD EVENT SYSTEM

BEST PRACTICES FOR PROGRAMMERS

http://www.fmod.org

Updated: Jan 09

LEGAL NOTICE

TABLE OF CONTENTS

This page is intentionally blank…

INTRODUCTION

What is the difference between FMOD Ex, FMOD Event System and FMOD Designer?

- FMOD Ex

  This is the “low-level” engine. It contains all the low-level “primitives” and capabilities that make FMOD so powerful e.g. the software mixer, DSP engine, hardware interface output modules, 3D capabilities etc. It’s quite reasonable to decide to use just FMOD Ex in your game especially if you’re transitioning from a legacy system or you already have sound designer tools that you want to keep using.

  If you do decide to use just FMOD Ex for your game, you won’t be able to use the FMOD Designer tool to create content for it.

- FMOD Event System

  The FMOD Event System is built on top of FMOD Ex. It is a powerful application layer on top of FMOD Ex that adds a wealth of features to the already powerful “low-level” engine. If FMOD Ex were likened to “C” (raw and powerful), the FMOD Event System could be likened to “C++” (higher level concepts, more abstract).

  The FMOD Event System is used to play back content created with the FMOD Designer tool – this vastly simplifies life for the Programmer because FMOD Designer puts a lot of creative power in the hands of the Sound Designer and the FMOD Event System takes care of a lot of the grunt work that the Programmer would ordinarily have to do.

  The FMOD Event System runs on content created by FMOD Designer. The two are inseparable.

- FMOD Designer

  FMOD Designer is a sound designer tool used for authoring complex sound events and music for playback with the FMOD Event System. It’s an easy to use graphical application (for both Windows and Macintosh) that puts a lot of creative power in the hands of the sound designer and allows them to create freely without having to collaborate with the programmer on every trivial matter. It presents the sound designer with a broad canvas and a palette that contains all of FMOD’s most powerful features and it stimulates experimentation and encourages creativity.

BENEFITS OF USING FMOD EX AS OPPOSED TO FMOD EVENT SYSTEM

It's ok to **not** use the FMOD Event System if it will only get in the way. Choose what to use based on your unique game/application. Specifically, it may be better not to use the FMOD Event System if you have to integrate with an existing legacy system that is very different in concept. Study the FMOD Event System and decide what features you would benefit from and what features would present themselves as limitations – don't commit to taking on the FMOD Event System and then fight it at every step. In this case, it may be better to just use FMOD Ex.

Features of FMOD Ex can quite easily be used alongside the FMOD Event System. You can pick and choose what to use for different parts of your game depending on your unique situation. For instance, you may decide to use the FMOD Event System for all your general sound events and music but you might use FMOD Ex to implement the low-level details of a legacy lip-synch engine which is tightly integrated into your animation system.

The choice is yours – you don't have to go all FMOD Ex or all FMOD Event System.

PURPOSE OF THIS DOCUMENT

This document is intended to provide advice on best practices for programmers using the FMOD Event System. Readers are advised to make themselves familiar with the Event System API and example programs that are shipped with the API before reading this document.

This document assumes the reader has programming experience and understands the basic methods required to implement the FMOD Event System.

FMOD EVENT SYSTEM

In this chapter of the document, the use of data in the FMOD Event System is described in detail. The chapter begins will an overview of the basic operations of the Event System. Memory management and load strategies are also considered in later sections.

WHAT FILES FMOD EVENT SYSTEM USES AT RUNTIME

When a Sound Designer builds a project using FMOD Designer, a number of files are produced. Of these files, only two file types are used at runtime. The FMOD Event System uses:

- .FEV

  An FEV file contains all the event definitions for one whole project. This is the file that the programmer loads (using EventSystem::load). It contains all information required to create events and it also contains information about its associated FSB files. The fact that there is information on the associated FSB files within each FEV means that you cannot generally rebuild the FSB files without rebuilding the FEV as well.

- .FSB

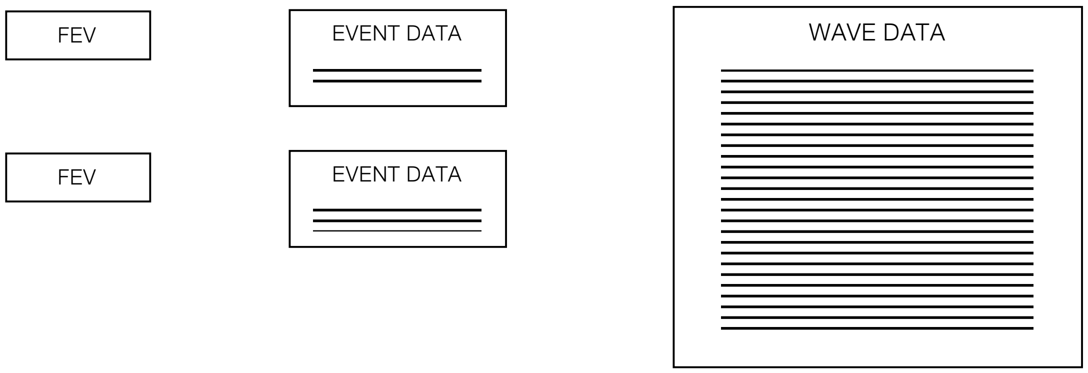
  An FSB file is a collection of sounds. There may be one or more FSB files associated with an FEV. FSB is a generic format that can be used with both FMOD Ex and FMOD Event System. In the case of the FMOD Event System, FMOD Designer creates one FSB for every Wave bank that the Sound Designer specifies. This is where the actual wave data for the events is stored.
  NOTE: Do not rebuild FSB files behind the FEV's back!

MEMORY MANAGEMENT

To begin with, it's highly recommended that you provide FMOD with a pre-allocated pool of memory to work with by calling FMOD::Memory_Initialize. FMOD will then use its own memory allocator and operate solely within that block of memory. This will help to reduce the load on the system heap - meaning less fragmentation of the system heap and possibly even faster performance. On many platforms, the system memory allocator is very slow and performance can suffer if it's used.

There are three major areas where FMOD Event System uses memory:

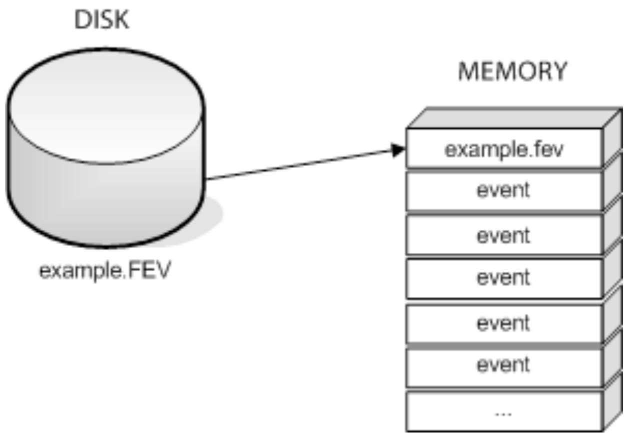
- FEV data in memory
- Event instance data
- Wave data



▲ **Figure 1: FEV data**

FEV DATA IN MEMORY

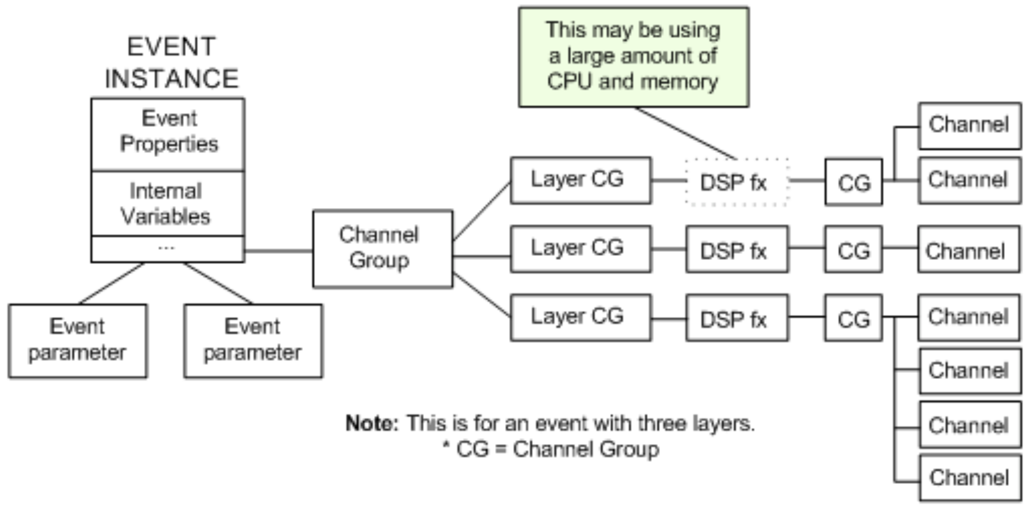
This is the memory FMOD Event System uses to store the FEV file in memory. When you load an FEV file you can expect it to use slightly more memory than it uses on disk. This memory is allocated when EventSystem::load() is called and will remain in use until the project is unloaded with EventProject::release() or EventSystem::unload().



▲ **Figure 2: FEV data in memory**

EVENT INSTANCE DATA

This is the memory required for instances of all the events in an FEV. If you think of FEV data in memory as the “definition” of an event, then an event instance is an actual working instance of that definition that can be manipulated at runtime, started, stopped, moved in 3D space etc. DSP effects that are placed in event layers are regarded as event instance data because they may need to allocate their own history buffers etc. for each event separate event instance. All the layers, sounds and effects in an event are tied together internally by ChannelGroups which must also be instanced in the same way.



▲ **Figure 3: Hierarchy of an event instance**

The number of event instances that FMOD Event System will create for each event corresponds directly to the event property “Max playbacks” which is controlled by the Sound Designer using FMOD Designer. If the Sound Designer sets “Max playbacks” to 50 then FMOD Event System will allocate memory for 50 instances of that event.

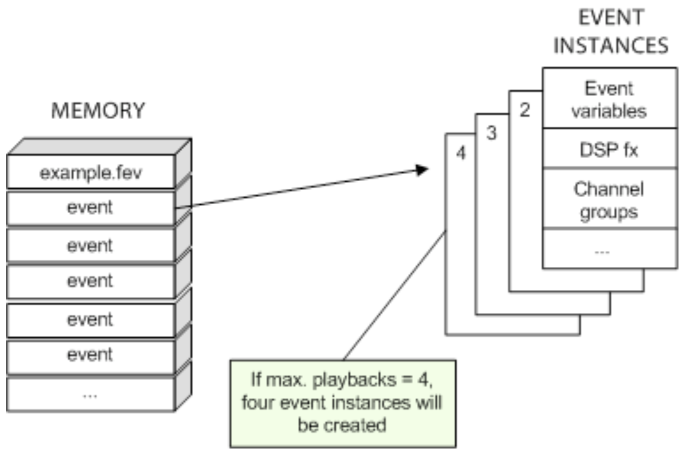
Event instances are created at one of two times:

- When you call getGroupXXX() with “cacheevents” = true, the FMOD Event System will create instances for **all** events in that group and all of that group’s subgroups.
- When you call getEventXXX(), the FMOD Event System will create instances for just that event.

Event instances are only freed when you call EventGroup::freeEventData() or when the project is unloaded. If you pass an event instance handle to EventGroup::freeEventData() then event instance data for just that event is freed (along with wave data), otherwise event instance data for the whole group will be freed (along with wave data).

When the FMOD Event System needs to allocate event instance data for an event, it always allocates **all** instances of an event at the same time. Conversely, it always frees all event instance data for an event at the same time. This means that if you have an event that has “Max playbacks” set to 50, FMOD Event System will allocate 50 instances up front the first time you get the event with getEventXXX().

Incidentally, setting “Max playbacks” to 50 is incredibly wasteful of memory. As a guideline, most events should have “Max playbacks” less than 10, usually less than 5.



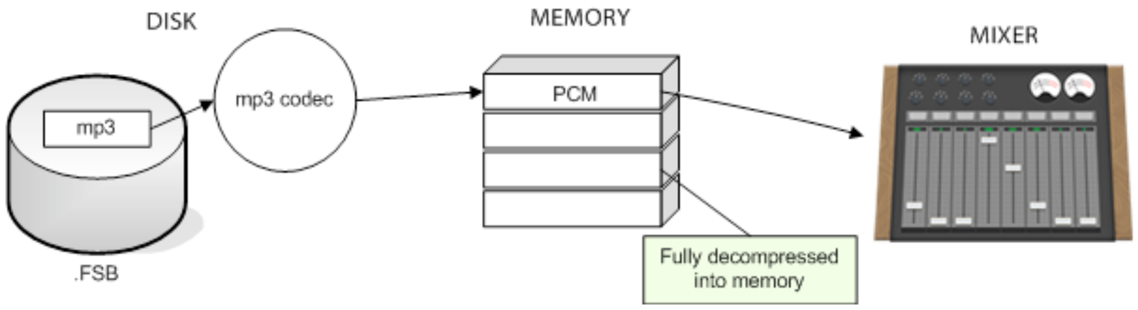
▲ Figure 4: Mulitple event instances

WAVE DATA

This is the actual waveform data used by the events. It is, by far, the largest user of memory in the FMOD Event System. Wave data is stored in .FSB files on disk and, when loaded into memory, will use different amounts of memory depending on the value of the wave bank property "Bank type".

Decompress into memory

When the FMOD Event System needs to load a subsound from a "Decompress into memory" wave bank, it will decompress the subsound into memory in full. This means it will be decompressed from MP3 or ADPCM or whatever the compression format of the bank into PCM in memory. As such it will use the most memory of any of the different bank types.

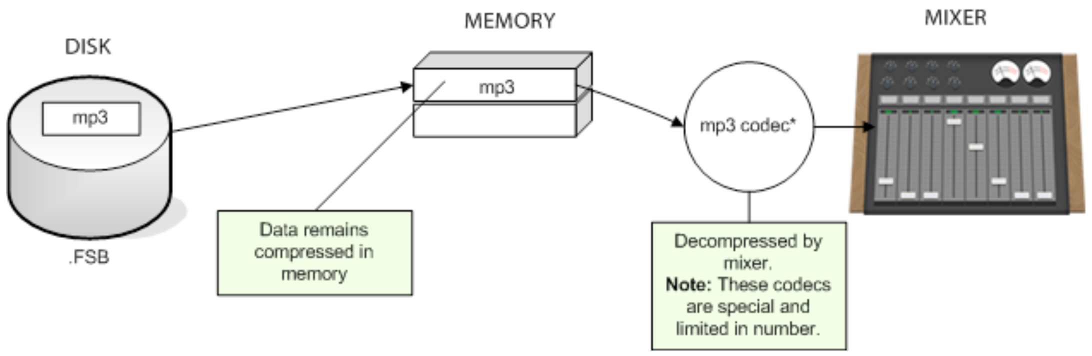


▲ Figure 5: Decompressing wave data into memory

Load into memory

Load into memory is similar to "Decompress into memory" in that the whole subsound is loaded into memory. The difference is that each subsound is stored in memory in its compressed form so it uses less memory. This means that if the wave bank property "Compression" is set to "MP3", the subsound will be stored in memory as MP3 data and fed directly to FMOD's software mixer. This method uses less memory but it uses more CPU when a sound is played because the MP3 is effectively being decompressed in real-time.

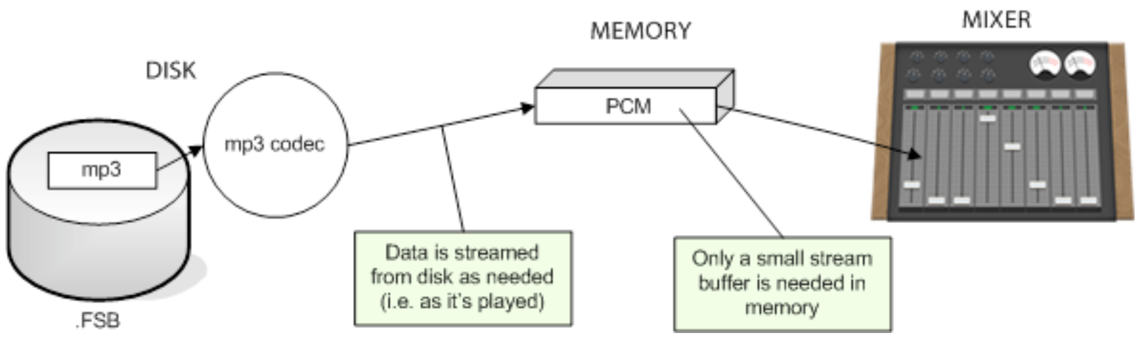
Note that not all formats can be played back directly from compressed data. Currently only MP3, MP2, ADPCM and XMA are supported. See FMOD_CREATECOMPRESSEDSAMPLE. If you use “Load into memory” with an unsupported format it will fall back internally to “Decompress into memory”.


DISK
mp3
.FSB
MEMORY
mp3
Data remains compressed in memory
mp3 codec*
Decompressed by mixer.
Note: These codecs are special and limited in number.
MIXER

▲ **Figure 6: Compressed wave data in memory**

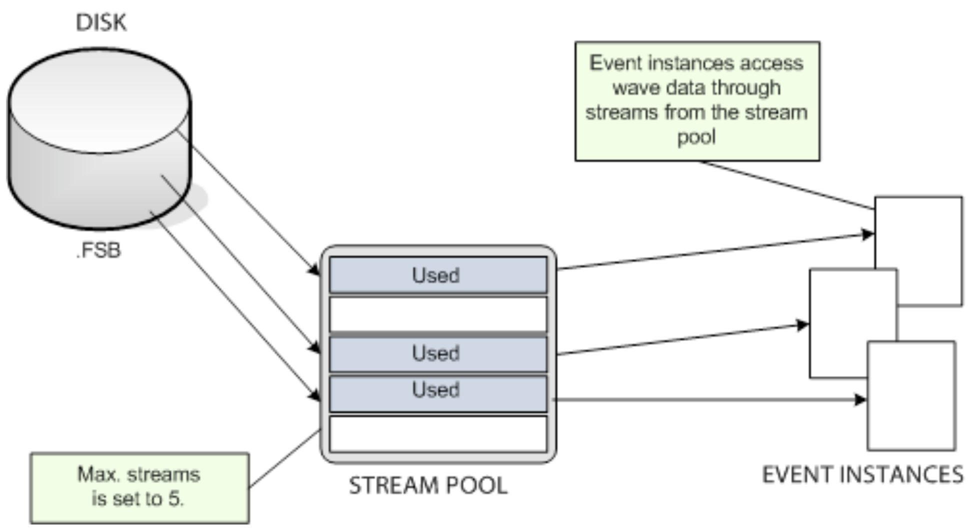
Stream from disk

With “Stream from disk” wave banks, subsounds are not loaded completely into memory – subsounds are streamed from disk as they’re played. This means they use a lot less memory than the other bank types but they generate regular disk reads as they’re played. If disk reads aren’t completed in a timely manner, e.g. other game systems are using the disk at the same time, playback of the subsound can stutter due to buffer underrun. Disk access is relatively slow so there may also be some latency between when a streamed subsound is triggered and when it is audible.


DISK
mp3
.FSB
mp3 codec
MEMORY
PCM
Data is streamed from disk as needed (i.e. as it’s played)
Only a small stream buffer is needed in memory
MIXER

▲ **Figure 7: Stream from disk**

It’s only possible to stream one sound at a time from any given wave bank – this is fundamental to how streams work. So, if the FMOD Event System needs to play two sounds at once from the same streamed wave bank then it needs to use two streams. If it needs to play fifty sounds then it needs to use fifty streams. The more streams accessing the disk at the same time the slower those disk accesses will be, to the point that streams may stutter due to underrun. In order to limit the number of streams that the FMOD Event System will create for any given wave bank, each wave bank has a “Max streams” property.


DISK
.FSB
Event instances access wave data through streams from the stream pool
Used
Used
Used
Max. streams is set to 5.
STREAM POOL
EVENT INSTANCES

▲ **Figure 8: Demonstrating maximum streams**

LOADING STRATEGIES

IN GENERAL

Loading

There are different methods available to Programmers to load data using the FMOD Event System. They are (in part):

- Use EventGroup::loadEventData to load event data per EventGroup
- Use getEventXXX to load event data per event

EventGroup::loadEventData loads data for all events in a group - getEventXXX loads data for just the specified event. You can call getEventXXX on a couple of events in a group and, at a later time, call EventGroup::loadEventData to load the rest of the group. FMOD will never load something twice so it's safe to call getEventXXX and loadEventData as many times as you like and FMOD will make sure everything's loaded correctly.

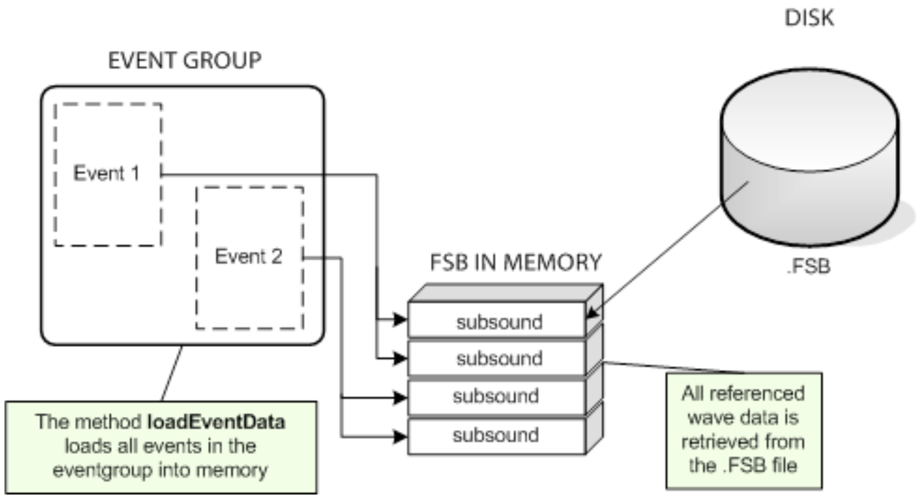
Freeing

If you keep calling getEventXXX for different events, FMOD will happily load event data for those events and that data will keep building up in memory unless you call EventGroup::freeEventData at some point (or EventProject::release, EventSystem::unload or EventSystem::release.)

This is a common pitfall! If your memory usage keeps rising the more you play your game, make sure you're calling EventGroup::freeEventData appropriately.
You can free event data per EventGroup or per event.

'LOAD EVERYTHING UP FRONT' MODEL

For small games or games that are level based etc. it may make sense to use EventGroup::loadEventData to load whole EventGroups worth of data at once. This will incur an initial delay while loading is occurring and may cause a large memory footprint but the trade off is that, after that initial hit, everything is ready to play instantaneously. You can then freeEventData on the same EventGroups when the level is unloaded.

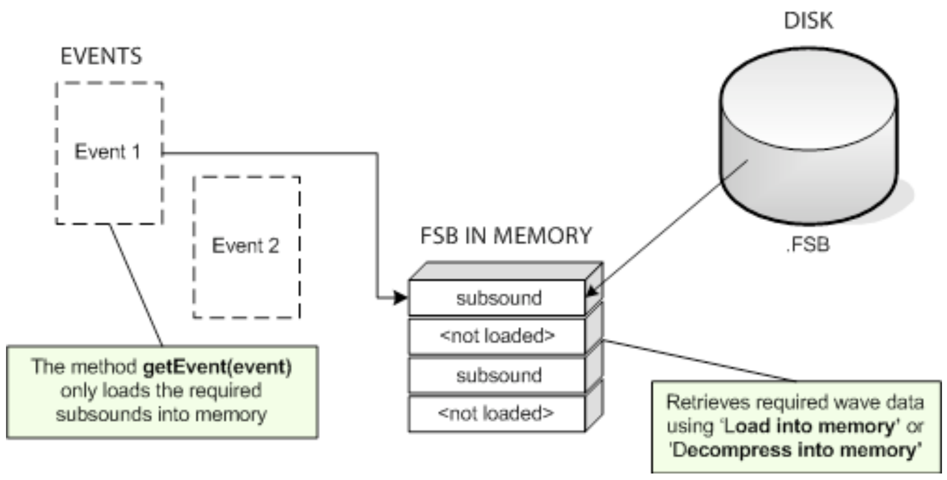

EVENT GROUP
Event 1
Event 2
FSB IN MEMORY
subsound
subsound
subsound
subsound
DISK
.FSB
The method **loadEventData** loads all events in the eventgroup into memory
All referenced wave data is retrieved from the .FSB file

▲ **Figure 9: Demonstrating the loadEventData method**

'RANDOM ACCESS' MODEL

For free-roaming games, mmo's etc. it makes more sense to use getEventXXX to load data per event as needed because you can't know ahead of time what to load. This means you have to accept that loading will occur on-the-fly and loading takes time – you need to handle the potential loading delay in your game. Load events before you need them e.g. if you're approaching a certain creature, call getEventXXX for the events it uses before the player gets to it so the events are ready to play when the player needs to hear them.

When an event is no longer required, use freeEventData and pass it the event handle to free the event data for just that event. To avoid constantly loading/freeing the same events, let unused events stay in memory for a while and only free them after they've been unused for a certain amount of time. Old, unused events will gradually get freed and you'll be left with a slowly evolving "working set" of events that balances memory footprint against excessive disk loading.


EVENTS
Event 1
Event 2
FSB IN MEMORY
subsound
<not loaded>
subsound
<not loaded>
DISK
.FSB
The method getEvent(event) only loads the required subsounds into memory
Retrieves required wave data using 'Load into memory' or 'Decompress into memory'

▲ **Figure 10: Demonstrating the getEvent(event) method**

HYBRID MODEL

Most games will benefit from using a bit of both models. You'll most likely load some things up front and load some things on-the-fly. The peculiarities of your game will dictate which method to use where.

REAL EXAMPLE

- UI and first person player sounds e.g. HUD, player grunts and groans. Loaded up front so they're always available in an instant. Group them into an event group in FMOD Designer and load the whole event group at once.
- Creature sounds. Make an event group per creature – load/unload whole event group as needed.
- Physics sounds like footsteps on grass, concrete, gravel etc. Load/unload per event as needed. Use a lazy unloading mechanism so events that haven't been used in a while get unloaded.
- Music. Load/unload per event. Wave banks set to "Stream from disk". Or, alternatively, use the interactive music feature of FMOD Designer.
- Voice-over/dialog. Use one event in conjunction with a "programmer" sound. Voice-over event is always loaded – programmer is responsible for loading/unloading each dialog line as needed. (See below)

′INFOONLY′ EVENTS

When calling one of the getEventXXX functions, it's possible to specify FMOD_EVENT_INFOONLY as the "mode" parameter. This flag tells FMOD to retrieve a *handle* to the actual FEV data of the event, rather than retrieve a real instance of the event.

When you specify FMOD_EVENT_INFOONLY, FMOD will return a handle to the 'INFOONLY' or 'parent' event – this is effectively a 'prototype' for that event. You cannot call Event::start on an 'INFOONLY' event – you need to get a real event instance to do that - but you can get and set the properties of an 'INFOONLY' event.

The 'INFOONLY' event has a number of uses:

- You can use an 'INFOONLY' event to query event properties without loading all the event data for that event. If you call getEventXXX without specifying FMOD_EVENT_INFOONLY, FMOD will make sure all instances of that event are allocated and all wave data for that event is loaded so that it's ready to be played. When you specify FMOD_EVENT_INFOONLY, FMOD will not load wave data or allocate instances for the event. This is an efficient way to query event properties while avoiding data loading and memory allocating when playback is not required.

- You can set 'default'" values for event properties using 'INFOONLY' events. If you get an 'INFOONLY' event and set it's 'Volume' property to 0.5f, whenever you subsequently call getEventXXX to get a real event instance, that real instance's 'Volume' property will be initialized to 0.5f by FMOD. This is a handy shortcut and it works for all event properties.

Setting default values is also fundamental to using the 'just fail if quietest' 'Max playbacks behavior'. This is explained in more detail below in the 'USE JUST FAIL IF QUIETEST' section.

COMMON USAGE

This section contains some common usage patterns and advice on optimal usage of FMOD Event System in general. As always, "there's more than one way to do it" so take this advice as a starting point in creating the best solution for your particular game.

USE A WRAPPER CLASS AND CALL GETEVENTXXX AS NEEDED

To save memory you need to keep the event property "Max playbacks" as low as possible. This means that you'll have, say, 1000 torches on the wall of your dungeon but only 4 event instances to produce the sound for all of them. This is fine – the player never needs to hear 1000 torches playing simultaneously.

So, you'll have 1000 "torch" game objects and each of these game objects can call getEventXXX when they're in suitable range of the player in order to acquire an event instance. When you call getEventXXX N+1 times (with N being "Max playbacks") a previously acquired event instance will be stolen – this means that one of your game objects will now have an event instance handle that has been made invalid. Whenever you use that invalid handle, FMOD will give you an FMOD_ERR_INVALID_HANDLE error.

You need to throw that event instance handle away because it's useless now. If that torch needs to be audible at a later time, it can call getEventXXX again to reacquire an event instance.

USE "JUST FAIL IF QUIETEST"

When you call getEventXXX and all instances of that event are in use, an event instance that's already playing must be stolen. Precisely which event instance gets stolen is determined by the "Max playbacks behavior" for the event in question. The most interesting "Max playbacks behaviour" for 3D events is "just fail if quietest".

When the "just fail if quietest" stealing behavior is invoked, FMOD calculates what the audibility of the new event instance will be and, if the new event instance will be less audible than any of the currently playing event instances, the getEventXXX will fail and no event instance will be stolen.

This means you can call getEventXXX for all torches within, say, a 20 meter radius of the player every game frame and FMOD will fail all the quietest events and allow only the most audible events to keep playing. This is the best way to keep "Max playbacks" low while still having a world containing thousands of sounds.

In order for this method to work, FMOD must calculate what the audibility of the new event instance will be. In order to do this, you need to set some parameters and properties for the event before calling getEventXXX. How is this done? Call getEventXXX with the FMOD_EVENT_INFOONLY flag. Set the required parameters and properties on the "INFOONLY" event and then, finally, call getEventXXX without the FMOD_EVENT_INFOONLY flag to attempt to get the event for real.

When setting parameters and properties on the "INFOONLY" event, set everything that will affect the event's final audible volume. This may include event volume, 3D position, orientation (if cones are used), any event parameters that contain volume curves, occlusion etc.
See the "max_playbacks" example program for a demonstration of this technique.

USE EVENT CALLBACKS

Use Event::setCallback to set a callback function for your events. This function takes a "userdata" parameter which you can use to pass in a pointer to your wrapper class. Then, in the callback function, which must be a C function, you can cast the "userdata" parameter back to your wrapper class and have full access to the wrapper class that owns the event instance in question.

When an event instance is stolen, the callback function for that event instance is triggered with FMOD_EVENT_CALLBACKTYPE_STOLEN. This allows you to access your wrapper class immediately and clean up all references to the event instance just before it gets stolen.

You may also be interested in FMOD_EVENT_CALLBACKTYPE_EVENTFINISHED. This will be called when the event instance is stopped or finishes playing for any reason. This is useful for catching when oneshot events have finished playing.

USE "PROGRAMMER" SOUNDS FOR DIALOG/VOICE OVER

It's easy to waste a lot of memory when implementing dialog with FMOD Designer. The naïve approach is to create a separate event per dialog line. This generally means you'll have thousands of events which are identical in every way except for the fact that they play a different wave file.

This is incredibly wasteful because you've got a lot of duplicated data when all you really need is to choose a different wave file at runtime depending on what dialog line you want to play. Instead of creating an event per dialog line, the Sound Designer should create a single event to handle all dialog and make use of a "Programmer" sound entry. Note: Refer the Sound Designer to the FMOD Designer user manual for detailed instructions on creating an event using a 'Programmer' sound entry.

A "programmer" sound is like an empty container. When a "programmer" sound is played, FMOD calls the event callback with FMOD_EVENT_CALLBACKTYPE_SOUNDDEF_CREATE. The Programmer then creates an FMOD::Sound any way they like (maybe call System::createSound, maybe call getSubsound on an already open FSB stream, it doesn't matter) and passes a pointer to it back to FMOD. FMOD then uses this FMOD::Sound as it would any other that it created itself.
When FMOD has finished with the FMOD::Sound it calls the event callback again but this time with FMOD_EVENT_CALLBACKTYTPE_SOUNDDEF_RELEASE. This tells the Programmer that FMOD has finished with that FMOD:Sound and will not use it again. The FMOD::Sound can now be freed or released or whatever is necessary.

So, instead of thousands of nearly identical events, you now have only one event that has an empty container that the Programmer fills at runtime depending on which line of dialog is required… and you've saved yourself a very significant chunk of memory.

The creating and freeing of "programmer" sounds is completely independent of FMOD Designer and the FMOD Event System. You don't need to make a wave bank with all the dialog lines in it – you don't need to include all those lines anywhere in an FMOD Designer project. It's completely up to you how you manage the dialog lines – you may keep them as separate MP3 files (or even OGG if you like). You may use the FSBank tool to create a wave bank for all your dialog lines. The decision is yours.

LOAD/UNLOAD PROJECTS AS NEEDED

Multiple projects may be loaded at once. In many scenarios it's beneficial to break up your audio data into multiple projects and load/unload them as needed. This can be done to reduce the memory footprint and it will also make it easier for multiple Sound Designers to work on different parts of the game simultaneously. Break up projects by "level", "world", "creature" or whatever is applicable to your game.

USE NONBLOCKING LOADING

Pass FMOD_EVENT_NONBLOCKING to all calls to EventGroup::loadEventData and getEventXXX to tell FMOD to use nonblocking loading when loading data for those event(s). This will instruct FMOD to perform all loading in a separate thread so it won't block your main thread when it needs to perform lengthy file operations.

It's important to be consistent when using the FMOD_EVENT_NONBLOCKING flag because you can't change your mind at a later date. If you initially call EventGroup::loadEventData or getEventXXX with FMOD_EVENT_NONBLOCKING then those event(s) will always be in nonblocking mode until EventGroup::freeEventData is called on them. Likewise, if you initially call EventGroup::loadEventData or getEventXXX without FMOD_EVENT_NONBLOCKING then they'll be blocking until EventGroup::freeEventData is called on them.

To reduce blocking when loading FEV files with EventSystem::load, load them from memory. This means you physically load the FEV file into a memory buffer (any way you like) and then pass an FMOD_EVENT_LOADINFO structure with a valid "loadfrommemory_length" when calling EventSystem::load. If "loadfrommemory_length" is non-null, the "name_or_data" parameter to EventSystem::load is interpreted as a pointer to a memory buffer containing the FEV file.

When EventSystem::load returns you can immediately free the FEV file buffer.

OFFLINE BUILD PROCESS

There's a command line version of FMOD Designer called "fmod_designercl.exe". This version can be used to integrate the building of FMOD Designer projects with the rest of your automated build process.

When FMOD Designer builds a project it can (optionally) output a programmer report and a header (*.h) file containing defines for everything in the project. Using your favorite text processing language (like Perl for instance) you can parse the programmer report and extract information about the project, convert the data into a different form that is usable by the game, create some glue code, generate statistics and throw up warnings if a limit is exceeded or invoke other post-process steps on the built files. The format of the programmer report is specifically designed for this purpose.

Each project can also have a list of "Pre-build", "Post-build", "Pre-save" and "Post-save" commands that are executed at the appropriate time. "Pre-build" and "Post-build" commands are also run when using fmod_designercl.exe. Use these hooks to do pre- and post-processing – a common usage here is to interface with your source code management system like Perforce for example.

Incidentally, the .FDP files created by FMOD Designer use XML and can be read, modified and written by a savvy script programmer. Instead of parsing the programmer report, it may suit you better to parse the .FDP file itself and extract information from it directly.

A final word: Only the .FEV and .FSB files need to ship with the final game. Nothing else!

SIMPLE EVENTS

FMOD Designer and FMOD Event System have an optimized code and data path for "simple events" that uses less memory and less CPU at runtime.

FMOD Designer detects "simple events" at build-time and builds them into the .FEV file differently than normal events. FMOD Event System then uses an optimized code path to handle playback and manipulation of these "simple events" at runtime. There is no special switch or property that needs to be set to flag a specific event as "simple", FMOD Designer automatically treats an event as "simple" if it meets certain criteria. An event will be considered a "simple event" only if **all** the following criteria are met.

A "simple event" has:

- No parameters
- No effects
- No user properties
- Only one layer
- Only one sound (the sound definition may have multiple entries)

"Simple events" use significantly less resources than normal events and their use is encouraged where appropriate. By all means, make your "feature" events as complicated as you like but, if you can get away with using a "simple event" then do so.