Intro

All sounds in sm64 are processed in what are known as sequences. Sequences are binary packed music notation files that are similar to midi, and are written in what's known as Music Macro Language (MML). Sequences are generally referred to as m64 files, though other file types are used such as .com or .bin. This is because the same sequence file type is used in many other games, so other file names are used in those communities. For this tutorial I will refer to the entire sequence file as an MML or M64 file, and to the actual music part of it as a sequence (will make sense later).
Along with MML files, there are two other important audio formats that I won't go over in detail, which are audio banks and audio samples.
This tutorial is made under the assumption you have a basic working knowledge of how to make MML files with seq64, and is meant to supplement technical knowledge of how MMLs work in the game, as well as how to do more advanced techniques and effects.
If you are completely new to making music in sm64, I recommend you watch one of these video tutorials:

- https://www.youtube.com/watch?v=g-niLeto3yU
- https://www.youtube.com/watch?v=E8-RKA6Sw8M

MML Format

There are three objects that make up the MML file hierarchy, each one corresponding to a part of the sequence.
At the top level is the **sequence** object itself, which contains all data for the sequence. Everything declared in this object affects all other objects in the MML file, and the sequence object's most common use is declaring the location of its child objects, channels.
Channels are containers for holding effects and event info like which instrument to use, or how loud to play the notes. If you are familiar with midi, the channels here are very similar. There can be up to 16 channels in one sequence. Channels contain the location of the final object used in sequences called layers.
Layers are the smallest part of a sequence and they contain the actual notes played in the sequence. There can be 4 layers activated at one point in time per channel, and each layer can only play one note at a time.
So to recap, a MML file is made from the sequence object, which can hold up to 16 channel objects, each of which can hold 4 layer objects. Layers hold the notes that are played, channels hold the effects, instruments and other data about how to play those notes, and sequences hold the info and data that affect all the channels.

Sequence File Representations

M64 is a packed binary format, which means that it is not realistic to expect humans to be able to read and edit them. In order to do so, we use an intermediary format called MML (really just assembly). This format is not really intended for music editing, but it provides a plain text way of accessing the events which is fine for just tweaking the file. If you intend to edit a sequence seriously, convert it to a midi, or hopefully you have the midi it started with already.
The best way to get an MML is using seq64, which is a program for converting between binary, MML and midi formats. MML formatting is made entirely up using macros, which all correspond to a set of bytes in the binary version of the sequence.
When speaking about how sequences are constructed and MML events, it's best to use the MML macros to describe it. The macros will map to midi, but not 100% the same, so to be accurate you have to always describe how the MML works and how we try to represent that in midi files.

Understanding Sequence Macros

Macros are instructions to an assembler to do one or several operations, with one named instruction that is not part of the default set. MML is made up entirely of these, so MML is basically assembly with only custom instructions. This is a lot simpler than it seems, as the instructions are named to what they functionally do.
Macros will always follow the format of *Name* arg1, arg2 etc. The name will tell you the action, and the arguments will tell you what data the action will use. Ex. *seq_settempo* 0x80 will set the sequence tempo to be 0x80, or 128. What the values mean will depend on each macro, in this case it is in bpm, beats per minute.

```
_start:
      seq_setmutebhv 32
      seq_setmutescale 50
      seq_initchannels 0x0FFF
      seq_setvol 120
tsec0:
      seq_startchannel 0, tsec0_chn0
      seq_startchannel 1, tsec0_chn1
      seq_startchannel 2, tsec0_chn2
      seq_startchannel 3, tsec0_chn3
      seq_startchannel 4, tsec0_chn4
      seq_startchannel 5, tsec0_chn5
      seq_startchannel 6, tsec0_chn6
      seq_startchannel 7, tsec0_chn7
      seq_startchannel 8, tsec0_chn8
```

```
    seq_startchannel 9, tsec0_chn9
    seq_startchannel 10, tsec0_chn10
    seq_startchannel 11, tsec0_chn11
    seq_settempo 100
    seq_delay 6144
    seq_jump tsec0
    seq_disablechannels 0x0FFF
    seq_end
```

This is a sequence header that was generated by seq64. The setup here is simple, there is a start, a section, *tsec0,* and then the sequence ends.
You can functionally tell what each macro does from the name, for example *seq_setvol 120* sets the sequence volume to 120, *seq_startchannel 0, tsec0_chn0* defines where channel 0 is. It is a lot to take in all at once, but you won't have to read and look through all of the macros to understand a sequence, you will just have to know which specific ones to look for, which I will cover in a later section. Though if you are interested in learning all of these, you can find a list of all macros in sm64 decomp at this link: seq_macros.inc

MML Effects and Events

Effects are processes that are applied to every note while they are being played. Vibrato, volume, transposition, reverb and panning are the main effects used.
In general, effects are applied on layers or channels exclusively, and only affect notes allocated to that object.
Events are the driving force of MML. Every processed macro is an event, and each one will declare some action for the sequence to take until it is ready for the next event. The most common event is the note event, which simply just plays a note. Events can usually be applied on multiple objects, such as the volume change event which can be used in sequences and channels.
Events map very similarly to events in the midi standard. Usually, if there is an event in the MML format, then there is an equivalent in midi.

Important Effects and Events

- Volume - Available in channels and sequences, determines loudness of notes
- Instrument - Determines what instrument to use for notes. Used per channel or layer.
- Panning - Determines left/right mix of sound. Per channel or layer.
- Transposition - Shifts all notes in semitones. Available in all objects.
- Pitch Bend - Shifts note frequencies in cents. Used per channel.

- Tempo - Determines playback speed. Set in sequence object only.
- Vibrato - Continuous change in pitch back and forth over a range. Can alter the rate and amount of pitch change, channels only.
- Reverb - Simulates sound reverberating as if played in a concert hall, sets the wet/dry mix or echo. Used per channel.

There are more effects and events than these, but these are the ones you should focus on to start with. Events can usually be used in multiple objects, and will sum together. For example a transposition on a note will be the layer's transposition plus that layer's channel transposition plus the sequence's transposition. When making sequences, try to pay attention to these effects.

Instruments and Notes

Now that we know how events and effects are used in sequences, we can talk about the most important ones of all, instruments and notes. Primarily, these will be the most used events. Instruments will determine how notes sound, and notes will determine how that sound is played.
Instruments are decided by an index in an instrument bank. SM64 has 37 different instrument banks, though most of them have repeated instruments within them or are used for sound effects.

Instruments

An **instrument** itself is simply a sound sample (or samples) with info on its sounds. It tells the game which notes map to which sound samples, and how they're tuned. An instrument usually has one sample, but can have up to three different ones. To find out how instruments are mapped, you can use an age old hacking resource with all the instrument lists: Inst List. Or you can read the source inside of sm64 decomp. In decomp, if you navigate to */sound/sound_banks/* you will find all of the banks as json files. Json is not the file format the Nintendo devs used, but it's what the people who made decomp decided on. As a brief explanation on JSON, it can simply be thought of as a tree of information; an object full of variable definitions that can branch to more variable definitions. JSON variables can be numbers, strings, lists or dictionaries. A dictionary is a set of named pairs, and the root object of a JSON is a dictionary in banks.

Banks

Banks are a collection of instrument definitions, they are most easily accessible through the JSONs inside of the sm64 decomp source code. At the root of our bank JSONs we see a curly brace, this means our root data type is a dictionary. At the root there are five keys: date, sample_bank, envelopes, instruments, and instrument_list.

Envelopes simply describe how the volume of the sample changes over time, it's made up of pairs of 2 values, the first is a time, the second is a volume.
Sample_bank tells us what bank of sample files will be used for this sound bank. You can only use one, and in decomp this corresponds to the name of a folder in /sounds/samples/. For most sound banks, it uses “instruments”, though some other banks are used, for example “bowser_organ” is one such other bank. The consequence of this is that if you wanted to make a custom bank, you could not combine bowser organ sounds with default instrument sounds.
Instruments define what actual instruments we will be using. Instruments themselves are dictionaries, and they have several potential values. The important ones are: release_rate, envelope, sound, sound_hi/lo, normal_range_hi/lo. Envelope defines what envelope we use for the inst, and sound what sample we use. Sound hi/lo and normal range hi/lo will change what sample is used based on the note. Lo is the bottom of the range, and hi is the top of the range. Sound is what will be used between them.
To add some detail to ranges, notes start at 22 off from where gen midi starts, so the bottom is note 22. The lo range tells us where it goes up to, so for a value of 28, it means the sample goes from 0-27, which is 22-49 in gen midi. The hi range tells us where it starts after. So if the hi was also 28, it means it starts at note 29, or 51 in gen midi. Another way of viewing it, is that the lo and hi tell us the boundary, but are not inclusive, so sound will take all values between lo and hi inclusive. Ex. If lo = 14 and hi = 27, then sound will then have notes 14 to 27, and you can add 22 to convert to gen midi notes.
Instrument list is finally how we access instruments from the MML. When you define an instrument to be used for notes, you just give the channel a number. That number corresponds to the number in this list. Each channel can choose one bank, and from that bank we choose one inst for our notes to play. If there is a null in this list, it just inherits the last instrument defined in the list.

Percussion

On top of instruments, there is also percussion which is considered separately. Percussion requires many different sounds so it is not declared the same way in game; it has its own special mapping. Each bank has one set of percussion, which is always on instrument 0x7F. Percussion will be defined in the bank JSON as “percussion” inside the instrument dictionary. The big difference is that percussion defines a sound for each individual note and how each note is tuned, instead of using a set of samples and a predefined note tuning table.

Waves

On top of instruments, there are basic wave type instruments that you can always use in any bank. These are sine, square, triangle and sawtooth waves. These are generated

instruments, and don’t use sound samples. To use them, you need to use instruments 0x80 - 0x83. Unfortunately, you cannot select these instruments in midi, so you will always have to set these manually yourself in MML format. Since these are pure waves, they can be grating or hard to use properly, but if used well they grant a very unique sound, play around with them when you can.

- saw (80)
- tri (81)
- sine (82)
- square (83)

Notes

Notes are events inside a layer that tell the sequence player to play an instrument at a given pitch and volume for a certain amount of time. You can only have one note at a time in one layer, and they play one after the other inside the layer (rests can be also in layers though). A note event has 4 parameters it uses: duration, volume (or velocity), pitch and next event time. Once a note event is reached, it will play the note over the duration supplied at the given volume, and then wait until the next event which is given in a timestamp on the current note event.
Notes pitches are determined using a frequency table. To play a note at a certain pitch, the game multiplies the playback rate by a factor to simulate an increased frequency. A change of one octave is changing the pitch by a factor of 2. Going up is doubling the speed, going down is halving the speed. This not only changes the pitch but also the rate of sample playback. If you have a voice sample, a high pitch will sound like they are talking very fast.
When using effects or instrument tuning that changes the pitch, it will simply multiply the lookup value for pitch for that note by the effect frequency. For example if I pitch bend by 127, I increase pitch by one octave, and multiply my note frequency by an additional 2x.
There are several note event types, but all of them do the same thing described above, the only difference is how they are encoded into the file, which is not something you should worry about.

Sequence Parsing

Sm64 updates audio 240 times a second, or 4 times a frame at 60 ticks. During each audio update, all sequences are updated. There are three sequence channels in sm64: background, event, and sfx. Background is meant for the level music, and event is meant for songs played temporarily during events, such as the star collect jingle.
Each sequence channel can have one sequence playing at a time, and can use one audio bank at a time per channel. Every update, sequences are parsed one at a time,

then in each sequence all of the channels are parsed, and then in each channel all active layers are parsed.
Parsing of each object works the same. The game will read the first macro in the MML data stream for that object and execute its function, and then keep doing so until it reaches a macro that tells it to stop. For sequences and channels this is done with a delay macro, or an end/disable macro. For layers, it is done with a play note event, or a delay, or an end macro.
In general, a sequence will run through all the necessary setup at the beginning and only be left with layer parsing as it goes through notes. A sequence needs only to set up things like volume and tempo, and then it can wait with a long delay until it needs to change something; channels can do the same. In this circumstance, the sequence parsing will see that the sequence is waiting, look at channels, see they are waiting, and then it will parse the layers which will either execute a new note, or also be waiting, and then it will continue playing active notes.
The consequence of this is that sequences setup and fill a buffer of notes to execute, and then idle until called upon again. This makes active management of effects difficult via the sequence and channel objects. In order for there to be changing effects, events need to be pre programmed to execute at certain times, much like note events need to play at a certain time in order to form the correct melody. This also means you have to run through all events in order, you cannot pick up at any spot in the sequence, unless it is a pre-programmed section to start at.

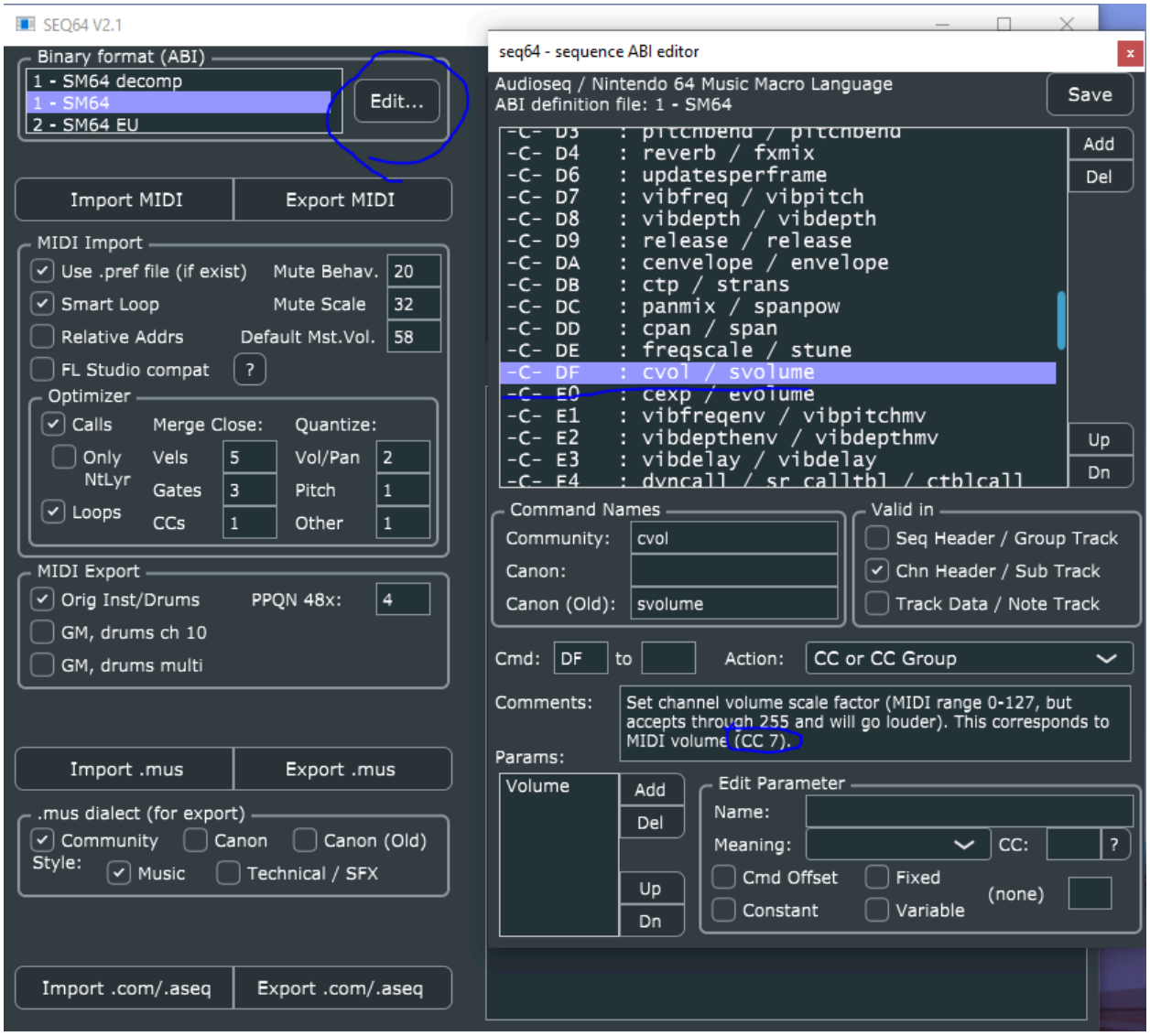
Creating Practical Effects for Songs

So now that we understand the general structure of MML and how it is read, we will learn about how to actually use it. I will be going over everything here in the MML format, but also briefly cover how to edit these effects inside of your midi if you're using seq64 2.x (1.5 for few effects).

Midi

As a brief mention, I will talk about how seq64 understands midi and how midi knows what it's actually doing to play notes. Everything in midi is controlled by events, very similar to what I covered earlier about MML. The difference in midi is that all events are covered inside of channels besides for what would be the equivalent of the sequence object, which really only contains the tempo and time signature (TS doesn't exist in MML though).
So this means that notes exist inside of channels and there can be as many at a time as you could potentially fit in a musical scale and these exist alongside all the other channel events. The important part here is how *other events* are edited inside of midi files.

If you use any event besides for basic *note_on*, *note_off*, *program_change*, and *pitch_bend* it will be in what is called a CC event, or a midi continuous controller. There are 128 midi CC events and each event has a 7 bit argument field, so you can have arguments up to 127.
As an example, if you want to define the volume for your channel, you can place midi CC 7 - *Channel_Volume*. You can see what midi CC events are mapped to MML events by looking at the XML file you load for seq64. It is even more clear in seq64 2.x as the comments line will usually tell you what midi CC to use for the corresponding macro.



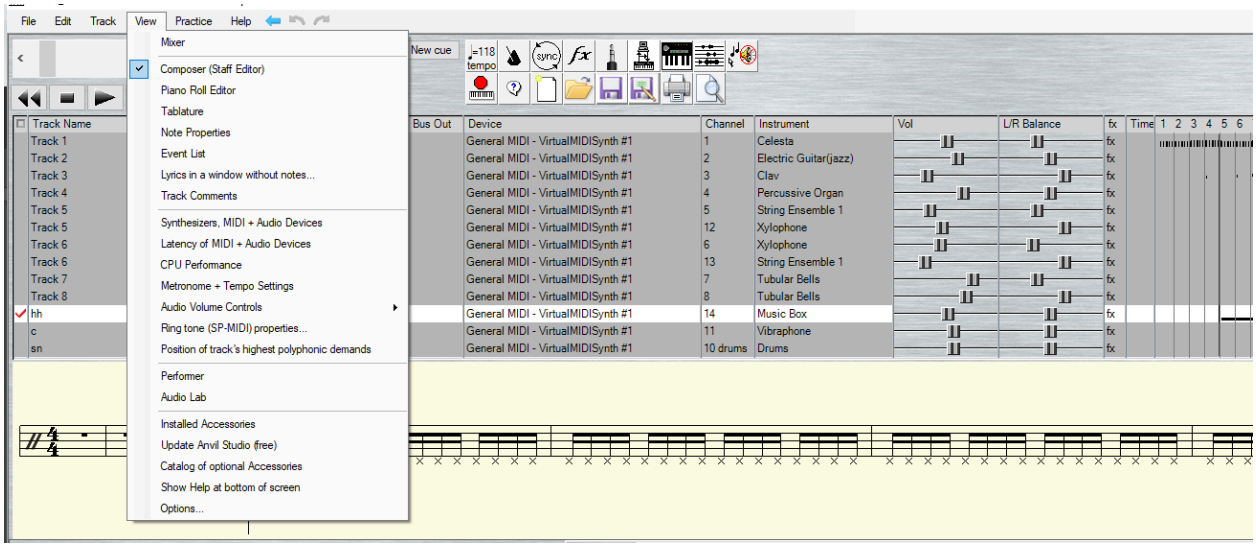
Here in seq64 2.1 I open the XML file by choosing sm64 and hitting edit, then I just navigate to the volume macro and read the comment to see which midi CC I need to edit to change how the volume will be translated. You can change this by using the UI options at the bottom under the comments and saving the XML if you want to as well, but currently it is mapped quite well. Keep this in mind as I cover effects.

Channel Effects

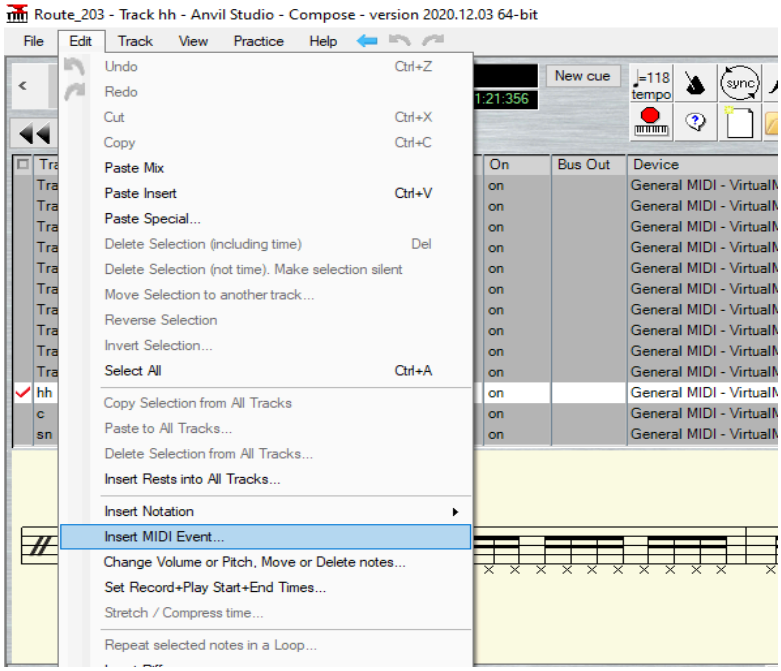
Effects that I covered in MML Effects and Events can be added easily to channels in MML format, but it can be confusing or unclear on how to add inside the midi itself so that it appears during import. Understanding how to do this will be the key to tuning your songs so that they sound good, and for most applications this is all you will need to understand.

As I mentioned earlier, MML events are converted by using equivalent midi CC events. So in order to have the effects you want present, you need to place the equivalent midi CC events in the correct place. For very common events, like volume and present, you will usually have direct control in the UI, and can set it there, but for less common ones like reverb, you need to place an event. Every midi sequencer should have an ability to do this. I will cover Anvil Studio as it's the most commonly used one for sm64.

Inside Anvil Studio, you need to open up the composer, or piano roll views and select a time where you want your effect to be present from.



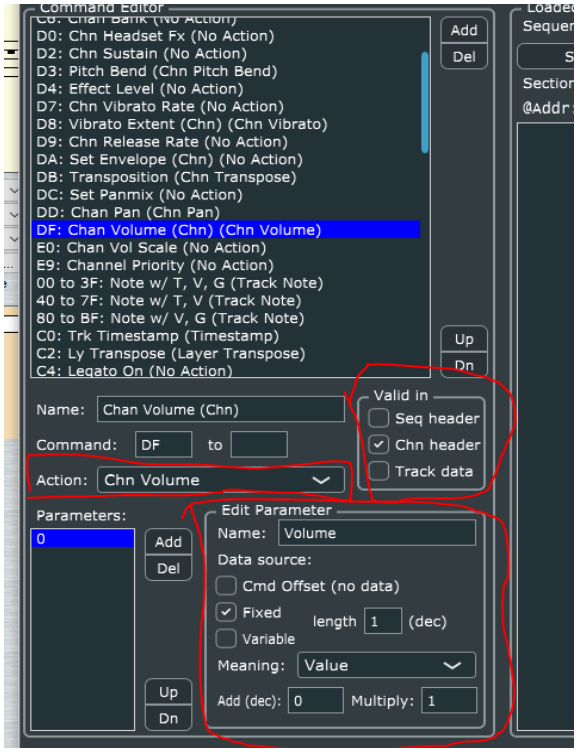
Now once you are here, you go into the *Edit* menu and select *Insert Midi Event...*



Now you will get a prompt telling you to select the kind of event and the value. Here is where we will take a break, and look to understand where to find our mappings to the MML effects. First I will start with seq64 1.5.

MML event mappings in SEQ64 1.5

All of the mappings to MML events are controlled by the XML file you decide to import. For seq64 1.5, this mapping is more crude, so I took it upon myself to make a more accurate mapping, and you can find that XML file here sm64 info XML. To view what this XML does, you need to load it via *RomDes->Load...* and then go to the AudioSeq tab. In the top left corner there is a *Command Editor* window that has all of our MML macros listed by their MSB or identifying byte and a text description of what they do. Click one of these commands, and below it you will see the encoding of how seq64 knows when to use it. The key components are: Valid In, Action, and Parameters.

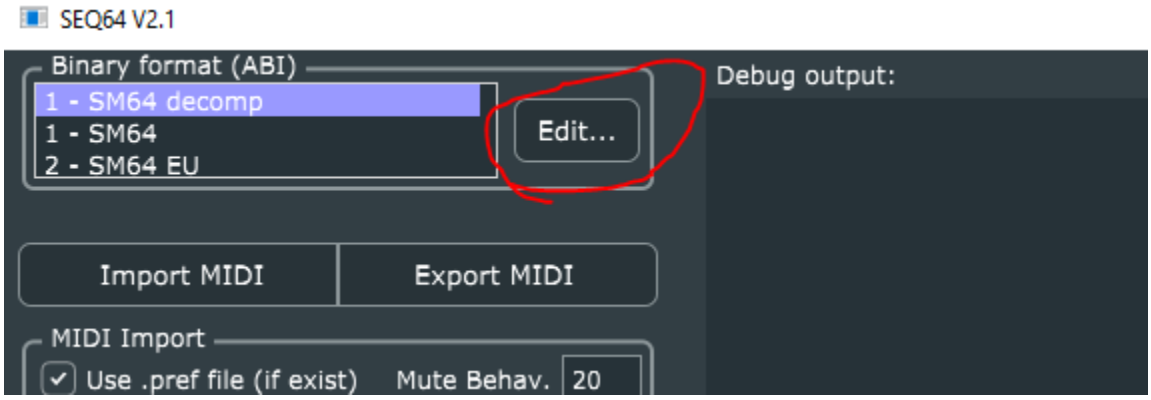


Valid in will tell you what objects to place the event if it finds it, though really this is only useful for placing events yourself, midi only has channel events, so it will only detect channel events. Action will tell you what action it is looking for, this is the event type. Chn Volume is a channel set volume midi event. There are others, like Chn Effects, or Chn Vibrato which is used for reverb and vibrato respectively. The parameter window will tell you how it interprets the value of the midi CC, and what other data you will have to put in yourself.
There is no direct indication of what midi CC event numbers to use in this version of seq64, you have to go off of naming, which is good enough for the basic effects. If you use the XML I posted earlier, everything should be set up to be intuitive, so you shouldn't have to edit anything or really check these. Now in our example, if we wanted to add in reverb to our channel, we would add event 91, ReverbSendLevel (naming potentially different in diff programs).

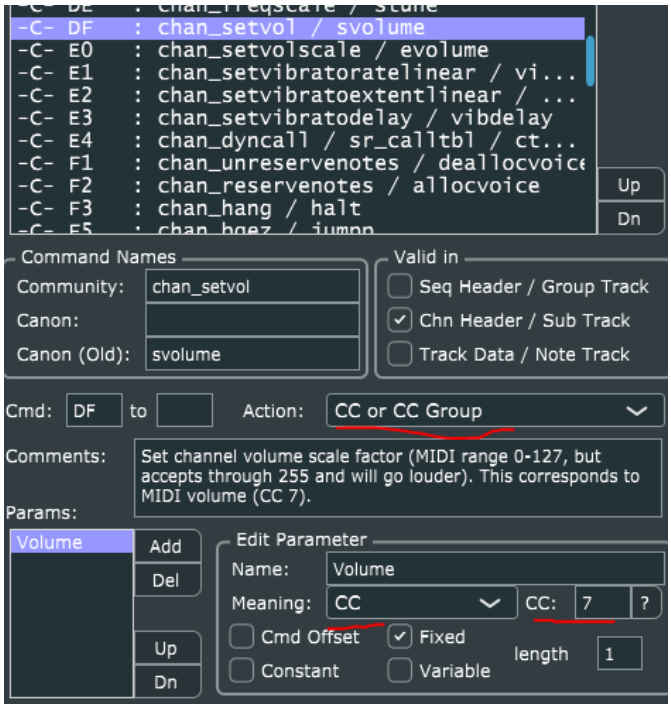
MML events in Seq64 2.1

Midi event mapping in seq64 2.1 is much more direct than in 1.5 as this was made with full exposure to the source, so all of the macros are accurate and are mapped to just one midi CC that matches in function. That said, I personally find the naming confusing and don't like the way it doesn't match sm64 decomp, so I made my own mapping that matches it, which you can find here sm64 decomp XML. Place this XML in the abi folder

included with your download of seq64 2.1. To expose the macros, select the appropriate XML file in the top window called *Binary Format (ABI)* and then hit Edit...



Now if you click any of these commands, you can look at Action, Params and Comments to figure out how it maps. If it is an event controlled by a midi CC (most are) then you should see something like this.



Action is wset to CC or CC Group, and our parameter tells us which CC to use. Here we can see that the volume is midi CC 7. Commands in this version can map to multiple CC, which is necessary for some more complex effects, I will go over those when I cover those effects. You can also see in the comments a good description of how this works, and the midi CC it will be using.

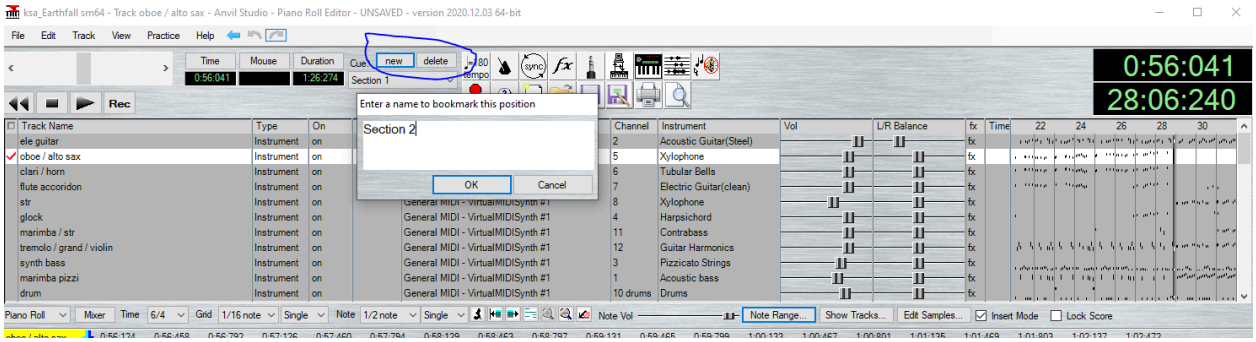
Common effects and their midi CCs

- Reverb - CC event 91 - Up to 127 in midi, but takes 255 in MML

- Vibrato Range - CC event 77 - Value of 4 per note of range
- Vibrato Rate - CC event 76 - Default is 32, range from 2048 - 8 ticks per cycle
- Sustain - CC event 64 - Off/On control in midi but value control in MML
- Program Change - CC event 129 - Sets the instrument, special midi event.
- Bank Change - CC event 0 - Sets the bank (2.1 exclusive)
- Release rate - CC event 72 - Must be after program change to work. Lower value is slower release.
- Pitch Bend - I do not recommend adding pitch bends via channel events, but otherwise, it is a special one, not a CC event.

Custom Loop Point

A loop point is made by separating all the events before and after a certain time, and then jumping back to that point in time at the end instead of the beginning. The only practical way to do this is using seq64 and editing the midi. Seq64 detects the points to separate events by using channel markers. Channel markers are meta midi events, and depending on your program how you define it is different, though the general idea is you want to select a point in your channel, and click *add midi event* in a menu somewhere, and then name the section “Section X” where X is the number of your section.



In anvil studio, you can do this by selecting where you want in the song and clicking the new button next to Cue.

Now inside of your MML file, you will notice you have multiple sections, this is signified by a label in the sequence header like this:

```
_start:
      seq_setmutebhv 128
      seq_initchannels 0x001F
tsec0:
      seq_startchannel 0, tsec0_chn0
      seq_startchannel 1, tsec0_chn1
      seq_startchannel 2, tsec0_chn2
```

```
      seq_startchannel 3, tsec0_chn3
      seq_startchannel 4, tsec0_chn4
      seq_settempo 128
      seq_delay 15
tsec1:
      seq_startchannel 0, tsec1_chn0
      seq_startchannel 1, tsec1_chn1
      seq_startchannel 2, tsec1_chn2
      seq_startchannel 3, tsec1_chn3
      seq_startchannel 4, tsec1_chn4
      seq_setvol 65
      seq_delay 6912
      seq_jump tsec1
      seq_disablechannels 0x001F
      seq_end
```

Our sections in this sequence are *tsec0* and *tsec1.* Now to loop to my custom point, I change *seq_jump* to point to *tsec1* instead of *tsec0* which is the default loop point.

Changing Banks

You can use multiple banks in a single sequence. This is done by editing the *gALBankSets* table, which is better known as */sound/sequences.json.* To add an extra bank, you need to to just add to the list after the name of your sequence, so you would replace *"03_level_grass": ["22"]* with *"03_level_grass": ["22", “23”]* to use both banks 0x22 and 0x23. This is not possible in Rom Manager.
By default, the first bank in the index is used for all channels. To use a different bank, you need to use the *chan_setbank, <bank>* macro. After you set the bank, you need to set the instrument. If you don’t it will default to instrument 0.

```
tsec0_chn0:
      chan_largenoteson
      chan_setlayer 0, tsec0_chn0_ly0
      chan_setbank 0x01
      chan_setinstr 6
      chan_delay 768
      chan_end
```

This is a typical channel for a sequence, the only difference is I set the bank before choosing my instrument. Layers will always use the channel's instrument unless defined

otherwise, so you can update the instrument in the channel anytime and it will affect all layers. The bank I choose will be the second one in the list defined in sequences.json. If this were the above sequence, this channel would move from using bank 0x22 to bank 0x23.

Tempo Changes

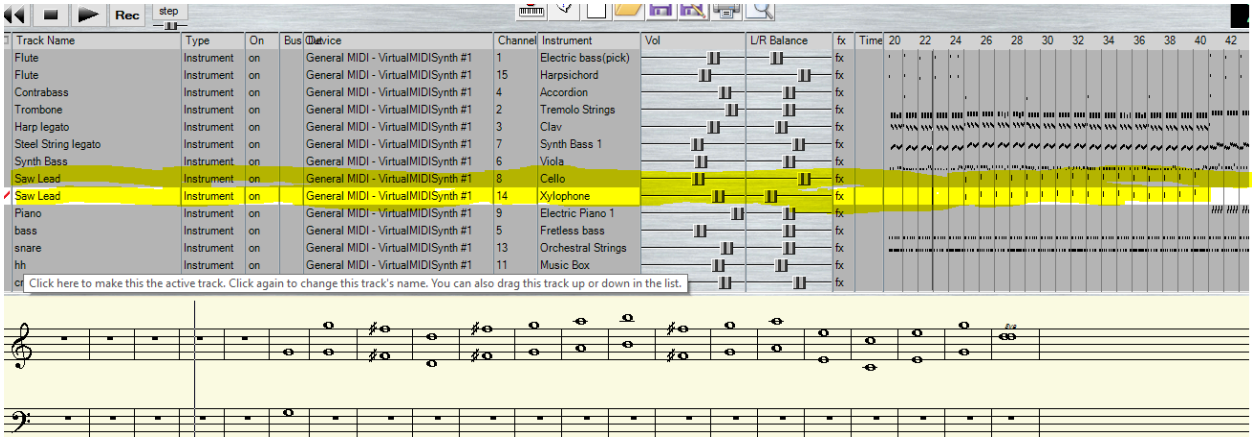
Tempo is controlled across the entire sequence. Any change made here will be a sequence event. To change the tempo, you should use *seq_settempo, <tempo>*, or you can insert a tempo event in the song properties. If you use anvil studio you can click the tempo button and it will prompt you to edit a new bpm at the selected time.
In MML you can also increase tempo by a certain amount. This is done by using *seq_addtempo, <increase>*. The point of this macro is to increase the tempo by a specific amount after a condition is met. It is important to note that this is a constant increase, not an incremental one. If you use this macro multiple times it will only increase tempo by the last macro used.
One interesting thing to do is to increase the tempo after looping a certain number of times. Here is a typical way in which you would do such a thing in MML.

```
_start:
      seq_setmutebhv 128
      seq_initchannels 0x001F
tsec0:
      seq_setvol 65
      seq_startchannel 0, tsec0_chn0
      seq_startchannel 1, tsec0_chn1
      seq_settempo 128
      seq_delay 15
tsec1:
      seq_loop 5
      seq_startchannel 0, tsec1_chn0
      seq_startchannel 1, tsec1_chn1
      seq_delay 6912
      seq_loopend
      seq_addtempo 25
      seq_jump tsec1
      seq_disablechannels 0x001F
      seq_end
```

So this is a pretty standard header, but I have modified the way it loops so that instead of simply jumping back to *tsec1* indefinitely, it will loop 5 times, use *seq_addtempo* and then jump to *tsec1.* This means after 5 loops, it will increase in speed, and then it will remain at that higher speed indefinitely.

Multi Layer Sound

When you want to fill out a sound, you add in layers, or in more literal terms, you double or triple the sound and alter it slightly so it can resonate. This is easy enough to do in a midi sequencer, just clone a channel, and then alter the volume/pan/instrument and you have something more full, and you can tune it how you want. As an example, a recent port I made had this backing track doubled and altered.



You can see that I altered the volumes, pan and instruments, but it's the same track. Though this adds an extra channel that you may not have access to, thankfully there is another way to do this with just one channel. If we look at a channel in MML format, we can see the recipe easily.

```
tsec1_chn5:
      chan_setinstr 46
      chan_setlayer 0, tsec1_chn6_ly2_1
      chan_setlayer 1, tsec1_chn6_ly1
      chan_delay 9216
      chan_pitchbend 0
      chan_end
```

We have two layers in our channel, but room for four, so we have extra room we are not utilizing in this channel to fill it out. If we duplicate the layer and apply a different effect on it we are getting more sound for little work, though the issue is, all of our effects must only act on layers. So we need to look at our possible layer effects. We can transpose, pan, and set an instrument, which is a bit limiting but is good enough for what we need most of the time. So to accomplish my goal, I will duplicate a layer, and then change the instrument.

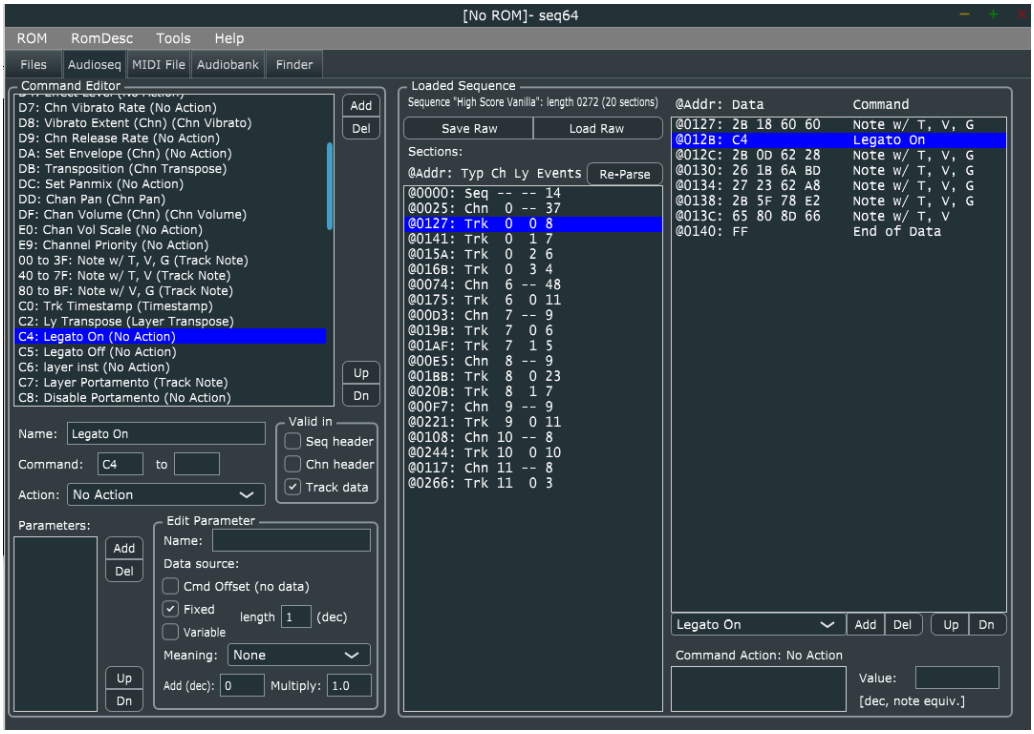
```
tsec1_chn5:
      chan_setinstr 46
      chan_setlayer 0, tsec1_chn6_ly2_1
      chan_setlayer 1, tsec1_chn6_ly1
      chan_setlayer 2, tsec1_chn6_ly2
      chan_delay 9216
      chan_pitchbend 0
      chan_end


tsec0_chn6_ly2:
      layer_setinstr 35
tsec0_chn6_ly2_1:
```

I gave the layer an index of 2, so I would not overwrite any other layer, and then a different name so it could hold my layer effect. Then in that layer, I simply set the instrument, and then place it before the layer I want it to inherit from. So in this case, layer 2 is the same as layer 0 but with a different instrument.

Legato

Legato is an effect that means to blend together, or to continuously play notes. In a technical sense, this means you don't restart the sample between notes. Legato is practically never used in sm64 because it is not an easy effect to use. Legato is a layer event exclusively, which means it will never inherit from midi, we must manually place it in our layers while in MML format.
If you are using seq64 1.5, I have added legato as a cmd, you can add it by clicking a Trk in the *Loaded Sequence* window and then add *Legato On*.



Legato will only work if there is no gap between notes, otherwise it will sound the same as if there was no legato.
In seq64 2.1, you need to export it to .mus (mml format) and then inside of your layer, you need to add *layer_somethingon* which is an awful name, but it is what it is called in sm64 decomp.
If you can't picture what this sounds like, here is an audio example. ghost town legato

Conclusion

With this tutorial, you should know how to do everything basic you need to do with sequences. If you have any questions contact me in a way I can be contacted,

- sm64romhackswebsite@gmail.com
- https://romhacking.com/user/jesusyoshi54
- https://www.youtube.com/c/jesusyoshi54
- https://gitlab.com/scuttlebugraiser

I will cover dynamic sequences, how to write asm like structures, envelopes, sfx and how to do completely overdo sequences in another tutorial (or multiple ones) somewhere down the line.