Intro

I will go over advanced effects and how to create dynamic sequences with scripting and create complex musical effects. You should have a working knowledge of sequences and how they work on a technical level, which you can get from reading my previous music tutorial here: SM64 Music Tutorial: Sequence Format Explained

Assembling Sequences

To do anything sufficiently advanced, you need to move beyond the realm of what seq64 can do by itself. Seq64 is still an important tool for IO between formats, but it does not support things midi does not support. Since all the things I will be going over in this tutorial require more than what midi provides, we need to be able to edit the sequence ourselves manually.

To do this, I suggest editing text macros in MML format (.mus) and then assembling to binary (.m64/.com/.aseq) yourself using armips. To do this, we need armips, a definition file with our macros, and some basic asm knowledge. If you don't know any asm, you can learn on the fly, we will only work with macros so you don't need much.

Necessary Programs and Files

First you need armips to assemble, get that by downloading the latest release here: armips link

The readme on that page will also have our reference on how to use macros and assembler functions to make our jobs easier.

Next you will need my macro inclusion file. This is the same as the decomp one, but edited to follow armips syntax. You can do this yourself, but mine is already formatted: seq_macros.asm

Finally to assemble, we need a .mus file exported with seq64 using my decomp matching xml, which you can get here: 1 - SM64 decomp.xml

Exporting midi to .mus then to .m64

- First take your midi and load it into seq64 while selecting my referenced sm64 decomp XML.
- Export in .mus format to the same folder with armips and the referenced seq_macros.asm inclusion file
- Add in the following header to the top of the file

```
sequence_start equ 0
.include "seq_macros.asm"
.n64
.create "test.m64", 0 ; export name can be changed at your
convenience
```

- Add in ".close" to the end of the file
- Open cmd, and execute armips by navigating to the folder with armips and typing "armips <sequence_name>.mus"

Previewing

Now to preview your file you can load it as any normal .m64 and play it, but if you're doing complex effects, I recommend you use my program to externally load sequences. You can download it here: link

You will have to compile it yourself from the source, it will require a working PC Port setup.

This port should allow you to load sequences easily from file, and edit parameters if you need to fine tune your sequences (might be a WIP depending on time since 4/16/22).

Complex Effects

Now with our ability to assemble sequences from .mus format, we can start with the effects seq64 cannot do. The most important one I have yet to cover and most useful to general users is portamento.

Portamentos

Portamento, or glissando or maybe just slurred notes, all refer to a similar effect where notes continuously change pitch instead of only existing at discrete steps. Portamento is a layer macro, and affects note cmds following it by altering their pitch. The portamento macro allows for a combination of several different pitch sliding effects, each with a specific use case. These are referred to as **modes** in the decomp source code.

Portamento Modes

There are 5 modes, and they change the amount of notes affected, and how the pitch ranges are decided. For the following modes, defined pitch refers to the pitch inside the portamento macro, and note pitch the one in the note macros.

1. Affects the next note only, Defined pitch -> Note pitch
2. Affects the next note only, Note pitch -> Defined pitch
3. Affects all notes until new mode. Defined pitch -> Note pitch
4. Affects all notes until new mode. Note pitch -> Defined pitch
5. Affects all notes until new mode. Pitch slides go from last note to current note.

Alongside defining how it slides, a portamento will define a target pitch, and a transition time. The sample used is always the sample on the higher of the two available note pitches.

Using portamento

The portamento macro is *layer_portamento* <mode>, <target pitch>, <slide time>. You can disable portamento by using the macro *layer_disableportamento*. Disabling portamento is the same as using mode 0. The time argument is based on sequence updates, which is the same time scale notes use. This time scale is in tatums, which are each 1/48th of a beat. One whole note in 4 • 4 music would be 192, which is one measure.

For modes 1 and 2, a common usage would be to use portamento before a percussion note to create a unique sound effect. You can use it on cymbals or drums to create unique sounds. Percussion generally has a narrow pitch range or only one allowed pitch, a portamento will allow you to change that. This makes it easy to know what pitches you are going to use, and what you should experiment with while a normal instrument track will have a lot of different notes and thus be hard to add portamento to.

- Portamento Modes Showcase Cabasa Beat

```
layer_loop:
     layer_portamento 0x81, 60-22, Quarter ; Quarter is 48
     ; 60-22 just translates midi note to sm64 note
     layer_note1 83-22, Quarter, 127

     layer_portamento 0x84, 79-22, 1
     layer_note1 83-22, Quarter/3, 127
     layer_note1 83-22, Quarter/3, 127
     layer_note1 83-22, Quarter/3, 127

     layer_portamento 0x81, 60-22, Quarter
     layer_note1 83-22, Quarter, 127
```

```
    layer_portamento 0x84, 79-22, 1
    layer_note1 83-22, Quarter/4, 127
    layer_note1 83-22, Quarter/4, 127
    layer_note1 83-22, Quarter/4, 127
    layer_delay Quarter/4

    layer_portamento 0x82, 50-22, Quarter/2
    layer_note1 83-22, Quarter, 127

    layer_jump layer_loop
```

I place the portamento before single notes, and sets of notes to set different sounds on each one. The first portamento is mode 1, so it slides from the target pitch to the note pitch. This makes it seem like the cabasa is being hit hard or slapped. The second usage is just to pitch shift the cabasa since it cannot go any lower on its own. I use mode 4 and a length of 1, so that it instantly goes to my target note. I use portamento mode 2 on the last note so that it stays at a low pitch for a long time. This gives the beat a dramatic type of finish and produces a sound you wouldn't be able to get otherwise.

For portamento modes 3 and 4 in normal songs, I would use them in specific phrases to give extra flare to notes. I prefer to use these on short notes, with short times. Whether you start or end at your target note depends on the melody, here are some examples.

- Portamento mode 3 - Sewer Speedway - CTR
- Portamento Mode 4 Steel Drum - Los Angeles Laps

For mode 5, I think of it as simply a way to continuously change between the notes playing in that layer. I place it at the start of a layer, and set the pitch to match the first note. Time can be tuned as needed.

- Portamento Mode 5 Synth Bass - Rave in the Grave - Shantae and the Pirate's Curse
- Portamento Mode 5 Steel Drum - Raise Thy Sword - Soul Calibur 2

Custom Envelopes and Release Rates

Instruments define how your notes are played by using a sound sample, a frequency tuning scale, an envelope and a release rate. When we define an instrument in our bank, all of these are set to make a specific sound, but we can alter these parameters per channel to emulate new instrument definitions inside our sequence.

Tuning is already set by notes, but could also be changed by transposing or with pitch bends, it is not that interesting to mess with outside of what you can do with midi. Release rate is a variable that defines how long it takes for the volume to fade to zero after ending a note. Envelope determines how the volume changes while the note is held. These two effects are what I will be focusing on for unique sounds.

Envelopes

All notes start at zero volume, and can scale to a max of two times their sample volume, though it is recommended to stay at or below 1x. **Envelopes** are pairs of time and volume values that determine how the note volume changes over time. Each pair represents one key frame that the volume will linearly interpolate to over the given time period; note volume starts at zero for the first keyframe. Envelopes have three meta commands that change the flow of the script. There is "stop" which will stop the note completely, "goto" which will go to the specified value pair, and "hang" which holds the current volume until the note is released.

Envelopes must be declared in the sequence, and then added to the channel using *chan_setenvelope <envelope>*. This macro needs to be used after the instrument is defined, or else the instrument's envelope will overwrite the defined one. Envelopes can be used to do several effects, or to alter the sound slightly to fit the song.

Tremolo

Tremolo is an effect where you modulate volume while you hold a note to give the feeling that the note is trembling. This is done by making the envelope oscillate between high and low values and repeating.

```
envelope_tremolo:
      envelope_line 1, 0x1800 ; time is in audio updates, 240/sec
      envelope_line 20, 0x7fff
      envelope_line 20, 0x1800
      envelope_goto 1

half_chn2:
      chan_largenoteson
      chan_setlayer 0, half_chn2_ly0
      chan_setinstr 3
      chan_setenvelope envelope_tremolo
      chan_delay 3648
      chan_end
```

This envelope would sound something like this: Tremolo on a simple C scale. You can alter the times and volumes to fine tune the effect.

High/Low Attack Envelopes

Some samples have an even volume (a wave or synth) while others have an aggressive ramp and decay (like a slap, or strum). You can alter these using envelopes. To make a wave feel accented or to enunciate the beat hit you can use a high attack envelope.

High Attack Envelope Cymbals - Expel - Star Ocean 2

```
env_cymbal:
      envelope_line 1, 0x6000
      envelope_line 10, 0x8f00
      envelope_line 100, 0x4000
      envelope_hang 4

tsec0_chn1:
      chan_largenoteson
      chan_setlayer 0, tsec0_chn1_ly0
      chan_setinstr 0xB
      chan_setvol 92 ; you would have to raise this to compensate for
      ; envelope overall being lower, I didn't do that here
      chan_setenvelope env_cymbal
      chan_call tsec0_chn0_call5
      chan_end
```

You can do the opposite to make certain instruments less aggressive, this can be good to create a dull hit effect, or emulate a reverse cymbal.

- Reverse Cymbal Audio Example

```
env_reverse_cymbal:
      envelope_line 2, 0x2000
      envelope_line 110, 0x6000
      envelope_line 30, 0x9f00
      envelope_hang 4

half_chn0:
      chan_largenoteson
```

```
      chan_setlayer 0, half_chn0_ly0
      chan_setinstr 11
      chan_setenvelope env_reverse_cymbal
      chan_setdecayrelease 0xff
```

Using a custom envelope along with legato can also be interesting as new notes will not reset the envelope. I think this is good to use for synths/strings in the background to add in ambient effects playing constantly throughout the song. You can combine these with loops to add them manually very easily.

- siren effect legato + tremolo

```
env_waves:
      envelope_line 1, 0x1000
      envelope_line 500, 0x2000
      envelope_line 500, 0x6000
      envelope_goto 1

half_chn0:
      chan_largenoteson
      chan_setlayer 0, half_chn0_ly0
      chan_setinstr 7
      chan_setenvelope env_waves
      qwerwqerqwerqwer:
      chan_delay 600
      chan_jump qwerwqerqwerqwer ; loops forever, but will go back to
start when seq does

half_chn0_ly0:
      layer_transpose 0
      layer_somethingon
layer_loop:
      layer_portamento 0x81, A(4), Whole ; A(4) is a func to define
note value, it just assembles into the corresponding pitch
      layer_note1 C(5), Whole, 127
      layer_jump layer_loop
```

Envelope IO

Seq64 does not do IO for envelopes well at all. It does not recognize custom envelopes, so if you do make any, save the .mus file and store it so you can retain the envelope information. If you do not save and try to convert your .m64 to a .mus, seq64 will export it as a table of u16s or bytes. Saving and retaining the .mus is necessary while you are editing and fine tuning the midi, and may be nice to do for safekeeping should you ever want to remix the song later.

Release Rate

Release rate determines how much a volume changes each update after being released. The higher the rate, the more a note is released. A release rate of 0 holds the note forever, while a rate of 255 removes it instantly.

Release rates, like envelopes, need to be set after instruments so the instrument's release rate does not overwrite the custom one. You can define a release rate using *chan_setdecayrelease <rate>.*

Release rate is best used to alter instruments to fit a more specific tone. For example you can use a low rate on cymbals to make them ring longer, or a high one on notes with reverb to make it feel choppy.

- Reverb + Release Rate (on perc) - Lets Run Through - Metal Slug 4

```
tsec1_chn9:
      chan_setlayer 0, tsec1_chn9_ly0
      chan_setreverb 112 ; higher reverb = much more echo
      chan_setdecayrelease 0xF0 ; higher is choppier
      chan_delay 9839
      chan_end
```

- Low Release Rate Cymbals High on Banjo - Route 203 - Pokemon Legends Arceus

Sequence Script IO and Values

Sequences support more than just playing notes; they can execute logic as well. This logic is very basic, and allows for just a few options: reading and writing values, jumping, branching and doing basic math operations. This does allow for much by itself, but for those who know a bit about computers, this is enough to be “Turing Complete”, which means theoretically it can do anything a computer is capable of computing.

The more important question is what would you actually want to do with this power. We have access to source code, so we could do any complex or computationally difficult problem there. What is advantageous to do inside the sequence itself? The answer lies in abstraction. We can heavily simplify our C code if it can generically be applied to all sequences, and if the sequence itself maps the applied functions to how it should sound. This can help with your own hack, but in the world of decomp, it helps everyone else who uses your code. Ideally, our sequence logic should need nothing more than code that writes values to the sequence to trigger our desired effects, and the sequence should map that to what events to run.

Script IO

Each channel in a sequence has an array of 8 bytes dedicated for IO between the sequence and external code. We can access this via this data path:

```
gSequencePlayers[ player ].channels[ channel ]->soundScriptIO[ IO slot ]
```

You can change the sequence ID, channel and script IDs to match your needs. Script IO values get initialized to -1 when you initialize a channel. This happens at the beginning of a sequence, during the *seq_initchannels <channel bitmask>* macro. Inside of a sequence channel, you can read/write that channel's, or another channel's script IO values with a macro for each operation.

- chan_iowriteval <IO slot>
- chan_ioreadval <IO slot>
- chan_ioreadvalsub <IO slot> ;value -= number in IO slot
- chan_iowriteval2 <chan>, <IO slot>
- chan_ioreadval2 <chan>, <IO slot>

When you execute a read cmd, it gets stored to the sequence register called **value**, when you write, it stores value to the designated spot. Value is a temporary u8 variable that exists for use until it is overwritten by another cmd that uses value, or until the sequence object currently being processed finishes. So for channels, this would be a *chan_delay <time>*, or a *chan_end* cmd, or anything else that reads/sets value. When you use *chan_ioreadval* specifically for script IO values under 4, it resets that script IO value to -1.

Value

Value is our singular temporary register during processing, and is used for many operations. It is used for our math operations: *chan_subtract <imm>* and *chan_bitand <imm>* and is used for branch checks and dyntables as well. The cmd *chan_bltz <target>* will check if value is less than zero, then branch to target for example. Most

importantly however is that value will be used for *chan_readseq* and *chan_writeseq*. This is unlike the IO read/write because these cmds affect the sequence itself. Now rather than go over the cmds in detail I will just start with examples of how to use these and ramp up the difficulty as I go.

Sequence Scripting Examples

Dyntables

Dyntables are the sequence equivalent of jump tables. A dyntable is an array of pointers, each called *sound_ref*, that point to layers, channels or other dyntables. Dyntables are set and called from channels exclusively, and use value to determine which pointer the sequence will access.

Set a dyntable with *chan_setdyntable <dyntable>,* and use it with one of the following.

- chan_dyncall
- chan_dynsetlayer <layer ID>
- chan_dynsetdyntable

The most common use for this is in sfx. You can find all the sfx in a file called (./sound/sequences/) sound_player.s which is actually sequence 0 in the game. Using a dyntable, you can use value to swap between many different sections of channel data while only using one actual channel object at a time. Sound_player.s combines this with sound script IO to choose channel data that has a single piece of sfx data on demand.

Dyntable usage

To use dyntables you need to create a table, then declare it in a channel.

```
.channel0_table:
     sound_ref .sound_action_jump_default
     sound_ref .sound_action_jump_grass
     sound_ref .sound_action_jump_water
     sound_ref .sound_action_jump_stone
     sound_ref .sound_action_jump_spooky
     sound_ref .sound_action_jump_snow
     sound_ref .sound_action_jump_ice
     sound_ref .sound_action_jump_sand
     sound_ref .sound_action_landing_default
...

.channel0:
```

```
	chan_largenoteson
	chan_setinstr 0
	chan_setval 0
	chan_iowriteval 5
	chan_setdyntable .channel0_table
```

The dyntable is a list of *sound_ref* cmds which are just byte encoded pointers. Each index is 2 bytes, which is one *sound_ref.* To find which value you need to use to jump to a desired location, you need to count the sound_refs encountered until your target. For example if we wanted *.sound_action_jump_sand* it would need value 7. So to jump to that, we would do this.

```
.call_act_sand:
	chan_setdyntable .channel0_table
	chan_setval 7
	chan_dyncall
	chan_end
```

We set value to 7 to select our target, and go to it using *chan_dyncall*. We use *chan_end* after it because our dyncall returns to the line after the call when it ends, we have to add *chan_end* or else it will start executing undefined behavior.

In general, you will mostly use these for sfx, so this covers the extent of necessary knowledge for most users. To add a new sfx, add a new *sound_ref* and the channel data for that ref. The sfx itself will just be normal sequence data in one channel.

Variations and Dyntables

In a regular sequence, or just one made to play background music, there is little reason for a dyntable. That said, it can be good to apply various effects abstractly based on IO values or variations instead of baking them into each specific channel.

Take for example a sequence you are going to use in multiple levels. You could use it unedited that way, but it might get stale or not fit exactly what you need. Adding in slight changes could spice up the sequence and make the player feel like the extra attention to detail is present.

Variations

The built-in way to alter how a sequence plays is called a **variation**. Sm64 supports calling a sequence with a variation by default, but only one variation. What this looks like is replacing the sequence id with seqID & SEQ_VARIATION. This results in a variation equal to -128, otherwise it is 0. This does not allow for much variety, but it is all the original game used. Examples of variations are the powerup theme for shells vs cap, the demo screen has a variation for game over and the water theme (DDD/JRB) changes which tracks play based on the variation.

When you call a sequence using a variation, it sets the variation in the sequence object to -128. You can access the variation inside the sequence using *seq_getvariation <unused>*. This will set value equal to the current variation. You can also set the variation and subtract value by the current variation inside sequence objects. This combined with the math operations we can do on value is enough to set the variation to any number we want, or manipulate it how we want. Unfortunately, we can only init it to -128 or 0, so it is not very useful for actually creating variations callable in standard ways.

If this is unsatisfactory, you can change the code yourself to allow for more. SEQ_VARIATION is defined in *./include/seq_ids.h* and the variation is set in *seq_player_play_sequence* inside of *./src/audio/external.c*. If you do not want to be editing the sequence after it has already started playing to have multiple variations, I would heavily recommend you change the way it is set.

Using Variations

The basic usage of a variation is to change what section of a sequence plays, or what channels play. You should create sections inside your sequence before importing so that you have places to jump to for your variations. To make use of variations, the first thing you need to do is init the sequence with your desired variation, and then inside the header you need to get the variation, and branch or jump based on that variation.

```
_start:
      seq_setmutebhv 128
      seq_initchannels 0xC000
      seq_getvariation 0 ; argument does not matter, but can be
unsafe if it is over 15
      seq_bltz tsec0 ; -128 is var is used
      seq_initchannels 0xCE43
tsec0:
      seq_startchannel 0, tsec0_chn0
```

```
      seq_startchannel 1, tsec0_chn1 ; Does nothing if chan not init
      seq_startchannel 6, tsec0_chn6
      seq_startchannel 9, tsec0_chn9
      seq_startchannel 10, tsec0_chn10
      seq_startchannel 11, tsec0_chn11
      seq_startchannel 14, tsec0_chn14
      seq_startchannel 15, tsec0_chn15
      seq_settempo 133
      seq_setvol 75
      seq_delay 9600
      seq_jump tsec0
      seq_disablechannels 0xCE43
      seq_end
```

This is the water sequence used in the original game, slightly modified for easier reading. What we vary with the variation is the channels used; we do this by using *seq_inithchannels <channels bitwise>*. If the variation is -128, we will init 0xC000, which is only channels 14 and 15. Otherwise with init 0xCE43 which is all of the listed channels.

```
_start:
      seq_setmutebhv 128
      seq_initchannels 0x3FFB
tsec0_:
      seq_startchannel 0, tsec0_chn0
      ... ; repeat until chan 13
      seq_setvol 80
      seq_settempo 172
      seq_delay 1
      seq_getvariation 0
      seq_bgez tsec_10 ; if variation is 0 goto tsec_10
tsec1:
      seq_startchannel 0, tsec1_chn0
      ... ; repeat until chan 13
      seq_settempo 99
      seq_delay 48
      seq_settempo 90
      seq_delay 48
```

```
      seq_settempo 78
      seq_delay 384
      seq_settempo 119
      seq_delay 96
      seq_settempo 172
      seq_delay 192
      seq_jump tsec0_15
; tsec0
; only played on variation
tsec_10:
      seq_startchannel 0, tsec0_chn0_28
      ... ; repeat until chan 13
      seq_setvol 80
      seq_delay 384
tsec0_15:
      seq_startchannel 0, tsec0_chn0_41
      ... ; repeat until chan 13
      seq_delay 13824
      seq_jump tsec0_15
      seq_disablechannels 0x3FFB
      seq_end
; Section total ticks: 14208
```

Above we have another use of variation, this time from the demo screen. This is a variation used to play an entirely different section of the song when present. At the start, we init everything the same, and start the same channels. We delay one update, and then check the variation and branch. This is done because the first set of channels have initialization data necessary to play notes before the later channels get set.

So if we have a variation, we play a different section of music, and then both sections merge into one final section. This results in a different intro playing depending on the variation, but the loop sounding the same. This variation is used on game over, to play the section of music where everything sounds disorienting and wobbly, where normally it plays the intro loop instead.

More complicated variations

If you want to alter the sound of your music without creating entirely unique song sections, then you will need to get more involved with your sequence. In this example, I want to lower tempo, add reverb, lower the volume, lower pitch and increase panning

with my variation to represent being underground in a second area. Many of these are channel effects, but I do not want to have copies of all my channels as that would increase the size of my sequence by a lot and be troublesome to edit. Instead, I will use a dyntable and common effect channels.

```
_start:
      seq_setmutebhv 32
      seq_setmutescale 50
      seq_initchannels 0x7DFF
      seq_getvariation 0
      seq_beqz tsec0
underground_chans:
      seq_startchannel 0, under_0
      ... ; repeat until 14
      seq_setvol 100 ; you could edit vol here if you wanted
      seq_delay 1 ; I need one update to init everything with
      ; proper effects, then I can skip to second section
      seq_settempo 123
      seq_jump tsec1 ; skip the tempo set
tsec0:
      seq_startchannel 0, tsec0_chn0
      ... ; repeat until 14
      seq_setvol 100
      seq_settempo 130
      seq_delay 1536
_start_20:
      seq_settempo 133
tsec1:
      seq_startchannel 0, tsec1_chn0
      ... ; repeat until 14
      seq_delay 9984
      seq_jump tsec1
      seq_disablechannels 0x7DFF
      seq_end
```

Inside my sequence I get my variation and enter my custom section when the variation is set. I set my lower tempo in this new section, and go to a different set of channels which is from a second section inside my midi. I have to create new channels for the

custom section I made, but they will follow a simple prototype, so I will just post one and it should be easy to copy it for all the other channels.

```
; underground_chans
under_0:
      chan_setval 0
      chan_jump tsec0_chn0 + 2
; tsec 0
; this is the original channel, with an extra cmd added to set val
tsec0_chn0:
      chan_setval 1
      chan_largenoteson
```

All I alter here is the value for my underground channel and normal channel. I jump to the equivalent normal section from my underground channel but after the value cmd. I do this so I can use a dyntable unique to each channel if needed with my effect variations, or in other words, I have a unique value in each channel based on the variation. This can abstract to as many variations as you want, with just copy/pasting and replacing words to create unique section names, so it's easy to expand out with just basic word processing software. All the work will be abstracted away to snippets of channels that carry the effects, I will not edit any effects inside the original channels so that my sequence stays clean.

```
; dyntables
dyn_effects_1:
      sound_ref effects_1_underground
      sound_ref effects_1_normal

dyn_effects_2:
      sound_ref effects_2_underground
      sound_ref effects_2_normal

effects_2_underground:
      chan_setreverb 40
      chan_transpose 0
      chan_setvolscale 0x68
      chan_setpanmix 0xC0
      chan_end
```

```
effects_1_underground:
      chan_setreverb 50
      chan_transpose -1
      chan_setvolscale 0x60
      chan_setpanmix 0xC0
      chan_end

effects_1_normal:
      chan_setreverb 5
      chan_end

effects_2_normal:
      chan_setreverb 5
      chan_end
```

For my sequence, it is enough to just have two dyntables. I create my effects using *chan_setreverb*, *chan_transpose*, *chan_setvolscale* and *chan_setpanmix*. I use these effects because they are ways of modifying the scale of common effects without actually altering them per channel. This way I can amplify/dampen a sound without having to change the tuning of each channel. In my two tables, I have one where all my effects are applied, and another where I do not lower volume as much, use lower reverb, and do not transpose. I use this on percussion and the bassline, which I do not think sounds as good with such heavy effects.

Now in order to execute these, I need to set a dyntable and call it in each channel. I do this by adding these cmds to the top of every channel in the first section.

```
tsec0_chn0:
      chan_setval 1
      ; underground chans jump here with val 0
      Chan_setdyntable dyn_effects_1
      chan_dyncall
      chan_largenoteson
```

If there is a variation, I am going to go into the underground section of my sequence, which acts the same as the normal sequence except with lower tempo, and each channel gets a value of 0. In my normal channels I have set a value of 1, added a dyntable and dyncall to choose my desired effects. I can fine tune all of my effect

scalings inside of the effect channels without having to touch the channel original tuning, and if I want to have different effect scaling on each channel, I can choose a different dyntable. In my sequence, I swap *dyn_effects_1* for *dyn_effects_2* on the channels where I see fit. Here are the final results.

- Default Aurora Area - Kirby Return to Dreamland
- Underground Effects Aurora Area - Kirby Return to Dreamland

Dynamic Pitch

Dynamic pitch is how red coin sounds are programmed. Sfx programming is relatively simple, but first I need to cover the adhoc IO values sfx uses. IO value 4 is the sound ID, 0 enables the bank and 5 is for reverb. These are set by external functions when you call *void play_sound(s32 soundBits, f32 *pos)*.

```
.sound_menu_collect_red_coin:
chan_setinstr 128
chan_setnotepriority 14
chan_setpanmix 0
chan_setenvelope .envelope_3378
chan_ioreadval 4 ; start of dynamic effects
chan_subtract 0x28
chan_readseq .major_scale
chan_writeseq 0, .transpose_by_coin_index, 1
chan_setlayer 0, .layer_3146
chan_setlayer 1, .layer_3168
chan_setlayer 2, .layer_3148
chan_end
```

This MML is extracted from sound_player.s and is the red coin sound effect. This effect is obtained by using a table that reads script IO values and branches to this label with a dyntable. There is more to the overall sfx structure but it is not important to understanding this individual sfx.

After setting basic chan params, we have *chan_ioreadval 4*. Four refers to the sound ID, which for reds is 0x28 + coin number. After we read our sound ID value, we subtract 0x28 so we are left with just our coin number in value. Next we read from *.major_scale*.

```
.major_scale:
     .byte 0
     .byte 2
```

```
    .byte 4
    .byte 5
    .byte 7
    .byte 9
    .byte 11
    .byte 12
```

The cmd *chan_readval <target>* reads from target plus the current value. For example, if target is at offset 0x500 in the sequence, and value is 5, we will read from 0x505. So what we end up reading is *.major_scale* offset by our coin number, which is just setting our value to our scale number. So if our value is 2, it reads 2 bytes after the target specified, which would be *.byte 4* in this case. This allows us to read our major scale as if it was an array. So ultimately, we converted our sound ID into a value in an array. Next we write to *.transpose_by_coin_index*.

```
.transpose_by_coin_index:
    layer_transpose 0
    layer_end
```

The cmd *chan_writeseq <imm>, <target>, <offset>* has three args, Imm, target, and offset. The cmd writes imm + value at location target + offset. So in our case, *chan_writeseq 0, .transpose_by_coin_index, 1*, we write value to *.transepose_by_coin_index* plus 1. That location is the argument to *layer_transpose*.

So in conclusion, we load the sound ID from our soundscript, subtract 0x28, read a value in an array using that new value and set our transposition to our array value. Thus giving us a dynamic pitch using the same sounds. If we wanted to do this without script IO, we would need to copy the sound 8 times, which is not efficient.

Conclusion

This tutorial should reasonably cover the complex effects you may want to try in order to spice up your sequences, as well as how to make use of scripting and variations at a basic level. I will have one tutorial on scripting and how to do basically anything with MML. If you have any questions or suggestions about this or other tutorials, contact me in a way I can be contacted.

- https://www.discordapp.com/uers/scutte
- https://romhacking.com/user/jesusyoshi54
- https://www.youtube.com/c/jesusyoshi54
- https://gitlab.com/scuttlebugraiser