Intro

I will go over scripting inside of sequences, composing in MML and how the audio system is structured in general.

Before reading this, you should be very familiar with sm64 sequences or read my previous tutorials which you can find here.

Scripting

Scripting in sequences has 3 main goals. To control what part of a sequence is executed on demand, to edit sequence parameters on demand and to be able to write back data for external access when required.

In the previous tutorial I covered what value is and some ways we can use it and variations to execute different pieces of the sequence, or edit how the sequence plays. All of the examples provided were executed from sequence start, but to master scripting we need to be able to control value at any point in the sequence.

Accessing Sequences in Code

There is a lot you can do outside of sequences with code and it is usually much easier than doing the equivalent actions in MML. You can write to any of the sequence player struct values with one line, and if you know what sequence you are playing this can achieve almost all of the same effects as scripting effects in your MML can. Even if you are set on using MML scripts to do certain tasks, you need to write to the MML in order to respond to the game state anyway. Below is a quick cheat sheet on accessing sequence player values. Make sure you inlucde */src/audio/load.h* in your file to get the variable definitions.

```
/*
Use these defs to pick the correct player
#define SEQ_PLAYER_LEVEL            0  // Level background music
#define SEQ_PLAYER_ENV              1  // Misc music like the puzzle
jingle
#define SEQ_PLAYER_SFX              2  // Sound effects
*/
// set a sound script IO for a channel
gSequencePlayers[player].channels[j]->soundScriptIO[i];
// set the variation for a player
```

```
gSequencePlayers[player].seqVariation;

// setting other values directly via code
// seq volume
gSequencePlayers[player].fadeVolume = Float_Value;
// seq tempo
gSequencePlayers[player].tempo = S16_Value;

// chan reverb
gSequencePlayers[player].channels[chan]->reverb = S16_Value;
// chan volume
gSequencePlayers[player].channels[chan]->volume = Float_Value;
// chan transposition
gSequencePlayers[player].channels[chan]->transposition = S16_Value;
// ... you can find other values in /src/audio/internal.h by looking
at struct definitions
```

There is nothing inherently dangerous with doing all sequence manipulation in the game thread instead of using scripting for dynamic sequences. In fact I recommend you do that for most things, but there is a disadvantage with doing this.

When you are playing a sequence, you don't exactly know where you are in the MML unless you create your own time management system. If you were to do this, you would then also have to write specific timing instructions for each sequence if you wanted to sync events. You also don't have direct access to when effects are applied and would have to create data tables for each sequence to manage effects and preset values, which you would have to manage separately from the actual sequence. Anything related to the internal state of the sequence would require constant monitoring to stay on top of, which is why I would recommend doing these things inside of the sequence itself.

Continuous Script Updates

Our goal is a responsive script that will change state when called upon, rather than when it is initiated. To create a responsive script we need to continuously check for changes in our script state while parsing the sequence, and then create the proper controls to change smoothly when needed.

To continuously check for changes is a simple task. We can do this using *chan_jump <target>* cmds within our script and *chan_delay <length>* cmds. The basic setup of a channel is to create a layer, add effects, and then wait until new effects or layers are

needed. During this waiting time, we can loop or jump backwards in the script to execute sequence logic and check for state changes.

```
chan0_tsec0:
      chan_largenoteson
      chan_setlayer 0, layer0_test
      chan_setinstr 4
      chan_setvol 100
chan0_loop0:
      chan_delay 192 ; 4 beats worth
      chan_IO_set 0, exec_chan_effect ; this is custom macro
      chan_jump chan0_loop
```

Channels can loop forever, or can be set to loop a finite number of times. Channels are restarted whenever the sequence loops, so the end time on them is of no consequence. This allows us to easily add our scripting checks at the end of a channel with no worry of timing synchronization.

In this example, I check for effect updates every 4 beats, which is one measure in 4 • 4 music. This is something much more difficult (or annoying at least) to do via code alone. You can replace this with *48*num_beats* for any other time signature your music has, or whatever frequency you want really and it will work just as well.

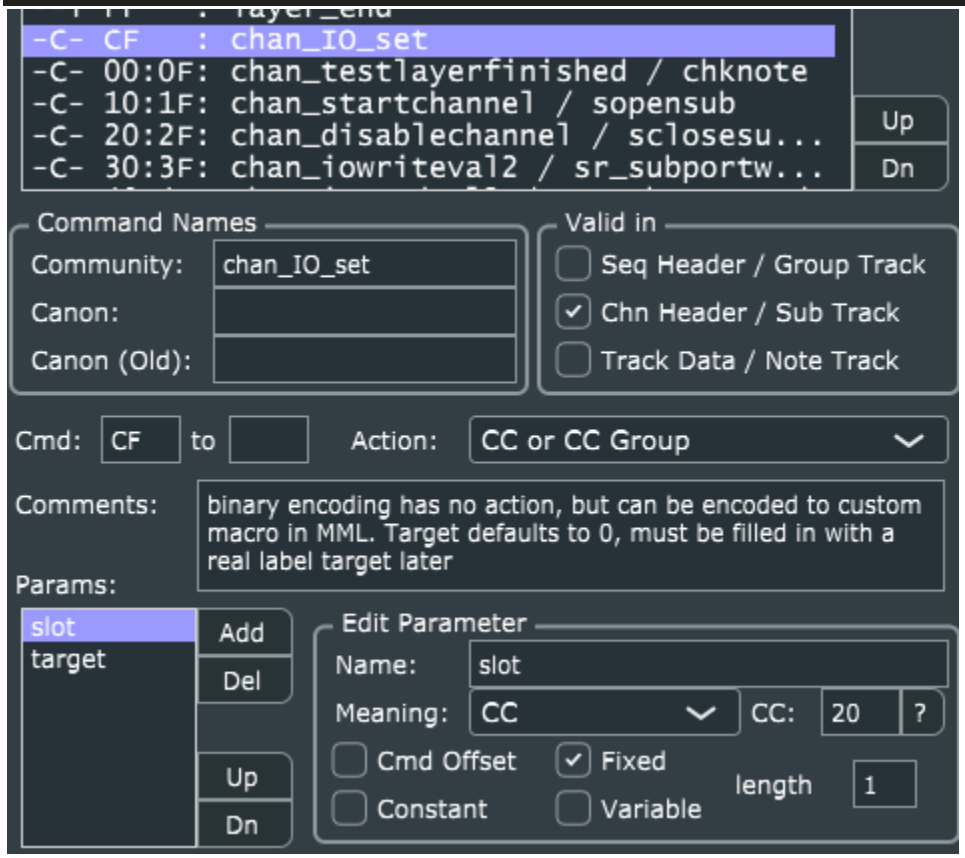
If we have various channel events throughout the timeframe of our sequence, it becomes harder to add these checks. We need to either manually sync time up with existing cmds, or destroy the timing in those cmds. This is something commonly seen in channels with pitch bends or volume fade/swells.

```
chan0_tsec0:
      chan_largenoteson
      chan_setlayer 0, layer0_test
      chan_setinstr 4
      chan_setvol 100
      chan_delay 50
      chan_setvol 90
      chan_delay 10
      chan_setvol 80
      chan_delay 10
      chan_setvol 70
      chan_delay 192-10-10-50 ; 122
      chan_IO_set 0, exec_chan_effect ; this is custom macro
```

```
      ...repeat for rest of chan data
```

I manually add together the delays and then check for changes every 192 updates. You can also add the custom macros into your midi file itself by adding a new midi CC event and mapping it to a custom MML cmd in seq64. This is the easiest method and the one with the most permanence. You can add as many new macros as you want to customize your specific midi as needed, and you can even create new ABI mapping files for specific needs.

```
.macro chan_IO_set, slot, target
      chan_ioreadval slot
      .if slot < 4
            chan_iowriteval slot ; slot is reset if < 4
      .endif
      chan_bltz @skip ; @ creates a static label
      chan_call target
      @skip:
.endmacro
```



I create mappings for custom MML cmds that are macros I add to my seq_macros.asm file. You can customize the macros to your liking but keep in mind midi CCs can only have byte arguments. I can add these to my midi file by picking the same midi CC event I mapped them to, choosing the correct arguments, and then filling in the text targets later.

Responsive Channel Effects

With the above scheme, we can regularly check for updates to our sequence via *chan_IO_set <slot> <target>* or similar macros. In order to execute logic, we need to read inputs to our channel and execute channel/sequence events, without breaking the timing of our script, meaning we need to do it all in one audio update.

Set Effect To Target On Demand

The easiest thing to do is to set a channel effect to a value. We can do this by setting up an ad hoc schema relating scriptIO slots to specific effects. This is something the SFX script does, and can easily be done for either just one channel or all channels at once. In this example, I will set slot 0 as the channel volume and update it constantly. I will define a new macro for this that requires only a slot number so I can insert this with a single midi CC.

```
chn0_tsec0:
	chan_largenoteson
	chan_setlayer 0, test_ly0
	chan_setvol 100 ; gen by orig script
	chan_setinstr 7
	chan_update_vol 0 ; custom macro
	chan_end

.macro chan_update_vol, slot
	@loop:
		chan_ioreadval slot
		chan_bltz @no_vol_set
		chan_writeseq 0, @vol_set, 1
	@vol_set:
		chan_setvolscale 127
	@no_vol_set:
		chan_delay1
	chan_jump @loop
.endmacro
```

In my macro I read from the sound script IO using *chan_ioreadval 0* and write value to the channel volscale cmd in my macro. I use volscale so that I can make this arbitrary for each channel rather than try to balance channel volumes in the sound script IO values.

I check if value is negative so I don't initialize my volscale to -1, but only update it if I set the sound script IO. I don't need to check if the slot is less than 4 and preserve value as it does not matter if it is reset or not in this case, I only care if the volume is set.

I can use this to set all channels at once to my target value if I replace the *chan_ioreadval <slot>* with *chan_ioreadval2 0, <slot>*. This will make any channel with this macro set use channels 0 slot to sync the volscale.

Set To Preset Value on Demand

Instead of setting a target value, we might want to set our effects to some preset value. This is something I did in the previous tutorial, but with variations. Now we will do the same thing on demand.

The calling convention is the same as before, we will regularly call our channel effect update script, we just need to edit our script macros to use dyntables.

```
chn0_tsec0:
      chan_largenoteson
      chan_setlayer 0, test_ly0
      chan_setvol 100 ; gen by orig script
      chan_setinstr 7
      chan_update_dyn 0, eff_table_0 ; custom macro
      chan_end

.macro chan_update_dyn, slot, dyntable
      chan_setdyntable dyntable
      @loop:
            chan_ioreadval slot
            .if slot < 4
                  chan_iowriteval slot
            .endif
            chan_dyncall
            chan_delay Whole ; check every 4 beats, Whole equal to 192
      chan_jump @loop
.endmacro
```

```
eff_table_0:
      sound_ref preset0
      sound_ref preset1
; just short examples for this tut, use whatever you want
preset0:
      chan_setreverb 60
      chan_setvolscale 127
      chan_end
preset1:
      chan_setreverb 30
      chan_setvolscale 100
      chan_end
```

I am using a dyntable here with several preset channel effects. I create a dyntable and dyncall based on the script IO value to choose a combination of effects. You can choose any number of effects from just volume to an entire suite of channel parameters. Dyncall already checks for an unset value so there is no need to add an extra safety check.

What's great about this is you can customize presets per channel and sequence without having to write specific macros per channel or sequence. Simply change the dyntable referenced in each channel and your effects and you have full control over it. You can even bundle a generic set of effects and dyntable in a macro so that adding it takes just one line.

Responsive Channel Management

Swapping channel data will allow us to change the notes playing, and enter new musical sections. This is something that we can use in several ways, from simply making an iterative sequence that only advances on demand to one that progressively adds melodies as you reach certain gameplay milestones. How we use it is up to the song setup and there are tons of possibilities. I will cover some likely scenarios and setups that allow versatile use.

Progressive Sequence Advancement

In this setup, we will advance our sequence to the next section only when allowed to, and otherwise will loop. This works best with something that has a short loop, or for a song you can loop one measure at the tail end of a section that you don't expect to loop for long. Try to make the advancement checkpoints close together and the respective gameplay tolerant of music not exactly in sync.

```
_start:
```

```
      seq_setmutebhv 32
      seq_setmutescale 50
      seq_initchannels 0x7DFF
tsec0:
      seq_startchannel 0, tsec0_chn0
      ; ... repeat for all channels
      seq_setvol 100
      seq_settempo 130
; I create this new label so I don't init my channels on loop and
potentially interrupt their advancements
tsec0_end:
      seq_delay 1536
      seq_jump tsec0_end ; added so that section 0 loops until I
allow it to progress

/* This section would normally play with all these channels, but I am
skipping it
This can create an issue because I have a tempo change at the start
of the new section
I will show how to deal with this in another example.
_start_20:
      seq_settempo 133
tsec1:
      seq_startchannel 0, tsec1_chn0
      ; ... repeat for all channels
      seq_delay 9984
      seq_jump _start_20
*/
_end:
      seq_disablechannels 0x7DFF
      seq_end

; I skipped tsec0_chn0 because it had a lot of bloat in my example
midi

tsec0_chn1:
      chan_largenoteson
      chan_setlayer 0, tsec0_chn1_ly0
      chan_setlayer 1, tsec0_chn1_ly1
```

```
      chan_setlayer 2, tsec0_chn1_ly2
      chan_setvol 119
      chan_setpan 64
      chan_pitchbend 0
      chan_setinstr 44
      chan_delay 1536 ; delay is calculated by seq64
      repeat_chan_chk_adv_sec 0, tsec1_chn1, tsec0_chn1 ; my custom
macro
      chan_end

; other tsec0 channels omitted for length

; section 1 channels

tsec1_chn1:
      chan_setlayer 0, tsec1_chn1_ly0
      chan_setlayer 1, tsec1_chn1_ly1
      chan_setlayer 2, tsec1_chn1_ly2
      chan_delay 9970
      chan_pitchbend 0
      chan_delay 14
      chan_jump tsec1_chn1 ; seq header no longer sets channels so we
have to loop channel data ourselves

.macro repeat_chan_chk_adv_sec, slot, next_chan, cur_chan
      chan_ioreadval slot
      .if slot < 4
            chan_iowriteval slot ; slot is reset if < 4
      .endif
      chan_bltz cur_chan ; repeat channel
      chan_jump next_chan ; adv to next channel
.endmacro
```

What we do is simple: we check for state changes after our section ends and then we jump to the next section's channel if true. In order to make this work in the MML, you will need to create multiple sections inside the midi. This works because of the manipulation of the sequence object. Instead of advancing to the next section after a delay, I just repeat the delay with a jump cmd, and then I comment out all of the other section data in the sequence object.

As I mention in the comment, this causes an issue as we lose sequence data like the tempo changes. Ideally, you would simply set the variation data for the sequence and branch that way rather than use the channel data, but this is a contrived example for the tutorial's sake. If you want to branch via variation to advance sections, then you just need to set up a variation read and branch in the header instead of a jump like I have.

```
_start:
	seq_setmutebhv 32
	seq_setmutescale 50
	seq_initchannels 0x7DFF
tsec0:
	seq_startchannel 0, tsec0_chn0
	; ... repeat for all channels
	seq_setvol 100
	seq_settempo 130
	seq_delay 1536
	seq_getvariation 0
	; seq_bitand 1 ; you can use this to check for up to 8
different advancement flags if you wanted to for various sections
	seq_beqz tsec0 ; loop when no variation is set

_start_20:
	seq_settempo 133
tsec1:
	seq_startchannel 0, tsec1_chn0
	; ... repeat for all channels
	seq_delay 9984
	seq_jump _start_20

_end:
	seq_disablechannels 0x7DFF
	seq_end
```

Adding and Removing Channels on Demand

If you want to have a dynamic melody that changes depending on where you are (like in DDD) you might want to add and remove channels dynamically. The original game does this by editing the volume of entire channels with code, but I will cover how to do it inside the sequence. The advantage of this is basically that you can manage which channels get isolated in the sequence, rather than managing data tables in your C code,

which may or may not be a tiny bit more optimal but more importantly results in much cleaner code.

```
tsec0_chn1:
      chan_largenoteson
      chan_setlayer 0, tsec0_chn1_ly0
      chan_setvol 119
      chan_setinstr 44
      chan_setval 0x80
      chan_iowriteval 4 ; the slot that will hold chan volume
      volscale_loop ; my macro
      Chan_end

.macro RAMP_VOL, slot
      @repeat:
      chan_ioreadval 4 ; represents current volume scale
      chan_bgez @inc_vol
      chan_jump @end_ramp ; end the ramp at vol 0x80
      @inc_vol:
      chan_writeseq_nextinstr 1, 1 ; write value
      chan_setval 0
      chan_iowriteval 4 ; inc vol by 1
      chan_writeseq_nextinstr 1, 1 ; write volscale arg
      chan_setvolscale 0
      chan_delay1
      chan_jump @repeat

      @end_ramp:
      chan_setval -1
      chan_iowriteval2 0, slot
      chan_end
.endmacro

.macro FADE_VOL, slot
      @repeat:
      chan_ioreadval 4 ; represents current volume scale
      chan_subtract 1
      chan_bgez @sub_vol
      chan_jump @end_fade ; end the fade at vol 0
```

```
	@sub_vol:
	chan_iowriteval 4 ; lower vol by 1
	chan_writeseq_nextinstr 0, 1 ; write volscale arg
	chan_setvolscale 0
	chan_delay1
	chan_jump @repeat

	@end_fade:
	chan_setval -1
	chan_iowriteval2 0,slot
	chan_end
.endmacro

.macro chan_IO_set, slot, target
	chan_ioreadval2 0, slot
	chan_bltz @skip ; @ creates a static label
	chan_call target
	@skip:
.endmacro

vol_ramp:
	chan_ioreadval 4 ; check if vol needs ramping
	chan_bgez @@exec_ramp ; anything from 0 to 127
	chan_setval -1
	chan_iowriteval2 0, 0
	chan_end
	@@exec_ramp:
	RAMP_VOL 0

vol_fade:
	chan_ioreadval 4 ; check if vol needs fading
	chan_subtract 1
	chan_bgez @@exec_fade ; 128 to 1
	chan_setval -1
	chan_iowriteval2 0, 1
	chan_end
	@@exec_fade:
	FADE_VOL 1
```

```
.macro volscale_loop
     @chan_loop:
     chan_IO_set 0, vol_ramp ; slot 0 to ramp
     chan_IO_set 1, vol_fade ; slot 1 to fade
     chan_delay 1 ; set freq as you want
     chan_jump @chan_loop
.endmacro
```

The setup for this effect is the same as previous examples but with two different functions on different slots. These are checked via my macro *chan_IO_set <slot>, <target>* by using different slots and different targets. The two actions are to fade the volume to 0 from max, and to max out the volume from 0. Volume will start at max which is 0x80, but we can start at zero with a variation if necessary.

The ramp and fade functions work by utilizing a channel sound script IO slot to hold the current volume. I start my functions by checking if they need to be executed, and if they do I will enter the ramp and fade loop macros. They will need execution if the volume register is not at the expected place. For ramping, it's if the volume is below max, and for fading it's if the volume is above.

These macros each start by reading the slot value. For ramp, I check if the value is non negative, I increase it by 1 and then write that value to the volscale cmd and slot. *chan_writeseq <imm>, <target>, <offset>* will write imm + value to target + offset. This allows me to increase value by one and write it to the *chan_setval* cmd. This is the only way to increase the value. I then write it to the slot and volscale cmd. I delay for one update and repeat.

After ramping, the value will be negative at 0x80, which is a volscale of just over 1.0 and our max volume. When this happens we will hit our branch condition, set the value to -1 and write it to the slot which will exit the function.

The fade function is basically the same as ramping, but instead of adding one with *chan_writeseq* I use *chan_subtract <imm>* right after reading the register. The branch conditions are when value is zero for the start of the function, and less than zero for the loop. This is because the loop subtracts before writing.

This ramping requires each channel to use its own slot to store the volume, otherwise they would get in each other's way while executing the loops, but I can use a common start condition since once the loop starts it is locked in. I have to check for above or less than zero instead of just at the bounds in case there are multiple sections, which can interrupt the fade/ramp processing whenever the section starts.

The requirement to loop at the end means I cannot have these loops in any channel with cmds throughout the sequence, which generally makes this inferior to doing the same thing with code. If you were to do this with code, I would recommend you still use MML scripting, but instead use script IO slots to flag which channels should be faded/ramped. For example if you init the slot IO channels to -128 like I did, you don't have to keep track of what channels need fading on a per sequence basis, you can check for the slot value instead.

Sequence Timing and Game Syncing

If you want to create gameplay that syncs to music then you would normally create a timer and then try to sync it to the timing of the musical beat. An example of this is beat block objects. This method works fine, but there is a possibility of the game going out of sync with the music. As with the previous example with adding/removing channels, this is mostly a technique you would use for finer and more accurate control, and to simplify code by putting the details in the sequence.

Syncing Outputs To a Beat

Syncing an output to a beat is easy, you can choose to do it with a channel output or a variation, it is up to what you decide is easier to manage from the code side.

```
_start:
      seq_setmutebhv 32
      seq_setmutescale 50
      seq_initchannels 0x0005
qwerqwer:
      seq_startchannel 0, half_chn0
      seq_setvol 108
beat_set_loop:
      seq_assign_var 0; init
      seq_delay Quarter*4 ; 4 beats of nothing
      seq_assign_var 1; beat 1
      seq_delay Quarter
      seq_assign_var 2; beat 2
      seq_delay Quarter
      seq_assign_var 3; beat 3
      seq_delay Quarter
      seq_assign_var 4; beat 4
      seq_delay Quarter
      seq_jump beat_set_loop ; repeat
```

```
      seq_disablechannels 0x0005
      seq_end

.macro seq_assign_var, val
      seq_setval val
      seq_setvariation 0
.endmacro
```

In this example I have set up my variation to match the hits necessary for a beat block, assuming that my sequence starts immediately at time zero. I set the variation to zero, and then wait 4 beats. I then increase the variation by one each beat and loop. The advantage to this over timing is that if I were to ever edit the tempo this would automatically sync, and I could also decide to change the beat structure and it would work just fine. You can do the same with channels, just set the scriptIO slot value instead.

```
half_chn0:
      chan_largenoteson
      chan_setlayer 0, half_chn0_ly0
      chan_setinstr 12
chan_beat_loop:
      chan_setval 0
      chan_iowriteval 0
      chan_delay Quarter*4
      chan_setval 1
      chan_iowriteval 0 ; beat 1
      chan_delay Quarter
      chan_setval 2
      chan_iowriteval 0 ; beat 2
      chan_delay Quarter
      chan_setval 3
      chan_iowriteval 0 ; beat 3
      chan_delay Quarter
      chan_setval 4
      chan_iowriteval 0 ; beat 4
      chan_delay Quarter
      chan_jump chan_beat_loop
```

Syncing Outputs To Data Finishing

In certain instances, you may want to test if a section or channel is finished. There are built in MML macros to do this called *seq_testchdisabled* <*chan*> and *chan_testlayerfinished* <*layer*>. Both of these will set value to 1 if the target is finished, and 0 otherwise. You will want to use this when you are not sure when the piece of channel data is going to start, for example in the case of sfx, or if you want to send outputs to time gameplay to it, like in a cutscene.

```
.poll_023589:
chan_delay1
chan_ioreadval 0 ; force stop if slot is true
chan_bltz .skip_023589
  chan_beqz .force_stop_023589
  chan_jump .start_playing_023589
.skip_023589:
chan_testlayerfinished 0 ; stop playing if layer 0 has finished
chan_beqz .poll_023589
chan_jump .main_loop_023589
.force_stop_023589:
chan_freelayer 0
chan_freelayer 1
chan_freelayer 2
chan_jump .main_loop_023589
```

Here is an excerpt from *00_sound_player.s* where we test if a layer is finished. Layer 0 in this channel is used as a signal to end the sound effect. It frees the layers and then returns to the main loop, which is where it waits for new requests to play a new sound effect. It can be forced to stop with an IO value, but otherwise it will auto stop when a layer is finished which will allow new sfx to be played.

Composing

With scripting, we were mostly worried about being able to control and respond to changes in the sequence with game code to create responsive sequences. With composing we are interested in making static sequenced music as optimal as possible, with as much control over the sound as we can get.

I covered many complex effects and sequence control in my previous tutorial, which you should definitely read if you have not (link here). The main focus of this tutorial is to

consolidate special effects into simple and easy to use events, and to create special events that could only be done by using the special abilities of MML.

Channel Effect Lerp

A lerp is a shorthand term for linear interpolation. This is when we change from one value to another over evenly paced steps. This is often done in sequences with volume, or panning to create dynamics which is a musical term that basically means music that changes.

I have gone over setting values and even ramping/fading values in this tutorial in a responsive manner, but if we are to do this for composition, it needs to exist anywhere in the sequence at any time, and needs to be added via a midi CC (or group) if possible. We also want to be able to do this without interrupting normal channel channel processing.

Simple Lerp With Manual Inputs

This is a simple effect where we will lerp from a given start value to an end value on a single effect. We can do this with a lot of normal event commands, or a bit of MML scripting. The downside of using scripting is it will block other events from executing while it happens.

```
chan0_tsec0:
chan_largenoteson
chan_setlayer 0, layer0_test
chan_setinstr 4
chan_setvol 100
chan_delay 50
chan_lerp_pan 64, 127, 30
chan_delay 600 - 30 ; subtract prev effect time, not ideal but
limitation of seq64 conversion
chan_setvol 80
...repeat for rest of chan data

.macro chan_lerp_pan, start, end, length
     step equ max( int( abs(end-start)/int(2*length) ), 1) ; calc
our step change
     len equ min( int(abs(end-start)/step), 2*length, 128) ; don't
go over 128 or min step change

     chan_setval start
```

```
     chan_writeseq 0, @pan_val, 1
     chan_loop len
     @pan_val:
     chan_setpan start
     chan_setval 0
     chan_readseq @pan_val + 1
     .if end < start
          chan_subtract abs(step)
          chan_writeseq 0, @pan_val, 1
     .else
          chan_writeseq step, @pan_val, 1
     .endif
     chan_delay Sixteenth ; you can set it how you please
     chan_loopend
     chan_setpan end
.endmacro
```

We execute this lerp by using a macro that calculates the step change and then loops the specified number of times, increasing our panning by step each loop. If the step was negative, we would use subtract instead, we can build this into the macro using the .if directive. I set the pan with a writeseq at the start in the case where you choose to loop this macro so that we always know where the pan is starting. If we don't do this, on loop we would start from the last end value instead of our start.

I coded this with the idea of creating it entirely with midi CC events, so the max input is only 127, which maxes the pan range. If we want to do longer lerps, you need to change the frequency, which is controlled by the delay. You can hardcode this or make it an extra argument to the macro.

Our lerp is almost entirely contained within the macro, but we are limited with timing on future events due to added delays seq64 cannot account for. So we need to subtract the lerp length from the next delay to get back into sync. This is annoying but is the most I could do with current tooling.

If you know the next event is going to be a delay longer than your lerp, which should always be the case if you're using this lerp, then you can code in a few extra lines to automate this at the cost of extra processing.

```
.macro chan_lerp_pan, start, end, length
... same as prev macro
```

```
chan_setval 0
chan_readseq @macro_end +1
chan_subtract len
chan_writeseq 0, @macro_end, 1
@macro_end:
.endmacro
```

We add in a *chan_writeseq* to reduce the delay by len. This will allow us to create this lerp using 3 midi CC events only, and would mean it is something we can conserve during importing and exporting files with seq64.

Continuous Lerps With Manual Composition

If we abandon the use of seq64 conversion for our lerps, we have more flexibility over what we can do. This doesn't mean we are going to write the sequence by hand, but we will have to write in the lerp event arguments by hand and will have to convert the seq64 static events to something new.

The basic idea behind continuous lerps is the same as with continuous updates. Instead of using long delays and statically placed channel events, we are going to use loops and scripting to change parameters as needed. How we do this is by converting the channel data to a new format so that multiple events can occur at the same time.

```
.macro ChanDat, start, length, type, arg, effect
.halfword start
.byte length, type, arg, effect
.endmacro
; Just a basic explanation, not full working macro which will have
tons of ifs in for each type
```

We create an array of data structs that will hold our effects and read from that instead of static channel data. How it works is that we execute the effect specified on the required data type by using dyntables to choose functions that will set the parameters we need, then we will advance to the next ChanDat struct after delay has passed.

Due to the complexity of this, I don’t really recommend ever doing this. I have a working prototype of this made, but like in the Adding and Removing Channels on Demand section, this is something better done with code or just with keeping static cmds as the processing power and sequence bloat of it is much worse than doing the same with C code, even if you have to use data tables to manage everything.

Sidechain Composition

Sidechaining is basically when you trigger events using other tracks. This is helpful to reduce the complexity of composition, and to create better sounding tracks in an automated manner.

The goal of a sidechain is to make channel management easier, so we ideally want our sidechain system to be easy. Unfortunately as we've seen with lerps, actually controlling effects continuously is difficult, and intrusive to the static composition midi generally has. In order to use sidechains, we need to continuously monitor our channel effects and be able to respond when needed. As usual, this is a big headache when events are common and not a big deal when they're sparse.

Single Channel Volume Sidechain

A common use of side chains is to use the kick to control bassline volume. These are tracks that hopefully don't have too much dynamics inside the channel. If you're composing, try to keep volume changes on the notes instead of in channels.

```
attack equ 80

register_side_kick:
     .byte 127

.macro chan_set_sidechain, vol
     ; lower volume in two steps
     chan_setval vol-int(attack/2)
     chan_writeseq 0, register_side_kick, 0
     chan_delay 4
     chan_setval  vol - attack
     chan_writeseq 0, register_side_kick, 0
     chan_delay Sixteenth - 4

     ; restore volume
     chan_setval 127
     chan_writeseq 0, register_side_kick, 0
.endmacro

.macro chan_update_volscale
     @macro_start:
     chan_setval 0
```

```
	chan_readseq register_side_kick
	chan_writeseq 0, @volscale, 1
	@volscale:
	chan_setvolscale 127
	chan_delay1
	chan_jump @macro_start
.endmacro

tsec0_chn7:
	chan_largenoteson
	chan_setlayer 0, tsec0_chn7_ly0
	chan_setpan 64
	chan_setvol 120
	chan_pitchbend 0
	chan_setinstr 24

	@side_jump:
	chan_set_sidechain 127
	chan_delay 60-Sixteenth
	chan_set_sidechain 127
	; chan_delay 12-Sixteenth ; This is zero
	chan_set_sidechain 127
	chan_delay 108-Sixteenth
	chan_set_sidechain 127
	chan_delay 36-Sixteenth
	chan_set_sidechain 127
	chan_delay 168-Sixteenth
	chan_jump @side_jump

	chan_delay 5760
	chan_end

tsec0_chn10:
	chan_largenoteson
	chan_setlayer 0, tsec0_chn10_ly0
	chan_setlayer 1, tsec0_chn10_ly1
	chan_setvol 53
	chan_setpan 106
	chan_pitchbend 0
```

```
    chan_setinstr 1
    chan_update_volscale
    chan_end
```

Here we update our bassline using channel events in the kick channel. We have to manually place these events as there is no other way to know when to lower. We can place these in the midi editor easily as they only require one argument. If we are doing a simple kick loop then we can loop the events on the layer end to save space, and seq64 is usually smart enough to loop channel data on its own anyway.

The tuning of how your sidechain affects volume should be set up in the macro *chan_set_sidehcain*. I chose to use the param as a return volume, but you can do other things such as set the attack volume. When you add these macros, make sure you set the optimizer Quantize setting to 0 for Other in seq64 or else it may remove cmds with the same argument.

The setup is quite simple compared to earlier examples. We write to a register, and then read that and use it to set the volume scale in another channel. We can add this sidechain to anything else and it should work easily. When you do this, make sure you define these macros inside the MML file instead of in the seq_macros.asm file. It will create issues if you reference a global variable inside a macro in a generic file.

Using this sidechain means you cannot have sidechain effects in the kick channel, and normal events after sidechain control in other channels. If you need these things, you will have to use the continuous event control scheme laid out in the continuous lerp section, and add a function for sidechain control and writing.

The Audio Thread

The audio thread contains all the code that updates sound processing. It runs at 60 fps and executes audio tasks on the RSP. Audio tasks are the parallel to gfx tasks if you're familiar with how graphics are generated on the N64. To make it brief, the game generates a list of microcode cmds and sends it to the RSP. This task list is made of ABI cmds, which stands for Audio Binary Interface. While I don't particularly care about how the ABI works (at least for this tutorial series), it is important to understand how we generate it.

In order to create an audio task, we need to process all the sounds currently happening in game, and synthesize them to create a list of sound data with the applicable transformations to play. The sounds from all different sources are combined in *synthesis_execute(u64 *cmdBuf, s32 *writtenCmds, s16 *aiBuf, s32 bufLen)* located in

/src/audio/synthesis.c. This function gathers these by processing our sequences, which will gather all the notes currently being played. Then all the notes will be processed, which will create the list of all sounds that need to be played and how to transform them. Finally all the sounds are synthesized together (the specifics are not that important for us) and an audio task is created.

For the purposes of control over sequences and sound, we have two main places we can access to write custom code without modifying this overall process, which is sequence processing and note processing.

Sequence Processing Editing

All of the above MML control we added via macros, we can add in with custom code to the sequence processing script. We can get finer control over ins and outs of the sequence script and consolidate multi line macros to one.

More interestingly though, we can add in new effects that are not possible with MML alone. MML is turing complete, so it technically can do everything, but I am speaking about things that cannot be executed with just one or two macros and extra data. In general, this is extra control in layers or sequence objects, and finer continuous control in MML objects that doesn't require us to rewrite the entire script like continuous channel updates do.

Sequence Dyntables

While we could use variations, bitand and branching to do the same effect, a dyntable is a more direct way of controlling sequence flow. We can add in a dyntable by basically copying the same way channels do it. We need to allocate a set dyntable, a dyncall, and dynset channel cmds in our macro file.

We need to add in the code for our cases to the sequence processing loop. We need to choose cmd bytes that are unused and after that we can basically copy what channels do for each cmd except with swapping seqChannel references to seqPlayer ones.

```
case 0xc1: // seq_dyntable
     if (value != -1) {
          seqPlayer->dynTable = (void *) (GetSeqorExtData(seqPlayer)
+ m64_read_s16(state));
     }
     break;
case 0xc2: // seq_dyncall
     if (value != -1) {
```

```
            seqData = (*seqPlayer->dynTable)[value];
            state->depth++, state->stack[state->depth - 1] =
state->pc;
            sp5A = ((seqData[0] << 8) + seqData[1]);
            state->pc = GetSeqorExtData(seqPlayer) + sp5A;
      }
      break;
case 0xc3: // seq_dynsetdyntable
      sp5A = (u16)((((*seqPlayer->dynTable)[value])[0] << 8) +
(((*seqPlayer->dynTable)[value])[1]));
      seqPlayer->dynTable = (void *) (GetSeqorExtData(seqPlayer) +
sp5A);
      break;
...
case 0x10: // seq_dynsetchannel
      if (value != -1) {
            seqData = (*seqPlayer->dynTable)[value];
            sp5A = ((seqData[0] << 8) + seqData[1]);
            sequence_channel_enable(seqPlayer, loBits,
GetSeqorExtData(seqPlayer) + sp5A);
      }
      break;
```

So with our new dyntables, we can execute more complicated sequence branching logic. This would be helpful in a sequence with many different variations that hold different channel data, or for controlling preset channel parameters like we previously did with channel dyntables.

Sequence Dyntable Examples

We can choose what section to play in our sequence using value and a dyntable. We can use variations with this to get easy control over sequence flow. Generally you would want to use this for setting channels or choosing sections of a sequence but you could also use it for preset effects such as transposition, tempo and volume.

```
tsec0:
      seq_startchannel 0, tsec0_chn0
      ...; other channels
      seq_settempo 132
```

```
	seq_setval -1
	seq_setvariation 0
	; loop for just first section, has a bit of time where nothing
is looping
	seq_setdyntable seq_dyntable

	seq_loop 6

	seq_loop 255
	seq_getvariation 0
	seq_dyncall
	seq_delay1
	seq_loopend

	seq_loopend

	seq_delay 1536-(255*6)
tsec1:
	seq_startchannel 0, tsec1_chn0
	... ; other channels
	@loop:
	seq_getvariation 0
	seq_dyncall
	seq_delay1
	seq_jump @loop
	... ; end seq object

seq_dyntable:
	sound_ref effects_0
	sound_ref effects_1
	sound_ref effects_2
	sound_ref effects_3
	sound_ref effects_4
; all the effects are similar to this, I'll skip them to save space
effects_0:
	seq_settempo 102
	seq_transpose -2
	seq_setvol 100
	seq_end
```

We use the dyntable here basically the same as we had used the dyntables earlier in the Set To Preset Value on Demand section. I have a bunch of target preset settings here that affect the entire sequence, and can select them on demand with my code.

Lerp Events

Lerps are possible in default MML, but it is much more convenient to have a dedicated cmd for it. I will cover how to create channel lerps, and it should be reasonably easy to use this for other objects if you choose.

```
struct SequenceChannel
{
...
u8 Lerp;
u8 LerpVal;
s16 LerpStep;
u16 LerpTime;
f32 LerpFloat;
}
```

I add these lerp variables to the *SequenceChannel* struct to control my lerp. I have a value and a float variable for the different types on various cmds.

```
.macro chan_lerp, type, start, length, end
      .byte 0xd5, type, start
      .halfword length
      step equ max( int( (abs(end-start)*256.0)/length ), 1) ; calc
our step change
      .if end > start
            .halfword step
      .else
            .halfword -step
      .endif
.endmacro
```

I set up my lerp macro with 4 arguments, which is similar to the last lerp macro except with an extra *type* parameter. This param will control which field I am going to lerp over in the seqChannel.

```
case 0xd5: // chan_lerp
      seqChannel->Lerp = m64_read_u8(state);
      // start
      seqChannel->LerpVal = m64_read_u8(state);
      seqChannel->LerpFloat = (f32) seqChannel->LerpVal;
      // time
      seqChannel->LerpTime = m64_read_s16(state);
      // (end - start) / time
      seqChannel->LerpStep = m64_read_s16(state);
      break;
```

I add the new cmd to my sequence channel processing function. I just read variables here, the calculation portion was relegated to the macro.

```
if (seqChannel->Lerp){
      seqChannel->LerpTime--;
      seqChannel->LerpFloat += seqChannel->LerpStep / 256.0f;
      switch(seqChannel->Lerp){
            case 1:
                  seqChannel->pan = seqChannel->LerpFloat/128.0f;
                  break;
            case 2:
                  seqChannel->volume = seqChannel->LerpFloat/128.0f;
                  break;
            case 3:
                  seqChannel->freqScale =
seqChannel->LerpFloat/128.0f;
                  break;
            case 4:
                  seqChannel->volumeScale =
seqChannel->LerpFloat/128.0f;
                  break;
      }
      if(!seqChannel->LerpTime){
            seqChannel->Lerp = 0;
      }
}
```

Finally I add the lerp calculation to my channel processing. I add this before it processes cmds. I check if I need lerp, calculate the step change, and then apply that change to the chosen parameters. Then I disable the lerp if the time is zero.

Using this is as simple as before in the Simple Lerp With Manual Inputs section. The big difference is we have less complexity and more precision with our changes.

Conclusion

This tutorial should cover basically anything you want to do with sequence scripting and probably a lot more. If you have any questions about the content or have suggestions to improve this or any other tutorial contact me in places where 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