# WARNING

**This page is under construction. Many/most links will not work.**

---

## Formatting music for FF6 with MML -- "quick" guide & reference, part 1

aka emberling's tutorial

With the recent release of Beyond Chaos EX, there have been a good few people expressing an interest in learning how to expand the library for the included music randomizer. I've been meaning to write a detailed guide for FF6 music hacking for a while now, documenting all the new tools and discoveries since I came on the scene. This is not that guide.

In the interest of getting this information out ASAP (read: ever), I'm not going to worry about explaining terminology, or filling in basics, whether about music theory, synthesis, or romhacking. Y'all are smart, you can read between the lines, or look things up, or ask in the ff6hacking or BC discord. Then maybe I'll go back and fill in the bits people had trouble with.

The process is not simple – that is, it's not a matter of just dropping a MIDI file into a converter and saying go. But if you think of it as a process of *arranging* rather than *converting* – and that is what it is, since you are adapting a song to work with a different set of instruments than it was originally designed for – then as arrangements go, it can be very simple. Mostly, there's just a lot of information to learn and then apply.

This guide will not cover hex editing or the bytecodes used in the FF6 music system. These are well documented elsewhere, and there is little to no need for hex editing in the songmaking process – though you should understand hex numbers. This guide will also focus specifically on adding music to BCX / NaO, so many topics involved in creating your own custom music hack, such as customizing the sample set, will not be covered.

---

### GETTING SET UP

Before you make any music, you need to be able to test it. You'll need three things:

- ROM with the correct sample set.
- Saves with files in the appropriate places to hear inserted music.
- A script for inserting music into the ROM.

This package contains an IPS for the BCX/NaO sample set (apply to clean unheadered 1.0 ROM), an appropriate SRM save file, and my quick insertion script. You will need Python 3.x to run the script.

Note: A Beyond Chaos EX ROM is not usable with the insertion script, because the instrument data location is moved and its location is not predictable. The package README contains instructions for how to generate a ROM with a newer version of the sample set than is included here, or with a customized sample set.

Place the script and ROM in the same directory. Name the ROM so that it will load the save file. Edit the script and adjust the ROMFILE path to point to your ROM (use forward slashes). You should now be able to drag any MML file from the custom/music folder in BCX onto the test script, and it will insert it into the ROM. The default version of the song will replace Searching For Friends. If there is an a “nat” (native instruments only) variant of the song, this will replace The Mines of Narshe. The Phantom Train slot is also used for a few purposes: if there is a “tr” variant, it appears here; otherwise, either “nat” or default will be duplicated here, depending on the “multipart_use_enh” setting in the _TEST script. The train's event tiles (at doorways of save point rooms) can be used to test songs that use the conditional jump feature.

If you need to get deeper analysis of what you've created beyond just listening to it play back in game, you'll need to use an advanced SPC player.

First, you need to rip the SPC. In snes9x, this can be done from the File menu → Save Other… → Save SPC Data. This will dump the SPC700's memory at the time it's selected, which typically means that songs will not start from the beginning. To get the whole song, you need to dump the SPC during the gap between one song ending and the next song beginning. This is why the save files are used instead of just replacing the prelude or intro musics - so that you can control and time exactly when the songs change over. With the first save file, dumping immediately as the file is loaded works reliably. Later save files have a delay before the prelude fades out; this can take some practice to get right. Making a save state on the save menu is recommended.

For analyzing the SPC, I recommend using both SNES SPC700 Player and kuronekoSPC. They largely provide the same detailed information about what's going on under the SPC hood, but each has its own specific advantages and drawbacks.

---

## GETTING STARTED WITH MML

MML (Music Macro Language) is a general term for a method of representing music digitally. It originated with the earliest PCs as a method of generating sounds in BASIC. Most SNES games, and many other console games, used a form of MML to store music data, probably because memory is at a premium in these systems and MML is very compact compared to other means of representing music, like MIDI. However, each game has its own distinct language, so generally speaking, MML for one game can't simply be transferred into another.

Final Fantasy VI's sound engine was coded by Minoru Akao, Square's long-standing main sound programmer. It uses the fourth major revision of Akao's SNES driver, or AKAOSNES4. (This song format will almost, but not quite, translate cleanly to or from many of Square's other later games, but currently getting an *actual* clean translation requires some major edits in most cases.)

While we have no idea what the original MML text files that Uematsu & co. created look like, the mfvitools MML format provides full access to all the game's music functions as well as quite a few convenience features. The MML commands map one-to-one with AKAOSNES4 bytecode commands, so songs can be converted to MML and back to binary data losslessly.

(The mfvitools MML command set was based on, and intended to be largely compatible with, the MML used in rs3extool for Romancing SaGa 3, which was previously used by FF6 hackers as well despite some key differences in the music engines of the game.)

---

Before jumping in to full songs, let's look at the structure of a very simple MML:

```
#WAVE 0x20 0x8B
{1}
t120 %v50 %b0,70 %f0,0
$ |0 v64 p64 %e1 o5
cdefgab<c> l16 bb-aa-g8g-8f8ee-dd-c8
;
```

```
#WAVE 0x20 0x8B
```

Loads sample id 8B into program (slot) 20. The slots for samples are 20 through 2F (16 total max). Sample 8B in the BCX/NaO library is a piano suitable for use in the octave 5 range.

```
{1}
```

This sets channel 1 to begin reading instructions at this point.

```
t120 %x224 %v50 %b0,70 %f0,0
```

This is a set of global instructions that apply to all channels, and are typically found at the beginning of songs to set up the environment.

- `t120` - sets tempo to 120 beats per minute. This is the only global instruction that is strictly required at the start, as without a tempo set, playback will freeze.
- `%x224` - sets global volume to 224 out of 255
- `%v50` - sets echo volume to 25 out of 63
- `%b0,70` - sets echo feedback to 70 out of 127
- `%f0,0` - sets echo filter to default

Echo settings should be set at the beginning of each song to avoid unpredictable results, as they can carry over between songs. These values are used in many FF6 native songs, and I find that they make a good starting point.

```
$ |0 v64 p64 %e1 o5
```

Some settings to initialize the channel so it produces some useful sound.

- $ - This sets the point where the channel will resume after looping.
- `|0` - Sets current instrument to sample 0 (i.e., program 0x20)
- v64 - Sets volume to 64/127
- p64 - Sets pan to 64/127 (center)
- `%e1` - Enable echo for this channel
- o5 - Set octave to 5

```
cdefgab<c> l16 bb-aa-g8g-8f8ee-dd-c8
```

Notes representing a C major scale upward, then a chromatic scale with some rhythmic variation downward.

- Letters from A to G are interpreted as notes. R is a rest (silence), while ^ ("tie") continues the previous note seamlessly (used for fine control of note lengths).
- At the beginning of any segment, the default note length is 8. The l command changes this. Available lengths are 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, and 64. (These are the standard fractional lengths of music notation – so one 1 lasts the same time as four 4s or twelve 12s, and one 4 lasts the same time as three 12s.)
- < and > raise and lower the octave, respectively. (Note: some MML variants have these oriented in the opposite direction instead. I think of these as greater-than/less-than symbols, or as sides of a hill on a topographical map. The alternate orientation's core metaphor - as arrows that point a direction on a piano keyboard - I feel is less effective, but if you're already used to it, you can include the line "#REPLACE <> ><" to invert them.)
- Following a note with + or - will shift the tone by one semitone, allowing access to sharps/flats. (Note: Since the notes are recorded without specifying any octave, "c-" will jump up to the B above, at the highest end of the current octave, and "b+" will jump down to the C at the beginning of the octave. These can't be used to avoid explicit octave changes.)
- Following a note with a valid number will use that number as the duration of the note. Here, in the second measure, we have a default length of 16, but some notes are marked as 8, so those notes are held for twice the duration.
- Following a note duration with a period will extend the note by half its stated length. This can be done any number of times, with each addition half the size of the last. This also works on default length notes. Examples:
    - `l4 c.d.e` – produces a measure with two notes of 3/8 length and one of 1/4.
    - `c2.d4` – produces a measure with one note of 3/4 length and one of 1/4.
    - `c2..d8` – produces a measure with one note of 7/8 length and one of 1/8.

Finally, the ';' ends the current segment, either by jumping back to the most recent '$' within the segment, or by stopping playback permanently if no '$' is found.

Most songs consist of eight (or sometimes fewer) segments, beginning with a channel pointer ({1} - {8}), containing a loop point ($) somewhere, and ending with ;. Remember, note length and loop point commands only apply within their segment. With some exceptions in cases of non-standard layouts, this allows us to treat each channel's sequence as completely independent from all others.

Note that between commands, whitespace and linebreaks are completely ignored, except in preprocessor directives and comments. Nothing is case sensitive, except the #REPLACE, #WAVE, #SFXV, #VARIANT directives due to an oversight which will be corrected in future releases. Anything in a line following # is a comment, unless the line is a preprocessor directive (#WAVE, #REPLACE, #def etc, at the beginning of a line). To distinguish comments from directives in notepad++'s syntax highlighting, I use a convention of two # for comments, but this is not required. Inline comments are also available, by placing the comment between braces {like so} – aside from the numbers 1 through 16, which will be interpreted as channel markers regardless of what else is in the braces. Another unofficial means of producing the effect of an inline comment is to use single quotes – as long as the contents of the quotes don't match a macro name, it will be discarded like a comment.

## PUTTING THE MACRO IN MUSIC MACRO LANGUAGE

Let's look at this MML next.

```
#WAVE 0x20 0x8B
#def global= t120 %v50 %b0,70 %f0,0
#def piano= |0 v64 p64 %e1 o5

{1}
'global'
$ 'piano'
cdefgab<c> l16 bb-aa-g8g-8f8ee-dd-c8
;
```

You may have guessed that this produces exactly the same output as the previous example. The #def directive defines a macro; the name of the macro, in single quotes, can then be used to invoke the commands and/or notes assigned to the macro. These can be used simply to make things easier to read, but their most important use is to allow us to edit in only one place to adjust a pattern that is used in more than one place. Specifically, I recommend using macros to set instruments, because it drastically simplifies one of the most important steps in the whole process: selecting and perfecting the various timbres and instruments, and mixing them appropriately. I also recommend including an absolute octave (e.g. o5) with these timbre macros, and using only relative octaves outside them, because you will often need to test several different samples for a part, and many samples are transposed up or down one or more octaves due to limitations in the SPC700's pitch range.

Note that macros may be nested, e.g.: `#def pianoloud= 'piano' v96`

Also, macro definitions and all similar preprocessor-style declarations, including #WAVE, apply to the entire file, no matter where they are located in the file. If there is a conflict, the last one takes precedence.

## LOOPS AND FLOW CONTROL

`[4 cdefgab<c>]` - Simple loop. The sequence enclosed in square brackets is played four times.

`[cdefgab<c>]` - If the number of times to loop is not specified, it will play twice.

`[[cde]fd]` - Loops can be nested inside each other up to 4 deep. This will play 'cdecdefd' twice.

`[3 cd j3 e]` - The jX command will jump out of the current loop if the loop counter is X or more. Flow will continue from after the next ] command, and the loop will no longer be active. This example will play 'cdecdecd'.

`[2 cd j2,1 [ef] ] $1` - Note that if a nested loop occurs after the jX command but before the regular end of the loop, the “next ]” shortcut used by jX will fail, and the sequence will desync and probably cut off early (the end loop command will stop playback for the channel permanently if no loop is currently active). But the jX command is actually just a shortcut for the jX,Y command, which allows you to control where playback will resume after exiting the loop. $X will set up a jump target with id X. These ids must be unique throughout the MML file. jX,Y will exit the current loop at X iterations and continue playback at the jump target labeled Y.

`$1 cdefgab<c> ;1` - ;X is a “hard jump”, or unconditional jump. The song will continue from the specified jump target. This is the same underlying command as the $ / ; infinite loop discussed previously, but allows for more control. For the most part, you'll only see this in files converted from binary to MML, since mfvi2mml does not distinguish between these use cases of the hard jump command. This can also be used to have two channels share one segment, though this is irreversible aside from a very odd corner case that I will explain later.

## MML AND BYTECODE

Armed with the ability to use some sequence data more than once, possibly with some variables in different states, it becomes important to understand which MML elements are actual commands that will be executed whenever encountered and which are just syntax and shortcuts. This also becomes important when optimizing sequence size in memory.

- Macros are simply shortcuts for the commands stored inside the macro. The concept of macros does not exist in the game's bytecode (though there are traces that indicate that a similar scheme of macros was probably used in the development process).
- “Default length” does not exist in bytecode. Every note has an included length. Therefore, something like `l8 [cd l16 efga]` will play the C and D as eighth notes in both iterations. This is in contrast with the usual loop behavior; for example, `|0 [cd |1 ef]` will play two notes with program 0, then six with program 1.
- The only dotted notes that exist in bytecode are the dotted fourth and dotted eighth. For rendering dotted notes into bytecode, mml2mfvi will use these when possible, and otherwise use ties to extend the note.
- The markers that begin channels do not exist in the bytecode sequence. They are stored in the song data header instead. There are 16 of these total; the first eight represent the “normal” song, while the second eight allow for an alternate starting point. As far as I know, the alternate starting point functionality is never actually used. (In the OST, “Awakening” and “Fanatics” have alternate starting points, but I found no sign of these being called in the event code, or even a command capable of calling them.)
- Loop points and jump targets are not stored in the sequence directly. Instead, jump commands in bytecode specify an address to jump to.
- Anything not recognized by the MML parser, including comments, is simply ignored and has no effect on the generated bytecode.
- For the most part, you can assume that any command takes one byte plus one more for each parameter. Channel volume (vX) is two bytes, channel volume fade (vX,Y) is three bytes, vibrato (mX,Y,Z) is four bytes, and so on. Toggles, like echo on (%e1), are one byte. Jumps use one extra byte. Note that the loop twice ([) and jump to next ] (jX) commands are shortcuts for more complex commands, so they actually use two and four bytes respectively.

## CHANNELS

SPC700 has eight channels, all with identical capabilities. This means we get up to eight sounds at a time and never any more. There is a lot of variation in how SNES games handle sound, but specifically for FF6 (and all other AKAOSNES variants):

Polyphony is module-like, not MIDI-like - in other words, any time we play a note or a rest in a channel, any other sound in that channel will immediately stop; and the channels in the sequence map exactly onto the eight available voices, with no ability to use multiple voices for one channel and no way to attempt to play more notes at once than are actually available.

All eight channels are used for music, which means that any sound effects will partially interrupt the music. The interrupted channel will remain synced and will not miss any commands, but it will only resume making sound at the first BEGINNING of a note it encounters. This can be problematic for very long notes; in the case of the wind sounds on Dark World, if interrupted by a sound effect, they will never resume, because the notes used for this never end or restart. Channel use must be planned around the possibility of interrupts from sound effects. Generally speaking, sound effects start from channel 8 and move down. Rapid menuing in the field can easily produce some modest interrupts of channel 6, as well as major interrupts of channels 7 and 8. In battle conditions, scrolling through the menu during an effect like Ultima or Merton will occasionally even interrupt channel 5. I haven't found any situations in which channel 4 or below appears to be affected; I suspect that the sound effect code doesn't allow this, though I haven't been able to confirm.

In arranging songs to best protect them from channel interruptions, I originally tried to minimize the disruption by keeping longer notes in safe channels and shorter notes (often percussion) in the sacrificial channels. This works well in the most common scenarios - occasionally a sound effect happens, occasionally a menu makes noise, but there is little overlapping and it ends quickly. However, in more extreme situations like menuing during Ultima, the complete loss of percussion and some melody was very distracting.

My current strategy seems to work much better in these cases. Essentially, I try to imagine that I'm creating a demake for NES, with only four channels - a melody, a countermelody or harmony, a bassline or other low foundation, and a percussion track. What has to go in there to capture the essence of the song? What can't be lost? The answer gives me my first four channels. Sometimes I can't answer it in four, so I extend it to five. But you should be able to play the song with channels 6 through 8 muted and have it still sound like a half-decent rendition.

## LIMITS

- Total sample memory is limited to 3746 blocks (33,714 bytes - a BRR block is 9 bytes). Exceeding this amount will write the excess samples into the echo buffer (echo work memory), which causes a distinctive pop at the start of the song. Samples that overflow will be overwritten by echo data, and so will make essentially random sounds when played. [This spreadsheet] will help plan out your sample use to avoid this problem. For examples of this glitch in action, just play T-Edition. Some inaccurate emulators use hacks to avoid this issue, and the Japanese SPC players allow you to configure whether to use accurate or compatible echo writes.
- Total sequence data is limited to $1000 (4096) bytes. Exceeding this amount will overflow into the area used to store sound effects. This seems to cause dramatic corruption to any sound effects that play during an overly-long song, even more than one might expect based on the cause. However, if you can be sure that no sound effects will be played during a song, it is safe to make it much longer. In vanilla FF6, both ending tracks are over $1000. To check the length of your song, run mml2mfvi on it directly (without invoking the _TEST script).
- The highest pitch a sample can be played at is two octaves above its base pitch. (16384 vxPitch, visible in knSPC on the “ADSR/BRR” page). Going above this will play apparently arbitrary and untuned tones. Vibrato can trigger this, creating an odd (maybe exploitable?)

gating effect.
- There is a short gap between adjacent notes, as the ends of notes are quickly faded out rather than simply stopping in place, in order to avoid pop artifacts. This makes for a sometimes dramatic loss of quality and/or volume with very fast notes, rendering tracker-style arpeggios largely impractical.
- There is no way to actively keep the channels synchronized. They are fully independent, and there is no way to specify an absolute time for an event. You must simply be very careful to keep track of each channel's timing to avoid desyncing.

TO BE CONTINUED … (will be covered in part 2)

- discussion of timbral effects (ADSR, vibrato)
- advanced timing (ticks, &)
- “percussion mode”
- brief overview of transcription sources
- other tips and tricks (wacky jumps)
- embedded SFX (ruin, train, zozo)
- tierboss

REFERENCES THEORETICALLY UNDER CONSTRUCTION:

- detailed description of all MML commands
- description of all samples
- samples of all samples "in action"