
SBF - Sound Block Format

Volume channels:

For correct initialization, the player should set all volumes to 15 (mute).

VC - C = number of frames till next update, 0-15 (0=next frame)
V = volume to apply

Normal 8-bit RLE/string compression occurs on volume stream, so long steady volumes should still compress well. No timestream is needed and the 4 unused bits are now used.

Tone and Noise channels:

For correct initialization, the player should set all tone channels (and the noise type) to ‘1’. If the song ever intentionally uses ‘1’ as the first note, the data may not reflect the load.

1 timestream channel is kept for this protocol. It should be far less busy since it no longer has to consider volume changes. Arpeggio will still keep it busy, sadly, but should still compress well. (There are actually RLE types to consider 2, 3, and 4 tone arpeggio).

TC - C = number of frames till update, 0-15 (0=next frame)
T = channels to update, one bit each. 0 is legal if there are no updates.

The timestream byte contains the channels to load, and then the number of frames to delay (0-F = 1-16 frames)

0x80 - tone 1, 0x40 - tone 2, 0x20 - tone3, 0x10 - noise

Note that the first timestream byte indicates which channels are actually active in the song.

Channels use the same 256 entry note table as previously, so one byte per note. Noise still uses the actual noise command. Putting noise here rather than making it the same as volume channels costs a little more, but allows us to preserve the noise trigger and not reload the noise command when we aren't supposed to.

Tone table is stored as two bytes per entry, in the format used by the configured sound chip.

- For PSG, the first byte contains the least significant nibble (with the most significant nibble blank so that the command nibble can be ORd in), this is the first byte to write. The second byte contains the most significant 6 bits, which is the second byte to write.
- For AY, the first byte contains the least significant byte for the first (fine tune) register, and the second byte contains the most significant nibble for the second (coarse) register.

----

9 Streams:
Tone1 Tone2 Tone3 Noise Vol1 Vol2 Vol3 Vol4 Timing

Tone - byte index into note table
Vol - *low* nibble - frames till next byte, *high* nibble - volume
Noise - low nibble - noise type
Timing - *low* nibble - frames till next byte, *high* nibble - channels to update (1234)

AY notes:
Noise channel has 5 bits of frequency rather than a type
Vol4 is now a mixer command. I had considered pre-shifting it up, but it only requires 1 extra SLA on the Z80 (2 cycles) and that's worth avoiding the confusion. So the data is the same - high nibble is mixer command, low nibble is count:

- This means CBAN#### - the mixer needs 00CBAN00 - so shift right twice then AND #$3C
- 'N' is a single bit of tone control - we use this to map noise when it's possible to tone channel C (this may cause shifting of tones and is better externally processed)
- Remember that 0 means ON - this means that the first two tone channels are always on.

Based on the notes from the old format. TI and AY files are the same except as discussed above with respect to noise.

Header

The file begins with two 16-bit indexes:

0000 - offset to the song stream pointer table. Each table is 18 bytes in length.
0002 - offset to the note lookup table - up to 512 bytes long.

Tables are stored at the end of the packed data, and must be sequential.

To determine how many songs are packed in the file, subtract the offset of the song stream pointer table (at >0000) from the note table offset (at >0002), and divide by 18. The note table is technically unterminated, but cannot exceed 512 bytes.

Song Stream Pointer Table

For each song in the container, contains 9 stream pointers.

- Four pointers to frequency streams (8-bit).
- Four pointers to volume streams (8-bit).
- One pointer to frequency timestream (8-bit).

* For the AY, the noise volume stream is instead a mixer stream - contains the actual register data for R7 (with the I/O ports set to 0 and all three tone channels enabled). This allows it to move the noise output to the closest matching tone channel. The noise frequency also has a larger range (5 bits vs 4), but then so do the tone frequencies (12 bits vs 10).

To reach songs after the first, add 18 bytes for each song desired.

Format of a Song

Each song (which can also be a sound effect) consists of 9 compressed streams, each compressed in the same manner. For each of the sound chip's four voices, there are two streams: a tone stream and a volume stream (mixer stream for AY noise). The decompression indicates when a stream is finished, meaning that it has no more data to provide. When all streams are finished, the song is over.

You have to know in advance if your stream is for the PSG or the AY. Though they are close, they are not interchangeable.

When updating volume, the returned byte will contain the actual attenuation value in the most significant nibble, and the least significant nibble will contain the number of frames to delay before loading the next byte. (0 = load on next frame). For the TI, you must mask out the delay and OR in the appropriate command bits to send to the sound chip.

For the AY, the most significant nibble contains the mixer bits for register 7 (BBBN), and must be masked and shifted right 2 positions. All remaining bits should be left at zero.

When updating frequency on voices 1-3, the byte extracted will be an index into a tone lookup table. Pull the appropriate 16-bit word, OR the command code into the MSB, and send to the sound chip, MSB first. Note that the PSG has a range of only 0x000-0x3FF, while the AY range is 0x000-0xFFF.

When updating frequency on voice 4, the noise channel, the returned byte the actual noise command in the least significant nibble. OR in the command code, and send it to the sound chip. For the AY, the least significant 5 bits contain the actual noise shift rate for register 6.

Format of a Compressed Stream

Streams are compressed 8 bits at a time.

All streams are compressed in the same manner, using a combination of run-length encoding and string back-references. As no decompression buffer is used, the back-references refer to the *compressed* data, rather than the decompressed data.

The stream consists of a sequence of variable-length blocks, each starting with a control byte that identifies the type and length of the block.

The control byte has the following format:

```
80 40 20  10 08 04 02 01
   |      length of data
   +-- control bits*
```

* Bit 0x20 is part of the length for non-RLE types.

The control bits define the following types of data:

00x - inline run of data - take bytes directly from this point in the stream. The 6 least significant bits are the length value. Add one to length (so 0 means take 1 byte - this is the only way to get only 1 byte of data embedded). Range is thus 1-64 bytes.

010 - RLE - the next byte is repeated 'length of data' times. Add three to length (so 0 means repeat three times). Range is thus 3-34 times.

011 - 32-bit RLE - the next four bytes are repeated 'length of data' times (big endian order, MSB first). Add two to length (so 0 means repeat twice). Range is thus 2-33 times.

100 - 16-bit RLE - the next two bytes are repeated 'length of data' times (big endian order, MSB first). Add two to length (so 0 means repeat twice). Range is thus 2-33 times.

101 - 24-bit RLE - the next three bytes are repeated 'length of data' times (big endian order, MSB first). Add two to length (so 0 means repeat twice). Range is thus 2-33 times.

11x - Back reference - the next two bytes are the offset WITHIN THE ENTIRE FILE. The 6 least significant bits are the length value. Add four to length (so 0 means take 4 bytes). Range is thus 4-67 bytes. However, a reference offset of 0x0000 means end of stream.

The decompression of a stream is independent of the playback of the data within it.

PSG file format

PSG files are data formatted for the TI SN or the AY-8910 sound chip. Much of the metadata is contained in the filename when output by the xxx2PSG converters, and in that case, each file contains just one channel.

xxxxx_AAABB.CCCC

- AAA is the data type tag:
  - _noi - PSG noise channel with shift rate same as tone channel (so needs mapping for PSG noise rates). See below for additional flags.
  - _ton - Tone channel - shift rate for default PSG shifts.
- BB is the channel number, 0 based and 0 padded. This allows up to 100 channels, which is frankly ridiculous. ;) (00-99)
- CCCC is the playback rate, and may be 60hz, 50hz, 30hz or 25hz

Ex: mysong_ton01.60hz is the second channel of the tune (01), it's tone data (_ton) at 60hz (.60hz)

The contents of the file are ASCII formatted hexadecimal values with a preceding "0x", padded to 8 characters, using native line endings. (Convert using your local tools if needed). The data is followed by a comma, and then the volume padded to two characters. Each line represents one instant of time at the rate specified in the filename.
Ex: 0x0000013F,0xF0

The frequency data is always in PSG shift counts (even for noise), and the volume is always linear from 0-255, where 0 is mute, and 255 is maximum. These will require conversion back to a real PSG, of course.

The noise channel has *additional flags* used to ensure correct playback. The lower 16-bits are reserved for the shift rate (normally no more than 1023 on a TI chip). Three flags are used for mapping it to the correct type:

0x0001xxxx - retrigger flag - the sound has been retriggered by reloading the type register. It's necessary to honor this flag for correct sound, and only reload the noise type register on frames it is set.
0x0010xxxx - periodic - the sound is a periodic noise. When not set the sound is white noise. Note that this is the inverse of the TI PSG, which sets a bit for white noise (types 4-7) and clears it for periodic (types 0-3). But white noise is the default through this system.
0x0100xxxx - channel 3 mapped - this is usually not important to the next stage because the shift rate is already provided, but the shift rate was extracted from channel 3 and it's expected to play back that way. *(This flag might not be exported and should not be relied upon)*

A song consists of up to 100 channels, which is very silly for this chip. Most tunes will have up to 4 (the limit of the real chip), but when converted from other protocols, more channels may be created.

For the sake of conversion, here is the volume table used for conversion from PSG to linear - it may also be used in reverse. The original source was a document from SMS Power:

```
unsigned char volumeTable[16] = {
	254,202,160,128,100,80,64,
	50,
	40,32,24,20,16,12,10,0
};
```

Likewise, here is the shift table for the fixed shift rates on the PSG, and can be used to convert fixed shift rates back to noise types:

```
static const int noiseTable[3] = { 16, 32, 64 };
// type 0, 1, 2 (periodic) or type 4, 5, 6 (white)
```

To edit a PSG file, you can use a normal text editor. Note that all files in the song must have the same number of lines, and if you insert or remove lines from one file - make sure to make the same change *at the same place* in the other files, too. Otherwise the channels will be out of sync.

When changing values, note that for any particular line both values must be hexadecimal (specified by 0x), or decimal. You cannot mix hex and decimal on the same line.

The channel numbers in the filenames are arbitrary, with the only restriction being that they are unique. Therefore, you can rename them to renumber them without consequence. This is useful if you need to extract multiple chip types from a single file, for instance. Knowing that each chip has only four voices, you can extract the first chip, rename the output files out of the way (for instance, rename the channel numbers from 00,01,02,03 to 10,11,12,13). Then you can safely extract the second chip without fear of overwriting. The tools can all handle any indexes from 00-99.

A very similar format is used when using the "prepare4sn" or "prepare4ay" tools, except these tools will emit all four channels in a single row. The filename does not require metadata in this case.

Tools

Test tools

Vgm_id - identify the known sound chips in a VGM
testPlayer - play a PSG or AY converted song without channel limits
psg2vgm - convert PSG data back to a VGM
analyzeStream - dump contents of a single stream from an SBF packed file

Conversion tools

vgm_psg2psg - extract PSG audio data from a VGM and export to PSG format (dual chip ok)
vgm_ay2psg - extract AY audio data from a VGM and export to PSG format (dual chip ok)
vgm_gb2psg - extract Gameboy data from a VGM and export to PSG format (dual chip ok)
vgm_pokey2psg - extract Pokey data from a VGM and export to PSG format (dual chip ok)
vgm_nes2psg - extract NES APU data from a VGM and export to PSG format (dual chip ok)

Processing tools

prepare4sn - reads 4 PSG channel files and enforces limits appropriate to the SN PSG, outputting a combined file.
prepare4ay - reads 4 PSG channel files and enforces limits appropriate to the AY PSG, outputting a combined file.
vgmcomp2 - compresses a prepared PSG file into a final SBF file.

----

vgm_id <filename>

Examines a VGM or VGZ file and reports whether there are any supported chips inside, and any warnings that might adversely affect conversion of the tune to PSG.

The chip detection lines always start with “* Detected” to aid use in scripts.

```
D:\>vgm_id.exe ab_psg.vgz
VGM_ID - v03082020

Reading ab_psg.vgz - 2512 bytes
Signature not detected.. trying gzip (vgz)
Decompressed to 16293 bytes

Reading version 0x101
Refresh rate 60 Hz
```

```
* Detected PSG with clock of 3579540Hz
  Shift register size: 16


** DONE **
```

----

testPlayer [-ay|-sn] [-sbfsong x] [-hidenotes] [-heatmap] [<file prefix> | <file.sbf> | <track1> <track2> …]

Test plays the output of the xxx2psg conversion step on a simulated SN PSG. There are no channel limits (up to the 100 channel limit of the protocol), so that this tool can be used to verify the decode steps worked correctly.

There are two ways to select files. If passed a prefix, it will locate all of the _tonXX and _noiXX files and load them in, then play them back. If passed an SBF file, it will unpack and load all four channels.

Alternately, you may provide a list of files to explicitly load and play. These may be individual channels (with a 60hz, 50hz, 30hz or 25hz extension), or PSG files (from prepare4PSG) specifying four files pre-formatted for the TI PSG. You may also mix and match up to the channel limit (except for SBF files). (TODO: the non-60hz files are not yet supported).

The data is normally played as recorded, without limits. If you need to test against your target sound chip (AY or PSG), pass the “-ay” or “-sn” switches, and the data will be verified before playback. It does not need to have been prepared, but the restrictions for that chip will be verified. Note that these are normally optional, however, if you wish to import a finished SBF file you MUST specify which chip it is meant for. This is because the noise volume is encoded differently. It’s intended you would load and play only a single SBF file at a time - for general testing load the independent files.

Timing is an estimate, and the actual playback speed may not be reflected depending on your system’s timing accuracy. Just basic sleeps are used which may not work well on some systems. The FPS is displayed while playing so you can determine if it’s playing slow - if it doesn’t hit the target rate (60fps right now), then expect it to sound slower than hardware.

In addition, due to using DirectSound, this tool is probably the only one that is Windows centric. Mind, it is probably straight forward to port, with just a few calls for the streaming buffer.

Supported options:
-ay - force AY8910 restrictions on playback. Out of range notes are not allowed and noise must share a tone channel’s volume or import will fail.
-sn - force SN PSG restrictions on playback. Out of range notes are not allowed and the noise channel must match a fixed rate or the rate of tone channel 3, or import will fail.
-sbfsong x - play song ‘x’ from the SBF file instead of song 0
-hidenotes - do not display the note and volume on each channel as it is playing. This sometimes interferes with playback and makes it a little slower, if you have trouble, don’t do it. ;)
-heatmap - graphically displays a heatmap of the relative file offset each note is read from while playing. This is only useful for the SBF import, though other types may try to show something,

it’s just going to advance. Each time a byte in the compressed file is read, it is indicated relatively on the screen with an ‘O’ character, which will fade out over about a quarter second.

Note that all files must be the same number of lines - variations will cause problems at the end of the song.

```
D:\>testPlayer.exe VampPSGAY.vgz
Working with prefix 'VampPSGAY.vgz'... Found extension 60hz
Reading TONE channel 0... read 1842 lines
Reading TONE channel 10... read 1842 lines
Reading TONE channel 11... read 1842 lines
Reading NOISE channel 12... read 1842 lines
.. and playing...
```

----

psg2vgm [-ay|-sn] [-sbfsong x] (<file prefix> | <file.sbf> | <track1> <track2> ...) <outputfile.vgm>

Converts tracks back to VGM for test purposes. This takes in the same inputs as testPlayer above, but limits the results to a single chip. If not specified as -ay, it will be treated as SN data.

No attempt is made to compress or optimize the VGM, you will get every row. As a result, this file is guaranteed to be rather large.

```
D:\work\TI\vgmcomp2\test>..\release\psg2vgm.exe sil_nes.vgz.psg
sil_nes_test.vgm
VGMComp VGM Test Output - v20200602
Reading TONE channel 0 from sil_nes.vgz.psg... read 4692 lines
Reading TONE channel 1 from sil_nes.vgz.psg... read 4692 lines
Reading TONE channel 2 from sil_nes.vgz.psg... read 4692 lines
Reading NOISE channel 3 from sil_nes.vgz.psg... read 4692 lines
Going to write VGM file 'sil_nes_test.vgm'
** DONE **
```

AnalyzeStream <name.sbf> <stream index (0-based)> [-old]

Dumps the encoding of a single stream for manual analysis. If you have multiple songs, add 9 to the stream count for each song after the first.

Options
-old - You can dump the contents of a vgmcomp version 1 stream with this switch. If you have multiple songs, add 12 instead of 9 for each subsequent song.

D:\work\TI\vgmcomp2\test>..\release\AnalyzeStream.exe sil_nes.vgz.sbf
VGMComp2 Stream Analysis Tool - v20200615

AnalyzeStream <name.sbf> <stream index (0-based)> [-old]

Pass optional switch "-old" to analyze an old v1 stream
Note that if you have multiple songs, you need to manually
specify a higher stream index (9 streams per song for new)

```
D:>AnalyzeStream.exe sil_nes.vgz.sbf 1
VGMComp2 Stream Analysis Tool - v20200615

Processing stream 1
Tone stream...
INLINE - 53 bytes: 01 00 06 07 08 09 0A 0B 06 07 ...
BACKREF- 12 bytes: 1E 20 22 20 1E 24 26 28 26 24 ...
BACKREF- 41 bytes: 06 07 08 2C 08 07 2E 30 32 30 ...
BACKREF- 12 bytes: 07 08 06 07 08 09 0A 0B 06 07 ...
BACKREF- 30 bytes: 0D 0E 0F 10 11 10 0F 12 13 14 ...
…
BACKREF- 10 bytes: 62 63 62 61 64 65 66 65 64 61
BACKREF- 10 bytes: 62 63 62 61 64 65 66 65 64 61
BACKREF- 10 bytes: 62 63 62 61 64 65 66 65 64 61
BACKREF- 10 bytes: 62 63 62 61 64 65 66 65 64 61
INLINE - 44 bytes: 67 68 69 6A 6B 6C 6D 6E 6F 70 ...
BACKREF- 4  bytes: --END--
```

----

vgm_psg2psg [-q] [-d] [-o <n>] [-add <n>] [-notunenoise] [-noscalefreq] [-ignoreweird] <filename>

Extracts PSG channels (ie: TI SN7489 and variants) from a VGM or VGZ file, dumping the individual tracks in PSG format. Dual chip is supported. The following flags are recognized:

-q - quieter verbose data - less output data
-d - enable parser debug output - every chip command and frame tick is output
-o <n> - output single channel 'n' - used to test a single channel
-add <n> - add a fixed offset to the channel output index (helpful for mixing multiple chips)
-notunenoise - Do not retune for noise - this returns the noise frequencies if the noise scaler is detected to be 16 bits wide instead of 15 bits like the TI version.
-noscalefreq - do not apply frequency scaling - if a non-NTSC clock (or close to it) is detected, tones are normally rescaled to the NTSC clock range. This skips that step.
-ignoreweird - ignore anything else unexpected - for instance, a 17-bit shift register or 70hz clock. Ignored values are treated as if they were 0 and set to default. This is not always correct! Sometimes you need to edit the source file.

```
D:\>vgm_psg2psg.exe ab_psg.vgz
Import VGM PSG - v03082020
Reading ab_psg.vgz - 2512 bytes
Signature not detected.. trying gzip (vgz)
Decompressed to 16293 bytes
Reading version 0x101
Refresh rate 60 Hz
Selecting 16-bit shift register.
```

```
File 1 parsed! Processed 5042 ticks (84.033333 seconds)
Adapting user-defined shift rates...0 notes tuned.
-Writing channel 0 as ab_psg.vgz_ton00.60hz...
-Writing channel 2 as ab_psg.vgz_ton01.60hz...
-Writing channel 4 as ab_psg.vgz_ton02.60hz...
-Writing channel 6 as ab_psg.vgz_noi03.60hz...
Skipping channel 8 - no data
Skipping channel 10 - no data
Skipping channel 12 - no data
Skipping channel 14 - no data
done vgm_psg2psg.
```

----

vgm_ay2psg [-q] [-d] [-o <n>] [-add <n>] [-noscalefreq] [-ignoreweird] <filename>

Extracts AY channels (AY-3-8910 and variants, including YM2149) from a VGM or VGZ file, dumping the individual tracks in PSG format. Dual chip is supported and envelopes are emulated. The following flags are recognized:

-q - quieter verbose data - less output data
-d - enable parser debug output - every chip command and frame tick is output
-o <n> - output single channel 'n' - used to test a single channel
-add <n> - add a fixed offset to the channel output index (helpful for mixing multiple chips)
-noscalefreq - do not apply frequency scaling - if a non-NTSC clock (or close to it) is detected, tones are normally rescaled to the NTSC clock range. This skips that step.
-ignoreweird - ignore anything else unexpected - for instance, a 70hz clock. Ignored values are treated as if they were 0 and set to default. This is not always correct! Sometimes you need to edit the source file.

Note that the vgmcomp system is a 60hz frame-based playback, and AY envelopes can cycle much faster than this. A warning will be emitted if envelopes are fast enough to be unlikely to reproduce well, but at the moment editing the source file (or output file) is the only way to change it.

```
D:\>vgm_ay2psg.exe tp_ay.vgz
Import AY PSG - v03082020
Reading tp_ay.vgz - 917 bytes
Signature not detected.. trying gzip (vgz)
Decompressed to 3570 bytes
Reading version 0x151
Dual AY output specified (we shall see!)
Subtype is AY8910...
Non-zero AY flags ignored
Warning: rate set to zero - treating as 60hz
Refresh rate 60 Hz
Selecting 16-bit shift register.
reading offset from file, got 0x4C
file data offset: 0x80
Warning: Delay time loses precision (total 851, remainder 116
samples).
```

```
Warning: fine timing lost.
File 1 parsed! Processed 488 ticks (8.133333 seconds)
Adapting noise shift rates...
-Writing channel 0 as tp_ay.vgz_ton00.60hz...
-Writing channel 2 as tp_ay.vgz_ton01.60hz...
-Writing channel 4 as tp_ay.vgz_ton02.60hz...
Skipping channel 6 - no data
-Writing channel 8 as tp_ay.vgz_ton03.60hz...
Skipping channel 10 - no data
Skipping channel 12 - no data
Skipping channel 14 - no data
done vgm_ay2psg.
```

----

vgm_gb2psg [-q] [-d] [-o <n>] [-add <n>] [-wavenoise|-wavenone] [-enable7bitnoise] [-ignoreweird] <filename>

Extracts Gameboy DMG channels from a VGM or VGZ file, dumping the individual tracks in PSG format. Dual chip is supported. The following flags are recognized:

-q - quieter verbose data - less output data
-d - enable parser debug output - every chip command and frame tick is output
-o <n> - output single channel 'n' - used to test a single channel
-add <n> - add a fixed offset to the channel output index (helpful for mixing multiple chips)
-wavenoise - Treat the wave channel as noise
-wavenone - ignore the wave channel (ie: do not include it)
-enable7bitnoise - retune the noise channel when it's in 7 bit more (not recommended)
-ignoreweird - ignore anything else unexpected - for instance, a 70hz clock. Ignored values are treated as if they were 0 and set to default. This is not always correct! Sometimes you need to edit the source file.

The Gameboy has a 32-sample wavetable channel that is used for additional instruments. Normally, this converter will average it for volume, and then apply its frequency to the third tone channel. You can make it a noise channel instead with "wavenoise", if the tune primarily uses it that way, or disable it completely with "wavenone" if it gets in the way.

```
D:\>vgm_gb2psg.exe smb_gb.vgz
Import VGM DMG (Gameboy) - v03202020
Reading smb_gb.vgz - 1368 bytes
Signature not detected.. trying gzip (vgz)
Decompressed to 5318 bytes
Reading version 0x161
Warning: rate set to zero - treating as 60hz
Refresh rate 60 Hz
Warning: fine timing lost.
Warning: ignoring unsupported duty cycle
File 1 parsed! Processed 1559 ticks (25.983333 seconds)
Adapting tones for PSG clock rate... 157 tones clipped
-Writing channel 0 as smb_gb.vgz_ton00.60hz...
-Writing channel 2 as smb_gb.vgz_ton01.60hz...
```

```
-Writing channel 4 as smb_gb.vgz_ton02.60hz...
-Writing channel 6 as smb_gb.vgz_noi03.60hz...
Skipping channel 8 - no data
Skipping channel 10 - no data
Skipping channel 12 - no data
Skipping channel 14 - no data
done vgm_gb2psg.
```

----

vgm_pokey2psg [-q] [-d] [-o <n>] [-add <n>] [-disableperiodic] [-ignorehighpass] [-ignoreweird] <filename>

Extracts Atari Pokey channels from a VGM or VGZ file, dumping the individual tracks in PSG format. Dual chip is supported. The following flags are recognized:

-q - quieter verbose data - less output data
-d - enable parser debug output - every chip command and frame tick is output
-o <n> - output single channel 'n' - used to test a single channel
-add <n> - add a fixed offset to the channel output index (helpful for mixing multiple chips)
-disableperiodic - The shorter noise filters on the Pokey will be output as periodic noise rather than white noise, but this won't always (ever?) work well. Use this switch to always output white noise.
-ignorehighpass - The Pokey has very crude high pass filters on two of the channels. This attempts to incorporate the spirit of them, but does so on raw frequency alone. If you find it interfering with the song, this will disable the high pass muting.
-ignoreweird - ignore anything else unexpected - for instance, a 70hz clock. Ignored values are treated as if they were 0 and set to default. This is not always correct! Sometimes you need to edit the source file.

Noise is managed as best as possible, with noises split out to a separate channel from tones for further processing. This means that a single 4 channel Pokey can output up to 8 channels of audio.

Unfortunately there were not very many Pokey VGMs at the time of writing, so I've really only tested it against arcade Tetris.

```
D:\>vgm_pokey2psg.exe brad_pokey.vgz
Import VGM Pokey - v03222020
Reading brad_pokey.vgz - 5815 bytes
Signature not detected.. trying gzip (vgz)
Decompressed to 26565 bytes
Reading version 0x161
Dual Pokey output specified (we shall see!)
PSG clock scale factor 0.125000
Warning: rate set to zero - treating as 60hz
Refresh rate 60 Hz
Warning: fine timing lost.
Warning: fine timing lost.
```

```
Warning: Delay time loses precision (total 148, remainder 148
samples).
Warning: fine timing lost.
Warning: fine timing lost.
File 1 parsed! Processed 4144 ticks (69.066667 seconds)
Adapting tones for PSG clock rate...
Skipping channel 0 - no data
Skipping channel 2 - no data
-Writing channel 4 as brad_pokey.vgz_ton00.60hz...
Skipping channel 6 - no data
Skipping channel 8 - no data
Skipping channel 10 - no data
-Writing channel 12 as brad_pokey.vgz_ton01.60hz...
Skipping channel 14 - no data
Skipping channel 16 - no data
Skipping channel 18 - no data
-Writing channel 20 as brad_pokey.vgz_ton02.60hz...
Skipping channel 22 - no data
Skipping channel 24 - no data
Skipping channel 26 - no data
Skipping channel 28 - no data
Skipping channel 30 - no data
done vgm_pokey2psg.
```
----

vgm_nes2psg [-q] [-d <n>] [-o <n>] [-add <n>] [-triangle <n>] [-enableperiodic] [-disabledmcvolhack] [-dmcnoise|-dmcnone] [-ignoreweird] <filename>

Extracts NES APU channels from a VGM or VGZ file, dumping the individual tracks in PSG format. Dual chip is supported. The following flags are recognized:

-q - quieter verbose data - less output data
-d <n> - enable detailed debug for channel <n> - channel is 1-5
-o <n> - output single channel 'n' - used to test a single channel
-add <n> - add a fixed offset to the channel output index (helpful for mixing multiple chips)
-triangle <n> - set triangle volume - new value is 0 (mute) to 15 (loudest). Default is 8.
-enableperiodic - use periodic noise for short pattern - otherwise always use white noise
-disabledmcvolhack - don't adjust triangle and noise volume from dmc
-dmcnoise - **play dmc frequencies on noise channel**
-dmcnone - disable dmc channel
-ignoreweird - ignore anything else unexpected - for instance, a 70hz clock. Ignored values are treated as if they were 0 and set to default. This is not always correct! Sometimes you need to edit the source file.

The NES soundchip has a number of unusual features. Of note here, the triangle channel has no volume control, so a midrange volume is used by default. You can adjust this default value with the -triangle switch.

The noise generator has a long period and a short period - if 'enableperiodic' is passed, then the short period will be represented by periodic noise instead of white noise. This is unlikely to be correct, but the option is provided.

By default, the DMC volume level can modulate the volume of the triangle and noise channels slightly. (This is a hardware artifact in the NES). If you are getting undesirable volume changes, use -disabledmcvolhack to disable this.

The DMC (Delta Modulation Channel) itself is a DMA-driven waveform playback channel, which can not be reproduced by this system. The samples are averaged over each frame period and converted to a volume level, and the shift rate is stored as a playback frequency, with the channel stored as a PSG-like square wave channel. Whether this will be satisfactory depends entirely on how the music uses this channel. In some cases, further tuning will be required. In some cases, noise will better reproduce the sound - in that case use the “-dmcnoise” switch. And sometimes you can’t do anything for it (for instance, if voices are used). The “-dmcnone” switch will disable the dmc output.

```
D:\>vgm_nes2psg.exe sil_nes.vgz
Import VGM NES - v20200328
Reading sil_nes.vgz - 29317 bytes
Signature not detected.. trying gzip (vgz)
Decompressed to 101319 bytes
Reading version 0x161
Warning: rate set to zero - treating as 60hz
Refresh rate 60 Hz
Unsupported duty cycle ignored.
Unknown NES register write: $400D = 0x08
Warning: fine timing (1 samples) lost.
Unknown NES register write: $4009 = 0x08
Copy DMC data to 0xC000, length 0x4000
Warning: fine timing (9 samples) lost.
Warning: Delay time loses precision (total 730, remainder 730
samples).
Unknown NES register write: $400D = 0x08
Unknown NES register write: $4009 = 0x08
Warning: fine timing (7 samples) lost.
Warning: fine timing (7 samples) lost.
File 1 parsed! Processed 4692 ticks (78.200000 seconds)
Adapting tones for PSG clock rate...
-Writing channel 1 as sil_nes.vgz_ton00.60hz...
-Writing channel 2 as sil_nes.vgz_ton01.60hz...
-Writing channel 3 as sil_nes.vgz_ton02.60hz...
-Writing channel 4 as sil_nes.vgz_noi03.60hz...
-Writing channel 5 as sil_nes.vgz_ton04.60hz...
Skipping channel 6 - no data
Skipping channel 7 - no data
Skipping channel 8 - no data
Skipping channel 9 - no data
Skipping channel 10 - no data
done vgm_nes2psg.
```

----

prepare4SN <tone1> <tone2> <tone3> <noise> <output>

Takes in three tone channels and one noise channel, any of which may be "-" to leave muted. This will parse the PSG format inputs and produce a combined output file ready to be compressed for the SN PSG chip.

The output is a combined file with 8 comma-separated, hexadecimal values per row, representing the data for the PSG for each frame, alternating channel shift rate, volume. The noise is presented as the PSG noise command, optionally with the trigger bit set (0x10000).

Ex: `0x0017B,0x3,0x000D5,0x2,0x0011C,0x2,0x10002,0x0`

It will scale the volume levels from 8-bit linear to the PSG's 4-bit logarithmic attenuation, and then it will convert the noise channel from shift rates to the PSG's noise 'type'. If the noise shift rate is not one of the fixed rates, then it will attempt to use channel three to map the shift rate (with no additional scaling). If no voices are free for this, then it will map to the closest fixed shift rate.

The periodic and trigger flags are recognized on the noise channel, and will be incorporated into the mapping. Periodic will set the appropriate noise type while the trigger flag is passed out for the compressor. Note that the periodic flag output is the inverse of the hardware, which sets the bit 0x02 for white noise (types 4-7) and clears it for periodic (types 0-3).

The tool assumes that the input files are the correct type - but it is acceptable to pass a tone input to the noise channel - it will be mapped to white noise as close as possible and probably won't do what you want. It also does not pay attention to frame rate. The length of the output file will be the length of the shortest input file.

For tones, the only limitation is that tones with a pitch too low to be played (ie: with a counter greater than 0x3ff) will be clipped to 0x3ff. Preprocess with other tools for finer control.

```
D:\>prepare4SN.exe takom.VGM_ton00.60hz takom.VGM_ton01.60hz
takom.VGM_ton02.60hz takom.VGM_noi03.60hz takonout.psg
Opened tone channel 0: takom.VGM_ton00.60hz
Opened tone channel 1: takom.VGM_ton01.60hz
Opened tone channel 2: takom.VGM_ton02.60hz
Opened noise channel 3: takom.VGM_noi03.60hz
Imported 15868 rows
0 custom noises (non-lossy)
0 tones moved   (non-lossy)
0 tones clipped (lossy)
0 noises mapped (lossy)
** DONE **
```

----

prepare4AY <tone1> <tone2> <tone3> <noise> <output>

Takes in three tone channels and one noise channel, any of which may be "-" to leave muted. This will parse the PSG format inputs and produce a combined output file ready to be compressed for the AY PSG chip.

The output is a combined file with 8 comma-separated, hexadecimal values per row, representing the data for the PSG for each frame, alternating channel shift rate, volume. The noise is presented as the PSG noise command, optionally with the trigger bit set (0x10000).

Ex: `0x0017B,0x3,0x000D5,0x2,0x0011C,0x2,0x10002,0x0`

It will scale the volume levels from 8-bit linear to the PSG's 4-bit logarithmic attenuation, and then it will map the noise channel to the appropriate volume attenuator. If the noise volume does not exactly match one of the tone volumes, and no tone channel is currently muted to be used, then the closest match will be selected. Ideally, noise volume should be externally processed before reaching this tool to avoid this.

In addition, noises with a shift rate greater than the maximum of 0x1f will be clipped to 0x1f. Preprocess with other tools to avoid this.

The periodic and trigger flags are ignored on the noise channel.

The tool assumes that the input files are the correct type - but it is acceptable to pass a tone input to the noise channel - it will be mapped to white noise as close as possible and probably won't do what you want. It also does not pay attention to frame rate. The length of the output file will be the length of the shortest input file.

For tones, the only limitation is that tones with a pitch too low to be played (ie: with a counter greater than 0xfff) will be clipped to 0xfff. Preprocess with other tools for finer control.

```
D:\>prepare4AY.exe ab_psg.vgz_ton00.60hz ab_psg.vgz_ton01.60hz
ab_psg.vgz_ton02.60hz ab_psg.vgz_noi03.60hz ab.psgay
VGMComp2 AY Prep Tool - v20200525

Opened tone channel 0: ab_psg.vgz_ton00.60hz
Opened tone channel 1: ab_psg.vgz_ton01.60hz
Opened tone channel 2: ab_psg.vgz_ton02.60hz
Opened noise channel 3: ab_psg.vgz_noi03.60hz
Imported 5042 rows
396 tones moved   (non-lossy)
0 tones clipped (lossy)
0 noises clipped (lossy)
949 noises mapped (lossy)
** DONE **
```

----

vgmcomp2 [-d] [-dd] [-v] [-minrun s,e] [-alwaysrle] [-norle] [-norle16] [-norle24] [-norle32] [-nofwd] [-nobwd] <-ay|-sn> <filenamein1.psg> [<filenamein2.psg>...] <filenameout.sbf>

Takes in one or more prepared SN or AY files, and creates an output SBF (sound block format) file for either AY or SN playback (it matters a little bit). The following flags are supported:

-d - enable detailed debug information about the data packing (not normally useful)

-dd - enable detailed debug information about the string evaluation (not normally useful)
-v - enable verbose information about the data packing (often interesting! For what it's worth, I always like to see this information.)
-ay - specify data is intended for AY8910 playback. This or -sn is mandatory.
-sn - specify data is intended for SN PSG playback. This or -ay is mandatory.
-minrun s,e - the default search for minimum runs in 0,7, you can change this to optimize your search. It should be rarely, if ever needed. The maximum value is 20.

There are a number of parameters to disable tests in the compressor. In general, the compressor should make good decisions about the best choice (it tries enough of them!), but in rare cases disabling one or more modes may help, particularly the larger RLE modes.

-alwaysrle - always use RLE if available, rather than only if it seems to beat string. Other disables are still honored.
-norle - disable single-byte RLE
-norle16 - disable 16-bit RLE
-norle24 - disable 24-bit RLE
-norle32 - disable 32-bit RLE
-nofwd - disable forward searching (this speeds compression a lot)
-nobwd - disable backward searching (this pretty much defeats the purpose)

And, there are some 'secret' switches to trigger debug information. Some of these switches may cause malfunction, don't use them in production work.

-dd - dump detailed information about the encoding process
-deepdive - test each stream with every combination of disable switches. This can take a long time to run and usually doesn't save more than a hundred bytes or so. It is safe to use it if you need it, however.
-checkanyway - if a size mismatch is detected during encoding, do the row test anyway. This can cause a crash as the size mismatch indicates something went wrong during the encode.

Either **-ay** or **-sn** MUST be specified. You can not mix formats in one file (or if you do, it is up to you to be aware of the differences). Note that the output file is specified before any input files are listed. Recommend that the output file be named .sbfsn or .sbfay to keep track of it, but this is not enforced.

Current differences are summarized here:

	AY8910	**PSG**
Tone counter range	0x001 - 0xfff	0x001 - 0x3ff
Tone volume levels	Absolute: 0 (mute), 0xF (loudest)	Attenuation: 0 (loudest), 0xF (mute)
Noise range	0x00 - 0x1f (counter)	0x00 - 0x0f (lookup table)

Noise volume stream	Mixer command, shifted	Actual volume 0x00-0x0f
Note Table	Least significant 8 bits first for fine tune (first) register, then most significant 4 bits for coarse tune (second) register.	Least significant 4 bits for command byte (OR in command nibble), then most significant 6 bits for data byte.

Except for the note table, these differences all occur during the “prepare4XXX” step. While I have not attempted it, in theory you could combine the two types of data using the AY mode (which clips less), and just deal with the note table difference to the SN yourself.

```
D:\>vgmcomp2 -sn contra_gb.psg output.sbf
Successful!

1 songs (3.550000 seconds) compressed to 226 bytes (63.662000
bytes/second)

** DONE **
```
