What is it?

This is a tool set aimed at converting and compressing audio files for playback on the TI SN series of PSG audio chips. The goal is to find a reasonable tradeoff between CPU time, compression size, and RAM usage, so that it may be used on smaller 8-bit systems. The playback routines specifically target the TI-99/4A and ColecoVision, but are adaptable to others.

Because the ColecoVision provides an AY-8910 expansion (via the SGM, Phoenix, and similar compatibles) and the TI offers a C64 SID expansion (the SID Blaster), degenerate support for these chips is also offered. What I mean by degenerate here is that while data will be formatted to the restrictions of the chips and playback routines will honor them, they will be treated simply as extra SN channels and the extra features (waveforms, filtering, etc) will not be used. They will simply play back additional square wave tone and/or noise channels.

This is important to note. Support for the AY and SID does NOT mean feature support. Even if you convert an AY song and play it back on an AY chip, the conversion step will adapt it for playback on an SN and any special features will be emulated, or at worst, discarded.

SBF - Sound Block Format

This is the compressed binary output file. Unless you are writing your own player, you can skip the rest of this section.

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 can not 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).
    - This means ####CBA0 - the mixer needs 00CBA000 - so shift left twice then AND #$38

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

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 (3 on SID), 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, AY or SID. Though they are close, they are not interchangeable (actually, the SID isn't even close).

Volume differs the most. When updating volume on the SN, 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. (The least significant nibble still contains the delay).

The SID is formatted similar to the SN, but uses positive volume rather than an attenuation. This is used to set the sustain level for a channel.

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 (for SN), 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. The SID supports a full 16-bits, but is limited here to the same range as the AY.

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. The

SID does not have a dedicated noise channel, and whether a channel is noise or tone is set before the song is started.

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.

110 - Back reference - the next two bytes are the offset from the CURRENT file pointer - that is, add this 16-bit signed offset to the address of the two offset bytes, before incrementing mainptr. This strange reference just allows slightly more optimal code since that information is available

at the time we read it! 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

This is an ASCII text format used as an intermediate formate for all the sound chips. This section is useful to understand how the editing tools find data. It is generated by the tools that end with “2psg”.

Normally each file contains just a single channel, and chip-specific restrictions are relaxed to reduce loss of resolution while editing. The filename contains important metadata:

xxxxx_AAABB.CCCC

- xxxxxx - your chosen filename
- 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 SN PSG shift counts (even for noise), and the volume is always linear from 0-255, where 0 is mute, and 255 is maximum.

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 can not 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 and two digits. 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. (Many tools also let you specify an offset when exporting).

A very similar format is used when using the "prepare4sn", "prepare4ay" and “prepare4sid” tools, except these tools will emit all four channels in a single row (the fourth channel for SID will be a dummy channel). The filename does not require metadata in this case.

Tools

This section describes all the tools available, first in a summary list, then each tool in detail.

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

mod2psg - convert Protracker style MODs and MIDI to PSG format (many formats ok)
sid2psg - convert 6510-based SID files to PSG format
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_md2psg - extract Megadrive/Genesis data (YM2612 and SN PSG) from a VGM
vgm_nes2psg - extract NES APU 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_psg2psg - extract PSG audio data from a VGM and export to PSG format (dual chip ok)
voice2psg - convert a voice sample to PSG data

Editing Tools

arptones - merge two (normally tone) channels with arpeggio
arp3tones - merge three (normally tone) channels with arpeggio
bass2noise - convert bass tones to SN periodic noise
changespeed - change playback speed from the default 60hz
cleardupes - mute duplicate notes on a channel
clipmaxvol - clamps the maximum volume on a channel
cliptones - clips out of range notes into range on a channel
despeckle - remove single-row sounds from a channel
findmuted - check all channels of a song for empty channels
fixsnnoise - remap a noise channel to only use SN fixed shift rates
forcenoisetype - force noise type on a channel to either white or periodic
maximizevolume - maximize volume over an entire song
mergenoise - merge two (normally noise) channels by volume

mutetones - mutes out of range notes on a channel
padsilence - add silence to the beginning and end of a song
reducetonecnt - reduce the number of frequencies in a song to fit the 256 note limit
reducevolumecnt - reduce the number of distinct volume levels to try and improve compression
scalepitch - change the pitch of a channel
scalevolume - change the volume of a channel
smoothvolume - smooth out noisy volume (such as mod2psg can do)
trimsilence - remove silence from beginning and end of song
underlaytones - combine two channels, giving one priority

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.
prepare4sid - reads 3 PSG channel files and enforces limits appropriate to the SID PSG, outputting a combined file.
QuickPlayer - Windows tool to convert an SBF into a playable file for TI or ColecoVision
quickplayercmd - same tool but for command prompt
vgmcomp2 - compresses a prepared PSG file into a final SBF file.
bestPacker - takes a list of prepared PSG files, and tests combinations with vgmcomp2 to see which songs pack well together. Output is a list of commands for a batch 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.
-hz x - set playback to specified rate, instead of 60hz. Files with an extension like ".60hz" will overrule this option, but it is helpful for PSG or SBF files.
-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--
```

---

mod2psg [-q] [-d] [-o <n>] [-add <n>] [-speedscale <n>] [-vol <n>] [-volins <ins> <n>] [-tunetone <n>] [-tunenoise <n>] [-tuneins <ins> <n>] [-snr <n>] [-usedrums] [-forcenoise <str>] [-forcetone <str>] [-forcefreq <ins> <n>] <filename>

Converts a tracker-style song (such as Protracker and its many variants) into the vgmcomp data format for further processing and conversion.

MODs are sample based, complex music, and a direct conversion is not straight-forward. The software gives a pretty good starting point, but you should expect to do some tuning, and you may need to view and/or edit the original MOD file in order to get the information you need for a clean conversion.

The converter will move each channel's noise to a separate dedicated noise channel, so a 4 channel MOD can double to 8 channels, and so forth, depending on where the noises play. Other tools can be used to combine the channels if needed.

-q - quieter verbose data - less output data
-d - enable detailed debug - more output data
-dd - enable even more detailed debug - hidden option - also outputs spectrum BMPs for each sample and a wave file named modsound.wav
-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)
-speedscale <float> - scale song rate by float (1.0 = no change, 1.1=10% faster, 0.9=10% slower) Use this if the song seems too fast or too slow. Note that this just changes how many samples represent a frame.
-vol <float> - scale song volume by float (1.0=no change, disables automatic volume scale) - use this if the entire song is too loud or too quiet
-volins <instrument> <float> - scale a specific instrument volume by float (1.0=no change, 2.0=octave DOWN, 0.5=octave UP, default 1.0) - use this if a single instrument is too loud or too quiet. You can mute an instrument with 0.0, or max it out with 255.0
-tunetone <float> - scale tone frequency by float (1.0=no change, 2.0=octave DOWN, 0.5=octave UP, default 1.0) - use this to transpose all tone samples up or down (remember that fractions are HIGHER pitch and whole numbers are LOWER).
-tunenoise <float> - scale noise frequency by float (1.0=no change, 2.0=octave DOWN, 0.5=octave UP, default 0.5) - use this to transpose all noise samples up or down, same as tone. Note that the default for noise IS to transpose to one octave higher.
-tuneins <instrument> <float> - scale a specific instrument by float (1.0=no change, 2.0=octave DOWN, 0.5=octave UP, default 2.0) - use this to transpose a single instrument higher or lower. Note that the default IS to transpose to one octave lower.
-snr <int> - SNR ratio from 1-100 - SNR less than this will be treated as noise (use -d to see estimates) - use this to adjust the threshold for noise detection in the automatic SNR estimation. Use -d to see the calculated values - the default is 70.
-usedrums - read the old DRUMS$ tag from the sample name from the old mod2psg - no autodetection - the old mod2psg tool allowed you to predefine the samples which would be noise channels with a specially named sample - this makes the program read that tag. The tag *must* exist if you specify this switch. It's not recommended.

-forcenoise <str> - 'str' is a comma-separated list of sample indexes to make noise, no spaces! - for instance, “-forcenoise 3,4,7,8”. Use this to override the autodetection.
-forcetone <str> - 'str' is a comma-separated list of sample indexes to make tone, no spaces! - for instance “-forcetone 1,5,9”. Use this to override the autodetection.
-forcefreq <instrument> <freq> - force an instrument to always use a fixed frequency. This can be helpful especially for drums that don’t change frequency in the MOD, but you want them all to output as different frequencies. For the SN chip, frequencies that correspond to fixed shift rates are 16 (highest pitch), 32, and 64 (lowest pitch).
<filename> - MOD file to read.

mod2psg uses the ModPlug library and should be able to read anything it can. Supported types should be: ABC, MOD, S3M, XM, IT, 669, AMF, AMS, DBM, DMF, DSM, FAR, MDL, MED, MID, MTM, OKT, PTM, STM, ULT, UMX, MT2 and PSM.

If you have Timidity .pat patches for MIDI, you can set the MMPAT_PATH_TO_CFG environment variable and it should be able to use them. *However, note that MIDI doesn’t appear to work correctly at the moment.*

```
D:\>mod2psg.exe DLT1.MOD
Import MOD tracker-style files - v20200716

Module name: delta remix (no.1)
Module type: MOD
Channels: 4
Sample  1: st-01:touch                          17:
Sample  2: st-43:acidbass                       18:
Sample  3: st-07:mfcelectronic                  19:
Sample  4: st-48:snare                          20:
Sample  5: st-02:bassdrum5                      21:
Sample  6: st-06:wind                           22:
Sample  7:                                      23:
Sample  8:                                      24:
Sample  9:                                      25:
Sample 10:                                      26:
Sample 11:                                      27:
Sample 12:                                      28:
Sample 13:                                      29:
Sample 14:                                      30:
Sample 15:                                      31: DRUMS$18000000
Sample 16:
Scanning for volume...
Calculated volume scale of 1.564417
Playing out song...
```

```
-Writing channel 0 as DLT1.MOD_ton00.60hz...
-Writing channel 1 as DLT1.MOD_ton01.60hz...
-Writing channel 2 as DLT1.MOD_ton02.60hz...
-Writing channel 3 as DLT1.MOD_ton03.60hz...
Wrote 6765 rows...

** DONE **
```

---

quickplayercmd (-ti|-coleco) [-loop] [-sn <sn music>] [-sid <sid music>] [-ay <ay music>] [-sidctl c1 c2 c3] [-text <textfile>] <outputfile>
(and QuickPlayer for Windows)

Converts an SBF compressed file (or two files, if they use the SN/SID for TI or SN/AY for ColecoVision) into a standalone playable file for the system. TI will output E/A#5 files in TIFILES format, and Coleco will output a standard 32k ROM image. The maximum music size for either system is 24k.

-ti - select TI output - this or -coleco *MUST* be specified
-coleco - select ColecoVision output - this or -ti *MUST* be specified
-loop - loop the music - if not specified the music will stop at the end
-sn - specify file for SN chip - no SN is played if blank
-sid - specify file for SID chip - no SID is played if blank. Valid on TI only. Requires sidctl.
-ay - specify file for AY chip - no AY is played if blank. Valid on Coleco only.
-sidctl - set SID control registers - 0x80 for noise, 0x40 for pulse, 0x20 for sawtooth, 0x10 for triangle. All three voices will be pulse if not specified. Other values may cause unintended effects.
-text - text file to display - maximum line length is 32 characters, and maximum 24 lines. Blank screen is displayed if absent.

---

sid2psg [-q] [-d] [-o <n>] [-add <n>] [-speedscale <n>] [-subtune <n>] <-len <n>> <filename>

Converts a 6510-based SID file to a PSG. Full emulation of the 6510 is included and up to three SIDs are supported, thanks to the csid codebase by Hermit.

-q - quieter verbose data
-d - enable parser debug output
-dd - secret option for more detailed debug, also writes a wave file ‘sidsound.wave’
-o <n> - output only channel <n> (1-max)

-add <n> - add 'n' to the output channel number (use for multiple chips, otherwise starts at zero)
-speedscale <float> - scale song rate by float (1.0 = no change, 1.1=10% faster, 0.9=10% slower)
-subtune <n> - play subtune 'n', from 1-64 (default 1)
-len <n> - play duration in seconds (MUST be specified! Unfortunately since it is played via emulation, there's no recognizable way to know when a song ends. Use an external player to determine how much time you need.)
<filename> - SID file to read.

---

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_md2psg [-q] [-d] [-o <n>] [-add <n>] [-ignoreweird] [-dacvol <n>] [-fmvol <n>] [-snvol <n>] [-notunenoise] [-noscalefreq] <filename>

Extracts MegaDrive/Sega Genesis channels from a VGM or VGZ file, dumping the individual tracks in PSG format. Both the YM2612 and SN PSG (if present) are extracted. The following flags are recognized:

-q - quieter verbose data - less output data
-d - enable parser debug output - increased debug information is output, including a raw wave file (sndout.raw) - though this contains only the YM levels and the DAC (16-bit signed stereo)
-dd - enable more debug output - hidden command for additional debug information. This also outputs a raw wave file which includes the original YM waveform and YM DAC.
-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)
-dacvol <n> - scale the DAC volume by a floating point value - 1.0 is no change. See help for the default (which is 1.0 as of this date)
-fmvol <n> - scale the FM volume by a floating point value - 1.0 is no change. See help for the default (which is 1.1 as of this date)
-snvol <n> - scale the SN PSG volume by a floating point value - 1.0 is no change. See help for the default (which is 1.0 as of this date)
-notunenoise - Do not retune SN 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 SN 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.

The YM channel 6 can output as either DAC or FM. When in DAC mode, this converter will average the output volume and generate a fixed noise frequency for it. Other FM channels are written as TON. The PSG, when present, will process as in VGM_PSG2PSG as additional channels.

This converter uses a cycle-accurate YM emulator by Alexey Khokholov (Nuke.YKT), under GPLv2. This causes it to run a little slower than other converters so your process may take some time. During the conversion, the number of seconds in the song so far will be displayed.

```
D:\>vgm_md2psg.exe SonicTitle.vgz
Import VGM MD (MegaDrive/Genesis) - v20201002
Reading SonicTitle.vgz - 8376 bytes
Signature not detected.. trying gzip (vgz)
Decompressed to 95726 bytes
Reading version 0x150
Refresh rate 60 Hz (735 samples per tick)
Selecting 16-bit shift register.
Warning: fine timing (3 samples) lost.
8 seconds...
File 1 parsed! Processed 525 ticks (8.750000 seconds)
Adapting user-defined shift rates...0 notes tuned.

-Writing channel 1 as SonicTitle.vgz_ton00.60hz...
-Writing channel 2 as SonicTitle.vgz_ton01.60hz...
-Writing channel 3 as SonicTitle.vgz_ton02.60hz...
-Writing channel 4 as SonicTitle.vgz_ton03.60hz...
-Writing channel 5 as SonicTitle.vgz_ton04.60hz...
Skipping channel 6 - no data
-Writing channel 7 as SonicTitle.vgz_noi05.60hz...
Skipping channel 8 - no data
Skipping channel 9 - no data
Skipping channel 10 - no data
-Writing channel 11 as SonicTitle.vgz_noi06.60hz...
done vgm_md2psg.
```

---

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.
```

---

voice2psg [-q] [-d] [-o <n>] [-add <n>] [-speedscale <n>] [-volume <n>] [-channels <n>] [-maxfreq <n>] <filename>

While crude, this tool can convert some voice samples to recognizable PSG files that play back at 60hz. The samples must be clear, loud, and without any background noise (or music). Even with that, some voices will simply not convert very well. Music WILL NOT convert well.

It might be advantageous to clamp the maximum frequency in order to restrict the number of harmonics that are captured. In my testing 2000Hz is the minimum frequency to get recognizable speech back out, but many samples performed better with the maximum range.

By default the converter will use 3 channels for playback on the SN chip, however, it often takes 5 channels to get acceptable quality. There seems to be no benefit to capturing additional channels and merging them with the arp tool, however.

Volume is often quite low - the default internal scale is set to 4.0. If this clips or is still too quiet, you can override it with the -volume switch.

What makes a good sample conversion is still a subject of research. This tool is presented as-is.

-q - quieter verbose data
-d - enable parser debug output
-dd - enable detailed debug (hidden option)
-o <n> - output only channel <n> (1-max)
-add <n> - add 'n' to the output channel number (use for multiple chips, otherwise starts at zero)
-speedscale <float> - scale sample rate by float (1.0 = no change, 1.1=10% faster, 0.9=10% slower)
-volume <float> - scale volume by float (4.0 = default, 4.1=louder, 3.9=quieter)
-channels <n> - number of channels to create (1-15, default 3)
-maxfreq <n> - maximum frequency to search for (default 22050)
<filename> - wave file to read.

```
d:\>voice2psg.exe ti.wav
Parse voice to PSG - v20200801

-Writing channel 0 as ti.wav_ton00.60hz...
-Writing channel 1 as ti.wav_ton01.60hz...
-Writing channel 2 as ti.wav_ton02.60hz...
Wrote 260 rows...

** DONE **
```

---

arptones <chan1> <chan2>

Inputs two channels, and merges the second into the first. In cases where the channels conflict, they will be combined by alternating channels as an arpeggio - chan1 on even channels, and chan2 on odd channels.

Original files are renamed with *.arp.old* extension.

```
D:\>arptones DIGILOO.MOD_ton00.60hz DIGILOO.MOD_ton03.60hz
VGMComp2 Arpeggio Tool - v20200717

Opened channel 0: DIGILOO.MOD_oldton00.60hz
Opened channel 1: DIGILOO.MOD_oldton03.60hz
Imported 8685 rows
5930 tones moved   (non-lossy)
660 tones arped   (lossy)
** DONE **
```

---

arp3tones <chan1> <chan2> <chan3>

Inputs three channels, and merges the second and third into the first. In cases where the channels conflict, they will be combined by alternating channels as an arpeggio.

Original files are renamed with *.arp3.old* extension.

```
D:\>arp3tones guile_gb.vgz_ton00.60hz guile_gb.vgz_ton01.60hz
guile_gb.vgz_ton02.60hz
VGMComp2 Arpeggio3 Tool - v20200717
```

```
Opened channel 0: guile_gb.vgz_ton00.60hz
Opened channel 1: guile_gb.vgz_ton01.60hz
Opened channel 2: guile_gb.vgz_ton02.60hz
Imported 4183 rows
317 tones moved   (non-lossy)
3837 tones arped   (lossy)
Writing: guile_gb.vgz_ton00.60hz
** DONE **
```

---

bass2noise [-q] [-nomute] <song>

Parses an entire song, and moves the loudest out-of-range bass frequency (for the SN chip) to a newly created noise channel as tuned periodic noise. Other bass on the same row, if any, is muted by default, or can be left alone with the -nomute flag.

You must pass the *full* filename to *any* channel in the song.

-q - quiet verbose output
-nomute - do not mute additional bass notes. In the event this occurs, the song will need further processing to be played on the SN.

One option to dealing with multiple bass notes on the same row is to run this tool multiple times - each pass will generate another noise channel. But for play on a single SN, you will still need to combine the noise channels eventually. (However, something like the ForTI on the TI will give you four SNs to play with, so this may still be useful.)

Original files are renamed to .bass2noise.old

```
D:\>bass2noise.exe Omake5.vgz_ton00.60hz
VGMComp2 Bass to SN Noise conversion Tool - v20201006

Opened channel 0: Omake5.vgz_ton00.60hz
Opened channel 1: Omake5.vgz_ton01.60hz
Opened channel 2: Omake5.vgz_ton02.60hz
Opened channel 3: Omake5.vgz_ton03.60hz
Opened channel 4: Omake5.vgz_ton04.60hz
Opened channel 5: Omake5.vgz_ton05.60hz
Opened channel 6: Omake5.vgz_ton06.60hz
Opened channel 7: Omake5.vgz_ton07.60hz
Opened channel 8: Omake5.vgz_ton08.60hz
```

```
Opened channel 9: Omake5.vgz_noi09.60hz

3763/45562 notes moved (non-lossy)
952/45562 notes muted (lossy).

Writing: Omake5.vgz_ton00.60hz
Writing: Omake5.vgz_ton01.60hz
Writing: Omake5.vgz_ton02.60hz
Writing: Omake5.vgz_ton03.60hz
Writing: Omake5.vgz_ton04.60hz
Writing: Omake5.vgz_ton05.60hz
Writing: Omake5.vgz_ton06.60hz
Writing: Omake5.vgz_ton07.60hz
Writing: Omake5.vgz_ton08.60hz
Writing: Omake5.vgz_noi09.60hz
Writing NEW: Omake5.vgz_noi10.60hz
** DONE **
```

---

changespeed [-q] [-notrigger] [-noarp] [-novol] <song> <output HZ>

Changes the speed of an entire song.

-q - quiet verbose output
-notrigger - don't check removed rows for noise trigger (rarely needed)
-noarp - don't check removed rows for arpeggio (rarely needed)
-novol - don't check removed rows for maximum volume (only needed if a previously muted sound continues forever)
song - pass in the entire name of ONE channel, the entire song will be found and updated.
output HZ - desired playback rate in HZ (normally 25, 30, 50 or 60)

Resamples an output song to run at the desired target rate. 25/30hz files can be played every other frame to save some CPU time (or split the channels so half are processed each frame). The reduced amount of data sometimes also reduces file size (but not always!) Usually they sound almost indistinguishable from the original. Extremely complex sounds, like speech samples, may be the exception.

Original files are renamed to *.speed.old*

You can specify any rate that you like, but note that the Testplayer tool only recognizes the four rates listed here. Also, whatever rate you choose, you need to be able to play back on your target hardware.

```
D:\>changespeed.exe splash.s3m_noi03.60hz 30
VGMComp2 Change Speed Tool - v20200803

Source file at 60Hz
Desired rate   30Hz
Ratio: 2.000000
Opened channel 0: splash.s3m_ton00.60hz
Imported 3392 rows
Sampled 1696 rows
38 arpeggios moved
662 volumes saved
Writing: splash.s3m_ton00.30hz
Opened channel 1: splash.s3m_ton01.60hz
Imported 3392 rows
Sampled 1696 rows
16 arpeggios moved
698 volumes saved
Writing: splash.s3m_ton01.30hz
Opened channel 2: splash.s3m_ton02.60hz
Imported 3392 rows
Sampled 1696 rows
374 volumes saved
Writing: splash.s3m_ton02.30hz
Opened channel 3: splash.s3m_noi03.60hz
Imported 3392 rows
Sampled 1696 rows
978 volumes saved
Writing: splash.s3m_noi03.30hz
4 channels converted to 30Hz
** DONE **
```

---

cliptones (-SN|-AY|-SID) <channel input>

Clips out-of-range notes in a channel to in-range.

-SN - specifies SN range
-AY - specifies AY range
-SID - specifies SID range

Exactly 1 of -SN, -AY or -SID must be specified.

Original files are renamed to *.cliptone.old*

```
D:\>cliptones.exe -SN DOLPHINS.MOD_ton01.60hz
VGMComp2 Tone Clipping Tool - v20200924

Opened channel 0: DOLPHINS.MOD_ton01.60hz
Imported 21467 rows
3 notes clipped (lossy)
Writing: DOLPHINS.MOD_ton01.60hz
** DONE **
```

---

cleardupes <chan1> <chan2>

Inputs two channels and checks for overlapping notes. If both channels have the same frequency, then the second channel will be muted. If the second channel was louder, its volume is copied to the first channel. Both files are written back out.

This can be useful for simplifying a song by removing duplicate data, and may be helpful for merging channels by providing less data to merge.

Original files are renamed with *.dupes.old* extension.

```
D:\>cleardupes.exe a_ton00.60hz a_ton01.60hz
VGMComp2 Duplicates Tool - v20200721

Opened channel 0: a_ton00.60hz
Opened channel 1: a_ton01.60hz
Imported 15260 rows
2076 channel 1 volume updated
4809 channel 2 mutes
Writing: a_ton00.60hz
Writing: a_ton00.60hz
** DONE **
```

---

clipmaxvol <max> <channel input>

Clamps any volume louder than ‘max’ to ‘max’. Max may be from 1 (quietest) to 254 (loudest). *(Because 0 is mute - to mute just delete the channel. 255 is maximum volume, and so processing is not required.)*

Original file is renamed to *.clipmaxvol.old*

```
D:\>clipmaxvol.exe 192 guile_gb.vgz_ton00.60hz
VGMComp2 Volume Clamping Tool - v20201008

Opened channel 0: guile_gb.vgz_ton00.60hz
Imported 4183 rows
608 volumes clipped (lossy)
Writing: guile_gb.vgz_ton00.60hz
** DONE **
```

---

despeckle <channel input>

Sometimes processing functions will leave solitary notes behind in a channel when nearby notes have been moved elsewhere, and these single notes leave an annoying click when hit. If they are no longer desired, run this tool and they will be muted.

The tool mutes a note that is active for exactly one tick, with mutes before and after it.

Original files are renamed to *.despeckle.old*

```
D:\>despeckle.exe Omake5.vgz_ton00.60hz
VGMComp2 Despeckle Tool - v20201006

Opened channel 0: Omake5.vgz_ton00.60hz
Imported 4142 rows
15 single-tick notes muted (lossy)
Writing: 43 - Omake 5.vgz_ton00.60hz
** DONE **
```

---

findmuted [-q] <song>

Analyzes an entire song and reports if any channel is entirely muted. Muted channels can simply be deleted at your whim.

-q - quiet verbose output

You must pass the *full* filename to *any* channel in the song.

The original song is not modified.

D:\work\TI\vgmcomp2\test\y>..\..\release\findmuted.exe "43 - Omake 5.vgz"
VGMComp2 Mute Channel Finding Tool - v20201006

Please enter the full name of a single channel, not the prefix. We'll find the rest!

```
D:\>findmuted.exe Omake5.vgz_ton00.60hz
VGMComp2 Mute Channel Finding Tool - v20201006

Omake5.vgz_ton00.60hz: 2680 notes / 4142 total (64 %)
Omake5.vgz_ton01.60hz: 838 notes / 4142 total (20 %)
Omake5.vgz_ton02.60hz: 3789 notes / 4142 total (91 %)
Omake5.vgz_ton03.60hz: 2892 notes / 4142 total (69 %)
Omake5.vgz_ton04.60hz: 3772 notes / 4142 total (91 %)
Omake5.vgz_ton05.60hz: 2896 notes / 4142 total (69 %)
Omake5.vgz_ton06.60hz: 2643 notes / 4142 total (63 %)
Omake5.vgz_ton07.60hz: 2643 notes / 4142 total (63 %)
Omake5.vgz_ton08.60hz: 2418 notes / 4142 total (58 %)
Omake5.vgz_noi09.60hz: 3444 notes / 4142 total (83 %)
Omake5.vgz_noi10.60hz: 3763 notes / 4142 total (90 %)
** DONE **
```

---

fixsnnoise <noise channel input>

Parses a noise channel (MUST contain “_noi” in the filename) and remaps all noise frequencies to one of the three fixed shift rates used on the SN sound chip. This ensures that channel 3 frequencies will not be required for playback.

Original files are renamed with .fixsnnoise.old

```
D:\>fixsnnoise.exe "43 - Omake 5.vgz_noi09.60hz"
VGMComp2 SN Noise Fix Tool - v20201006

Opened channel 0: 43 - Omake 5.vgz_noi09.60hz
Imported 4142 rows
3444 noises remapped (lossy)
```

```
Writing: 43 - Omake 5.vgz_noi09.60hz
** DONE **
```

---

forcenoisetype [-white|-periodic] <noise channel input>

Forces all noise in a channel to either white noise or periodic noise.

-white - change all noise to white noise
-periodic - change all noise to periodic noise

One of -white or -periodic *must* be specified.

Original file is renamed with *.forcenoisetype.old* extension.

```
D:\>forcenoisetype.exe -periodic guile_gb.vgz_noi02.60hz
VGMComp2 SN Noise Type Force Tool - v20201008

Opened channel 0: guile_gb.vgz_noi02.60hz
Imported 4183 rows
4183 noises updated (lossy)
Writing: guile_gb.vgz_noi02.60hz
** DONE **
```

---

maximizevolume [-q] [-max <n>] <song>

Reads and entire song and changes the volume so that the maximum volume is 255 (or whatever value you specify.)

-q - quiet verbose output
-max <n> - sets the maximum volume to <n> (default 255)
song - pass in the entire name of ONE channel, the entire song will be found and checked.

Original files are renamed to .maximizevolume.old

```
D:\>maximizevolume "Opening Theme.vgz_ton00.60hz"
VGMComp2 Song Volume Maximize Tool - v20210707

Opened channel 0: Opening Theme.vgz_ton00.60hz
Opened channel 1: Opening Theme.vgz_ton01.60hz
```

```
Opened channel 6: Opening Theme.vgz_ton06.60hz
Imported 866 rows...
Current maximum volume: 132
Desired maximum volume: 255
Scale percentage: 1.931818
Writing: Opening Theme.vgz_ton00.60hz
Writing: Opening Theme.vgz_ton01.60hz
Writing: Opening Theme.vgz_ton06.60hz
** DONE **
```

---

mergenoise <chan1> <chan2>

Inputs two channels, and merges the second into the first. In cases where the channels conflict, they will be combined by selecting the louder of the two - this is usually appropriate only for noise channels.

Original files are renamed with *.merge.old* extension.

```
D:\>mergenoise DIGILOO.MOD_noi04.60hz DIGILOO.MOD_noi05.60hz
VGMComp2 Noise Merge Tool - v20200718

Opened channel 0: DIGILOO.MOD_noi04.60hz
Opened channel 1: DIGILOO.MOD_noi05.60hz
Imported 8685 rows
272 tones moved      (non-lossy)
403 tones overridden (lossy)
** DONE **
```

---

mutetones (-SN|-AY|-SID) <channel input>

Mutes out-of-range notes in a channel.

-SN - specifies SN range
-AY - specifies AY range
-SID - specifies SID range

Exactly 1 of -SN, -AY or -SID must be specified.

```
D:\>mutetones.exe -SN DOLPHINS.MOD_ton01.60hz
```

```
VGMComp2 Tone Muting Tool - v20200924

Opened channel 0: DOLPHINS.MOD_ton01.60hz
Imported 21467 rows
3 notes muted (lossy)
Writing: DOLPHINS.MOD_ton01.60hz
** DONE **
```

---

padsilence [-q] <start rows> <end rows> <song>

Adds silence to the beginning and end of a song.

-q - quiet verbose output
song - pass in the entire name of ONE channel, the entire song will be found and checked.

Enter the number of rows for start and end, use 0 for no padding. 60 rows is 1 second.

Original files are renamed with *.padsilence.old* extension.

```
D:\>padsilence.exe 5 10 guile_gb.vgz_noi03.60hz
VGMComp2 Silence Adding Tool - v20201008

Opened channel 0: guile_gb.vgz_ton00.60hz
Opened channel 1: guile_gb.vgz_ton01.60hz
Opened channel 2: guile_gb.vgz_ton02.60hz
Opened channel 3: guile_gb.vgz_noi03.60hz
Imported 4182 rows...

5 rows will be added to start
10 rows will be added to end

Writing: guile_gb.vgz_ton00.60hz
Writing: guile_gb.vgz_ton01.60hz
Writing: guile_gb.vgz_ton02.60hz
Writing: guile_gb.vgz_noi03.60hz
** DONE **
```

---

reducetonecount [-q] [-nonoise] [-accept] [-notes <x>] [-notetol <x>] <song>

Reduces the number of unique tones in a song using a popularity based algorithm, weighted to prefer notes appearing to be musical.

-q - quiet verbose output
-nonoise - do not include the noise channel in processing (helpful if you know the noise channel is set for SN frequencies already)
-accept - accept larger values than normal on following arguments, if you have a reason for it
-notes <x> - reduce the song to <x> or fewer unique notes. 256 if not specified. (You can only request MORE than 256 if you use the *-accept* keyword first.)
-notetol <x> - specifies how many counts away from a defined musical note still gets a popularity boost for being musical. (The default value is 1, and this does not really cover the logarithmic scale - but for some songs a larger value may help reduce detuning. For a value larger than 25, pass the *-accept* keyword first.)

The <song> argument is specified as any channel in the song - the entire song will be loaded. Note this differs from the testPlayer which allows you to specify the song by a "prefix". This just avoids this tool needing to do all the complex searching and guessing that the testPlayer does.

Although the example below shows an 8 channel song being processed, it usually makes more sense to do this after converting to 4 channels, since that process will by definition remove some notes.

```
D:\>reducetonecnt.exe -notes 200 AWSOME4.MOD_noi04.60hz
VGMComp2 Tone Count reduction Tool - v20200923

Using musical note tolerance of 1
Opened channel 0: AWSOME4.MOD_ton00.60hz
Opened channel 1: AWSOME4.MOD_ton01.60hz
Opened channel 2: AWSOME4.MOD_ton02.60hz
Opened channel 3: AWSOME4.MOD_ton03.60hz
Opened channel 4: AWSOME4.MOD_noi04.60hz
Opened channel 5: AWSOME4.MOD_noi05.60hz
Opened channel 6: AWSOME4.MOD_noi06.60hz
Opened channel 7: AWSOME4.MOD_noi07.60hz
Counting number of frequencies...228 frequencies used (want 200).
-Popularity adjusted 5 ticks at step 2
Counting number of frequencies...227 frequencies used (want 200).
-Popularity adjusted 31 ticks at step 3
Counting number of frequencies...222 frequencies used (want 200).
-Popularity adjusted 46 ticks at step 4
Counting number of frequencies...213 frequencies used (want 200).
```

```
-Popularity adjusted 55 ticks at step 5
Counting number of frequencies...209 frequencies used (want 200).
-Popularity adjusted 86 ticks at step 6
Counting number of frequencies...198 frequencies used (want 200).
223/54120 notes adjusted
198 frequencies in final.
Writing: AWSOME4.MOD_ton00.60hz
Writing: AWSOME4.MOD_ton01.60hz
Writing: AWSOME4.MOD_ton02.60hz
Writing: AWSOME4.MOD_ton03.60hz
Writing: AWSOME4.MOD_noi04.60hz
Writing: AWSOME4.MOD_noi05.60hz
Writing: AWSOME4.MOD_noi06.60hz
Writing: AWSOME4.MOD_noi07.60hz
** DONE **
```

---

reducevolumecnt <channel> <new cnt>

Channel data is normally maintained with an 8-bit resolution, up sampled from the original source (which usually has 16 levels). In some cases, it may be helpful to reduce this to 8 or fewer levels before compression to gain a few more bytes. Alternately, some of the processor tools may increase the original number of levels from 16 to closer to the 256 count supported (ie: smoothing or filtering), and you desire to hear closer to the final result in the test player. In that case, just specify the channel file to edit and the new number of levels desired.

Due to rounding errors, the final output count of volumes may not be exactly what you requested, particularly when using a non-power-of-two value.

Original files are renamed with *.reducevol.old* extension.

```
D:\>reducevolumecnt wbtest.60hz 4
VGMComp2 Volume Resolution Reduction Tool - v20200919

Opened channel 0: wbtest_ton00.60hz
Imported 2879 rows
4 volume levels in output
0 volumes clipped
Writing: wbtest_ton00.60hz
** DONE **
```

---

scalepitch <scale> <channel input>

Scales the pitch (frequency) of all notes on a channel.

Scale - floating point scale to scale by. Octaves generally double or halve, so 2.0 is one octave lower (except on the SID where it's higher), and 0.5 is one octave higher (except on the SID where it's lower). However, you may enter any value.

Original file is renamed with *.pitch.old* extension.

```
D:\>scalepitch.exe 0.5 wb_psg2.vgz_ton05.60hz
VGMComp2 Pitch Scaling Tool - v20200724

Opened channel 0: wb_psg2.vgz_ton05.60hz
Imported 2879 rows, scaling by 0.500000
0 notes clipped (lossy)
0 notes zeroed (lossy)
Writing: wb_psg2.vgz_ton05.60hz
** DONE **
```

---

scalevolume (-maximize|<scale>) <channel input>

Scale the volume on a single channel louder (>1.0) or quieter (<1.0).

-maximize - instead of providing a value, calculate a scale to bring the loudest tick up to 255.

You must specify *either* -maximize, or a floating point value to scale by. In the case of -maximize, the selected scale will be emitted so that you know how much to scale the rest of the song by (if you want it to be consistent.)

```
D:\>scalevolume.exe -maximize DOLPHINS.MOD_ton00.60hz
VGMComp2 Volume Scaling Tool - v20200924

Opened channel 0: DOLPHINS.MOD_ton00.60hz
Maximum volume read 127/255
Imported 21467 rows, scaling volume by 2.007874
0 volumes clipped (lossy)
0 volumes zeroed (lossy)
Writing: DOLPHINS.MOD_ton00.60hz
```

```
** DONE **
```

---

smoothvolume [-hermite|-cubic] <channel input>

Performs a smoothing function on the volume component.

-hermite - Use a Hermite interpolation instead of average
-cubic - use a cubic interpolation instead of average

By default each sample is averaged with the two surrounding it, but you can also try different smoothing filters as above. Note that all of these filters will smooth volume changes and may cause fast notes to blend together. You can repeat for a stronger effect but tread carefully. Smoothing volumes may improve compression.

Original file is renamed with *.smooth.old* extension.

```
D:\>smoothvolume.exe GALAXYII.MOD_ton02.60hz
VGMComp2 Volume Smoothing Tool - v20200720

Opened channel 0: GALAXYII.MOD_ton02.60hz
Imported 15260 rows
12459 volumes updated
0 volumes clipped
** DONE **
```

---

trimsilence [-q] [-skipstart] [-skipend] <song>

Trims silence from beginning and end of a song.

-q - quiet verbose output
-skipstart - skip trimming the beginning
-skipend - skip trimming the end
song - pass in the entire name of ONE channel, the entire song will be found and checked.

This tool processes the ENTIRE song to find rows where all channels are silent. This will check both volume and frequency to determine mute (frequency counts less than 7 are considered mute, this is about 16khz).

Unless otherwise specified, start and end are both trimmed.

Original file is renamed with *.trimsilence.old* extension.

```
D:\>trimsilence.exe guile_gb.vgz_noi03.60hz
VGMComp2 Silence Trimming Tool - v20201008

Opened channel 0: guile_gb.vgz_ton00.60hz
Opened channel 1: guile_gb.vgz_ton01.60hz
Opened channel 2: guile_gb.vgz_ton02.60hz
Opened channel 3: guile_gb.vgz_noi03.60hz
Imported 4186 rows...

1 rows trimmed from start
3 rows trimmed from end

Writing: guile_gb.vgz_ton00.60hz
Writing: guile_gb.vgz_ton01.60hz
Writing: guile_gb.vgz_ton02.60hz
Writing: guile_gb.vgz_noi03.60hz
** DONE **
```

---

underlaytones <chan1> <chan2>

Merges chan2 into chan1. Chan1 always gets priority if both channels are active. Both original files are renamed with *.underlay.old* extension.

```
D:\>underlaytones.exe "TakeaChance.vgz_ton00.60hz"
"TakeaChance.vgz_ton01.60hz"
VGMComp2 Underlay Tool - v20201005

Opened channel 0: TakeaChance.vgz_ton00.60hz
Opened channel 1: TakeaChance.vgz_ton01.60hz
Imported 3057 rows
2292 tones moved   (non-lossy)
734 tones lost    (lossy)
Writing: TakeaChance.vgz_ton00.60hz
** DONE **
```

---

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. Note that because the source tools generate data intended for the TI SN chip, AY specific features such as envelopes are not supported.

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 range, 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 **
```

---

prepare4SID <tone1|noise1> <tone2|noise2> <tone3|noise3> <output>

Takes in three tone or noise channels, 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 SID PSG chip. Note that because the source tools generate data intended for the TI SN chip, SID specific features such as envelopes and filters are not supported. In addition, no metadata is stored regarding which channels are tone and which are noise - you must track this yourself.

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. There is no difference between tone and noise channels, so it is important to remember which is which. The fourth channel is output as the compressor requires it, but contains only dummy data.

Ex: `0x0017B,0x3,0x000D5,0x2,0x0011C,0x2,0x00001,0x0`

It will scale the volume levels from 8-bit linear to the SID's 4-bit linear range, and then it scales the frequency shift rates to match a 1MHz SID. If the resulting shift exceeds the 16-bit range, it will be clipped (use other tools to avoid this).

The periodic and trigger flags are ignored on any noise channels. It also does not pay attention to frame rate. The length of the output file will be the length of the shortest input file.

The only limitation is that tones or noises with a pitch too high to be played (ie: with a counter greater than 0xffff, roughly 29 in the SN range - the SID uses higher counter values for higher pitch, which is inverse to the SN) will be clipped to 0xffff. Preprocess with other tools for finer control.

```
D:>prepare4SID.exe ab_psg.vgz_ton00.60hz ab_psg.vgz_ton01.60hz
ab_psg.vgz_ton02.60hz ab.psgsid
VGMComp2 SID Prep Tool - v20200620

Opened SID channel 0: ab_psg.vgz_ton00.60hz
Opened SID channel 1: ab_psg.vgz_ton01.60hz
```

```
Opened SID channel 2: ab_psg.vgz_ton02.60hz
Imported 5042 rows
1160 notes clipped (lossy)
** DONE **
```

---

vgmcomp2 [-d] [-dd] [-v] [-minrun s,e] [-forcechan c] [-alwaysrle] [-norle] [-norle16] [-norle24] [-norle32] [-nofwd] [-nobwd] <-ay|-sn|-sid> <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, -sn or -sid is mandatory.
-sn - specify data is intended for SN PSG playback. This, -ay or -sid is mandatory.
-sid - specify data is intended for SID PSG playback. This, -sn 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.
-**forcechan c** - normally, channels that are mute for the entire song are dropped to save space and playback CPU, this will force it to be included anyway. This can be useful for the SN chip which uses channel 2 to control the frequency of the noise channel 3 - if it was muted it would otherwise be dropped. This may be repeated for multiple channels.

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, -sn** or **-sid** 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 or .sbfsid to keep track of it, but this is not enforced.

Current differences are summarized here:

	AY8910	**SN**	**SID**
Tone counter range	0x001 - 0xfff	0x001 - 0x3ff	0x0000-0xffff (inverted)
Tone volume levels	Absolute: 0 (mute), 0xF (loudest)	Attenuation: 0 (loudest), 0xF (mute)	Absolute: 0 (mute), 0xF (loudest) (linear)
Noise range	0x00 - 0x1f (counter)	0x00 - 0x0f (lookup table)	0x0000-0xffff (inverted)
Noise volume stream	Mixer command, shifted	Actual volume 0x00-0x0f	Noise channel and volume stream are not stored separately (pointer of 0x0000)
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.	Least significant byte for Freq Lo is first, most significant byte for Freq Hi is second.

Except for the note table, these differences all occur during the "prepare4XXX" step. In theory, it would be possible to create a single combined SBF file that had multiple chip types encoded within it. The tools do not currently support it, and may never do so. Because of the note table differences, the note channels would differ greatly between the different chip types, and it's unlikely there would be much compression gain.

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

```
Successful!

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

** DONE **
```

---

vgmcomp2 -pack <filenamein.bin> <filenameout.sbf>

This alternate form of the vgmcomp2 exe will pack any binary file as a single stream in compressed sbf form. The existing library can be used to unpack the data.

All the switches in the above documentation not relating to multiple streams or chip types can be used with this form as well.

See CPlayerTest.c in Players/CPlayer for an example of how to use it, but in short:

- Declare a STREAM object *(if you're clever, you might reuse one of the player's)*
- Initialize all members to 0, except for the mainPtr which should point to the base of the compressed data.
- If you are on Coleco, you can simply call getCompressedByte, passing pointers to the stream object and the base of the compressed data.
- On the TI, you will need to add the following wrapper function for the assembly interface:

```
uint8 __attribute__ ((noinline)) getCompressedByteWrap(STREAM *str, uint8 *buf) {
    __asm__(                                                        \
        "mov r1,r15\n\t"                                            \
        "dect r10\n\t"                                              \
        "mov r11,*r10\n\t"                                          \
        "li r6,>0100\n\t"                                           \
        "bl @getCompressedByte\n\t"                                 \
        "mov *r10+,r11\n"                                           \
        : /* no outputs */                                         \
        : /* no arguments */                                        \
        : "r1","r2","r3","r4","r5","r6","r7","r8","r9","r11","r15","cc"   \
        );
}
```

This will allow you to retrieve the data one byte at a time.

```
D:\>vgmcomp2 -pack chuckchars.TIAC chuckcol.sbf
VGMComp2 Compression Tool - v20210901

 - read 2064 bytes...
```

```
Binary compressed to 1740 bytes

** DONE **
```

---

bestpacker [-maxsize <n>] [-args "<args for vgmcomp2>"] filename1 filename2 [...]

Takes in the list of filenames, and tries them all against vgmcomp2 to find the best combination for sbf packing. The search is linear, not comprehensive, but it generally finds good matches. This can take a very long time, exponentially with the number of songs provided.

When finished, the program will output a list that can be copied to a batch file of commands, including remarks that have the output size of each file.

-maxsize <n> - specify the maximum size of the output sbf file - if not specified, 65536 is used. If a particular song can not fit in maxsize, then the entire process is aborted, so it's wise to test each song before starting.
-args "<args>" - pass here arguments for vgmcomp2, in quotation marks. Normally you will need at least "-sn" or "-ay" or "-sid". They are passed unmodified, so a typo here will cause errors.
-q - quiet verbose output
filename1 - (and filename2, and so on) pass here the processed PSG files to give to vgmcomp2. There is no upper limit

Note: vgmcomp2.exe *MUST* be in your executable path for this command to work.

Warning: vgmcomp2 is run creating a test output file of "dummyxyzx.xxx". You may delete this file when it is finished, but be aware that any existing file with this name will be overwritten.

```
D:\>path=%path%;d:\ti\vgmcomp2\

D:\>bestpacker -maxsize 8192 -args "-sn" ab_psg.psgsn splash30.psgsn
wb.psgsn
VGMComp2 'Best' Packing Tool - v20210704

Going to work with 3 filenames, maximum output size of 8192 bytes
Testing ab_psg.psgsn (with 2 files left)...
>> Execute vgmcomp2 -sn ab_psg.psgsn dummyxyzx.xxx
>> Execute vgmcomp2 -sn ab_psg.psgsn splash30.psgsn dummyxyzx.xxx
File splash30.psgsn reduced bestSize to 7666
>> Execute vgmcomp2 -sn ab_psg.psgsn wb.psgsn dummyxyzx.xxx
File wb.psgsn reduced bestSize to 6584
```

```
Placed wb.psgsn...
>> Execute vgmcomp2 -sn ab_psg.psgsn wb.psgsn splash30.psgsn
dummyxyzx.xxx
Finished command string 'vgmcomp2 -sn ab_psg.psgsn wb.psgsn' with
best size 6584 bytes
Testing splash30.psgsn (with 0 files left)...
>> Execute vgmcomp2 -sn splash30.psgsn dummyxyzx.xxx
Finished command string 'vgmcomp2 -sn splash30.psgsn' with best size
4736 bytes
* Search finished. 2 commandlines found.
@rem Output size: 6584 bytes
vgmcomp2 -sn ab_psg.psgsn wb.psgsn OutFile00.sbf
@rem Output size: 4736 bytes
vgmcomp2 -sn splash30.psgsn OutFile01.sbf

**DONE**
```

EXAMPLE

To convert a VGM file containing SN compatible music (such as from the Sega Master System) for SN output, with no editing or conversion, you would follow these steps:

vgm_psg2psg.exe ab_psg.vgz

- My example here is the file “ab_psg.vgz”, which is a VGM containing music from Afterburner on the Master System
- This will output 4 files, one for each channel. They will use the file name “ab_psg.vgz” as a prefix, and then append information about the channel, in particular “_ton00” for the first tone channel, “_ton01” and “_ton02” for the second and third tone channels, and “_noi03” to indicate the fourth channel is noise data. All files will have the extension “.60hz” to indicate they are intended to play at 60hz.

prepare4SN.exe ab_psg.vgz_ton00.60hz ab_psg.vgz_ton01.60hz ab_psg.vgz_ton02.60hz ab_psg.vgz_noi03.60hz ab_psg.psgsn

- This program reads in the processed (if any) psg files and combines them into a single text file containing all four channels. It also enforces any chip level restrictions on frequency or volume at this point - in short this output file is now SN-valid data
- I named my output file ab_psg.psgsn - I like to use “psgXX” to remember which chip the data is formatted for, but no syntax is enforced here.
- This particular file output a warning about “47 mutes mapped” - all this means is that there were channels currently muted, but they changed their frequency. To improve

compression, that change was discarded until it was needed, if ever. The report will tell you if any changes were ‘lossy’, meaning the output quality is affected. In all cases you can use other tools to control how those lossy values are handled - the prepare tool will use the most heavy handed mapping available to it.
- The output file is now ready to be compressed.

vgmcomp2.exe -sn ab_psg.psgsn ab_psg.sbf

- The -sn is passed to tell the compressor it is working on an SN file. It will do a final check for valid data but will just fail out if anything is not valid.
- We then pass the input file, and specify the binary output file. I use the extension .sbf, but this is not mandated.
- You can pass multiple input files to compress them into a single sound block. You should not mix chip types in the same compressed file.
- You will get some statistics about how well it compressed, as well as the cost in bytes per second to play back.

Also, when editing, remember that **testplayer.exe** can play can the **psg** files (just pass the prefix to auto-find them all), the **psgsn** file, and even the final **sbf**. You can use this both as a test, and especially as a tool while you are working to hear how it is doing. Testplayer can operate without restrictions, letting you listen to many more channels than the target supports - which is helpful when reducing channel count.