HuSound 1.2c

Macro instrument sound driver for the NEC PC-Engine / TurboGrafx-16
Care and Feeding Instructions

Written and Designed by: Osman Celimli

◊Table of Contents

►What is HuSound?

HuSound is a music and sound effects driver for NEC's PC-Engine / TurboGrafx-16 console based upon macro instruments. This allows the instruments used in a given composition to have all the complexities of sound effects as opposed to just ADSR envelopes. Internally, the driver treats both instruments and sound effects as a single entity, the only difference being that instruments are played with a frequency offset (i.e. their "note").

Other features include 6.992KHz PCM playback on any of the PC-Engine's six hardware channels and LFSR / Noise mode on channels five and six. Stereo panning and attenuation may be used to enhance both music tracks and sound effects during playback. The base driver tick is tied to the vertical blank interrupt (usually 60Hz). Also notable is that HuSound was written specifically for use with the MCGenjin memory mapper, natively supporting its extended data capacity and region switching capability.

Ville Helin's WLA-DX is the chosen assembler for HuSound and its included auditioning program, HuListen. However, you are free to modify HuSound for use with other development environments. Please note that the authors are not responsible for any personal distress, harm, or destruction of property while using HuSound; however unlikely they may be.

○MCGenjin Primer

MCGenjin is a PC-Engine / TurboGrafx-16 memory mapper designed to support dynamic region switching, extended game capacity (up to eight megabytes as of writing), and two user device selections for components such as additional work RAM. Its formal documentation may be found in the `.\HuListen\EQU_MCG.asm` file in this package. HuListen has been written to use this mapper which is available in hardware or via emulation (fully supported in Mednafen).

While this means that the HuSound libraries natively support several megabytes of audio data, there will be some new concepts for the programmer to use in his or her game software. The most important of which is the 4-Byte MCGenjin Address. These are used throughout the driver and its interfaces to the user to locate and access data, and are arranged as follows:

Byte	Equate	Purpose
0	LO	Low Address
1	HI	High Address
2	BANK	HuC6280 Bank Number
3	MCMAP	MCGenjin Page

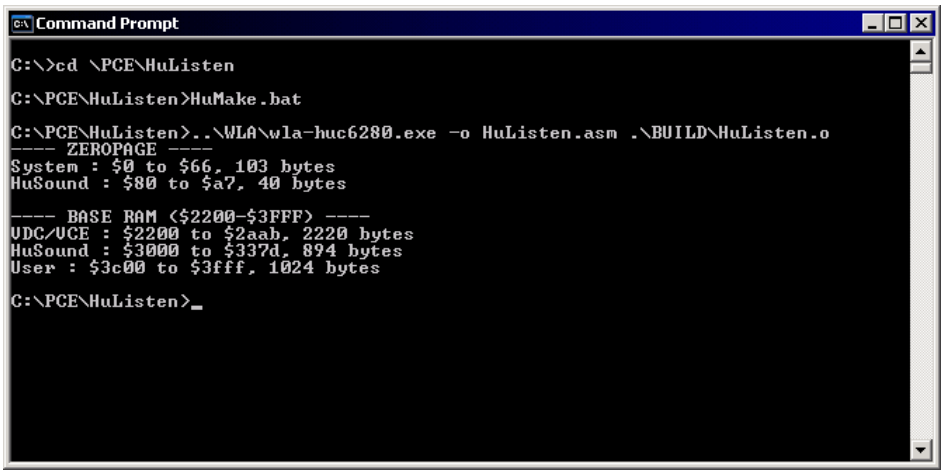
Where the LO and HI address specify the data's location after being mapped in to MPR4 ($8000 - $9FFF), the BANK specifies which 8KB HuC6280 bank the data are located in, and the MCMAP indicates which 256KB MCGenjin page the data are in. For example, if a piece of data were located at $42800 from the start of the cartridge it would be in MCMAP $01 ($42800 / 256KB), BANK $02 (($42800 % 256KB) / 8KB), and LO,HI $0800 ($42800 % 8KB).

○Setting Up Your Development Environment

The contents of this folder may be extracted wherever you like. For convenience I recommend a folder close to root, such as `C:\PCE\` or `C:\DEV\PCE`. You will need to supply a copy of `wla-huc6280.exe` and `wlalink.exe` in a `.\WLA` subfolder. After completing this step your development directory should have the following residents:

File / Folder	Description
.\HSCC\	HuSound Code Compiler (PCE-SASS)
.\HuListen\	HuListen Auditioning Suite
.\HuSound\	HuSound Driver
.\WLA\	WLA-DX HuC6280 Assembler and Linker
.\HuSound Programmer's Manual.pdf	This Document

You should now be able to build the included HuListen Auditioning Suite and its included demo sound effects and music. Navigate to the `.\HuListen` folder in Windows Command Prompt, run the `HuMake.bat` script, and you should be greeted with the following:



The newly built binary will be available in `.\HuListen\BUILD\HuListen.pce`, which may be written to an MCGenjin 4MB Plus Development Card or run in an emulator. The operating details of the HuListen Auditioning Suite are available on page 24.

○Using HuSound in Your Software

The most important aspect of a sound driver is *usually* generating sound. However, in order to get HuSound up and running with your current project you'll need to observe the following requirements:

- **Hardware**
  - **HuC6280 TIMER & TIMER IRQ**
    Required if the user wishes to play PCM Samples.

- **Required Definitions / Equates**
  - **HUSOUND_CHANNELS**
    How many channels (out of the six present in the PC-Engine hardware, starting from the first channel) that the HuSound Library will have control over. This should be set to **6** unless you'd like to merge HuSound with a driver of your own.
  - **HUSOUND_STACKLEN**
    The music and sound effect script decoder stack length. This indicates how many **CALL**s and **LOOP**s you can use in your scripts. A value of **4** is recommended.
  - **HUSOUND_REQLEN**
    Specifies the depth of the request queue for sound effects. Increasing this allows more sound effects to be queued up by the user in between driver ticks. A value of **8** is the default, but this can be lowered to **4** for most software.
  - **HUSOUND_ZP_BASE** and **HUSOUND_MEM_BASE**
    Base addresses for HuSound's variables in the ZeroPage and Work RAM respectively.
  - **.\HuListen\EQU_PCE.asm**
  - **.\HuListen\EQU_MCG.asm**
  - **.\HuListen\ZP_SYS.asm**
  - **.\HuListen\SUB\System***
    While HuSound is fairly modular, it is still dependent on many of the hardware definitions, macros, and system calls present in these files.

- **Included Files**

- **.\HuSound\HuSound.asm**
- **.\HuSound\HuSFX.asm**
- **.\HuSound\HuMusic.asm**
- **.\HuSound\HuSample.asm**
  Driver code and its subcomponents, these should be placed in a fixed bank.
- **.\HuSound\ZP_HuSound.asm**
- **.\HuSound\VAR_HuSound.asm**
  ZeroPage and Work RAM variable definitions.

- **Consumables / Sound Data**
  HuSound accepts four types of consumables: Sound Effects Packages, Music Tracks, Music Track Packages, and PCM Sample Packages. Any one of these may be omitted if they are not required by your project, and all of them excluding individual music tracks must be configured prior to initialization time. You may let the driver know of a consumable's existence by setting one of the variables below to a value other than zero:
  - **HuSFX_PackBase**
    4-Byte MCGenjin Address indicating the base of the Sound Effects Package to use. This has no alignment restrictions and the data may spread over several banks.
  - **HuMusic_PackBase**
    4-Byte MCGenjin Address indicating the base of the Music Track Package to use. This has no alignment restrictions and the data may spread over several banks.
  - **HuSample_DirBase**
    4-Byte MCGenjin Address indicating the starting address of the PCM Sample Package. This requires its first several bytes (encompassing the PCM directory) to be located within the same bank, so it is recommended just to make this resource bank-aligned.

- **Initializing and Running**
  After configuring your consumables as described above, you can initialize the driver by performing a `jsr` to **HuSound_Init**. This should be performed a single time before your vertical blank interrupt has been enabled and your main game routine has begun. Once you have configured the driver it may be run once per frame in your vertical blank interrupt handler by performing two subroutine calls:
  - **HuSound_Writeback**
    This performs all update writes to the PC-Engine's sound hardware. This should be called as early in the interrupt handler as possible in order to guarantee consistent timing.
  - **HuSound_Main**
    This performs all sound effects and music decoding and is not time-critical. Therefore it can be done after more important things such as data or palette transfers.

- **Commonly Used Subroutines**
  Once HuSound has been initialized and is executing once every vertical blank, the following subroutines will be useful to the programmer:
  - **HuSound_Pause**
  - **HuSound_UnPause**
    Pause and silence or unpause and resume all driver activity.
  - **HuSound_StopAll**
    Cease playback of any currently decoding music, sound effects, or samples.

  - **HuSFX_Play**
    Request playback of the sound effect whose number is in `A` with panning whose value is in `Y`.

- **HuSFX_Stop**
  Cease playback any sound effect whose priority is in A.
- **HuSFX_StopAll**
  Cease playback of any sound effects. Music and any currently active instruments will be left unaltered.

- **HuMusic_TrackReq**
  Request playback of the music track in the currently configured music track package whose number is in A. This should be the primary method of music playback used in most software.
- **HuMusic_Play**
  Request playback of the music track whose 4-Byte MCGenjin Address is in HuMusic_SongBase. This should only be used by software which requires dynamically received music data, such as a “download music from x” program.
- **HuMusic_Stop**
  Cease playback of any currently decoding music track.
- **HuMusic_ReqAtten**
  Request a change in the current music attenuation level supplied in A, this may range from 0 (no attenuation, loudest) to 15 (max attenuation, silent).

- **HuSample_PlaySimple**
  Play the PCM sample whose number is in A over the first audio channel.
- **HuSample_StopSimple**
  Cease playback of any sample whose playback was started using the routine above.

►Audio Channel Structure

The PC-Engine contains six hardware audio channels with *mostly identical* features. All of them may be used to playback unique 32 Sample @ 5-Bit waveforms or stream PCM Samples, the last two also feature an alternate LFSR / Noise generation mode, and the first two may be paired to generate simple frequency modulation (this is unsupported by HuSound as of writing). In short, any channel can play back a waveform or PCM Sample. But only channels `4` and `5` may be used to play white noise.

Channel	Waveform Mode	PCM Playback	LFSR / Noise	LFO
0	Y	Y	N	Y/Unsupported
1	Y	Y	N	Y/Unsupported
2	Y	Y	N	N
3	Y	Y	N	N
4	Y	Y	Y	N
5	Y	Y	Y	N

PC-Engine Channel Features

That hardware note aside, for all intents and purposes each channel has the properties listed below:

- **Priority**
  The current priority of the instrument or sound effect playing in the channel. When sound effects and instruments compete for a given audio channel, those with higher priorities win out. A priority of zero indicates the channel is currently free (no one is using it).
- **Mode**
  Indicates whether the channel is playing a waveform (`WAVE`), being used for PCM playback (`PCM`), or generating white noise (`NOISE`).
- **Frequency**
  A 12-Bit (`WAVE`) or 5-Bit (`NOISE`) divider value indicating what frequency the channel will be clocked at. PCM Samples have no inherent frequency and are fixed at a 6.992KHz playback rate. The final frequency is actually the sum of several sub-frequencies:
  - **Base**
    16.8-Bit precision value representing the current base frequency of the channel. This is ignored (zeroed) when playing a sound effect, but is used for the note frequency in instrument decoding.
  - **Pitch and Pitch Adjustment**
    16.8-Bit precision value representing the current offset from the base frequency and how much to adjust this offset each driver tick. This is ignored (zeroed) when playing a sound effect, but is used for detuning and pitch bends in instruments.
  - **Offset and Offset Adjustment**
    16.8-Bit precision value representing the offset from the sum of the base frequency and pitch. This is the sole frequency used by sound effects, but is used by notes for variation from their base frequency.

    Taking the above into account, we can calculate the divider value as follows:
    - **For Instruments**
      ```
      pitch += pitchAdjustment;
      offset += offsetAdjustment;
      divider = base + pitch + offset;
      ```
    - **For Sound Effects**
      ```
      offset += offsetAdjustment;
      divider = offset;
      ```

    The final clock rate may then be computed based upon the mode:
    - **For Instruments**
      ```
      frequency = 3.58MHz / divider;
      ```

    - **LFSR / Noise**
      `frequency = 3.58MHz / (64 x divider);`

- **Volume and Volume Adjustment**
  An 8.8 precision value of which 5-Bits are used to indicate the current volume for `WAVE` or `NOISE` mode. Each driver tick the volume adjustment value is added to the current volume value, allowing for simple amplitude modulation. Channels in `PCM` mode have a fixed volume at its maximum of `$1F` (`31`). Their amplitude may be controlled using the panning registers.
- **Panning**
  An 8-Bit value representing the panning of the channel. Bits 7-4 are the attenuation for the left speaker, and bits 3-0 are for the right speaker. `1111` is loudest, `0000` is off.
- **Writeback Disabling**
  HuSound will release control of any channel flagged as having writebacks disabled. Music and sound effects will continue to decode normally, but will not touch the actual audio hardware. This is useful for temporarily borrowing the sound hardware for other uses. For example, the PCM playback routines set a channel's writeback disable flag before taking it over.

►Instruments and Sound Effects

In HuSound, instruments and sound effects have been merged into a single entity. This allows the audio programmer to not only create elaborate instruments with all the features of sound effects, but also reduces driver size and complexity.

While the instruments and sound effect scripts are decoded identically, they are not stored in the same data structures. Sound effects live inside the Sound Effects Package, while the instruments for a particular music track are always stored along with it. The headers prefacing individual sound effects and instruments differ as well. A description of this formatting and encoding follows.

○Sound Effects Packages

The current sound effect package's base address is stored in `HuSFX_PackBase`, and must reside in ROM. A well behaved package will be formatted as shown below:

```
Offset    Contents
0         'S'
1         'F'
2         'X'
3         Number of Sound Effects
4+
 \
  0-3           Start Addresses (Relative Lo, Hi, Bank, Map)
(4 * (Number of Sound Effects + 1))
 \
  Sound Effects
```

○Sound Effect Headers

Sound effects may be composed of one to six channels of script data prefaced by a small header and descriptors for each of their channels. Since the PC-Engine does not have uniform capabilities across all its audio channels, each channel descriptor must include a "requirements" tag indicating whether or not the channel script will need `NOISE` mode.

```
Offset    Contents
0         Priority
1         Total Channels (1-6)
2+        Channel Script Descriptors:
 \
  0             Channel Requirements ($00 = WAVE / PCM, $FF = NOISE)
  1-4           Start Address (Relative Lo, Hi, Bank, Map)
(2 + (5 * Total Channels))
 \
  Sound Effect Data / Script
```

○Instrument Headers

Instruments are limited to only a single channel and thus discard the sound effect header shown above and instead start with a 4-Byte relative (to the instrument block base within a given music track) address of the instrument's note off portion of the script, which is then followed by the data / script itself.

○Instrument and Sound Effect Scripts

Following the instrument or sound effect's header are the scripts for each of its channels. This script is a simple bytecode with commands for adjusting the properties of the audio channel

and controlling the execution path of the sound script itself. The commands, their format, and operations are listed below:

- **`0: HUSFX_END()`**
  Close the sound effect decoding process and silence the channel.
- **`1: HUSFX_DELAY(Delay)`**
  Delay the decoding of the sound effect script by the given count in ticks.
- **`2: HUSFX_LOOP_START(Loop Count)`**
  Set the start of a looping region and define how many counts the loop will last. Negative loop counts are considered infinite.
- **`3: HUSFX_LOOP_END()`**
  Decrement the current loop count, branching back to the loop start if the loop count is greater than zero or negative (infinite).
- **`4: HUSFX_WAVEFORM(Waveform Data / 32 Bytes of 5-Bit uSamples)`**
  Put the channel into `WAVE` mode and assign it the 32-Bytes of 5-Bit unsigned waveform data included in the stream.
- **`5: HUSFX_SAMPLE(Sample Number)`**
  Put the channel into `PCM` mode and start playback of the specified sample.
- **`6: HUSFX_LFSR()`**
  Put the channel in NOISE mode. If this functionality is not supported on the currently decoding channel, the command will be ignored.
- **`7: HUSFX_FREQUENCY(Frequency Lo, Hi, Adjustment Lo, Hi, De)`**
  Set the frequency offset and frequency offset adjustment for this channel. The decimal of the frequency offset is cleared.
- **`8: HUSFX_VOLUME(Volume, Volume Adjustment Lo, De)`**
  Set the volume and volume adjustment for this channel.

►Music

Music, like sound effects and instruments, uses a simple scripting language. Instruments may be played for various lengths of time with specified frequency offsets, pitch slides, and panning. Looping is also supported, and as with sound effects, the maximum loop depth is configurable by adjusting `HUSOUND_STACKEN`. Infinite loops as applicable in sound effects are identical in music scripts. Script patterns/subroutines may also be called and returned from, with the same depth limit as loops. PCM sample cues may be used in any channel, allowing playback of 6.992KHz samples in time with music.

As the PC-Engine has six audio channels, each music script is composed of up to six voices, all of which have a priority which is adjustable over the course of the music script's life. This allows tracks of music to become more or less important than the sound effects competing with them for the hardware. By specifying a priority of zero, a music script is stopped, so a piece of music which does not need all six channels may use less.

Two data structures are used to store music: songs and track packages (collections of songs). While playback of individual songs is possible, it is recommended that all songs be combined into a track package. The instruments for a particular music track are packed along with it. While the music data format allows the instruments to be located virtually anywhere, the included SASS compiler (HSCC) places them at the end of the music track.

○Track Packages

The current track package's base address is stored in `HuMusic_PackBase`, and must reside in ROM. A well behaved track package will be formatted as shown below:

```
Offset    Contents
0         'T'
1         'R'
2         'K'
3         Number of Songs / Tracks
4+
 \
  0-3           Song Start Addresses (Relative Lo, Hi, Bank, Map)
(4 * (NumTracks + 1))+
 \
  Song Data
```

○Individual Tracks / Songs

The current song's base address is stored in `HuMusic_SongBase`, which will automatically change when using the track package playback functions. Note that while Track Packages (and their included songs) are required to fall in the MCGenjin swappable ROM region, individual songs may be placed in memory arbitrarily. Like a track package, a song also begins with a header whose format is given below:

```
Offset    Contents
0         'M'
1         'U'
2         'S'
3         Number of Instrument Definitions
4-7       Address of Instrument Table (Relative Lo, Hi, Bank, Map)
8+        Channel Script Descriptors:
 \
  0             "C"
  1-4           Start Address (Relative Lo, Hi, Bank, Map)
  5             Initial Priority (0 if the channel is unused)
(8 + (6 * 6))+
```

```
    \
     Music Data
```

○Music Script

Following a music track's header are its music scripts, of which there may be up to six. The scripting commands are made up of one byte blocks and are similar to those of sound effects. The commands, their format, and descriptions of their operations are listed below:

- **`0: HUMUSIC_SET_PRIORITY(Priority)`**
  Set the music track's current playback priority. Setting the priority to zero will terminate the decode process.
- **`1: HUMUSIC_SET_PANNING(Panning)`**
  Set the music track's current stereo panning. The one byte panning value is in the format `%LLLLRRRR`, where larger values indicate greater amplitude. So a pan of `$F0` would yield the loudest volume through the left speaker.
- **`2: HUMUSIC_DELAY(Delay)`**
  Delay the track's decoding for the specified count in ticks.
- **`3: HUMUSIC_LOOP_START(Loop Count)`**
  Set the start of a looping region and define how many counts the loop will last. Negative loop counts are considered infinite.
- **`4: HUMUSIC_LOOP_END()`**
  Decrement the current loop count, branching back to the loop start if the loop count is greater than zero or negative (infinite).
- **`5: HUMUSIC_CALL(Dest Lo, Dest Hi, Dest Bank, Dest Map)`**
  Call ("`jsr`") the music script at the supplied relative address. The target address is equal to Dest + Song Base Address.
- **`6: HUMUSIC_RETURN()`**
  Return ("`rts`") from the current music script position to the command after the last CALL performed.
- **`7: HUMUSIC_BREAK()`**
  Return ("`rts`") all channels (not just the current one) to the lowest return address found in their pattern CALL stack, effectively bringing all channels back to their "main" script.
- **`8: HUMUSIC_PITCH(Pitch Lo, Hi, Pitch Adjust Lo, Hi, De)`**
  Set the current pitch offset and adjustment rate.
- **`9: HUMUSIC_NOTE_ON(Instrument, Frequency Lo, Hi)`**
  Start playing the specified intstrument at the given base frequency.
- **`10: HUMUSIC_NOTE_OFF()`**
  Have the currently playing instrument (if any) move to the note off portion of its script.
- **`11: HUMUSIC_SAMPLE(Sample Number)`**
  Start playing the specified PCM sample.

►PCM Samples

HuSound supports playback of up to six PCM Samples (one for each of the hardware channels) simultaneously through requests from sound effect scripts or music scripts. Playback in the first channel may also be accomplished through a call to `HuSample_PlaySimple`, with the sample number in A. Samples are required to be stored as 5-Bit Unsigned Monophonic PCM @ 6.992KHz, and are kept together in a PCM Sample Package. This begins with a sample directory which is arranged in the following format:

```
Offset    Contents
0         'P'
1         'C'
2         'M'
3         Number of Entries
4+        8-Byte PCM Descriptors:
 \
  0             "X"
  1-4           Start Address (Relative Lo, Hi, Bank, Map)
  5-7           Stream length (Lo, Mid, Hi)
```

The directory must not cross a bank boundary (at its max size of 256 entries, the directory will be 2052 bytes), and it's recommended to just keep it bank aligned. PCM streams following the directory have no alignment or boundary restrictions, but must not exceed a maximum size of 8MB.

►HuSound Code Compiler (HSCC)

The HuSound Code Compiler (HSCC) generates both Sound Effect Packages and Music Tracks from textual source files written in a music-oriented programming language called Sound ASSembler (SASS). Two syntaxes are used, one for instrument and sound effect definitions, the other for music tracks. HSCC can also organize several Music Tracks into Track Packages and create PCM Sample Packages from 8-Bit Unsigned Monophonic Samples @ 6.992KHz.

HSCC is a commandline operation and can operate in four modes: Sound Effects, Music, Track Package, and PCM. Executing HSCC with no parameters will yield the following response:

```
C:\PCE\HSCC>hscc
Usage: hscc [-sfx/-mus/-pak/-pcm] outfile [source data]

SFX:   hscc -sfx output.sfx scripts.txt
       An equates file with the names and numbers of your sound
       effects will be created in the same directory as your output
       file and may be included in a WLA-DX project.

MUSIC: hscc -mus output.mus instruments.txt channel_0.txt
              [channel_1.txt] [...]
       Note that the argument numbers do correspond to the
       respective channels on the PC-Engine.

PACK:  hscc -pak output.trk song_0.mus [song_1.mus] [...]
       A maximum of 255 songs may be included in one track pack.

PCM:   hscc -pcm output.pcm sample_0.raw [sample_1.raw] [...]
       Each included file must be an 8-Bit unsigned PCM stream.
```

In Sound Effects mode, HSCC requires two filenames: the first of which indicates where the generated Sound Effects Package will be stored and the second of which is the file containing the SASS Sound Effects Scripts to parse. A third file will be generated automatically in the same directory as your output file which contains equates for all the sound effects and their priorities.

Building the included example sound effects file can be accomplished as follows:

```
C:\PCE\HuListen\SASS>..\..\HSCC\HSCC -sfx SFXPAK.bin TEST_SFX.txt
SFX Mode...
A total of 7 SFX were found
The current SFX binary is : 804 bytes
Saving SFX binary to : SFXPAK.bin
Saving SFX equates to : SFXPAK.equ
```

In Music mode, HSCC requires anywhere from three to eight arguments, the first of which indicates where the generated Music Track will be stored, the second of which is the file containing the SASS Instrument Scripts to parse, and the third through eighth contains the SASS Music Scripts to parse for each of the six channels on the PC-Engine. Any channel other than the first is optional, and if a piece of music requires less than the total of six hardware channels, the respective files may be omitted from the argument list.

Building the included example music file can be accomplished as follows:

```
C:\PCE\HuListen\SASS\DEMO_MUS>..\..\..\HSCC\HSCC -mus DEMO_MUS.mus Instruments.txt
0_LEADHI.txt 1_LEADLO.txt 2_BACK.txt 3_BASS.txt 4_PERCHI.txt 5_PERCLO.txt
MUS Mode...
A total of 11 instruments were found
The current instrument table is : 817 bytes
The current MUS binary is : 1177 bytes
Saving music binary to : DEMO_MUS.mus
```

In Track Package Mode, HSCC requires two or more arguments, the first of which indicates where the generated Track Package will be stored, and the second and above argument(s) are the Music Tracks to add to the Track Package.

Creating a Track Package which includes the HuSound Driver and HSCC Compiler test track (`.\HuListen\SASS\TEST_MUS\TEST_MUS.mus`) and the HuSound Demo Track (`.\HuListen\SASS\DEMO_MUS\DEMO_MUS.mus`) can be accomplished as follows:

```
C:\PCE\HuListen\SASS>..\..\HSCC\HSCC -pak MUSPAK.bin .\TEST_MUS\TEST_MUS.mus
.\DEMO_MUS\DEMO_MUS.mus
The current PAK binary is : 1816 bytes
Saving track package to : MUSPAK.bin
```

In PCM Mode, HSCC requires two or more arguments, the first of which indicates where the generated PCM Sample Package will be stored, and the second and above argument(s) indicate which PCM Samples to add to the package. The supplied PCM Samples must be in RAW 8-Bit Unsigned Monophonic PCM @ 6.992KHz. HSCC will automatically convert these to 5-Bit granularity before inclusion in the PCM Sample Package.

Creating a PCM Sample Package using various RAW PCM Samples (not included) may be accomplished as follows:

```
C:\PCE\HuListen\SASS>..\..\HSCC\HSCC -pcm PCMPAK.bin ..\..\Samples\RZ1_Kick.raw
..\..\Samples\RZ1_Snare.raw ..\..\Samples\RZ1_TomH.raw ..\..\Samples\RZ1_TomM.raw
..\..\Samples\RZ1_TomL.raw ..\..\Samples\RAP_Snare.raw ..\..\Samples\PS2_Snare.raw
..\..\Samples\S3_Snare.raw
PCM Mode...
The current PCM binary is : 12346 bytes / 1.765732 seconds
Saving PCM binary to : PCMPAK.bin
```

►PCE-SASS Music-Oriented Language

The HuSound Code Compiler (HSCC) uses an extension of the SASS language called PCE-SASS, which caters to the unique features of the PC-Engine sound hardware and the HuSound driver. A reference to the general SASS language and its PCE-SASS extensions is provided below:

○Constants

Values may be entered in decimal, hexadecimal, and binary using the following syntax:

```
        32              ; Decimal (No prefix)
        -32             ; Negative Decimal (Leading "-")
        $20             ; Hexadecimal (Leading "$")
        %100000         ; Binary (Leading "%")
```

As HuSound uses fractional-integer values (i.e. 16.8 or 8.8 precision math), a decimal may be assigned to each value as follows:

```
        32.16           ; Decimal
        -32.16          ; Negative Decimal
        $20.10          ; Hexadecimal
        %100000.10000   ; Binary
```

The format of the decimal matches that of its leading integer. Values written without a decimal are assumed to have one with value zero.

○Comments

Follow a semicolon ";" and span a single line:

```
        ; This is a comment, it will span the whole line
        as4 60 ; Comments may also follow commands and values
```

○Timing

All values quantifying time (note on, rests, waits, etc.) are given in 60Hz HuSound driver ticks. Using this detail we can write the note sequence below:

```
        f.3 60          ; Play an F in the third octave for one second
        rest 20         ; Note off and rest for 1/3 second
        as5 40          ; Play an A sharp in the fifth octave for 2/3 second
```

○Instruments and Sound Effects

As the decoding of instruments and sound effects is identical in HuSound, their definitions are also similar. Instruments are expected to be used in music tracks under the control of the music playback routine. Sound effects are available for playback at any time by the user or game routine. Therefore **priorities** are exclusive to sound effects, while **note offs** and **tuning** are only available to instruments.

```
name priority                                   ; <- Label
        volume value                            ; <- Header / Init
        frequency value
        wave waveform
        noise
        sample number
        tuning value
        {
                rest ticks                      ; <- Body

                volume value
                frequency value
                loop count
                        ; (Loop Body)
                endloop

                noteoff
                end
        }
```

All definitions begin with a **label**, which contains its **name** and **priority** (if a sound effect). **Names** are composed of one or more ASCII characters (such as "square," "explosion," or "powpow"), are used to identify instruments inside of music scripts, and generate equates for sound effects. **Priorities** are used only for sound effects and represent the importance of the particular sound effect relative to both the music tracks and other sound effects. For instrument definitions, the priority should be excluded.

Following the label is the definition's **header data**. This specifies the **initial state** of the instrument or sound effect through various **parameter adjustment** commands. Some common parameter adjustment commands are **volume**, **frequency**, and **tuning**. **Tuning** is a special case, as it is only used for instruments, and ignored for sound effects.

The remaining portion of the definition is the **body**, which specifies how the sound will change over time and is composed of one or more **note**, **flow control**, and **parameter adjustment** commands within **{** Curly Braces **}**. Most of these are available for both instruments and sound effects, but **noteoff** only applies to instruments, and will have no effect on sound effects.

As single-channel sound effects can quickly become a limiting factor in game audio fidelity, HuSound supports the definition of multi-channel sound effects. These resemble a normal sound effect definition, except with a **{** Curly Brace **}** region immediately after the label, containing one or more channel headers and bodies:

```
name priority                                   ; <- Label
{
        volume value [adjustment]               ; <- 1st Channel Header / Init
        frequency value  [adjustment]
        noise
        {
                rest ticks                      ; <- 1st Channel Body
                loop count
                        ; (Loop Body)
                endloop
                end
        }

        volume value [adjustment]               ; <- 2nd Channel Header / Init
        frequency value [adjustment]
        wave waveform
        {
                rest ticks                      ; <- 2nd Channel Body
                end
        }
}
```

Commands may be divided into three categories: **note**, **parameter adjustment**, and **flow control**. **Note** commands control delays and note off actions. **Parameter adjustments** control how the instrument or sound effect will actually create its sound, these are also the only commands which may be used in the channel header area. **Flow control** commands include loops and end markers. A basic command listing follows:

- **noteoff,n** : note off
  Applicable only to instruments. Flags the portion instrument body following it as the note off (or "release") part of the script. If an instrument is playing and the music script encounters a rest command, this area of the instrument body will begin decoding on the next update cycle.

```
        ...             ; When a rest command is encountered in
        noteoff         ; the music script while an instrument
        rest 8          ; is playing, the instrument will start
        end             ; executing the portion of its script
        ...             ; directly following the noteoff command.
```

- **rest,r ticks** : rest
- **wait,w ticks** : wait
  Pause for the specified number of ticks before advancing to the next command. The instrument or sound effect will continue to play during this time.

```
        rest 14         ; Wait 14 ticks
        end             ; Stop
```

- **volume,v value [adjustment]**: set the current volume
  Set the channel volume to a value of 0-31, where larger values are louder. The adjustment field is an optional 8.8 precision value which will be added to the current volume every driver tick.

```
        volume 26 -2    ; Set amplitude to 26/31, -2 each tick
        rest 12         ; Wait 12 ticks, amplitude will be near zero
        end             ; Stop
```

- **frequency,f value [adjustment]**: set the current frequency offset
  Set the channel frequency offset, where larger values yield a larger divider (lower frequency). The adjustment field is an optional 16.8 precision value which will be added to the current frequency offset every driver tick.

```
        frequency 64 8 ; Set frequency offset to 64, +8 each tick
        rest 16        ; Wait 16 ticks
        end            ; Stop
```

- **wave,wv waveform**: switch to `WAVE` mode, using the supplied waveform.
  Places the channel into `WAVE` mode and uses the 32 supplied values each ranging from 0-31 as the new waveform to be played.

```
...
wave 16,19,22,24,27,29,30,30,31,30,30,29,27,24,22,19,16,13,10,8,5,3,1,1,0,1,1,3,5,8,10,13
...                             ; ^ Set the current waveform to a sinusoid
```

- **noise,no**: switch to `NOISE` mode.
  Places the channel into `NOISE` mode.

```
        ...
        noise
        ...
```

- **sample,s number**: switch to `PCM` mode, playing the specified sample.
  Place the channel into `PCM` mode and begin playback of the specified PCM Sample in the current PCM Sample Pack. After the sample has finished playing, the channel will return to its previous mode.

```
        sample 2        ; Start playing sample #2
        rest 5          ; Wait a bit for it to finish
        end             ; Stop
```

- **tuning,t value**: specify instrument tuning
  Instrument-only command specifying the tuning of the current instrument when playing notes. The value corresponds to the period of the currently assigned waveform. For example, a 32-Sample long sawtooth waveform would require a tuning of `32.0`. However, a waveform with the first 16-Samples representing a square pulse and the next 16 samples a sine would have a tuning of `16.0` (since the value reflects waveform *period*, not total sample count). The instrument may be detuned by using decimal values such as `31.6` or `8.2`.

```
...                             ; The configured sinusoid has a period of 32 Samples...
wave 16,19,22,24,27,29,30,30,31,30,30,29,27,24,22,19,16,13,10,8,5,3,1,1,0,1,1,3,5,8,10,13
tuning 32.0                     ; Giving it a period of 32.0
{
        ...
```

- **loop,l count** : start of loop
  Specifies the start of a loop, and the number of times it will repeat. If negative values are used, the loop will continue indefinitely.

```
        ...
        loop -1                 ; Repeat outer loop forever
                loop 10         ; Inner loop 10 times
                        rest 6
                endloop
        endloop
        end                     ; Stop
```

- **endloop,el** : loop end
  Specifies the end of a given loop.

```
        ...
        loop -1                 ; Repeat forever
                rest 18
        endloop                 ; Marks end of above loop
        end                     ; Stop
```

- **end,e** : sound end
  Stops decoding of the instrument or sound effect when reached, and frees the given channel**.**

```
        ...
        end                     ; Stop
```

○Music Tracks

Music tracks are composed of one or more script blocks which contain commands representing how and when to play instruments, samples, or sound effects. A listing of the format structure is shown below:

```
name priority                                   ; <- Main Block
{
        using instrument
        sample number
        priority value
        pan value

        call name
        loop count
                ; (Loop Body)
        endloop

        rest ticks
        as3 ticks
        wait ticks
        end
}

name                                            ; <- Sub Block
{
        ...
        return  / break
}
```

All script blocks start with a **label**, which at minimum contain the block's **name**. Each music track must contain a single **main block** which is the first to be played and also contains a starting **priority** in its **label**. **Names** are composed of one or more ASCII characters and **priorities** are an integer value, with higher values indicating a more important music track. Using a priority of zero will cause the music track to never decode.

Following the **label**, and enclosed between two **{** Curly Braces **}** are one or more commands composing the **block body**. The basic command set may be divided into

three categories: **notes**, **parameter adjustment**, and **flow control**. **Note** commands specify when to turn on and off a particular note (ala key on and key off). **Parameter adjustment** commands allow control over which macro instrument will be used and how it will be played. **Flow control** commands change how the SASS script will decode, allowing for loops, calls to different music blocks, and termination of playback.

A basic command listing follows:

- **Note Commands**
  A key on event at a given note may be specified with a three letter note command, followed by a duration in driver ticks. The format is as follows:
  (Note Letter)(Natural, Sharp, or Flat)(Octave Number)
  Note letters may be **c**, **b**, **d**, **e**, **f**, **g**, and **a**. Natural, Sharp, and Flat for the given note may be specified using "**.**" (period), "**s**", and "**b**" respectively. The octave number may range from **0-9**.

```
...
c.4 60          ; C-Natural 4th octave for 60 ticks
rest 10
bs4 20          ; B-Sharp 4th octave for 20 ticks
rest 10
bb2 80          ; B-Flat 2nd octave for 80 ticks
rest 10
...
```

- **sfx,s ticks [base]** : play instrument as sound effect
  Begins playback of the current instrument as a sound effect with optional frequency base (set to zero if excluded). This is useful for percussion.

```
...
loop 9
        using HiHat_Closed
        sfx 6
        sfx 6
        using HiHat_Open
        sfx 12
endloop
...
```

- **pitch,pt offset [adjustment]** : set pitch offset and adjustment
  Set the current pitch offset and its optional adjustment each driver tick.

```
...
as3 5           ; Start playing a note, wait five ticks
pitch 0 12      ; DECREASE the frequency 12 counts / tick
wait 20         ; After 20 ticks, it will be offset by
pitch 240 0     ; 240 counts, hold it there...
wait 5
...
```

- **rest,r ticks** : rest
  Acts as a key off event if following a note command, or a general delay if used on its own.

```
...
as4 20
rest 30         ; Key off, wait for 30 ticks
...
rest 60         ; Wait for 60 ticks
...
```

- **wait,w ticks** : wait
  Pauses decoding for the specified number of ticks. Useful for delays where note off behavior is not desired.

```
...
wait 60         ; Wait for 60 ticks
...
```

- **using,u instrument** : set current instrument
  Select the instrument used for playback in the music track.

```
...
using piano     ; Using instrument "piano"
g.3 20          ; The following notes will be played with
rest 10         ; "piano," with each note request
b.3 20          ; specifying a key on, and each rest
rest 10         ; indicating a key off.
a.3 20
rest 10
...
```

- **pan,p value** : set current panning
  An 8-Bit value representing the panning of the channel. Bits 7-4 are the attenuation for the left speaker, and bits 3-0 are for the right speaker. `1111` is loudest, `0000` is off.

```
...
using piano     ; Using instrument "piano"
pan $f0         ; Set panning to full volume on the LEFT
as3 12          ; Play the note in the LEFT speaker
rest 24
pan $0f         ; Set panning to full volume on the RIGHT
as3 12          ; Play the note in the RIGHT speaker
rest 24
...
```

- **sample,s number** : play the specified PCM sample
  Place the channel into `PCM` mode and begin playback of the specified PCM Sample in the current PCM Sample Pack. After the sample has finished playing, the channel will return to its previous mode.

```
...
sample 0
wait 24
sample 0
wait 24
sample 0
wait 24
...
```

- **priority,pr value** : set track priority
  Adjust the current playback priority of the music track. This may be useful to give certain parts of a piece more or less importance relative to sound effects.

```
...
priority 120    ; Set priority to 120
...
priority 240    ; Set priority to 240 (higher)
...
```

- **loop,l count** : start of loop
  Specifies the start of a loop, and the number of times it will repeat. If negative values are used, the loop will continue indefinitely.

```
        loop -1                 ; Repeat outer loop forever
                loop 10         ; Repeat inner loop 10 times
                        as4 20
                        rest 20
                endloop
                gs3 30
                rest 20
        endloop
        end                     ; Stop
```

- **endloop,el** : loop end
  Specifies the end of a given loop.

```
        loop -1                 ; Repeat forever
                fs4 20
                rest 30
                call drumSolo
        endloop                 ; Marks end of above loop
        end                     ; Stop
```

- **call,c blockLabel** : call script block
  Begin decoding a given script block with the name specified in **blockLabel**.

```
        ...
        call pianoSolo          ; Call the script block below
        ...
}

pianoSolo
{
        ...
        return
}
```

- **return,rt** : return from called script block
  Resume decoding from where a given script block was called.

```
        ...
        call pianoSolo
        ...
}

pianoSolo
{
        ...
        return          ; Resume decoding after "call pianoSolo"
}
```

- **break,b** : pattern break
  Returns all tracks (not just the one encountering the command) to the lowest entry in their CALL stack. Effectively, all tracks will be returned to the "main" script if they are not there already.

```
mainTrack 120
{
        ...
        call pianoSolo
        ...
}

pianoSolo
{
        ...
        call drumminThang
}

drumminThang
```

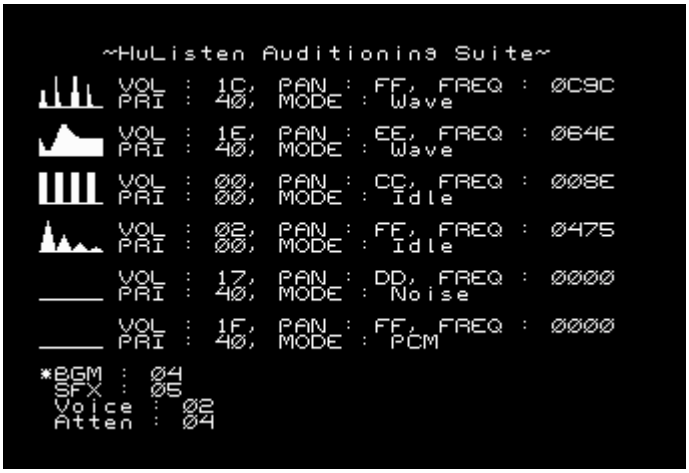
```
{
        ...
        break           ; Resumes decoding after "call pianoSolo"
}
```

- **end,e**
  Stops decoding of the music track and frees the channel

```
        ...
        end             ; Stop
```

►HuListen Auditioning Suite

HuListen is a simple PC-Engine / TurboGrafx-16 program for testing composed Music, Sound Effects, and PCM Samples. The state of the audio hardware registers and internal HuSound variables are displayed in order to assist with SASS Script debugging. Simply swap out the Sound Effect Package (`.\HuListen\SASS\SFXPAK.bin`), Music Track Package (`.\HuListen\SASS\MUSPAK.bin`), or PCM Sample Package (`.\HuListen\SASS\PCMPAK.bin`) with your own and rebuild using (`.\HuListen\HuMake.bat`) to quickly test your new sounds.



HuSound playing some sound effects and music in HuListen

Controls:

- **Up/Down**: Select BGM / SFX / Voice
- **Left/Right** : Select Item
- **I** : Play Item
- **II** : Stop Item

Register Guide:

- **VOL** : Channel Volume
- **PAN** : L/R Attenuation.
- **FREQ** : Frequency Divider
- **PRI** : Priority
- **MODE** : Wave / Noise / PCM / Idle Mode Indicator