HandyMusic 1.40cx+

Macro instrument sound driver for the Atari Lynx
Care and Feeding Instructions

Written and Designed by: Osman Celimli

Table of Contents

What is HandyMusic?

HandyMusic is a music and sound effects driver for Atari's Lynx console which is 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"). Single channel 8KHz PCM Playback is also supported, and samples are streamed from the cartridge to keep a small memory footprint. Stereo panning and attenuation as featured in the later Lynx revisions are also supported in music tracks, but sound effects are always played through both speakers.

Bastian Schick's BLL kit is required to use HandyMusic, although you are free to modify HandyMusic for other environments (or other platforms). The contents of this development package should be directly extracted to BLL's C:\Lynx folder.

HandyMusic was designed specifically for the Lynx and its limited memory, and thus only uses about 1.7KB of RAM for the sound driver itself, which is expected to reside in $A000-$A6AF. Operation is fairly simple and after initialization HandyMusic should be called once every VBlank through a JSR to "HandyMusic_Main." All decoding of the music script and sound effect envelope updates are performed during this call. Thus, all timing in HandyMusic is based upon a 60Hz tick. Greater precision may be obtained by tying the driver to a faster timer base (such as 120Hz or 240Hz), but this is not recommended due to the increased CPU overhead.

HandyMusic is completely free to use and modify in your own projects. However, the authors are in no way responsible for any personal distress, harm, or destruction of property while using HandyMusic; however unlikely it may be.

- ## Memory Usage:

  HandyMusic in its entirety is designed to occupy $A000-$BFFF on the Lynx, and is divided into three parts: The driver itself, a sound effects block, and the music data. They should all be allocated in the following fashion:

  - **HandyMusic Driver: $A000-$A6AF**
    The sound driver, composed of the music, sound effect, and sample playback code.
  - **Sound Effects Block**
    This may be located anywhere in memory by setting the appropriate variables (HandyMusic_SFX_AddressTableLoLo/LoHi/HiLo/LoLo, HandyMusic_SFX_AddressTablePriLo/Hi) in HandyMusic. However, it is most convenient to place it in the **$A6B0-$AFFF** range right after the driver.
  - **Music Block: $B000-$BFFF**
    The current music track, only one is kept in memory at a time.

- ## Hardware Dependencies:

  In addition to controlling the Lynx's audio hardware, HandyMusic requires use of the following components.

  - **Timer 3 Hardware**
    Is used for PCM Playback, but it can be used by the game software for other purposes if HandyMusic's PCM playback routine is inactive.
  - **Cartridge Hardware**
    Is used to load new music tracks and stream PCM. Loading a new music track will effectively lock the cartridge until the data has been copied, but as PCM streams from it, ensure "Sample_Playing" is zero before using the cartridge hardware.

- ## Required Pre-Initialization Variables:

  These variables must be set by the programmer prior to calling HandyMusic_Init in order to ensure proper operation of the driver.

  - **HandyMusic_SFX_AddressTableLoLo**
    The low byte of the SFX low address table.
  - **HandyMusic_SFX_AddressTableLoHi**
    The high byte of the SFX low address table.
  - **HandyMusic_SFX_AddressTableHiLo**
    The low byte of the SFX high address table.
  - **HandyMusic_SFX_AddressTableHiHi**
    The high byte of the SFX high address table.
  - **HandyMusic_SFX_AddressTablePriLo**
    The low byte of the SFX priority table.
  - **HandyMusic_SFX_AddressTablePriHi**
    The high byte of the SFX priority table.

- ## Required Constants:

  These constants must be defined prior to assembling HandyMusic.

  - **FileNum_MusicBase**
    The file number of the first music track stored on the cartridge (all music files are assumed to be located sequentially).
  - **FileNum_SampleBase**
    The file number of the first PCM Sample on the cartridge (all samples are assumed to be located sequentially).

- ## Commonly Used Subroutines:

  The subroutines most useful to the game programmer are listed below along with details of their operation.

  - **HandyMusic_Init**
    The initialization routine, should be called once at system startup after setting the variables detailed above.
  - **HandyMusic_Main**
    The entry point for HandyMusic's processing. Call every VBlank.
  - **HandyMusic_PlaySFX**
    Play the sound effect whose number is stored in A in the first available channel, scanning from 3 to 0. If no free channels are found, instruments or sound effects with a lower priority than the requested sound effect will be replaced. If no such instruments or sound effects are found, the request will be ignored. Only one sound effect may be requested per frame, in the case of multiple requests the sound effect with the highest priority will win out.
  - **HandyMusic_StopSoundEffect**
    Stops the first located instrument or sound effect whose priority is in A, the scan direction is from channel 3 to 0.
  - **HandyMusic_PlayMusic**
    Stops any currently playing music and begins playing the music block loaded into $B000-$BFFF.
  - **HandyMusic_StopMusic**
    Stops the currently playing music track.
  - **HandyMusic_StopAll**
    Stops all currently playing music and sound effects.
  - **HandyMusic_Pause**
    Pauses HandyMusic and silences all channels (such as when a game is paused). Music decoding is disabled and the driver will ignore any sound effect requests made during this period.
  - **HandyMusic_UnPause**

Restores HandyMusic from a paused state, resuming any previously playing music or sound effects

- **HandyMusic_LoadPlayBGM**
  Stops playback of the current music track, and loads another track from the cartridge whose number is in A, then begins playback. This routine is “PCM Safe” in that it will wait for the current PCM Sample to finish streaming before it loads the new music track from the cartridge.
- **PlayPCMSample**
  Starts streaming the PCM Sample to Channel 0 whose sample number is in A. Any currently playing music or SFX in the channel will be silenced. Requesting another sample to be played while one is already streaming will replace the previously playing sample with the new one.

Audio Channel Structure

The Lynx contains four identical audio channels which vaguely resemble its eight hardware timers. A timer is used to clock a polynomial counter, which drives the waveform generation hardware. There is no envelope hardware whatsoever in the Lynx, and HandyMusic generates all envelopes in software. However, for all intents and purposes, and because of the way HandyMusic handles the abstraction of the Lynx's audio hardware, each channel has the following properties:

- **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). **DO NOT USE THE SAME PRIORITIES FOR INSTRUMENTS AND SOUND EFFECTS.**
- **Base Frequency:** A 10-bit value representing the current base frequency of the channel. This is set to zero when playing a sound effect, but is used for the note frequency on instruments. This is actually stored in 16.8 precision, but the bottom eight and top six bits are ignored by the hardware and used only to simplify calculations. Each frame the Base Pitch Adjust gets added to the current Base Frequency, with the new value overwriting the old.
- **Base Pitch Adjust:** A 10-bit value which is added to the Base Frequency every audio frame, useful for things like pitch slides in music tracks. This is actually stored in 16.8 precision, but the bottom eight and top six bits are ignored by the hardware and used only to simplify calculations. Like the base frequency, this is note exclusive and is set to zero when playing a sound effect.
- **Frequency Offset:** A 10-bit value used as the offset from the base frequency. This is the sole frequency used by sound effects, but is used by notes for variation from their base frequency. This is actually stored in 16.8 precision, but the bottom eight and top six bits are ignored by the hardware and used only to simplify calculations. Each frame the Frequency Offset Pitch Adjust gets added to this value, with the new value replacing the old. Thus the actual timer write used to drive the polynomial counter at any given audio frame is:

(Frequency Offset += Frequency Offset Pitch Adjust) + (Base Frequency += Base Pitch Adjust)

Where this is a 10-bit result, bits 9-7 are used as the prescale value, and 6-0 are used as the divider value. Note that HandyMusic treats the prescale and divider values a differently than the hardware, so consider them to work like this:

Prescale Value:	Clock:
0-1	1us (Consider the Divider 8 bits in this case)
2	2us
3	4us
4	8us
5	16us
6	32us
7	64us

While the divider is a 7 bit value. Thus the final clock is:

$$\frac{1}{\text{Prescale Clock} * (\text{Divider Value}+1 \text{ if Prescale is } <2, \text{ or } +128 \text{ If } >=2)}$$

Which ranges from 61.5Hz to 1MHz in a nonlinear scale. Note that there is NO clipping protection in any of these calculations. So be careful with your frequency choices.

When converting between frequencies (Hz) and HandyMusic's 16.8 format, the following equation should be observed:

F = Desired Frequency
p = Number of Polynomial Counter clocks for one full period of the waveform
For example, this would be 2 for a 50% duty cycle pulse using feedback setting $01, or 3 for a 33% duty cycle pulse using feedback setting $3

```
Fp = F*p;

if(Fp > 1000000.0) div = 0;
else if(Fp > 3906.25) div = (1000000/Fp)-1;
else if(Fp > 1953.13) div = (500000/Fp)+128;
else if(Fp > 976.563) div = (250000/Fp)+256;
else if(Fp > 488.281) div = (125000/Fp)+384;
else if(Fp > 244.141) div = (62500/Fp)+512;
else if(Fp > 122.07) div = (31250/Fp)+640;
else if(Fp > 61.0352) div = (15625/Fp)+768;
else div = 1023;
```

div = HandyMusic Frequency value used in bits 0-9 of the 16.8 format. All other bits should be set to zero.

- **Frequency Offset Pitch Adjust:** A 10-bit value which is added to the Frequency Offset every audio frame, useful for things vibrato in notes and frequency sweeps in sound effects. This is actually stored in 16.8 precision, but the bottom eight and top six bits are ignored by the hardware and used only to simplify calculations.
- **Volume:** A simple, 8.8 precision volume. Only the top 8 bits are significant to the hardware. Each audio frame the Volume Adjustment is added to this value, the new result replacing the old value. There is no clipping in this calculation, be careful. Note that the 8-bit value written to the volume register is signed, so a negative value is just a positive value with a different phase. Also, the actual volume of the audio channel depends upon the channel mode. In nonintegrated mode, the amplitude of the pulse generated by the hardware is the same as this value. In integrate mode, the output value is a running total of the previous volumes, which are either added or subtracted based upon the output of the poly counter (1=add volume, 0=subtract volume).
- **Volume Adjustment:** A simple, 8.8 precision volume adjustment. Only the top 8 bits are significant to the hardware. Each frame this is added to the Volume value, the new result replacing the old. This is useful for volume envelopes in both instruments and sound effects.
- **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 the attenuation for the right speaker. 1111 is loudest, 0000 is off. This is only useable for music, all sound effects are forced to use a panning value of $FF (center). Note that this really only does anything in the Lynx II, as the original Lynx is monophonic.
- **Waveshape Selector Mode:** The Lynx has two methods for generating waveforms based upon the output of its polynomial counter. In nonintegrate mode, the output of the polynomial counter is directly reflected as a pulse waveform with an amplitude equivalent to the contents of the Volume setting. These are generally square waveforms. In intergrate mode, the current amplitude of the channel is adjusted based upon the output of the polynomial counter. If the output of the polynomial counter is 1, the channel's amplitude is increased by the value in the volume register. If the output is zero, the channel's amplitude is decreased respectively. These are much more triangular waveforms.

- **Shift Register:** The 12-Bit contents of the polynomial counter's shift register.
- **Feedback Taps:** 9-Bits of feedback connected to outputs 11, 10, 7, 5, 4, 3, 2, 1, and 0 of the shift register in the polynomial counter. This and the shift register directly effect the waveform generated. For example, a shift register setting of 0, and feedback tap 0 set to 1, a square pulse will be generated by the polynomial counter.
- **Writeback Disabling:** HandyMusic 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, writebacks on Channel 0 are temporarily disabled while playing PCM samples in HandyMusic.

Instruments and Sound Effects

In HandyMusic, instruments and sound effects have been merged into a single entity. This not only allows the programmer to create complex sounding instruments, but allows HandyMusic to use less space in the Lynx's already limited memory, leaving more for use by game software.

However, while the instruments and sound effects are decoded identically, they are not stored together. Sound effects are kept on their own in a specially formatted block which may be located anywhere in memory, while the instruments for a particular song are always packed along with it.

The basic format for an instrument or sound effect definition is a simple scripting language supporting looping and termination which is used to change the state of the audio hardware over time. A description of the formatting and encoding of the instruments and sound effects follows.

- **Instrument/SFX Header:**

  Each instrument/sound effect contains a 112-bit header which is decoded before any of its script is processed. The header is formatted as follows:

```
[NoteOff Lo][NoteOff Hi][ShiftReg Lo][ShiftReg Hi]
0           8           16           24
[Feedback Lo][Feedback Hi + Integrate Flag]
32           40
[Volume]
48
[Volume Adjustment Lo,Decimal]
56
[Frequency Offset Lo,Hi]
72
[Frequency Offset Adjustment Lo,Hi,Decimal]
88                                       111
```

  - **NoteOff Lo/Hi**
    The low and high addresses of the note off section of the instrument script. This is used exclusively for instruments, and is empty for sound effects.
  - **ShiftReg Lo/Hi**
    Initial setting of the polynomial counter's 12-bit shift register. The contents of the low variable directly correspond to bits 0-7 of the shift register. However bits 8-11 are shifted up into bits 7-4 of the high variable to better reflect the actual Lynx register structure.
  - **Feedback Lo/Hi+Integrate Flag**
    These contain the inital settings for the feedback register and integrate mode flag. However, their organization is not very straightforward. The low variable contains bits 0, 1, 2, 3, 4, 5, 10, and 11 of the feedback enable register. The high variable contains feedback bit 7 in its own bit 7, and the integrate flag in bit 5. The rest are unused.
  - **Volume**
    The one byte initial volume setting. The decimal is left out here and is always zeroed at the start of an instrument or sound effect.
  - **Volume Adjustment**
    The two byte initial volume adjustment value. First is the one byte volume adjustment value, then the one byte decimal.
  - **Frequency Offset**
    The two byte initial frequency offset value. First the low byte, then the high. The decimal is excluded and always assumed to be zero.
  - **Frequency Offset Adjustment**
    The three byte initial frequency offset adjustment. First the low byte, then the high, and finally the decimal.

- **Instrument/SFX Script:**
  Following the instrument/sound effect's header is its sound script. 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: Stop Script Decoding**
    - Format: `[0] (1 byte)`
    - Immediately stops the decoding of the sound script and frees the channel.
  - **1: Wait**
    - Format: `[1][Number of Frames] (2 bytes)`
    - Pauses sound script decoding for the specified number of frames.
  - **2: Set Shift, Feedback, and Integrate Mode**
    - Format: `[2][ShiftReg Lo][ShiftReg Hi][Feedback Lo] [Feedback Hi + Integrate Flag] (5 bytes)`
    - Replaces the current shift and feedback register contents with the specified new values, stored in the same format as in the instrument/sfx header.
  - **3: Set Volume and Volume Adjustment**
    - Format: `[3][Volume][Volume Adjustment][Volume Adjustment Decimal] (4 bytes)`
    - Replaces the current volume and volume adjustment values with the specified new values.
  - **4: Set Frequency Offset and Frequency Offset Adjustment**
    - Format: `[4][Frequency Offset Lo][Frequency Offset Hi] [Frequency Offset Adjustment Lo] [Frequency Offset Adjustment Hi] [Frequency Offset Adjustment Decimal] (6 bytes)`
    - Replaces the current frequency offset and frequency offset adjustment values with the specified new ones.
  - **5: Set Loop Point**
    - Format: `[5][Number of Times to Loop] (2 bytes)`
    - Defines the location in the script following this command as a loop point which will be returned to the specified number of times. If a negative number is used, the loop will continue infinitely.
  - **6: Loop**
    - Format: `[6] (1 byte)`
    - If the loop is not infinite, the loop counter is decremented and unless the counter has reached zero, the script decoder will continue decoding at the loop point. Loops may be four-deep.

- **Instrument/SFX Packing:**
  Instruments and sound effects are packed together in the following format which may be placed anywhere in memory (assuming the pointers used are correct) for sound effects, and will be packed along with each piece of music for instruments.
  - **Instrument/SFX Low Address Pointers**
    - Contains all of the low bytes of the sound script addresses. Max 256 entries.
  - **Instrument/SFX High Address Pointers**
    - Contains all of the high bytes of the sound script addresses. Max 256 entries.
  - **SFX Priorities (Excluded for Instrument Blocks)**
    - Contains all the priorities of the sound scripts. Max 256 entries.
  - **The sound scripts themselves follow in any order you wish**

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, panning, and timing. In most respects, the HandyMusic format may be considered a stripped down MIDI script composed only of note on, note off, panning, pitch slide, and rest commands. Looping is also supported, and like the sound effects, the loops may be four levels deep. 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 Channel 0 only, allowing playback of 8KHz samples in time with music.

As the Lynx has four channels, each music script is composed of up to four 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 four channels may use less by specifying an initial priority of zero in its header.

Instruments for a given piece of music are packed with it, located after the header, but before the music scripts themselves.

- **Music Header:**
  Located before each piece of music's instrument data and sound scripts is a 128-bit header containing information about the track priorities, locations, and instrument table pointers. The format is as follows:

```
[Track 0 Priority][Track 1 Priority]
0                 8
[Track 2 Priority][Track 3 Priority]
16                24
[Track 0 Script Lo Address]
32
[Track 1 Script Lo Address]
40
[Track 2 Script Lo Address]
48
[Track 3 Script Lo Address]
56
[Track 0 Script Hi Address]
64
[Track 1 Script Hi Address]
72
[Track 2 Script Hi Address]
80
[Track 3 Script Hi Address]
88
[Instrument Low Address Table Pointer Lo,Hi]
96
[Instrument High Address Table Pointer Lo,Hi]
112                                     128
```

  - **Track X Priority**
    The priority of the given track at its start, if this is a value of zero, the track is effectively ignored. This may be used to generate music with less than four tracks.
  - **Track X Script Lo/Hi Address**
    The starting address of the sound scripts for each track, which should be located after the instrument block.
  - **Instrument Low/High Address Table Pointers**
    The addresses of two tables which contain the low and high addresses of the instrument scripts included with the piece of music.

- **Music Script:**
  Following a music track's header and instrument block is its music scripts, of which there may be up to four. The scripting commands are made up of one byte blocks and are slightly similar to those of sound effects, with a few changes. The commands, their format, and descriptions of their operations are listed below:

  - **0: Set Priority**
    - Format: **`[0][Priority] (2 bytes)`**
    - Sets the priority of the track relative to the sound effects. Higher priorities win out when competing for channels, the same priorities should **never** be used between sound effects and instruments. **A priority of zero stops the track from decoding.**
  - **1: Set Panning**
    - Format: **`[1][Panning] (2 bytes)`**
    - Sets the panning of the instruments played in the current channel.
  - **2: Note On**
    - Format: **`[2][Instrument][Base Frequency Lo]`**
      `[Base Frequency Hi][Delay Lo] (5 bytes)`
    - Plays a given instrument with the specified base frequency, then waits for the given one byte delay
  - **3: Note Off**
    - Format: **`[3][Delay Lo] (2 bytes)`**
    - Forces the currently playing instrument into the note off portion of its script, then waits for the specified one byte delay.
  - **4: Set Base Frequency Adjustment**
    - Format: **`[4][Base Frequency Adjustment Lo]`**
      `[Base Frequency Adjustment Hi]`
      `[Base Frequency Adjustment Dec] (4 bytes)`
    - Sets the Base Frequency Adjustment value, which can be used for pitch slides, etc. This value is initialized to zero on the start of a song.
  - **5: Set Loop Point**
    - Format: **`[5][Number of Times to Loop] (2 bytes)`**
    - Defines the location in the script following this command as a loop point which will be returned to the specified number of times. If a negative number is used, the loop will continue infinitely. Loop values of $80 (-128) should not be used, as the driver requires this value as a special tag for pattern calls and returns.
  - **6: Loop**
    - Format: **`[6] (1 byte)`**
    - If the loop is not infinite, the loop counter is decremented and unless the counter has reached zero, the script decoder will continue decoding at the loop point. Loops may be four-deep.
  - **7: Wait**
    - Format: **`[7][Delay Lo][Delay Hi] (3 bytes)`**
    - Stops decoding for the specified two byte delay.
  - **8: Play Sample**
    - Format: **`[8][Sample Number] (2 bytes)`**
    - Begin playback of the specified PCM sample. Only usable in Channel 0.
  - **9: Pattern Call**
    - Format: **`[9][Address Lo][Address Hi] (3 Bytes)`**
    - Jumps to the address of the music script given, and sets up the current position in the music script as the destination for the Pattern Return command.

- **10: Pattern Return**
  - Format: **`[A] (1 byte)`**
  - Returns to the portion of the music script which was being played before the last Pattern Call command.
- **11: Short Note On**
  - Format: **`[B][Base Frequency Lo] [Base Frequency Hi][Delay Lo] (4 bytes)`**
  - Plays the last used instrument with the specified base frequency, then waits for the given one byte delay
- **12: Pattern Break**
  - Format: **`[C] (1 byte)`**
  - Returns all channels (not just the one encountering the command) to the portion of the music script they were at when they made their first CALL (if any). Effectively returns all channels to the "main" pattern.

• Music Data Packing:

Music data is stored in a block format consisting of the music header, instrument block, and track data. It is expected to reside at $B000-$BFFF only. New music data may be used by stopping playback of the current music track, and loading another in this memory space.

- **Header**
  - The Song Header
- **Instrument Block**
  - Two tables of the high and low addresses of the instrument scripts (see instruments/sound effects section), followed by the instruments themselves.
- **Track Data**
  - The music script data for up to four tracks of music.

PCM Sample Playback

HandyMusic supports streaming a single PCM Sample from the game cartridge on request by the music script or a call to **`PlayPCMSample`**. Samples are required to be stored as 8KHz 8-Bit signed monophonic raw PCM data. There is no concept of priority between samples and the music or sound effects, and sample playback is always performed in Channel 0. Any music or sound effects currently occupying that channel will be muted for the duration of the sample playback, but will continue to decode normally.

As PCM samples are streamed from the cartridge, it is not safe to access the cartridge interface while a PCM samples is playing. Thus it may be wise to ensure **`Sample_Playing`** is equal to zero before loading from the cartridge in your game software. Any loading routines included with HandyMusic such as **`HandyMusic_LoadPlayBGM`** automatically check for activity by the sample playback routine and will wait for it to finish before accessing the cartridge for any of their own data. Sample playback may also be disabled by writing a nonzero value to **`HandyMusic_Disable_Samples`**, upon which all sample playback requests both from music scripts and user calls will be ignored.

IRQ-Based Sample playback is extremely resource intensive on the Lynx, especially at 8KHz. Thus it is suggested that sample playback only be used in non-game critical areas or in a situation where the game tick is less than 60Hz.

HandyMusic Code Compiler (HMCC)

The HandyMusic Code Compiler (HMCC) generates both sound effect and music binaries from textual source files written in a music-oriented programming language called Sound ASSembler (SASS). The structure of these scripts is a simple assembly-like syntax based upon Epyx's original HSPL sound suite for the Lynx. Two formats are used, one for instrument and sound effect definitions, the other for music tracks.

HMCC is a commandline application, and operates in either sound effects or music compiling modes. Executing HMCC without any parameters will yield the following response:

```
C:\Lynx\HMCC>hmcc
Usage:
hmcc $dest_addr sfx_file.txt
hmcc $dest_addr outfile.mus inst_file.txt trk_1.txt [trk_2.txt] [trk_3.txt] [trk_4.txt]
```

In sound effects mode, HMCC requires a destination address ("$A6B0" is recommended) and a sound effect script file. For example, building the included example sound effects file can be accomplished as follows:

```
C:\Lynx\HMCC>hmcc $A6B0 C:\Lynx\SASS\DemoSFX\DemoSFX.txt
Sound Effects Mode...
Saving sfx equates to : C:\Lynx\SASS\DemoSFX\DemoSFX.equ
6 instruments/sfx were found
The current instrument/sfx table is: 143 bytes
Saving sfx binary to : C:\Lynx\SASS\DemoSFX\DemoSFX.sfx
```

Both an equate (*.equ) and sound effects binary (*.sfx) bearing the same name as the sound effects source will be generated on a successful compile.

In music mode, HMCC requires a destination address ("$B000" is recommended), desired output file, instrument script file, and one to four music track scripts. Building the example music track can be accomplished as follows:

```
C:\Lynx\SASS\DemoMusic>C:\Lynx\HMCC\HMCC.exe $B000 C:\Lynx\HandyAudition\SASS\Demo
Music.mus C:\Lynx\SASS\DemoMusic\DemoInstr.txt C:\Lynx\SASS\DemoMusic\DemoTrack1.txt
C:\Lynx\SASS\DemoMusic\DemoTrack2.txt C:\Lynx\SASS\DemoMusic\DemoTrack3.txt C:\Lynx\
SASS\DemoMusic\DemoTrack4.txt
Music Mode...
12 instruments/sfx were found
The current instrument/sfx table is: 468 bytes
The current music binary size is : 913 bytes
Saving music binary to : C:\Lynx\HandyAudition\SASS\DemoMusic.mus
```

A single binary music file (*.mus) with the name given in the parameter "outfile.mus" will be generated on a successful compile. Music binaries containing less than four tracks will leave the unallocated channels available for sound effect playback at all times.

- **Constants:**

  HMCC allows for all values to 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 the majority of HandyMusic's operation depends upon both 16.8 and 8.8 formats, 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 its leading integer. Values written without a decimal are assumed to have a decimal with value “0.”

- **Comments:**

Follow a semicolon “;” and span a single line. They may be entered as shown below:

```
; This is a comment, it will span the whole line
as4 60 ; Comments can also follow commands in both SFX and Music modes
a.4 40 ;Starting the text immediately after the semicolon is fine

rest 40; But this is invalid
```

- **Timing:**

All values following a note play or rest command are given in driver ticks which are 60Hz by default. For example:

```
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
```

However, if the rate at which HandyMusic processes envelope frames is altered (such as attaching it to a 120Hz or 240Hz timer) the tick duration will change accordingly.

- **Lynx-SASS Instrument and Sound Effect Definitions:**
  As instruments and sound effects are considered the same entity in the HandyMusic driver, they share a definition format in Lynx-SASS. A listing of the format structure is shown below; items in **orange** may be used only in instrument definitions, and items in **green** pertain only to sound effects.

```
name priority                                   ;>   Label
       waveform feedback [shift]                ;\
       volume amplitude [adjustment]            ; \  Header Data
       frequency divider [adjustment]           ; /
       tuning value                             ;/
       {
              rest ticks                        ;\
              waveform feedback [shift]         ; \
              volume amplitude [adjustment]     ;  \
              frequency divider [adjustment]    ;   \
              loop count                        ;    Instrument/SFX Body
                     ; (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 “triangle_powpow”), and are used to identify instruments inside of music scripts, and generate equates for sound effects. **Priority** may range from 0-255 and represents the importance of the sound effect relative to both the music tracks and other sound effects. If a sound effect is requested by the user and it cannot find a free channel, it will knock out the first music track or sound effect it finds with a lower priority than it. If nothing of a lower priority is found, the request will be ignored and the sound effect will not play. A priority of 0 will result in the sound effect never playing, and **priorities should never be shared across music tracks and 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 the **waveform**, **volume**, **frequency**, and **tuning** commands. **Tuning** applies only to instruments, and will have no effect on 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 commands within **{** Curly Braces **}**. **Noteoff** only applies to instruments, and will have no effect on sound effects. A complete command listing follows:

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

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

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

```
       volume 70 -3    ; Set volume to 70, subtract 3 every tick
       rest 14         ; Wait 14 ticks, but still process the above
       end             ; Stop
loop
```

- **waveform,w feedback [shifter]** : set channel waveform
  Sets the feedback and shift register contents of a channel's polynomial counter, therefore specifying the waveform in a given instrument or sound effect. The feedback and shifter values are 16-bits wide, but their organization is not very straightforward. The feedback low byte contains bits 0, 1, 2, 3, 4, 5, 10, and 11 of the feedback enable register. The high byte contains feedback bit 7 in its own bit 7, and the integrate flag in bit 5. The rest are unused.
  The shifter low byte directly corresponds to bits 0-7 of the shift register. However bits 8-11 are shifted up into bits 7-4 of the high byte to better reflect the actual Lynx register structure. Excluding the shifter value in the waveform command will yield a default value of zero.
  The Lynx can create a wide variety of sounds using the various combinations of feedback and shifter values, and experimentation is extremely helpful.

```
waveform $1     ; 50% duty cycle pulse.
...
waveform $3     ; 33% duty cycle pulse.
...
waveform $2040 ; But this combination of waveform and
volume 32       ; volume settings will make a triangle.
```

- **volume,v amplitude [adjustment]** : set channel volume and amplitude sweep
  Sets the current instrument or sound effect volume, and (if given) its adjustment value each tick afterward. The adjustment is added to the current volume setting every driver tick, and thus may be used for volume sweeps and envelopes.

```
...
volume 100 -4   ; Set volume to 100, and subtract 4 every
...             ; tick afterward.
```

- **frequency,f divider [adjustment]** : set channel frequency and pitch sweep
  Sets the current instrument or sound effect frequency offset, and (if given) its adjustment value each tick afterward. The adjustment is added to the current frequency offset every driver tick, and thus may be used for pitch sweeps, envelopes, and vibrato.
  Note that the term "frequency" is a bit of a misnomer, as all Lynx audio is done through dividers. Thus larger values correspond to lower pitches, and smaller values correspond to higher pitches.

```
...
frequency $200 12      ; Set frequency to 512, add 12 every
...                    ; tick afterward. (slide down)
```

- **loop,l count** : start of loop
  Specifies the start of a loop, and the number of times it will repeat (0-127). If negative values are used, the loop will continue indefinitely. Example:

```
loop -1                         ; Repeat outer loop forever
        loop 10                 ; Repeat inner loop 10 times
                volume 100 -5.8
                rest 6
        endloop
        volume 80
        frequency 500 -1
endloop
end                             ; Stop
```

Loops may be up to four levels deep, any higher will cause erratic behavior:

```
        l -1                            ; Repeat outer loop forever
                ; (Outer loop body)
                l 30                    ; Repeat inner loop 30 times
                        ; (Inner loop body)
                        l 20            ; This one 20 times
                                ; (2nd Inner loop body)
                                l 10    ; and this 10 times
                                        ; (3rd Inner loop body)
                                el
                        el
                el
        el
        end                             ; Stop
```

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

```
        loop -1                         ; Repeat forever
                frequency 300 -10.7
                rest 18
        endloop                         ; Marks end of above loop
        end                             ; Stop
```

- **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 the
        noteoff         ; music script while an instrument is
        volume 80 -5    ; playing, the instrument will start
        rest 8          ; executing the portion of its script
        end             ; directly following the noteoff command.
```

- **tuning,t** : specify instrument tuning
  This value corresponds to the number of poly counter clocks required for a single period of the instrument waveform, and will be different for the various configurations possible in the Lynx hardware.

```
square
        waveform $1     ; A 50% duty cycle pulse (feedback $1) has a
        tuning 2.0      ; period of two clocks, thus a tuning of 2.
        {
                ...
        }
pulse33
        waveform $3     ; But a 33% duty cycle pulse has a period of
        tuning 3.0      ; three clocks, and a tuning of 3.
        {
                ...
```

Lynx-SASS 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 Label
{
        using instrument                        ;\
        priority value                          ; \
        pan attenuation                         ;  \
        call blockLabel                         ;   \
        loop count                              ;    \
                ; (Loop Body)                   ;     \
        endloop                                 ;      Block Body
        rest ticks                              ;     /
        pitch adjustment                        ;    /
        as3 ticks                               ;   /
        sample number                           ;  /
        sfx ticks [divider]                     ; /
        wait ticks                              ;/
        end                                     ;
}

name                                            ;>      Sub Block Label
{
        ...                                     ;\
        return  / break                         ;/      Sub Block Body
}
```

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** may range from 0-255, just as in sound effects. Again, **priorities should never be shared across music tracks and sound effects.** Using a priority of zero will result in the music track never decoding.
Following the **label**, and enclosed between two **{** Curly Braces **}** are one or more commands composing the **block body**. A complete 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 rage 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
...
```

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

```
...
using pulse33   ; Using instrument "pulse33"
g.3 20          ; The following notes will be played with
rest 10         ; "pulse33," 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
...
```

- **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 than the sound effects. Again, values should not be shared between the music track and sound effects.

```
...
priority 120    ; Set priority to 120 (midrange)
...
priority 255    ; Set priority to 255 (highest possible)
...
```

- **pan,p attenuation** : set channel panning
  Set the stereo attenuation for a music track using a one-byte value. Bits 7-4 indicate the attenuation in the left speaker and Bits 3-0 indicate the attenuation on the right. A value of $0 is silent while $f is loudest.

```
...
pan $00         ; Silent
pan $f0         ; Full volume in left speaker
pan $0f         ; Full volume in right speaker
pan $ff         ; Full volume in both speakers
...
```

- **loop,l count** : start of loop
  Specifies the start of a loop, and the number of times it will repeat (0-127). If negative values are used, the loop will continue indefinitely. Values of $80 (-127) should not be used, as they are a special case for CALLs.

```
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
```

  Loops may be up to four levels deep, including calls.

```
l -1                            ; Repeat outer loop forever
        ; (Outer loop body)
        l 30                    ; Repeat inner loop 30 times
                ; (Inner loop body)
                l 20            ; This one 20 times
                        ; (2nd Inner loop body)
                        l 10    ; and this 10 times
                                ; (3rd Inner loop body)
                        el
                el
        el
el
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
```

- **sample,smp number** : request sample playback
  Requests playback of a given PCM sample. Note that script decoding continues while the sample is playing, and thus this command will usually be followed by a rest. This command should only be used in track zero of any given music piece.

```
...
sample 3                        ; Stream PCM sample 3
rest 240                        ; Wait until it's done
...
```

- **sfx,s ticks [divider]** : play current instrument as sound effect
  Play back the currently selected instrument as a sound effect for a given length of time. A divider offset may be specified, and is assumed zero if excluded. This is especially useful for percussion or instruments with no key off action.

```
...
using snare
sfx 20                          ; Play the snare instrument
sfx 3                           ; with a frequency offset of
sfx 3                           ; zero for the given number
sfx 15                          ; of ticks.
...
```

- **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 frames. Useful for delays where note off behavior is not desired.

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

- **pitch,pt adjustment** : set per-note pitch slide
  Sets a value which will be added to a given instrument's divider value every driver tick after its key on event, allowing for pitch slides up or down. The effect may be disabled by setting the pitch adjustment value to zero.

```
        ...
        pitch 12                        ; Slide down
        as4 20
        rest 20
        pitch -16                       ; Slide up
        as4 20
        rest 20
        pitch 0                         ; Back to normal
        ...
```

- **call,c blockLabel** : call script block
  Begin decoding a given script block with the name specified in **blockLabel**. Calls may be up to four levels deep, including loops.

```
        ...
        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.

```
mainTrack1 120
{
        ...
        call pianoSolo
        ...
}

pianoSolo
{
        ...
        call drumminThing
}

drumminThing
{
        ...
        break           ; Resumes decoding after “call pianoSolo”
}
```

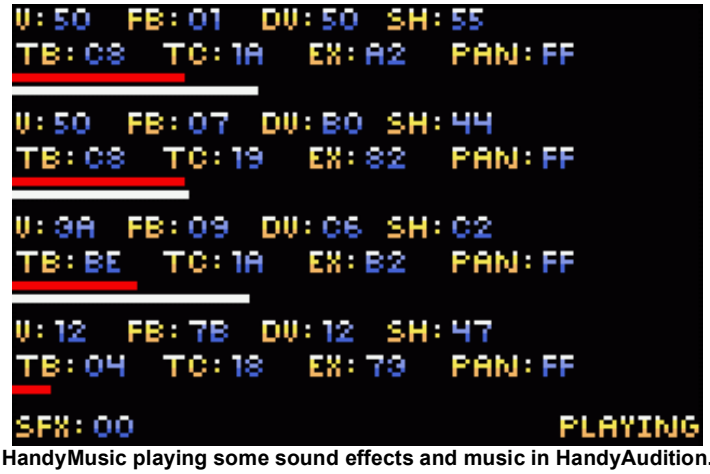
- **end,e**
  Stops decoding of the music track and frees the channel. This is equivalent to setting the priority to zero through “`priority 0`.”

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

HandyAudition

HandyAudition is a simple Lynx program for testing composed music and sound effects without having to configure HandyMusic for your current game project. The state of the audio hardware registers are displayed to help with script debugging. Simply swap out the sound effect and music blocks used by HandyAudition to quickly test your new sounds. Note that the default memory map is used for HandyMusic, so the maximum size of a music track is 4KB.

New music tracks may also be sent from a PC to the ComLynx port for a faster turnaround time. Tracks should be sent as raw binaries in 8N2 @ 9600bps (8-Data Bits, Two Stop Bits) with no handshaking.



HandyMusic playing some sound effects and music in HandyAudition.

Controls:

- **Left/Right** : Select Sound Effect to Play
- **A** : Play Sound Effect
- **B** : Stop Sound Effect
- **Option 1** : Play Music
- **Option 2** : Stop Music

Register Guide:

- **V** : Channel Volume
- **FB** : Low Feedback Register
- **DV** : Direct Volume (Actual channel amplitude)
- **SH** : Low Shift Register
- **TB** : Timer Backup
- **TC** : Timer Control
- **EX** : Extra Audio Bits
- **PAN** : L/R Attenuation, will read as open bus in the original Lynx

Statuses:

- **READY** : Idle
- **PLAYING** : Playing the current music track
- **RECEIVING** : Receiving a new music track over ComLynx
- **RX INTEGRITY ERROR** : A framing or overrun error was encountered while receiving new music data. The baud rate and format used by the host may be incorrect.
- **RX SIZE ERROR** : The data received was too large to fit in music track memory (4KB).