Creating Music for the Lynx....*The Real Story*

Version 1.1, August 1990

Chris Grigg

Introduction

This document is intended for Lynx programmers and music artists, and describes the Lynx music authoring process and tools. Although I helped design these tools, their implementation was out of my hands, and when in the fullness of time I took over writing Lynx music I started this document as a place to hold notes on how the current rev of the tools actually work. I have tried only to include things that I've learned from experience, i.e. that approach the truth, and that aren't adequately explained elsewhere. I hope that, even though it's densely packed with information, this will explain Lynx music creation lots better than the other music and sfx documents do.

The Big Picture

The following is a bottom-up explanation of the Lynx music authoring process and tools; bottom-up because that's the way its creators approached the task, and it makes more sense from that viewpoint.

The Lynx has four functionally identical, totally independent audio channels (also called voices). Lynx music is therefore constructed in up to four monophonic (1-voice) parts, depending upon the musical, technical and gameplay requirements for the piece. Each voice's architecture is similar to the Atari VCS, consisting of a time base generator driving a 12-bit shift register that includes a 9-bit bitwise feedback and XOR gate mechanism. The output of the shift register may or may not be accumulated, and there's a signed 8-bit volume control (so a volume of +80 sounds the same as one of -80). For more details, see the HANDY AUDIO HARDWARE OVERVIEW, aka. HANDY AUDIO SYSTEM DESCRIPTION. There is no envelope generation mechanism in hardware, so we must use software to shape sounds over time; this includes the amplitude envelope. The first rack of Epyx Lynx games all used a 240 Hz audio interrupt rate to keep envelopes smooth; this is a big cost to the game in terms of performance, and sound load is a recurring problem for Lynx game designers. Lynx voice timbre ("instrument sound") is controlled by the contents of the shift register and the setting of the shift register's feedback mask, the timebase's frequency, and the voice's volume; any of these can be varied over time to create dynamic timbres.

The Handy system software package includes the HMUSIC music driver, which is thoroughly described in its own document. Other music drivers will certainly exist in the future, but for now all of the supplied music tools exist specifically to create data for HMUSIC. HMUSIC supports notes, rests, calling HSFX sound effects (HSFX is the sound effects driver and editor system), and a number of timbre controls. It interprets data in a particular, very compact, format: each note or rest is only one byte long. There is currently only one tool for building HMUSIC data, and that is the HSPL music compiler (although other tools could be built; see the HMUSIC document, sections SONG DATA TABLE LAYOUT and VOICE DATA: NOTES AND COMMAND FLAG WORDS for the exact data format used by the driver). HSPL is the Handy Sound Programming Language.

The HSPL compiler produces a binary output file in HMUSIC format from one or more input text files in the HSPL language. HSPL source has flexible formatting. It can look like this:

```
; === Track 1:  SpaceOne
; this is a comment, as is anything on a line after a semicolon.
; there can be at most one SPL statement per line of source file.
; the rest of this example is commented ridiculously thoroughly to
;    answer many of the questions the statements may raise.
PRI 232                         ; priority of 232
DFV                             ; use default voicing setting
AGD 38                          ; articulation: gate-duration.
                                ;    Gate-on for the 1st 38 frames of each note
ADSR 7,105,12,85,12             ; volume envelope:
                                ;    attack time, peak value, decay time,
                                ;    sustain value, and release time
TPO 4                           ; tempo: 4 audio interrupts per HSPL unit time
GTR -32                         ; global transpose: all tracks down 32
                                ;     halfsteps from score

; Bar 1 -- barlines are comments only.
        f.3     12              ; this is a note. The number is the duration in
                                ;     in HSPL time units, AKA mindurs.  SPL has
                                ;     no 'note' statement, you'll, uh, note.
                                ;     The compiler is smart enough to recognize
                                ;     the note name.
        c.4     11
        rest    1               ; this is a rest.
                                ;     The number is similarly a duration.
        f.4     9
        rest    3
        as3     12              ; the s means sharp
        c.4     12
        as4     12
        rest    24
; Bar 2
        gs5     9
        rest    15
        g.5     12
        rest    12
```

etc.

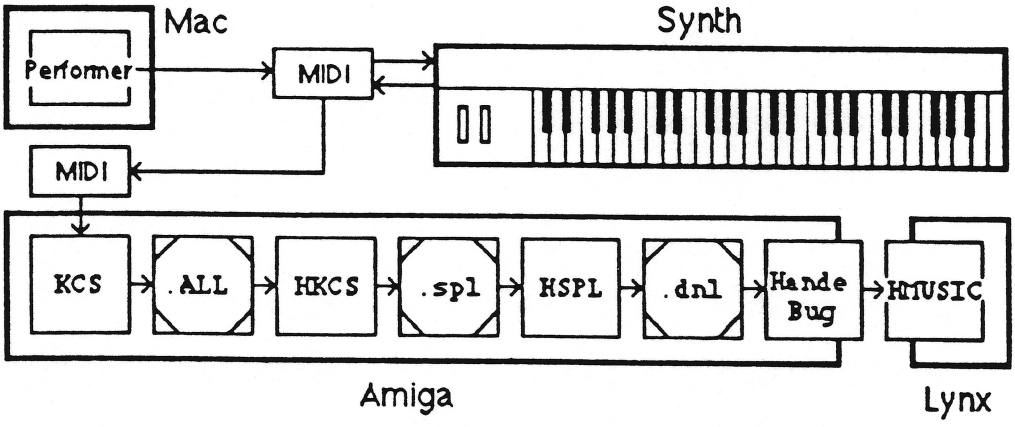
HSPL source can be typed in by hand, or can be created by a score conversion program. Usually a score is converted by a tool and then iteratively tweeked and voiced by hand and ear, listening to an actual Lynx development card. Every time the sound artist wants to hear changes made to the HSPL source for a piece of music, he or she recompiles and downloads with the `play` shell script and Handebug, the Handy debugger. The HSPL language is described in the HSPL document. We've provided the HKCS score converter program, which turns .ALL files created by Dr. T's Keyboard Controlled Sequencer (KCS), Version 1.6A, into HSPL source that can then be text-edited and compiled with the HSPL compiler. This provides a path from KCS into SPL.

KCS, a widely available third-party Amiga application *not* supplied with the Lynx development system, is an Amiga-native MIDI music sequencer/editor that can be used to develop music sequences for Lynx (the first set of Epyx titles were mostly done this way). Musicians may find that KCS' ideosyncracies take some getting used to. Because KCS can accept external MIDI sync, it can also serve as a port into the Amiga for music developed on non-Amiga sequencers, which many musicians prefer to use (I like to use Performer on the Mac, and Bob Vieira likes Notator on the ST).

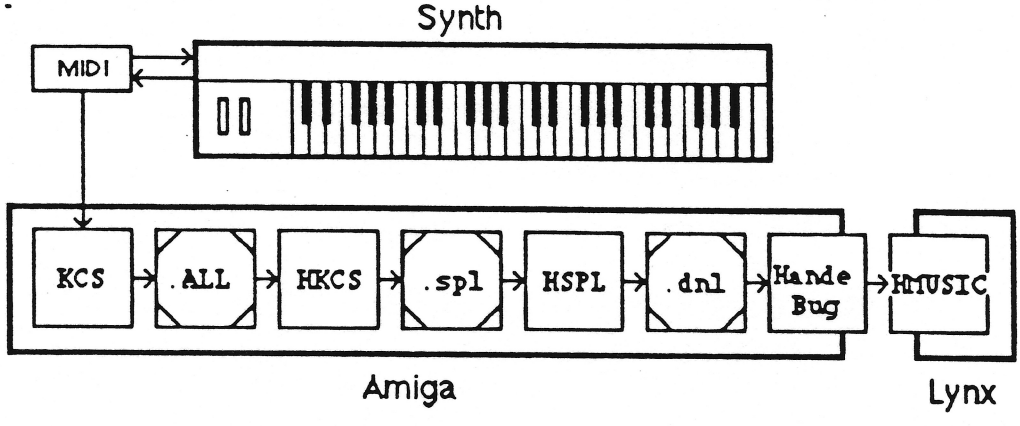
There is currently no way to play the Lynx audio hardware in real time directly from a MIDI sequencer. First music is written and arranged into 1 to 4 parts with a MIDI sequencer and MIDI instruments and/or controllers (there's no need for a Lynx development board at this point), then it's turned into SPL on the Amiga, and then it's voiced and otherwise adjusted by ear with a Lynx development board and an Amiga. Score and voicing revisions in SPL are done with your favorite text editor.

In summary, you have several options for developing Lynx music:

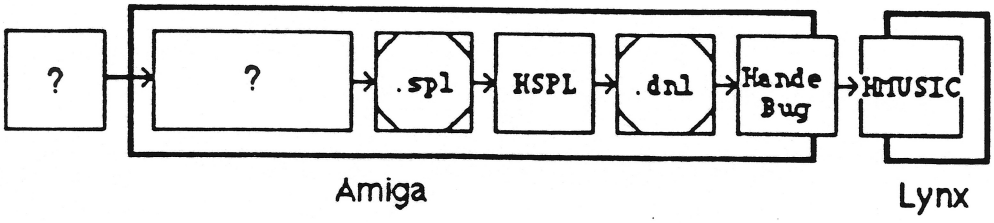
- You can write your music on any MIDI sequencer, Amiga or non-Amiga, that sends MIDI starts and stops and MIDI clocks, and/or MIDI Time Code (MTC), but you'll have to buy and use KCS and an Amiga MIDI interface, and use HKCS to get into HSPL, where you'll add voicing commands and tweeking. This how we Epyxies like to do it, and it lets your musicians work with their own equipment for the composition & arranging phase.



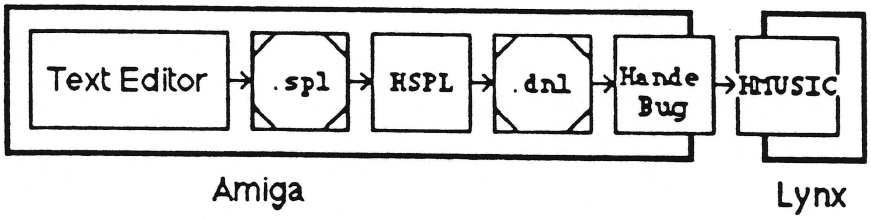
- Or you can buy KCS and an Amiga MIDI interface and write your MIDI music right on the Amiga, using HKCS to get into HSPL, where you'll add voicing commands and tweeking. This is a lower-cost alternative.



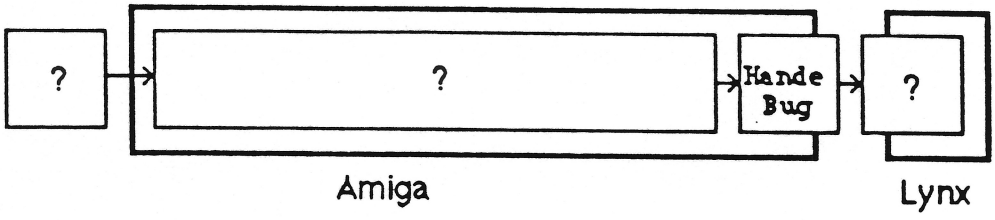
- Or you can use some other music authoring system, either commercial or of your own device, and develop your own converter to turn its output into either HSPL text or HMUSIC binary format. SMUS to SPL or MIDIfile to SPL come to mind. This is for people who like to do their own thing.



- Or you can type in HSPL text, bypassing the sequencer entirely. This is for people who like tedium.



- Or you can bypass the HMUSIC driver totally and roll your own driver and music authoring system. This is for people (brave few!) who aren't afraid of interdependencies between the HMUSIC driver code and other parts of the Handy system software, and who won't want to look at the rest of this document.



The rest of this document describes parts of the existing music authoring system not well documented elsewhere.

Getting Started

Unless you're just transcribing already-written sheet music, you'll need some amount of MIDI gear for writing music. We use a Roland D10 synth, a Macintosh SE running the sequencer program Performer 3.1, and an Austin Development MIDIface II (Mac MIDI interface). You may want less or more; see above for alternative music development approaches.

You'll need time on an Amiga 2000 with the regulation directories `handy:` and `sound:`. You may also need Dr. T's Keyboard Controlled Sequencer, v1.6A, and an Amiga MIDI interface (we use a Creative Microsystems MIDI 1). And you'll need time on an Amiga with a Lynx development board to voice and tweek your compositions.

See the Music and Sfx Examples Disk for complete HSPL music examples that you can `play` with to start learning about the system and the SPL language.

Glossary: Beware Weird Epyx Terminology

- The word *mindur* appears in our documentation occasionally. It means minimum duration, and is another word for the SPL time unit as used in notes, SFX commands, and rests. Its origins are murky and legend-encrusted, dating from Waveform Corporation and MusiCalc.

- *KCS* means Keyboard Controlled Sequencer. It's an Amiga MIDI sequencer program from Dr. T's Music Software. We use version 1.6A.

- *HKCS* is the Amiga program that comes with the Lynx development system and converts KCS's .ALL score files into HSPL source.

- *HSPL* is the Amiga program that compiles HSPL source code into HMUSIC binary data. It's also the name of the compiler's source code language, Handy Sound Programming Language.

- *SPL* refers to Epyx' Sound Programming Language in general. There were many other machine-versions of SPL long before there was a Handy.

Recommended Practices

- Keep A Seperate Directory Per Piece Of Music

  Aside from this being a good basic organizing practice, the HSPL compiler takes its parameters from a file, called `sounds.env`, in the current directory. This makes multiple pieces of music per directory cumbersome. KCS sequencer files can be kept in the same directory as SPL source files. We also make release directories off of the piece directories; every time a version of the music is given to anybody else (like the game programmer), a release directory is made and a copy of the current state of the piece is saved there. We like to group piece directories under level and game directories.

- Keep Handebug in Code Mode, Not Data Mode

  To get into Code mode, type Ctrl-C (doesn't break, don't worry) or click the Code box in the lower left corner of the Handebug screen. Data mode, which as a music author you almost never need, constantly talks to the Lynx development board, slowing the Amiga as a whole and sometimes interfering with interlaced displays.

- Write Music In Mono Tracks

  Whatever sequencer you use, this makes it easier to transfer into KCS and SPL. Avoid note overlaps (starting a new note before a previous note is released), which may confuse HKCS.

- File Naming Conventions

`.ALL`	KCS music score files
`.bin`	`asm` output files
`.bfx`	`asm`'ed, `asmstrip`'ed binary sfx file made from a HSFX output 6502 source file, for use in `SFX` statements in HSPL
`.dnl`	HSPL output/HMUSIC input files
`.ENV`	KCS program parameter files, except the file sounds.env which contains HSPL parameters for the current directory's piece of music
`.sfx`	HSFX editor binary sfx files
`.spl`	HSPL source files
`.src`	HSFX editor output 6502 source files

Using KCS

Expect to spend some time with the KCS manual. Its operation is not very intuitive, so it's good that the manual's thorough. Use Track Mode.

KCS is a copy-protected product, so keep the master disk, which is a key disk, close at hand for insertion at the annoying prompt. To start it, `cd` to the directory you want to work in, make sure `kcs` on your path, and type run kcs. KCS runs under multitasking, and although it has no screen gadgets, you can Amiga-N and Amiga-M the Workbench screen to the top and bottom, and you can get to any other programs' screens with the Workbench screen gadgets.

Note: HKCS also writes a `.ENV` file with its `.ALL` file, and `.info` files for each. Assuming you're working with CLI, you may want to remove those `.info` files...but leave the `.ENV` file, which sets up various KCS program parameters per your latest settings.

Transferring Sequences Into KCS Via MIDI

To transfer a sequence into KCS, you need to make your external sequencer transmit MIDI sync and set KCS to take its sync from the Amiga MIDI In, and make sure that the various sync parameters of each side agree. The procedure will be different depending upon the external sequencer, and may require some cleverness. Here's how I set parameters for KCS on the Amiga and for Performer on the Mac to transfer from Mac to Amiga:

Performer:
- Transmit sync, via modem
- Generate MIDI beat clocks
- 24 clocks per qtr note
- First clock is 0

KCS Set Options Page (press key 0 to bring it up):

All options off (not highlighted) *except*:

- MIDI: MIDI Merge — This lets you listen to the input stream over the Amiga MIDI Out as the transfer happens
- OTHER: Activity Display
- VELOCITY PEDAL: Off
- TIMING: MIDI w Song Pointer

In the Set Options page, set the number next to Internal Clk to your song's tempo for verifying your sequence transfers. You have to go to Set Options and choose Internal Clk to play back from KCS, then change back to MIDI w Song Pointer to receive another sequence. This is just switching KCS's time base source between internal and external (MIDI clocks from the other sequencer).

The HKCS Score Converter

HKCS converts a KCS `.ALL` output file into a text file in HSPL source format with the same root filename but with the extension `.spl` in place of the `.ALL`. The `.spl` file will include a header of comments with information about the conversion, and one list of Track information for each track present in the `.ALL` file. These tracks must be hand-cut with a text editor and saved into seperate files if the HSPL compiler is to see them individually.

To start HKCS from the CLI:

```
hkcs InputFile.ALL
```

or:

```
hkcs InputFile.ALL kcsTicksPerSplBar
```

By default, the duration conversion will turn 96 KCS ticks into 1 SPL bar; to override this default, include (at the end of the command line) the number of KCS ticks you want to use per bar; see Examples below.

Tip: It's a good idea to keep the SPL duration numbers as small as possible (without affecting the music itself), because this lets you use bigger `Tempo` statement (`TPO`) numbers in SPL to get the same apparent tempii, so that you have finer resolution when adjusting tempii. If you think small duration numbers won't work for your piece, try using small numbers anyway and then play with the SPL articulation commands (`AGD`, `ASD`, `AGS`) to control the size of the implied rest after each note. You have to make the duration conversion decision at the HKCS stage (pre-SPL), but you can always go back and re-do it if you change your mind.

Examples:

To convert the file `oompa.ALL` into `oompa.spl` with 96 KCS ticks per bar, you'd do this:

```
hkcs oompa.ALL
```

The file `oompa.spl` will be created.

To convert the file `loompa.ALL` into `loompa.spl` with each SPL bar equal to 16 KCS ticks, you'd do this:

```
hkcs loompa.ALL 16
```

The file `loompa.spl` will be created.

Touching-Up HKCS Output

HKCS saves a lot of time, but is imperfect.

• After running HKCS, which puts converted SPL output for all tracks present in the `.ALL` file into the same `.spl` file, you'll probably need to cut each voice's part into a seperate file so that the HSPL compiler can deal with them individually.

• It seems that the terminal `EMS` statement in each track must be in caps, not lowercase (which HKCS writes) to be recognized by the HSPL compiler.

• See the Musical Time In SPL section for how to set your `Tempo/TPO` correctly.

• Depending upon your sequencer quantizing settings, your playing style, and the composition of the piece, there may be unnecessary intervening rests between notes. Unnecessary because SPL includes articulation commands that can be used to imply a note-off before the end of each note, without taking up the space for a seperate rest event. You have to remove these rests by hand to save RAM/ROM space and get better control of voicing (although as you use the system more and more, you'll get a feel for ways of playing and quantizing that avoid them). When you remove a rest, remember to increase the duration of the immediately preceding note, `rest`, or `SFX` by the duration of the rest being cut. These unwanted rests can appear because MIDI sequencers are dumb about the player's articulation, and only record note-ons and note-offs, not scored note durations. Another good reason to remove intervening rests is that the `ADSR` volume, `VFM` pitch, and `VFBM` feedback envelopes are finished at the end of a note, i. e. silence; if you use the articulation controls instead of rests between notes, you can use the release phase of the envelopes between the notes to keep the sound going and changing.

For example, you could reduce this fragment...

```
as3     4
rest    4
b.3     4
rest    4
cs4     4
rest    4
ds4     4
rest    20
```

...to this easier-to-read fragment, which has fewer events:

```
AGS     1       ; do note-offs halfway through notes- rightshift dur
                ;    by 1 bit.
as3     8
b.3     8
cs4     8
ds4     8
rest    16
```

Musical Time In SPL

Bar markings in SPL source are comments only, and are not checked for correct placement. HKCS lets you tell it how many KCS ticks to use per bar when converting KCS to SPL. HMUSIC doesn't know from bars.

Not unrelatedly, SPL doesn't have predefined duration types or tokens, like quarterNote or wholeNote; instead, every note, `SFX`, or `rest` has a numerical duration parameter. This gives more flexibility, but also allows you to confuse yourself. You can always define and use your own duration constants for your piece if you prefer, thus:

```
sxn = 1                 ; sixteenth note
ein = sxn + sxn         ; eighth note
dein = ein + sxn        ; dotted eighth note
qtn = ein + ein         ; quarter note
dqtn = qtn + ein        ; dotted quarter note

etc.
```

The note, rest, and sfx duration parameter is in mindurs; the actual realtime duration of a note, sfx, or rest will be its duration number, times the current `TPO` (`Tempo`) setting, times the audio interrupt period; i. e. mindurs x `TPO` provides a number, and the note will end that number of audio interrupts after it starts. Currently we use a 240 Hz audio interrupt.

Counterintuitively, a higher `TPO` number results in a greater time delay per mindur, and so a slower tempo. The `TPO` number is *not* in terms of beats per minute. Some day, someone should extend the table on the following easy-to-xerox page, using this formula:

$$\text{tempo in bpm} = \frac{60 * \text{audio interrupt rate in Hz}}{\texttt{TPO} \text{ number} * \text{number of mindurs per beat}}$$

240 Hz Tempo Quanta

TPO Number	Musical Duration of Minute			
	32nd Note (8 per beat)	16th Note (4 per beat)	8th Note Triplet (3 per beat)	8th Note (2 per beat)
1	1800	3600	4800	7200
2	900	1800	2400	3600
4	450	900	1200	1800
6	300	600	800	1200
8	225	450	600	900
10	180	360	480	720
12	150	300	400	600
14	128.6	257.2	342.9	514.3
16	112.5	225	300	450
18	100	200	266.7	400
20	90	180	240	360
22	81.8	163.6	218.2	327.3
24	75	150	200	300
26	69.2	138.4	184.6	276.9
28	64.3	128.6	171.4	257.1
30	60	120	160	240
32	56.3	112.6	150	225
34	52.9	105.8	141.2	211.8
36	50	100	133.3	200
38	47.4	94.8	126.3	189.5
40	45	90	120	180
42	42.9	85.7	114.3	171.4
44	40.9	81.8	109.1	163.6
46	39.1	78.3	104.3	156.5
48	37.5	75	100	150
50	36	72	96	144
52	34.6	69.2	92.3	138.5
54	33.3	66.7	88.9	133.3
56	32.1	64.3	85.7	128.6
58	31	62.1	82.8	124.1
60	30	60	80	120

Tempos are in Beats Per Minute

The HSPL Music Compiler and HSPL Language

The HSPL document does not tell you about some critical stuff, which I will try to explain here.

• To run the compiler from the CLI:

```
HSPL
```

No command line parameters. HSPL takes all of its parameters from the file `sounds.env` in the current directory. This approach was chosen because a piece of music may be compiled hundreds of times as it develops, and we thought that because HSPL can have very many parameters, putting them all in a file (that you edit only when needed) was more sensible than a lengthy command line syntax.

• Consequently, you must have a `sounds.env` file in the current directory before you can use HSPL. You should also have an `sfxlist` file and a `muslist` file (although these names can change; see below). `sounds.env` should contain the parameter settings you want for your piece of music. The Sound Environment Specification Manual describes the contents of this file in a thorough but difficult-to-sort-out way...here's a short version:

- First line can be:

```
Handy
```

  This tells the compiler what version of the SPL language it's compiling, but seems to be omittable.

- If your SPL source calls any HSFX sfx (via the `SFX` command), which by the way are great for drums, then your next line should be:

```
SFXFile sfxlist
```

  You can omit this line if you don't use any `SFX` commands in your piece. This line tells the compiler to look in the file `sfxlist` for a list of binary HSFX sfx files to include in the output. You can use a name other than `sfxlist` if you like. See below for recommended `sfxlist` contents.

- Next line should be:

```
MusicFile muslist
```

  This tells the compiler to look in the file `muslist` for a list of HSPL source files to compile and include in the output. You can use a name other than `muslist` if you like. See below for recommended `muslist` contents.

- Next line should be:

```
ModuleName yourOwnName.dnl
```

  This example tells the compiler to call its output file `yourOwnName.dnl`. You should, of course, change that name to the name of your piece of music.

• The file `sfxlist` (or the file named in the `SFXFile` line in your `sounds.env`) must be a list of the filenames of all binary HSFX sfx (`.bfx` files; see Using HSFX Sfx In HSPL Music below) called by any of the HSPL source files comprising this piece of music. There must be only one filename per line. The files listed here will be included in the compiler's binary output file. For example, if you have lines that say `SFX snare1.bfx`, `SFX omi3.bfx`, and `SFX snare3.bfx` in your `.spl` files, then your `sfxlist` will need to look like this:

```
snare1.bfx
omi3.bfx
snare3.bfx
```

• The file `muslist` (or the file named in the `MusicFile` line in `sounds.env`) must be a list of the filenames of the HSPL (`.spl`) source files for this piece of music. There must be only one filename per line. They will be compiled in the order listed and the results included in the compiler's binary output file. For example, if your piece lives in the files `gooch1.spl`, `gooch2.spl`, `gooch3.spl`, `gooch4.spl`, and `gooch5.spl`, then your `muslist` will need to look like this:

```
gooch1.spl
gooch2.spl
gooch3.spl
gooch4.spl
gooch5.spl
```

The first four files listed in `muslist` (or the file named in the `MusicFile` line in `sounds.env`) will be the four parts that play when the song is started by the game programmer via HMUSIC's PLAYMUSIC command. Beyond the first four, additional part files can be used in at least two ways: the programmer can use HMUSIC's ADDVOICE command to start an individual track at any time, and one HSPL track can call another as a subroutine with the `GSJ` or `GosubSoundJob` statement (see the HSPL document for details).

There is currently no way to start a song with fewer than four voices. To make a voice silent, put only a short rest and an EMS into it.

• For the experienced programmer who knows how to have a good time with such things, the SPL language provides powerful C-like compiler directives, expression evaluation, operators, and conditional compilation (including `#elseif`, `#ifdef`, and `#ifndef`). The compiler also supports defines (=) and parameterized macros. We suggest you have some fun with these; see the HSPL document for details.

• In the SPL language, the `Tempo` or `TPO` statement operates counterintuitively. The parameter specifies the number of audio interrupts per mindur, i.e. it's a delay counter; therefore larger `Tempo` numbers produce slower actual music tempii. See the Musical Timing In SPL section for more.

• The CLI message `Unknown command hspl` can mean that the compiler failed due to one or more syntax errors in an SPL source file, rather than that the OS couldn't find the compiler in the file system. Look for an `SLP` without a matching `LOP`.

• Small changes in `ADSR`, `VFM`, and `VFBM` envelope parameter settings may produce large, discontinuous audible differences, and the audible respose to changes (from compile to compile) may not be linear across the parameter's value range. This is because for each envelope phase the compiler has to divide the difference between the starting and ending values by the phase's duration to find an integer increment rate to use in an HSFX frame. This division can produce roundoff errors. It would have been better, although less musical, for the envelope syntax to have been in terms of the actual keyframe parameters' increment ("interpolation") fields; at least the response would have been predictable. Maybe next rev.

• The compiler will sometimes bomb if you:
  - Don't use the extension `.spl` for your filenames
  - Use non-alphanumeric characters in your filenames (`&` and `-` failed for me; maybe the expression evaluator tried to use them as operators and got confused)
  - Try to `.include` a nonexistant file

• A GSJ will not compile if the name of the called file begins with a numeral.

Using HSFX Sfx In HSPL Music

HSPL lets you call HSFX sfx at musically-synchronized times. This is great for drums, among other things. The HSPL document tells you the syntax for the SFX command, but it doesn't tell you a) that you have to include the `.bfx` suffix in the sfx name (for example, `SFX slime1.bfx, 16`), and b) that, unlike the sfx 6502 source files you give game programmers, these HSPL-called sfx must be in a raw binary format which the HSFX editor does not write directly. To create and use these HSPL-callable `.bfx` files from SPL:

- Use the HSFX editor to create the sounds you want to use, and save them as 6502 text files (`.src` extension) as if you were going to give them to your programmer.

  Example: Save as `slime1.src`

-Then, for each sfx (`.src`) file:

- Use a text editor to remove everything from the top of the file down to, but excluding, the label `SFX_1`, and resave the file under the same name.

- Assemble the `.src` file into a `.bin` file with the same name.

  Example: `asm slime1` (`asm` assumes the `.src` extension if you provide none). This creates `slime1.bin`.

- Strip the ubiquitous 3-word assembler header from the output file, simultaneously creating a `.bfx` file, with `asmstrip`.

  Example: `asmstrip slime1.bin slime1.bfx`

- Use a text editor to add the `.bfx` filename to the `sfxlist` file; if there is no `sfxlist`, create it and add the line `SFXFile sfxlist` to `sounds.env`.

  Example: add `slime1.bfx` to `sfxlist`.

- Use a text editor to add `SFX` statements to your song's HSPL source to call the sfx when desired.

  Example:

```
; Bar 12
SFX slime1.bfx, 16      ; quarter notes
SFX slime1.bfx, 16
SFX slime1.bfx, 8       ; eighth notes
SFX slime1.bfx, 8
SFX slime1.bfx, 4       ; sixteenth notes
SFX slime1.bfx, 4
SFX slime1.bfx, 4
SFX slime1.bfx, 4
```

Tip: To make global changes in SFX sounds really easy, create an SPL macro for each sound you're using...something like this:

```
#macro bd ?0              ; Bass Drum macro
   SFX slime1.bfx, ?0
#endm
```

Now you can do things like this...

```
bd 2   ; 1
hh 2
hh 2
hh 2
sd 2   ; 2
hh 2
hh 2
hh 2
bd 2   ; 3
hh 2
bd 2
hh 2
sd 2   ; 4
hh 2
sd 2
sd 2
```

...and be able to change your drum sounds by simply changing the .bfx filename in the macro definition to the name of another file. I *believe* the compiler won't bomb if you then define, for example, the above bar as a (parameterless) macro, which would let you cut & paste your drum patterns in terms of bars or patterns.

play: Easy Music Auditioning System

It often takes hundreds of compiles to get a piece of music sounding the way you want. Here's how we make it go faster.

- Before starting, make sure the virtual disk volume `vdk:` contains the following files:

```
asm              hspl
lx.dnl           LXfast.bin
lxshell.bin      LXshell.src
lxshell.sym      play
```

  These can be found in the directory `sound:`. Our `s/startup-sequence.epyx` copies these to `vdk:` for us.

- Also make sure Handebug is running and conversing with your Lynx development board, and that its Download requester is set as follows:

  -Name: `LxShell.bin`
  -Drawer:<nothing>
  -Disk: `vdk:`
  -Load & Go, not Load & Wait
  -Formatted, not Unformatted

Once set up thus, the following (short!) sequence will compile and download the HSPL music in the current directory:

1. At the CLI, type `play yourName` where `yourName` is the `ModuleName` you set in your `sounds.env` file. This will run the HSPL compiler from `vdk:` and copy its output to `vdk:lx.dnl`. We use the command line history editor ConMan to avoid having to type `play yourName` over and over again.

2. Use the Window-to-Back gadget(s) to bring up the Handebug screen, click Download and then click OK!.

The music will play. Sometimes you have to hit NMI once before Downloading.

You can also do like we do and keep your multitasking-compatible text editor (we use CED) up during these steps, saving changes to disk before each compile, so that you can edit-compile-download in a (sometimes seemingly...) endless cycle.

end of document