Sequencer64 (seq64) Developer Notes

0.91.0

Generated by Doxygen 1.8.14

Contents

Chapter 1

Sequencer64 Developer Notes

Author(s) Chris Ahlstrom 2017-04-30

1.1 Introduction

Sequencer64 is a major cleanup, refactoring, and documentation of the *Seq24* live-play MIDI sequencer.

The current document, generated by Doxygen, describes the approaches and issues for various problems we have solved in *Sequencer64*.

Also read the ROADMAP, README, and contrib/bugs_to_investigate files to understand the genesis of this project and the things that still need to be done with Sequencer64.

Also, we have pretty deeply documented *Seq24* and *Sequencer64* with PDF files that can be generated by git-cloning the following projects, installing a number of tools related to PDF and LaTeX, and running "make":

- https://github.com/ahlstromcj/seq24-doc.git
- https://github.com/ahlstromcj/sequencer64-doc.git

These project also have prebuilt PDFs should one not want to bother building them.

Some useful notess:

- http://acad.carleton.edu/courses/musc108-00-f14/pages/04/04StandardMIDIFiles.html
- http://www.midimusicadventures.com/qs/midi-zips/soundtracks/kq6gm.zip

Chapter 2

MIDI File Parsing in Sequencer64

Author(s) Chris Ahlstrom 2018-01-11

2.1 Introduction

This section describes the parsing of a MIDI file (and a few other topics). We wanted to add the reading of SMF 0 files to *Sequencer64*. We started with the main format that is supported, SMF 1. Once we understood that we, we figured out how to split a SMF 0 tracks correctly.

We split the midifile::parse() function into two sections. The first section analyzes the header of the MIDI. Then, based on whether the file is SMF 1 (the normal case) or SMF 0, either the parse_smf_1() function of or the parse_smf_0() function is called. The parse_smf_0() function creates one sequence object per channel present in the SMF 0 file, plus the original track. The last pattern slot (sequence 16) will contain the original track data, and the rest will contain common data and then channel data for each channel. After the parsing is done, all the tracks (including the original track) will be added to the performance. The user then has the option of deleting the original track, which will be the last track.

One note about the portability of *Sequencer64* MIDI files. The *Rosegarden* sequencer reads them well, as does *Ardour*, which also plays it properly (ignoring the triggers). The *Python* program *mido3-player* does not handle the sequence number (0x3FFF) of the proprietary track, but that is really a bug in mido.

Another thing to note is that the sorting of the events can vary from save to save, which is harmless, if annoying to the programmers.

2.2 SMF 1 Parsing

This section describes the parsing of the header chunk, MThd, and the track chunk, MTrk.

The midifile::parse() function starts by opening the MIDI file, getting its file-size, pre-allocating the data vector to that size, reading all of the characters into that vector, and then closing the file.

2.2.1 MIDI File Header, MThd

The data of the header is read:

```
Header ID:     "MThd"         read_long()       4 bytes
MThd length:     6            read_long()       4 bytes
Format:        0, 1, 2        read_short()      2 bytes
No. of track:  1 or more      read_short()      2 bytes
PPQN:           192           read_short()      2 bytes
```

The header ID and it's length are always the same values. The formats that Sequencer64 supports are 0 or 1. SMF 0 has only one track, while SMF 1 can support an arbitary number of tracks. The last value in the header is the PPQN value, which specifies the "pulses per quarter note", which is the basic time-resolution of events in the MIDI file. Common values are 96 or 192, but higher values are also common. Sequencer64 and its precursor, Seq24, default to 192.

2.2.2 MIDI Track, MTrk

Sequencer64 next reads the tracks specified in the file. Each track is assumed to cover a different MIDI channel, but always the same MIDI buss. (The MIDI buss is not a data item in standard MIDI files, but it is a special data item in Seq24/Sequencer64 MIDI files.) Each track is tagged by a standard chunk marker, "MTrk". Other markers are possible, and are to be ignored, if nothing else. Here are the values read at the beginning of a track:

```
Track ID:      "MTrk"         read_long()       4 bytes
Track length:  varies         read_long()       4 bytes
```

The track length is the number of bytes that need to be read in order to get all of the data in the track.

Next, a new sequence object is created, with the PPQN value passed to its constructor. The sequence then is hooked to the master MIDI buss object. The "RunningTime" accumulator is set to 0 for that track.

Next, the parse() function loops through the rest of the track, reading data and logging it to the sequence. Let's go through the loop, which is the meat of the processing.

TODO: An empty event is created before track processing, and re-used for every track and event. This seems dangerous. We moved the event constructor two levels of nesting deeper, and it seems to work fine.

Delta time. The amount time that passes from one event to the next is the *delta time*. For some events, the time doesn't matter, and is set to 0. This values is a *variable length value*, also known as a "VLV" or a "varinum". It provides a way of encoding arbitrarily large values, a byte at a time. For now, just note that a varinum is 1 or more bytes, and MIDI provides a way to tell when the varinum is complete.

```
Delta time:    varies         read_varinum()    1 or more bytes
```

2.2.2.1 Channel Events

Status. The byte after the delta time is examined by masking it against 0x80 to check the high bit. If not set, it is a "running status", it is replaced with the "last status", which is 0 at first.

```
Status byte:   varies         read_byte()       1 byte
```

If the high bit is set, it is a status, and is passed to the setter `event::set_status()`.

The "RunningTime" accumulator is incremented by the delta-time. The current time is adjusted as per the PPQN ratio, if needed, and passed to the setter `event::set_timestamp()`.

Now what does the status mean? First, the channel part of the status is masked out using the 0xF0 mask.

If it is a 2-data-byte event (note on, note off, aftertouch, control-change, or pitch-wheel), then the two data bytes are read:

```
Data byte 0:   varies         read_byte()       1 byte
Data byte 1:   varies         read_byte()       1 byte
```

If the status is a note-on event, with data[1] = 0, then it is converted to a note-off event, a fix for the output quirks of some MIDI devices, and the status of the event is amended to EVENT_NOTE_OFF.

If it is a 1-data-btye event (program change or channel pressure), then only data byte 0 is read.

Then the one or two data bytes are added to the event by overloads of `event::set_data()`, the event is added to the current sequence by `sequence::add_event()`, and the MIDI channel of the sequence is set by `sequence::set_midi_channel()`.

Note that this is the point where parsing could detect a change in channel, and select a new sequence to support that channel, and add the events to that sequence, if the file were SMF 0.

Also note that the channel of the sequence is set every a new channel event/status is read. This should be done once, and then simply warned about if a non-matching channel occurs.

Lastly, note that it might be better to do the sequence function calls at the end of processing the event.

2.2.2.2 Meta Events

If the event status masks off to 0xF0 (0xF0 to 0xFF), then it is a meta event. If the status is 0xFF, it is called a "Sequencer-specific", or "SeqSpec" event. For this kind of event, then a type byte and the length of the event are read.

```
Meta type:     varies         read_byte()       1 byte
Meta length:   varies         read_varinum()    1 or more bytes
```

If the type of the SeqSpec (0xFF) meta event is 0x7F, parsing checks to see if it is one of the Seq24 "proprietary" events. These events are tagged with various values that mask off to 0x24240000. The parser reads the tag:

```
Prop tag:     0x242400nn      read_long()       4 bytes
```

These tags provide a way to save and recover Seq24/Sequencer64 properties from the MIDI file: MIDI buss, MIDI channel, time signature, sequence triggers, and (new), the key, scale, and background sequence to use with the track/sequence. Any leftover data for the tagged event is let go. Unknown tags ate skipped.

If the type of the SeqSpec (0xFF) meta event is 0x2F, then it is the End-of-Track marker. The current time is set using `sequence::set_length()` and then `sequence::zero_markers()` is called, and parsing is done for that track.

If the type of the SeqSpec (0xFF) meta event is 0x03, then it is the sequence name. The "length" number of bytes are read, and loaded by `sequence::set_name()`.

If the type of the SeqSpec (0xFF) meta event is 0x00, then it is the sequence number, which is read:

```
Seq number:    varies         read_short()      2 bytes
```

Note that the sequence number might be modified latter to account for the current screenset in force for a file import operation.

Anything other SeqSpec type is simply skipped by reading the "length" number of bytes.

To summarize the process, here are the relevant event and sequence setter calls typically made while parsing a MIDI track:

1. `perform::add_sequence()`
   (a) `sequence::sequence()`
   (b) `sequence::set_master_midi_bus())`
   (c) `sequence::add_event()`
      i. `event::event()`
      ii. `event::set_status()`
      iii. `event::set_timestamp()`
      iv. `event::set_data()`
   (d) `sequence::set_midi_channel()`
   (e) `sequence::set_length()`
   (f) `sequence::zero_markers()`
   (g) `sequence::set_name()`
   (h) `sequence::set_midi_bus()`
2. `xxxxx::yyyy()`

2.2.3 Meta Events Summary

Here, we summarize the MIDI meta events for your edification.

1. `FF 00 02 ssss`: Sequence Number.
2. `FF 01 len text`: Text Event.
3. `FF 02 len text`: Copyright Notice.
4. `FF 03 len text`: Sequence/Track Name.
5. `FF 04 len text`: Instrument Name.

6. `FF 05 len text`: Lyric.
7. `FF 06 len text`: Marker.
8. `FF 07 len text`: Cue Point.
9. `FF 08 len text`: Patch/program Name.
10. `FF 09 len text`: Device Name.
11. `FF 0A through 0F len text`: Other kinds of text events.
12. `FF 20 01 cc`: MIDI channel (obsolete, used by Cakewalk)
13. `FF 21 01 pp`: MIDI port (obsolete, used by Cakewalk)
14. `FF 2F 00`: End of Track.
15. `FF 51 03 tttttt`: Set Tempo, us/qn.
16. `FF 54 05 hr mn se fr ff`: SMPTE Offset.
17. `FF 58 04 nn dd cc bb`: Time Signature.
18. `FF 59 02 sf mi`: Key Signature.
19. `FF 7F len data`: Sequencer-Specific.

The next sections describe the events that *Sequencer* tries to handle. These are

- Sequence Number (0x00)
- Track Name (0x03)
- End-of-Track (0x2F)
- Set Tempo (0x51) (Sequencer64 only)
- Time Signature (0x58) (Sequencer64 only)
- Sequencer-Specific (0x7F)
- System Exclusive (0xF0) Sort of handled, functionality incomplete..

2.2.3.1 Sequence Number (0x00)

```
FF 00 02 ss ss
```

This optional event must occur at the beginning of a track, before any non-zero delta-times, and before any transmittable MIDI events. It specifies the number of a sequence.

2.2.3.2 Track/Sequence Name (0x03)

```
FF 03 len text
```

If in a format 0 track, or the first track in a format 1 file, the name of the sequence. Otherwise, the name of the track.

2.2.3.3 End of Track (0x2F)

```
FF 2F 00
```

This event is not optional. It is included so that an exact ending point may be specified for the track, so that it has an exact length, which is necessary for tracks which are looped or concatenated.

2.2.3.4 Set Tempo Event (0x51)

The MIDI Set Tempo meta event sets the tempo of a MIDI sequence in terms of the microseconds per quarter note. This is a meta message, so this event is never sent over MIDI ports to a MIDI device.

After the delta time, this event consists of six bytes of data:

```
FF 51 03 tt tt tt
```

Example:

```
FF 51 03 07 A1 20
```

1. 0xFF is the status byte that indicates this is a Meta event.
2. 0x51 the meta event type that signifies this is a Set Tempo event.
3. 0x03 is the length of the event, always 3 bytes.
4. The remaining three bytes carry the number of microseconds per quarter note. For example, the three bytes above form the hexadecimal value 0x07A120 (500000 decimal), which means that there are 500,000 microseconds per quarter note.

Since there are 60,000,000 microseconds per minute, the event above translates to: set the tempo to 60,000,000 / 500,000 = 120 quarter notes per minute (120 beats per minute). This is a 24-bit binary value, so each byte covers the full range of 0x00 to 0xFF.

This event normally appears in the first track. If not, the default tempo is 120 beats per minute. This event is important if the MIDI time division is specified in "pulses per quarter note", which does not itself define the length of the quarter note. The length of the quarter note is then determined by the Set Tempo meta event.

Representing tempos as time per beat instead of beat per time allows absolutely exact DWORD-term synchronization with a time-based sync protocol such as SMPTE time code or MIDI time code. This amount of accuracy provided by this tempo resolution allows a four-minute piece at 120 beats per minute to be accurate within 500 usec at the end of the piece.

We have now added the Tempo meta event (and the Time Signature meta event) to the track, which allows other sequencers to obtain these values from a Sequencer64 MIDI file. Here are the original headers for a normal MIDI file and its legacy (Seq24) conversion, as shown by the *midicvt* application:

```
hymne.asc                           hymne-ppqn-384.asc

MThd 1 4 96                         MThd 1 4 384
MTrk                                MTrk
0 Meta SeqName "Vangelis: Hymne"    0 SeqNr 0
0 TimeSig 4/4 24 8                  0 Meta SeqName "Vangelis: Hymne"
0 Tempo 750000                      0 SeqSpec 24 24 00 08       (no triggers)
0 Meta TrkEnd                       0 SeqSpec 24 24 00 01 00    (MIDI buss 0)
TrkEnd                              0 SeqSpec 24 24 00 06 04 04 (beats, width)
                                    0 SeqSpec 24 24 00 02 00    (MIDI ch. 0)
                                    96 Meta TrkEnd
                                    TrkEnd
```

(*midicvt* is available at https://github.com/ahlstromcj/midicvt.)

Here is the header data that result from the new conversion, which is used if the "legacy" option is not in force:

```
MThd 1 4 192
MTrk
0 SeqNr 0
0 Meta SeqName "Vangelis: Hymne"
0 TimeSig 4/4 24 8
0 Tempo 750000
0 SeqSpec 24 24 00 08
0 SeqSpec 24 24 00 01 00
0 SeqSpec 24 24 00 06 04 04
0 SeqSpec 24 24 00 02 00
48 Meta TrkEnd
TrkEnd
```

2.2.3.5 Time Signature Event (0x58)

After the delta time, this event consists of seven bytes of data:

```
FF 58 04 nn dd cc bb
```

The time signature is expressed as four numbers. `nn` and `dd` represent the numerator and denominator of the time signature as it would be notated. The numerator counts the number of beats in a measure (beats per measure or beats per bar). The denominator is a negative power of two: 2 represents a quarter-note, 3 represents an eighth-note, etc. The denominator specifies the unit of the beat (e.g. 4 or 8). In Seq24/Sequencer64, this value is also called the "beat width".

The `cc` parameter expresses the number of MIDI clocks (or "ticks", or "pulses") in a metronome click. The standard MIDI clock ticks 24 times per quarter note, so a value of 6 would mean the metronome clicks every 1/8th note. A `cc` value of 6 would mean that the metronome clicks once every 1/8th of a note (quaver). This MIDI clock is different from the clock (PPQN) that determines the start time and duration of the notes.

The `bb` parameter expresses the number of notated 32nd-notes in a MIDI quarter note (24 MIDI Clocks). The usual value for this parameter is 8, though some sequencers allow the user to specify that what MIDI thinks of as a quarter note, should be notated as something else. For example, a value of 16 means that the music plays two quarter notes for each quarter note metered out by the MIDI clock, so that the music plays at double speed.

Examples:

```
FF 58 04 04 02 18 08
```

1. 0xFF is the status byte that indicates this is a Meta event.
2. 0x58 the meta event type that signifies this is a Time Signature event.
3. 0x04 is the length of the event, always 4 bytes.
4. 0x04 is the numerator of the time signature, and ranges from 0x00 to 0xFF.
5. 0x02 is the log base 2 of the denominator, and is the power to which 2 must be raised to get the denominator. Here, the denominator is 2 to 0x02, or 4, so the time signature is 4/4.
6. 0x18 is the metronome pulse in terms of the number of MIDI clock ticks per click. Assuming 24 MIDI clocks per quarter note, the value here (0x18 = 24) indidicates that the metronome will tick every 24/24 quarter note. If the value of the sixth byte were 0x30 = 48, the metronome clicks every two quarter notes, i.e. every half-note.

7. 0x08 defines the number of 32nd notes per beat. This byte is usually 8 as there is usually one quarter note per beat, and one quarter note contains eight 32nd notes.

A time signature of 6/8, with a metronome click every 3rd 1/8 note, would be encoded:

```
FF 58 04 06 03 24 08
```

Remember, a 1/4 note is 24 MIDI Clocks, therefore a bar of 6/8 is 72 MIDI Clocks. Hence 3 1/8 notes is 36 (=0x24) MIDI Clocks.

There should generally be a Time Signature Meta event at the beginning of a track (at time = 0), otherwise a default 4/4 time signature will be assumed. Thereafter they can be used to effect an immediate time signature change at any point within a track.

For a format 1 MIDI file, Time Signature Meta events should only occur within the first MTrk chunk.

If a time signature event is not present in a MIDI sequence, 4/4 signature is assumed.

In *Sequencer64*, the c_timesig SeqSpec event is given priority. The conventional time signature is used only if the c_timesig SeqSpec is not present in the file. NEEDS TO BE TESTED.

2.2.3.6 SysEx Event (0xF0)

If the meta event status value is 0xF0, it is called a "System-exclusive", or "SysEx" event.

```
F0 len data F7
```

Sequencer64 has some code in place to store these messages, but the data is currently not actually stored or used. Although there is some infrastructure to support storing the SysEx event within a sequence, the SysEx information is simply skipped. *Sequencer64* warns if the terminating 0xF7 SysEx terminator is not found at the expected length. Also, some malformed SysEx events have been encountered, and those are detected and skipped as well.

2.2.3.7 Sequencer Specific (0x7F)

This data, also known as SeqSpec data, provides a way to encode information that a specific sequencer application needs, while marking it so that other sequences can safely ignore the information.

```
FF 7F len data
```

In *Seq24* and *Sequencer64*, the data portion starts with four bytes that indicate the kind of data for a particular SeqSpec event:

```
c_midibus        ^  0x24240001  Track buss number
c_midich         ^  0x24240002  Track channel number
c_midiclocks     *  0x24240003  Track clocking
c_triggers       ^  0x24240004  See c_triggers_new
c_notes          *  0x24240005  Song data, notes
c_timesig        ^  0x24240006  Track time signature
c_bpmtag         *  0x24240007  Song beats/minute
c_triggers_new   ^  0x24240008  Track trigger data
c_mutegroups     *  0x24240009  Song mute group data
c_midictrl       *  0x24240010  Song MIDI control
c_musickey       +  0x24240011  Track key (Sequencer64 only)
c_musicscale     +  0x24240012  Track scale (Sequencer64 only)
c_backsequence   +  0x24240013  Track background sequence (Sequencer64 only)

* = global only; ^ = track only; + = both
```

In *Seq24*, these events are placed at the end of the song, but are not marked as SeqSpec data. Most MIDI applications handle this situation fine, but some (e.g. *midicvt*) do not. Therefore, *Sequencer64* makes sure to wrap each data item in the 0xFF 0x7F (SeqSpec) wrapper.

Also, the last three items above (key, scale, and background sequence) can also be stored (by *Sequencer64*) with a particular sequence/track, as well as at the end of the song. Not sure if this bit of extra flexibility is useful, but it is there.

2.2.3.8 Non-Specific End of Sequence

Any other statuses are deemed unsupportable in *Sequencer64*, and abort parsing with an error.

If the –bus option is in force, `sequence::set_midi_bus()` is called to override the buss number (if any) stored with the sequence.

Finally, `perform::add_sequence()` adds the sequence to the encoded tune.

2.3 SMF 0 Parsing

After parsing SMF 1 track data, we end up with a number of sequences, each on a different MIDI channel. With SMF 0, data for all channels is present in a single track. Sequencer64 will read SMF 0 data, but we really need to be able to have one MIDI channel per track. So we need to take the data from the sequence and use it to make more sequences.

- `sequence::add_event()`.
- `sequence::set_midi_channel()`.
- `sequence::set_length()`.
- `sequence::set_midi_bus()`.
- `perform::add_sequence()`.

This code basically works. For now, please look at the source code for more details. Also, the reading of SMF 0 MIDI files is described in the *sequencer64-doc* project on GitHub.

2.4 Running Status

When we apply the *midicvt* application to a file saved by *Sequencer64*, we can end up with a successful ASCII conversion that ends with an error message:

```
$ midicvt hymne-seq64.midi -o hymne-seq64.asc
? Error at MIDI file offset 12155 [0x2f7b]
Error: Garbage at end 'readtrack(): unexpected running status'
```

Is this a problem in *midicvt* or *Sequencer4*? Let's learn about running status.

Running status is a way to speed up the sending of MIDI bytes to a synthesizer or sequencer by taking advantage of redundancy where possible. For example, if we're sending a consecutive group of Note On and Note Off messages to a particular channel, we can save some time by not sending the channel status byte after the first time. Here's an example with Note On on channel 1:

```
0x90 3C 7F
0x90 40 7F
0x90 43 F3
```

Since no change in status occurs after the first of these three events, we can drop the subsequent status bytes:

```
0x90 3C 7F
40 7F
43 F3
```

The 0x90 byte is saved in a "running status buffer" (RSB), and is filled in by the receiving device.

Here is the sequence of events for operating with running status.

1. Clear the RSB buffer (RSB = 0) to start.
2. If a **Voice Category Status** (VCS) byte is received, then set RSB = VCS. VCS bytes range from 0x80 to 0xEF. This is binary 1000000 to 11100000.
3. If a data byte is received (data bytes range from 0x00 to 0x7F, binary 0000000 to 0111111; that is, bit 7 is always 0 in a data byte):
   (a) If RSB != 0, first insert the RSB into the incoming data stream, then insert the data byte.
   (b) If RSB == 0, then just insert the data byte into the incoming data stream.
4. Clear the RSB buffer (RSB = 0) when a System Common Message (SCM) status byte is received. SCM bytes range from 0xF0 to 0xF7.
5. The message after an SCM **must** begin with a status byte. That is a byte with bit 7 set.
6. Do no special action when a Realtime Category Message (RCM) byte is received. RCM bytes range from 0xF8 to 0xFF.

Note that some events, such as Tempo, assume that its bytes are all data bytes.

2.5 MIDI Save Changes

Things have progressed over the months with the conversion and saving of MIDI files. For example, when we read the `CountryStrum.mid` file and save it, in recent versions of *Sequencer64* (e.g. 0.94.0 and above), the new version is a lot longer (59 Kbytes) than the version saved previously (33 Kbytes). What accounts for this difference?

Well, converting both versions to ASCII via *midicvt* reveals that the newer version of *Sequencer64* adds missing Note Off events to the file. And there are a lot of them! Almost doubles the size of the file, naturally.

In addition, SeqSpec `24 24 00 02` (channel number has been added to some tracks). SeqSpec `24 24 00 08` adds data that wasn't output before. And SeqSpec `24 24 00 14` (sequence transpose) has been added. SeqSpec `24 24 00 07` (song BPM) now shows the proper BPM from the conversion. Finally, the magic number for the final, proprietary track has been changed from `0x7777` to the MIDI-compliant value of `0x3FFF`.

Another bunch of data, `24 24 00 09` (mute groups), is still present, but there are now options to eliminate its 4096 bytes of data.

Chapter 3

JACK, Live, and Song Modes in Sequencer64

Author(s) Chris Ahlstrom 2017-05-01

3.1 Introduction

This section describes the interactions between JACK settings and the Live/Song Mode settings, with an eye to describing the proper behavior of Sequencer64 with JACK settings, how the Live/Song modes are supposed to work, and what bugs or issues remain in Sequencer64's JACK handling.

There is currently no description of the 0.90.x line's native JACK MIDI support, except what is found in the developer's reference manual and in the updated sequencer64-doc project at GitHub. We will rectify that at some point.

3.2 JACK Functions

Please study the following URL and note these important points:

```
http://jackaudio.org/files/docs/html/transport-design.html
```

- The timebase master continuously updates position information, beats, timecode, etc. There is at most one master active at a time. If no client is registered as timebase master, frame numbers will be the only position information available.
- The timebase master registers a callback that updates position information while transport is rolling. Its output affects the following process cycle. This function is called immediately after the process callback in the same thread whenever the transport is rolling, or when any client has set a new position in the previous cycle.
- Clients that don't declare a sync callback are assumed ready immediately, anytime the transport wants to start. If a client doesn't require slow-sync processing, it can set its sync callback to NULL.
- The transport state is always valid; initially it is JackTransportStopped.
- When someone calls `jack_transport_start()`, the engine resets the poll bits and changes to a new state, JackTransportStarting.
- When all slow-sync clients are ready, the state changes to JackTransportRolling.

Does Sequencer64 need a latency callback?

```
http://jackaudio.org/files/docs/html/group__ClientCallbacks.html
```

The next section provide summaries of some the JACK functions used in the jack_assistant and (soon) the midi_jack modules.

3.2.1 jack_client_open ()

Open a client session with a JACK server. More complex and powerful than `jack_client_new()`. Clients choose which of several servers to connect, and how to start the server automatically, if not already running. There is also an option for JACK to generate a unique client name.

```
const char *   client_name,
jack_options_t    options,
jack_status_t *   status,
   ...
```

client_name of at most jack_client_name_size() characters. The name scope is local to each server. Unless forbidden by the JackUseExactName option, the server will modify this name to create a unique variant, if needed.

options formed by OR-ing together JackOptions bits. Only the JackOpenOptions bits are allowed.

status (if non-NULL) an address for JACK to return information from the open operation. This status word is formed by OR-ing together the relevant JackStatus bits.

Optional parameters: depending on corresponding [options bits] additional parameters may follow status (in this order).

[JackServerName] (char $*$) server_name selects from among several possible concurrent server instances. Server names are unique to each user. If unspecified, use "default" unless $JACK_DEFAULT_SERVER is defined in the process environment.

It returns an opaque client handle if successful. If this is NULL, the open operation failed, and $*$status includes JackFailure, and the caller is not a JACK client.

3.2.2 jack_on_shutdown ()

Registers a function to call when the JACK server shuts down the client thread. It must be an asynchonrous POSIX signal handler: only async-safe functions, executed from another thread. A typical function might set a flag or write to a pipe so that the rest of the application knows that the JACK client thread has shut down. Clients do not need to call this function. It only helps clients understand what is going on. It should be called before `jack_client_↩ activate()`.

3.2.3 jack_set_sync_callback ()

Register/unregister as a slow-sync client; it can't respond immediately to transport position changes. The callback is run at the first opportunity after registration: if the client is active, this is the next process cycle, otherwise it is the first cycle after `jack_activate()`. After that, it runs as per JackSyncCallback rules. Clients that don't set this callback are assumed ready immediately any time the transport wants to start.

3.2.4 jack_set_process_callback ()

Tells the JACK server to call the callback whenever there is work. The function must be suitable for real-time execution, it cannot call functions that might block for a long time: malloc(), free(), printf(), pthread_mutex_lock(), sleep(), wait(), poll(), select(), pthread_join(), pthread_cond_wait(), etc. In the current class, this function is a do-nothing function.

3.2.5 jack_set_session_callback ()

Tells the JACK server to call the callback when a session event is delivered. Setting more than one session callback per process is probably a design error. For a multiclient application, it's more sensible to create a JACK client with only one session callback.

3.2.6 jack_activate ()

Tells the JACK server that the application is ready to start processing.

3.2.7 jack_release_timebase ()

TODO

3.2.8 jack_client_close ()

TODO

3.2.9 jack_transport_start ()

Starts the JACK transport rolling. Any client can make this request at any time. It takes effect no sooner than the next process cycle, perhaps later if there are slow-sync clients. This function is realtime-safe. No return code.

3.2.10 jack_transport_stop ()

Starts the JACK transport rolling. Any client can make this request at any time. This function is realtime-safe. No return code.

3.2.11 jack_transport_locate ()

Repositions the transport to a new frame number. May be called at any time by any client. The new position takes effect in two process cycles. If there are slow-sync clients and the transport is already rolling, it will enter the JackTransportStarting state and begin invoking their sync_callbacks until ready. This function is realtime-safe.

3.2.12 jack_transport_reposition ()

Request a new transport position. May be called at any time by any client. The new position takes effect in two process cycles. If there are slow-sync clients and the transport is already rolling, it will enter the JackTransport↩ Starting state and begin invoking their sync_callbacks until ready. This function is realtime-safe. This call, made in the position() function, is currently disabled.

3.2.13 jack_transport_query ()

Query the current transport state and position. This function is realtime-safe, and can be called from any thread. If called from the process thread, pos corresponds to the first frame of the current cycle and the state returned is valid for the entire cycle.

The first parameter is the client, which is a pointer to the JACK client structure.

The second parameter is a pointer to structure for returning current transport position; pos->valid will show which fields contain valid data. If pos is NULL, do not return position information.

This function returns the current transport state.

3.3 Modes Operation

3.3.1 No JACK, Live Mode

In ~/.config/sequencer64/sequencer64.rc, set:

- jack_transport = 0
- jack_master = 0
- jack_master_cond = 0
- song_start_mode = 0

By changing the start mode to 0 (false), Sequencer64 is put into Live Mode. With this setting, control of the muting and unmuting of patterns resides in the main window (the patterns window). One can start the playback in the performance (song) window, but it will not affect which patterns play, at all.

Note that this option is part of the *File / Options / JACK/LASH* configuration page.

3.3.2 No JACK, Song Mode

In ~/.config/sequencer64/sequencer64.rc, set:

- jack_transport = 0
- jack_master = 0
- jack_master_cond = 0
- song_start_mode = 1

By changing the start mode to 1 (true), Sequencer64 is put into Song Mode.

With this setting, control of the muting and unmuting of patterns resides in the song window (the performance window). The patterns shown in the pattern slots of the main window turn on and off whenever the progress bar is in the pattern as drawn in the perforance window.

Note that this option is part of the *File / Options / JACK/LASH* configuration page.

3.3.3 JACK Transport

In ~/.config/sequencer64/sequencer64.rc, set:

- jack_transport = 1
- jack_master = 0
- jack_master_cond = 0
- song_start_mode = 0 or 1 (see previous section)

The current behavior is that qjackctl and sequencer64 playback/progress seem to be independent of each other.

The workaround seems to be to set seq24/sequencer64 as JACK Master, or if another *application* (e.g. Qtractor) is JACK Master.

OLD BEHAVIOR. Start qjackctl, verify that it sets up correctly, then click it's "play" button to start the transport rolling. Run sequencer64, load a file. Then note that starting playback (whether in the main window or in the performance window) is ineffective, but resets the time counter in qjackctl. Why? With JACK sync enabled by the macro:

```
[JACK transport slave]
jack sync(): zero frame rate [single report]!?
[JackTransportRolling]
[JackTransportStarting]  (every time space bar pressed)
[Start playback]
. . .
```

End of OLD BEHAVIOR.

3.4 Breakage

Old message about seq24 being broken appears here.

http://lists.linuxaudio.org/pipermail/linux-audio-user/2010-November/073848.html

```
i dont see the transport synchronisation working with a jack1 svn version.
you are still using only a sync callback.

and you are relying on the transport to go through the
JackTransportStarting state.

this issue should be fixed.
iirc we came to the conclusion, that seq24 is broken, and we will not
revert the changes in jack, which break it.

the quick and dirty fix on your side, would be to register an empty
process_callback.

but the issue still remains. seq24 is NOT a slow sync client. but it
registers a sync_callback.
and it even takes a lock in the sync callback.

the patch for jack-session support didnt get merged either.
```

Another one (no need for a URL):

```
I use seq24 for the majority of my projects but it isn't ideal (I should
point out that I never finish anything). I don't like seq24's pianoroll
editor, the way you do CC envelopes isn't ideal, it uses alsa-midi, there's
unnecessary complexity in switching from pattern-trigger mode to song mode,
and its insistence on being transport master while not even being able to
adjust tempo when live is annoying
```

3.5 JACK References

- http://libremusicproduction.com/articles/demystifying-jack-%E2%80%93-beginners-guide
- http://jackaudio.org/files/docs/html/transport-design.html
- http://kxstudio.linuxaudio.org/Repositories
- http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothi
- http://vagar.org/wordpress/?tag=jack-midi

Chapter 4

User Testing of Sequencer64 with Yoshimi

Author(s) Chris Ahlstrom 2017-06-11

4.1 Introduction

This section describes user testing of Sequencer64 using Yoshimi. It will expand as we work our way through all the many use-cases that can be achieved with Sequencer64 and Yoshimi.

Please note that the most advanced and recent testing can be found currently in the document `contrib/notes/jack-testing.txt`. We will eventually merge the final tests here... someday.

Also note that the default executable is no long `sequencer64`... it is now `seq64`. Furthermore, this new version has a native JACK MIDI feature that is not yet part of the testing! So much to catch up with!

4.2 Smoke Test

Every so often we run Sequencer64 with a software synthesizer to make sure we haven't broken any functionality via our major refactoring efforts. We call it a "smoke test". We fire up the two applications, and see if anything smokes.

This smoke test sets up Yoshimi with a very simple ALSA setup, and no instruments are loaded. Instead, only the "Simple Sound" is used on all channels. We've been doing this test with Yoshimi 1.3.6. The current Debian Sid ("testing") version of Yoshimi is 1.3.6-2, pulled from SourceForge. It seems to have issues, so we've been cloning and pulling the code from:

```
https://github.com/Yoshimi/yoshimi.git
```

After getting the application build and installed, the next step is to run it, using ALSA for MIDI and for audio:

```
$ yoshimi -a -A
```

Next, fix up the configuration files for Sequencer64, `~/.config/sequencer64/sequencer64.rc` and `~/.config/sequencer64/sequencer64.usr`.

First hide `sequencer64.usr` somewhere, or delete it, as it will determine what MIDI devices are available, and we don't want that (yet). Second, make sure that `sequencer64.rc` makes the following setting:

```
[manual-alsa-ports]

# Set to 1 if you want sequencer64 to create its own ALSA ports and
# not connect to other clients

0   # number of manual ALSA ports
```

Next, run the newly-built version of Sequencer64. If desired, use the –bus option described below to force the buss number to the buss you need, as shown in the second version of the command:

```
$ sequencer64/sequencer64 &
$ sequencer64/sequencer64 --bus 5 &
```

In *File / Options / MIDI Clock*, observe the MIDI inputs made available by your system. Our system shows:

```
[0] 14:0 (Midi Through Port-0)
[1] 128:0 (TiMidity port 0)
[2] 128:0 (TiMidity port 1)
[3] 128:0 (TiMidity port 2)
[4] 128:0 (TiMidity port 3)
[5] 129:0 (input)
```

For some reason (a bug in Yoshimi?), input "[5]" doesn't indicate that it is Yoshimi, but it is. Take note of that input number... that is the MIDI buss number that is needed to drive Yoshimi.

Also make sure that of the clock settings for those busses are "Off".

The next instruction still works, but it is easier to simply pass the option `–bus 5` to Sequencer64 when starting it up.

Now open the file `sequencer64/contrib/midi/b4uacuse-GM-format.midi` in Sequencer64. For all of the patterns (slots) that have lots of data in them, right click on the pattern and select *Midi Bus / [5] 129:0 (input)* and the desired channel number. (Doesn't matter much, just use up the lower channel numbers first).

Back in Yoshimi, select each Part corresponding to the channels you selected. Make sure *Enabled* is checked for each desired channel.

Back in Sequencer64, click on each pattern you want to hear, which highlights them in black. Now click the play button (green triangle). The song should play, with each part using the "Simple Sound". Not too bad for a bunch of sine waves, eh?

Now we can test the application more fully. Note that the instructions here are very light. Detailed instructions on the usage of Sequencer64 can be found in the following project, which contains a PDF file and the LaTeX code used to build it:

```
https://github.com/ahlstromcj/sequencer64-doc.git
```

Although it applies to an earlier version of the project, it still mostly holds true for Sequencer64.

4.3 Tests in the Patterns Window

The Patterns window is the inside portion of the main window, supported by the `mainwid` class. it contains a grid of boxes or slots, with each slot potentially containing a pattern, sequence, or track. Empty tracks (i.e. tracks that contain no events, like title-only tracks) are highlighted in yellow.

This window supports only a single variant of mouse-handling.

4.3.1 Button Clicks on a Pattern

A left-click on a pattern slot should cause the following to happen:

1. The pattern will be highlighted (white on a black background). This won't occur until the button is released.
2. During playback, the pattern will emit MIDI events and play its sequence.
3. If the pattern is dragged to another slot, whether playing is in progress or not, releasing the button in the destination slot will move the pattern to that slot.

A right-click on a pattern slot should cause the following to happen:

1. If the pattern is empty, then a pop-up menu to make a New pattern, paste a pattern, or make other selections will appear.
2. If the pattern is active, then a pop-up menu to Edit the pattern or make other selections will appear.
3. A second right-click, just off the menu, will dismiss the menu.

4.3.2 Patterns Window Key Shortcuts

First, note the selection of the File / Options / Keyboard / Show keys option. The tests here should work whether or not it is selected. The only difference is if the keys are shown.

We got a segfault during this test, when we weren't being systematic about it.

4.3.3 The Sequencer64 User File

To be discussed.

4.4 Tests Using Valgrind

Valgrind is a very useful tool for unearthing memory issues and other issues in an application, especially when one has the source code and can build the code with debugging information.

One runs the application from the command line, preceding its command line with valgrind and some of its options.

4.4.1 Valgrind Suppressions

One problem with valgrind is that it also uncovers errors in system libraries that one has no control over. These errors clutter the output, so we suppress them using a valgrind "suppressions" file. Here's how to create one:

```
$ valgrind --gen-suppressions=yes --log-file=val.supp ./Sequencer64/sequencer64
$ valgrind --gen-suppressions=all --log-file=val.supp ./Sequencer64/sequencer64
```

As the program runs, one is asked to print a suppression. If the error is due to a system or third-party library, answer "Y return", and then copy-and-paste the suppression to a file, giving it a name. For example, we provide a file `contrib/seq64.supp` containing suppressions of errors that annoy us. There are way too many "errors" in ALSA, GTK+, gtkmm, glibc, and more.

The second command collects all the suppressions. Passing the val.supp file through sed makes it immediately usable:

```
$ sed -i -e /^==/g val.supp
```

Running valgrind like this then shows mostly the errors we care about:

```
$ valgrind --suppressions=val.supp  ./Sequencer64/sequencer64
```

We've added some other suppression files to the `contrib` directory. Too much! For example:

https://github.com/dtrebbien/GNOME.supp

However, overall this process is very painful, and we're going to eventually do all the valgrind work on the unit-test project for Sequencer64:

https://github.com/ahlstromcj/seq64-tests

4.4.2 Full Valgrind Leak-Checking

Here's how to capture errors, while suppressing the system errors and while generating a log file:

```
$ valgrind --suppressions=contrib/valgrind/seq64.supp \
   --leak-check=full --leak-resolution=high \
   --track-origins=yes --log-file=valgrind.log --show-leak-kinds=all \
   ./Seq64rtmidi/seq64
```

The errors can be also be re-routed to a log-file via the "2> valgrind.log" shell redirection.

Another good idea is to precede the valgrind command with the following construct:

```
$ G_SLICE=debug-blocks valgrind ...
```

G_SLICE=debug-blocks will turn off gtk's advanced memory management to allow valgrind to show correct results. This results in an amazing plethora or invalid read and invalid write errors in GNOME-related libraries. Sheesh!

And don't forget about Valgrind's "massif" memory-tracking tool! (More to come!)

4.4.2.1 Leak-Checking Basic Operation

For the first pass, just run Sequencer64, then immediately exit. Then scan the log file to see if any "errors" can be pinpointed to the application and library code.

Don't forget to run the same scenario without valgrind, in a console window, to see if any of our own debug/problem output occurs.

In any case, leakage tagged as "still reachable" isn't as bad as leakage tagged as "definitely lost" or "indirectly lost".

But good luck finding a Sequencer64 bug buried in the chaff of 3rd-party valgrind reports, even with some suppressions enabled. Apparently a lot of them have to do with data structures that are intended to last the full life of the application.

One can make the search a little easier by searching for the "seq64" namespace in the valgrind log.

4.5 Specific Fault Debugging

This section goes through specific debugging cases we encountered. They should be part of the regular testing of Sequencer64.

4.6 Snipping of a MIDI file.

In order to have a test file for the *seq64-tests* project, we loaded up the `b4uacuse-GM-format.midi` file, removed all but four of the tracks, and saved it as `b4uacuse-snipped.midi`. Loading this file into Sequencer64 caused the following:

```
   $ ./Sequencer64/sequencer64
[Reading user configuration /home/ahlstrom/.config/sequencer64/sequencer64.usr]
[Reading rc configuration /home/ahlstrom/.config/sequencer64/sequencer64.rc]
get_sequence(): m_seqs[4] not null
Segmentation fault
```

First step, fire up a debugger and see what happened. We use *cgdb* , a text-based front-end for gdb with a "vi" feel.

```
   $ cgdb ./Sequencer64/sequencer64
```

Just hit "r", do *File / Open* , navigate to `b4uacuse-snipped.midi`, select it, and watch what happens.

The "bt" (backtrace) command shows a pretty large stack, 52 items. Page up to the top of the stack, and select frame 1 ("fr 1"). This shows a mutex at a very low address, 0x650! Frame 2 shows we are in the automutex constructor, calling lock() on that same badly-located mutex. Frame 3 is in sequence::event_count(), same bad mutex, and the m_events member is at address 0x0. Obviously, we're dealing with an unallocated sequence.

Frame 4 is in mainwid::draw_sequence_on_pixmap(), just after we've retrieved the next sequence via perform↩ ::get_sequence(4). But that would be the fifth sequence (the sequence numbers start at 0), and we snipped all but 4 from the file before we saved it.

So, one thing we need to do is *check* the value returned by get_sequence() before we try to use it. The other thing to do is figure out how we got to the fifth sequence, and fix that code as well. Using the command "p perf().↩ sequence_count()", we verify that there are indeed only 4 sequences allocated.

Frame 5 is in mainwid::draw_sequences_on_pixmap(). That function tries to load all sequences on the current screen-set, from 0 to 31, without checking to see how many their actually are. Inefficient and dangerous.

Frame 6 is in mainwid::reset(). We could pass perf().sequence_count() here for checking, or get it in mainwid↩ ::draw_sequences_on_pixmap().

Before we fix this issue, we need to load a file that works, to see why it does not fail for most files. We will put a breakpoint at the top mainwid::draw_sequences_on_pixmap().

We hit the breakpoint before even loading a file, with a sequence_count() of 0. The call to valid_sequence(0) passes the test. We may want to make valid_sequence() take the sequence_count() into account. But the call to perf().is_active(0) prevents anything bad from happening at startup time.

Once we load a good file, the sequence_count() is 14 in mainwid::draw_sequences_on_pixmap(). We turn on the display of "offset" using the command "display offset", and "c" (for "continue") until offset = 14, which means we are beyond that last sequence. That bad access is prevented by perf().is_active(14).

So the fundamental problem is that perf().is_active(4) is not protecting the access when we load the "bad file". We need to find and fix that issue before papering over the problem with better access checks.

Start again, putting a breakpoint in the call to "new sequence(m_ppqn)" in midifile. This call sets up some members and clears the list of 256 playing notes. Add another breakpoint at "a_perf.add_sequence()" to see what's happening there.

What we find is that the first two tracks have proper sequence numbers as read from the MIDI file, 0 and 1. But the third one preserves the number from the old file, 4. We have a disjunction between the track number and the sequence number, a conceptual problem. We can leave it as is, and beef up the error-checking, or replace the sequence number with the track number when loading the file. What to do?

- Make sure that the is-active flag for all sequences is "false", that the pointers are always null, and make sure to test both of these items (depending on context) before doing anything with the sequence.
- Convert the sequence number to the track number upon saving the MIDI file, or upon reading the MIDI file, and use that number when adding the sequence to the perform object. This might affect some seq24/sequencer64 functionality, however. It's big move.

We need information on reading and importing.

First, if we look at a file that we created long ago by importing `b4uacuse.mid`, `b4uacuse-GM-format.↩ midi`, it has its fourteen sequence numbers identical to their track numbers. No problem.

Second, if we just read `b4uacuse.mid`, a non-seq24-created MIDI file, we see that each of its tracks have no sequence number – they are all zero. The perform::add_sequence() simple iterates from the beginning of m_seqs[] until it finds an inactive m_seqs[i], and uses that element to hold the sequence pointer.

But now it also segfaults! Let's fix all the non-checked get_sequence() calls right away, it is too big an issue to ignore.

In the end, we have to be aware that a screen-set can have blank (null) slots interspersed amongst the active slots.

Chapter 5

Speed Issue of Sequencer64

Author(s) Chris Ahlstrom 2017-04-30

5.1 Introduction

This section describes some speed issues of Sequencer64. A new issue, reported by users, concerning the native JACK MIDI support, needs to be addressed, but has not been addressed yet.

Early on in our reboot of *seq24*, we noticed that some of our larger files took a noticeable time to load. It was only a few seconds, but seemed like a long time for such small files. We fixed the issue using an alternate container implementation.

We recently added a large MIDI file to test, `b4uacuse-stress.midi`, and all hell broke loose.

5.2 Initial Change of Containers

In the beginning, we noticed that the MIDI container implementation used the `std::list` container, and also that it called `std::list::sort()` after each event was added to the container.

Our first thought was to replace the `std::list` with an `std::multimap`. Insertions into this container are made in the appropriate location (rather than at the end), and so are automatically sorted. We kept the old code around, but enabled the new multimap code via the `SEQ64_USE_EVENT_MAP` macro. This decreased the time of loading.

(It also exposed a small number of bugs that users of *Sequencer64* discovered and fixed.)

At the back of our minds was the possibility that the longer time needed to increment a multimap iterator versus a list iterator would prove to limit the amount of data that could be played back. Once we finally created a large file, `b4uacuse-stress.midi`, a 1.5 Mb file, we experienced the limitations of that iterator during playback. On our main development laptop, a near-gaming `Intel i7` machine, there were minor artifacts in playback. On our old single-core laptop with 32-bit `Debian` installed, the sequence would not play, and would continually and visibly refresh the main window display.

5.3 Back to the Original Container

So then we re-enabled the old *seq24* list implementation, and found that the time needed to load `b4uacuse-stress.midi` was over 6 minutes on our near-gamer laptop and 13 minutes on the single-core laptop.

So, we had to find a way to get the fast loading speed of the `std::multimap` and the faster speed of the `std::list` iterator. The obvious way was to go back to the `std::list` container and stop sorting the container after every insert, when loading the MIDI file.

This worked, but had some side-effects that had to be fixed. We found that the `sequence::verify_and_link` function required that the container be sorted first, and so we had to find the places where that function was called, and make sure that the sorting had occurred.

Anyway, the current configuration for the usage of `std::multimap` versus `std::list` and the sorting of the MIDI container after every event insertion versus after all the events are now permanent and hardwired. The default is to not use the event multimap, and to not presort events. This makes loading fast, and playback able to handle more sequences. One can also try to use the multimap, or use the list with pre-sorting, if bugs appear when building the application with the default setting. However, we really want to get the post-sorted list implementation to work, to get fast loading speed and higher throughput at the same time.

The other options are available as a fallback in case one gets struck by bugs in the default, and can afford slower loads or less throughput.

5.4 What is Next, the Vector?

Or, what's your vector, Victor?

For playback, the `std::vector` iterator can be even faster than the `std::list` iterator, because the vector does not use stored pointers to the next element, and, since its storage layout is essentially like a standard C array, processor caching can add even more to the speed of access.

However, this change will require carefully analysis and even more careful work to correctly implement the change. We will log this as a future improvement.

Actually, now we have reverted back to the `std::list` implementation, with the key difference that, when loading a file, we do not sort until we have read all of the events. So we get a fast load time and higher maximum throughput during playback.

Now to test the hell out of the next version!

Chapter 6

OSC Support in Sequencer64

Author(s) Chris Ahlstrom 2017-04-30

6.1 Introduction

This section describes how we might support OSC. One of the first things we need to do is get an idea of how it works. The next thing is a long list of commands to implement, such as various MIDI controls and the loading of specific MIDI files upon command.

We do have some things to think about:

- Should this command set be adapted with the exact same format, for compatibility with Renoise, or should we make our own format that better fits the Sequencer64 model?
- What other commands will we need? Should we support every GUI control and keystroke control that Sequencer64 supports?
- Which OSC library will we use?
  - oscpack
  - liblo
  - lopack
  - loscpack??? (rolling our own)
- Test applications
  - MidiOSC https://github.com/jstutters/MidiOSC

6.2 OSC Format

This section is a condensation of the descriptions at the following URL.

http://opensoundcontrol.org/book/export/html/1

OSC data used these data types. All numbers are big-endian. The atomic size unit is 32-bits. All values are padded, if necessary, to a multiple of 32 bits (i.e. a multiple of 4 bytes).

- `int32`. Two's complement integer.
- `timetag`. 64-bit fixed point.
- `float32`. IEEE 754 float value.
- `string`. ASCII characters, 0 terminated, padded.
- `blob`. An `int32` count of bytes, followed by data bytes, padded.

An application that sends OSC packets is a client. An application that receives OSC packets is a server. An OSC packet is one of the following items, distinguished by the first byte of the packet.

- `OSC Message`.
  - *Address pattern*.
  - *Type tag string*. Older implementations might omit this one. There are four: i = int32, f = float32, s = string, and b = blob, but some applications may add a bunch of non-standard types.
  - *Arguments*. Zero or more values, represented of a contiguous sequence of binary bytes.
- `OSC Bundle`.
  - *"#bundle"*.
  - *Time tag*.
  - *Bundle elements*. Zero or more elements consisting of size and contents.

6.3 Prospective Commands Via Renoise

The following set of commands was copped from the *Renoise* project. We have changed "renoise" to "sequencer64", and have indicated how or if we might implement the command. This information comes from the chapter *The Default OSC Implementation of Renoise*, from the link

http://tutorials.renoise.com/wiki/Open_Sound_Control

6.3.1 /sequencer64/evaluate(string)

Evaluate a custom Lua expression (e.g. renoise.song().transport.bpm = 234). We doubt we will implement this command.

6.3.2 /sequencer64/song/bpm(number)

Set the song's current BPM value [32 - 999]. We will implement this command and make it fit the range allowed by *Sequencer64*.

We will also add to that four commands for BPM increment coarse and fine and BPM decrement coarse and fine.

6.3.3 /sequencer64/song/edit/mode(boolean)

Set the song's global Edit Mode on or off. Does not apply to *Sequencer64*.

6.3.4 /sequencer64/song/edit/octave(number)

Set the song's current octave value [0 - 8]. This will likely be implemented as a "transpose" command, which is already supported by *Sequencer64*. It transposes all of the patterns that are flagged as transposable.

```
/sequencer64/song/edit/transpose(number)
```

6.3.5 /sequencer64/song/edit/pattern_follow(boolean)

Enable or disable the global Pattern Follow mode. Need to do some research into this one.

6.3.6 /sequencer64/song/edit/step(number)

Set the song's current Edit Step value [0 - 8]. We will not implement this command.

6.3.7 /sequencer64/song/instrument/XXX/macro1-8(number)

Set instrument XXX's macro parameter value [0 - 1]. XXX is the instrument index, -1 the currently selected one. We will not implement this command.

However, if we do implement commands that follow the pattern of selecting the number of a pattern, the name of a pattern, or all patters, it would probably look like the format

```
/sequencer64/song/pattern/command_function(pattern_tag, parameter_value)
```

where *pattern_tag* is a name, a number, or "all", and *parameter_value* is the value to apply to the command.

6.3.8 /sequencer64/song/instrument/XXX/monophonic(boolean)

Set instrument XXX's trigger mono mode [True/False]. XXX is the instrument index, -1 the currently selected one. We will not implement this command.

6.3.9 /sequencer64/song/instrument/XXX/monophonic_glide(number)

Set instrument XXX's trigger mono glide amount [0-255]. XXX is the instrument index, -1 the currently selected one. We will not implement this command.

6.3.10 /sequencer64/song/instrument/XXX/phrase_playback(string)

Set instrument XXX's phrase playback mode [Off, Program, Keymap]. XXX is the instrument index, -1 the currently selected one. We will not implement this command.

6.3.11 /sequencer64/song/instrument/XXX/phrase_program(number)

Set instrument XXX's phrase program [0 - 127]. XXX is the instrument index, -1 the currently selected one. We will not implement this command.

6.3.12 /sequencer64/song/instrument/XXX/quantize(string)

Set instrument XXX's trigger quantization [Note, Line, Beat, Bar]. XXX is the instrument index, -1 the currently selected one.

We will probably implement a similar command to quantize a numbered pattern, a named pattern, or all patterns.

6.3.13 /sequencer64/song/instrument/XXX/scale_key(string)

Set instrument XXX's note scaling key [C, C#... B]. XXX is the instrument index, -1 the currently selected one. We will probably implement a similar command to set the key of a numbered pattern, a named pattern, or all patterns.

6.3.14 /sequencer64/song/instrument/XXX/scale_mode(string)

Set instrument XXX's note scaling mode. XXX is the instrument index, -1 the currently selected one. We will probably implement a similar command to set the scale of a numbered pattern, a named pattern, or all patterns.

6.3.15 /sequencer64/song/instrument/XXX/transpose(number)

Set instrument XXX's global transpose [-120 - 120]. XXX is the instrument index, -1 the currently selected one. We won't implement this one, it is covered above.

6.3.16 /sequencer64/song/instrument/XXX/volume(number)

Set instrument XXX's global volume [0 - db2lin(6)]. XXX is the instrument index, -1 the currently selected one. We will probably implement a similar command to set the scale of a numbered pattern, a named pattern, or all patterns.

6.3.17 /sequencer64/song/instrument/XXX/volume_db(number)

Set instrument XXX's global volume in dB [0 - 6]. XXX is the instrument index, -1 the currently selected one. We will probably implement a similar command to set the scale of a numbered pattern, a named pattern, or all patterns.

6.3.18 /sequencer64/song/lpb(number)

Set the song's current Lines Per Beat value [1 - 255]. We will probably implement a similar command to set the visible grid-lines per beat setting. Probably for a numbered pattern, a named pattern, or all patterns. Most likely just "all".

6.3.19 /sequencer64/song/record/metronome

Enable or disable the metronome. This one is something to think about. Not yet a feature of Sequencer64.

6.3.20 /sequencer64/song/record/metronome_precount

Enable or disable the global metronome precount. This one is something to think about. Not yet a feature of Sequencer64.

6.3.21 /sequencer64/song/record/quantization(boolean)

Enable or disable the global Record Quantization. We will probably implement this one.

6.3.22 /sequencer64/song/record/quantization_step(number)

Set the global Record Quantization step value [1 - 32]. We will probably implement this one.

6.3.23 /sequencer64/song/sequence/schedule_add(number)

Add a scheduled sequence playback position. Not sure what this one means, need to research it.

6.3.24 /sequencer64/song/sequence/schedule_set(number)

Replace the currently scheduled sequence playback position. Not sure what this one means, need to research it.

6.3.25 /sequencer64/song/sequence/slot_mute(number, number)

Mute the specified track at the specified sequence slot in the matrix. We will implment this, and provide a number, name, or "all" variant. We will also provide a variant the uses the defined mute/unmute keys.

6.3.26 /sequencer64/song/sequence/slot_unmute(number, number)

Unmute the specified track at the specified sequence slot in the matrix. We will implment this, and provide a number, name, or "all" variant. We will also provide a variant the uses the defined mute/unmute keys.

6.3.27 /sequencer64/song/sequence/trigger(number)

Set the playback position to the specified sequence position. We will implement this one, and it will probably apply to the Song Editor.

6.3.28 /sequencer64/song/tpl(number)

Set the song's current Ticks Per Line value [1 - 16].

If adding a track action, then Renoise helps. For track actions Renoise assumes the OSC address pattern will begin with /renoise/track/XXX, where XXX is the track index. Action-handling code then needs to specify the remaining part of the address pattern.

More research to do.

6.3.29 /sequencer64/song/track/XXX/device/XXX/bypass(boolean)

Set the bypass status of a device [true or false]. (XXX is the device index, -1 chooses the currently selected device) We will not implement this one.

6.3.30 /sequencer64/song/track/XXX/device/XXX/set_parameter_by_index(number, number)

Set the parameter value of a device [0 - 1]. (XXX is the device index, -1 chooses the currently selected device) This one will require a lot of thought and some more details, like how to specify the desired parameter to set.

6.3.31 /sequencer64/song/track/XXX/device/XXX/set_parameter_by_name(string, number)

Set the parameter value of a device [0 - 1]. (XXX is the device index, -1 chooses the currently selected device) Similar to the previous setting.

6.3.32 /sequencer64/song/track/XXX/mute

Mute track XXX. (XXX is the device index, -1 chooses the currently selected device) Similar to the mute command above. Redundant?

6.3.33 /sequencer64/song/track/XXX/output_delay(number)

Set track XXX's delay in ms [-100 - 100]. (XXX is the device index, -1 chooses the currently selected device) This is not yet a feature of Sequencer64.

6.3.34 /sequencer64/song/track/XXX/postfx_panning(number)

Set track XXX's post FX panning [-50 - 50]. (XXX is the device index, -1 chooses the currently selected device) We could add this as an OSC-only parameter that can be applied to patterns by number, name, or "all".

6.3.35 /sequencer64/song/track/XXX/postfx_volume(number)

Set track XXX's post FX volume [0 - db2lin(3)]. (XXX is the device index, -1 chooses the currently selected device) We will not implement this one.

6.3.36 /sequencer64/song/track/XXX/postfx_volume_db(number)

Set track XXX's post FX volume in dB [-200 - 3]. (XXX is the device index, -1 chooses the currently selected device) We will not implement this one.

6.3.37 /sequencer64/song/track/XXX/prefx_panning(number)

Set track XXX's pre FX panning [-50 - 50]. (XXX is the device index, -1 chooses the currently selected device) We will not implement this one.

6.3.38 /sequencer64/song/track/XXX/prefx_volume(number)

Set track XXX's pre FX volume [0 - db2lin(3)]. (XXX is the device index, -1 chooses the currently selected device) We will not implement this one.

6.3.39 /sequencer64/song/track/XXX/prefx_volume_db(number)

Set track XXX's pre FX volume in dB [-200 - 3]. (XXX is the device index, -1 chooses the currently selected device) We will not implement this one.

6.3.40 /sequencer64/song/track/XXX/prefx_width(number)

Set track XXX's pre FX width [0, 1]. (XXX is the device index, -1 chooses the currently selected device) We will not implement this one.

6.3.41 /sequencer64/song/track/XXX/solo

Solo track XXX. (XXX is the device index, -1 chooses the currently selected device) This will be a good one to implement.

6.3.42 /sequencer64/song/track/XXX/unmute

Unmute track XXX. (XXX is the device index, -1 chooses the currently selected device) This will be a good one to implement. Similar to the version defined above.

6.3.43 /sequencer64/transport/continue

Continue playback. This might be better as an implementation of the existing pause functionality.

6.3.44 /sequencer64/transport/loop/block(boolean)

Enable or disable pattern Block Loop. Will need to see if this a useful feature for Sequencer64.

6.3.45 /sequencer64/transport/loop/block_move_backwards

Move the Block Loop one segment backwards Not yet sure what this might mean.

6.3.46 /sequencer64/transport/loop/block_move_forwards

Move the Block Loop one segment forwards Not yet sure what this might mean.

6.3.47 /sequencer64/transport/loop/block(boolean)

Enable or disable looping the current pattern. Not sure how this would differ from muting/unmuting a pattern.

6.3.48 /sequencer64/transport/loop/sequence(number, number)

Disable or set a new sequence loop range. Not sure what this would mean in Sequencer64.

6.3.49 /sequencer64/transport/panic

Stop playback and silence all playing instruments and effects. This needs to be a new feature for Sequencer64, as well as an OSC command.

6.3.50 /sequencer64/transport/start

Start playback or restart playing the current pattern. We will implement this one.

6.3.51 /sequencer64/transport/stop

Stop playback. We will implement this one.

6.3.52 /sequencer64/trigger/midi(number)

Trigger a raw MIDI event. arg#1: the MIDI event as number

Sounds like a useful function.

6.3.53 /sequencer64/trigger/note_off(number, number, number)

Trigger a Note OFF.

arg#1: instrument (-1 chooses the currently selected one) arg#2: track (-1 for the current one) arg#3: note value (0-119)

Sounds like a useful function.

6.3.54 /sequencer64/trigger/note_on(number, number, number, number)

Trigger a Note-On.

arg#1: instrument (-1 for the currently selected one) arg#2: track (-1 for the current one) arg#3: note value (0-119) arg#4: velocity (0-127)

Sounds like a useful function.

Note that the default OSC implementation [in Renoise] can be extended by editing the file "GlobalOscActions.lua" in the "Scripts" folder, found within the directory where you installed Renoise (on OSX this is found in the app bundle).

6.4 MidiOSC Messages

The *MidiOSC* application converts MIDI input to OSC messages with the following address.

```
/midi/devicename/channel
```

where "devicename" and "channel" present the parameters of the MIDI message. It can convert the following OSC messages to MIDI.

```
Message type        MIDI byte   Arguments
note_off            0x80        2
note_on             0x90        2
key_pressure        0xa0        2
controller_change   0xb0        2
program_change      0xc0        2
channel_pressure    0xd0        2
pitch_bend          0xe0        2
song_position       0xf2        2
song_select         0xf3        2
tune_request        0xf6        2
timing_tick         0xf8        0
start_song          0xfa        0
continue_song       0xfb        0
stop_song           0xfc        0
Sysex messages (input only)
```

```
Sysex Message type  MIDI byte
mmc_stop            0x01
mmc_play            0x02
mmc_fast_forward    0x04
mmc_rewind          0x05
mmc_record          0x06
mmc_pause           0x09
```

Chapter 7

Licenses

Library This application and its libraries, sub-applications, and documents.

Author(s) Chris Ahlstrom 2015-09-10

7.1 License Terms for the This Project.

Wherever the tag $XPC_SUITE_GPL_LICENSE$ appears, or wherever reference to the GPL licensing scheme (any version) is mentioned, substitute the appropriate license text, depending on whether the project is a library, application, documentation, or server software. We're not going to include paragraphs of licensing information in every module; you are responsible for coming here to read the licensing information.

These licenses apply to each sub-project and file artifact in the project with which this license description was packaged.

Wherever the term **XPC** is encountered in this project, it refers to my projects, which go beyond the package that contains this document.

7.2 XPC Application License

The **XPC** application license is either the **GNU GPLv2**. or the **GNU GPLv3**. Generally, projects that originate with me use the latter language, while projects I have extended may specify the former license.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the

```
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301, USA.
```

The text of the GNU GPL version 3 license can also be found here:

http://www.gnu.org/licenses/gpl-3.0.txt

7.3 XPC Library License

The **XPC** library license is the **GNU LGPLv3**.

7.4 XPC Documentation License

The **XPC** documentation license is the **GNU FDLv1.3**.

7.5 XPC Affero License

The **XPC** "Affero" license is the **GNU AGPLv3**.

At the present time, no **XPC** project uses the "Affero" license.

7.6 XPC License Summary

Include one of these licenses in your Doxygen documentation with one of the following Doxygen tags specified above:

```
\ref gpl_license_subproject
\ref gpl_license_application
\ref gpl_license_library
\ref gpl_license_documentation
\ref gpl_license_affero
```

For more information on navigating GNU licensing, see this page:

```
http://www.gnu.org/licenses/
```

Copies of these licenses (and some logos) are provided in the `licenses` directory of the main project (or you can search for them at *gnu.org*).