Contents

Chapter 1

Internals – how it works

This chapter explains how MusE is built internally, and is meant to be an aid for developers wanting to quickly start up with MusE. For details on *why* stuff is done please refer to the following chapter.

1.1 User interface programming

We use the QT Toolkit for GUI- and other programming. The *QT-Assistant* is an important tool for getting help. Almost everything can be looked up there.

GUIs can be either be hardcoded (see `arranger.cpp` for an example) or can be created using *QT-Designer* (see the dialogs under `widgets/function_dialogs/` for mostly cleanly-written examples). Don't forget to add your `cpp`, `h` and `ui` files to the corresponding sections in the `CMakeLists.txt`!

Additionally, MusE offers some custom widgets, like menu title items etc. Following, there will be a small, unordered list about custom widgets:

- `MusEGui::MenuTitleItem`: Provides a title-bar in a `QMenu`.
  Usage: `someMenu->addAction(new MusEGui::MenuTitleItem(tr("fnord"), someMenu));`
  Defined in `widgets/menutitleitem.h`.

- `MusEGui::PopupMenu`: Provides a `QMenu`-like menu which can stay open after the user checks a checkable action.
  Usage: just create a `new PopupMenu( true|false )` instead of a `new QMenu()`. (`true` means 'stay open')
  Defined in `widgets/popupmenu.h`.

1.2 Configuration

Configuration is a bit pesky in MusE in its current state. If you get confused by reading this chapter, that's a sign of a sane mind.

There are three kinds of configuration items:

- (1) Global configuration, like coloring schemes, plugin categories, MDI-ness settings
- (2) Per-Song configuration, like whether to show or hide certain track types in the arranger
- (3) Something in between, like MIDI port settings etc. They obviously actually are global configuration issues (or ought to be), but also obviously must be stored in the song file for portability. (This problem could possibly be solved by the feature proposal in 3.7.

Reading configuration `void MusECore::readConfiguration(Xml&, bool, bool)` in `conf.cpp` is the central point of reading configuration. It is called when MusE is started first (by `bool MusECore::readConfiguration()`), and also when a song is loaded.
It can be instructed whether to read MIDI ports (3), global configuration and MIDI ports (1+3). Per-Song configuration is always read (2).

When adding new configuration items and thus altering `readConfiguration()`, you must take care to place your item into the correct section. The code is divided into the following sections:

- Global and/or per-song configuration (3)
- Global configuration (1)
- Code for skipping obsolete entries

The sections are divided by comments (they contain `--`, so just search for them). Please do not just remove code for reading obsolete entries, but always add an appropriate entry to the 'skipping' section in order to prevent error messages when reading old configs.

Writing configuration Global configuration is written using the `MusEGui::MusE::writeGlobalConfiguration()` functions, while per-song-config is written by `MusEGui::MusE::writeConfiguration()` (notice the missing `Global`; both implemented in `conf.cpp`).

`writeConfiguration` is actually just a subset of the code in `writeGlobalConfiguration`. **Duplicate code!**

Song state Additionally to per-song configuration, there is the song's state. This contains "the song", that is all tracks, parts and note events, together with information about the currently opened windows, their position, size, settings and so on. Adding new items here is actually pretty painless: Configuration is read and written using `MusECore::Song::read` and `::write`, both implemented in `songfile.cpp`. There are no caveats.

How to add new items When adding global configuration items, then add them into the second block ("global configuration") in `readConfiguration` and into `writeGlobalConfiguration`.

When adding just-per-song items, better don't bother to touch the "configuration" code and just add it to the song's state (there might be rare exceptions).

When adding global configuration items, make sure you add them into the correct section of `readConfiguration`, and into `writeGlobalConfiguration`.

1.3 User controls and automation

1.3.1 Handling user input

Plugins and synthesizers

Overview When the user launches a plugin's GUI, either a MusE-window with the relevant controls is shown, or the native GUI is launched. MusE will communicate with this native GUI through OSC (Open Sound Control). The relevant classes are `PluginGui`, `PluginIBase` (in `plugin.h`) and `OscIF` (in `osc.h`).

If the user changes a GUI element, first the corresponding control is disabled, making MusE not steadily update it through automation while the user operates it. Then MusE will update the plugin's parameter value, and also record the new value. When appropriate, the controller is enabled again.

Processing the input, recording Upon operating a slider, `PluginIBase::setParam` is called, which usually writes the control change into the ringbuffer `PluginI::_controlFifo`. (`PluginI::apply()`, `DssiSynthIF::getData()` will read this ringbuffer and do the processing accordingly). Furthermore, `AudioTrack::recordAutomation` is called, which either directly modifies the controller lists or writes the change into a "to be recorded"-list (`AudioTrack::_recEvents`) (depending on whether the song is stopped or played).

The `AudioTrack::_recEvents` list consists of `CtrlRecVal` items (see `ctrl.h`), which hold the following data:

- the frame where the change occurred
- the value
- the type, which can be `ARVT_START`, `ARVT_VAL` or `ARVT_STOP`. `ARVT_VAL` are written by every `AudioTrack::recordAutomation` call, `ARVT_START` and `ARVT_STOP` are generated by `AudioTrack::startAutoRecord` and `stopAutoRecord`, respectively.
- and the id of the controller which is affected

It is processed when the song is stopped. The call path for this is: `Song::stopRolling` calls `Song::processAutomationEvents` calls `AudioTrack::processAutomationEvents`. This function removes the old events from the track's controller list and replaces them with the new events from `_recEvents`. In `AUTO_WRITE` mode, just all controller events within the recorded range are erased; in `AUTO_TOUCH` mode, the `ARVT_START` and `ARVT_STOP` types of the `CtrlRecVal` events are used to determine the range(s) which should be wiped.

How it's stored Automation data is kept in `AudioTrack::_controller`, which is a `CtrlListList`, that is, a list of `CtrlList`s, that is, a list of lists of controller-objects which hold the control points of the automation graph. The `CtrlList` also stores whether the list is meant discrete (a new control point results in a value-jump) or continuous (a new control point results in the value slowly sloping to the new value). Furthermore, it stores a `_curVal` (accessed by `curVal()`), which holds the currently active value, which can be different from the actually stored value because of user interaction. This value is also used when there is no stored automation data.

`AudioTrack::addController` and `removeController` are used to add/remove whole controller types; the most important functions which access `_controller` are:

- `processAutomationEvents`, `recordAutomation`, `startAutoRecord`, `stopAutoRecord`: see above.
- `seekPrevACEvent`, `seekNextACEvent`, `eraseACEvent`, `eraseRangeACEvents`, `addACEvent`, `changeACEvent`, which do the obvious
- `pluginCtrlVal`, `setPluginCtrlVal`: the first returns the current value according to the `_controller` list, the second only sets the `curVal`, but does not insert any events.

Whenever a `CtrlList` has been manipulated, `MusEGlobal::song->controllerChange(Track*)` shall be called, which emits the `MusEGlobal::song->controllerChanged(Track*)` signal in order to inform any parts of MusE about the change (currently, only the arranger's part canvas utilizes this).

Enabling and disabling controllers Disabling the controller is both dependent from the current automation mode and from whether the GUI is native or not. In `AUTO_WRITE` mode, once a slider is touched (for MusE-GUIs) or once a OSC control change is received (for native GUIs), the control is disabled until the song is stopped or seeked.

In `AUTO_TOUCH` (and currently (r1492) `AUTO_READ`, but that's to be fixed) mode, once a MusE-GUI's slider is pressed down, the corresponding control is disabled. Once the slider is released, the control is re-enabled again. Checkboxes remain in "disabled" mode, however they only affect the recorded automation until the last toggle of the checkbox. (Example: start the song, toggle the checkbox, toggle it again, wait 10 seconds, stop the song. This will NOT overwrite the last 10 seconds of automation data, but everything between the first and the last toggle.). For native GUIs, this is a bit tricky, because we don't have direct access to the GUI widgets. That is, we have no way to find out whether the user doesn't touch a control at all, or whether he has it held down, but just doesn't operate it. The current behaviour for native GUIs is to behave like in `AUTO_WRITE` mode.

The responsible functions are: `PluginI::oscControl` and `DssiSynthIF::oscControl` for handling native GUIs, `PluginI::ctrlPressed` and `ctrlReleased` for MusE default GUIs and `PluginI::guiParamPressed`, `guiParamReleased`, `guiSliderPressed` and `guiSliderReleased` for MusE GUIs read from a UI file; `guiSlider*` obviously handle sliders, while `guiParam*` handle everything else which is not a slider. They call `PluginI::enableController` to enable/disable it.

Furthermore, on every song stop or seek, `PluginI::enableAllControllers` is called, which re-enables all controllers again. The call paths for this are:

- For stop: `Song::stopRolling` calls `Song::processAutomationEvents` calls `Song::clearRecAutomation` calls `Track::clearRecAutomation` calls `PluginI::enableAllControllers`
- For seek: `Audio::seek` sends a message (`"G"`) to `Song::seqSignal` which calls `Song::clearRecAutomation` which calls `PluginI::enableAllControllers`

Chapter 2

Design decisions

2.1 Automation

As of revision 1490, automation is handled in two ways: User-generated (live) automation data (generated by the user moving sliders while playing) is fed into `PluginI::_controlFifo`. Automation data is kept in `AudioTrack::_controller`, which is a `CtrlListList`, that is, a list of `CtrlLists`, that is, a list of lists of controller-objects which hold the control points of the automation graph. The `CtrlList` also stores whether the list is meant discrete (a new control point results in a value-jump) or continous (a new control point results in the value slowly sloping to the new value).

While `PluginI::_controlFifo` can be queried very quickly and thus is processed with a very high resolution (only limited by the minimum control period setting), the automation value are expensive to query, and are only processed once in an audio *driver* period. This might lead to noticeable jumps in value.

This could possibly be solved in two ways:

Maintaining a slave control list This approach would maintain a fully redundant slave control list, similar to `PluginI::_controlFifo`. This list must be updated every time any automation-related thing is changed, and shall contain every controller change as a tuple of controller number and value. This could be processed in the same loop as `PluginI::_controlFifo`, making it comfortable to implement; furthermore, it allows to cleanly offer automation-settings at other places in future (such as storing automation data in parts or similar).

Holding iterators We also could hold a list of iterators of the single `CtrlLists`. This would also cause low CPU usage, because usually, the iterators only need to be incremented once. However, it is pretty complex to implement, because the iterators may become totally wrong (because of a seek in the song), and we must iterate through a whole list of iterators.

Just use the current data access functions By just using the current functions for accessing automation data, we might get a quick-and-dirty solution, which however wastes way too much CPU ressources. This is because on *every single frame*, we need to do a binary search on multiple controller lists.

Chapter 3

Feature requests

3.1 Per-Part automation and more on automation

Automation shall be undo-able. Automation shall reside in parts which are exchangeable, clonable etc (like the MIDI- and Wave-Parts). Global per-synth/per-audiotrack automation shall also be available, but this can also be implemented as special case of part automation (one long part).

3.2 Pre-Rendering tracks

3.2.1 The feature

All tracks shall be able to be "pre-renderable". Pre-rendering shall be "layered". Pre-rendering shall act like a transparent audio cache: Audio data is (redundantly) stored, wasting memory in order to save CPU.

That is: Each track owns one or more wave-recordings of the length of the song. If the user calls "pre-render" on a track, then this track is played quasi-solo (see below), and the raw audio data is recorded and stored in the "layer 0" wave recording. If the user has any effects set up to be applied, then each effect is applied on a different layer (creating layer 1, layer 2 etc).

This means, that also MIDI and drum tracks can have effects (which usually only operate on audio, but we HAVE audio data because of this prerendering).

Furthermore, MusE by default does not send MIDI events to the synthesizers but instead just plays back the last layer of the prerecording (for MIDI tracks), or does not pipe the audio data through the whole plugin chain (causing cpu usage), but instead just plays back the last layer. The hearable result shall be the same.

Once the user changes any parameter (automation data or plugins for wave tracks, MIDI events or effect plugin stuff for MIDI tracks), then MusE shall generate the sound for this particular track in the "old" way (send MIDI data to synthes, or pipe audio data through plugins). (So that the user will not even notice that MusE actually pre-renderered stuff.) Either MusE automatically records this while playback (if possible) or prompts the user to accordingly set up his cabling and then record it. Or (temporarily) disables prerecording for this track, falling back to the plain old way of generating sound.

Quasi-solo means: For wave tracks, just solo the track. For MIDI tracks, mute all tracks which are not on the same synth (channel?), and mute all *note* events which are not on the quasi-soloed track. This causes MusE to still play any controller events from different tracks, because they might have effects on the quasi-soloed track. (You can have notes on channel 1 on one track and controller stuff on channel 1 on another track; then you would need quasi-solo to get proper results.)

3.2.2 Use cases

Saving CPU On slow systems, this is neccessary for songs with lots of, or demanding (or both) soft synthes / plugins. Even if the synth or plugin is so demanding that your system is not able to produce sound in real-time, then with this feature you'll be able to use the synth (this will make editing pretty laggish, because for a change you need to re-render at least a part before you can listen to it, but better than being unable to use the synth at all!)

Exporting as audio project Using pre-rendering on all tracks, you easily can export your project as multi-track audio file (for use with Ardour or similar DAWs). Just take the last layer of each track, and write the raw audio data into the file, and you're done. (Maybe we are even able to write down the raw-raw layer0 audio data plus information about used plugins and settings etc..?)

Mobile audio workstations You might want to work a bit on your audio projects on your notebook while you're not at home, not having access to your hardware synthesizers. Using this feature, you could have pre-recorded the stuff in your studio before, and now can at least fiddle around with the non-hw-synth-dependent parts of your song, while still having your *full* song with you.

Applying effects on MIDI tracks If you have many physical audio inputs, you might already be able to apply effect chains on MIDI tracks, by wiring the synthes' audio outputs to your soundcard's inputs, and applying the effects on dedicated input tracks you have to create. This requires you to have expensive hardware, and is pretty complicated, because you need one additional track per MIDI synth.

This feature allows you to apply effects on single MIDI tracks, and not only on full MIDI synthes, and doesn't require you to be have that many physical audio inputs (you need to manually replug your synthes, however).

3.2.3 Possible scenarios

Setting it up Create a wave track, MusE will allow you to set or unset prerendering for every plugin in the plugin rack (recording the actual track is useless because it would be a plain copy). Create a MIDI track, MusE will ask you on which physical audio input your synth is connected. Setting up multiple synthes on one physical audio in is allowed, see below.

Pre-rendering stuff When the user presses the "pre-render" button, all tracks which have been changed since their last pre-rendering will be re-rendered. If you have multiple hardware synthes set up as they were connected to one physical audio input port, MusE will prompt you to first plug the proper cable in.

Making changes Change a note in a MIDI part, move or delete a part or change automation parameters. MusE will temporarily disable the pre-rendered information and instead generate the sound via sending out MIDI events, piping stuff through effect chains or similar. If you play back the whole song, or if you manually trigger a re-rendering of a track via the context menu, MusE will play back the stuff, record it again and re-enable the pre-rendered information.

3.2.4 Extensions

Automatic discovery of physical audio connections The user plugs all (or only some) synthes' audio outs into the available audio inputs, then runs automatic discovery. This will send MIDI events to each synthesizer, and look on which audio in there's activity. Then it will assume that the synthesizer is connected to that particular audio in. Audio ins which show activity before any MIDI events were sent are not considered, as they're probably connected to microphones or other noise-generating non-synthes.

Audio export As described in the Use cases, MusE can allow you to export your song in some multi-track audio format.

Cheap/Faked changes For expensive or unavailable synthes, changing the Volume midi controller, the Pan controller or similar "easy" controllers will not trigger a complete re-rendering, but instead "fake" the change, by changing the volume data directly on the recorded wave. This might require some learning and might even get pretty complicated.

Intelligent re-recording For tiny changes, MusE shall only re-render the relevant part. If you change some MIDI notes, then begin re-recording shortly before the changes, and end re-recording as soon as the recorded stuff doesn't differ to much from the stuff coming from the synth. Then properly blend the old recording with the updated part.

3.3 Slotted editors

Currently, MusE has the pianoroll editor, drum editor, score editor, then the controller editor which is inside the pianoroll/drum editor. All these editors have a very similar concept: the "time axis" is vertical and (almost) linear, they handle parts, and events are manipulated similarly.

A unified editor shall be created which allows you to combine different kinds of editors in one window, properly aligned against each other. These "different kinds of editors" shall be handled as "slots"; one unified editor window consists of:

- A menu bar, containing stuff suitable for the complete window, which might include window name, MDI-ness etc.
- A toolbar which contains controls suitable for every single slot.
- A container with one or more slots; the slots can be scrolled in y-direction if there are multipe slots.
- A time-scrollbar with zoom

Each slot contains the following:

- A menu button, button box or control panel for setting up this particular slot. This could contain "note head colors", "show a transposing instrument" etc for score edit slots, "event rectangle color", "grid size" and "snap to grid" for pianoroll/ drum editors.
- The actual canvas
- A y-direction scroll bar, possibly with zoom control (for pianoroll editor)

The main window does not show its scroll bar if there is only one slot, because the slot's scrollbar is sufficient then.

Slots can be added, destroyed, moved around, maybe even merged (if the slot types allow it); basically, you can compare them with the staves in the score editor.

The slots shall align against each other, that is, if a score editor slot displays a key change with lots of accidentials, then all other slots shall either also display the key change (if they're score slots) or display a gap. Events which happen at the same time shall be at the same x-coordinate, regardless which slot they are.

3.4 Controller master values

All controllers (MIDI-controllers and also automation controllers) shall have one set of "master values" which allow you to set a gain and a bias. Instead of the actual set value, value $*$ bias + $textrmbias$ shall be sent to the MIDI device / the plugin. For controllers like "pan", the unbiased values shall be transformed, that is, a pan of 64, with bias = 2 and gain = 0.5, shall be transformed to 66 (because 64 is actually 0, while 0 is actually -64). These values shall be set in the arranger and whereever the actual controller/automation values can be edited.

3.5 Enabled-indicator while recording

The MusE-plugin-GUIs shall display a small LED displaying whether a controller is currently enabled or disabled. By clicking this LED, the enabled state shall be switched.

Furthermore, there shall be a dedicated window which only lets you switch enabled/disabled states. This will be useful when using external GUIs or the MIDI-controller-to-automation feature, to re-enable a controller when in `AUTO_TOUCH` mode.

3.6 Linear automation editing

While holding some modifier key (like shift), operating the MusE-native- GUI sliders shall only generate control points when clicking and when releasing the slider. This will result in linear graphs for continous controllers, and in large steps for discrete controllers (which is in particular useful for stuff like "which low/high-pass filter type to use").

Maybe make this behaviour default for discrete controllers?

3.7 Symbolic names for MIDI ports

MIDI ports shall have a user-defined symbolic name (like "Korg" or "Yamaha DX 7"). The mapping between these symbolic names and the hardware port (like "ALSA midi out port") is stored in the global configuration.

Song files only specify the symbolic names as the ports associated with their tracks. No information about physical devices/port names, but only symbolic names are stored in the song file.

This resolves the issues mentioned in 1.2, and also allows the user to share his pieces with other people: They would only have to set up that symbolic-to-hardware mapping once (collisions are unlikely, because an equal symbolic name should usually mean the same device) and are happy, instead of having to re-map *every* port for *every* song.