Advanced Gtk+ Sequencer

Linux Audio Conference 2018

Copyright (C) Joël Krähemann.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

COLLABORATORS			
	TITLE : Advanced Gtk+ Sequencer		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Joël Krähemann	January 12, 2018	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Advanced Gtk+ Sequencer origin

As a blood young hacker being naive and thinking doing all his goals within one 1 or maybe 2 years, the idea of free sequencer was developed. Joël recognized later, that his idea of a nested AgsRecycling tree, which provides some kind of "wormhole" to the ordinary setup with AgsAudio and AgsChannel, was important. But the entire code base needed to be implemented and it is going to take much more time.

12 years later first major release was published. After hacking 4 years fulltime on GSequencer and as of version 1.2.7, the recycling tree should be working without bigger issues. The massive use of threads nowadays, just could be achieved because he was convinced of the "wormhole". This allows you to have 3 different multi-threaded scopes (playback, step-sequencer and notation).

GSequencer provides its functionality by these libraries:

- libags.so.1, libags-thread.so.1, libags-server.so.1 - basic abstraction
- libags-audio.so.1 - audio tree and processing facilities
- libags-gui.so.1 - reusable Gtk+-2.0 widgets
- libgsequencer.so.0 - the GSequencer blue-print

Chapter 1

The AgsThread object

libags-thread.so.1 contains the AgsThread object. Its class provides the ::run() method and shall synchronize all tics within the thread tree by the ::clock() event. E.g. you synchronize 1000 times per second all your threads ::clock() is invoked for each tic. Whereas ::run() is only called as many times as ::clock() tells you to run. This might be 0 or 1 times. Further the count of tics per second an audio thread is actually run, is calculated as following:

Example 1.1 Thread application context

```
{
  guint tics_per_second;

  tics_per_second = (guint) ceil((AGS_SOUNDCARD_DEFAULT_BUFFER_SIZE /
          AGS_SOUNDCARD_DEFAULT_SAMPLERATE) *
         AGS_THREAD_MAX_PRECISION);
}
```

You might think of a unneeded over-head but remaining tics can do actually something useful. Like reading from a MIDI source, done by an intermediate pre-sync or write to an audio sink, done by intermediate post-sync. This setup uses 2 additional tics. The AgsThread:max-precision property can be specified in the ~/.gsequencer/ags.conf file.

1.1 AgsAudioLoop implementing AgsMainLoop

Within the audio layer AgsAudioLoop is the topmost thread and implements the AgsMainLoop interface. It is responsible to wait until all AgsAudioThread objects did their job and program flow is allowed to continue.

AgsTaskThread is usually provided along the audio loop. libags-audio1 provides various tasks to run in a thread-safe state of the engine. Tasks are running between the tics, so there is thread-safe access guaranteed. However the UI may access audio related objects and shall be protected by mutexes anyway.

Since Gtk+-2.0 doesn't allow access to the UI from different threads, there was a message delivery system developed. See AgsMessageDelivery and AgsMessageQueue.

1.2 AgsPlaybackDomain and AgsPlayback

There are 3 multi-threaded scopes and threads per instrument channel. The scopes defined by the enumerations AgsPlaybackDomainScope and AgsPlaybackScope match each. In fact `playback_domain->audio_thread[AGS_PLAYBACK_DOMAIN_SCOP` synchronizes with `playback->channel_thread[AGS_PLAYBACK_SCOPE_SEQUENCER]` by doing conditional-locks.

AgsPlaybackDomain is provided as a property of AgsAudio. The audio object is a container of AgsOutput and AgsInput, which inherit of AgsChannel. Further it specifies how the output and input is aligned. Either AGS_AUDIO_SYNC this means mapped 1:1 or AGS_AUDIO_ASYNC this means mapped 1:n.

AgsPlayback is provided as a property of AgsChannel. The channel object contains a references to AgsRecycling objects. By specifying its boundaries AgsChannel:first-recycling and AgsChannel:last-recycling, all containing AgsAudioSignal within this range shall be processed.

The AgsAudioSignal contains your finite audio stream, it can be sampled from templates either as fixed length (pattern mode) or by :loop-start and :loop-end property as length with frame count (notation mode). It is usually assigned to a AgsRecallID. This makes it possible that every thread can have its own set of audio data and process audio lock free.

Chapter 2

Hierarchical computation of sound

Processing audio data is done by sub-types of AgsRecall. There are for each tree context a predefined recall you should inherit from.

- AgsRecallAudio and AgsRecallAudioRun
- AgsRecallChannel and AgsRecallChannelRun
- AgsRecallRecycling
- AgsRecallAudioSignal

AgsRecallAudio may provide ports being modified or automated by the UI. AgsRecallAudioRun is provided as a templated and has to be instantiated per thread.

AgsRecallChannel and AgsRecallChannelRun is analogous to above but instead applying to AgsAudio it is related to AgsChannel.

AgsRecallRecycling is a child of AgsRecallChannelRun. It basically tells what AgsRecallAudioSignal instance needs to be created.

AgsRecallAudioSignal does lock free computation of your effect. This could be envelope, compressor, echo delay ...

Note there are recalls just providing parts of this construct. Like "ags-play-notation" recall instantiated by `ags_recall_factory_cr` it only provides AgsPlayNotationAudio and AgsPlayNotationAudioRun.

2.1 AgsAudio

The audio object contains many meta informations as well the segmented data AgsNotation, AgsAutomation and AgsWave. It contains lists of AgsRecallID and AgsRecyclingContext, note they both are related. Every audio and channel object has got its very own recall id but the recycling context assigned to it actually clarifies the parallelism context. Further it may contain recalls and recall containers.

2.2 AgsChannel

The channel object does provide references to previous and next channel/pad. The link field tells us in what hierarchical position it is. All processing starts with an AgsChannel, that's why `ags_channel_recursive_play_init()`, `ags_channel_recursi` and related exist.

2.3 AgsRecycling

As its name tells us, the audio data is reused across AgsChannel upto the the next AgsRecycling. The recycling brings in a new cycle of AgsAudioSignal. It is usually reproduced by "ags-buffer" or "ags-copy" recalls. The wormhole is believed to bring major advantages, like reusable audio data and no additional amount of toplevel threads.

2.4 AgsAudioSignal

The audio data represented by AgsAudioSignal shall not take too much time to be duplicated from its template. As you might want to use larger audio files you can yield to AgsWave.

Chapter 3

Effect processors as recalls

AgsRecall is your base object to implement your own effect processor. Its class provides abstraction for most common use cases. But feel free to implement your own events to listen to.

The AgsDynamicConnectable's interface methods ::connect-dynamic() and ::disconnect-dynamic() are related to dynamic dependencies. These are dependencies requiring to be resolved. This needs to be done during AgsRecall::resolve-dependencies() event as part of the initialization process.

::run-init-pre(), ::run-init-inter() and ::run-init-post() are the 3 stages of recall initialization. This especially applies to AgsRecallAudioRun and AgsRecallChannelRun, which have a dynamic scope. These methods are called only once per lifetime of the recall.

The recall may provide AgsPort instances to modify the behaviour of your effect processor in realtime. Each port might be automated but don't have to be. The ::automate() signal does usually apply the automation data. It is provided by AgsRecallAudio and AgsRecallChannel.

AgsAutomation is segmented into chunks after AGS_AUTOMATION_DEFAULT_OFFSET number of audio tics (here each invocation AgsThread::run() is a tic), is a new automation object introduced for the very same port.

::run-pre(), ::run-inter() and ::run-post() are the 3 stages of processing. They are called continuesly in their well defined order. During one audio tic all 3 events are emitted.

As most of the computation shall occur during ::run-inter(). Some specialized recalls use ::run-pre() like "ags-play-notation" or "ags-copy-pattern". Since they produce new AgsAudioSignal objects needed by ::run-inter(). Providing feedback like "ags-peak" to a UI level indicator widget is processed during ::run-post().

::done() signal notifies about terminating the processing of recall.

3.1 LADSPA plugin host

AgsLadspaPlugin provides an object to interface with LADSPA plugins. The LADSPA plugins are collected by AgsLadspaManager. You might want to set LADSPA_PATH environment variable. Thought the processing is done by a recall - AgsRecallLadspa and AgsRecallLadspaRun. All ports are exposed by a matching AgsPort.

3.2 DSSI plugin host

AgsDssiPlugin and AgsDssiManager does basically the same as the LADSPA counterparts. The difference is, its effect processors relay on additional recalls. AgsPlayDssiAudio as well AgsPlayDssiAudioRun are responsible to play AgsNotation related keys. Likewise is AgsRouteDssiAudio and AgsRouteDssiAudioRun which does the same. The differ in overhead versus flexibility. The later is able to use its own envelope information.

Note there is no support for OSC protocol.

3.3 LV2 plugin host

AgsLv2Plugin and AgsLv2Manager is all the same as the other plugins. Like the DSSI host functionality it has AgsPlayLv2Audio and AgsPlayLv2AudioRun for instruments as well AgsRouteLv2Audio and AgsRouteLv2AudioRun.

Worker threads, presets, programs and other interfaces are supported.

Chapter 4

Supported backends

There is support for various backends like ALSA, OSSv4, JACK Audio Connection Kit, Pulseaudio and Core-Audio.

4.1 Soundcards

AgsSoundcardThread does talk to your soundcard or "virtual" soundcard by the AgsSoundcard interface. Prior only output was supported but audio input is currently developed as part of the 1.4.x release.

The thread does intermediate post synchronization in view of the audio processing thread AgsAudioLoop. It can poll your file descriptors to schedule work.

AgsClearBuffer task clears some of the ring-buffer in order next additive mixing can do its work. This is performed by "ags-play-channel-run" or alike.

AgsSwitchBufferFlag controls the currently used part of the ring-buffer.

Note polling is done by AgsPollingThread and the library maps it to AgsPollFd object.

4.2 Export to audio files

AgsExportThread does output to an audio file. Currently supported formats are WAV, FLAC, AIFF and OGG. However libsndfile provides more formats but their suffix isn't checked, yet.

AgsAudioFile is used to export your data in realtime. FYI the buffer is only flushed as the file object is closed.

4.3 MIDI instruments

AgsSequencerThread gets MIDI data from your instrument, by using AgsSequencer interface. Currently only MIDI input is supported as version 1.4.x.

The thread does intermediate pre synchronization in view of the audio processing thread AgsAudioLoop. It checks for new bytes to be read and provides it as ring-buffer.

AgsClearBuffer task clears some of the ring-buffer in order next bytes can be provided. Them are used by "ags-record-midi" recall. The MIDI note-on and note-off are interpreted to AgsNote objects.

AgsSwitchBufferFlag controls the currently used part of the ring-buffer.