Harmonicon Technical Documentation

version 0.3 from 02/20/08

Table of Contents

1 Overview

1.1 About this document

This is a technical documentation of the Harmonicon synthesizer. It mainly describes the architecture of the implementation. There are also detailed installation instructions.

1.2 About the Synthesizer

The implemented system is a real-time software synthesizer written in pure Java. It implements the SoundFont 2 standard, which defines a synthesis engine based on wavetable samples. Real-time MIDI signals and/or Standard MIDI Files (SMF) can be used to trigger notes for playback. The rendered audio stream is played on the computer's soundcard.

The main design goals were:

- modularity and flexibility
- high quality
- conforming to standards
- leverage Java's features like garbage collection and multi threading

1.2.1 Modularity and Flexibility

The implementation abstracts and encapsulates the logical units like MIDI Input, Audio Output, and the rendering engine so that changing and extending the system is easily possible. Parametrization of a wide range of aspects of the synthesizer allows fine-grained adjustments of quality, performance, and features. Most modules and settings can be changed at runtime during operation of the synthesizer. The modularity and flexibility makes it easy to compare measurement results with different settings or alternative implementations to evaluate performance vs. quality trade-offs.

1.2.2 High Quality

In order to not compromise audio fidelity, internal processing is done with 64-bit floating point precision (Java data type double), including all MIDI parameters to be ready for future MIDI standards providing higher data resolution. Output rendering can be done at an arbitrary sample rate and with up to 32-bits per sample. "Note stealing" (termination of notes for lack of resources) is avoided by allowing arbitrary polyphony.

The implementation allows very low latency and jitter to provide responsive and accurate music performance. Latency is mainly affected by the audio buffer size used in the rendering engine, but also by MIDI and audio hardware, and the quality of its drivers. To keep jitter down, we use forward synchronous timing [6], allowing the notes to be inserted with sample accurate timing into the rendered audio stream.

1.2.3 Standard Conformance

For wide interoperability and usability, no proprietary standards are used. The synthesis standard SoundFont 2 is an open standard developed by Creative Labs. Furthermore, the native accessor libraries for the hardware were implemented for Linux/GNU.

1.2.4 Standard Java Code

The implementation extensively uses Java's features like object orientation, threading/locking, garbage collection, and the class libraries. This allows a code base that is safe, portable, and easy to maintain. We explicitly avoided programming constructs that introduce complexity or that reduce flexibility and modularization.

1.3 The SoundFont 2 standard

The SoundFont 2 standard is based on a wavetable oscillator. For playback of a note, a recorded audio sample is processed and mixed together with other playing notes, then sent to the audio device. The processing step includes pitch shifting to adjust the pitch of the sample to the played note, low pass filtering, envelopes for volume, pitch, and filter and low frequency oscillators (LFO) for volume and pitch. Furthermore, many parameters can be changed in real time to affect future notes and also currently playing notes.

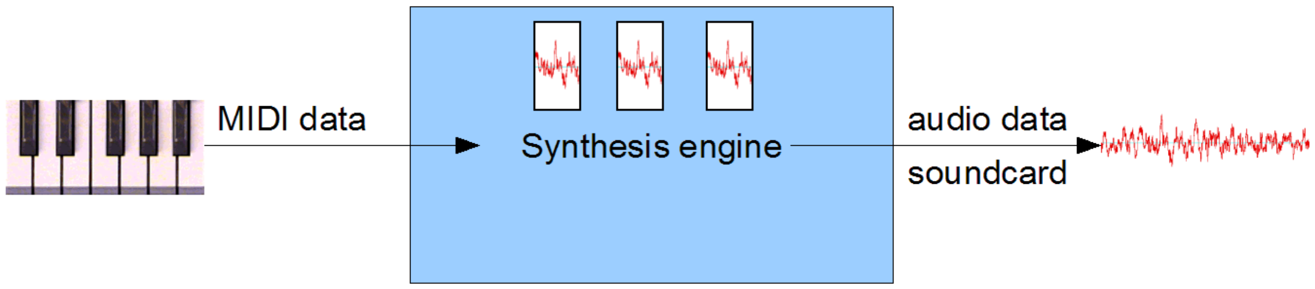
1.4 Advanced Features

- Drift compensation is used to synchronize the MIDI time stamps provided by MIDI driver and the computer's real time clock to the audio card clock.
- The engine is prepared for multi-channel operation, for future 3-D and multi-track extensions.
- The rendering engine can be set to use multiple threads, leveraging multiple CPU cores.
- For very low buffer sizes, note scheduling can be done asynchronously in order to not disrupt the audio rendering loop.
- For low latency and low jitter, we implemented low-overhead native libraries for MIDI port input and audio output.
- arbitrary polyphony, only limited by processor speed and memory
- arbitrary sample rate
- output bit depth 8, 16, 24, or 32 bits

2 Architecture

2.1 Introduction

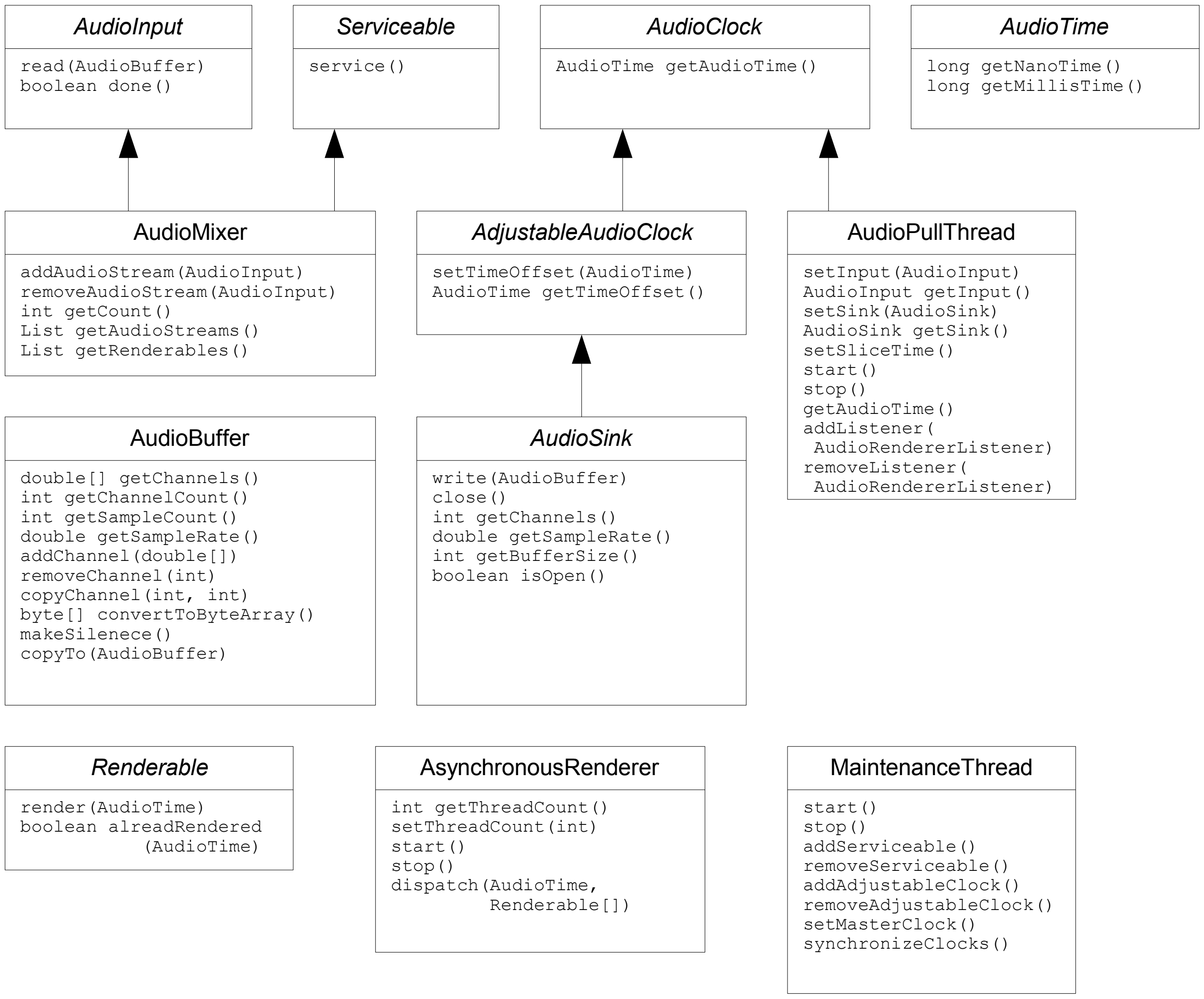
In general, the synthesizer takes MIDI notes as input and creates an audio stream with the rendered music as output. In order to render the individual notes, a soundbank is used, which governs the audio generation for each note. A widely used synthesizer architecture is a wave table, where each instrument is mainly defined by a pre-recorded sound of that instrument. The Harmonicon synthesizer currently implements a wavetable synthesizer following the SoundFont 2.01 standard.



Drawing 1: High level synthesizer overview

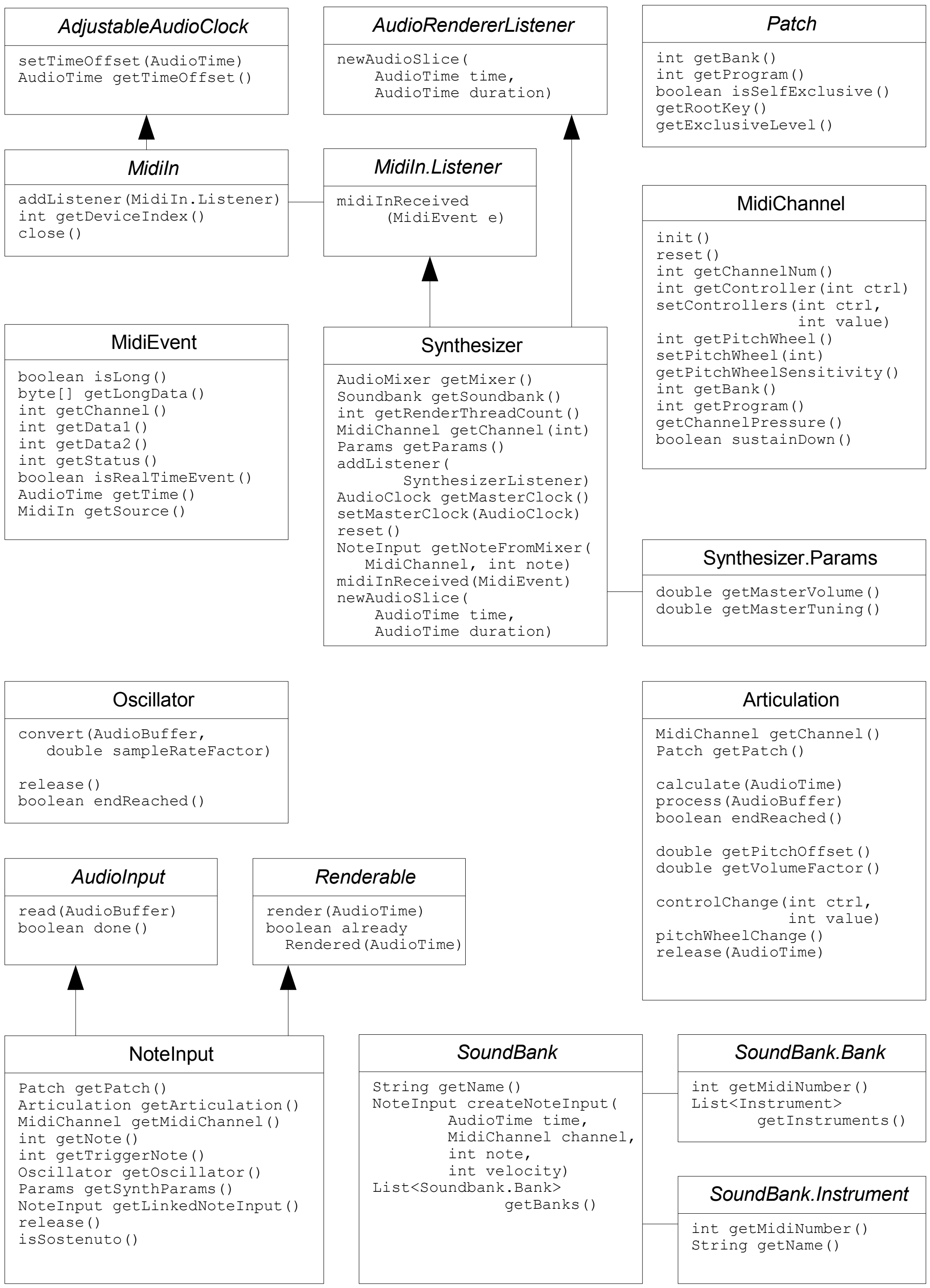
2.2 Class Diagrams

2.2.1 Audio Engine Class Diagram



Drawing 2: Audio Engine Class Diagram

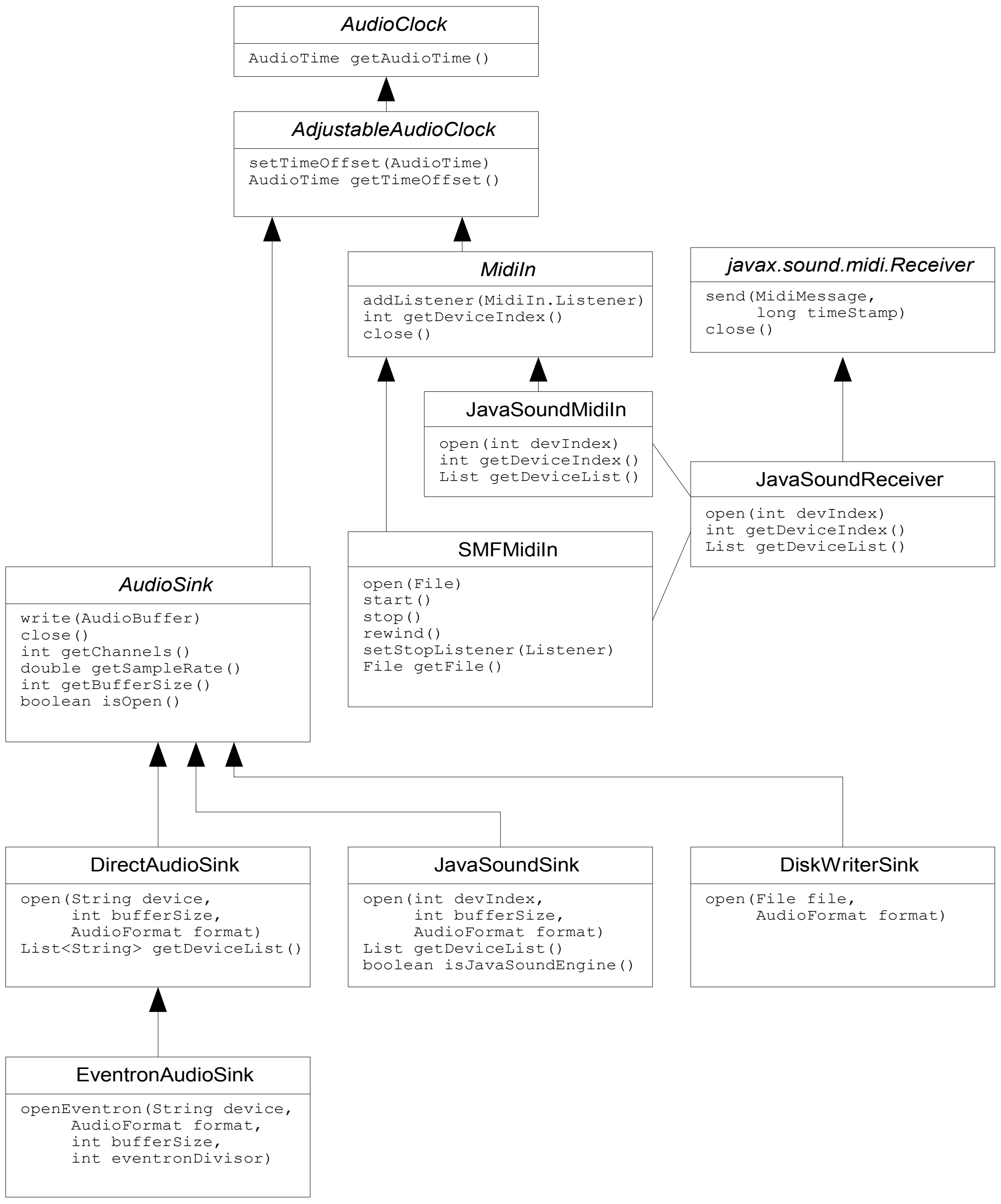
2.2.2 MIDI Engine Class Diagram



Drawing 3: MIDI Engine Class Diagram

2.2.3 Modules Class Diagrams

Interchangeable implementations of the core interfaces.



Drawing 4: Modules Class Diagrams

2.3 Data Flow

2.3.1 Default Wiring

The synthesizer is a flexible, modular system. The architecture presented here is a standard way of “wiring” the different modules for a straight-forward MIDI synthesizer. The test programs use this architecture.

Definitions/variables:

1. Define the **hardware audio format**
   The audio sink (i.e. the soundcard) needs to be opened with a particular hardware format.
2. Define the audio sink's **buffer size**/hardware latency
   This is the buffer size of the soundcard (or its driver) that is used. The soundcard is waiting until this number of samples is written to its buffer before transferring it to the soundcard's hardware buffer. The buffer size corresponds directly to a minimum hardware delay.
   A safe value for today's computers is 20 milliseconds. This value will be noticed as an unwanted delay when playing in real time on an attached MIDI keyboard. Values below 5 milliseconds could not be noticed by the keyboard player in our informal tests.
   The smaller the sink buffer, the less latency of the audio sink, but the higher the overhead per time quantum. Very small buffer sizes (<1 millisecond) will cause very high processor and i/o overhead. If overhead becomes too high, buffer underruns are occurring, with generally undefined effects.
   Also, smaller buffer sizes will require more precise scheduling, because each buffer has to be written at very precise times to the audio device. For a 1 millisecond buffer, each write must occur in a 1-millisecond window. This window, however, is also used to render the sound. If a write operation misses the deadline, the soundcard runs out of data, and an underrun occurs.
   Audible effects of underruns include scratches, quick repetitions, clicks, and bad timing.
3. Define the engine's **slice time** (aka quantum size)
   The engine generates several samples at once in a buffer. This buffer size is the quantum size. It should be equal or an integral fraction of the audio sink's buffer size. Otherwise there would be unwanted misalignments when writing rendered buffers to the audio sink, causing more latency than necessary.
   Certain actions are only executed at the beginning of a slice, therefore a larger slice time will corrupt timing of the generated notes. Unless performance constraints inhibit it, set the slice time to 1 millisecond or less.
4. Define the **number of rendering threads**
   The synthesizer engine can be run with simultaneous rendering threads. By default, the Synthesizer class checks the number of physical processor cores. If only one, no separate rendering threads are used and rendering is done sequentially from the main

mixing thread (aka AudioPullThread). Otherwise, one rendering thread is created for each processor core. Such asynchronous rendering will only be used for buffers that require rendering of more than a certain number of notes at the same time (polyphony). The polyphony threshold for asynchronous rendering is set in the variable `Synthesizer.ASYNCH_RENDER_STREAM_THRESHOLD.`
This default settings for asynchronous rendering can be overridden in class `Synthesizer` with the method `setRenderThreadCount(int)`.

5. Define the **note dispatching mode**
Starting notes is a computationally intense operation, because the instrument has to be retrieved from the soundbank, all articulation objects need to be set up and initialized. Therefore, dispatching a note from the main rendering loop can cause delays, in worst case causing an underrun. So, for very small buffer sizes, an asynchronous note dispatcher is used, which will not disrupt the main rendering loop. If a particular note takes more initialization time than would be available in the main rendering loop, it will just be inserted late, but it will not disrupt playback. So the asynchronous note dispatcher reduces likelyhood of underruns at the cost of less precise timing.
6. Define a **master clock** instance
Several components (`MidiIn` objects, audio sinks) have or can act as a clock (implementing the `AudioClock` interface). One instance should be defined as the master clock, where all other clocks are regularly offset'ed to match the master clock. Typically, the clock with the highest resolution is used, usually the soundcard's sample clock.

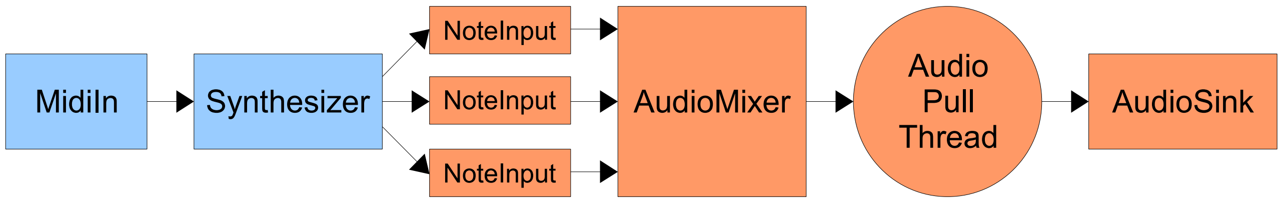
For the default setup, the following components will be created:

- a `SondFontSoundbank` object, loading the sf2 file and responsible for creating the audio input lines in form of `NoteInput` objects
- an `AudioMixer` object, owning all audio inputs that are created for the MIDI instruments
- an `AudioPullThread` object. This is the main rendering thread continously reading from the mixer and writing to the audio sink.
- the audio sink. Currently, there are 3 implementations of AudioSink available: `JavaSoundSink`, `DirectAudioSink`, and `EventronAudioSink`. The latter two use a native library on Linux for direct access to ALSA without Java Sound's overhead.
- if the audio stream should be written to a wave file in parallel, a `DiskWriterSink` object is created as well
- a `Synthesizer` object – this is the main object handling and dispatching incoming MIDI events
- a `MidiIn` object (in form of `JavaSoundMidiIn` or `DirectMidiIn`) if realtime MIDI from a MIDI port is wished
- a `MaintenanceThread` object which will regularly call registered `Serviceable`

instances and synchronize clocks, if necessary.

Finally, the objects need to be set up and “wired” in the following fashion:

- the `Soundbank` loads a .sf2 file
- the audio sink's audio format and buffer size is set
- the `AudioPullThread` is configured with the slice time. It owns the audio rendering buffers. The input line is set to the `AudioMixer` instance, the output line is set to the audio sink. If a `DiskWriterSink` was created, it is set as secondary sink for the `AudioPullThread`.
- The synthesizer is initialized with the hardware delay, so that it can compensate any audio output delay in scheduling (and offsetting) the timing of incoming events. The audio sink is set as the synthesizer's master clock. Finally, the synthesizer is registered as a listener of the `AudioPullThread`. This will enable the synth to carry out scheduling before an audio buffer is rendered from all NoteInput objects.
- The maintenance thread is configured with the master clock (the audio sink), and with one `Serviceable`, the mixer: the mixer requires regular maintenance calls for cleaning up unneeded audio lines. Furthermode, all other clock sources (namely `MidiIn` implementations) are registered as slave clocks (“adjustable clock”) so that they are synchronized in regular intervals. The maintenance thread is started.
- Finally, to start rendering, call the audio pull thread's `start()` method.



Drawing 5: General Architecture of the standard Synthesizer

2.3.2 MIDI Data Flow

As an example, the path of an incoming MIDI event is described.

A MIDI event comes in through a hardware MIDI port. An implementation of the `MidiIn` interface receives the event and passes it on to its registered listener(s). Currently, there are two implementations of `MidiIn`, namely `JavaSoundMidiIn` and `DirectMidiIn` in package `com.ibm.realtime.synth.modules.` The former uses Java Sound's MIDI facilities. Java Sound delivers MIDI events through a `javax.sound.midi.Receiver` implementation, available in Harmonicon as class `JavaSoundReceiver`. This receiver instance instanciates and dispatches a Harmonicon MidiEvent class upon receiving a MIDI event by way of Java Sound.

The `DirectMidiIn` implementation uses a native library to directly access the audio device. For Linux, this library is directmidiin.c in the `native` directory.

In the default wiring, the `Synthesizer` instance is registered as listener for the `MidiIn` implementation, so `Synthesizer`'s `midiInReceived(MidiEvent)` method is called. This method will adjust the MIDI event's time stamp to include the system's delay. If the event comes without a time stamp, the current time of the master clock (plus the system delay) is used as time stamp. Then the event is added to a FIFO list of MIDI events, the event queue (sorted by time stamp). If the event's new time stamp is before the current rendering time, the event is not enqueued and dispatched directly. This completes the `midiInReceived` call.

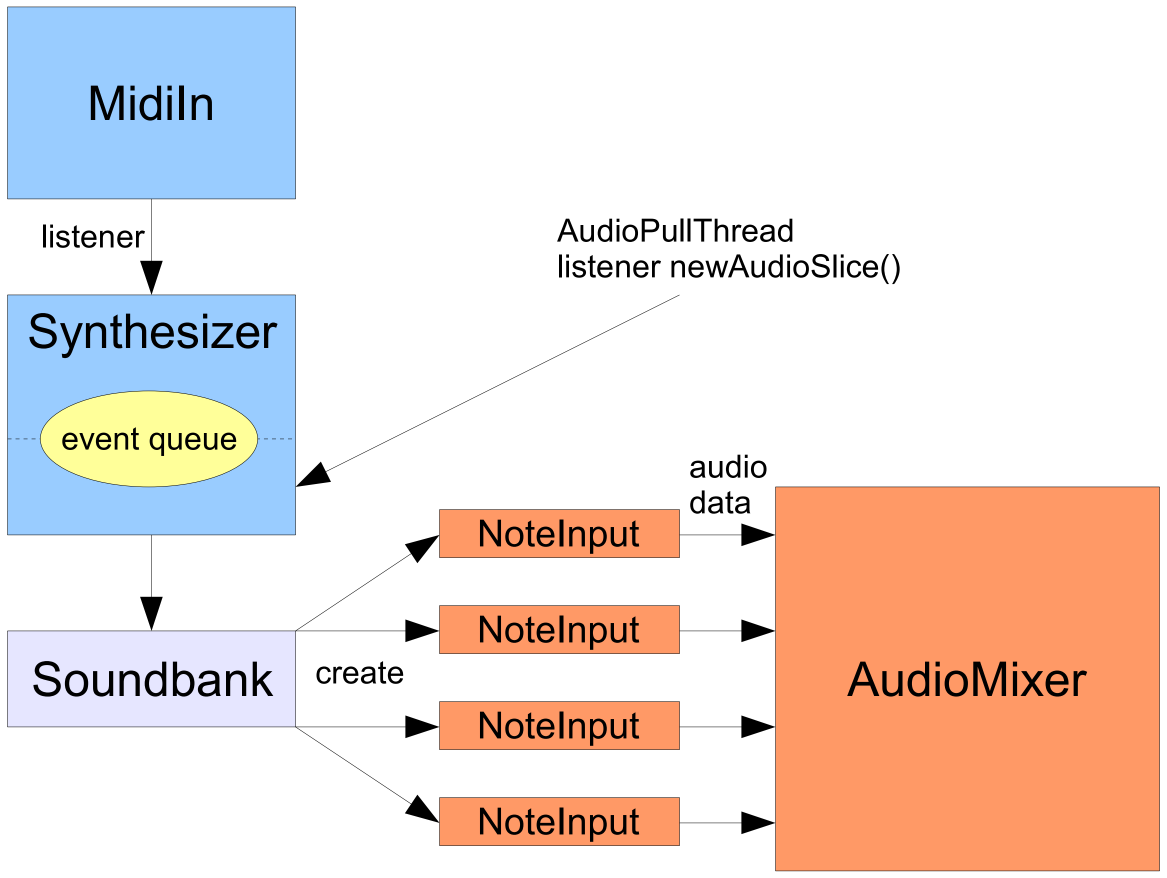
The `AudioPullThread` calls its listeners just before a new audio slice is rendered. The Synthesizer's `newAudioSlice(AudioTime time, AudioTime duration)` method is therefore called in regular intervals. Every event in the event queue with a time stamp before or equal the `time` of the next audio slice will be dispatched. Different MIDI messages cause different dispatching.

For a MIDI *Note On* message, `Soundbank`'s `createNoteInput()` method is called, returning a new `NoteInput` instance for that note. The `NoteInput` object is rendering the audio data to the audio buffers. For every played note, one `NoteInput` object is created. Multi-patches and stereo instruments will even create several `NoteInput` objects at once. The class implements the `AudioInput` interface and its instances are added as an input line to the Mixer. From now on, the mixer will mix this `NoteInput`'s data to the audio stream.

For a *Note Off* message, the corresponding `NoteInput` instance(s) currently attached to the mixer are retrieved, and its `release()` method is called. This will cause the `NoteInput` object to enter the envelopes' release segment, eventually causing the end of this note.

For controller messages, the corresponding state variables are set in Synthesizer's MidiChannel objects.

This is as far as MIDI is processed and used in the Synthesizer.



Drawing 6: MIDI Data Flow

2.3.3 Audio Data Flow

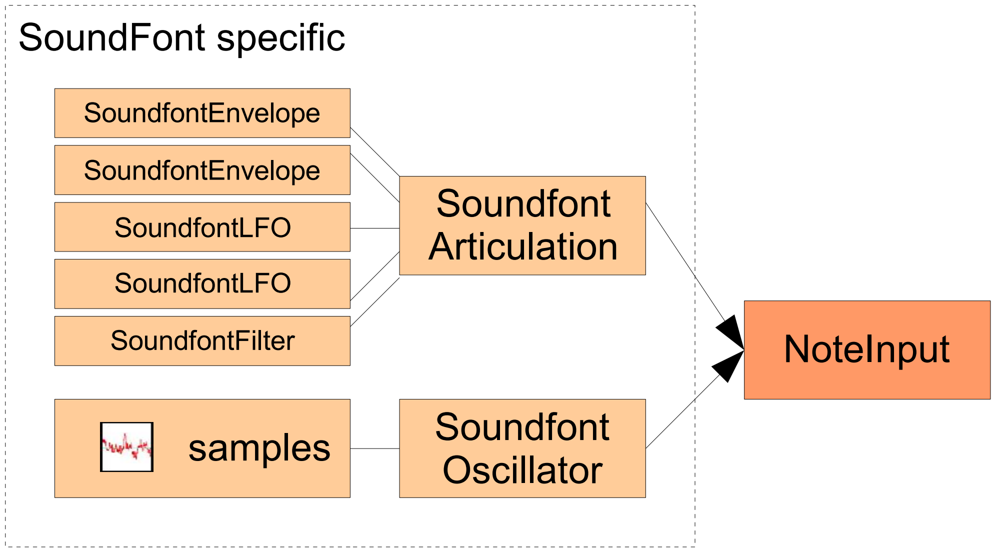
As an example, the lifetime of an audio buffer is described.

One instance of `AudioBuffer` is created by `AudioPullThread` which is reused for rendering. In a loop, the thread calls these methods:

1. `listener.newAudioSlice()`
2. `audioInput.read(buffer)`
3. `audioSink.write(buffer)`

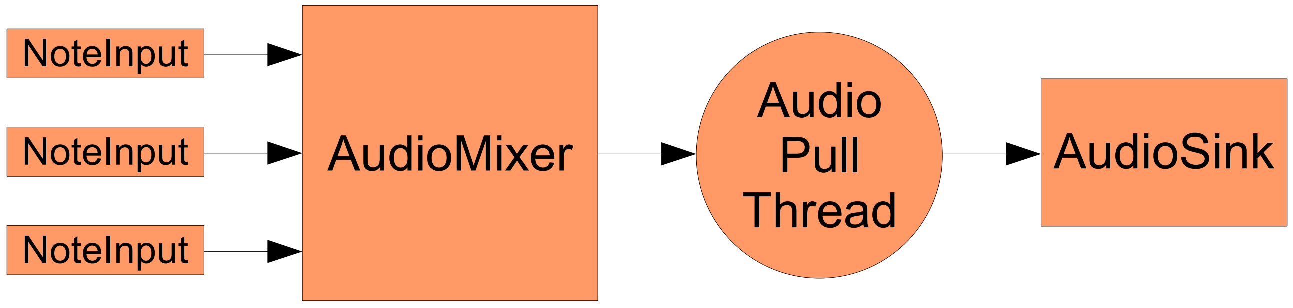
It also maintains a time counter which increases each time a buffer is written. Furthermore, the buffer needs to be cleaned (silenced) before passed to `audioInput.read()`, since the `AudioInput.read()` method is specified to *mix* into the buffer (rather than overwriting the contents).

Since the `AudioMixer` instance is registered as the audio pull thread's input, the buffer will first be passed to `AudioMixer.read()`. The mixer will loop over all its `AudioInput` input lines and call their respective `read()` method. The input lines are of the concrete class `NoteInput`, rendering the sound of an instrument, using the soundfont's sample and articulation definition. For that, the `Oscillator` provides the raw sample audio data, followed by articulation modules (volume, pitch, filter envelopes and LFO's), then the actual filter and sample rate converter to achieve different pitch and to convert the audio data to the buffer's sample rate.



Drawing 7: NoteInput with SoundFont

Every input line sequentially mixes its data to the buffer by adding the samples to the existing samples. When the mixer has finished all lines, control flow is given back to `AudioPullThread`, which will then write this buffer to the registered `AudioSink`. The `AudioSink` writes the buffer to the soundcard for playback.



Drawing 8: Audio Data Flow

2.4 Synchronization

There are several clocks in the system which need to be synchronized so that their time stamps can be used:

- the soundcard (audio out)
- incoming time stamped MIDI messages
- realtime clock of the computer

Each class that provides a clock implements the interface `AudioClock`. A sub-interface `AdjustableAudioClock` allows setting the instantaneous time (i.e. an offset) of the clock.

Inevitably, different clocks will drift and so a rudimentary drift compensation algorithm is implemented: the class `MaintenanceThread` will regularly synchronize the clocks. For that, all instances of `AudioClock` are registered with the `MaintenanceThread`, and one of them is designated as the master clock – usually the audio output clock. Now in regular intervals, the other clocks are set to the current master clock's time.

2.5 Multithreaded Rendering

This synthesizer implementation in class `Synthesizer` allows multi-threaded rendering by using the class `AsynchronousRenderer`. Recommended is to use the number of threads corresponding to the number of processor cores/hyperthreaded cores.

2.5.1 Asynchronous Execution

The audio pull thread signals the start of a new audio slice by calling its list of `AudioRendererListeners`. First, the synth dispatches all MIDI events for that slice (in the `AudioRendererListener` thread). Once that is done, it signals the `AsynchronousRenderer` thread the beginning of a new slice. From now on, all `AsynchronousRenderer` threads simultaneously render the `Renderable` lines of the mixer. Meanwhile, the `AudioPullThread` will finish serving `AudioRendererListeners` and then call the mixer's `read()` method. The mixer is unaware of any `AsynchronousRenderer` and will read all `AudioInput` lines in order. So, all threads, plus the thread in `mixer.read()` compete for rendering the line's slices.

When the mixer calls a line's `read()` method (in the thread of `AudioPullThread`), the following alternatives may happen:

- This line is not yet rendered by an asynch thread. This will cause the `read()` method to call the `render()` method, so that effectively this line is rendered in the `AudioPullThread` thread. For an optimal resource usage, this is OK.
- Or this line is already fully rendered by an asynchronous thread. The `read()` method will

notice that and just copy the pre-rendered buffer to the mix buffer.

- Or this line is currently being rendered by an asynchronous `AsynchronousRenderer` thread. This causes the entry of `AudioInput.read()` to be blocked, because `read()` and `render()` are both synchronized to the line's instance. If there are enough `AsynchronousRenderer` threads, this sort of blocking is no waste, since the asynchronous threads will sufficiently exploit all processor cores.

The first scenario above is particularly efficient since all threads start working on the lines in the same order. That means that statistically, render threads may render higher numbered lines, while the read() method already mixes the beginning lines.

2.5.2 Bottlenecks

- Render threads must not start a slice before all events for the slice are dispatched. So the dispatching action currently forces the render threads to wait.
- Once `AudioPullThread` writes to the `AudioSink`, it needs to wait until the sink returns from writing to the device. During that time, the render threads are idle, although they could already start rendering the next slice. This is a question of priority: to maintain lowest latency, any rendered buffer needs to be passed on to the audio sink as fast as possible. Now if the next slice is started before writing the last buffer to the sink, the event dispatching may delay the writing unnecessarily and in an uncontrolled way. If the next slice is started after writing, we miss the opportunity to use the blocking time of the sink for rendering.

2.5.3 Class Asynchronous Renderer

By default, this class partitions the `Renderables` in the array of to be rendered lines. That means, that if there are two render threads, the first one will render all the instances at uneven indices in the array of `Renderables`, and the second thread will render the uneven `Renderable` instances. By way of experimentation we found that this partitioning (deterministic scheduling of Renderables to threads) yields better performance.

The disadvantage is that Renderables may have very different computational requirements. So it may happen that e.g. every second Renderable is much faster to render than the others, causing the first thread to work much longer while the second thread is idle. In practice, such effects will rarely happen in such an extreme fashion.

The alternative way will let all render threads try to render all Renderable objects, using synchronization to prevent that the same Renderable is rendered multiple times. However, in practice the overhead exceeded the possiblly better exploitation of the processor cores.

2.6 Asynchronous Note Dispatching

As said above, note dispatching is a computationally intense operation: it will set up all modules for the note (e.g. filters, envelopes, etc.) and initialize the wave table for use by this note. Additionally, music tends to start multiple notes at the same time, e.g. in polyphonic chords, or drums. In tests, we have seen that for very small buffer sizes (e.g. smaller than 1ms), note dispatching could take longer than the available time quantum for this audio slice, causing a buffer underrun.

For these cases, note dispatching can be configured to be handled asynchronously. This will cause the initialization and setup code to be run in a separate thread. If many notes are inserted at the same time, and note dispatching takes longer than the slice time, some notes will be inserted into the rendered stream with the next audio slice. Although this will cause the note to be heard slightly too late, but at least there will not be a buffer underrun.

2.7 Timing

2.7.1 Time Stamps

In order to provide accurate timing, the system accepts time stamped MIDI events as input. If the events are not time stamped, it assigns them the current time as determined by the master clock (see `Synthesizer.midiInReceived`).

To this time stamp is added an offset of the current system delay (usually twice the audio buffer time). This is the time at which this note will be inserted into the audio stream.

2.7.2 Forward Synchronous Timing

We use forward synchronous timing, meaning that we know by way of the sample number, when a particular sample will be played, so we can insert notes with sample accurate timing. For that, each audio slice that is being rendered is marked with the current (calculated) time stamp with the logical time when this audio slice will be played. Now if the start of a note falls into this audio slice, rendering of it starts at the sample corresponding to the time stamp of that MIDI event.

Forward synchronous timing adds a constant delay to all MIDI events, but causes the timing of the events to be accurate to the individual samples (e.g. at 44100Hz sampling rate, events are accurate to 23 microseconds).

2.7.3 Low Latency Mode (disable Forward Synchronous Timing)

For very small buffer sizes (smaller than 1 millisecond), forward synchronous timing does not provide much increased accuracy, so you can switch it off with the parameter `-Xlowlatency`. This will generally yield better performance with sub-millisecond buffer sizes.

3 Implementation Details

3.1 Implementation Remarks

3.1.1 TODO and FIXME marks

Incomplete code or code that can be extended in future is marked with “TODO”. Development tools like Eclipse can generate a list of code lines with such TODO comments. The same applies to FIXME comments, which mark code that should be reviewed again for a possible bug or unclear intentions.

These marks should not be seen as lack of code quality.

3.1.2 DEBUG flags

Some java files have a boolean static native DEBUG flag. A centralized DEBUG flag would be nicer, but this model allows fine-grained source level turning on/off debugging for specific modules. Furthermore, these definitions can easily be searched/replaced to be a class-private DEBUG flag, which, when set to false, will cause the compiler to completely remove all code inside a “`if (DEBUG)`” block from the compiled .class file, optimizing the code by removing the `if` statements.

The DEBUG code is deliberately kept in the code, because it is a somewhat natural documentation, and it may help for future debugging sessions.

4 Running and Using Harmonicon

4.1 Prerequisites

The following components should be installed and running:

- Java 1.5 or higher
- ALSA on Linux
- SWT library

4.1.1 Installing Java

Verify that Java is installed:

```
java -version
```

The version should be at least 1.5.

To install a newer Java version, it is usually sufficient to unzip a JDK archive anywhere and place the bin directory in the PATH. If the JDK is shipped as an rpm, use

```
rpm -Uv <jdk.rpm>
```

to install it.

Use

```
rpm -l -q -i -p <jdk.rpm>
```

to see which files are installed by this rpm, and to find out the path to it.

The `Makefile` for the native libraries expects a JDK in `/usr/lib/java`, so it is a good idea to make sure that this is a valid directory or a symbolic link pointing to the base directory of a JDK.

4.1.2 ALSA

Most likely, ALSA is already installed on your system. Version 1.0.0 or later is required. You can find out the version of ALSA by typing

```
more /proc/asound/version
```

If this file does not exist, ALSA is not running – most likely it is not installed then. If the script exists, you can use

> `/etc/init.d/alsasound start`
>
> or `/etc/init.d/alsasound stop`.

Which hardware is configured to be used by ALSA (and which drivers to use) is configured in `/etc/modprobe.conf` (aka `/etc/modules.conf` aka `/etc/conf.modules`). Most

Linux distributions have GUI programs to set up these entries.

Playing a file with ALSA directly is done with

```
aplay $BASE/Sounds/wav/pcm16_s_44.wav
```

If this does not produce audible sound,

1. check the volume levels with `alsamixer` or the window manager's GUI audio mixer.
2. check your audio cable setup

Checking out MIDI functionality can be done with amidi:

See the installed MIDI devices:

```
amidi -l
```

MIDI dump, print out all incoming MIDI data:

```
amidi -p hw:2,0,0 -d
```

(you may need to specify another device, from the list of MIDI devices)

For information how to install ALSA, check out the Old Installation instructions (next chapter).

4.1.3 Installing SWT

You can download SWT from `http://www.eclipse.org/swt/` . Unzip the files into `lib/linux` (or `lib/windows`) of the RunSynth or SystemTest package directories.

4.2 RunSynth Package

This binary distribution of Harmonicon contains most needed components for running Harmonicon in various configurations.

4.2.1 Running Harmonicon from the RunSynth Package

Refer to the `Install.pdf` document included with the binary distribution.

4.2.2 Creating the RunSynth Package

The workspace creates a script to create a tarball with all required files for running Harmonicon.

The following commands will create the tarball:

```
        cd scripts
        sh createRunSynthPackage.sh
```

If any components are missing, it will tell you. It will include the following components from the workspace:

- native JNI libraries for direct audio and direct MIDI
  You can compile the native libraries in directory `native` by running `make` (see chapter 4.7 below).
- soundbank (from media directory in workspace)
- MIDI files (from media directory in workspace)
- SWT files: need to be copied to `lib/linux` and, if needed, `lib/windows`.
- start scripts for running the console or GUI version

4.3 SystemTest Package

This binary distribution of Harmonicon contains most needed components for running Harmonicon's system test. The system test is used for gathering traces from many test runs in different configurations and VMs.

4.3.1 Running the System Test from the SystemTest Package

Refer to the `Install.pdf` document included with the binary distribution.

4.3.2 Creating the SystemTest Package

Creating the system test tarball is very similar to creating the RunSynth tarball. Please follow the instructions above, replacing `createRunSynthPackage.sh` with `createSystemTestPackage.sh`.

Additionally, the system test package can or should include JDK's, so it's worthwhile populating the `scripts/SystemTest/jdks` directory with one or more JDKs following the naming conventions outlined in the `readme.txt` file in that directory.

4.4 Eclipse/CVS

Harmonicon was developed with Eclipse, providing some convenience for creating the jar file or the javadoc.

You should start by checking out the Harmonicon workspace from CVS:

Connection type:	`extssh`
User:	`synthcvs`
Host:	`12fb.com`
Port:	`12656`
Repository Path:	`/home/synthcvs/cvsroot`
Module:	`Synthesizer`

You can easily create synth.jar by right-clicking on MakeLocalJar.jardesc and selecting *Create Jar*.

4.5 CVS access

4.5.1 ssh access to 12fb.com

This is optional unless you want to update the synth from the development repository.

For CVS access, you need to set up ssh access to 12fb.com:

Hostname: 12fb.com

Port: 12656

Username: synthcvs

It's easiest to make these settings by editing the file `~/.ssh/config` and adding these lines:

```
Host 12fb.com
        Port 12656
        Protocol 1
Host synthcvs
        HostName 12fb.com
        User synthcvs
        Protocol 1
        Forward X11 no
        Port 12656
```

The CVS scripts use the synthcvs alias as hostname. This makes it easy to move the repository to a different server by just editing `~/.ssh/config`.

You can test this configuration by calling

```
ssh synthcvs
```

It should ask for the synthcvs password and then give you a shell on 12fb.com.

For additional security and convenience, use host based authentification. This requires generating a public/private key pair and adding the public key to the authorized keys on 12fb.com.

Generate the keys with

```
ssh-keygen -t rsa1
```

Respond all queries with [ENTER] (default locations are fine, no need for a password).

Copy the public key to 12fb.com:

scp ~/.ssh/identity.pub synthcvs:

and append it to its list of authorized keys:

```
ssh synthcvs
cat identity.pub >> ~/.ssh/authorized_keys
```

Now you should be able to ssh into 12fb.com without providing a password. This mechanism is more secure, because no password needs to be transmitted for authorization. It requires, however, that access to the computer/login is not generally available.

4.5.2 Get/Update the CVS workspace

The source code is maintained on a CVS repository on the server `12fb.com`. For CVS, use these environment variables to use the synthcvs ssh settings (see previous chapter):

```
export CVSROOT=synthcvs@synthcvs:/home/synthcvs/cvsroot
export CVS_RSH=ssh
```

The CVS module is `Synthesizer`.

After setting the variables above, use the following command to download a fresh local repository from CVS:

```
        cvs -z3 co Synthesizer
```

Use this command to update the local CVS copy:

```
        cvs -z3 update -d -P Synthesizer
```

4.6 Compile Harmonicon's Java classes

Creating the JAR is easiest from Eclipse, but you can also do it from the command line.

Harmonicon's source code is under `Synthesizer/com`. Before compiling, make sure that the SWT jar and native libs are available, e.g. in `Synthesizer/lib`. Then, you will use the following commands to create `synth.jar` in `Synthesizer/lib`:

```
        cd Synthesizer
        javac -d classes \
            -cp xlib/tuningForkTraceGeneration.jar:\
                xlib/xrts.jar:lib/swt.jar \
            `find -name *.java`
        (cd classes ; jar c0f ../lib/synth.jar *)
```

4.7 Compile Harmonicon's native libs

Harmonicon has direct ALSA support via a JNI library. The source code is available in `Synthesizer/native` and can be compiled using the `Makefile`. Compiling the JNI library requires ALSA headers (`asound.h`, usually in `/usr/include/alsa`) and the ALSA library (`libasound.so`, usually in `/usr/lib`). Also, the Makefile assumes /usr/lib/java to be the base directory of a JDK installation.

Compile with

```
make
```

If compilation fails for incompatibility or missing files, re-install ALSA as outlined above.

The two .so files need to be added to the library path, usually by defining them in LD_LIBRARY_PATH. The RunSynth and SystemTest scripts expect the libraries in `lib/native`.

4.8 Compile Harmonicon's native libs for 64-bit Linux

Note: the 64-bit version has only been tested on amd64 (aka EM64T on Intel) systems.

Make sure the following are installed:

- gcc
- alsa headers (in debian: package libasound2-dev)
- JDK 5.0 or higher
- the Harmonicon CVS workspace

How to compile:

- in the workspace, go to `Synthesizer/native`
- adapt the `Makefile`:
  uncomment 64bit and comment out 32 bit:
  ```
  SYSDEFINE     = -DSYSTEM64BIT
  #SYSDEFINE      = -DSYSTEM32BIT
  ```
- verify JDK path
- verify source and libs paths

Then compile as described in the previous chapter.

5 Old Installation Instructions

These are old instructions for installation. With availability of the RunSynth package, these shouldn't be necessary anymore.

The synthesizer's full potential is only available on Linux, so these instructions are for Linux.

5.1 Prerequisites (Outdated)

A running Linux system with

- kernel 2.6 or higher
- root/sudo access

5.2 Directory Outline (Outdated)

All files are installed under a base directory, in the following referred to as `$BASE`.

The package installs these directories under `$BASE`:

5.2.1 ALSA

This directory serves an installation from scratch of ALSA. In most cases, it won't be needed, as modern Linux systems ship with ALSA by default.

5.2.2 bin

Directory for executables and symlinks. It is particularly used for symlinks to the java executables like `java`, `javac`, etc. Different symlinks are used to choose a particular VM.

5.2.3 include

A directory with native header files for compilation of native libraries. Usually, it is not needed. Only if the system does not ship with ALSA header files (check for `/usr/include/alsa/asoundlib.h`), this directory can be used for ALSA headers.

5.2.4 Install

This directory contains installation archives e.g. for SWT, ALSA, JDK, etc. This is for convenience only, and you may want to use updated versions, if available.

5.2.5 JVMs

This directory serves to unify access to one or more JVMs. It is intended to contain symlinks, e.g. `jdk5sun` or `jdk5ibm` pointing to the root of an installed Java SDK. The symlinks in

`$BASE/bin` use these links. You can also directly install private copies of JDKs here.

5.2.6 lib

A directory for Java jar libraries (e.g. for SWT), and for native libraries in the `native` subdirectory.

5.2.7 ListMixer

This directory contains a Java Sound test application, which shows a list of installed soundcards. You can use it to verify proper installation of Java and ALSA. Note that Harmonicon ships with its own ALSA audio driver, so this test program is left here reference only.

5.2.8 ShowMidi

This directory contains a Java Sound test application, which shows a list of installed MIDI devices. You can use it to verify proper installation of Java and ALSA.

5.2.9 Sounds

This directory contains sounds, soundbanks, and music files in wav, sf2, and mid format. The sf2 files are required for Harmonicon. The mid files can be used to render entire songs with Harmonicon.

5.2.10 Synth

This is the directory containing the actual work directory for Harmonicon. There are many scripts for

- compiling Harmonicon
- starting the different incarnations of Harmonicon, e.g. the GUI version or the command line version,
- running automated performance tests
- updating the workspace from CVS

The subdirectory Synthesizer contains a local CVS workspace.

5.3 Installation Steps (Outdated)

A full installation requires

- ALSA 1.0.0 or higher
- Java SDK 1.5 or higher

5.3.1 ALSA checks

Most likely, ALSA is already installed on your system. Version 1.0.0 or later is required. You can find out the version of ALSA by typing

```
more /proc/asound/version
```

If this file does not exist, ALSA is not running – most likely it is not installed then. If the script exists, you can use

```
        /etc/init.d/alsasound start
```

or `/etc/init.d/alsasound stop`.

Which hardware is configured to be used by ALSA (and which drivers to use) is configured in `/etc/modprobe.conf` (aka `/etc/modules.conf` aka `/etc/conf.modules`). Most Linux distributions have GUI programs to set up these entries.

Playing a file with ALSA directly is done with

```
aplay $BASE/Sounds/wav/pcm16_s_44.wav
```

If this does not produce audible sound,

3. check the volume levels with `alsamixer` or the window manager's GUI audio mixer.
4. check your audio cable setup

Checking out MIDI functionality can be done with amidi:

See the installed MIDI devices:

```
amidi -l
```

MIDI dump, print out all incoming MIDI data:

```
amidi -p hw:2,0,0 -d
```

(you may need to specify another device, from the list of MIDI devices)

5.3.2 ALSA installation

If you need to (re)install ALSA, follow these steps (as root):

1. Remove an existing ALSA installation
   Stop ALSA:
   ```
   /etc/init.d/alsasound stop
   ```
   Find existing rpm packages:
   ```
   rpm -qa | grep alsa
   ```
   Uninstall them with
   ```
   rpm -e --nodeps <package name>
   ```
   E.g.:

```
#  rpm -qa | grep alsa
alsa-lib-1.0.6-5.RHEL4.i386
alsa-utils-1.0.6-5.i386
# rpm -e -nodeps alsa-lib-1.0.6-5.RHEL4.i386
# rpm -e -nodeps alsa-utils-1.0.6-5.i386
```

2. Get the latest ALSA drivers, libs, and utils from `http://www.alsa-project.org` (or use the ones in the `$BASE/Install` directory)
3. untar the 3 archives into `$BASE/ALSA`
4. as root, use the Makefile in `$BASE/ALSA` by calling
   `make install_alsa`
   This should compile and install ALSA drivers, lib, and utils.
5. If ALSA is not running yet, start it with `/etc/init.d/alsasound start`

Follow the previous chapter to verify the ALSA installation.

5.3.3 JDK checks

Verify that Java is installed:

```
java -version
```

The version should be at least 1.5.

5.3.4 JDK symbolic links

The presented way of setting up symbolic links for the choice of the JDK is a convenient proposal, but are not mandatory. Other setups can be used, but then the convenience start scripts will fail or need to be adapted.

The central location to choose a particular JDK is in $BASE/JVM

It contains the following symbolic links:

`$BASE/JVMs/jdk5ibm`

This symlink points to an IBM JDK version 1.5, e.g. `/opt/ibm/java2-i386-50`.

`$BASE/JVMs/jdk5sun`

This symlink points to a Sun JDK version 1.5, e.g. `$BASE/JVMs/jdk1.5.0_05`.

`$BASE/JVMs/jdk5metronome`

This symlink points to an IBM realtime JDK version 1.5, e.g. `/opt/ibm/WebSphereRealTime-1.0`

The start scripts use the following symlinks which should exist (otherwise the start scripts cannot be used, or have to be modified):

`$BASE/bin/java`

> This symlink points to a default java executable, e.g. `$BASE/JVMs/jdk5ibm/bin/java`

`$BASE/bin/java5`

> This symlink points to a default java 1.5 executable, e.g. `$BASE/JVMs/jdk5ibm/bin/java`

`$BASE/bin/java5ibm`

> This symlink points to the IBM java 1.5 executable, should point to `$BASE/JVMs/jdk5ibm/bin/java`

`$BASE/bin/java5sun`

> This symlink points to a Sun java 1.5 executable, should point to `$BASE/JVMs/jdk5sun/bin/java`

`$BASE/bin/java5metronome`

> This symlink points to an IBM realtime java 1.5 executable, may use a different JDK than the regular IBM VM.

`$BASE/bin/javac`

> This symlink points to a default javac executable, e.g. `$BASE/JVMs/jdk5ibm/bin/javac`

`$BASE/bin/javac5`

> This symlink points to a default javac 1.5 executable, e.g. `$BASE/JVMs/jdk5ibm/bin/javac`

5.3.5 JDK Installation

Usually, it is sufficient to unzip a JDK archive anywhere and adjust the corresponding symlink in `$BASE/JVMs`. If the JDK is shipped as an rpm, use

```
rpm -Uv <jdk.rpm>
```

to install it. Use

```
rpm -l -q -i -p <jdk.rpm>
```

to see which files are installed by this rpm, and to find out the path to it.

The `Install` directory may contain one or more (stripped) JDK's ready to be unzipped.

5.3.6 SWT installation

This is optional if you only need the command line version of Harmonicon.

The SWT GUI requires the SWT libraries to be installed. The start scripts will pick up the SWT library if it is installed in `$BASE/lib` and if the native libs are installed in `$BASE/lib/native`. For that, unzip the SWT distribution (download from www.eclipse.org or use the one in `Install`), and copy `swt.jar` to `$BASE/lib` and `*.so` to `$BASE/lib/native`.

5.3.7 ssh access to 12fb.com

This is optional unless you want to update the synth from the development repository.

For CVS access, you need to set up ssh access to 12fb.com:

Hostname: 12fb.com

Port: 12656

Username: synthcvs

It's easiest to make these settings by editing the file `~/.ssh/config` and adding these lines:

```
Host 12fb.com
        Port 12656
        Protocol 1
Host synthcvs
        HostName 12fb.com
        User synthcvs
        Protocol 1
        Forward X11 no
        Port 12656
```

The CVS scripts use the synthcvs alias as hostname. This makes it easy to move the repository to a different server by just editing `~/.ssh/config`.

You can test this configuration by calling

```
ssh synthcvs
```

It should ask for the synthcvs password and then give you a shell on 12fb.com.

For additional security and convenience, use host based authentification. This requires generating a public/private key pair and adding the public key to the authorized keys on 12fb.com.

Generate the keys with

```
ssh-keygen -t rsa1
```

Respond all queries with [ENTER] (default locations are fine, no need for a password).

Copy the public key to 12fb.com:

scp ~/.ssh/identity.pub synthcvs:

and append it to its list of authorized keys:

```
ssh synthcvs
cat identity.pub >> ~/.ssh/authorized_keys
```

Now you should be able to ssh into 12fb.com without providing a password. This mechanism is more secure, because no password needs to be transmitted for authorization. It requires, however, that access to the computer/login is not generally available.

5.3.8 Get/Update the CVS workspace

This is optional unless you want to update the synth from the development repository.

The source code is maintained on a CVS repository on the server `12fb.com`. For CVS, use these environment variables to use the synthcvs ssh settings:

```
CVSROOT=synthcvs@synthcvs:/home/synthcvs/cvsroot
CVS_RSH=ssh
```

The CVS module is `Synthesizer`. These settings are available in `$BASE/Synth/settings.sh` for convenience.

The script `$BASE/Synth/cvs_update` will retrieve the latest version from CVS. The script `$BASE/Synth/cvs_checkout` will download a fresh local repository from CVS. All the CVS controlled files are under the `$BASE/Synth/Synthesizer` directory.

5.3.9 Compile Harmonicon's Java classes

You will only need to compile if you've updated the source files or if `$BASE/Synth/synth.jar` is missing.

Harmonicon's source code is under `$BASE/Synth/Synthesizer/com`. A script $BASE/Synth/compile can be used to compile the classes and create a synthesizer jar. It executes the following steps:

- clean previously compiled class files
- compile the classes in `$BASE/Synth/Synthesizer/com`. The compiled class files are written to `$BASE/Synth/classes`.
- Create the `$BASE/Synth/synth.jar` containing all required class files.

5.3.10 Compile Harmonicon's native libs

Most likely it will not be necessary to compile this library. A compiled copy of `libdirectaudiosink.so` in `$BASE/lib/native` should be sufficient.

Harmonicon has direct ALSA support via a JNI library. The source code is available in `$BASE/Synth/Synthesizer/native` and can be compiled using the `Makefile`. Compiling the JNI library requires ALSA headers (`asound.h`, usually in

`/usr/include/alsa`) and the ALSA library (`libasound.so`, usually in `/usr/lib`).

Compile it with

```
make
```

and copy it:

```
cp libdirectaudiosink.so $BASE/lib/native
```

If the headers are not available on the system, you can unzip them from `$BASE/Install/alsa_headers.tgz`. If compilation fails for incompatibility or missing files, re-install ALSA as outlined above.

5.4 Running the Test Programs (Outdated)

5.4.1 settings.sh

This include script contains variable definitions with common paths. All start scripts include the settings.sh file.

If the directory layout is retained as described in the previous chapter, no changes should be necessary to the file. Otherwise you can change global options here.

5.4.2 The command line program

The command line program is most useful for automated execution, or for performance measurements, where no GUI is interfering with performance. The main class for the command line program is `com.ibm.realtime.synth.test.SoundFont2Synth`. Running it is conveniently done by way of the `runConsole-*` scripts in `$BASE/Synth`. The scripts pass on all parameters to the program. Start by getting a list of supported parameters:

```
runConsole-ibm -h
```

It'll also show the available audio and MIDI devices.

The command line test program does not remember any settings. All options need to be specified on the command line.

5.4.3 The AWT GUI test program

The outdated AWT test program provides a GUI for convenient choice of options. All settings are made persistent in `~/.harmonicon.conf`. Run the `runGUI-*` scripts without parameters.

5.5 *Troubleshooting (Outdated)*

5.5.1 No Sound / Soundbank not available

The synthesizer requires a soundbank in SoundFont 2 format. Any such file can be used, however in practice a General MIDI SoundFont is the most useful choice, allowing correct playback of common MIDI files.

5.5.2 Direct Audio not available

Direct audio sink is not available (i.e. only Java Sound audio is available): the direct audio sink's native lib must be in the library path. First, make sure that libdirectaudiosink.so exists in `$BASE/Synth/Synthesizer/native`. Then either copy it to a directory in the predefined library path (`/usr/lib`, if you use the scripts: `$BASE/lib/native`), or set `LD_LIBRARY_PATH` accordingly. See the previous chapter for more information.

5.5.3 Direct MIDI is not available

If direct MIDI input is not available (i.e. only Java Sound MIDI devices appear in list), it probably means that the native lib is not available in the library path. As for the direct audio lib, make sure that libdirectmidiin.so exists in `$BASE/Synth/Synthesizer/native`. Then either copy it to a directory in the predefined library path (`/usr/lib`, if you use the scripts: `$BASE/lib/native`), or set `LD_LIBRARY_PATH` accordingly. See the previous chapter for more information.

6 Acronyms

Acronym	*Expanded name*	*Notes*
API	Application Programming Interface	
SWT		
GUI	Graphical User Interface	
I/O	Input/Output	
JDK	Java Development Kit	
JNI	Java Native Interface	
JVM	Java Virtual Machine	
LFO	low frequency oscillator	