PPM-java

Technical documentation

PH, 2017-01-13

1 Introduction

According to an Article on Wikipedia[1], "a peak programme meter (PPM) is an instrument used in professional audio for indicating the level of an audio signal." It is different from a normal VU meter in that it has a very short rise time (integration time) and a long return time. This allows audio producers to continouusly monitor the peaks of a programme signal. **ppm_java** implements a PPM type II which has a rise time of 23 dB in 10 ms and a fall time of -24dB in 2800ms.

The audio level maps to the meter scale as follows:

Input level [dB]		Meter mark	
min	**max**	**min**	**max**
-130	-24	0	1
-24	-20	1	2
-20	-16	2	3
-16	-12	3	4
-12	-8	4	5
-8	-4	5	6
-4	0	6	7
0	...	7	7

1 https://en.wikipedia.org/wiki/Peak_programme_meter

2 Running

System requirements

ppm-java requires

- Java (Tested with Java 1.8.)
- JackD audio server (Tested with version 1.9.10)

Test setup was a Linux machine (Kubuntu 14); program hasn't been tested on other platforms.

Example session

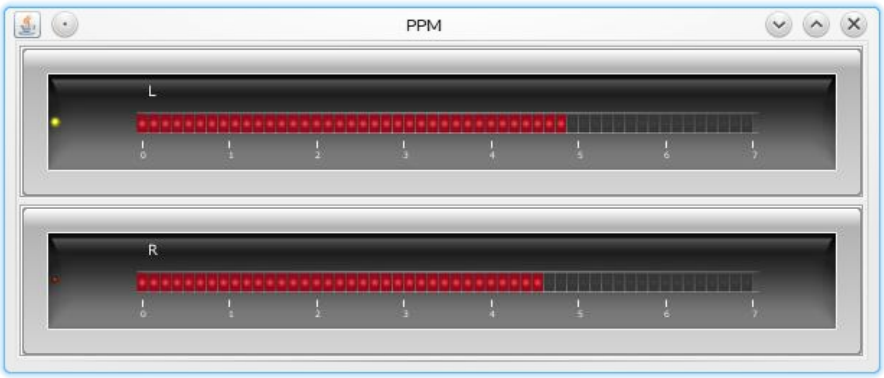
If JackD isn't running, start it:

```
user@local:~$ /usr/bin/jackd -r -dalsa -dhw:0 -r44100 -p1024 -n3 -Xraw
```

Start **ppm-java** from the commandline. We'll run it with the graphical linear gauge frontend:

```
user@local:~$ java -jar /path/to/ppm.jar -u guiLinear -l /var/log/ppm.log
```

This will bring up a horizontal linear gauge. If the meter is receiving audio data the gauge will show the current audio levels:

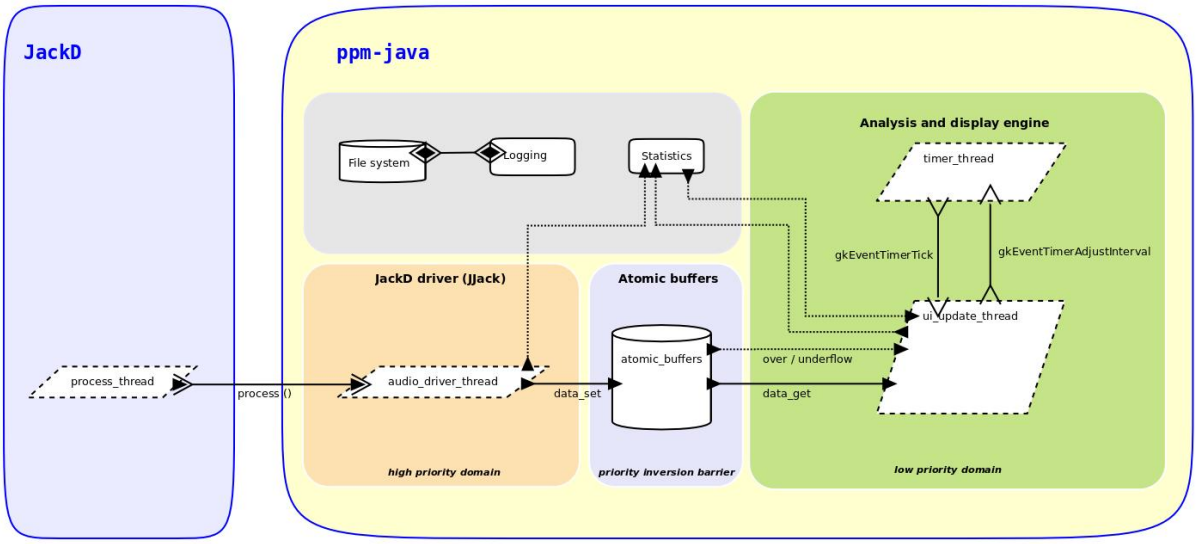


3 Implementation

Challenges

- Good balance between scalability and fitness for the purpose. The challenge is to create a framework which is *well suited for the task* whilst being *generic enough to accomodate further development*.[2]
- Making two parallel activities with different priorities work together. On one side we have got a high priority activity (the **`jackd`** process), on the other side we've got a low priority activity (user interface updates). Both sides run independently of each other, i.e. in separate threads. *We have to avoid priority inversion* where the low priority thread (user interface updates) blocks the high priority thread (**`jackd`**).
- Minimizing data loss. The high priority **`jackd`** thread needs to work need for a protection against priority inversion brings up the possibility of data loss. For example we get data loss if the user interface thread blocks, whilst the **`JackD`** thread keeps pushing audio data. The challenge is to build in some *mechanism that minimizes data loss*.
- Keeping performance up. We need a framework which is *able to process the audio data without much latency*, so that updates on the meter happen almost immediately after the corresponding audio has come into the meter. There's always some latency with any audio application, but for a a metering solution the latency should be less than 25ms.

Model



`ppm-java` continouusly receives chunks of audio samples from the Jack server, and, for each chunk,

- computes the absolute peak value,
- converts it from raw sample values to decibels,

2 This is a hard balancing act - it's tempting to totally overengineer and create micromodules or to go the shortcut route and hack some contorted monolithic construction. In both cases, half a year later even the author won't have a clue what this thing does!

- applies a stepping integrator (PPM ballistics),
- and displays the result on a frontend.

Two activites are happening here: On the one hand, Jack delivers sample chunks, and, on the other hand, the program updates a display. These are two distinct activities, and both run in separate threads. The Jack side of things is handled by the audio driver thread, and everything else runs within the analysis and display engine thread.

The audio driver thread is time critical as it integrates with the Jack server. This means we cannot afford to have the audio driver run slow on us, as this would affect the Jack server and potentially the whole ecosystem of clients connected with the Jack server. The analysis and display engine (GUI side) not time critical - if it hangs for a few moments it's user annoying but at least it doesn't pull the entire Jack system down with it! For the program to run well we need to prevent priority inversion where the low priority thread (display engine) blocks the high priority thread (audio driver). We need to decouple the display engine from the audio driver in such a way that the audio driver won't get stuck (even if the display engine is frozen) and the display engine receives the audio data with minimal data loss.

Because of the need for decoupling we cannot send data directly from the audio driver to the display engine. Instead we use a bank of special buffers (one per channel) which receives data from the audio driver and holds it ready for the display engine when it fetches the data. The buffers are designed such that setting and fetching are atomic, i.e. only one thread a time can read/write data to the buffer (That's why we call them 'atomic buffers'). The bank of buffers acts as a priority inversion barrier between the display engine and the audio driver.

Modules

Behind the simple purpose of this program is a modular machinery implementing it. The modular design is on purpose as it makes the program much easier to adapt to new situations. For example, all frontends except the **`consoleText`** frontend include PPM ballistics. Other features may follow in later versions. For example, the user might be interested in a history view to see the last minute of level measurements. Or, the meter could be used as a silence detector, sending an alarm event somewhere if the audio level falls below a threshold for a prolonged time. **`ppm-java`** comprises the following modules:

`TAudioContext_JackD`	The audio driver (backend).
`TNodePump`	Data pump. Fetches audio data from the attached atomic buffer and passes it on to the next module. Data is fetched and passed on each time a **`gkEventTimerTick`** is received.
`TNodePeakEstimator`	Calculates the absolute peak value of a sample chunk. Being absolute, the value is always positive.
`TNodeConverterDb`	Converts a sample (peak) value from raw level to dB (deci bel). Conversion according to $y = 20 \log_{10} (x)$ where x: normalized absolute sample value [Vnorm], $0 <= x <= 1$ y: sample value [dB]

TNodeIntegrator_PPMBallistics	A stepping approximator to emulate PPM ballistics.
VFrontend	The frontend used for the current session.

The various modules can be connected; once done they work together as a complete whole.

Backend

ppm-java runs as a Jack[3] client. The Jack server connects audio programs and hardware so they can communicate with each other. Jack runs in it's own process, **jackd**. This means that there must be a running instance of **jackd** before starting **ppm-java**. Once both of them are running, **ppm-java** will find the running j**ackd** server and register as a client. The user can then connect **ppm-java** to outputs of any other clients registered with **JackD** and display the audio levels on a frontend.

Through the Jack audio server, ppm-java can receive audio data from attached hardware or from other Jack compatible programs. All communication with Jack is encapsulated in the module **TAudioContext_JackD** which relies on the **JJack** library[4] for the actual connection with Jack.

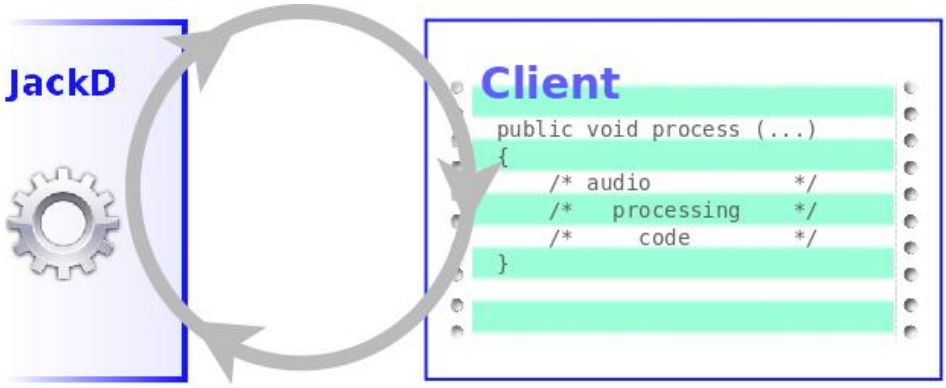
JJack makes it easy to write Jack clients. Here's a client that receives audio from a port on the Jack server and prints peak sample values on the console (full program, see appendix 1, listing 1):

```
package ppm_java._dev.concept.trial.JJack;

import... /* Various imports */

/**
 * Loads the JJack driver, connects to a running instance of JackD and prints
 * the peak of every incoming frame to stdout.
 *
 * @author peter
 */
public class TDev_Trial_JJack_process_01 implements JJackAudioProcessor
{
    public static void main (String[] arge) {/* Setup stuff... */}

    private int fIFrame = 0;
    public TDev_Trial_JJack_process_01 ()
    {
        fIFrame = 0;
    }

    public void process (JJackAudioEvent e)
    {
        FloatBuffer         inBuf;
        int                 nSamples;
        int                 i;
        float               s;
        float               sRect;
        float               peak;


        fIFrame++;
        inBuf       = e.getInput (0);
        nSamples    = inBuf.limit ();
        peak        = 0;
        for (i = 0; i < nSamples; i++)
        {
            /* Get next sample. */
            s = inBuf.get (i);

            /* Rectify (i.e. mirror a negative sample to it's positive opposite). */
            sRect = (s < 0)  ?  -s : s;

            /* Determine peak value. */
            peak = (sRect > peak)  ?  sRect : peak;
        }
        System.out.println ("Frame: " + fIFrame + "; Peak value (abs): " + peak + ".");
```

3 http://www.jackaudio.org
4 https://sourceforge.net/projects/jjack

```
    }
}
```

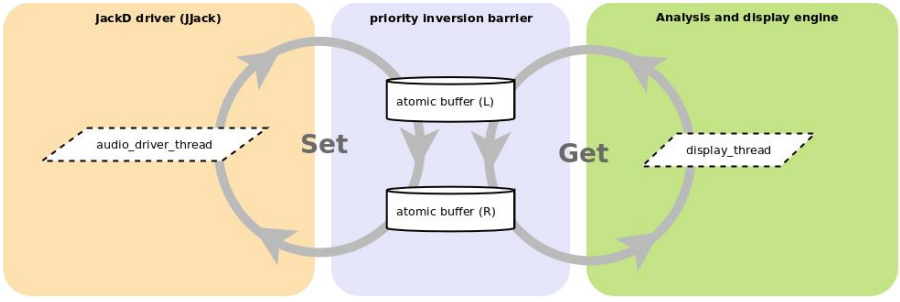
The central piece of this code is the method **`process(JJackAudioEvent e)`**. The method receives one parameter of type **`JJackAudioEvent`** (provided by **`JJack`**) which contains everything needed to extract and push audio data from/to the Jack server. All the client's sample processing is done in the **`process(...)`** method. The Jack server (via the **`JJack`** bridge) calls this method in an endless cycle (several times per second) until the program ends.



The audio driver thread is time critical, i.e. a call to **`process(...)`** method must be guaranteed to complete in a fixed time shorter than the cycle time of the **`jackd`** instance. Here's the program output:

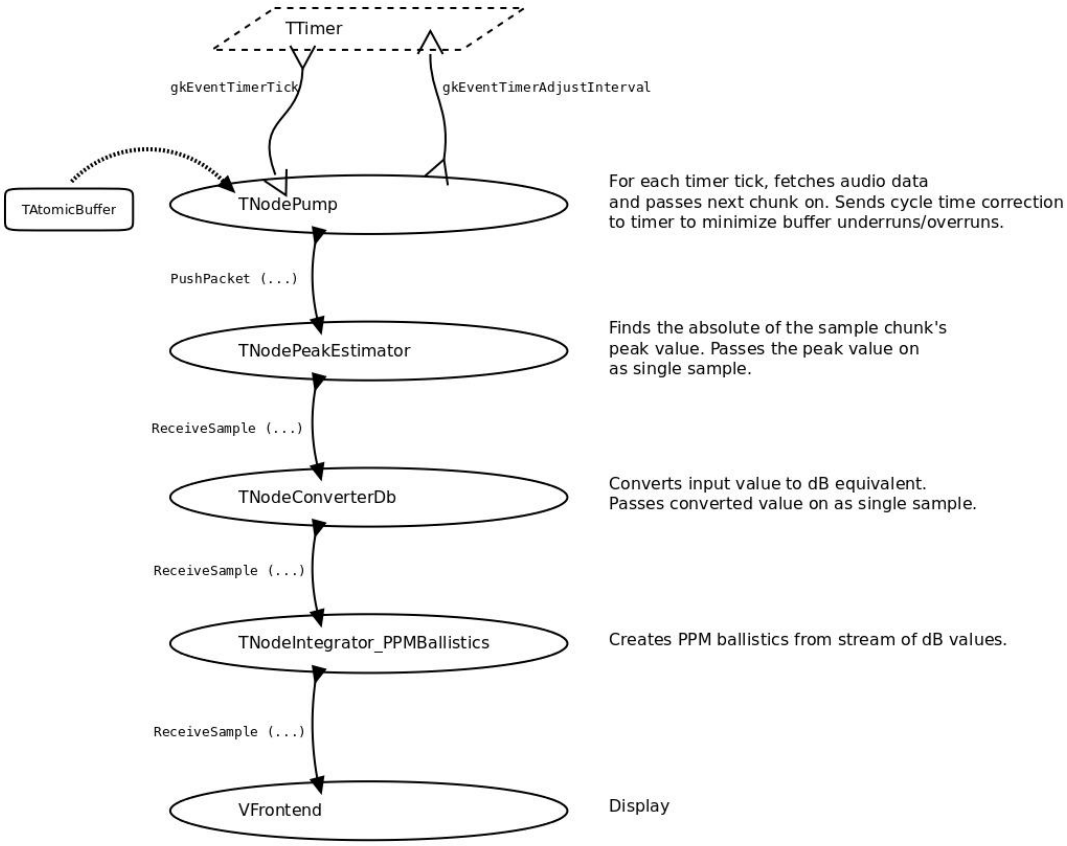
```
native jjack library loaded using system library path
natively registering jack client "JJack"
jack_client_new: deprecated
using 2 input ports, 2 output ports
Frame: 1; Peak value (abs): 0.0.
Frame: 2; Peak value (abs): 0.0.
Frame: 3; Peak value (abs): 0.0.
Frame: 4; Peak value (abs): 0.042320143.
Frame: 5; Peak value (abs): 0.036235876.
Frame: 6; Peak value (abs): 0.038238987.
Frame: 7; Peak value (abs): 0.038017016.
Frame: 8; Peak value (abs): 0.04076365.
Frame: 9; Peak value (abs): 0.022242684.
Frame: 10; Peak value (abs): 0.022647541.
```

In **`ppm-java`** we cannot push the data directly to the display engine. If the engine blocks whilst it receives data it would result in the audio driver being stuck in the **`process(...)`** method (priority inversion). To this from happening we push the audio data to a bank of atomic buffers (one per audio channel) from where the display engine will fetch it. The atomic buffers are designed to stream data from a high priority thread to a low priority thread without ever blocking the high priority thread. This way we decouple the time critical audio driver thread from the low priority display engine thread. The display engine might still get stuck - but now the only consequence would be lost data. The audio driver (i.e. the jack server) will continue to work, even if the display engine freezes up.



Analysis and display engine

The main work of **`ppm-java`** is done inside the analysis and display engine. This device takes the raw audio data and converts it to a stream of decibel values to display them on a front end. The engine comprises multiple modules, each one doing a small sub task of the data gathering, transformation and display.



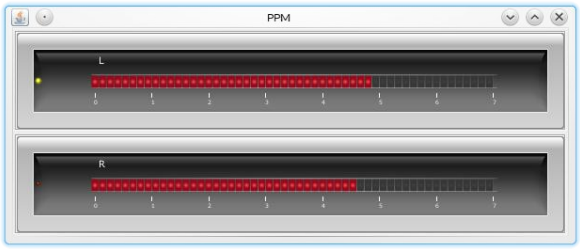
The analysis and display engine main work of ppm-java is done inside the low priority display thread.

Frontends

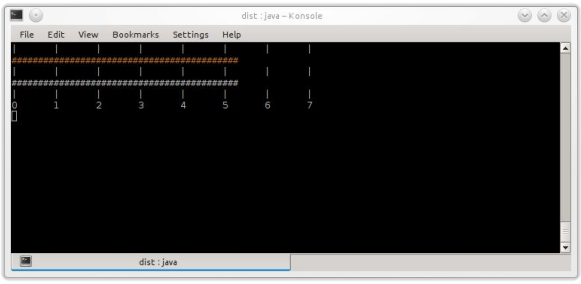
Various frontends are available. Some offer graphical user interface, others work in text-only mode, i.e. in the console from which **`ppm-java`** has been started. Most of them include PPM ballistics in the display.

guiRadial A PPM lookalike.

guiLinear Horizontal linear bargraph.

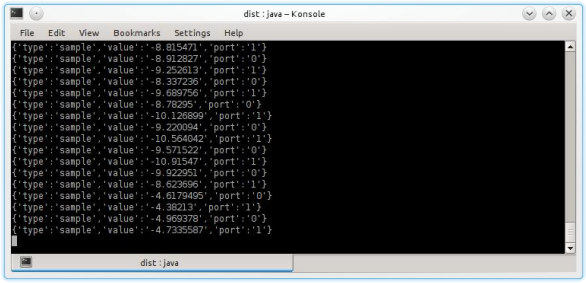


consoleLinear Horizontal linear bargraph on the console (text mode).



consoleText Text stream to stdout. Data is presented in JSON format. This frontend is designed to stream audio level data to another application - e.g. via a pipe.

When **ppm-java** runs with this frontend, there will be no PPM ballistics, i.e. the peak values will be displayed directly, without integration.



Atomic buffer [TODO]

We could have made our program very short and include all code inside the **`process(...)`** method:

```
public void process (JJackAudioEvent e)
{
    /* for each input channel...                    */
    /*     find peak in e.getInput()         */
    /*     convert peak value to dB          */
    /*     interpolate to PPM ballistics     */
    /*     update GUI                        */
}
```

Whilst this is a really nice shortcut, it's not a good idea to implement it! The **`process(...)`** method is time critical, and we would have put a lot of non-time-critical code into the time critical section that way - which could well choke the Jack server.if our non-time-critical code gets stuck somewhere. Furthermore we would update the GUI with the frequency of Jack calling the **`process(...)`** method which might be too slow for a smooth visual appearance.

Therefore we need a solution where we decouple the time critical section (in **`process(...)`**) from all the other activity of our program and place all the GUI updates into code that runs in another thread, separately from the Jack server process.

The central piece to the solution is the atomic buffer, implemented as class (**`ppm_java.util.storage.TAtomicBuffer`**). This class stores the audio samples coming from the input side and releases them to the GUI side when requested. The buffer is atomic in the sense that one thread at a time can read/write data to/from the buffer.

Class hierarchy [TODO]

4 Roadmap

- Finish documentation (est. one week).
- True peak capabilities. So far, we just take the highest absolute sample value per GUI cycle and use that for the display. However, in reality most peaks fall between peaks on the actual waveform. This means our meter does not display the waveform peaks but only the sample peaks, resulting in an underread of typically 3dB. A truepeak module is needed to reduce the underread. We would design the truepeak module according to the recommended algorithm in BS.1770-4[5] (est one week).
- Port to C++/QT. The **`ppm-java`** program is a proof of concept. And it was simply easier to do the design in Java (e.g. we don't have to keep track of header files). Once the design has matured, it should be fairly straight forward to port the program to C++, using the QT toolkit and other support libraries (est. four weeks).

5 BS.1770-4, Annex 2. https://www.itu.int/dms_pubrec/itu-r/rec/bs/R-REC-BS.1770-4-201510-I!!PDF-E.pdf

5 Appendices

App. 1: Example listings

Listing 1:

```
package ppm_java._dev.concept.trial.JJack;

import java.nio.FloatBuffer;
import de.gulden.framework.jjack.JJackAudioEvent;
import de.gulden.framework.jjack.JJackAudioProcessor;
import de.gulden.framework.jjack.JJackException;
import de.gulden.framework.jjack.JJackSystem;
import ppm_java.util.logging.TLogger;

/**
 * Loads the JJack driver, connects to a running instance of JackD and prints
 * the peak of every incoming frame to stdout.
 *
 * @author peter
 */
public class TDev_Trial_JJack_process_01 implements JJackAudioProcessor
{
    public static void main (String[] args)
    {
        TDev_Trial_JJack_process_01 processor;

        /* Setting up Jack client and terminator thread.  */
        processor  = new TDev_Trial_JJack_process_01 ();

        /* Connecting with the running instance of Jack */
        JJackSystem.setProcessor (processor);

        /* Terminator timer - must run in separate thread */
        new Thread ()
        {
            public void run ()
            {
                try
                {
                    Thread.sleep (500);
                    JJackSystem.shutdown ();
                }
                catch (JJackException | InterruptedException e)
                {
                    e.printStackTrace();
                }
                System.exit (1);
            }
        }.start ();
    }

    private int fIFrame;

    public TDev_Trial_JJack_process_01 ()
    {
        fIFrame = 0;
    }

    /**
     * The process callback. Called by the Jack server via the JJack bridge. Time critical!
     */
    @Override
    public void process (JJackAudioEvent e)
    {
        FloatBuffer         inBuf;
        int                 nSamples;
        int                 i;
        float               s;
        float               sRect;
        float               peak;

        fIFrame++;
        inBuf       = e.getInput (0);
        nSamples    = inBuf.limit ();
        peak        = 0;
        for (i = 0; i < nSamples; i++)
        {
            /* Get next sample. */
            s = inBuf.get (i);
```

```
            /* Rectify (i.e. mirror a negative sample to it's positive opposite). */
            sRect = (s < 0)  ?  -s : s;

            /* Determine peak value. */
            peak = (sRect > peak)  ?  sRect : peak;
        }
        System.out.println ("Frame: " + fIFrame + "; Peak value (abs): " + peak + ".");
    }
}
```

Glossary

Priority inversion

> In computer science, priority inversion is a problematic scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively "inverting" the relative priorities of the two tasks.
>
> This violates the priority model that high priority tasks can only be prevented from running by higher priority tasks and briefly by low priority tasks which will quickly complete their use of a resource shared by the high and low priority tasks.
>
> https://en.wikipedia.org/wiki/Priority_inversion

6 License

A program to display audio levels using PPM ballistics.