PPM-java

Technical documentation
PH, 2017-01-13

This guide has been written by a dude who isn't a technical writer.

Written with thanks to my better half who patiently endured my long computer sessions!

Table of Contents

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.)

- Jack audio server[2] (Tested with version 1.9.10)

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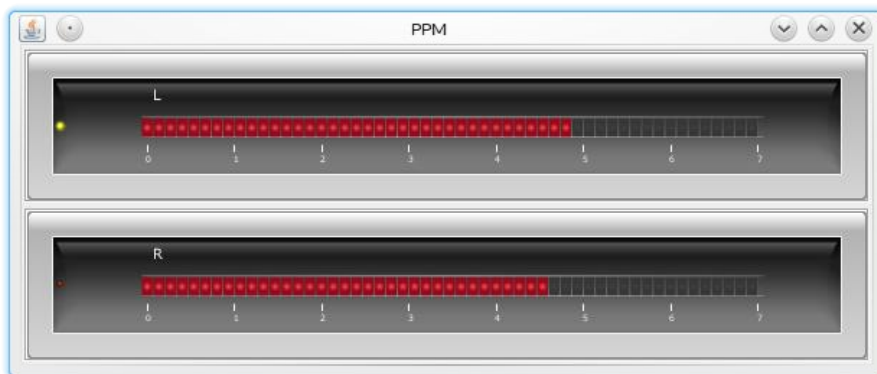
Example session

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:

---

2   The Jack server connects audio programs and hardware so they can communicate with each other. See http://www.jackaudio.org

3 Implementation

Design overview

ppm-java runs as a Jack client. 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 **jackd** 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** receives 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[3] for the actual connection with Jack.
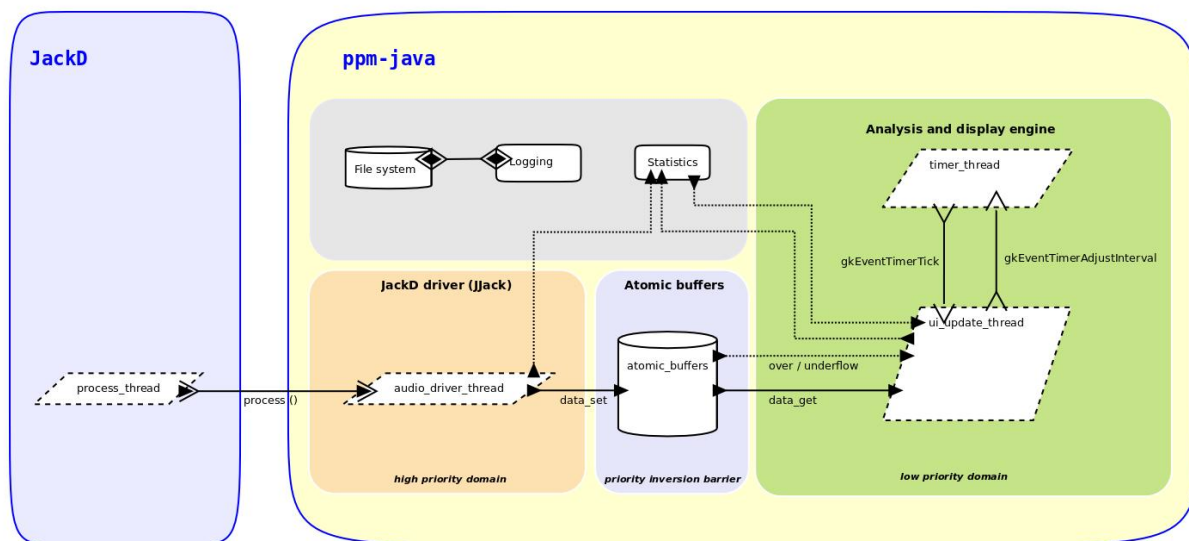
The program is designed in a modular fashion which requires a framework supporting the modular nature. This modular arrangement increases complexity initially, but the makes the design much more flexible. For example, it was easy to accomodate a special case for the **consoleText** frontend in that we don't include PPM ballistics. Also, a modular design aids development of new features. 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.

When running, **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,

- applies a stepping integrator (PPM ballistics),

- and displays the result on a frontend.

---

3   https://sourceforge.net/projects/jjack

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 display engine thread. Connecting both sides together presents an interesting challenge as we link a high priority thread (audio driver, time critical) and a low priority thread (display engine, may hang sometimes).

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[4]. The display engine (GUI side) isn't 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! We need to prevent priority inversion where the low priority thread (display engine) blocks the high priority thread (audio driver). We have 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) whilst 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.

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*[5].

- *We have to protect against priority inversion* where the low priority thread (display engine) blocks the high priority thread (audio driver).

- At the same time we have to *minimize data loss* between the audio driver and the display engine - as the protection will de-integrate them.

- We need 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.

---

4   Might not be as dramatic (Jack has got protection measures against slow clients), but it isn't prudent to assume external safety. No hard feelings and I do respect the Jack developers, but for our application development the safest assumption is the worst case presumption: Jack's a sensitive soul and has a crisis with slow clients. Let's have a well behaved client, not a liability!
5   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!

4 Multithreading

`ppm-java` makes extensive use of multithreading. As example, the atomic buffer (class `TAtomicBuffer`) stores the audio samples coming from the input side and releases them to the GUI side when requested. Setting data and fetching data is done by two different threads, at the same time. One thread constantly pushes data to the buffer, whilst the other thread constantly fetches it from the buffer. This is a classic producer/consumer scenario.
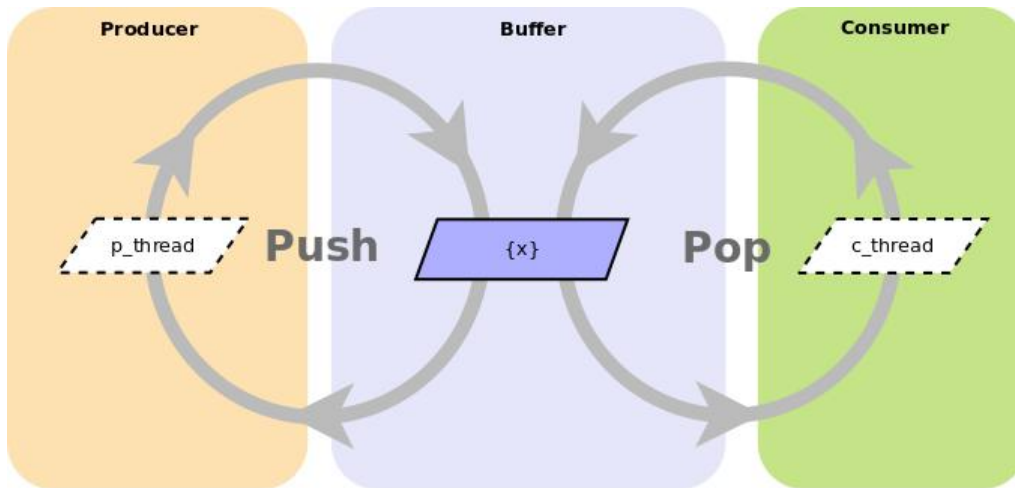
Without precautions, if multiple threads access the same object we can get surprising results. The following program has two theads simultaneously access a shared buffer. One thread pushes data, the other thread pops it[6]. No precautions are taken to make this program "multithread-proof".

---

6   Note that, in this program, push and pop just increment / decrement a counter. The point of the demo still holds.

Main program:

```
package ppm_java._dev.concept.example.multithread.unsafe;

public class TDev_Example_multithread_unsafe
{
    public static void main (String[] args)
    {
        TDev_Example_multithread_unsafe          unsafeClient;

        unsafeClient = new TDev_Example_multithread_unsafe ();
        unsafeClient.start ();
    }

    private TThreadConsumer          fConsumer;
    private TThreadProducer          fProducer;
    private int                      fValue;

    /**
     *
     */
    public TDev_Example_multithread_unsafe ()
    {
        fConsumer   = new TThreadConsumer (this);
        fProducer   = new TThreadProducer (this);
        fValue      = 0;
    }

    public void start ()
    {
        fConsumer.start ();
        fProducer.start ();
    }

    public void Pop ()
    {
        System.out.println ("Entering Pop  ().  Number: " + fValue);
        if (fValue > 0)
        {
            fValue--;
            try {Thread.sleep (500);} catch (InterruptedException e) {}
        }
        System.out.println ("Exiting  Pop  ().  Number: " + fValue);
    }

    public void Push ()
    {
        System.out.println ("Entering Push ().  Number: " + fValue);
        if (fValue < 1)
        {
            fValue++;
            try {Thread.sleep (500);} catch (InterruptedException e) {}
        }
        System.out.println ("Exiting  Push ().  Number: " + fValue);
    }
}
```

Producer:

```
package ppm_java._dev.concept.example.multithread.unsafe;

class TThreadProducer extends Thread
{
    private TDev_Example_multithread_unsafe          fHost;

    public TThreadProducer (TDev_Example_multithread_unsafe host)
    {
        fHost = host;
    }

    @Override
    public void run ()
    {
        int  i;
        int  delay;

        delay = 0;
        for (i = 1; i <= 10; i++)
        {
            fHost.Push ();
            delay += 10;
```

```
            try {Thread.sleep (delay);} catch (InterruptedException e) {}
        }
    }
}
```

Consumer:

```
package ppm_java._dev.concept.example.multithread.unsafe;

class TThreadConsumer extends Thread
{
    private TDev_Example_multithread_unsafe            fHost;

    public TThreadConsumer (TDev_Example_multithread_unsafe host)
    {
        fHost = host;
    }

    @Override
    public void run ()
    {
        int  i;
        int  delay;

        delay = 200;
        for (i = 1; i <= 10; i++)
        {
            fHost.Pop ();
            delay -= 10;
            try {Thread.sleep (delay);} catch (InterruptedException e) {}
        }
    }
}
```

The output of the program shows that `Push()` and `Pop()` are executed at arbitrary times. The producer and consumer just tread on each other's toes. The result is messed up data.

```
Entering Push ().  Number: 0
Entering Pop  ().  Number: 0     <--- Pop() should not be executed here.
Exiting  Push ().  Number: 0     <--- Number should be 1.
Exiting  Pop  ().  Number: 0
Entering Push ().  Number: 0
Entering Pop  ().  Number: 1
Exiting  Push ().  Number: 0
Entering Push ().  Number: 0
```

Thread cooperation primitives.

Since version 1.0 Java has support for multithreaded applications. Most notably, the language provides the `synchronized` keyword. This keyword allows us to make methods or code blocks thread-safe. The `synchronized` keyword creates a protected section of code that can only be entered by one thread a time. Such a section is called a *critical section*.

To make our previous example multithread proof we need to declare the main program's `Push()` and `Pop()` methods as `synchronized`.

```
public synchronized void Pop ()
{
    System.out.println ("Entering Pop  ().  Number: " + fValue);
    if (fValue > 0)
    {
        fValue--;
        try {Thread.sleep (500);} catch (InterruptedException e) {}
    }
    System.out.println ("Exiting  Pop  ().  Number: " + fValue);
}

public synchronized void Push ()
{
    System.out.println ("Entering Push ().  Number: " + fValue);
    if (fValue < 1)
    {
        fValue++;
        try {Thread.sleep (500);} catch (InterruptedException e) {}
    }
    System.out.println ("Exiting  Push ().  Number: " + fValue);
}
```

This creates order, as both threads politely wait for each other! Now, our data is correct:
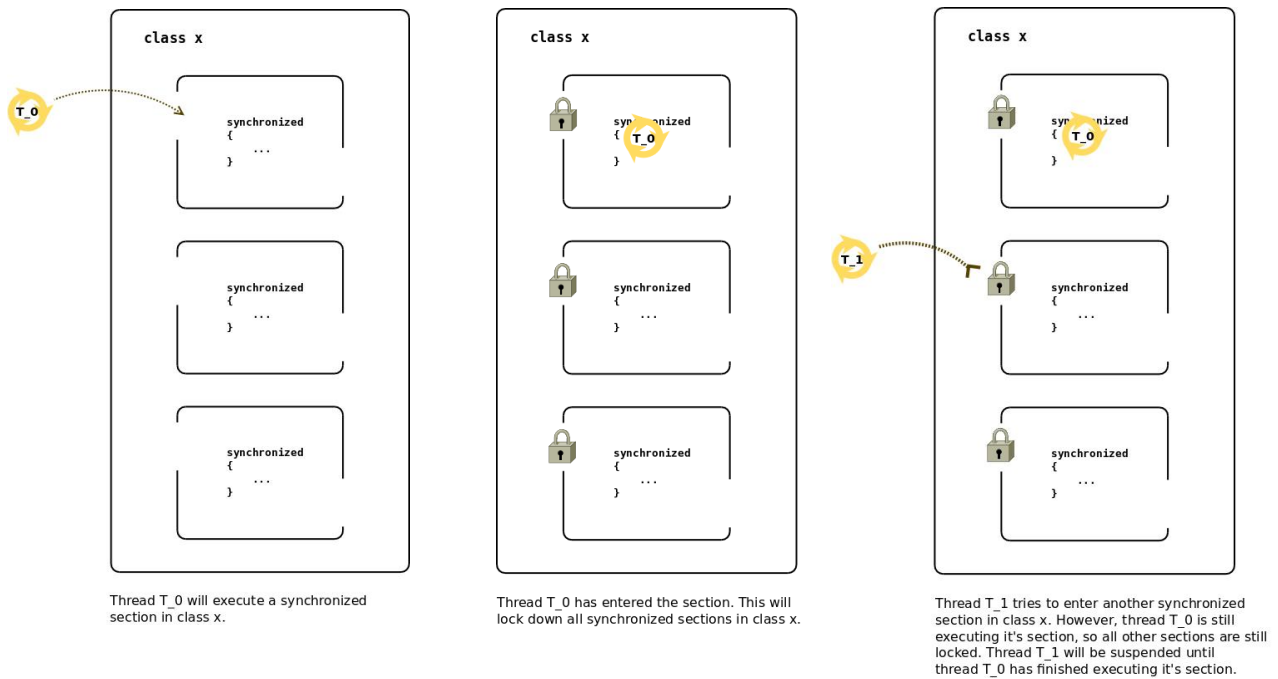
```
Entering Pop  ().  Number: 0
Exiting  Pop  ().  Number: 0
Entering Push ().  Number: 0
Exiting  Push ().  Number: 1
Entering Pop  ().  Number: 1
Exiting  Pop  ().  Number: 0
Entering Push ().  Number: 0
Exiting  Push ().  Number: 1
Entering Pop  ().  Number: 1
Exiting  Pop  ().  Number: 0
```

The **synchronized** keyword turns a method (or block of code) into a protected section. When a thread executes such a section then any other thread trying to execute the same section is suspended until the first thread has finished. This is the reason why the synchronization in our example program works. If the producer thread has entered the **Push()** method, and the consumer tries to enter the **Pop()** method before the producer has finished, then the consumer must wait until the producer has left the **Push()** method.

This mechanism operates object wide. Once a thread has acquired a protected section in an object, all other synchronized parts of that object are locked. The object is the lock; once it's acquired by a thread, no other thread can acquire it until the first thread has released the lock. In our example, both, **Push()** and **Pop()** are synchronized on the instance of **TDev_Example_multithread_safe**. This forces the producer and the consumer to wait for each other, ensuring data integrity.

Thread T_0 will execute a synchronized section in class x.

Thread T_0 has entered the section. This will lock down all synchronized sections in class x.

Thread T_1 tries to enter another synchronized section in class x. However, thread T_0 is still executing it's section, so all other sections are still locked. Thread T_1 will be suspended until thread T_0 has finished executing it's section.

Wait problem

Synchronization means waiting. If one thread enters a synchronized section, all other threads have to wait (a thread trying will be suspended). This can block a thread for an uneccessarily long time. Here is an example which demonstrates the problem of the long wait:

Main program:

```
package ppm_java._dev.concept.example.multithread.wait;

public class THouse
{
    private static final int        gkNumVisitors  = 5;
    private static final long       gkTimeAudience = 1000;

    public static void main (String[] args)
    {
        THouse              house;

        house = new THouse ();
        house.GetMeSomeVisitors ();
    }

    private TVisitor[]              fVisitors;

    public THouse ()
    {
        int i;

        fVisitors = new TVisitor [gkNumVisitors];
        for (i = 0; i < gkNumVisitors; i++)
        {
            fVisitors[i] = new TVisitor (this, i);
        }
    }

    public void GetMeSomeVisitors ()
    {
        int i;

        for (i = 0; i < gkNumVisitors; i++)
        {
            fVisitors[i].start ();
        }
    }

    public synchronized void Visit (int id)
    {
        try {Thread.sleep (gkTimeAudience);} catch (InterruptedException e) {}
    }
}
```

Visitor:

```
package ppm_java._dev.concept.example.multithread.wait;

class TVisitor extends Thread
{
    private THouse                  fHouse;
    private int                     fID;
    private String                  fPreamble;

    public TVisitor (THouse house, int id)
    {
        fHouse          = house;
        fID             = id;
        fPreamble       = "Visitor #" + fID;
    }

    public void run ()
    {
        long            t0;
        long            t1;
        long            dT;

        t0 = System.currentTimeMillis ();
        fHouse.Visit (fID);
        t1 = System.currentTimeMillis ();
        dT = t1 - t0;

        System.out.println (fPreamble + ": Had an audience! Time spent: " + dT + "ms.");
    }
}
```

A group of visitors would like to visit a house to have an audience with the home owner. The owner

speaks to one visitor a time, for exactly one second. All others have to wait. Unfortunately, the output shows excessive waiting times for most visitors:

```
Visitor #0: Had an audience! Time spent: 1000ms.    <--- Thread #0: Zero wait time!
Visitor #4: Had an audience! Time spent: 2000ms.
Visitor #3: Had an audience! Time spent: 3000ms.
Visitor #2: Had an audience! Time spent: 4000ms.
Visitor #1: Had an audience! Time spent: 5001ms.    <--- Thread #1: Four seconds wait time!
```

The problem is the synchronization mechanism: When one thread acquires a lock, all other threads trying to acquire the same lock will be suspended until the lock is released. And that can be a long time waiting in suspense. For a realtime thread such as the audio driver in `ppm-java`, any wait is prohibitive.

Wait free access

Since version 1.5 Java offers atomic variables in the package `java.util.concurrent.atomic`. Atomic variables are variables manipulated by one single processor instruction. As a side effect, this makes them inheritly thread-safe[7].

`java.util.concurrent.atomic` offers several types of atomic variables, but of special interest is the type `java.util.concurrent.atomic.AtomicInteger`, and especially it's method `compareAndSet()`. The proper signature of this method is

```
public final boolean compareAndSet (int expect, int update)
```

`compareAndSet` atomically sets the value to `update` if the current value equals `expect`. The method returns `true` if the value was successfully changed and `false` if the value change failed. This method makes it useful to coordinate multiple threads without locking them up. Here is the visitor program again, this time using an `AtomicInteger` as coordination primitive:

---

7   With plain variables, thread safety is not guaranteed. For example, with a plain variable a simple assignment may need multiple processor instructions to complete which opens it up to data corruptions if manipulated by multiple threads simultaneously.

Main program:

```
package ppm_java._dev.concept.example.multithread.waitfree;

import java.util.concurrent.atomic.AtomicInteger;

public class THouse
{
    private static final int        gkNumVisitors  = 5;
    private static final int        gkTimeAudience = 1000;
    private static final int        gkLocked       = 1;
    private static final int        gkUnlocked     = 0;

    public static void main (String[] args)
    {
        THouse              house;

        house = new THouse ();
        house.GetMeSomeVisitors ();
    }

    private AtomicInteger         fState;
    private TVisitor[]           fVisitors;

    public THouse ()
    {
        int i;

        fState   = new AtomicInteger (gkUnlocked);
        fVisitors = new TVisitor [gkNumVisitors];
        for (i = 0; i < gkNumVisitors; i++)
        {
            fVisitors[i] = new TVisitor (this, i);
        }
    }

    public void GetMeSomeVisitors ()
    {
        int i;

        for (i = 0; i < gkNumVisitors; i++)
        {
            fVisitors[i].start ();
        }
    }

    public boolean Visit (int id)
    {
        boolean isSuccess;

        isSuccess = fState.compareAndSet (gkUnlocked, gkLocked);
        if (isSuccess)
        {
            try {Thread.sleep (gkTimeAudience);} catch (InterruptedException e) {}
            fState.getAndSet (gkUnlocked);          // atomic setting
        }

        return isSuccess;
    }
}
```

Visitor:

```
package ppm_java._dev.concept.example.multithread.waitfree;

class TVisitor extends Thread
{
    private THouse               fHouse;
    private int                  fID;
    private String               fPreamble;

    public TVisitor (THouse house, int id)
    {
        fHouse        = house;
        fID           = id;
        fPreamble     = "Visitor: " + fID;
    }

    public void run ()
    {
        long            t0;
```

```
        long            t1;
        long            dT;
        long            tTot;
        boolean         isSuccess;

        tTot = 0;
        do
        {
            t0          = System.currentTimeMillis ();
            isSuccess = fHouse.Visit (fID);     // Costs next to no time if fHouse is busy, i.e. no suspending!
            t1          = System.currentTimeMillis ();
            dT          = t1 - t0;
            tTot        += dT;
            if (isSuccess)
            {
                System.out.println (fPreamble + ": Had an audience! Time spent (total): " + tTot + "ms.");
            }
            else
            {
                // Couldn't get an audience this time. At least, my thread is not
                // suspended. I'll go and do something else and try again later.
                try {Thread.sleep (500);} catch (InterruptedException e) {}
            }
        }
        while (! isSuccess);
    }
}
```

The output shows that there's a lot less waiting than when using **`synchronized`**. Now, each visitor spends zero seconds wait time[8]:
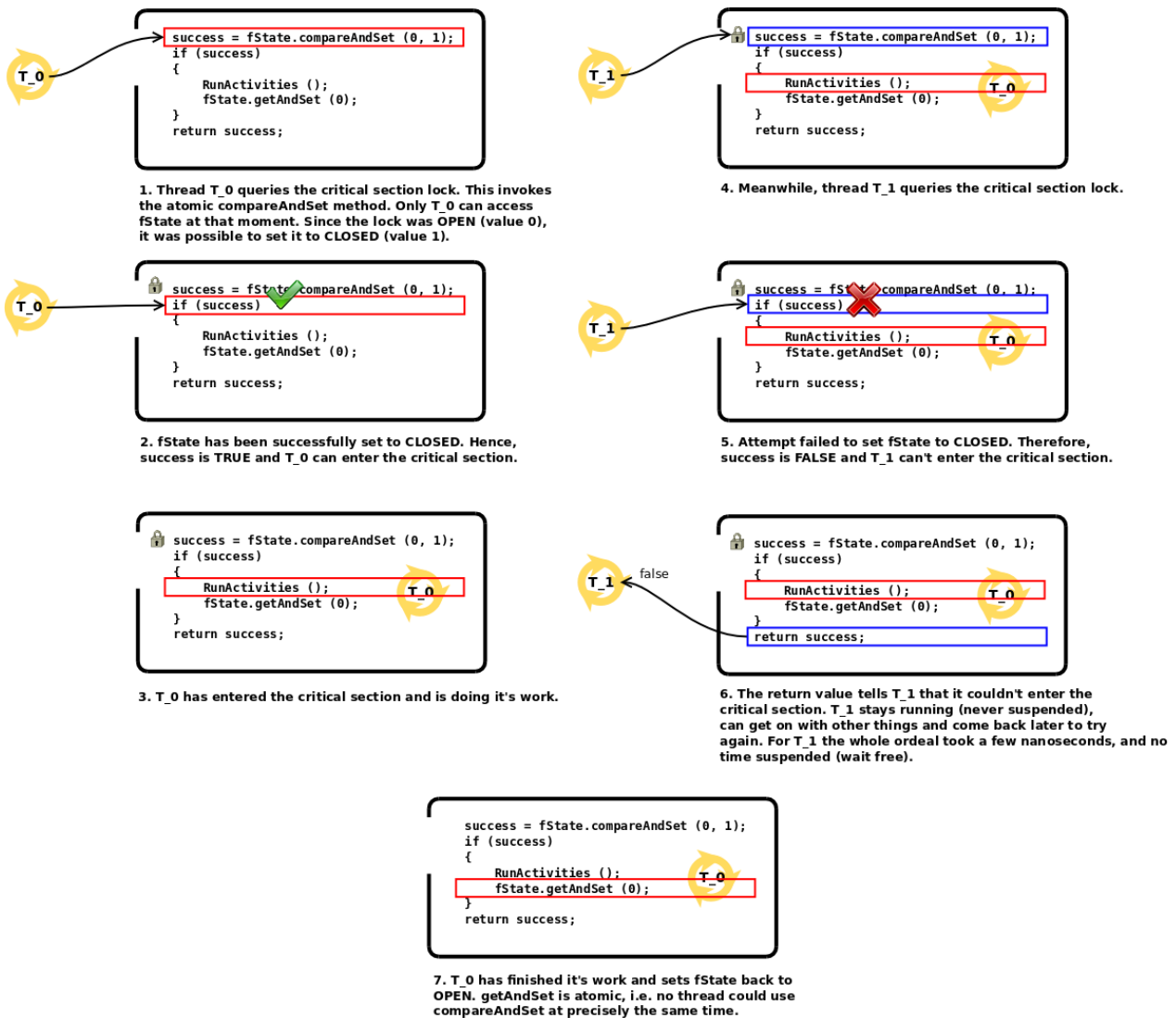
```
Visitor: 1: Had an audience! Time spent (total): 1000ms.
Visitor: 0: Had an audience! Time spent (total): 1000ms.
Visitor: 2: Had an audience! Time spent (total): 1000ms.
Visitor: 3: Had an audience! Time spent (total): 1001ms.
Visitor: 4: Had an audience! Time spent (total): 1000ms.
```

The advantage of this way of thread coordination is that no thread ever gets suspended. All threads will continue to work, also those which unsuccessfully tried to enter a critical section. This is mandatory for time critical threads (such as the one running the audio driver), as they will never freeze.

---

8   It may be confusing to you that it says "zero seconds of wait time" whilst the visitor thread does a **`Thread.sleep(500)`** when the **`isSuccess`** flag is **`false`**. Isn't **`Thread.sleep()`** a wait as well? In this context, it isn't! Because that sleep is not enforced from outside (via the **`synchronized`** mechanism), but chosen by the thread! Instead of sleeping the visitor could have done some other tasks, e.g. going for a walk, doing some shopping,... in the wait-free scenario it's up to the visitor thread what it does whilst waiting for the audience. With **`synchronized`** the sleep is enforced upon the visitor, i.e. the receptionist knocks him out.

```
success = fState.compareAndSet (0, 1);
if (success)
{
    RunActivities ();
    fState.getAndSet (0);
}
return success;
```

1. Thread T_0 queries the critical section lock. This invokes the atomic compareAndSet method. Only T_0 can access fState at that moment. Since the lock was OPEN (value 0), it was possible to set it to CLOSED (value 1).

```
success = fState.compareAndSet (0, 1);
if (success)
{
    RunActivities ();
    fState.getAndSet (0);
}
return success;
```

4. Meanwhile, thread T_1 queries the critical section lock.

```
success = fState.compareAndSet (0, 1);
if (success)
{
    RunActivities ();
    fState.getAndSet (0);
}
return success;
```

2. fState has been successfully set to CLOSED. Hence, success is TRUE and T_0 can enter the critical section.

```
success = fState.compareAndSet (0, 1);
if (success)
{
    RunActivities ();
    fState.getAndSet (0);
}
return success;
```

5. Attempt failed to set fState to CLOSED. Therefore, success is FALSE and T_1 can't enter the critical section.

```
success = fState.compareAndSet (0, 1);
if (success)
{
    RunActivities ();
    fState.getAndSet (0);
}
return success;
```

3. T_0 has entered the critical section and is doing it's work.

```
success = fState.compareAndSet (0, 1);
if (success)
{
    RunActivities ();
    fState.getAndSet (0);
}
return success;
```

false

6. The return value tells T_1 that it couldn't enter the critical section. T_1 stays running (never suspended), can get on with other things and come back later to try again. For T_1 the whole ordeal took a few nanoseconds, and no time suspended (wait free).

```
success = fState.compareAndSet (0, 1);
if (success)
{
    RunActivities ();
    fState.getAndSet (0);
}
return success;
```

7. T_0 has finished it's work and sets fState back to OPEN. getAndSet is atomic, i.e. no thread could use compareAndSet at precisely the same time.

Besides, some sources say that synchronization is much more expensive than atomic variables as the JVM does a lot of work in the background to operate it[9] whilst atomic variables are lightweight. Therefore, atomic variables incur less of a performance penalty than the synchronization primitive.

To confirm that this thread coordination primitive really works, here is the initial producer-consumer scenario revisited, this time without `synchronized` methods, but using an `AtomicInteger` as lock[10]:

---

9   https://www.ibm.com/developerworks/library/j-threads1
     http://baptiste-wicht.com/posts/2010/09/java-concurrency-atomic-variables.html
10   It's on purpose that we don't use the `AtomicInteger` for the `fValue` field as the intend is to show that an `AtomicInteger` can be used as a lock to make the program thread-safe.

Main program:

```
package ppm_java._dev.concept.example.multithread.waitfree_testsafe;

import java.util.concurrent.atomic.AtomicInteger;

public class TDev_Example_multithread_waitfree_safe
{
    private static final int              gkLocked       = 1;
    private static final int              gkUnlocked     = 0;

    public static void main (String[] args)
    {
        TDev_Example_multithread_waitfree_safe          unsafeClient;

        unsafeClient = new TDev_Example_multithread_waitfree_safe ();
        unsafeClient.start ();
    }

    private AtomicInteger               fState;
    private TThreadConsumer             fConsumer;
    private TThreadProducer             fProducer;
    private int                         fValue;

    public TDev_Example_multithread_waitfree_safe ()
    {
        fConsumer   = new TThreadConsumer (this);
        fProducer   = new TThreadProducer (this);
        fValue      = 0;
        fState      = new AtomicInteger (gkUnlocked);
    }

    public void start ()
    {
        fConsumer.start ();
        fProducer.start ();
    }

    public boolean Pop ()
    {
        boolean success;

        success = fState.compareAndSet (gkUnlocked, gkLocked);
        if (success)
        {
            System.out.println ("Entering     Pop  ().  Number: " + fValue);
            if (fValue > 0)
            {
                fValue--;
                try {Thread.sleep (500);} catch (InterruptedException e) {}
            }
            System.out.println ("    Exiting  Pop ().  Number: " + fValue);
            fState.getAndSet (gkUnlocked);
        }

        return success;
    }

    public boolean Push ()
    {
        boolean success;

        success = fState.compareAndSet (gkUnlocked, gkLocked);
        if (success)
        {
            System.out.println ("Entering     Push ().  Number: " + fValue);
            if (fValue < 1)
            {
                fValue++;
                try {Thread.sleep (500);} catch (InterruptedException e) {}
            }
            System.out.println ("    Exiting  Push ().  Number: " + fValue);
            fState.getAndSet (gkUnlocked);
        }

        return success;
    }
}
```

Producer:

```
package ppm_java._dev.concept.example.multithread.waitfree_testsafe;

/**
 * @author peter
 */
class TThreadProducer extends Thread
{
    private TDev_Example_multithread_waitfree_safe            fHost;

    /**
     *
     */
    public TThreadProducer (TDev_Example_multithread_waitfree_safe host)
    {
        fHost = host;
    }

    @Override
    public void run ()
    {
        int         i;
        int         delay;
        boolean     success;

        delay = 0;
        for (i = 1; i <= 10; i++)
        {
            delay += 10;
            do
            {
                success = fHost.Push ();
                try {Thread.sleep (delay);} catch (InterruptedException e) {}
            } while (! success);
        }

        System.out.println ("Producer finished!");
    }
}
```

Consumer:

```
package ppm_java._dev.concept.example.multithread.waitfree_testsafe;

/**
 * @author peter
 */
class TThreadConsumer extends Thread
{
    private TDev_Example_multithread_waitfree_safe            fHost;

    /**
     *
     */
    public TThreadConsumer (TDev_Example_multithread_waitfree_safe host)
    {
        fHost = host;
    }

    @Override
    public void run ()
    {
        int         i;
        int         delay;
        boolean     success;

        delay = 200;
        for (i = 1; i <= 10; i++)
        {
            delay -= 10;
            do
            {
                success = fHost.Pop ();
                try {Thread.sleep (delay);} catch (InterruptedException e) {}
            } while (! success);
        }

        System.out.println ("Consumer finished!");
    }
}
```

The program output shows two well behaved threads:

```
Entering     Pop  ().  Number: 0
    Exiting  Pop  ().  Number: 0
Entering     Push ().  Number: 0
    Exiting  Push ().  Number: 1
Entering     Push ().  Number: 1
    Exiting  Push ().  Number: 1
Entering     Push ().  Number: 1
    Exiting  Push ().  Number: 1
Entering     Pop  ().  Number: 1
    Exiting  Pop  ().  Number: 0
Entering     Push ().  Number: 0
    Exiting  Push ().  Number: 1
Entering     Pop  ().  Number: 1
    Exiting  Pop  ().  Number: 0
Entering     Push ().  Number: 0
    Exiting  Push ().  Number: 1
Entering     Push ().  Number: 1
    Exiting  Push ().  Number: 1
Entering     Push ().  Number: 1
    Exiting  Push ().  Number: 1
Entering     Pop  ().  Number: 1
    Exiting  Pop  ().  Number: 0
Entering     Push ().  Number: 0
    Exiting  Push ().  Number: 1
Entering     Pop  ().  Number: 1
    Exiting  Pop  ().  Number: 0
Entering     Push ().  Number: 0
    Exiting  Push ().  Number: 1
Entering     Pop  ().  Number: 1
    Exiting  Pop  ().  Number: 0
Entering     Push ().  Number: 0
    Exiting  Push ().  Number: 1
Entering     Pop  ().  Number: 1
Producer finished!
    Exiting  Pop  ().  Number: 0
Entering     Pop  ().  Number: 0
    Exiting  Pop  ().  Number: 0
Entering     Pop  ().  Number: 0
    Exiting  Pop  ().  Number: 0
Entering     Pop  ().  Number: 0
    Exiting  Pop  ().  Number: 0
Consumer finished!
```

In Summary: How to replace **`synchronized`** with a functionally equivalent wait-free lock[11]:

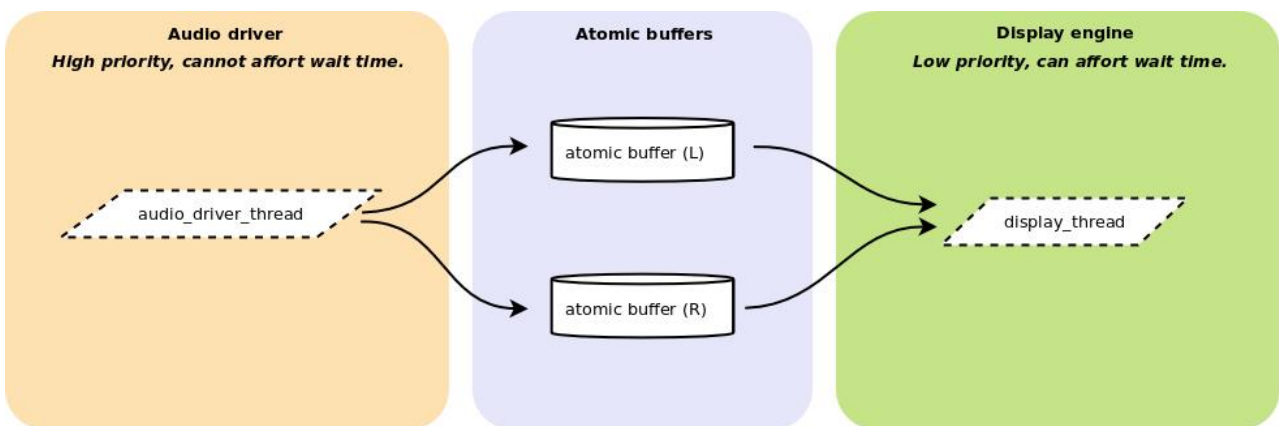| Synchronized	Wait-free lock
Threadsafe part:	Threadsafe part:

<pre>public class X
{
    public X ()
    {
    }
 	
    public synchronized void Do_Threadsafe_thing ()
    {
        /* The payload */
        Do_Work ();
    }
}
0.30 cm</pre>
	<pre>import java.util.concurrent.atomic.AtomicInteger;
	
	public class X
	{
	    private AtomicInteger fState;
	
	    public X ()
	    {
	        fState = new AtomicInteger (0);
	    }
	
	    public boolean Do_Threadsafe_thing ()
	    {
	        boolean success;
	
	        /* Lockdown if unlocked */
	        success = fState.compareAndSet (0, 1);
	
	        if (success)
	        {
	            /* The payload */
	            Do_Work ();
	
	            /* Unlock critical section */</pre>

---

11   You'll notice that the wait-free code is longer, but that's the price for the new wait-freedom!

```
                fState.getAndSet (0);
        }

        /* Tell client thread whether work done */
        return success;
    }
}
```

Thread using the threadsafe method:	Thread using the threadsafe method:
| ```
public class Y extends Thread
{
    private X      fSpecimen;

    public void run ()
    {
        fSpeciment.Do_Threadsafe_Thing ();
    }
}
``` | ```
public class Y extends Thread
{
    private X            fSpecimen;
    public void run ()
    {
    boolean success;
    do
    {
        success = fSpecimen.Do_Threadsafe_Thing();
        Do_other_Stuff_whilst_X_is_locked();
        try
        {    /* Thread puts itself to sleep
                so that we don't get a
                runaway thread freezing the
                machine. */
            Thread.sleep (yieldTime);
        }
        catch (InterruptedException e)
        {
        }
    } while (! success);
    }
}
``` |

5 Special constructs

Atomic buffer

`ppm-java` connects a high priority thread (the audio driver) with a low priority thread (the display engine). The high priority driver is integrated with a realtime application (`jackd`) and must not be slowed down. The low priority thread operates a display, and can afford the occasional slowdown. We have a constant flow of audio data from the audio driver to the display engine. We need a connection that passes the data on to the display engine without slowing down the audio driver. The connecting piece is a bank of atomic buffers (`ppm_java.util.storage.TAtomicBuffer`), one buffer per audio channel.

The atomic buffer is designed for thread safety and data safety: The audio driver and the display engine won't freeze each other (no priority inversion) and data is deep copied before it reaches the consumer thread.

Thread safety

The buffer uses the mechanisms described in chapter 4 (Multithreading) to provide thread coordination between the producer (audio driver) and the consumer (display engine). The thread coordination is wait-free for the high priority thread, and classically locking for the low priority thread (using a spin lock)[12].

---

12 The spin locking shouldn't be a problem, because when the high priority thread acquires the critical section it spends very little time in it. It does very little work there, basically another method call followed by an if-statement and an assignment.

Data safety

Due to the inner workings of the `JJack` library the audio frames are delivered as objects of type `java.nio.FloatBuffer`. If we passed these objects directly on to the consumer then any change done on the consumer side would be immediately mirrored on the producer side with potential side effects right up to the running `jackd` server[13]. That's why we perform a deep copy of the incoming `FloatBuffer` objects before passing them on to the consumer. This way, the consumer gets a copy and can change the data at will as it won't affect the producer.

As a downside, this introduces a performance penalty, as copying costs time. To protect the high priority thread from this penalty we use a copy policy which determines when the data will be copied[14]. This enables us to shift the extra copy work to the lower priority thread[15]. We have three possible copy policies (but the *third one is very risky* and should be used very sparingly!):

- **`kCopyOnSet`**   Atomic buffer will copy when the *producer* sets the data. Use this policy when the producer is the low priority thread. This is the setting used with the default constructor.

- **`kCopyOnGet`**   Atomic buffer will copy when the *consumer* fetches (gets) the data. Use this policy when the consumer is the low priority thread.

- **`kNoCopy`**   Data won't be copied, but we pass on the original **`FloatBuffer`** objects.
  This policy is necessary in rare situations to prevent unnecessary multiple copying of data frames (e.g. when data is passed from one module to another). **`kNoCopy` should be used very sparingly and carefully as it's a very risky option!**

Contention

The atomic buffer introduces a level of separation between the producer and consumer. For `ppm-java`, this means the audio driver and the display engine are less tightly integrated. Whilst it makes both sides more independent of each other it brings with it some level of contention which manifests itself as buffer overuns, buffer underruns and dropped data.

In `ppm-java` contention issues arise because the audio driver delivers sample chunks at one frequency and the display engine updates the display at another frequency[16]. As result, both sides run with different schedules which will inevitably lead to conflicts.

- For the audio driver, the update frequency depends on the sample rate and the size[17] of audio frames delivered. For example, if the sample rate is 44100 samples/sec and the frame size is 1024 samples per frame `jackd` will deliver a new frame every 1024 / 44100 s ≈ 20 ms.

- For the display engine, the update frequency is initially 30 display updates per second[18].

During a `ppm-java` session we encounter three types of contention issues, each of which are

---

13 Might not be as dramatic, but it's safest to make the worst-case assumption.
14 Policy set via parameter `copyPolicy` to the `TAtomicBuffer` constructor.
15 This brings it close to the "copy-on-write" technique.
16 It would be bad design to fix-bind the GUI update frequency to the Jack update frequency. Saying that - we do employ a compensation scheme where we change the GUI update frequency until buffer underruns and Buffer overruns are minimized.
17 Frame size depends on the running jackd instance, and that depends on how jackd was started (jackd commandline switch -p)
18 Tthis will be modified by the contention compensation

indicated by an appropriate counter increment:

- Buffer overrun. This happens when the producer (audio driver) pushes data onto the buffer whilst uncollected stored data is still loitering in the buffer. If that happens the atomic buffer will increment an overrun counter.

- Buffer underrun. This happens when the consumer tries to collect data from the buffer whilst it's still empty, i.e. the producer hasn't pushed any data onto the buffer yet. If that happens the atomic buffer will increment an underrun counter.

- Thread contention. This happens when both, producer and consumer, try to access the critical data at the same time. If that happens the atomic buffer will increment a contention counter. In practice, this should be rare - but if it's not then it means that both, producer and consumer, are too aggressively competing over the atomic buffer.

All of these three contention types are revealed through the three counters which are exposed to clients through **ppm-java**'s, statistics API. Clients can use the resp. counter values to mitigate over/underruns. Inside the buffer we use the following strategy to resolve contentions[19]:

- Overrun.

    1. Let producer set the new data, discarding stale data[20].

    2. Increment overrun counter.

- Underrun.

    1. Return new empty **FloatBuffer** or **null**[21] to the consumer.

    2. Increment underrun counter.

- Thread contention.

    1. Skip any data setting or getting, i.e.

        - For the producer, drop the new data.

        - For the consumer, return new empty **FloatBuffer** or **null**[22].

    2. Increment thread contention counter.

We haven't mentioned another policy: The **ifInvalidPolicy**[23]. This policy determines what will be returned to the consumer in case of an underrun or thread contention. This policy is there purely for the benefit of the consumer. The following policies are possible:

- **kReturnNull**. In case of contention, return **null** to the consumer. This makes consumer code a bit easier, as it's simpler to query for for null than to query whether a **FloatBuffer** is empty.

- **kReturnEmpty**. In case of contention, return an empty **FloatBuffer** to the consumer.

---

19 That's just the immediate dealing with a contention on the TAtomicBuffer side - the client has to still do additional steps to reduce contentions over time.
20 New data always takes priority over old data. Therefore we simply drop any uncollected data.
21 Depending on the ifInvalidPolicy.
22 ditto
23 Set via parameter ifInvalidPolicy to the TAtomicBuffer constructor.

There are situations where it's better to return a **FloatBuffer** to a consumer at all times, but never **null**, e.g. if a returned **null** causes a **NullpointerException**. That's why we provide the opportunity to return an empty **FloatBuffer**.

The **TAtomicBuffer** class could be changed to take other data items (e.g. arrays, **ArrayList**s, ...). It could be turned into a generic class, but this will need some thought as to the API of the stored objects - any stored object would need to support deep copying.

JJack integration

We already use a dedicated library[24] to connect to the running **jackd** instance. At first glance it seems overkill to provide a wrapper class around that library. However and in general, it's better to contain an external API in one place than having it spill over into multiple places in this project. This makes future developments easier as it's straight forward to accomodate changes to the library's API or switch to a different library entirely[25].

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):

---

24 `JJack`
25 `JJack` has come into its years. Currently, the last update on Sourceforge was 2014, and the latest version was uploaded in 2007. It still works, but for how long? Good enough for a concept application, though.

```
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 + ".");
    }
}
```
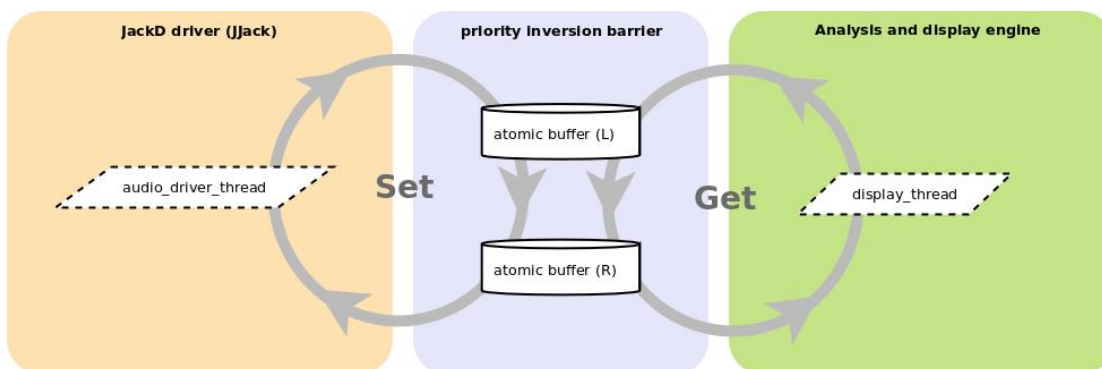
The central piece of this code is the method **process (JJackAudioEvent e)**. The method receives one parameter of type **JJackAudioEvent** (provided by **JJack**) which contains everything needed to extract and push audio data from/to the Jack server. The method does all the client's sample processing. The Jack server (via the **JJack** bridge) calls this method in an endless cycle (several times per second) until the program ends. It's called as part of a thread running within **jackd**. This makes the **process** method a part of the running **jackd** instance. It also means that the execution of this method 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.
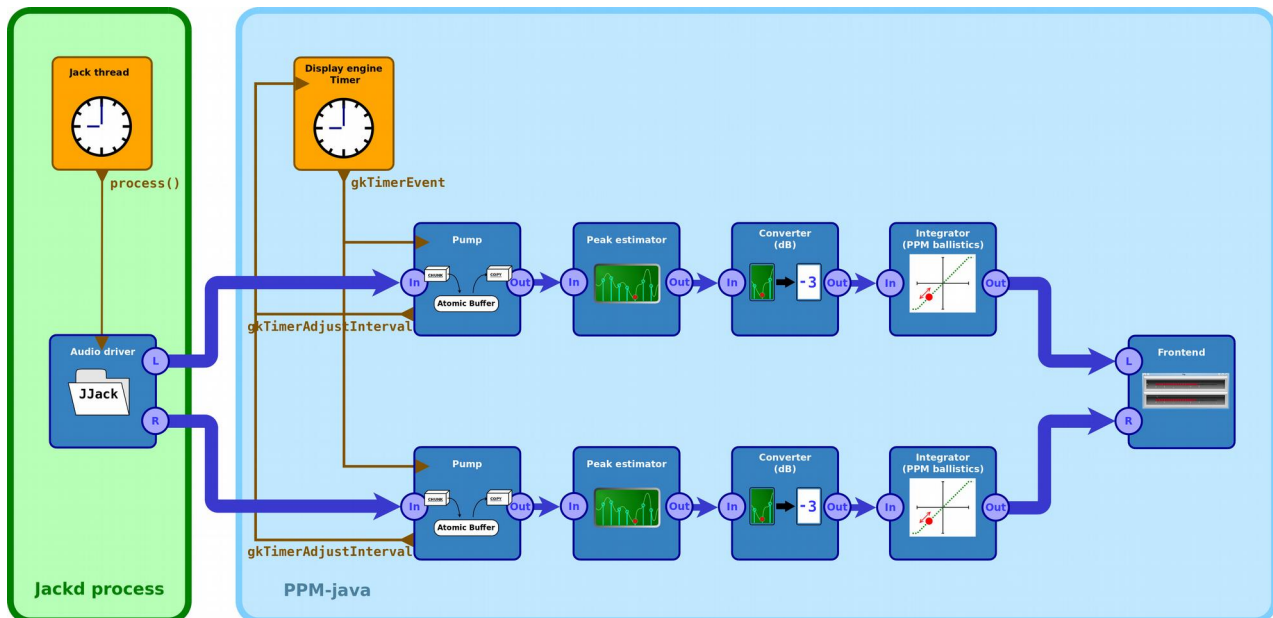
Here's the program output:

```
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 , with unknown side effects for the running `jackd` instance (priority inversion). To prevent 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.
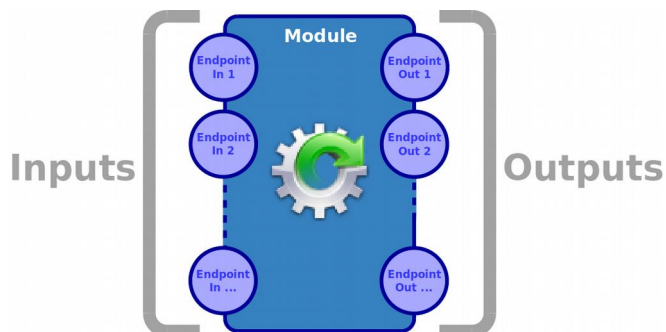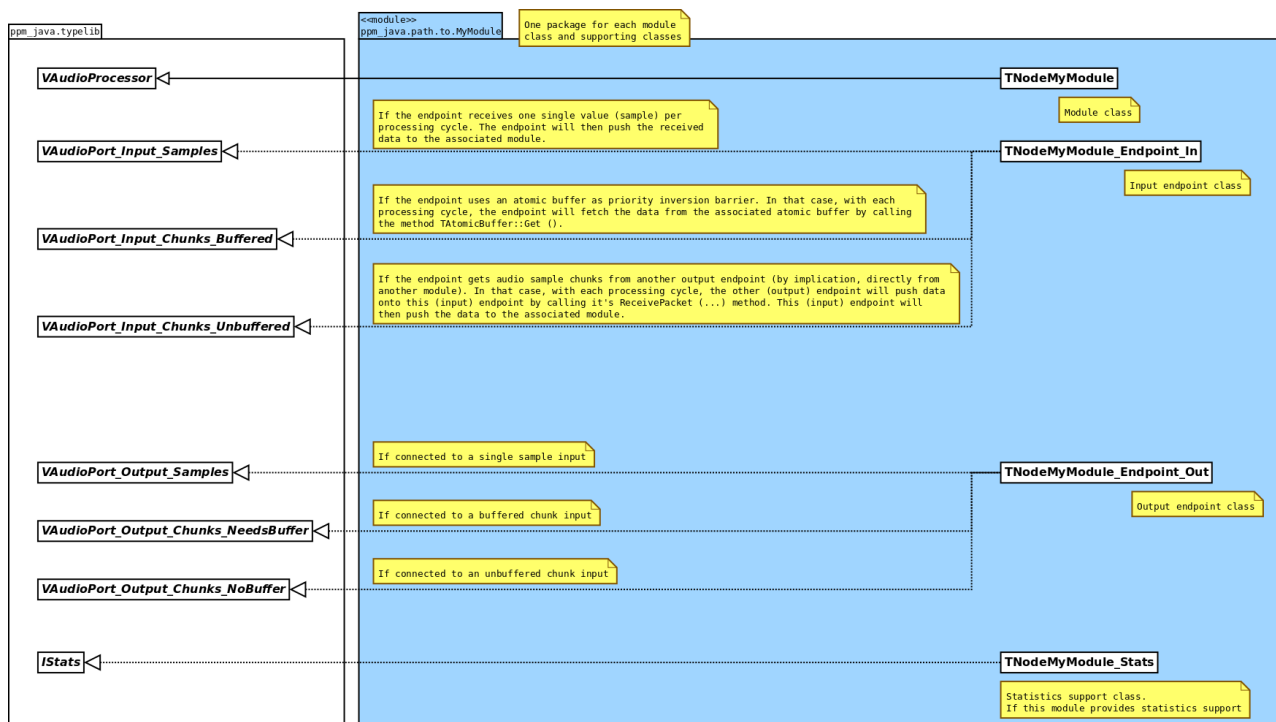
6 Modules

This program uses a modular machinery. Various processor modules work together, each one performing a very specific sub task. All together they form a signal processing network. Network topology is set up at the start of the program and persists until it terminates.

Modules are connected to each other via endpoints. A module can have an arbitrary number of endpoints.

ppm-java supports two categories of endpoints - those that handle individual sample values and those that handle sample chunks. Endpoints can only connect to matching endpoints, e.g. an output that sends samples must connect to an input which receives samples.

ppm_java.typelib

<<module>>
ppm_java.path.to.MyModule

One package for each module class and supporting classes

VAudioProcessor

TNodeMyModule

Module class

If the endpoint receives one single value (sample) per processing cycle. The endpoint will then push the received data to the associated module.

VAudioPort_Input_Samples

TNodeMyModule_Endpoint_In

Input endpoint class

If the endpoint uses an atomic buffer as priority inversion barrier. In that case, with each processing cycle, the endpoint will fetch the data from the associated atomic buffer by calling the method TAtomicBuffer::Get ().

VAudioPort_Input_Chunks_Buffered

If the endpoint gets audio sample chunks from another output endpoint (by implication, directly from another module). In that case, with each processing cycle, the other (output) endpoint will push data onto this (input) endpoint by calling it's ReceivePacket (...) method. This (input) endpoint will then push the data to the associated module.

VAudioPort_Input_Chunks_Unbuffered

If connected to a single sample input

VAudioPort_Output_Samples

TNodeMyModule_Endpoint_Out

Output endpoint class

If connected to a buffered chunk input

VAudioPort_Output_Chunks_NeedsBuffer

If connected to an unbuffered chunk input

VAudioPort_Output_Chunks_NoBuffer

IStats

TNodeMyModule_Stats

Statistics support class.
If this module provides statistics support

Modules and endpoints are implemented as bespoke classes. They extend abstract base classes in the package `ppm-java.typelib`.

- A module class extends the base class `VAudioProcessor`.

- An output endpoint extends one of the classes `VAudioPort_Output_Samples`, `VAudioPort_Output_Chunks_NeedsBuffer` or `VAudioPort_Output_Chunks_NoBuffer`.

- An input endpoint extends one of the classes `VAudioPort_Input_Samples`, `VAudioPort_Input_Chunks_Buffered` or `VAudioPort_Input_Chunks_Unbuffered`.

Input and output endpoints must be connected to their matching counterpart.

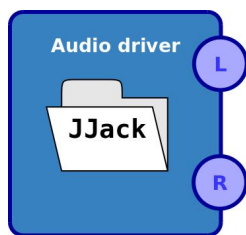| Output endpoint type	Connects to input point type
`VAudioPort_Output_Samples`	`VAudioPort_Input_Samples`
`VAudioPort_Output_Chunks_NeedsBuffer`	`VAudioPort_Input_Chunks_Buffered`
`VAudioPort_Output_Chunks_NoBuffer`	`VAudioPort_Input_Chunks_Unbuffered`

Modules can be designed in many ways (the framework offers a lot of flexibility). However, to avoid chaos, this project keeps to the following practice:

- One package for each module class and supporting classes, at least the classes for the module
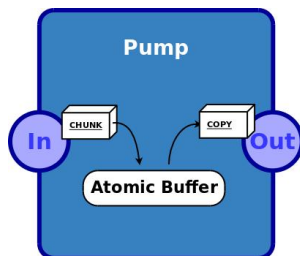
and its endpoints. If the module supports the statistics API, then those supporting classes will be in there, too.

- The module's package contains one class for the module, named `TNode{module_name}`. Module classes derive from class `ppm_java.typelib.VAudioProcessor`.

- The module's package contains one class for the input endpoints, named `{module_class_name}_Endpoint_In`. Data between modules is always sent from output endpoints to input endpoints (rather than directly from module to module).

- The module's package contains one class for the output endpoints, named `{module_class_name}_Endpoint_Out`. Data between modules is always sent from output endpoints to input endpoints (rather than directly from module to module).

- The module's package contains one class for the statistics retrieval, named `{module_class_name}_Stats`. This class provides runtime statistics of the associated module. The module class must expose the associated statistics object to outside callers.
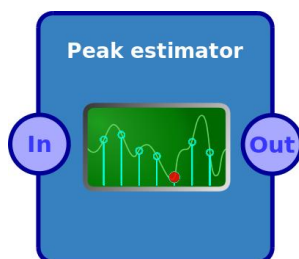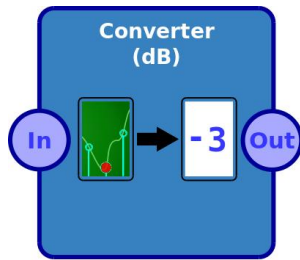
Module reference

Class: **TAudioContext_JackD**

Package: **ppm_java.backend.module.jackd**

Purpose: The audio driver (backend). Acts as bridge between `jackd` and `ppm-java`.

Class: **TNodePump**

Package: **ppm_java.backend.module.pump**

Purpose: Data pump. Fetches audio data from the attached atomic buffer and passes it on to the next module, each time it receives a `gkEventTimerTick`. Compensates for buffer underruns / overruns.

Class: **TNodePeakEstimator**

Package: **ppm_java.backend.module.peak_estimator**

Purpose: Calculates the absolute peak value of a sample chunk. Being absolute, the value is always positive.

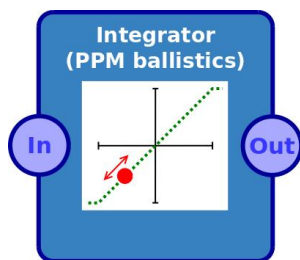| Class:	**TNodeConverterDb**
Package:	**ppm_java.backend.module.converter_db**
Purpose:	Converts a sample (peak) value from raw level to dB.

Conversion according to

$$y = 20 \log 10 (x)$$
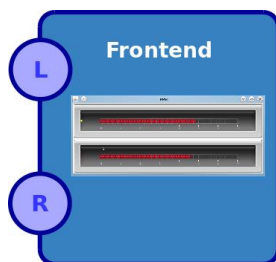
where

x: normalized absolute sample value [Vnorm],

$$0 \;<=\; x \;<=\; 1$$

y: sample value [dB]

Class:	**TNodeIntegrator_PPMBallistics**
Package:	**ppm_java.front.module.integrator**
Purpose:	A stepping integrator to emulate PPM ballistics.

Class:	**VFrontEnd** derivatives
Package:	**ppm_java.frontend.*.***
Purpose:	The frontend used for the current session.

7 Display engine

The main work of `ppm-java` is done inside the Display engine. This device takes the raw audio data and converts it to a stream of decibel values to be displayed on a front end or streamed via stdout to another application. With most frontends we include the PPM ballistics, streaming frontends omit PPM ballistics. As discussed elsewhere the analysis and display engine runs inside a low priority thread, but gets its input material from a high priority thread (audio driver) which requires extra measures to protect against priority inversion.

Data loss compensation

The priority inversion protection comes at the price of data integrity. If the display engine does block (e.g. graphics takes longer than usual to do a refresh) then it will miss some of the data values coming from the audio driver. There isn't really any other way to resolve freeze-ups of the display engine than to simply drop past data and carry on with the current data. However, the display engine compensates for data loss, so it's kept at a minimum over time. The compensation is done inside the data pump module which sends `gkTimerAdjustInterval` events to the timer module during each display engine cycle. This increases or decreases the engine update frequency until the number of buffer underruns / overruns is minimized. In detail,

- if we detect *underruns* then there's *too little* data coming in from the audio driver. We need less display updates so we don't starve the display engine. Therefore we will instruct the timer to increase the update interval.

- if we detect *overruns* then there's *too much* data coming in from the audio driver. We need more updates to use all the data. Therefore we will instruct the timer to decrease the update interval.

Tests show that this scheme does resolve data loss very quickly. It's a bit unfortunate that this makes the data pump module more complex, but overall it's the simplest design.
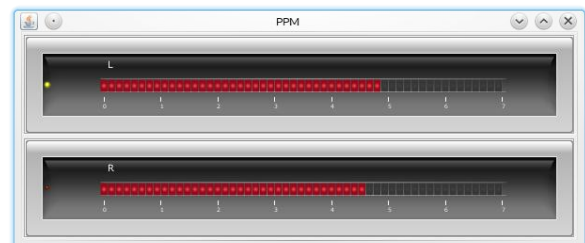
8 Frontends

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.
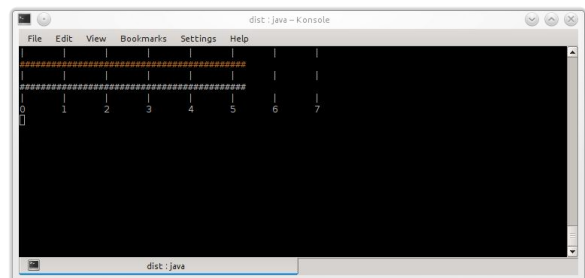
guiRadial   A PPM lookalike.

guiLinear   Horizontal linear bargraph.
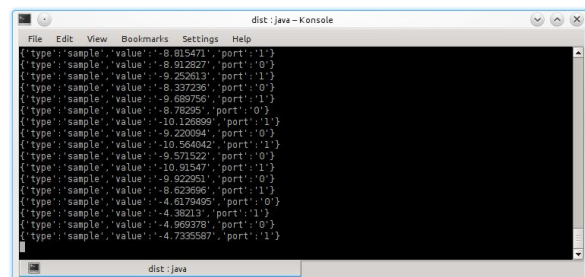

consoleLinear   Horizontal linear bargraph on the console (text mode).

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.

---

9 Session setup

ppm-java sets up the complete modular network right at the start of the program run. Network setup is done by the class **ppm_java.backend.boot.TSetup**. **TSetup** in turn uses the services of the class **ppm_java.backend.TController** which centralizes session setup and event routing. Everything pertaining to module creation and connection is done in the **TController** class, via a number of static methods. During the session each module can be addressed by a unique ID which we set during the setup phase. Modules, ports and frontends all inherit from the class **ppm_java.typelib.VAudioProcessor** and can hence be accessed by calling the method **TController.GetObject (String id)** with the unique ID of the object as parameter.

A network is built by

1. Creating modules,

2. Adding ports to each module,

3. Connecting ports for sample data,

4. Connecting modules to convey events.

Once built the network can be started.

Creating modules

Before we can do anything with modules we need to create them. **TController** offers the methods

*   **Create_Module_X** where **X** is the module to be created, e.g. for the **TTimer** module, we'd call the **Create_Module_Timer** method.

*   **Create_AudioContext** for the audio driver, **TAudioContext_JackD**.

Each method takes one parameter, the unique ID under which the module can be accessed during the session.

Adding ports

Once modules are created we can add ports to them. **TController** offers the methods

*   **Create_Port_In (String idModule, String idPort)**. Creates an *input* port on the module **idModule**. The port is named **idPort**.

*   **Create_Port_Out (String idModule, String idPort)**. Creates an *output* port on the module **idModule**. The port is named **idPort**.

Data connections

With all modules and ports created we can connect ports together, so that samples can flow (that is, sample chunks and single samples). **TController** offers the method

Create_Connection_Data (String idFromPort, String idToPort)

This method creates a unidirectional audio data connection with port **idFromPort** as source and port

---

`idToPort` as sink.

Note that ports must pair up correctly. You cannot connect a port which sends sample chunks to a port that receives single samples[26]. Such aberrations would throw an exception.

Event connections

Within the network we use an event system to trigger certain actions. For example, the event `gkEventTimerTick` is sent by the `TTimer` module to notify subscribers that one time period has passed. To subscribe modules to the event of another module, `TController` offers the method

`Create_Connection_Events (String idSource, String idRecipient)`

This method subscribes module `idRecipient` to events coming from module `idSource`. The recipient must provide event filtering as neccessary, for example if module `idSource` sends out multiple types of events.

Starting

Once everything has been created and connected we can start the network. This always happens in a predefined order - established by setting a few start entries using the method `TController.Create_StartListEntry (String id)` where `id` is the unique ID of the module to be started. Once all start entries are set we can start the network by calling `TController.start ()`. This will start the modules in the order in which we created the start list entries.

Stopping

When the application is finished we need to shut down the network by stopping the modules. As with the starting we need to stop modules in a predefined order, which we establish by setting entries in a stop list. A stop list entry is set by calling the method `TController.Create_StopListEntry (String id)` where `id` is the unique ID of the module to be stopped. Stopping happens automatically when the user presses CTRL-C or closes the GUI frontend.

---

26   This wouldn't make sense anyway

10 Events

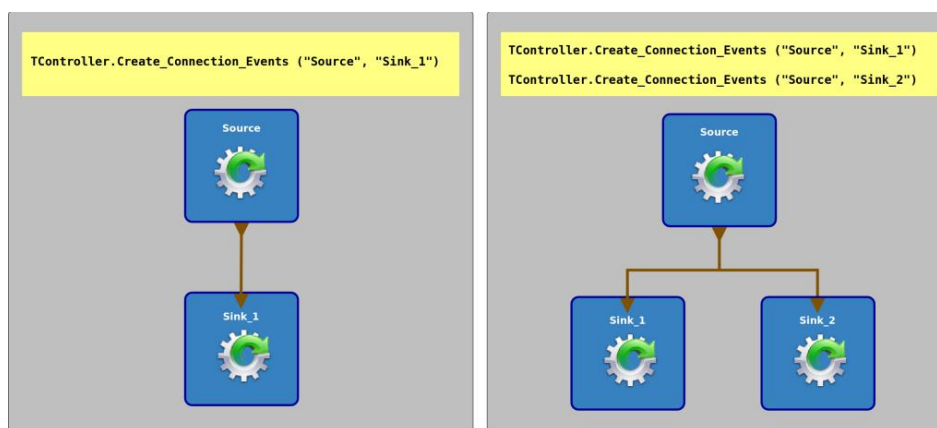As in any application there needs to be a measure of realtime control. `ppm-java` offers a simple event system which allows various modules to control aspects of the program whilst it's running.

Objects that handle events implement the interface `ppm_java.typelib.IEvented`. This interface provides the events used in the program as well as the method stubs to handle those (`OnEvent (...)`).

```
          ┌─────────────────────────────────┐
          │            IEvented             │
          ├─────────────────────────────────┤
          │ +gkEventStart: int = 1000       │
          │ +gkEventStop: int = 1010        │
          │ +...etc                         │
          ├─────────────────────────────────┤
          │ +OnEvent(e:int): void           │
          │ +OnEvent(e:int,arg0:int): void  │
          │ +OnEvent(e:int,arg0:long): void │
          │ +OnEvent(e:int,arg0:String): void│
          └─────────────────────────────────┘
```

```
┌──────────────────────┐        ┌──────────────────────────────────┐
│   TMyEventSender      │        │        TMyEventReceiver          │
│   A class sending     │        │          A class                 │
│       events          │        │         receiving                │
│                       │        │           events.                │
└──────────────────────┘        ├──────────────────────────────────┤
                                 │ +OnEvent(e:int): void            │
                                 │ +OnEvent(e:int,arg0:int): void   │
                                 │ +OnEvent(e:int,arg0:long): void  │
                                 │ +OnEvent(e:int,arg0:String): void│
                                 └──────────────────────────────────┘
```

Event routing must be set up at the start of a `ppm-java` run. This is done by calling `TController.Create_Connection_Events (String idSource, String idRecipient)` which subscribes the object `idRecipient` to all events from the object `idSource`. It's possible to have multiple recipients for one object, and that an object can receive events from multiple sources. Having an object subscribe to multiple event sources can present problems (e.g. an object subscribing to multiple `TTimer`s)[27].

When the session runs, the source object will send events by calling one of the

---

27   In that scenario there are ways to minimize problems, but the best way is to avoid this scenario if possible. For example, with the multiple timer scenario we might just use one timer sending gkEventTimerTick events at a high frequency which is a whole multiple of the frequency of all the TTimers we would have used and then to use frequency division and multiplexing in the object that receives the gkEventTimerTick events.

`TController.PostEvent` methods. The global `TController` object will pass this event on to all subscribers of the source object. This means that each subscribed recipients will receive *all* events emitted from the source object. It's up to each recipient to filter the events for those it's interested in.

Events are identified by plain vanilla integer constants. Normally we'd be using an enumerator or a base class for each event type. Both has disadvantages. An enumerator requires an extra import statement for each concrete implementation of `IEvented`, and an event class would need to be instantiated with every event fired. That's fine for the occasional event, but something like `TTimer` would generate many events in a short time, resulting in the creation of lots and lots of throw away event objects. And this would smell like a very silly design. Using plain-vanilla integers seemed the most economic idea, as it doesn't need any imports and does away with that crazy fabrication of throw away objects.

The `OnEvent` event handler is overloaded to receive one in-band parameter - as an `int`, a `long` or as a `String`. This is a bit cumbersome as it means that every implementation of `IEvented` needs to provide all overloaded versions of `OnEvent`[28], but the higher performance and possibility to provide in-band data is a good reason to go a bit more complex.

Here is a basic example of a network consisting of one event source connected to a number of receivers, each one writing to `stdout` on each event.

---

28   IDE's such as Eclipse insert the resp. method stubs automatically. Text editor users have to bite the bullet, I'm afraid...

TDev_Example_Event

```
package ppm_java._dev.concept.example.event;

import ppm_java.backend.TController;
import ppm_java.util.logging.TLogger;

public class TDev_Example_Event
{
    public static void main (String[] args)
    {
        TLogger.CreateInstance ();        /* Otherwise NullPointerException */

        new TSource ("Source");
        new TSink    ("Sink_1");
        new TSink    ("Sink_2");

        TController.Create_Connection_Events ("Source", "Sink_1");
        TController.Create_Connection_Events ("Source", "Sink_2");

        TController.Create_StartListEntry ("Source");

        TController.Start ();
    }
}
```

TSource

```
package ppm_java._dev.concept.example.event;

import ppm_java.backend.TController;
import ppm_java.typelib.IControllable;
import ppm_java.typelib.IEvented;
import ppm_java.typelib.VBrowseable;

class TSource extends VBrowseable implements IEvented, IControllable
{
    public TSource (String id)
    {
        super (id);
    }

    /**
     * Sends ten events (zeros) to any subscribers.
     */
    public void Start ()
    {
        String  id;
        String  msg;
        int     i;

        id = GetID ();
        for (i = 1; i <= 10; i++)
        {
            msg = "Event #" + i;
            TController.PostEvent (0, msg, id);
            try {Thread.sleep (1000);} catch (InterruptedException e) {}
        }
    }

    public void Stop ()                        {/* Do nothing. */}
    public void OnEvent (int e)                {/* Do nothing; this is just an event source */}
    public void OnEvent (int e, int arg0)      {/* Do nothing; this is just an event source */}
    public void OnEvent (int e, long arg0)     {/* Do nothing; this is just an event source */}
    public void OnEvent (int e, String arg0)   {/* Do nothing; this is just an event source */}
    protected void _Register ()
    {
        TController.Register (this);
    }
}
```

TSink

```
package ppm_java._dev.concept.example.event;

import ppm_java.backend.TController;
import ppm_java.typelib.IControllable;
import ppm_java.typelib.IEvented;
import ppm_java.typelib.VBrowseable;

class TSink extends VBrowseable implements IEvented, IControllable
{
    public TSink (String id)
    {
        super (id);
    }

    public void OnEvent (int e, String arg0)
    {
        String msg;

        msg = GetID () + ": " + "Received messaging event. Message: " + arg0;
        System.out.println (msg);
    }

    public void Start ()                        {/* Do nothing */}
    public void Stop ()                         {/* Do nothing */}
    public void OnEvent (int e)                 {/* Do nothing */}
    public void OnEvent (int e, int arg0)       {/* Do nothing */}
    public void OnEvent (int e, long arg0)      {/* Do nothing */}
    protected void _Register ()
    {
        TController.Register (this);
    }
}
```
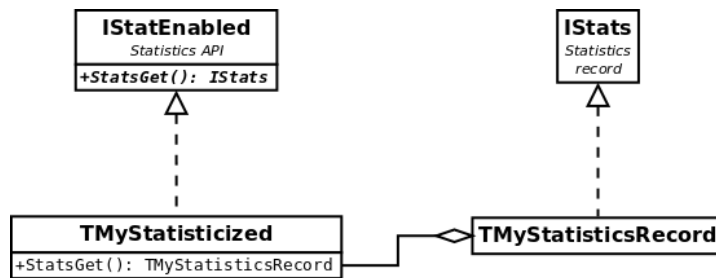
Here is the program output (stdout):

```
Sink_1: Received messaging event. Message: Event #1
Sink_2: Received messaging event. Message: Event #1
Sink_1: Received messaging event. Message: Event #2
Sink_2: Received messaging event. Message: Event #2
Sink_1: Received messaging event. Message: Event #3
Sink_2: Received messaging event. Message: Event #3
Sink_1: Received messaging event. Message: Event #4
Sink_2: Received messaging event. Message: Event #4
Sink_1: Received messaging event. Message: Event #5
Sink_2: Received messaging event. Message: Event #5
Sink_1: Received messaging event. Message: Event #6
Sink_2: Received messaging event. Message: Event #6
Sink_1: Received messaging event. Message: Event #7
Sink_2: Received messaging event. Message: Event #7
Sink_1: Received messaging event. Message: Event #8
Sink_2: Received messaging event. Message: Event #8
Sink_1: Received messaging event. Message: Event #9
Sink_2: Received messaging event. Message: Event #9
Sink_1: Received messaging event. Message: Event #10
Sink_2: Received messaging event. Message: Event #10
```

11 Statistics API

ppm-java offers a statistics API. Objects can be enabled to provide runtime statistics which interested parties can retrieve at any time. The runtime statistics API provides a unified way to do this.

Objects that provide statistics implement the `ppm_java.typelib.IStatEnabled` interface. This augments them with the method `StatsGet`. `StatsGet` returns a statistics record which contains the current runtime statistics of the `IStatEnabled` instance. The statistics record must be a class which implements the interface `ppm_java.typelib.IStats`. It is recommended to keep the statistics record class in the same package as the `IStatEnabled` instance.

Here is an example of a program which uses the statistics API:

{prg-example}

12 Class hierarchy

13 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[29] (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).

---

29 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`

14 Appendices

App. 1: Example listings

Listing 1:

```java
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

Critical section

In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed is protected. This protected section is the critical section or critical region. It cannot be executed by more than one process. Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that would not operate correctly in the context of multiple concurrent accesses.

https://en.wikipedia.org/wiki/Critical_section

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

15 License

A program to display audio levels using PPM ballistics.

Copyright (C) 2017 Peter Hoppe

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

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

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.