MPEG library software maintenance documentation

pakkaamo

Helsinki December 18, 2009

Software Engineering Project

UNIVERSITY OF HELSINKI
Department of Computer Science

Course

581260 Software Engineering Project (9 cr)

Project Group

Visa Hankala
Ville Kesola
Heikki Korhola
Kalervo Oikarinen
Sindi Poikelin
Tomi Ronimus

Client

Michael Przybilski, Taneli Vähäkangas

Project Masters

Juha Taina
Jari Suominen

Homepage

`http://cs.helsinki.fi/group/pakkaamo`

Change Log

Version	Date	Modifications
1.0	27.11.2009	First version

Contents

1 Introduction **1**

1.1 The project description 1

1.2 For Mp2 encoder developer 1

1.3 Legal issues 1

1.4 Document contents 2

1.5 User manual 2

2 MPEG encoder packages **3**

3 MPEG encoder classes **4**

4 The MPEG-1 Layer II encoder **5**

4.1 The audio encoding process 5

4.1.1 The analysis filter 6

4.1.2 Scalefactors 7

4.1.3 Bit allocation 7

4.1.4 Quantization 8

4.1.5 Bit-level encoding 8

4.2 The unit tests 8

4.2.1 The tests 8

4.3 Known issues 9

5 Description of the PA model **10**

5.1 Introduction 10

5.2 Implementation details 10

5.3 Overview of the unit tests 11

6 Concerning the multiplexer **12**

6.1 The idea 12

6.2 The classes 12

6.3 About the structure and workings 12

6.3.1 Stream abstraction 13

6.3.2 Packetization 13

6.3.3 Scheduling and multiplex generation 14
6.4 Further development 14
6.4.1 Adding new input types 14
6.4.2 Thread-safety needed 14
6.5 About the testing 15
6.5.1 What is tested 15
6.6 Known issues 15

7 Future development areas **17**

8 Implemented features **20**

References **22**

References **23**

1 Introduction

1.1 The project description

The Spring 2009 project Tensori created a prototype of a MPEG-2 video encoder library in Java. This documentation does not contain the video encoding process, only audio encoding and multiplexer implementation. If you are planning to create an audio/video encoder please see Tensori project documentation. The Autumn 2009 project pakkaamo further developed the prototype. Due to the use in DVDs, MPEG-2 is probably the most widely used encoding format for motion pictures. Typically MPEG-2 software encoders are implemented in C or C++ (e.g., ffmpeg, mpeg2enc) and heavily optimized on all levels to make the encoding process as fast as possible. Unfortunately this makes the code rather difficult to comprehend and hinders the adoption to other projects. The project would implement an MPEG-2 encoder purely in Java. The project will be facing a variety of very interesting challenges. Besides pure software development, these include a good understanding of mathematical algorithms and compression methods, such as discrete cosine transformation, but also typical project requirements, such as a basic understanding of the involved legal requirements. Altogether this project will give the students a comprehensive understanding of the subject of media coding, as well as challenges, typical for such a software development task. Specifically, the MPEG 2 Audio (MPEG 1- Layer II) would be of interest although layer III (MP3) and AC3 Audio might be possible as well.

1.2 For Mp2 encoder developer

This document contains the instructions how to implement and maintenance an MP2 encoder. We assume that you're familiar with MPEG specification. All package organization, class names and variable names are following the MPEG specification. Pakkaamo does not provide the MPEG specification on the distribution. Please get your legal and own copy from ISO organization.

1.3 Legal issues

MPEG technology contains several patent and license issues. If you consider to use this product on commercial purposes or distribute this product please check the project documentation called legal.pdf. MPEG LA offers several Portfolio Licenses which provide coverage under patents that are essential for use of various technology standards, including MPEG-2 Video and Systems (used together), MPEG-4 Visual (Part 2), AVC/H.264 (MPEG-4 Part 10), VC-1, ATSC, IEEE 1394, and MPEG-2 Systems (used separately from MPEG-2 Video). Under the Licenses, coverage is generally provided for end products and services that include these technologies. Accordingly, the party offering such end products and services (for example, under that party's own brand name) concludes

the appropriate License with MPEG LA and is responsible for paying the applicable royalties.

MPEG LA does not offer a license for MPEG-1 or any audio formats. With respect to MPEG-1, our only suggestion would be to contact the patent owners directly. Visiting the home page for the Moving Pictures Experts Group (MPEG) may get you pointed in the right direction (http://www.chiariglione.org/mpeg/).
Regarding audio licenses, our only suggestion would be to try contacting parties such as Fraunhofer Institute (http://www.iis.fhg.de/audio), Thomson Multimedia (http://www.thomson-multimedia.com), Philips licensing (http://www.philips.com), SISVEL (http://www.sisvel.com) and Audio MPEG (http://www.audiompeg.com). MPEG LA understand that Dolby (or its licensing company Via Licensing (http://www.vialicensing.org/) may provide licenses for various audio formats.

1.4 Document contents

Second chapter shows the package structure of the encoder. In the third chapter we take a look at the MPEG encoder classes. Fourth chapter goes through the encoding process on a higher level and then gives details on the testing. Then it presents some known issues with the encoder's implementation. Chapter five gives and overview on the psychoacoustic model, its implementation and testing. Chapter six shows the idea and structure of the multiplexer and gives information testing specific to the multiplexer. Information is also given about further development and known issues. Chapter seven shows the future development areas for the encoder library and chapter eight goes through the implemented items and offers information on the state of implementation and reliability.

1.5 User manual

If you want to use MP2 library add $PROJECT_HOME/lib/pakkaamo.jar, $PROJECT_HOME/lib/minim.jar, $PROJECT_HOME/lib/minim_spi.jar and $PROJECT_HOME/lib/core.jar to your build path. See the essential classes for audio encoding are MP2Encoder and AudioFrame.
See the examples and full descriptions from the JavaDoc.

2 MPEG encoder packages

Package name	Description
fi.helsinki.cs.ohtu.mpeg2	MPEG-2 systems
fi.helsinki.cs.ohtu.mpeg2.audio	MPEG-2 audio upper classes
fi.helsinki.cs.ohtu.mpeg2.audio.mpa	MPEG-1 audio encoder classes
fi.helsinki.cs.ohtu.mpeg2.util	Utility classes
fi.helsinki.cs.ohtu.mpeg2.video	MPEG-2 video encoding classes
examples	Examples on how to use the library

Figure 1: MPEG encoder packages

Figure 1 presents the package structure of Java MPEG-2 Encoder.

The packages fi.helsinki.cs.ohtu.mpeg2, fi.helsinki.cs.ohtu.mpeg2.video and fi.helsinki.cs.ohtu.mpeg2.util contain classes written by the previous group Tensori. Their classes were reorganized when the final package structure was made.

If a new encoder will be implemented, for example AC-3, it is suggested to locate in its own package fi.helsinki.cs.ohtu.mpeg2.audio.ac3.

3 MPEG encoder classes

Class name	Description
AudioEncoder	Encodes a single-channel or multi-channel frame of samples.
AudioFrame	Returns duration of frame in seconds and writes contents of frame to the given stream.
BitAllocator	Represents a bit allocation for subbands on multiple channels.
DummyPAModel	A dummy class for a psychoacoustic model.
MP2Data	This class encodes compressed audio data to binary form.
MP2Encoder	Implements an MPEG 2 encoder.
MP2Frame	Represents a coded MPEG-1 Layer II audio frame.
MP2Quantizer	Quantizes the subband samples.
MP123Header	Represents an MPEG-1 audio frame header as described in section 2.4.1.2 of ISO/IEC 11172-3.
PolyphaseQuadratureFilter	This class implements an analysis subband filter specified in ISO/IEC 11172-3. Maps samples to 32 subbands.
QuantizationTables	Quantization tables.
ScaleFactors	Scalefactors are taken from table 3-B.1 of ISO 11172-3.
StandardPAModel1	Implements the Psychoacoustic Model 1 described in Annex D of ISO 11172-3.

Figure 2: MPEG encoder classes

Figure 2 presents the most important classes implemented for MPEG-1 Layer II audio encoder. The most of the classes implement one step of the encoding process, for example BitAllocator implements the bit allocation phase. In addition, there are some general classes, like MP2Encoder, which extends AudioEncoder. This class wraps all the other classes in the encoder, and it is simple to use if you don't need any special settings.

Further descriptions and use case examples can be found in chapter 4, where the encoding process is explained.

4 The MPEG-1 Layer II encoder

The ISO/IEC 11172-3 standard document defines three layers, I, II and III of audio compression, each building upon and increasing in complexity over the preceding one. In this project, specifically a Layer II encoder conforming to the standard was implemented.

MPEG-1 audio compression is a lossy compression technique exploiting the weaknesses of the human auditory system. A psychoacoustic model is used to guide the compression process such that the essence of the signal is preserved while irrelevant portions are discarded.

The implementation supports monophonic, dual-monophonic and stereo encoding. The three sampling rates, 32, 44.1 and 48 kHz, defined in the MPEG-1 standard are supported. In addition, the encoder partially supports the MPEG-2 Lower Sampling Frequencies extensions: 16, 22.05 and 24 kHz. However, the current psychoacoustic model is restricted to MPEG-1 sampling rates only. A dummy model is used for the LSF rates.

Code example

```
MP2Encoder mp2Enc;
AudioFrame frame;

mp2Enc = new MP2Encoder(AudioEncoder.Mode.SINGLE_CHANNEL,
        AudioEncoder.SampleRate.SRATE_48000);

frame = mp2enc.encode(new double[mp2enc.getSampleFrameSize()]);
```

4.1 The audio encoding process

The implementation of the encoder follows the example in the Annex C of the ISO 11172-3 document. The example was chosen as the basis, since it was, afterall, relatively straightforward. Besides, it would have been very tedious to develop an equivalent encoder from scratch. The essential idea is to break the input signal into components that are then encoded in a (somewhat) smart way. The encoding consists of the following phases: Analysis filtering, the psychoacoustic model, scale factor calculation and selection, bit allocation, quantization and bit-level coding. The whole process is orchestrated by the `MP2Encoder` class which consists mainly of glue logic. An overview of the encoder and data flows within it are presented in Figure 3.

As an input, the encoder takes a frame of pulse-code modulated samples. The encoder outputs a coded frame. The frame size is fixed to precisely 1152 samples by the standard. In the encoding, the input samples are assumed to be in the open range $(-1, 1)$.

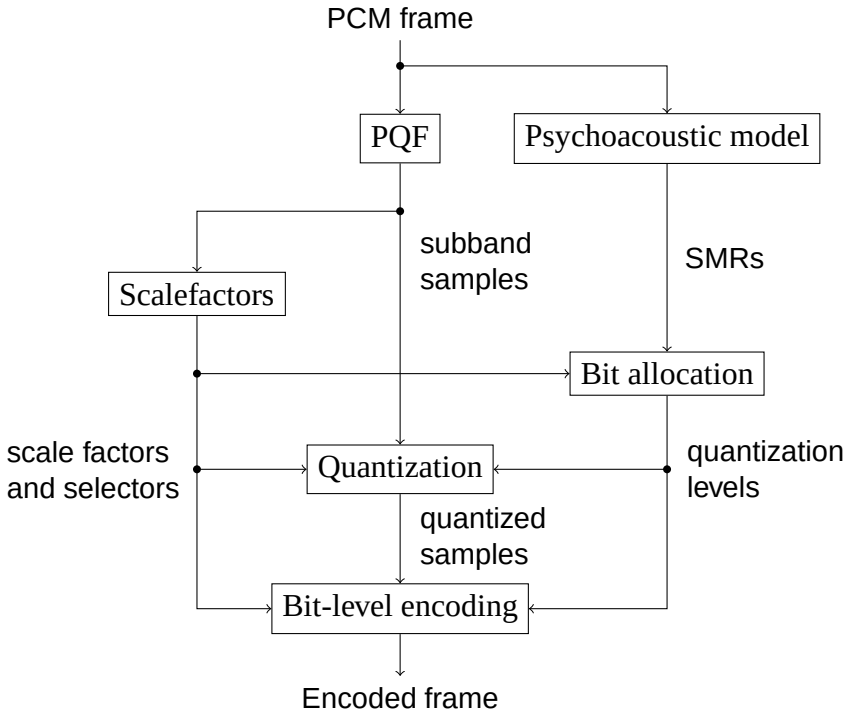

Figure 3: The general picture of the encoder

4.1.1 The analysis filter

The first phase of the encoding process is analysis filtering. The filtering is done by the class `PolyphaseQuadratureFilter` which implements a polyphase quadrature filter (PQF) described in Section 3-C.1.3 of the ISO/IEC 11172-3 document. The filter maps the input signal into 32 subbands.

Technical details

The sub-bands are evenly spaced in frequency: The middle frequency for a subband i (zero based) is $\frac{1}{64}(2\pi i + 1)$. Adjacent subbands are slightly overlapped. The filter has a delay of 256 samples due to internal windowing. The window size is 512 samples. As a consequence, the filter "remembers" a glimpse of the previous samples. In addition, there is a short "run-in phase" to fill the window at the start of filtering (initially it is full of zero).

Layer II coding groups are 3 of 12 samples per subband which is 1152 samples per frame. There can be up to 3 scale factors per subband to avoid audible distortion in special cases.

Code example

```
PolyphaseQuadratureFilter pqf = new PolyphaseQuadratureFilter();
```

```
double[] samples = new double[1152];
...
subbands = pqf.filter(samples, samples.length / SBLIMIT);
```

4.1.2 Scalefactors

The second encoding phase is scalefactor calculation and selection. The class `ScaleFactors` implements scalefactor calculations for both MPEG-1 Layer I and II audio. The scale factors are used in the encoding process to implement an application specific (read "home-made") floating-point representation of sub-band samples: The scalefactor indices are coded along the scaled (and quantized) subband samples. Thus the scaling can be reversed while decoding.

Code example

```
scfis = ScaleFactors.calcScale(samples);
...
scaledSample = ScaleFactors.scale(sample, scfi);
```

4.1.3 Bit allocation

The bit allocator is, more or less, the core of the encoder. The allocator determines how to fit given subbands into a coded audio frame. For every subband the allocator determines a quantization level. The implementation, the class `MP2Encoder` uses the class `BitAllocator` to do the actual allocation.

The essence of the allocator is as follows: Initially every sub-band sample is (almost) noiseless in terms of representation. Due to quantization, quantization noise is brought on. The bit allocator tries to eliminate the noise by incrementing the representation precision step by step. The psychoacoustic model has determined a signal-to-mask ratio for every subband. The quantization noise is eliminated once it has been "pushed" below the masking threshold: the noise can not be heard anymore. However, the number of representation bits is rather limited. The bits ought to be spent in subbands that need them truly.

The allocation is constrained by a quantization table. Table selection is implicit by bitrate and sample rate. The ISO/IEC 11172-3 document defines four different tables, Tables 3-B.2[a-d]. The ISO/IEC 13818-3 defines an additional table, Table B.1, which is used for MPEG-2 LSF. The `QuantizationTables` class stores these five tables. In addition to bit allocation, the tables are used in the quantization phase.

Code example

```
BitAllocator.Allocation alloc;
```

```
alloc = BitAllocator.allocate(quantizationTable[0],
                bitsAvail, scfsi, smr);
```

4.1.4 Quantization

The class `MP2Quantizer` implements a quantizer for subband samples. Annex C of the ISO/IEC 11172-3 document outlines the quantization. The quantizer maps subband samples represented in floating-point to a set of non-negative integers. The set is determined by the number of a subband and the quantization level specified by the bit allocator.

The codomains of quantized samples are stored in the `QuantizationTables` class. For every input sample s the condition $s \in [-1, 1)$ must hold.

Code example

```
QuantizationTable[] tables = new QuantizationTable[5];
for (int i = 0; i < tables.length; i++) {
    tables[i] = QuantizationTables.getTable(i);
}
...
```

4.1.5 Bit-level encoding

The bit-level encoding is the last phase of the encoding process. The quantized samples, along with scalefactor indices, selectors and quantization levels, are written to a bit stream. The class `MP2Data` is responsible for the bit fiddling. In short, it writes all the previously mentioned data in the form described in Section 2.4.1 of the ISO/IEC 11172-3. The coded data is put into an audio frame after an audio header. The class `MP123Header` writes the header, whereas the class `MP2Frame` writes the complete frame.

4.2 The unit tests

There are unit tests for every class in the encoder.

4.2.1 The tests

BitAllocator The allocator is fed with predefined inputs and the output is compared to hand calculated reference values.

Bit-level encoding classes The classes `MP123Header`, `MP2Data` and `MP123Header` are tested by setting an instance to a particular state and generating an array of bytes. The array is compared to handmade reference values.

MP2Encoder It is checked whether the encoder is able generate an encoded audio frame from a proper input frame. The encoder's accessors are tested as well. *The tests do not verify the real sanity of the output!*

MP2Quantizer Predetermined input values are quantized with the quantizer. The output values are compared to hand computed reference values.

PolyphaseQuadratureFilter The filter is applied with known inputs and the output is checked for validity. *The tests are rather austere. More interesting cases, such as frequency sweeps, are conspicuous by their absence.*

QuantizationTables The class is tested in a fashion of a spot test: Values from here and there are checked, emphasis being on borderline cases.

ScaleFactors The class is fed with hand picked input values. The output is then compared with precomputed reference values.

4.3 Known issues

There are some issues in the encoder:

- It is not possible to change a sample rate for an `MP2Encoder` instance. Some might consider this as a shortcoming, but in fact, it is a feature. At the minimum, switching a sample rate causes a transient glitch in playback (the synthesis filter has a memory!). Some oversensitive decoders might get confused temprarily or come to a complete halt. Yet, as the encoding is done on the frame level, the switch can be done (conciously) by creating a new encoder instance.
- The current implementation of the psychoacoustic model supports MPEG-1 sample rates only. For the MPEG-2 LSF, the encoder falls back to the dummy model.

5 Description of the PA model

5.1 Introduction

A psychoacoustic model determines the acoustically irrelevant parts of signal. From the human ear's standpoint, an audio signal has redundance both in the time domain as well as in the frequency domain. This redundance stems from the masking phenomenon, which exists in both time and frequency. Auditory masking means the human ear's inability to distinguish sounds in the presence of other sounds. In other words, this means that weak signal components in close proximity to more powerful components are masked, meaning they cannot be heard. This gives us room to introduce essentially inaudible quantization noise to some parts of the final encoded signal, greatly improving the compression ratio while still preserving signal quality.

The human ear's frequency resolution is non-linear across the audible frequency range (20Hz - 20kHz). The audible range can be broken up into frequency bands called the critical bands. The critical bands are of non-equal width, with the bands corresponding to the lower frequencies being much narrower relative to the higher frequency critical bands. This derives from psychoacoustic experiments, in which it has been concluded that the human ear has much better resolving ability at lower frequencies. In the time domain, the masking can be more easily approximated with the pre-masking occurring roughly 10 milliseconds before the masker and the post-masking effect lasting a little longer at about 50 milliseconds.

5.2 Implementation details

We have chosen to implement the MPEG-1 standard Psychoacoustic Model 1 which is, compared to for example the Model 2 in the same standard, less computationally expensive and makes some serious compromises in what it assumes a listener cannot hear. When we are converting samples to frequency domain we use a Hann weighting and then a Discrete Fourier Transform (DFT). Simply gives an edge artifact (from finite window size) free frequency domain representation. Model 1 uses 512 (Layer I) or 1024 (Layers II and III) sample window.

PA model need to separate sound into tones and noise components. Local peaks are tones, lump remaining spectrum per critical band into noise at a representative frequency. We have to calculate a masking threshold for each subband in the polyphase filter bank and select the minima of masking threshold values in range of each subband. Inaccurate at higher frequencies recall how subbands are linearly distributed, critical bands are not!

After the tonal and non-tonal components have been separated, they will be inspected in the decimation step. The non-tonal components are not decimated, as suspected due to their more pervasive effect. Instead a simple check is made to discard non-tonal components below the absolute threshold. As for the tonal components, for each tonal component a scan around the spectral line is performed with a width of 0.5 Bark. The tonal component with the maximum power in the are will be preserved and the others will be

ignored in further considerations.

The hard work is done now, we just calculate the signal-to-mask ratio (SMR) per subband SMR is signal energy per masking threshold. We pass our result on to the coding unit which can now produce a compressed bitstream.

Code example

```
double[] SMRs = computeSMRs(samples, scf);
```

5.3 Overview of the unit tests

Each step in the model has been unit tested. A crude ”system test” that tests the program flow through all steps has also been implemented.

6 Concerning the multiplexer

6.1 The idea

The multiplexer writes program stream in fixed sized chunks of bytes. Given a particular multiplex rate, a chunk corresponds a time span of precise length. This reduces the task of the multiplexer to time-division multiplexing: it has to decide what to do with every slot. Data for a single slot comes from at most one elementary stream. Not all slots are used for data. Sometimes it is necessary to output padding slots just to keep up the multiplex rate. If the rate is not fixed, the padding slots are not written for real.

Scheduling is the process of deciding from which stream to read data from. The implementation is based on the Program Stream system target decoder model (P-STD) described in Section 2.5.2 of ISO 13818-1. The model is a little (too) detailed. Some (small) corners have been cut in the implementation:

- The system clock is handled only in the accuracy of 90kHz which is more than enough for the task.
- The system clock reference present in the stream is counted from the first byte of the stream, not from timestamp itself. This helps to elude nasty offsets in the calculations. Besides, the error is very small if not insignificant and definitely not accumulating.
- The modelling of decoder buffers is rather approximative. However, the implementation should do its work. Errors should not be accumulating. In addition, real decoders tend to be rather varied. Thus working to the ultimate precision on a theoretic level would not really pay off.

6.2 The classes

The following are the most important classes of the multiplexer:

Class name	**Description**
AudioStream	Receives audio frames and generates a packetizable stream of access units.
AVPacketizer	Packetizes audio and video streams into PES packets.
ProgramStreamMuxer	Implements the core of the multiplexer. Controls packetizers and generates program stream.
VideoStream	Receives components of video and generates a packetizable stream of access units.

6.3 About the structure and workings

A general overview of data flows via method calls during multiplexing are is show in the Figure 6.3. The following sub-sections give a slightly more detailed view.

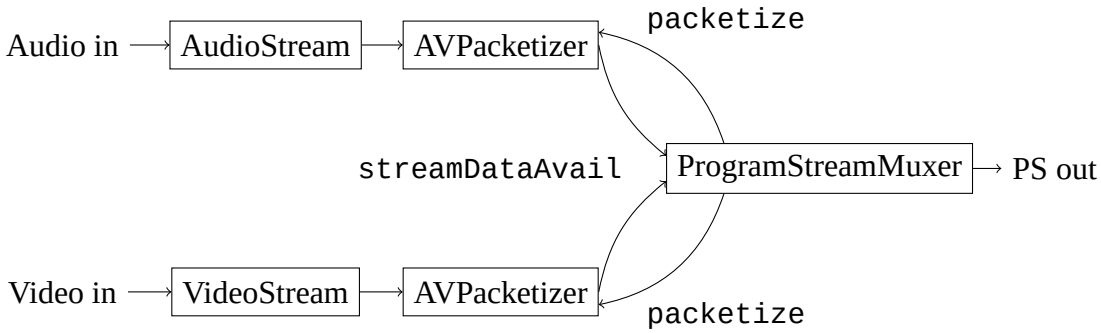

Figure 4: Data flow as method calls during multiplexing

6.3.1 Stream abstraction

The classes `AudioStream` and `VideoStream` implement a stream abstraction. They both take in pieces of data via various `write` methods. Received data are collected to form access units. Presentation and decoding timestamps are embedded into every access unit (the classes try to automatically maintain the timestamps). A stream class passes finished access units to packetizers associated to the class.

All the stream classes extend the abstract class `ElementaryStream`. For the moment, the base class has very little functionality. The class in itself is little artificial. Besides, the sub-classes implement redundantly the basic handling of packetizers. This has been conscious, as the sub-classes may have rather specific needs. Anyway, it seemed nicer to have a common base class.

The class `VideoStream` implements a simple minded verification of the video stream structure. Inside the class there is a twisted state machine which keeps track of incoming video elements.

6.3.2 Packetization

Instances of the `AVPacketizer` class accept access units via the `write(AccessUnit)` method. The access units are enqueued to be packetized later. When a packetizer determines it has enough data available, it notifies a packet listener (notifying does not need to happen immediately, slow “charging” and occasional “discharges” would actually make multiplexing more efficient). Thus the data flow does not need to go any further every time.

The multiplexer pulls data from a packetizer using the `packetize` method. The multiplexer defines which is the target size of a packet. The packetizer tries to fill the packet as full as it can. If there is not enough data available for a full packet, the packetizer retreats. This is necessary to avoid wasting bandwidth (in general, the multiplexer might not be able to get any benefit of the “spare space”).

6.3.3 Scheduling and multiplex generation

The core of the multiplexer lies in the `ProgramStreamMuxer` class. Once a multiplexer instance is notified of available data, it performs packet scheduling. At first, the multiplexer determines whether scheduling can be done at all. An imperative condition for scheduling is that every connected packetizer has either some data available or there will never be data to pull again. Without this being in force, the multiplexer can not make a choice between packet sources and scheduling has to be postponed. Owing to this, packetizers' data buffers can grow without a bound in the long run if streams receive data unevenly all the time.

After succesful scheduling, the multiplexer tries to get a packet from the selected packetizer using the `packetize` method. If the packetizer lacks data, the multiplexer has to fall back to wait. The scheduling has to be done again later. The decision should stay the same, though (here is a good opportunity for optimization). When a packet is finally generated, the multiplexer outputs the packet in a "chunk" of program stream.

6.4 Further development

6.4.1 Adding new input types

Adding a new input type for the multiplexer should be relatively easy as long as the format is either audio or video. At the bare minimum one needs only to feed an instance of `AVPacketizer` with the access units. A higher level interface, like `AudioStream` and `VideoStream`, is preferrable since it hides the details of packetization. To customize the packetization one has to sub-class `AVPacketizer`. For a simple example, see the implementation of the video stream packetizer, `VideoStream.VideoPacketizer`.

It will be more tricky to add an input type which does not follow the semantics implemented by `AVPacketizer`. If one can specify a decoding timestamp for every packet, the model of `AVPacketizer` might still do. However, it has to be generalized. Otherwise even the scheduler of the multiplexer needs to be altered.

6.4.2 Thread-safety needed

The present implementation of the multiplexer is not thread-safe. However, it would be very natural to use it simultaneously in multiple threads. There are two definite ways to do multithreading: The first one is to make the whole multiplexer thread-safe and let every thread multiplex. The second one is to employ a separate thread for multiplexing and make the access unit transfer from stream-level classes to packetizers thread-safe. The first option is more appealing since there would be no "hidden" threads beneath the surface. Then, in addition, the user would not be forced to use multithreading if one did not want to.

A simple, and inefficient, way to multithreading is accessing the multiplexer in `synchronized` blocks:

```
synchronized (psmux) {
    ...
    vs.write(...);
    ...
}
```

6.5 About the testing

Every component of the multiplexer has a unit test. The testing covers at least 80 per cent of the code, both in lines and blocks.

6.5.1 What is tested

AudioStream Basic timestamp manipulations and auto-increment are tested. In addition, the fundamental parts of packet streaming are tested.

AVPacketizer The packetizer is fed with an (artificial) input of access units. The workings of the decoder buffer model is checked. The model's effect on packetization is tested, as well. Furthermore, it is verified nothing is left into the packetizer's buffers after draining.

ProgramStreamMuxer Various knobs of the class are adjusted and then the effects are checked. A multiplex of two artificial audio streams is generated. The multiplex is inspected vaguely. *The uppermost test is missing: The validity of the multiplex is not verified! That would require at least a bitstream decoder which the library does not have.*

VideoStream A valid video sequence is applied to the class. The sequence contains I, P and B-frames. *The B-frame sequence is rather experimental.* Timestamp manipulations and auto-increment are tested, too. The video state tracking is tested by feeding the class a not-so-valid video sequence and checking whether exceptions are throw at the right moments. *The testing assumes the class refuses invalid input and retains consistent state.*

6.6 Known issues

The following are known issues in the multiplexer:

- The system clock precision is only 90kHz. If the multiplex rate is very high or if hair breaking between data sources is essential, the precision might not be enough. In spite of all, the multiplexer should be able to cope with any reasonable, normal multiplexing task. For rates over 1Gbit/s, this library might not be a sane solution ...

- Packetizer has to meet the target packet size fully or leave it short for at least seven bytes. Otherwise the multiplexer can not fit a padding packet into a chunk. Due to limitations in the `PESPacket` class it is troublesome to add stuffing bytes afterwards. Thus the packetizer has to take care of appropriate stuffing for the multiplexer (or leave a gap big enough).

- The packetizer does not “hoard” data before notifying the multiplexer. This induces often unnecessary scheduling as there is not enough data to fill a packet. A simple solution to the problem is to add a static level trigger: Notification is made only if the size of the available data exceeds the limit. A better solution would be an adaptive limit: The trigger level is constantly fine-tuned to match with the average data in a packet.

- The video state tracking in the `VideoStream` class is rather loose. It works only on the surface level. Thus many errors might go through unnoticed. At the same time, the class might complain about things that are in fact totally acceptable.

7 Future development areas

This chapter list and describes functionalities which have been omited from the encoder and what can be implement later.

- AC-3 encoding

  AC-3 is a product of Dolby Laboratories. The encoding is defined in the ATSC Digital Audio Coding (AC-3) Standard. Unlike the MPEG standards, the AC-3 standard is available for free. See the project documentation the prospects of Ac3.

- MP3 encoding

  MPEG-1 Layer III encoding is defined in the ISO/IEC 11172-3. It has a few similarities with the Layer II encoder. However, MP3 is mostly very different. See the project documentation the prospects of MP3.

- AAC encoding

  Advanced Audio Coding is a rather complex audio encoding scheme. AAC is defined in the MPEG-2 and MPEG-4 standards.

- Joint stereo support for the MP2 encoder

  In joint stereo mode high frequency sub-bands of a stereo signal are joined into one channel. In some cases this has only a small impact in the sound quality but helps to spend bits where they are really needed.

  A general idea for an implementation: The encoder needs a way to determine a sub-band from which to join the channels. It can be done by calculating how many bits are needed for a noiseless signal. The join-level is reduced as long as the coded frame will not fit into a frame. Yet the result might be rather rough. The channels can be joined simply by averaging. The joined sub-bands have a common quantized sample but their scale factors are separate. A humble solution would be to use the same factors for both the channels. Some improvement might be gained if the factors were separate. However, this affects the joining (simple average is not enough any longer). The bit allocation has to take into account the join. The quantization itself does not need to be changed. The bit-level coding class `MP2Data` has a support for the joint stereo mode already.

- Signal quality feedback for the MP2 encoder for VBR

  A signal quality feedback in the MP2 encoder would permit a rather generic way to implement a variable bitrate encoder. How it might be done: The encoder would output information on the signal quality as a by-product or in some other manner. Then, the information could be used to guide the encoder to do the encoding for real. A pseudo two-pass encoding might be implemented by windowing sample frames and encoding the oldest frames with the data from the window.

- Multithreading (or thread-safety)

  In some parts of the code multithreading would be a very natural way of doing things. For instance, the multiplexer has to deal with multiple data sources. Currently, it does its work by 'multiplexing' a single thread to do everything: processing input, encoding, multiplexing and so on. Doing all this by hand is rather awkward. Besides, in complex situations it might as prone to errors as is programming with multiple threads.

  Here multithreading has little to do with speed. For the most of the time there are data dependencies which inhibit truly parallel execution very quickly (or solving the dependencies would demand non-obvious solutions). To gain speed, doing less smarter is the way to begin with.

- Transport stream multiplexing

  It should be rather straightforward to implement a multiplexer for transport stream, once the idea of multiplexing is clear. The multiplexer has to cope with multiple programs each of which have a separate clock. Things are (relatively) easy as long as the eventual multiplex rate is fixed. Individual programs may be variable in bitrate, however. If the transport stream itself were of variable rate, it would be tricky to correctly advance the clocks. In addition to audio and video streams, the scheduler has to work with a system stream. The system stream has scheduling semantics different from audio and video.

  What (at least) is needed for a bare minimum implementation: The scheduler has to be adapted for transport stream. A transport stream packetizer is essential: It packs PES packets and system data into tiny TS packets. When necessary, it outputs null packets to keep up with the target bitrate. The multiplexer has to generate descriptive information, such as the program map, too. In overall, transport stream is relatively dynamic. To provide the user the sufficient amount of flexibility does require some care.

- Refactoring of the PESPacket class and its sub-classes

  The PESPacket class and its sub-classes should be refactored so that it would be possible to adjust packet's parameters at any moment without losing any contents. The current code behind the class PESPacket does not allow changing the parameters without wiping packet's contents. This behaviour is fine as long as the entire packet can be built at once. However, in some cases it is desirable to do small adjustments to the packet after it has been build initially. A good example is of adding stuffing bytes in the multiplexer.

- Support for Linear PCM multiplexing

  Although it has little to do with the MPEG standards, adding a support for linear pulse-code modulated (that is, uncompressed) audio to the multiplexer might be an interesting task. It would allow the easy trial of multiplexer extension without heavy background work (as would be the case with AC3, for example).

- Multichannel support

  The MPEG-2 audio (described in the ISO 13818-3 document) contains extensions to MPEG-1 audio to support multichannel coding. The feature requires (possibly rather extensive) changes on every stage of the encoder.

8 Implemented features

This chapter list and describes functionalities which have been implemented.

- PQF see sections 4.1.1. and 4.2.1.
- Scalefactors see sections 4.1.2. and 4.2.1.
- Bit allocation see sections 4.1.3. and 4.2.1.
- Quantization see sections 4.1.4. and 4.2.1.
- Packaging see sections 6.3.2. and 4.2.1.
- Multiplexer see sections 6.0 and 6.5.1.
- Psychoacoustic model. Step 1: FFT from Minim -library is used. [9]
- Psychoacoustic model. Step 2: determination of the sound pressure level

  This step is fairly straight forward, so there shouldn't be any problems with it. First we just find the highest scalefactor and calculate the power corresponding to it. Then we compare that power to the spectral line powers of that subband replacing the power if a higher one is found. Then the highest value is chosen for the subband in question.

- Psychoacoustic model. Step 3: determination of the absolute threshold

  This step is a simple selection of a right table, so there shouldn't be any problems here.

- Psychoacoustic model. Step 4: finding of tonal and non-tonal components

  This step if quite complicated. Tonal components should be calculated correctly. There might be some problems with non-tonal components. Are they located on the right spectral index and is the components power calculated correctly from the spectral lines.

- Psychoacoustic model. Step 5: decimation of maskers

  In this step the relevant tonal and non-tonal components are chosen for masking. Tests check that the right kind of components are chosen and unfit are discarded. This step should be correct, but if there is something wrong check the non-tonal components and that they are handled correctly.

- Psychoacoustic model. Step 6: calculation of individual masking thresholds

  This step is quite complicated with lots of calculations and tables. MATLAB implementation of the PA-model was used as a guideline to code this step. It should work correctly, but testing was difficult due to its complexity.

- Psychoacoustic model. Step 7: calculation of the global masking threshold

  This step takes the individual masking thresholds table from step 6. The mechanics of this step aren't too complicated, so the process has been tested with smaller tables and simple values. There shouldn't be problems with this step. MATLAB implementation of the PA-model was also used as a reference in this step.

- Psychoacoustic model. Step 8: determination of the minimum masking

  This step takes the global masking thresholds from the previous step and calculates the minimum values per subbands. The testing was simple, feeding the method a table and calculating by hand that the smallest value is selected.

- Psychoacoustic model. Step 9: calculation of the signal-to-mask ratios

  This step is simple operation of substacting one table from another and storing the results in a new table. There shouldn't be any problems here.

- System testing and Validation tested with mplayer Mplayer plays encoded audio file without errors. We have done a subjective hearing based comparison between original audio file and encoded files, where different bit rates were used.

- Sample rates for PA-model

  Sample rate support for the PA-model was a simple as adding more tables to the class from the MPEG-1 standard and choosing the right ones by parameters. MPEG-2 introduced some new sample rates for the PA-model which haven't yet been implemented.

References

1 ISO/IEC INTERNATIONAL 13818-1 STANDARD
Information technology - Generic coding of moving pictures and associated audio information: Systems
Second edition
2000-12-01

2 INTERNATIONAL STANDARD 13818-2
INFORMATION TECHNOLOGY - GENERIC CODING OF MOVING PICTURES AND ASSOCIATED AUDIO INFORMATION: VIDEO
1995 (E)

3 ISO/IEC 13818-3 Information Technology - Generic Coding of Moving Pictures and Associated Audio: Audio
1994

4 ITU-T H.222.0 SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS Infrastructure of audiovisual services - Transmission multiplexing and synchronization Information technology - Generic coding of moving pictures and associated audio information: Systems
05/2006

5 ITU-T H.222.0 SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS Infrastructure of audiovisual services - Transmission multiplexing and synchronization Information technology - Generic coding of moving pictures and associated audio information: Systems Amendment 2: Carriage of auxiliary video data
08/2007

6 ITU-T H.222.0 SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS Infrastructure of audiovisual services ¿ Transmission multiplexing and synchronization Information technology ¿ Generic coding of moving pictures and associated audio information: Systems Technical Corrigendum 1 Correction of zero_byte syntax element and stream_id_extension mechanism
06/2008

7 EclEmma Java code coverage tool
http://www.eclemma.org/

8 JUnit testing framework
http://www.junit.org/

9 Minim library
http://code.compartmental.net/tools/minim

References