Muksis project

Application report

Richard Domander
Tuomas Mäenpää
Teemu Nisu
Tommi Teistelä

Version: 1.0
Public
January 16, 2009

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Approver	Date	Signature	Name verification
Project manager	__.__.2009		
Customer	__.__.2009		
Supervisor	__.__.2009		

Document information

Authors:

- Richard Domander (RD) dimadoma@jyu.fi 050-3482668
- Tuomas Mäenpää (TM) tutamaen@jyu.fi 040-7600465
- Teemu Nisu (TN) tejonisu@jyu.fi 040-8349310
- Tommi Teistelä (TT) totateis@jyu.fi 045-6528709

Document title: Muksis project, Application report
Pages: 27
Source file: application_report-1.0.tex

Abstract: This is the application report of software project Muksis. This document introduces the reader to the existing architecture and inner workings of MPlayer (mainly version 1.0-rc1), and explores in detail the parts relevant to the features implemented in Muksis project.
Keywords: Software architecture, black frame detection, DVB subtitles, seek in MPEG TS, MPlayer

Project information

The architecture of MPlayer is highly modular, but somewhat ill-defined. It takes a considerable time to understand it, and to figure out which parts need to be altered during the implementation. This document is intended for people interested in further development of MPlayer, and the features implemented in the Muksis project.

Authors:

- Richard Domander (RD) dimadoma@jyu.fi 050-3482668
- Tuomas Mäenpää (TM) tutamaen@jyu.fi 040-7600465
- Teemu Nisu (TN) tejonisu@jyu.fi 040-8349310
- Tommi Teistelä (TT) totateis@jyu.fi 045-6528709

Contents

1 Introduction

The original purpose of the Muksis project was to provide commercial skipping and DVB subtitle support for TV recordings in Matthieu Weber's Leffakone home theater environment. Weber had already implemented these features for an older version of MPlayer (pre4) back in 2005, so we tried to either port or reimplement them for a newer version of MPlayer.

Commercial skipping required black frame detection to find the commercial breaks, and more accurate MPEG TS file seeking. DVB subtitle support required decoding, drawing and timing of subtitle packets. It also required various changes throughout MPlayer, because the program wasn't designed to work with TV recordings, only broadcasts. Weber also came up with an additional idea: he requested a possibility to normalize and compress audio during playback.

We set out with the latest stable release of MPlayer at the time (1.0-rc2), but later on Weber noticed he couldn't compile it in Leffakone environment, so we reverted back to version 1.0-rc1. During the project we decided to port the features for the development (or SVN) version of MPlayer too. The differences between versions affected mostly DVB subtitle support.

When we started the project, we thought that the implementation of the features could provide to be a challenge, because the team didn't have much experience in working with a larger application that is written in a non-object-oriented language. We estimated that handling the coherence and integration of the components is the most demanding task, but the components proved to be very independent of each other. The differences between the versions of MPlayer affected mostly the implementation of DVB subtitle support.

Chapter 3 introduces the general architecture of MPlayer, and explain which modules are used with MPEG TS files. Chapter 4 explains how MPlayer, and the features implemented in the project are used in Leffakone environment. Chapter 5 describes black frame detection, chapter 6 seeking, chapter 7 DVB subtitle support, and chapter 8 audio filtering in more detail. Chapter 9 explores the various file formats and data structures used in the project. In chapter 10 the implementation of the features is analyzed, and in 11 their testing is explained. All the chapters apply to MPlayer version 1.0-rc1 unless otherwise specified.

2 Acronyms

The following acronyms and terms appear in this document:

Back-to-back testing Execution of a test on the similar implementations of software and comparing the results

BFVF Black Frame Video Filter

CLUT Color look up table

Demultiplexing The act of separating elementary streams from MPEG TS

DVB subs Digital Video Broadcasting Subtitles

EDL Edit Decision List

EOF End of File

Elementary stream A video, audio or subtitle stream carried in MPEG TS

FIFO First In First Out

GOP Group of Pictures

GUI Graphical User Interface

MPEG TS Moving Picture Expert Group Transport Stream

PES Packetized Elementary Stream

PIC Programmable Interface Controller

PID Packet Identifier

PTS Presentation Time Stamp

3 MPlayer

MPlayer is a cross-platform, open source media player, whose architecture can be seen as an instance of the "Pipes and Filters" architectural pattern. This follows naturally from the process of media replay on computers. There are myriads of codecs, (de)multiplexers, and video processing filters for myriads of different medias, video, audio and subtitle formats, video output streams, video effects and platforms. The software is highly modular, but the responsibilities of the modules are somewhat ill-defined, and everything revolves around `mplayer.c`. The structure of the program has improved since MPlayer version 1.0-pre4, but is still in need of heavy refactoring.

`Mplayer.c` runs the show on a high level: it handles and delegates user input, keeps the play list up to date, and figures out the necessary components for playback (decodec, demultiplexer, subtitles, media handling...). With command line parameters the user can, amongst other things, choose a video codec, subtitle and audio language, and add various video filters and change their individual parameters. In the MPlayer software package there's also MEncoder which is a versatile video decoding, encoding and filtering tool.

Figure 3.1 explains what happens when you play a MPEG TS file with MPlayer. First the individual streams are demultiplexed in `demux_ts.c`. During the replay the user can seek the file (change position in the streams), or supply MPlayer an EDL file which, for example, makes MPlayer skip commercials. All seek commands are passed to `demux_ts.c`, because it knows how to change position in the streams, and how to resynchronize them after seeking. Before presentation the streams are decoded, for example, `dvbsub.c` handles the decoding of DVB subtitles. With filters various effects (audio compression, video de-interlacing...) can be added to video and audio streams before presentation. Black frame detection (`vf_bf.c`) also acts as a filter, but instead of processing the picture data, it only marks down all the black frames it finds into an EDL file.

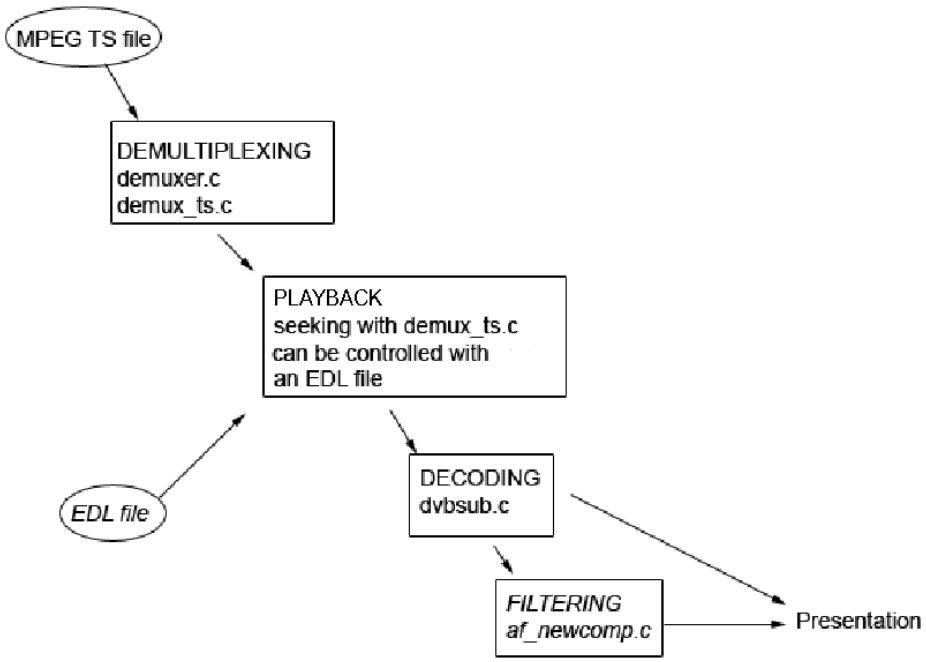

Figure 3.1: General architecture of MPlayer 1.0-rc1

4 Leffakone - the Target Environment

Leffakone is Matthieu Weber's home theater PC (see figure 4.1). Leffakone runs on an old PC, which has a DVB tuner, a timer device, and an infrared receiver. A TV antenna is connected to the DVB tuner. The timer is used via parallel port, and is connected to the Wake-on-LAN, so that the computer can be powered up when it's time to record a TV program. The infrared sensor is used to pick up signals from a remote control.

On the software level Leffakone consists of the assembly code for the PIC of the timer device, the python code for automating recording and shutting down the computer after recording, Lirc for handling infrared signals, and Freevo, which is a personal video recorder application for Linux. Freevo uses MPlayer (version 1.0-rc1) to display and record TV broadcasts. Freevo uses X Window System for its GUI.

4.1 Usage of Black Frame Detection

In Leffakone environment black frame detection is used in two phases. In phase one (see figure 4.2) shortly after the recording has started, MPlayer is starts to run the recording through the BFVF to create an EDL file (see chapter 9) for it. Only one video, audio and subtitle stream is recorded. The DVB tuner selects these streams from the MPEG TS. In phase two (see figure 3.1) the recording is opened with the newly created EDL file, which then controls the playback of the file, i.e. commercial breaks are skipped.

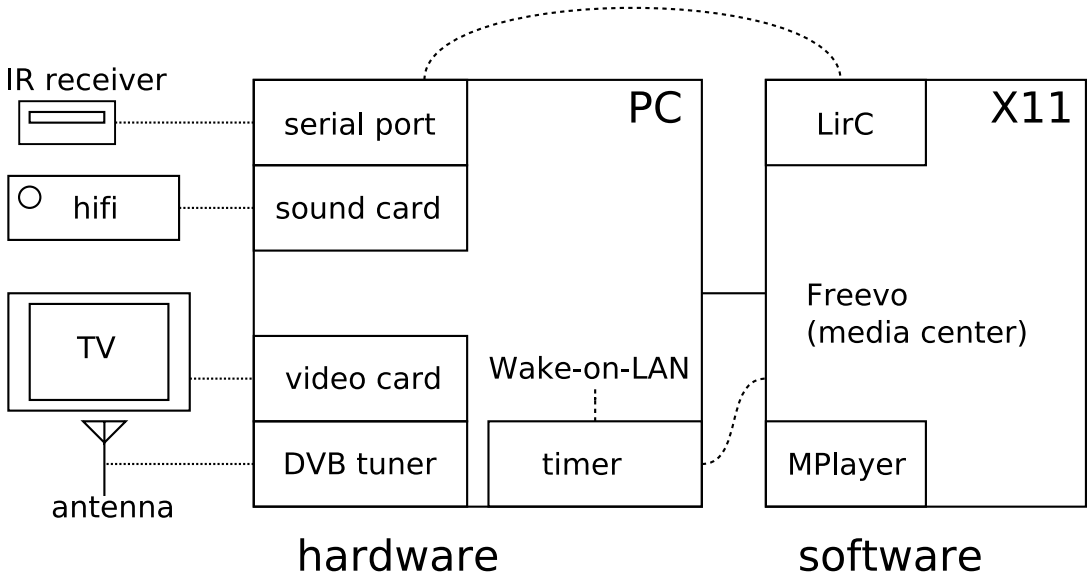

Figure 4.1: The structure of Leffakone

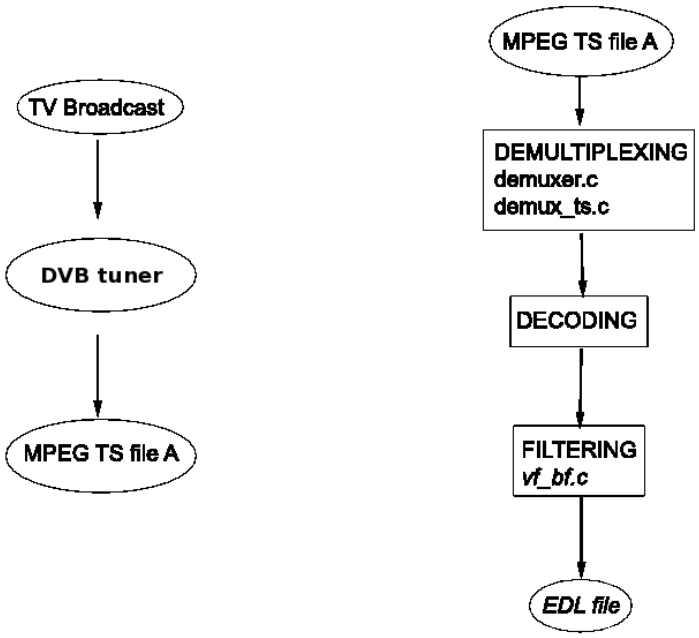

Figure 4.2: Creation of a TV recording and EDL file in Leffakone

5 Black frame detection

5.1 Background

The purpose of black frame detection is to create an EDL file, which can be used to skip commercials during playback of TV recordings. BFVF can be used to detect commercials because commercials are short sequences between black frames. When the user requests video filters with command line parameters, MPlayer passes the request to `vf.c`, which manages the video filter chain. `Vf.c` then parses the command line parameters, opens the specified filters, supplies them their individual parameters, and arranges them to a linked list. During playback decoded video frames pass through this chain of filters before they reach video out (see figure 5.1).

Video filters are very independent. In addition to programming the BFVF itself, you only have to mention the new filter in `vf.c` and the `Makefile` of `libmpcodec` so that MPlayer and the compiler find it. While processing the video BFVF's `put_image()` function writes the PTSs of the detected commercials into an EDL file. The file is opened in BFVF's `vf_open()` function, and freed in `vf_uninit()`. Together with accurate MPEG TS seeking the generated EDL file can be used to skip commercials during playback. The EDL file is described in detail in chapter 9.

The structure of MPlayer has improved since version pre4, and you don't need to get the PTS of a decoded frame as an external variable from `dec_video.c` anymore. Nowadays there's a version of `put_image()`, which has PTS as one of its parameters. The parameter is passed to it by `vf.c`.

5.2 Implementation

Black frame detection was ported from Matthieu Weber's old code. Some refactoring was done, for example `calcimg()`, which decides whether a frame is black or not, was cleaned up. There was some redundancy (copy-paste code), "magic numbers" i.e. constants with no explanation, and variables with confusing names. The detection algorithm itself wasn't changed. While doing "printf debugging" we noticed that `vf_config()` gets called every time video parameters change, for example, a commercial may use different aspect ratio than the program, so no initializations should be done in this function. There was also a memory leak:

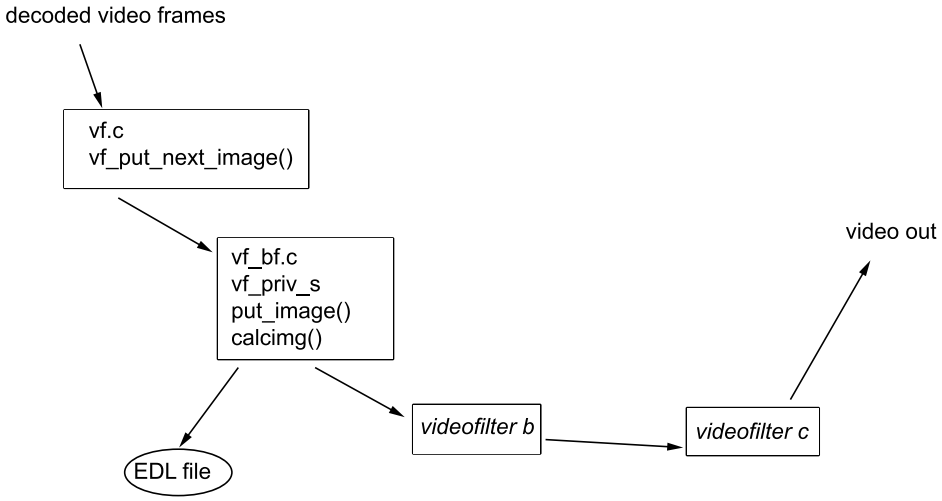

Figure 5.1: Architecture of black frame video filtering

`vf_priv_s` structure was never deallocated. Now `free()` for the structure is called in `vf_uninit()`.

`Vf_priv_s` structure holds the pointer to the EDL file to be written, and variables necessary for detection of commercials: length of the current sequence between black frames, time when the previous black frame sequence started, length of the previous black frame sequence etc.

`Calcimg()`, which is called in `put_image()`, decides whether a frame is black or not by calculating the average luminance value of the frame, and marking down the highest individual luminance value. If both values are below their corresponding thresholds, the frame is considered to be black.

`Put_image()` is called once for every video frame. It creates the markings for the resulting EDL file. A possible commercial break is considered to begin when a black frame is detected, i.e. a black sequence begins. The length of the non-black sequence following the black sequence is measured. If the length of the non-black sequence before a new black sequence is shorter than `ad_max_len` the sequence is considered to be an ad. As long as the program detects new ads the end of the commercial break

is moved to the middle of the latest black sequence (see figure 5.2). A commercial break ends when a non-black sequence longer than `ad_max_len` is encountered. If EOF is encountered, and the length of the sequence between the latest black frame and EOF is shorter than `ad_max_len` then the end of the commercial break is set to EOF (note: this is done in `vf_uninit()`!).

In Weber's patch the commercial break was marked to the EDL file from the beginning of the starting black sequence to the end of the ending black sequence. We changed it so that it now marks the skip from the middle of the starting black sequence to the middle of the ending black sequence. This was changed because of the MPEG format, i.e. if we jump to a frame that is not the key frame of the GOP an ugly "mosaic" is seen. This doesn't bother as much if the frame is black. Also, the short black sequence now seen in the middle of the program lets the user know a jump occurred. We think it's now less confusing if a new program starts after a commercial break.

The thresholds, EDL filename and `ad_max_len` can be changed by the user with command line parameters. The default values for the thresholds and `ad_max_len` were experimentally determined by Weber. According to him the program is "99% accurate" with them. The parameters are parsed in `vf_open()`. We changed the implementation so that an error message is printed if all the given parameters cannot be parsed. Default value is used for the faulty parameter, and all the parameters following it. Default values are also used if the user gives no parameters.

A mention of our BFVF was added to `vf.c` and Makefile. In versions pre4, rc1, and rc2 the Makefile in sub folder libmpcodecs needs to be changed. In the current development version of MPlayer you modify the Makefile in the root folder of the program.

Figure 5.2: Commercial detection in BFVF

6 MPEG TS seek

6.1 Background

In order to skip commercials during playback, we need to be able to seek accurately, i.e. to change position in the streams in MPEG TS. During playback `mplayer.c` waits for keyboard events, and stores the detected key presses into a FIFO buffer. If a seek command (i.e. a key mapped to seek is pressed) is detected, a seek is initiated by `mplayer.c`. A seek can also be automatically initiated by an EDL file. First `mplayer.c` calls the seek function of the generic demultiplexer (`demuxer.c`). The generic demultiplexer checks if the file format and selected media allow seeking (for example, DVD allows, TV does not), calculates the distance of the seek target from the current file position, and makes other necessary preparations (resets the synchronization of the demultiplexed streams, for example). The generic demultiplexer has a function pointer to the seek function of the format or media specific demultiplexer (in this case `demux_ts.c`). The specific seek function does the actual seeking. After the seek in `demux_ts.c` is done, that is: after the correct new position in the video is found and audio is synchronized with video, `mplayer.c` calls `update_subs()` function in `mpcommon.c` (see chapter 7). `Update_subs()` initiates the subtitle format specific tasks needed to display current subtitles.

6.2 Implementation

The improved seek code in `demux_ts.c` was redone to study different ways of implementing it. The basic principle remains the same, however.

As the MPEG-TS stream by itself doesn't contain a typical header-like structure to refer to, the seek function must work by estimating the bitrate of the stream and using the MPEG timestamps in it as references.

The modified seek function first stores the current MPEG video timestamp from the demuxer's state (`sh_video->pts`) in a temporary variable and resets the buffers used to store already decoded video, audio and subtitle frames by calling `reset_fifos()`. Then, it takes the demuxer's current bitrate estimate, calculates a new position in the input file and seeks in the raw file stream by calling `stream_seek()`.

Since seeking backwards accurately is generally more difficult, the modified seek

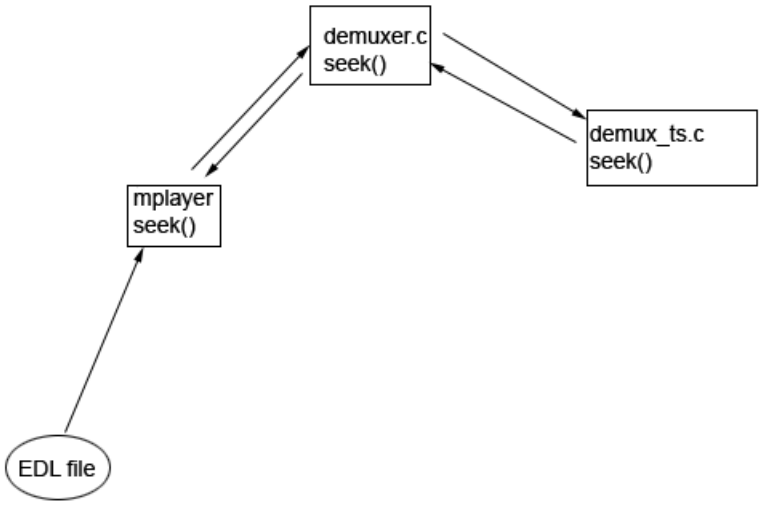

Figure 6.1: Architecture of MPEG TS seeking

function attempts to avoid doing so by performing shorter seeks when moving forward in the stream, and longer seeks when moving backwards. This also largely avoids the issue of running into the end-of-file when seeking is performed near the EOF and the demuxer has overestimated the stream's bitrate.

Following the seeking in the raw file stream the function attempts to find the video stream again by calling `sync_video_packet()` and, if any audio is present, re-synchronizes the audio and video streams by calling `skip_audio_frame()` until the audio timestamps' values are greater than those of the video timestamps.

After the synchronization the last video timestamp (updated during the synchronization) is compared to the one stored at the beginning of the seek. If the difference is more than 0.5 seconds and the seek attempt has been repeated less than 16 times, the seek function attempts to perform another seek using the difference as the seek duration. The 16 repeat limit prevents a situation where the synchronization code ends up repeatedly "cancelling" a short backwards seek, which could lead to an infinite loop.

If the difference is less than 0.5 seconds, the seek function simply stops (seeking

more accurately would require implementing the seek functionality on a lower level, taking the different types of MPEG frames into account and possibly requiring modifications elsewhere, too) The possibility of the timestamp values resetting somewhere between the beginning and the end of the seek is dealt with by testing if the new timestamp value is off by more than several minutes in the "wrong direction" (compared to the seek duration) and changing the initial timestamp variable to match the new situation.

7 DVB subtitles

7.1 Background

YLE channels in Finland use DVB subtitles, which are separate from the video stream. In order to show these subtitles, we need to implement timing and decoding of the DVB subtitles. Because both DVD and DVB subtitles are bitmaps (in most cases anyhow) and both formats have support for multiple languages, you can take advantage of the existing code for DVD subtitles in MPlayer. Essentially the data structure `spudec_handle_t` in `spudec_struct.h` (see figure 7.1) acts as a wrapper for DVB subtitles. In our implementation, we utilize the packet queue in this structure, and in the existing code decoded bitmap subtitles are passed on from this structure to the displaying functions.

When MPlayer opens, it initializes DVB subtitles in `mplayer.c` by calling `dvbsub_init_data()`. During playback, and after seeking, `mplayer.c` calls the function `update_dvdsubs()`. `Update_dvdsubs()` calls the functions necessary to handle DVD and DVB subtitles. The function deduces the type of the subtitles from the type of the demultiplexer. In case of DVB subs, `update_dvdsubs()` first calls `dvbsub_heartbeat()` in `dvbsub.c`, which checks whether the first packet in the decoding queue should be processed, i.e. is the current time stamp $\geq$ packet time stamp. When a packet is processed in `dvbsub_decode()` the resulting image is handed to the `spudec` structure, from which the image goes on to the displaying functions. After calling `dvbsub_heartbeat()`, `update_dvdsubs()` loops as long as it can get PES packet data from the buffer of the subtitle stream, i.e. as long as it can call `ds_get_packet_sub()` in `demux.c` successfully. After every call to `ds_get_packet_sub()`, `update_subs()` calls `dvbsub_assemble()`, and passes it the received packet data. `Dvbsub_assemble()` adds the packet data to the decoding queue of the spudec struct. When a new PES packet is added to the decoding queue in `dvbsub_assemble()`, time stamp and time out are added to the these packets to ensure correct decoding, and thus presentation order, time and duration. In summary, `update_dvdsubs()` does two things: it decodes the subtitle packets, and makes sure that they are presented at the right time.

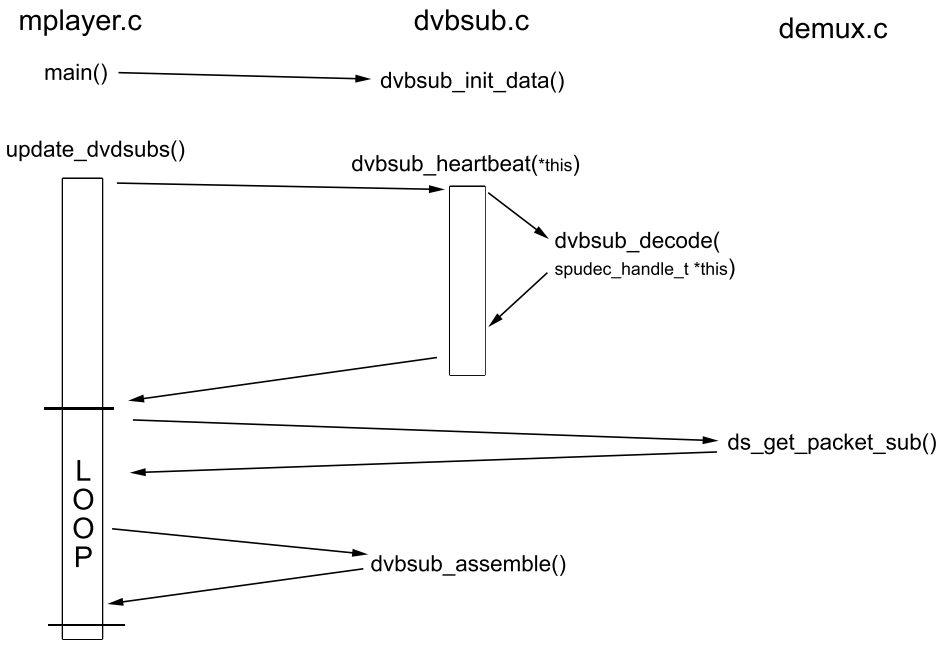

Figure 7.1: Architecture of DVB subtitling

7.2 Implementation

The DVB subtitle packet processing code from Weber's patch was used almost as is. Only the memory behavior was changed at first, and the drawing of the decoded image was put into one function instead of two. In the patch code memory for the decoded image was deallocated, and then allocated again every time the subtitles changed. Also, in the old code, the need for memory was dynamically calculated based on the dimensions of the subtitle regions. The code was changed so that the maximum possible amount of memory needed (less than a megabyte) is allocated before the first subtitle packet is decoded, and the subtitle regions are erased (filled with 0) before a new packet is decoded. This change in memory usage was later removed, because it created a conflict with `vf_scale.c`, and the gain in performance wasn't worth the time that would've been needed to resolve the issue.

During the project Weber noticed that he had supplied us with an outdated version of his code. He then sent us the latest version he had coded. The newer version rendered the subtitles better, so we included the changes it introduced to our version. The old version didn't initialize the pixel code (the CLUT index of the pixel's color)

to zero in `decode_4bit_pixel_code_string()`, and it miscalculated the alpha values in `set_palette()`. Because of these errors, the background of the subtitles wasn't completely transparent, and the glyphs didn't have sharp black edges.

During testing we noticed that with some files, the subtitles didn't get shown at all. There were two reasons for this problem. When a MPEG TS file is opened, `ts_detect_streams()` in `demux_ts.c` reads a couple of seconds of the file to find out which elementary streams it contains. The first reason for the problem was that this probing for streams ended after a video and audio stream had been found, i.e. it could end before the probing limit was reached, thus missing possible subtitle streams starting in the beginning of the file. The premature exit condition was changed so that the probing can end before the limit only if also a subtitle stream is found, or no subtitles were requested (dvdsub_id=-2). The second reason for the problem was that `demux_ts.c` was designed to work with a TV broadcast, not a TV recording, i.e. `ts_parse()` didn't recognize the type of new elementary streams without the meta data present in a TV broadcast (MPlayer records only one video, audio and subtitle stream). We modified the function `ts_parse()` so that now if it can't recognize the elementary stream by any other means, it reads a packet (with the help of `pes_parse2()`) from the file, and tries to get the type of the stream from it. If the type is not recognized, the packet is discarded. If the type is recognized (to be DVB subtitles) the position in the file is moved back to the beginning of the packet so that the actual contents can be read. Also, the demultiplexer is updated to use this stream, for example if the new stream is a DVB subtitle stream, the subtitle id is changed to PID of the newfound stream. `Ts_parse()` is a rather confusing function with a dual role: it is used both for the (initial) detection of streams, and packet parsing during playback (reading a packet from the file, and then put it to the corresponding output buffer [video, audio or subtitle] of the demultiplexer).

Some changes in `mplayer.c` were needed to make the subtitles work. Several variables in MPlayer refer to the state of different kinds of subtitles. Relevant for DVB subtitles are: `dvdsub->id`, which is used to request bitmap subtitles from the demultiplexer, sub_auto, which tells should subtitles be used at all, and `demuxer->sub->id`, which tells what is the PID of the subtitle stream currently used. If sub_auto=0 (the user used parameter -noautosub) DVB subtitles aren't initialized. During playback we check if a subtitle stream was found by comparing `dvbsub->id` to `demuxer->sub->id`. Both equal -2 if no subs are requested. If no subs were found in initial probing, both variables equal -1 when playback starts. If the variables become inequal during playback, it means subtitle stream has been found,

and we set `dvbsub->id` to `demuxer->sub->id`, and call `update_dvdsubs()` to show the first packet detected. The functionality now in `update_dvdsubs()` was initially in the `main()` function in `mplayer.c`.

7.3 Version differences

The implementation of DVB subtitles differs a bit between the versions. In version newer than rc2 function `update_subtitles()` in `mpcommon.c` is called instead of `update_dvdsubs()` in `mplayer.c` as is done in rc1. The change was made to clarify the responsibilities of different modules.

Instead of `dvbsub->id` and `demuxer->sub->id` as in rc1, `mpctx->d_sub->id` and `mpctx->demuxer->sub->id` are used in rc2 and newer versions to check whether we have found a subtitle stream during playback. Mpctx is a structure of type MPContext. This is a sort of superstructure, which holds all the variables and structures necessary for playback, so that they can be found in one place.

In versions newer than rc2 the spudec structure is initialized in function `init_vo_spudec()` in `mplayer.c`. In rc1 it is initialized in the `main()` function of `mplayer.c`.

8 Audio filtering

Both the actual "peak" volume level and perceived volume (affected by the frequency distribution and the dynamic range of the sound) tend to vary between different TV broadcasts, possibly more so than between different DVD movies, which MPlayer was originally made to play. MPlayer's current source tree does contain some audio filters intended to control these aspects of the sound being played, but both their documentation and implementation are somewhat lacking for the purposes of Leffakone. The project team has thus decided to look at either improving the existing filters or writing suitable ones from scratch.

MPlayer's audio filter interface resembles that of the video filters: a filter has a standard interface structure containing function pointers for key operations, such as (un)initializing the filter, configuring it and processing data through the filter. This structure is added to the list of audio filters in `af.c` and the rest is handled by MPlayer's existing code. Through this interface, audio filters may receive parameters from the command line or from other parts of the application, such as any of the GUI frontends for MPlayer. Audio filters may be "chained" in the same way as video filters.

8.1 Dynamic range compression

The "dynamic range" of any given information is the range of possible values between its lowest and highest levels. For audio, this can be considered the range between the most quiet (but still relevant for the experience) parts of the sound, such as quiet speech, and the loudest parts of the sound, like gunshots or explosions. The desirability of a high dynamic range in audio mostly depends on the quality of the listening environment; with low-quality speakers or a noisy environment the most subtle parts of high-dynamic range audio will be lost. Because of this the dynamic range of audio is often reduced through dynamic range compression.

The Muksis project has implemented a new simple "hard-knee" dynamic compression filter (see figure 8.1) similar to that used by many hardware DVD players when outputting sound to a device expected to be of low quality, such as the built-in speakers of a TV. The filter's source code can be found in `af_newcomp.c`, and it can be controlled from the command line with parameters like

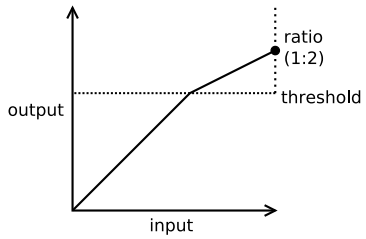

Figure 8.1: A "hard-knee" audio compression filter

`mplayer -af newcomp=0.7:0.5`, where 0.7 is the compression threshold and 0.5 the ratio by which anything exceeding the threshold is compressed.

The exact function used when the input exceeds the filter's threshold is:

$$output = (input - threshold) \times ratio + threshold \tag{8.1}$$

which scales the part exceeding the threshold by the compression filter's ratio setting.

8.2 Normalization

The actual "peak" volume level of audio tends to vary significantly from the maximum allowed by its format; this can make for an irritating listening experience. The volume level can be corrected through normalization: applying a multiplier to the audio sample values bringing the peak volume to the maximum level, or close to it – generally, most implementations do not raise the peak volume to the absolute maximum as this can result in distortion due to limitations in various parts of the audio pipeline.

For short audio clips such as recorded music, this can be achieved easily by simply scanning through the data and finding the highest absolute value, then amplifying the sound by the maximum value divided by the highest value found, scaled to whatever peak volume is desired. For very long audio clips or ones being currently streamed, this approach can't work and the normalization filter must instead react to changes in the volume dynamically. The Muksis project has been looking at ways to do this in MPlayer.

MPlayer's own volume normalization filter in `af_volnorm.c`, while apparently quite old and unmaintained in the rc1 version, was found adequate for the task.

The filter can take two parameters, which are the compression method to be used (the filter implements two, for some reason) and the average level the normalization filter tries to maintain, given in a syntax similar to the other audio filters. When run with no parameters, the filter tries to keep the peak volume at the maximum level allowed.

9 File formats and Data structures

9.1 EDL file

In our implementation EDL files are used to skip commercials during playback. These files are automatically created by BFVF (see chapter 5). The format is very simple, you only need begin and end timestamps (PTS) of the actions, and the types of the actions, i.e. do you want to mute (=1) or skip (=0) a part of the video. Below is an example EDL file.

Example EDL file

Begin	End	Type
8372.18	8409.26	0
8595.30	8869.18	0
9724.62	9965.10	0
10855.34	11084.90	0
11814.46	12008.66	0

9.2 Structure and contents of MPEG TS

In Finland digital television is broadcast in MPEG TS format. The broadcast consists of several MPEG TSs, called “channel packets”. One MPEG TS consists of one or more multiplexed programs, which consist of several streams. These streams consist of PES packets, which can contain video, audio, subtitle, meta, or other, so called private data (teletext, network information, encryption key for scrambled channels...). These packets carry PTSs, which are used for timing. The meta data tells which streams belong to which program. In TV recordings made by Leffakone (see chapter 4) there’s only one video, audio and subtitle stream present. During playback the demultiplexer separates these streams from each other.

The video format used in MPEG TS is MPEG 2. MPEG 2 videos consists of groups of pictures (GOP), which hold 0.5 seconds of video. GOPs consist of different kinds of frames: I-frames, P-frames and B-frames. I-frames are the key frames, they are full pictures. P-frames only carry data which is used to create a displayed frame by modifying the previous I-frame. B-frames are more complicated bi-predictive

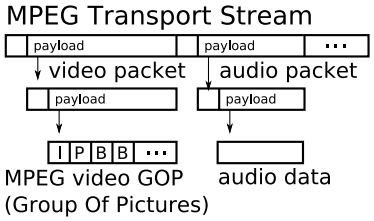

Figure 9.1: Structure of the MPEG Transport Stream

frames, i.e. they consist of modifications created by analyzing both the frame before and after the B-frame.

A DVB subtitle packet holds one page instance, which is an unchanging set of subtitles. A page instance consists of several regions (areas upon which subtitles can appear), which consist of objects. One object depicts one glyph. A page instance can also be an "end of display set", which is used to signal that the current subtitles can be cleared.

10 Analysis

It's not certain how much the implementation could be changed, because the biggest tasks came from the unexpected bugs that just needed to be solved the way they were. The parts that could've been implemented differently were the parts ported from Weber's patch, i.e. the rather independent technical solutions to the required features. Only our implementation of MPEG TS seeking differed in more than superficial manner.

- The advertisement recognition code in BFVF's `put_image()` function could be refactored to ease maintaining and understanding. One useful abstraction for it could be the state machine (i. e. state of recognition: black sequence has begun, in non-black sequence etc). The structure `vf_priv_s` in `vfbf.c` could be used to hold the state, and other variables of the state machine. On the other hand, state machine can make the code more complicated than necessary.
- `Calcimg()` could be changed to detect sequences of different (uniform) colors, because sequences between commercials aren't black in all countries. The color could be supplied with command line parameters. Sometimes TV broadcasters don't follow standards, and sequences between ads aren't uniform in color i.e. the network logo isn't removed. Commercial detection could be improved by monitoring the audio levels too, i.e. instead of searcing black frames, the filter would be searching black, silent frames. This interplay between audio and video inspection would be difficult, or rather arduous, to implement given the MPlayer architecture.
- The modified `demux_seek_ts()` function's implementation in `demux_ts.c` tries to maintain the MPlayer version's overall structure as far as possible to make the difference between versions seem less radical. Still, the complexity of the function is easily twice the original's (not that it was very complex). Some further possible cleanup and performance "tuning" will be investigated before the end of the project in January.
- As explained in chapter 7 we tried to change the memory behaviour of DVB subtitle decoding, but this caused a conflict with the software scaling filter (`vf_scale.c`). Still, the memory behaviour could probably be modified to use a constant memory block, and to only wipe clean the regions, which changed between page instances. One could try to add support for colored subtitles,

but that would require changing `spudec.c`, which uses a grayscale palette. Though, the more changes you make the more difficult it is to get your patch approved by the MPlayer developers.

- The audio compression filter in `af_newcomp.c` was done from scratch as the existing compression filter in MPlayer was found too complicated and apparently suffering from some bit rot. As the project's requirements for it were rather simple, so is its implementation – for example, it doesn't try to maintain the original volume range of the audio it receives (this could probably be changed in 5 lines of code or so), because the client was specifically interested in changing the absolute volume range from 0-1 to around 0-0.8 due his sound card producing distorted sound at peak levels. The input audio's range could also be restored by chaining MPlayer's normal audio volume filter after the compressor if necessary.

10.1 Known Bugs

The MPEG Transport Stream seeking code has a bug that can cause erratic behavior if the user rapidly triggers two seeks without allowing the player to play any video between them. This is caused by the seek code initially relying on a certain timestamp variable (there are multiple ones) in the demuxer's internal state that is only updated during normal video playback. If the seek attempts are in different directions and one is longer than the PTS reset detection threshold, this can also cause the PTS reset detection to trigger unexpectedly and seek well past the end of file, ending video playback.

Attempts to make the code update the variable after finishing a seek as well did not produce any results - the variable is touched by something else. Correcting the error would likely require changes elsewhere in the MPlayer code, possibly breaking other things the team is unaware of. Since the primary method of input for the Leffakone environment is a remote control, which can't easily produce input fast enough to trigger the bug, this was not considered a critical flaw by the client.

11 Testing

This chapter summarizes the testing process, and the results in general. Test round specific results are collected on separate testing reports. Methods of testing are presented in the Testing plan [1].

Test cases were used for getting a more systematic approach to the testing. This proved to be useful, since it brought up some of the issues that weren't noticed during the implementation phase. Running the test cases also supplemented the results of ad-hoc testing done during the implementation, and revealed if fixing a bug presented a new one.

Test cases were run four times with rc1 and once with the SVN version (the final rounds with the SVN version had not been run by the time of writing this document). The code for the new features is basically the same in both versions. So if bugs were found while testing with rc1, there was no point testing the code with the SVN version before fixing the bugs. A kind of back-to-back testing was used while solving some of the bugs to see if they are specific to a certain version of MPlayer. Beta testing was done with rc1 version only, since versions later than that do not compile on Leffakone.

Testing went mostly like planned, but it took more time than anticipated. This was mostly caused by the complexity of MPlayer, which made locating and fixing bugs quite difficult. Also fixing some bugs brought up new ones, which made the testing phase longer. Especially the MPEG TS seek feature was problematic in this manner. An overall result of testing was that many bugs were found, and almost all of them were fixed. Supposedly the only remaining issue is seek going wild when repeated rapidly (Described in Chapter 10). There was also a tricky subtitle delay problem, but luckily Matthieu found a solution for it. The summary of the testing is presented in Table 11.1.

Round	Date	Version	Errors	Comments
1	25.11.2008	rc1	2	First round with rc1. Problems with seek accuracy and subtitle timing.
2	26.11.2008	SVN	2	First round with SVN version. Had the same issues as rc1.
3	1.12.2008	rc1	2	Beta test by Matthieu Weber.
4	5.12.2008	rc1	3	Accuracy of MPEG TS seek was improved, but the correction brought up an issue with successive seeks. Subtitles were not tested, but the delay was still there.
5	13.1.2008	rc1	2	Final round with rc1. Subtitle delay fixed.

Table 11.1: Summary of the test rounds.

12 Bibliography

[1] Richard Domander, Tuomas Mäenpää, Teemu Nisu, Tommi Teistelä, "Testing plan of the Muksis software project", University of Jyväskylä, Department of Mathematical Information Technology, 2008.